SBRC 2019

Gramado | RS

XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

6 a 10 de maio

Gramado | RS

ANAIS DO WCGA 2019

sbrc2019.sbc.org.br

SBRC 2019

Gramado | RS

XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos

6 a 10 de maio

Gramado | RS

ANAIS DO WCGA 2019

sbrc2019.sbc.org.br

EDITORA
Sociedade Brasileira de Computação (SBC)

ORGANIZAÇÃO
Bruno Schulze (LNCC)
Luiz Bittencourt (UNICAMP)
Maríza Ferro (LNCC)
Alberto Egon Schaeffer Filho (UFRGS)
Weverton Luis da Costa Cordeiro (UFRGS)

REALIZAÇÃO
Sociedade Brasileira de Computação (SBC)
Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS)
Laboratório Nacional de Redes de Computadores (LARC)

Copyright ©2019 da Sociedade Brasileira de Computação
Todos os direitos reservados

Capa: Dórian Fogliatto
Produção Editorial: Carlos Raniery P. dos Santos (UFSM)

Cópias Adicionais:
Sociedade Brasileira de Computação (SBC)
Av. Bento Gonçalves, 9500- Setor 4 - Prédio 43.412 - Sala 219
Bairro Agronomia - CEP 91.509-900 - Porto Alegre - RS
Fone: (51) 3308-6835
E-mail: sbc@sbc.org.br

XVII Workshop em Clouds e Aplicações (12: 2019: Gramado, RS).

Anais / XVII Workshop em Clouds e Aplicações - WCGA; organizado por Alberto Egon Schaeffer Filho, Weverton Luis da Costa Cordeiro, Bruno Schulze, Luiz Bittencourt, Mariza Ferro - Porto Alegre: SBC, 2019

72 p. il. 21 cm.
Vários autores
Inclui bibliografias

1. Redes de Computadores. 2. Sistemas Distribuídos. I. Schaeffer-Filho, Alberto Egon II. Cordeiro, Weverton Luis da Costa III. Schulze, Bruno IV. Bittencourt, Luiz V. Ferro, Mariza VI. Título.

Sociedade Brasileira de Computação

Presidência
Lisandro Zambenedetti Granville (UFRGS), Presidente
Thais Vasconcelos Batista (UFRN),Vice-Presidente

Diretorias
Renata de Matos Galante (UFGRS), Diretora Administrativa
Carlos André Guimarães Ferraz (UFPE), Diretor de Finanças
Antônio Jorge Gomes Abelém (UFPA), Diretor de Eventos e Comissões Especiais
Renata Mendes de Araujo (UPM), Diretora de Educação
José Viterbo Filho (UFF), Diretor de Publicações
Claudia Lage Rebello da Motta (UFRJ), Diretora de Planejamento e Programas Especiais
Marcelo Duduchi Feitosa (CEETEPS), Diretor de Secretarias Regionais
Eliana Almeida (UFAL), Diretora de Divulgação e Marketing
Ricardo de Oliveira Anido (UNICAMP), Diretor de Relações Profissionais
Esther Colombini (UNICAMP), Diretora de Competições Científicas
Raimundo José de Araújo Macêdo (UFBA), Diretor de Cooperação com Sociedades Científicas
Cláudia Cappeli (UNIRIO), Diretora de Articulação com Empresas

Contato
Av. Bento Gonçalves, 9500
Setor 4 - Prédio 43.412 - Sala 219
Bairro Agronomia
91.509-900 – Porto Alegre RS
CNPJ: 29.532.264/0001-78
http://www.sbc.org.br

Laboratório Nacional de Redes de Computadores (LARC)

Diretor do Conselho Técnico-Científico
Paulo André da Silva Gonçalves (UFPE)

Vice-Diretora do Conselho Técnico-Científico
Rossana Maria de Castro Andrade (UFC)

Diretor Executivo
Ronaldo Alves Ferreira (UFMS)

Vice-Diretor Executivo
Danielo Gonçalves Gomes (UFC)

Membros Institucionais
SESU/MEC, INPE/MCT, UFRGS, UFMG, UFPE, UFCG (ex-UFPB Campus Campina Grande), UFRJ, USP, PUC-Rio, UNICAMP, LNCC, IME, UFSC, UTFPR, UFC, UFF, UFSCar, IFCE (CEFET-CE), UFRN, UFES, UFBA, UNIFACS, UECE, UFPR, UFPA, UFAM, UFABC, PUCPR, UFMS, UnB, PUC-RS, PUCMG, UNIRIO, UFS e UFU.

Contato
Universidade Federal de Mato Grosso do Sul (UFMS)
Faculdade de Computação
Caixa Postal 549
CEP 79.070-900 Campo Grande - MS - Brasil
http://www.larc.org.br

Organização do SBRC 2019

Coordenadores Gerais
Alberto Egon Schaeffer Filho (UFRGS)
Weverton Luis da Costa Cordeiro (UFRGS)

Coordenadores do Comitê de Programa
Antônio Jorge Gomes Abelém (UFPA)
Fabíola Gonçalves Pereira Greve (UFBA)

Coordenador de Palestras e Tutoriais
Ítalo Cunha (UFMG)

Coordenador de Painéis e Debates
Artur Ziviani (LNCC)

Coordenador de Minicursos
Miguel Elias Mitre Campista (UFRJ)

Coordenador de Workshops
Stênio Fernandes (UFPE)

Coordenador do Salão de Ferramentas
Leandro Villas (UNICAMP)

Coordenador do Concurso de Teses e Dissertações
Daniel Fernandes Macedo (UFMG)

Coordenadores do Hackaton
Raquel Lopes (UFMG)
Luis Carlos de Bona (UFPR)

Comitê de Organização Local
Avelino Zorzo (PUCRS)
Carlos Raniery Paula dos Santos (UFSM)
Cristiano Bonato Both (Unisinos)
Guilherme Rodrigues (IFSUL Charqueadas)
Jéferson Campos Nobre (UFRGS)
Juliano Wickboldt (UFRGS)
Marcelo Caggiani Luizelli (Unipampa)
Marcelo da Silva Conterato (PUCRS)
Rafael Pereira Esteves (IFRS)
Rodrigo Mansilha (Unipampa)
Tiago Ferreto (PUCRS)
Vinícius Guimarães (IFSUL Charquadas)

Comite Consultivo
Fabíola Gonçalves Pereira Greve (UFBA)

Paulo André da Silva Gonçalves (UFPE)
Fábio Luciano Verdi (UFSCAR)
Jó Ueyama (USP)
Antônio Jorge Gomes Abelém (UFPA)
Eduardo Cerqueira (UFPA)
Luiz Fernando Bittencourt (Unicamp)
Rossana Andrade (UFC)
Michele Nogueira (UFPR)
Edmundo Madeira (UNICAMP)

Mensagem dos Coordenadores Gerais

Mensagem do Coordenador de Workshops

Os Workshops do SBRC são uma importante oportunidade para aprofundamento no conhecimento de temas especializados ou emergentes na nossa comunidade científica. Como em anos anteriores, a coordenação manteve a *Chamada de Propostas de Workshops do SBRC*, que tem estimulado a comunidade brasileira de Redes e Sistemas Distribuídos a discutir a viabilidade de workshops com temas de pesquisa mais tradicionais, bem como estimular o debate para adoção de eventos que tratam de temas mais emergentes.

Tivemos uma diversidade salutar no Comitê de Avaliação em termos de localização geográfica e experiência, gerando, portanto, opiniões complementares sobre as propostas submetidas. Consideramos todas as propostas de altíssima qualidade, com temas, focos e escopo diversos. Nesta edição de 2019, buscamos equilibrar os benefícios trazidos aos participantes com a capacidade de alocação de espaços no evento. Desta forma, foram selecionados 10 Workshops de alta qualidade, além do tradicional Workshop da RNP (*WRNP*). Dentre as propostas aceitas, oito são reedições de workshops tradicionais do SBRC, a saber: **Gerência e Operação de Redes e Serviços** (*WGRS*), **Testes e Tolerância a Falhas** (*WTF*), **Pesquisa Experimental da Internet do Futuro** (*WPEIF*), **Segurança Cibernética em Dispositivos Conectados** (*WSCDC*), **Trabalhos de Iniciação Científica e Graduação** (*WTICG*), **Blockchain: Teoria, Tecnologias e Aplicações** (*WBlockchain*), **Clouds e Aplicações** (*WCGA*), e **Computação Urbana** (*CoUrb*). Como novidade, teremos dois novos workshops, a saber: o Workshop de **Teoria, Tecnologias e Aplicações de Slicing para Infraestruturas Softwarizadas** (*WSlice*) e o **Latin American Student Workshop on Data Communication Networks** (*LANCOMM Student Workshop*), que tem como público-alvo principal os alunos de pós-graduação e pesquisadores da América Latina, além de estreitar os relacionamentos da nossa comunidade com a Association for Computing Machinery (*ACM*).

Como coordenador dos Workshops do SBRC 2019, gostaria de agradecer a todos os envolvidos na seleção das propostas. Primeiramente aos coordenadores gerais do SBRC 2019, Weverton Cordeiro (UFRGS) e Alberto Egon Schaeffer Filho (UFRGS), pelo convite para a coordenação desta chamada de trabalhos, além de todo o apoio necessário para sua condução apropriada. Agradeço também a todos os membros do comitê de avaliação pelo imenso esforço nas revisões e discussões de alta qualidade de todas as propostas submetidas. Por fim, agradeço aos coordenadores dos workshops aceitos, pela dedicação no cumprimento dos prazos e na condução dos trabalhos internos de seleção dos artigos para seus respectivos eventos, mantendo a alta qualidade geral do SBRC. Os inscritos nos diversos workshops esperam ansiosamente pelas apresentações dos trabalhos e pelas frutíferas discussões que serão geradas.

Stênio Fernandes
Coordenador dos Workshops do SBRC 2019

Comitê de Programa

- Andrey Brito (Universidade Federal de Campina Grande)
- Antonio Tadeu Gomes (LNCC)
- Carlos Ferraz (UFPE)
- Cesar De Rose (PUCRS)
- Christian Esteve Rothenberg (University of Campinas - UNICAMP)
- Claudio Geyer (UFRGS)
- Daniel Batista (IME - USP)
- Daniel de Oliveira (Universidade Federal Fluminense)
- Edmundo Madeira (UNICAMP)
- Fabio Borges (National Laboratory for Scientific Computing - LNCC)
- Fabio Costa (UFG)
- Fabio Licht (Catholic University of Petrópolis (UCP)
- Fabricio Alves Barbosa da Silva (Fundação Oswaldo Cruz)
- Fernando Koch (University of Melbourne)
- Frederico Lopes (UFRN)
- Giacomo Mc Evoy (LNCC)
- Helio Guardia (Universidade Federal de São Carlos)
- José De Souza (UFC)
- Lisandro Zambenedetti Granville (UFRGS)
- Lourenco Pereira Jr (Instituto Tecnológico de Aeronáutica - ITA)
- Lucia Drummond (Universidade Federal Fluminense)
- Luis Carlos De Bona (University Federal of Parana)
- Marcelo Pasin (Université de Neuchâtel)
- Marco Netto (IBM Research)
- Marcos Assuncao (INRIA, ENS de Lyon)
- Nabor Mendonca (UNIFOR)
- Paulo Ferreira (University of Oslo / INESC-ID)
- Rafael Esteves (IFRS)
- Raphael Camargo (Universidade Federal do ABC)
- Rodrigo Righi (Universidade do Vale do Rio dos Sinos)
- Rossana Andrade (Universidade Federal do Ceará - UFC)
- Thais Vasconcelos Batista (UFRN)

Sumário

Sessão Técnica 1 1

RW-Through: A Data Replication Protocol Suitable for Geo Distributed and Read-Intensive Workloads 2
Wagner R. M. Barretto, Ana Cristina B. Kochem Vendramin, Mauro Sérigo Pereira Fonseca

Migração de Sistemas Monolíticos para Microserviço: Uma Abordagem Híbrida sem Downtime 16
Augusto Flávio Freire, Américo Sampaio, Otávio Medeiros, Nabor Mendonça

Sessão Técnica 2 30

Estratégias para Implementação de Coreografia de Microsserviços 31
João Pedro Monteiro Fernandes, Edson Tavares de Camargo

Impacto de LoRaWAN no Desempenho de Plataformas de IoT baseadas em Nuvem e Névoa Computacional 45
Ivan Zyrianoff, Alexandre Heideker, Dener Ottolini, João Kleinschmidt, Carlos Kamienski

Heap Allocator: Uma Polıtica de Alocação de Máquinas Virtuais em Ambiente de Computação em Nuvens baseada em Heap com Prioridade 59
Matheus Magalhães de Carvalho, Francisco Mardônio Vieira Filho, Filipe Fernandes S B de Matos, Rafael Lopes Gomes, Joaquim Celestino Júnior

XVII Workshop em Clouds e Aplicações
SBRC 2019
Sessão Técnica 1

Migração de Sistemas Monolíticos para Microserviço: Uma Abordagem Híbrida sem Downtime

Augusto Flávio Freire[1], Américo Sampaio[1], Otávio Medeiros[1], Nabor Mendonça[1]

[1]Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza
Av. Washington Soares, 1321, Edson Queiroz, CEP 60811-905, Fortaleza, CE, Brasil

augusto.arraes@ifce.edu.br, americo.sampaio@unifor.br, otaviomd@hotmail.com, nabor@unifor.br

Abstract. *Several organizations need to address the challenge to migrate current traditional monolithic applications in production to microservices, preferably, without having to schedule maintenances to take the application offline. This paper presents an approach for hybrid migrating to microservices with no downtime and practically no changes in the monolithic code. A real application is used as a proof of concept to demonstrate that the proposed approach enables to go "forward" or "backward" among different versions of the application without needing to take the application offline, considering code or data changes. An evaluation performed in the cloud demonstrates that this work does not introduce significant overhead.*

Resumo. *Várias organizações precisam solucionar o desafio de migrar as aplicações monolíticas tradicionais em produção para microserviço, de preferência, sem ter que causar manutenções programadas para retirar a aplicação do ar. Este artigo apresenta uma abordagem de migração híbrida para microserviços sem downtime e praticamente sem alteração de código do monolítico. Uma aplicação real é utilizada como prova de conceito para demonstrar que a abordagem proposta permite "ir" e "voltar" entre diferentes versões da aplicação sem a necessidade de retirar o sistema do ar, considerando modificações de código ou dados. Uma avaliação realizada na nuvem demonstra que este trabalho não introduz overhead significativo.*

1. Introdução

Microserviços vem se tornando uma das principais estratégias para construir aplicações distribuídas [Richardson 2017], [Newman 2015], [Balalaie et al. 2015a]. Com o paradigma de pequenas aplicações independentes, autônomas e escaláveis, os microserviços podem ser implantados e distribuídos de forma integrada e transparente ao usuário como se fosse uma única aplicação. Os maiores provedores de serviços do mundo, como Amazon, Netflix, Google e outros já utilizam o conceito de microserviços em suas aplicações.

Os microserviços vem inovando alguns conceitos da engenharia de software de como os sistemas estão sendo construídos, diferente dos tradicionais monolíticos. Diante dessa nova tecnologia de construir sistemas (independentes, escaláveis e integrados), há um grande desafio para o mercado e a comunidade acadêmica em migrar as aplicações monolíticas para microserviços. Comumente, as aplicações monolíticas já se encontram há um bom tempo em ambiente de produção sendo utilizadas por clientes,

já receberam várias atualizações e armazenam dados em bases de dados. Sendo assim, não é uma tarefa trivial migar estes monolíticos para microserviços, pois pode-se necessitar significativas alterações em código e nos dados armazenados necessitando de interrupções programadas para manutenção nos sistemas.

Outro fator complementar a se considerar numa possível migração do monolítico para microserviço é a escalabilidade da aplicação. Se um determinado módulo do monolítico é sobrecarregado, a aplicação inteira deverá ser replicada para atender a demanda das requisições pelo fato do monolítico possuir uma única unidade de implantação. Caso este determinado módulo fosse migrado para microserviço, apenas ele poderia ser replicado para escalar pois iria conter sua própria unidade independente de implantação. Esse é um dos fatores que motiva as organizações em migrar suas aplicações web tradicionais para microserviços podendo usufruir da escalabilidade independente e economia de recursos computacionais.

Migrar as aplicações monolíticas tradicionais para microserviço é um desafio de pesquisa em aberto. Decidir o que migrar, quando migrar e como migrar do monolítico para microserviço é uma tarefa complexa principalmente para monolíticos já em produção.

Este artigo apresenta uma abordagem híbrida de migração para microserviços sem *downtime* e praticamente sem alteração de código. Consideramos nossa abordagem híbrida pelo fato de que adotamos a estratégia de manter o monolítico em produção funcionando em conjunto com os microserviços que vão sendo desenvolvidos paulatinamente para substituir partes da funcionalidade dos módulos do monolítico. Para que possamos realizar as migrações sem *downtime* e praticamente sem alterações no código do monolítico adotamos conceitos de "*Reflection*" e Programação Orientada à Aspectos (AOP) que permite desviar chamadas à determinados módulos do monolítico (*Advice Around* da AOP) transformando-as em requisições REST (*Representational State Transfer)* ao código dos novos microserviços desenvolvidos.

Uma aplicação monolítica real é utilizada como prova de conceito para demonstrar que a abordagem proposta permite "ir" e "voltar" entre diferentes versões da aplicação sem a necessidade de retirar o sistema do ar, considerando modificações de código ou dados. Além disto, uma avaliação realizada na nuvem demonstra que a abordagem não introduz *overhead* significativo do ponto de vista de desempenho e nem de custos, e ainda permite suportar cargas maiores no monolítico devido à retirada de parte da funcionalidade do monolítico para ser processada em microserviço.

O presente trabalho está organizado da seguinte forma: A seção 2 descreve sobre o atual estado da arte de migração de aplicações monolíticas para microserviço. A seção 3 explica a abordagem de migração proposta neste trabalho, enquanto a seção 4 demonstra a avaliação de desempenho e de custo. A seção 5 apresenta a conclusão e os trabalhos futuros.

2. Migração de Aplicações Monolíticas para Microserviço

Eisele (2016) define três abordagens de migração para microserviço. A primeira, e menos arriscada, é migrar os serviços do monolítico que se adaptam melhor ao microserviço. É uma migração gradativa a cada serviço e os microserviços continuam associados aos respectivos serviços do monolítico. Em relação à proposta deste artigo, o microserviço se mantém de certa forma associado ao serviço do monolítico, pois no ato

da migração desacoplada essa propriedade é importante na chamada do microserviço correspondente. A outra abordagem de Eisele (2016) é semelhante a essa, mas o monolítico e o microserviço operam independentes. O monolitico não modifica os dados do microserviço, pois ocorre a migração da camada de dados e o microserviço mantém sua própria persistência, isolando a parte correspondente do monolítico. A terceira abordagem, a mais arriscada, sugere realizar a migração total para microserviços (*Big Bang Rewrite*). No entanto, é a abordagem menos recomendada para migração conforme afirma Eisele (2016), pois a disponibilidade dos serviços ainda poderá ser oferecida pelo monolítico. Já a proposta deste trabalho apresenta uma migração sem interrupções do monolítico. Sem precisar praticamente alterar código e sem realizar desligamentos programados ao sistema monolítico.

Outras abordagens de migração foram analisadas. Balalaie et al (2015b), Newman (2015), Richardson e Floyd (2016) tomam como estratégia de migração através da refatoração do monolítico identificando entidades de domínio e assim migrá-las para microserviços. Também Richarson (2014), Fowler e Lewis (2014) sugerem migrar para microserviços através de parâmetros convencionais da engenharia de software como tamanho do módulo, subdomínio da aplicação, ou relacionamento de funcionalidades e dados.

Uma abordagem semelhante ao trabalho deste artigo é do Richardson (2016b). O diferencial é que a abordagem deste artigo consegue voltar à versão anterior monolítica sem recompilar código, sem *downtime*, em tempo de execução. Outro trabalho semelhante que também adota técnica desacoplada de migração é do Bolar (2018). Esse utiliza *annotation* do Java, mas de toda forma altera o código do monolítico.

Medeiros (2018) realizou a migração de um serviço do monolítico para microserviço utilizando programação orientada a Aspectos, para interceptar a chamada do serviço no monolítico e encaminhar via REST para um microserviço correspondente atender a requisição solicitada. Essa abordagem é semelhante ao processo da "Ida" desta proposta de migração, ou seja, a realização da migração de fato do monolítico para o microserviço. Esta proposta de trabalho contribuiu com o caminho da "Volta", desfazendo a migração, caso isso seja decidido. Além de realizar a "Ida" e a "Volta" na camada de serviço (migração com Aspecto via REST), este trabalho também contribuiu com a "Ida" e com a "Volta" na camada de dados (microserviço com seu próprio banco de dados na "ida" e *recovery* dos dados do micro-banco ao banco monolítico na "volta"). Todo o processo ocorre praticamente sem alteração de código e sem *downtime*.

Diante da análise dessas abordagens de migração de aplicações monolíticas para microserviços, elaborou-se os principais requisitos (RM – Requisitos de Migração) necessários para avaliar uma abordagem de migração:

- [RM01] Capacidade de Refatoração: O quanto se pode refatorar um sistema monolítico em partes para ser migrado para microserviço. A refatoração poderia ser por módulo, por subdomínio do problema, por funcionalidades ou por serviço. Ou ainda, se é migrar parcialmente o monolítico [Balalaie et al. 2016], [Yanaga 2017], [Fowler e Lewis 2014] e [Newman 2015] ou refatorar por completo *(Big Bang rewrite)*.
- [RM02] Migração de Código: Depois de definir qual estratégia de refatoração será tomada, deve ser realizada a migração da parte do código do monolítico

para o código do microserviço. O microserviço deve atender e responder da mesma forma que o módulo correspondente no monolítico operava.

- [RM03] Alteração de Código: O quanto de código precisa ser alterado no sistema monolítico para realizar uma migração para microserviço, visto que segundo Eisele (2016) não é recomendado uma migração completa do monolítico para microserviço.
- [RM04] Associação entre Serviços: Que tipo de associação ou dependência existe durante e após a migração entre o que foi refatorado do monolítico com o microserviço correspondente [Eisele 2016].
- [RM05] Coexistência de ambos os paradigmas: Como seria a distribuição do monolítico e os microserviços, e também as respectivas bases de dados. Durante o processo de migração, ambos os serviços são balanceados para atender as requisições e qual dos serviços está responsável por atendê-las diante de quais estratégias a migração foi projetada.
- [RM06] Interrupção do Serviço: Na abordagem de migração, se deve interromper ou não o serviço do monolítico para realizar a migração para microserviço.
- [RM07] Migração de dados: Quando se trata sobre migração para microserviço, a primeira idéia é substituição de parte do código do monolitíco por código do microserviço. A migração de dados também deve ser um requisito, pois o microserviço encapsula ou está responsável por usa própria base de dados.
- [RM08] Caminho da "Volta": O fato de migrar para microserviço corresponde ao caminho da “Ida”. Mas se a migração apresentar problemas, ou se não corresponder como esperava, ou simplesmente se for uma migração temporária para atender uma certa demanda de requisições, de qualquer forma se deseja voltar para o monolítico também deve ser um requisito de migração para microserviço. Não há garantia de que todo o monolítico será migrado e também se o que foi migrado será definitivo. De toda forma, deve ser previsível a capacidade de voltar para o monolítico, pois este requisito pode ser bastante útil numa tomada de decisão necessária para desfazer a migração. E, se “Voltou”, após ajustes ou correções, pode realizar uma nova migração.

Esse último requisito não foi tratado nas abordagens de migração analisadas. Neste trabalho, além da migração sem *downtime* (desacoplada), foi proposto uma solução de “Voltar” para o monolítico, de forma transparente na operação da aplicação e sem interromper o serviço. Ao mesmo tempo, esse último também é um requisito proposto neste trabalho.

3. Abordagem Proposta

Dado um sistema monolítico que está sobrecarregado pela alta demanda de requisições por um determinado serviço, o qual necessita ser migrado para microserviço, há uma preocupação de como executar essa migração sem derrubar ou interromper a atividade do Monolítico. Essa questão é a motivação deste trabalho, cuja proposta é realizar uma migração parcial do Monolítico para o microserviço com zero *downtime*, com a capacidade de “ir” e “voltar”, ou seja, de poder ser migrado e também poder ser

retornado ao Monolítico (desfazer a migração), tudo em tempo de execução. Essa migração é de forma desacoplada, sem alteração de código do monolítico.

O trecho de código abaixo, Figura 1, mostra o Aspecto em Java que faz intercepção da chamada de serviço a ser migrado. Esse compara o serviço com os serviços do monolítico identificados pelo *Reflection* do Java, se o determinado serviço foi o escolhido para fazer a migração (seja um REST GET ou POST, por exemplo), a requisição HTTP (*HyperText Transfer Protocol)* é encaminhada ao microserviço. Essa técnica proporcionou a migração desacoplada e sem alterar o código original do monolítico.

```
1  @Around("bean(*Service) || bean(*Controller)")
2  public Object getIdMicroservice(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
3      Object microservice = null;
4      if( this.getIdMicroservice(proceedingJoinPoint.getSignature()) ) {
5          System.out.println(".. @MicroserviceGatewayAOP call Microservice via REST: "+ "http://"+this.URL_BASE+":"+this.URL_PORT+"/" + SERVICE_NAME );
6          // REST Method HTTP = GET
7          if( this.discoveryMethodHTTP(proceedingJoinPoint.getSignature().toString()).equals("GET") )
8              microservice = (Object) restTemplate.getForObject("http://"+this.URL_BASE+":"+this.URL_PORT+"/" + SERVICE_NAME, Object.class);
9          // REST Method HTTP = POST
10         if( this.discoveryMethodHTTP(proceedingJoinPoint.getSignature().toString()).equals("POST") ) {
11             HttpHeaders httpRequestHeaders = new HttpHeaders();
12             httpRequestHeaders.set("Content-Type", "application/json");
13             HttpEntity<Object> request = new HttpEntity<Object>(proceedingJoinPoint.getArgs()[0], httpRequestHeaders);
14             System.out.println(".. @OwnerAOP   restTemplate.postForObject");
15             microservice = (Object) restTemplate.postForObject("http://"+this.URL_BASE+":"+this.URL_PORT+"/" + SERVICE_NAME, request, Object.class);
16         }
17     }else { // continua no monolitico a chamada do serviço
18         System.out.println(".. @MicroserviceGatewayAOP prodeed, service call continues on Monolithic ");
19         microservice = (Object) proceedingJoinPoint.proceed();
20     }
21     return microservice;
22 }
23
24 private boolean getIdMicroservice(Signature methodSignature) {
25     String sig=methodSignature.toString();
26     System.out.println(".. @MicroserviceGatewayAOP methodSignature: "+sig);
27     boolean isMicroservice = false;
28     for (String keyvalueMonolithicServices:DynamicProperties.monolithicServiceMethods) {
29         if ( sig.substring(sig.lastIndexOf('.')+1,sig.lastIndexOf('(')).equals(
30                 keyvalueMonolithicServices.substring(keyvalueMonolithicServices.lastIndexOf('.')+1,keyvalueMonolithicServices.length()) ) ) {
31             isMicroservice = this.loadKeyValueMonolithicService(keyvalueMonolithicServices);
32             System.out.println(".. @MicroserviceGatewayAOP Target getIdMicroservice() : "+keyvalueMonolithicServices+" = "+isMicroservice);
33             // Faz VOLTA
34             mhc.goBackToMonolithic( keyvalueMonolithicServices, isMicroservice, "http://"+this.URL_BASE+":"+this.URL_PORT);
35             return isMicroservice;
36         }
37     }
38     return isMicroservice;
39 }
```

Figura 1. Migração para Microserviço desacoplada pelo Aspecto.

Para demonstrar a migração da abordagem proposta, um sistema Monolítico terceiro, desenvolvido em Java, utilizando o *Framework* Spring Boot [Macero 2017], foi escolhido: o PetClinic [Syer 2015]. O critério de escolha para o Monolítico era que estivesse escrito e organizado seguindo as boas práticas de desenvolvimento de sistemas em Java e de orientação a objetos. Outro fator também importante para a escolha, foi a disponibilidade do mesmo sistema em versão de Microserviço [Rey 2016], contemplando o *RM2*. Esse exemplo de aplicação, tanto em versão Monolítico quanto em versão Microserviço, foi adotado como ambiente de prova de conceito na migração híbrida (parcial e desacoplada) do Monolítico para um Microserviço, que correspondente ao um mesmo serviço do Monolítico. A migração do monolítico para o microserviço ocorre com zero *downtime e* sem alteração de código. E, caso a migração não seja mais necessária ou que já atendeu ao propósito, o serviço retorna ao Monolítico em tempo de execução (*RM08*), também sem causar *downtime* do mesmo.

Para esta abordagem, três fatores de migração são considerados quando se trata de migração híbrida de Monolítico para Microserviço, levando em questão "Ir" e "Voltar":

- Migração parcial: Algumas partes do monolítico são migradas para microserviço e não o monolítico completo (requisito *RM01*). Um ou mais serviços do monolítico são candidatos à migração, ou seja, há serviços que continuam sendo demandados para o monolítico e outros são migrados para microserviço (*RM05*).

Esses podem ser desativados e retornados ao Monolítico (*RM08,* caminho da "Volta") ou ainda podem ser migrados novamente em uma outra demanda.

- Migração Desacoplada: Não há necessidade de modificar o código do Monolítico (*RM03*) para realizar migração, seja na "Ida" ou na "Volta".
- Migração Zero *Downtime*: Durante todo o processo de migração, na "Ida" ou na "Volta", o monolítico não é desligado (*RM06*).
- Migração Micro-Banco: No conceito de Microserviço, cada um mantém a sua própria base de dados, diferente do Monolítico, no qual a base única é compartilhada para todos os serviços da aplicação. Então, atendendo essa premissa, ao migrar para microserviço, a entidade e os dados são extraídos (decompostos) da base monolítica e migrados para micro-base do microserviço correspondente. Assim as mudanças de estado de parte da aplicação serão mantidas pelo microserviço após a migração. Caso seja desfeita a migração, os dados serão atualizados na base monolítica (*RM07*).

3.1. Estratégia de Migração Híbrida de Sistema Monolítico para Microserviço

Numa migração híbrida de sistema monolítico para microserviço, para a abordagem proposta, deve-se levar em consideração os requisitos de migração e os pontos mencionados no item anterior: parcial, desacoplada, zero *downtime* e micro-banco. Tudo isso para tornar a migração híbrida de forma que possa "Ir" e "Voltar".

Na Figura 2, tem-se a modelagem da estratégia proposta. Nota-se que não há uma substituição definitiva de Monolítico para Microserviço (*RM01*). O contexto é de coexistência de ambos os modelos ainda como se fosse uma única aplicação (*RM05*). A implantação do Microserviço é de troca de requisição entre o Monolítico (*RM02*), de forma desacoplada, sem alterá-lo em código (*RM03*), e sem desligá-lo, com zero *downtime* (*RM06*), pois os demais serviços continuarão sendo atendidos pelo Monolítico (*RM05*).

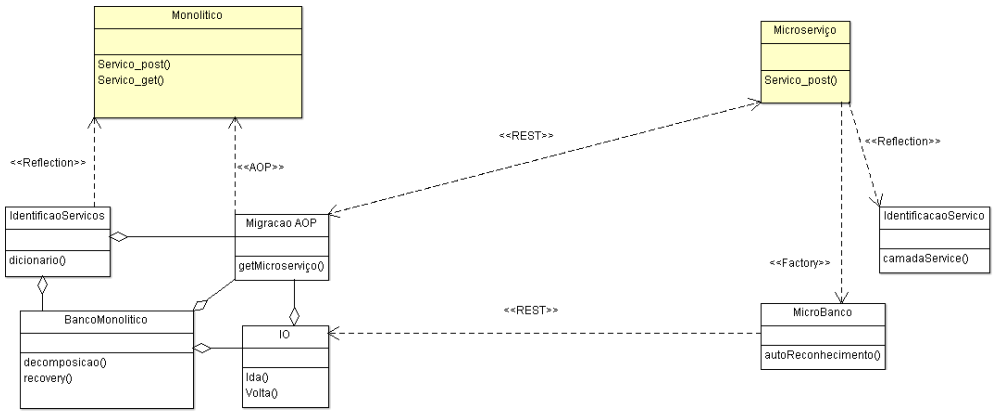

Figura 2. Estratégia proposta de Migração Híbrida.

De acordo com a estratégia proposta, a identificação de serviços é fundamental na camada de aplicação (*RM01*) e na camada de persistência. Na camada de aplicação, tem-se a visão do que será escolhido para ser migrado para Microserviço e do que permanecerá no Monolítico. O levantamento de informações mapeadas com as

dependências entre a camada de aplicação e de persistência estão disponibilizadas em todo o contexto para a migração do lado Monolítico.

Todos esses dados complementam a principal interface do padrão, a migração via AOP [Kiczales 1997]. É a interface de migração de fato entre o Monolítico e o Microserviço. De forma desacoplada, requisições HTTP ao Monolítico são interceptadas pelo Aspecto e encaminhadas via REST ao Microserviço (*RM02*), sem alteração de código do monolítico (*RM03*), o qual torna possível a não interrupção do monolítico para migrar a requisição (*RM06*). O serviço monolítico que será interceptado e como será executado (*RM04*) são garantidos pela identificação de serviços [Levcovitz et al. 2015] [Richardson 2016b]. Uma vez desacoplado, da mesma forma que foi migrado, pode ser desativado sem prejuízos à aplicação (*RM08*).

Durante a etapa de migração, como previsto no padrão, a criação do micro-banco (camada de persistência do Microserviço, *RM07*) é gerenciado do lado Monolítico. Essa criação do micro-banco é a decomposição da entidade (ou caso for mais de uma) e respectivos dados da base monolítica referente ao mapeamento do serviço-entidade. O micro-banco também é fatorado de forma desacoplada. Então, quaisquer alterações de dados, após a migração, na decisão de desfazer a migração (*RM08*), a "Volta", é garantida as diferenças de estado na base monolítica, de forma transparente e sem *downtime*. Uma mensagem via REST entre os paradigmas, Monolítico e Microserviço, através também do Aspecto, ao escutar que a migração foi desligada, é acionada para fazer o *recovery* dos dados do micro-banco para a base do monolítico.

Os elementos do padrão proposto foram elencados para garantir a "ida" (*RM01 a RM04*), de forma híbrida (*RM05*), e a "volta" (*RM08*) entre diferentes versões da aplicação sem a necessidade de retirar o sistema do ar, *RM06*.

3.4. Caso PetClinic: Cenário de Migração Híbrida do Sistema Monolítico para Microserviço

Uma API (*Application Programming Interface*) foi desenvolvida e implantada para sistema monolítico e outra distribuição voltada para microserviço também foi escrita. A API foi implantada no Monolítico PetClinic e também em um dos Microserviços do PetClinic. A API foi desenvolvida em Java baseada em Spring Boot.Tudo parte de uma escolha binária de uma lista de módulos do monolítico que possam ser migrados para microserviço. Essa lista é auto-identificada pela API. Uma vez escolhido o serviço, como Liga/Desliga, o serviço estará apto para ser migrado para microserviço. A migração ocorre sem interromper qualquer atividade do Monolítico. Utilizando programação orientada a Aspecto (AOP), a API intercepta a requisição da chamada da camada de serviço e de imediato encaminha ao referente microserviço via REST.

Enquanto estiver "*true*", a migração estará ocorrendo, em tempo de execução, a cada chamada nesse determinado serviço. Os demais serviços continuam sendo atendidos normalmente pelo Monolítico, a não ser que, algum outro também possa ser escolhido para ser migrado, bastando que: (1) o respectivo microserviço esteja desenvolvido e disponibilizá-lo, e (2) seja ativado, *true*", para ser também migrado. Essa decisão pode ser a qualquer momento, pois não é necessário programar interrupções no Monolítico para fazer a migração, sem *downtime*. A arquitetura da aplicação, durante essa migração parcial e desacoplada, torna-se híbrida, pois os

serviços da aplicação serão atendidos pelo monolítico e hora por um ou mais microserviços.

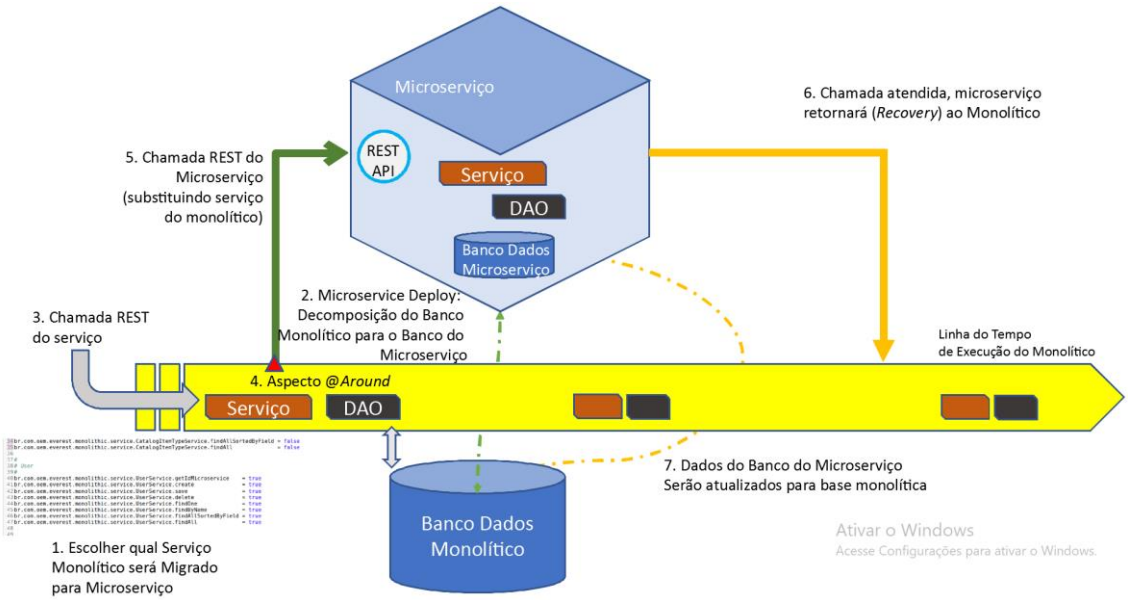

Figure 3. Ciclo de Migração Híbrida do Monolítico para Microserviço.

Todas essas operações ocorrem em tempo de execução, sem *downtime* e sem alteração de código. Conforme a Figura 3, observa-se que ao ser migrado o serviço monolítico para microserviço, também ocorre a migração da base de dados. Até o momento, verificou-se a migração na camada de aplicação. A API também realiza a migração na camada de dados, conforme previsto na estratégia.

No ato da migração e na primeira vez que o microserviço responder pelas chamadas do serviço via monolítico, a API decompõe o que o serviço utiliza de entidade na base de dados monolítica e cria em tempo de execução o micro-banco do microserviço. Neste momento, a mesma entidade da base monolítica não sofrerá mudanças de estado, pois isso será da responsabilidade do microserviço, já que o mesmo está atendendo as chamadas da migração via monolítico. Todas essas etapas de migração na camada de serviço e na camada de dados correspondem ao caminho da "Ida" da migração híbrida proposta neste trabalho.

Para o caminho da "Ida" no PetClinic, foi escolhido o serviço de cadastro de cliente e para microserviço também o mesmo serviço correspondente. Isso é uma simetria importante para migração entre o monolítico e o microserviço. Todas as requisições de cadastro solicitadas ao monolítico foram interceptadas via Aspecto e encaminhadas ao microserviço. Os dados já foram cadastrados no micro-banco desse microserviço, depois da decomposição da base monolítica da respectiva entidade do serviço. Um vídeo demonstrativo da API em funcionamento no PetClinic está disponibilizado [Freire 2019]. O vídeo demonstra tanto o caminho da "Ida" quanto o da "Volta".

Quanto à "volta", a decisão é se será "desligada" a migração. Essa decisão não afeta nenhuma interrupção da aplicação. Então ao ser desabilitada a migração, escolhendo "*false*", na próxima requisição ao serviço migrado ao monolítico, a API via Aspecto ainda a interceptará, mas não encaminha essa requisição ao microserviço.

Desse ponto em diante, o ambiente da aplicação, em tempo de execução, deixará de ser híbrido e ficará monolítico como antes, caso não haja mais nenhum módulo migrado a ser desligado.

Assim como na camada de serviço teve seu retorno para o monolítico, na "volta" na camada de dados também acompanha no mesmo retorno. Antes da requisição ser repassada de volta ao próprio monolítico, os dados do micro-banco serão refletidos para o banco do monolítico, caso tiverem sidos alterados. Quaisquer dados inseridos, alterados ou excluídos durante a migração serão refletidos no banco monolítico na "volta".

Na "Volta" no PetClinic, os dados que foram cadastrados na migração pelo microserviço foram refletidos na base monolítica. Isso acontece ao desligar a migração. Os dados da próxima requisição ao serviço do monolítico nesse momento já foram persistidos na base monolítica, logo após a atualização dos dados do micro-banco à base de dados monolítica. No mesmo vídeo, foi demonstrado também a "Volta" no PetClinic.

Concluída a volta, o monolítico mantém-se o mesmo devido à migração desacoplada, sem *downtime* e seu respectivo banco atualizado, de forma transparente ao usuário. A partir desse ponto, o monolítico poderá iniciar novo ciclo de migração híbrido.

4. Avaliação

Uma avaliação realizada na nuvem, Amazon AWS, demonstra que este trabalho não introduz *overhead* de desempenho e nem de custos significativos e ainda permite suportar cargas maiores no monolítico, devido à retirada de parte da sua funcionalidade para ser processada em microserviços.

Para a realização dos testes, foram escolhidos três serviços REST do Monolitíco PetClinic [Syer 2015], no qual 70% das requisições se concentram no serviço com método POST e os 30% restantes com as requisições dos outros serviços com método GET. O serviço com maior proporção de chamadas foi o candidato a microserviço. No entanto, os testes foram realizados em duas perspectivas de topologias: Homogênea (todas as requisições são realizadas ao Monolítico - forma tradicional) e Híbrida (todas as requisições chegam ao Monolítico, mas as requisições de um serviço POST representando 70% das requisições do teste são migradas para o microserviço, desacoplada via Aspecto; e, as demais requisições, GET dos outros dois serviços do teste, são atendidas pelo Monolítico).

Em ambas as topologias, um teste de carga foi aplicado durante 6 minutos, com 100 usuários-teste por 10 segundos submetendo, cada um, 10 requisições HTTP (via REST) sobre o Monolítico. Conforme proporção anterior, das 10 requisições, 7 são de cadastro de cliente (método POST) e 3 de visualizações (GET) de veterinários. Então, por cada minuto, uma carga de 60 mil requisições foi aplicada sobre a aplicação no monolítico (modo homogêneo) e depois, no modo híbrido, o monolítico recebendo todas as mesmas requisições, mas encaminhando ao microserviço as requisições POST (70%). Uma avaliação de desempenho e de custo foram realizadas nesse cenário.

4.1. Desempenho

Aplicado os testes, gerou-se métricas de uso de CPU, memória e tempo de resposta.

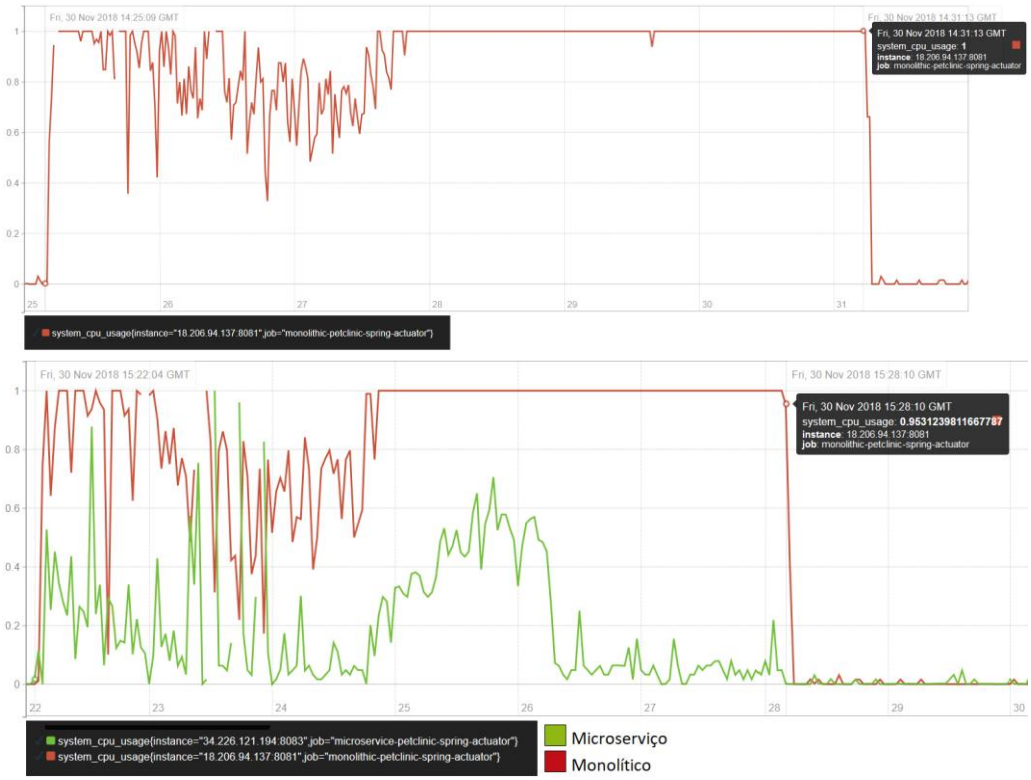

Figure 4. Consumo de CPU: Homogêneo e Híbrido.

No primeiro ciclo de testes, com 100 requisições, de acordo com a Figura 4, nota-se que na abordagem híbrida, com microserviço, que o esforço do monolítico chega a ser por volta de 20%, enquanto microserviço assume o maior esforço quando é responsável por atender o serviço requisitado. Enquanto, no primeiro gráfico, comparando na mesma ordem de tempo, o monolítico consumiu 60% (pouco antes do 3º. minuto). Outra observação é que num dado momento do teste de carga, nas duas perspectivas (tanto homogênea quanto híbrido), o consumo de CPU do monolítico chega ao máximo. No entanto, ainda assim, no híbrido, nota-se que o microserviço continua atendendo as requisições migradas do monolítico.

Figure 5. Consumo de memória: Homogêneo e Híbrido.

Tanto na perspectiva homogênea quanto na híbrida, a atividade de memória (Figura 5) manteve-se praticamente a mesma. Em ambas as situações, vale ressaltar que todas as requisições chegam ao monolítico. A diferença que na híbrida é possível visualizar a atividade de memória próximo a do homogêneo (quando apenas está o monolítico), pois o microserviço recebe 70% das requisições, conforme a configuração do teste.

Na topologia híbrida, mesmo que o microserviço consiga reduzir o processamento no monolítico por responder as requisições, ainda assim, todas as requisições continuam chegando primeiramente ao monolítico, antes de serem desviadas (interceptadas pelo Aspecto) ao microserviço via REST. Esse fato deve ser levado em consideração na leitura do tempo de resposta (Figura 6). Então, embora o tempo de resposta pareça ser o dobro da perspectiva homogênea (de 5k para 10K ms), isso já era esperado pelo fato de que as requisições têm um passo a mais para chegar no microserviço. Sendo assim, ainda é satisfatório pois praticamente manteve a mesma razão em relação ao homogêneo.

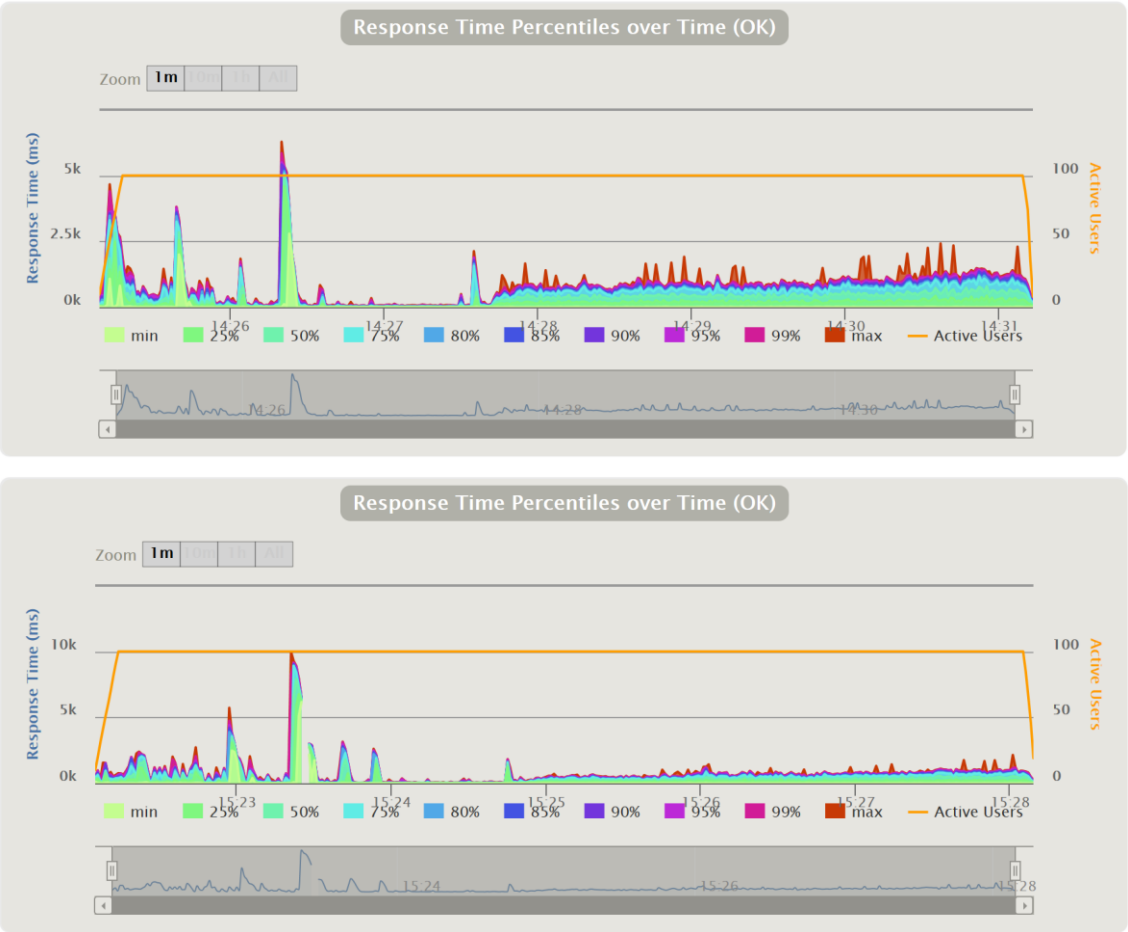

Figure 6. Tempo de resposta: Homogêneo e Híbrido.

4.2. Custo

A avaliação de custo da aplicação segue em duas perspectivas: na Homogênea, que tem apenas uma máquina na nuvem na Amazon, referente ao modelo Monolítico da aplicação; e, na Híbrida, que necessita uma máquina a mais para rodar o microserviço. Pode-se ter uma outra máquina para o serviço de base de dados (Ex: Amazon RDS - Amazon *Relational Database Service*), para ambas perspectivas.

Para a solução proposta, no uso da arquitetura Híbrida, o custo adicional por hora para a máquina do microserviço é de US$ 0,0116 (considerando-se uma máquina de 1 CPU e 1 Gb na Amazon - máquina t2.micro). Se o uso do microserviço for por algumas horas (por exemplo para atender a um determinado pico de demanda de requisições temporário), seu custo seria irrelevante. O uso por um mês completo o dia inteiro traria um custo adicional de aproximadamente 8,352 US$ por mês.

5. Conclusão e Trabalhos Futuros

Migrar todo o monolítico para microserviço ou migrar um módulo ou uma pequena parte do monolítico para microserviço definitivamente não são uma imposição. Certamente grandes organizações tomaram esse caminho. Porém, muitos sistemas tradicionais, nos quais já tem uma cultura legada em produção, podem testar, experimentar ou mesmo migrar temporariamente um de seus serviços para um ambiente

inovador, como o microserviço, apresentando neste trabalho. Um meio de migrar para microserviço sem precisar retirar do ar o monolítico, sem praticamente precisar alterar linhas de código do monolítico e sem ocasionar um *downtime* é o que foi a proposta da abordagem deste trabalho.

Uma aplicação real foi utilizada como prova de conceito para demonstrar que a abordagem proposta permitiu "ir" e "voltar" entre diferentes versões da aplicação sem a necessidade de retirar o sistema do ar tanto no que se refere à modificações de código quanto nos dados da aplicação. Uma avaliação realizada na nuvem demonstra que a abordagem não introduziu *overhead* significativo do ponto de vista de desempenho e nem de custos e ainda pode suportar cargas maiores no monolítico devido à retirada de parte da funcionalidade do monolítico para ser processada em microserviços.

Como trabalho futuro, aplicar um novo *script* de teste sobre dois monolíticos reais em produção (sendo um de empresa e outro de instituição pública) e avaliar o comportamento e desempenho de migração híbrida para microserviço. Também como contribuição futura, replicar a máquina do monolítico e comparar com a replicação das máquinas de microserviço em um número maior na nuvem, na perspectiva híbrida, para avaliar melhor a escalabilidade da aplicação.

Referências

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015a). Microservices migration patterns. Technical report, Department of Computer Engineering Sharif University of Technology.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015b). Migrating to cloud-native architectures using microservices: An experience report. In Advances in Service-Oriented and Cloud Computing -Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers, pages 201–215.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. IEEE Software, 33(3):42–52.

Bolar, T. (2018). Integrating microservices with a monolithic application. https://dzone.com/articles/integrating-micro-service-with-monolith-applicatio. Acessado: 18/12/2018.

Eisele, M. Modern Java EE Design Patterns: Building Scalable Architecture for Sustainable Enterprise Development, 2016. O'Reilly Media, Inc.

Fowler, M. and Lewis, J. "Microservices," 25 Mar. 2014; www.martinfowler.com/articles/microservices.html. Acessado: 18/12/2018.

Freire, A. Vídeo de Teste na Nuvem (AWS) da Abordagem de Migração Híbrida com PetClinic. 2019. Disponível em: https://www.dropbox.com/sh/5nzuypfg6xss33r/AAB-iE03-ueXmT2Mb_A1NIv2a?dl=0

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyv¨askyl¨a, Finland, June 9-13, 1997, Proceedings, pages 220–242.

Levcovitz, A., Terra, R., and Valente, M. T. (2015). Towards a technique for extracting microservices from monolithic enterprise systems. 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), p. 97-104, 2015.

Macero, M. Learn Microservices with Spring Boot, 2017. Apress, Inc.

Medeiros, O., Sampaio, A. and Arraes., A. Uso de AOP na Migração de Aplicações Monolíticas para Microservices. XVI Workshop em Clouds e Aplicações (WCGA), SBRC, 2018.

Newman, S. (2015). Building Microservices Designing Fine-Grained Systems, volume 2015. O'Reilly Media, Inc.

Rey, A. Distributed version of the Spring PetClinic Sample Application built with Spring Cloud. 2016. Disponível em: github.com/spring-petclinic/spring-petclinic-microservices.

Richardson, C. and Floyd, F. (2016). Microservices From Design to Deployment, volume 2016. NGINX, Inc.

Richardson, C. (2014). Pattern: Monolithic architecture. http://microservices.io/patterns/monolithic. Acessado: 18/12/2018.

Richardson, C. (2016b). Refactoring a monolith into microservices. https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/. Acessado: 18/12/2018.

Richardson, C. Microservices Patterns, Manning Publications, 2017.

Syer, D. Spring PetClinic Sample Application. 2015. Disponível em: github.com/spring-projects/spring-petclinic.

Yanaga. E. (2017). Migrating to Microservice Databases from Relational Monolith to Distributed Data, volume 1. O'Reilly

Estratégias para Implementação de Coreografia de Microsserviços

João Pedro Monteiro Fernandes, Edson Tavares de Camargo

[1] Universidade Tecnológica Federal do Paraná - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

ojoaofernandes@gmail.com, edson@utfpr.edu.br

Abstract. *The microservice architecture encourages the composition of services through choreography. Choreography favors low coupling and software decentralization. The big challenge is to find out suitable approaches to carry it out according to the architectural style proposed by the microservices. This paper describes and compares two strategies for choreographing microservices. The first is called event-based and makes use of a mediator to route the messages. The second is called choreographic programming, and its major advantage is to provide a global overview of choreography. A case study with four microservices was implemented using each strategy. Results include a comparative table and the number of messages per minute supported by each implementation.*

Resumo. *A arquitetura de microsserviços incentiva a composição de serviços através da coreografia. A coreografia favorece o baixo acoplamento e a descentralização. Surge então o desafio de encontrar abordagens adequadas para realizá-la de acordo com estilo arquitetural proposto pelos microsserviços. Este trabalho descreve e compara duas estratégias para coreografar microsserviços. A primeira estratégia é baseada em eventos e faz uso de um mediador para transmitir as mensagens. A segunda é chamada de programação coreográfica e sua grande vantagem é fornecer uma visão global da coreografia. Um estudo de caso com quatro microsserviços foi implementado em cada estratégia. Resultados incluem uma tabela comparativa e o número de mensagens por minuto suportado pelas implementações.*

1. Introdução

A arquitetura de microsserviços é um estilo arquitetural, inspirada na Arquitetura Orientada à Serviços (SOA), em que a aplicação é composta por uma suíte de serviços independentes e com responsabilidades bem definidas [Dragoni et al. 2017]. Comparados ao monólito, termo usado para definir uma aplicação que é executa em um único processo computacional, os microsserviços apresentam baixo acoplamento, garantem a interoperabilidade da aplicação e possibilitam a escalabilidade das funcionalidades que de fato requerem maior desempenho [Fowler and Lewis 2014]. A arquitetura vem chamando a atenção da comunidade de desenvolvedores e muitos são os trabalhos encontrados na literatura que contrapõem os microsserviços à arquitetura monolítica e mesmo à SOA [Cerny et al. 2018, Stubbs et al. 2015, Indrasiri and Siriwardena 2018].

Comparados à SOA, os microsserviços levam a uma maior autonomia dos serviços e promovem o desacoplamento uma vez que os serviços não precisam concordar em contratos em nível global [Cerny et al. 2018, Stubbs et al. 2015]. Dessa forma,

é possível implantar, corrigir, escalar individualmente cada serviço e distribuí-los em diferentes servidores localizados tanto em uma rede local quanto na Internet. Por outro lado, a característica distribuída da arquitetura de microsserviços requer que os serviços cooperem entre si para compor funcionalidades complexas e mais elaboradas da aplicação [Guidi et al. 2017]. Nesse contexto, há duas estratégias que se destacam: a orquestração e a coreografia. A estratégia de orquestração para a composição de microsserviços é obtida através da implementação de uma entidade central que atua como orquestrador. No entanto, essa estratégia implica no alto acoplamento da comunicação. Ao contrário do que tradicionalmente é feito em Serviços Web inspirados em SOA, microsserviços não contam com um componente responsável pela orquestração e por isso preferem a coreografia [Cerny et al. 2018].

Coreografia é um método de composição de serviços distribuídos em que cada entidade possui a responsabilidade de conhecer o momento e a forma de desempenhar sua função na composição [Bravetti and Zavattaro 2007]. O desafio em criar os microsserviços e coreografá-los é grande uma vez que os microsserviços ainda estão sendo conhecidos pela comunidade de desenvolvedores. A coreografia exige a comunicação distribuída entre unidades de software possivelmente localizadas em domínios distintos. Tal fato exige protocolos de comunicação leves para troca de mensagens, ferramentas para localização e a composição da regra de negócio usando unidades distribuídas.

O objetivo deste trabalho é apresentar e comparar duas estratégias para coreografia de microsserviços. A primeira estratégia é baseada em eventos e a segunda estratégia é chamada de programação coreográfica. Na estratégia de notificação de eventos, membros de uma coreografia atuam como produtores e/ou consumidores e a comunicação é mediada por sistemas de mensagens, como o Apache Kafka, ActiveMQ e RabbitMQ. Por outro lado, a programação coreográfica possui a vantagem de obter um sistema distribuído livre de *deadlocks*, comunicação ponto-a-ponto e a especificação de coreografia é representada pela formalização das interações esperadas entre os membros a partir da perspectiva global. Um sistema com quatro microsserviços foi proposto e implementado em cada estratégia. Resultados apresentados incluem uma tabela comparativa entre as estratégias a partir de nove característica elencadas. Também são apresentados o número de mensagens por minuto obtidos após executar cada implementação do estudo de caso proposto.

Este artigo segue organizado da seguinte forma. A Seção 2 apresenta a arquitetura de microsserviços. A Seção 3 explora a coreografia e apresenta duas estratégias para realizá-la. Um estudo de caso onde uma coreografia é implementada aplicando as estratégias descritas é apresentado na Seção 4. A Seção 5 apresenta os resultados obtidos. Por fim, a Seção 6 descreve a conclusão.

2. Arquitetura de Microsserviços

Monólito é um software em que suas abstrações menores não são independentemente executáveis e estão contidas no mesmo processo, fazendo assim com que a aplicação assuma uma arquitetura monolítica [Stubbs et al. 2015]. Por exemplo, considere uma arquitetura de software monolítica tradicional para o fornecimento de uma aplicação web. É importante destacar que embora haja a separação habitual em cliente, servidor de aplicação e

banco de dados, a implementação de todas as regras de negócio e execução das funcionalidades do sistema estão concentradas em um único processo no servidor de aplicação.

Uma aplicação de software cuja arquitetura seja monolítica possui benefícios durante os estágios iniciais de desenvolvimento. Além disso, a implantação da aplicação no servidor também é facilitada, pois será somente necessário realizar o *upload* do artefato executável e suas dependências para o ambiente de execução [Dragoni et al. 2017]. Por fim, no momento em que a aplicação não corresponder com o desempenho desejado, é possível escalar horizontalmente implantando outras instâncias da mesma aplicação em diferentes servidores e acessá-las através de um balanceador de carga [Stubbs et al. 2015]. Em contrapartida, há cenários em que esse estilo arquitetural dificulta o desenvolvimento. Dado que as regras de negócio são encapsuladas em um único artefato executável, possivelmente haverá um forte acoplamento o que acarreta negativamente a produtividade do programador, dificulta correções na aplicação e no gerenciamento de bibliotecas.

Uma possível solução é desacoplar as pequenas abstrações e torná-las independentes, como ocorre na arquitetura de microsserviços [Kozhirbayev and Sinnott 2017]. A arquitetura de microsserviços surgiu nos últimos anos como uma alternativa às abordagens tradicionais de desenvolvimento de software. Nesse estilo de arquitetura a aplicação é composta por uma suíte de serviços pequenos, independentemente executáveis, contidos em seus próprios processos e se comunicando através de mecanismos leves para troca de mensagens [Fowler and Lewis 2014]. Dentre suas principais características e benefícios, se destacam:

- Os microsserviços devem possuir limites e responsabilidades bem definidas, favorecendo o baixo acoplamento e alta coesão, seguindo o Princípio da Responsabilidade Única [Guidi et al. 2017];
- Independência e responsabilidades bem definidas elevam a produtividade da equipe, uma vez que resultam em um código fonte menor e de mais fácil entendimento, desenvolvimento e manutenção, tanto para desenvolvedores experientes quanto para novos membros da equipe [Indrasiri and Siriwardena 2018];
- Microsserviços favorecem a interoperabilidade, posto que a comunicação ocorre através de suas interfaces públicas e mecanismos leves. Dessa forma, diferentemente do monólito construído com uma única tecnologia, diferentes tecnologias podem ser empregadas para a construção de diferentes microsserviços [Dragoni et al. 2017];
- Como consequência da sua execução independente, microsserviços podem ser implantados e reiniciados sem impactar diretamente uns aos outros. Essa característica direciona os microsserviços ao uso de *containers* (conteinerização). Os *containers* conferem alta liberdade na configuração do ambiente e permite à aplicação se adequar ao que melhor se encaixa para a qualidade do serviço ou custo da infraestrutura [Kozhirbayev and Sinnott 2017];
- A escalabilidade de uma aplicação orientada a microsserviços se torna mais eficiente, dado que suas partes são independentes e podem ser escaladas conforme a necessidade aumenta [Fowler and Lewis 2014].

Embora a característica distribuída da arquitetura de microsserviços ofereça subsídios para solucionar muitos desafios impostos pelo monólito, ela apresenta algumas novas dificuldades. A execução em processos diferentes impede que os microsserviços

se comuniquem através de invocações de memória, logo, mecanismos para chamadas remotas, sejam elas locais ou através da rede, são necessárias [Dragoni et al. 2017]. Por exemplo, o protocolo HTTP (*HyperText Transfer Protocol*) é comumente empregado na comunicação síncrona entre os microsserviços, enquanto os Protocolos AMQP (*Advanced Message Queuing Protocol*) e MQTT (*Message Queuing Telemetry Transport*) são empregados na comunicação assíncrona [Indrasiri and Siriwardena 2018]. Por fim, vale lembrar que microsserviços podem ser considerados como uma evolução da arquitetura orientada a serviços representada pelos serviços web [Fowler and Lewis 2014]. Dessa forma, boas práticas e estratégias empregadas na arquitetura orientada a serviços podem ser reaproveitadas no cenário dos microsserviços.

Ao distribuir as regras de negócio através dos microsserviços na rede, a cooperação entre eles se torna fundamental para a execução de funcionalidades mais complexas e elaboradas da aplicação. A composição de microsserviços é obtida através da execução organizada seguindo um fluxo correto de ações a fim de completar uma requisição do cliente, seja ele humano ou outro sistema [Guidi et al. 2017]. Nesse contexto, há duas estratégias que se destacam: a orquestração e a coreografia. A estratégia de orquestração para a composição de microsserviços é obtida através da implementação de uma entidade central que atua como orquestrador, isto é, um microsserviço que conhece todo o fluxo necessário para alcançar determinada funcionalidade e seus participantes [Bravetti and Zavattaro 2007]. A orquestração, também comum na arquitetura orientada a serviços, é uma estratégia popular e largamente adotada, contudo não está alinhada com os princípios básicos da característica distribuída dos microsserviços. Escolher pela composição ao invés da orquestração resulta na concentração de responsabilidades em uma entidade central, que por sua vez aumenta a dependência de um único elemento e propicia o alto acoplamento da comunicação [Dragoni et al. 2017]. Por outro lado, a composição por coreografia não requer qualquer implementação central da responsabilidade do controle do fluxo de uma funcionalidade.

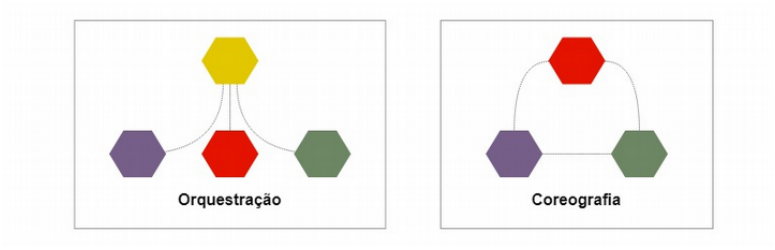

Figura 1. Diferença entre composição por orquestração e por coreografia [Gomes 2017].

A Figura 1 ilustra as duas abordagens para composição de microsserviços: orquestração e coreografia. É importante ressaltar a presença do serviço orquestrador na representação à esquerda da Figura 1, que exerce seu papel de através da comunicação com todos os outros serviços. Em contrapartida, a representação à direita ilustra a coreografia através da comunicação entre os próprios serviços. A seguir, descreve-se a coreografia de microsserviços.

3. Coreografia de Microsserviços

A coreografia é um modelo de composição de serviços em que o principal objetivo é proporcionar a colaboração entre membros independentes na execução de processos distribuídos. Nesse modelo, a colaboração ocorre através da troca de mensagens entre os próprios serviços. Suprime-se a necessidade de um orquestrador para coordenar o fluxo. Portanto, a coreografia é a descrição global das interações entre os membros de um sistema distribuído [Barker et al. 2009].

Durante o projeto de uma coreografia de serviços, três aspectos fundamentais precisam ser delineados: os processos distribuídos, os papéis e os protocolos de comunicação. O processo distribuído é um requisito do software que demandará cooperação entre dois ou mais participantes. Cada participante desempenha um papel, isto é, se comporta de tal modo a produzir e, ou, consumir mensagens para colaborar com a composição. Por fim, o protocolo de comunicação estabelece como as mensagens que asseguram a colaboração devem ser trocadas e interpretadas. Uma vez que cada papel foi executado corretamente e houve êxito na composição, afirma-se que houve uma encenação [Gomes 2017].

Embora a coreografia estabeleça uma descrição global das interações, não é necessário que cada participante tenha ciência de todos os fluxos distribuídos do software. Na verdade, é desejado que os participantes conheçam minimamente o escopo global, a fim de responder a estímulos (mensagens) conforme seu domínio e produzir insumos para a continuidade da composição. Dessa forma, diferentes implementações de um mesmo papel podem ser facilmente alternadas conforme necessário ou desejado, seja para buscar um resultado performático ou implementar novas regras de negócios.

Se por um lado a coreografia é benéfica para a colaboração entre serviços ao desacoplar a arquitetura e extrair o serviço orquestrador, por outro o modelo acrescenta obstáculos para a implementação e o gerenciamento do software. Como regras de negócios e controles de fluxo para consumo e produção de mensagens estão distribuídos entre os serviços, a complexidade geral do sistema aumenta drasticamente. Coreografias de microsserviços podem ser modeladas através de linguagens específicas. Essas linguagens são divididas em duas categorias independentes de implementação e específicas da implementação. A primeira modela coreografias na perspectiva de negócios e processos, sem necessariamente especificar detalhes de implementação. Nesse contexto é possível citar linguagens como Let's Dance e BPMN 2.0 [Gomes 2017]. A segunda linguagem modela coreografias com detalhes técnicos, como definições de protocolos e estrutura das mensagens. Estão nessas categorias linguagens como WS-CDL, WSFL e BPEL4Chor [Gomes 2017].

A linguagem gráfica Modelo e Notação de Processos de Negócios (Business Process Modeling Model, BPMN) oferece notação em alto nível para modelar a composição de papeis na coreografia. A linguagem possui 2 versões e a abordagem dos modelos produzidos pode ser diferente entre elas. A versão 1 oferece apenas recursos para especificação de coreografias através da interconexão de interfaces, ou seja, o modelo é restrito ao papel e não ilustra realmente como a colaboração acontece. Já a versão 2, mais recente, possui novas abstrações para permitir a especificação de coreografias através das próprias interações entre os papeis, o que garante a visão global da colaboração.

3.1. Coreografia Baseada em Eventos

Na estratégia de notificação de eventos, membros de uma coreografia atuam como produtores e/ou consumidores. Um consumidor se inscreve para receber determinados eventos. Sempre que um membro precisa se comunicar com outro, ele simplesmente publica ou produz um evento. Não é necessário conhecer a identidade dos possíveis consumidores [Zhang and Chen 2015]. Dessa forma, a comunicação de vários membros e a execução de diversas pequenas funcionalidades distribuídas podem resultar da notificação de um único evento.

A coreografia baseada em eventos diminui as interdependências entre os serviços, garantindo um baixo acoplamento. Do mesmo modo, os serviços podem ser implantados em diferentes servidores e contextos e se comunicar somente através da publicação e consumo de eventos. Além disso, o modelo baseado em eventos facilita a implementação de sistemas distribuídos em que os membros estão sujeitos a mudanças frequentes.

Tradicionalmente as abordagens do paradigma de notificação de eventos baseiam-se em comunicações mediadas [Ciancia et al. 2010]. Nessa abordagem, um mediador, ou um serviço terceiro, se torna responsável pelo roteamento da comunicação entre os membros. A mediação é empregada através da recepção de notificações, gerenciamento de eventos e sua distribuição aos respectivos consumidores interessados. É importante ressaltar que embora a abordagem induza a associação do mediador a um orquestrador, a sua responsabilidade se resume a notificar consumidores sobre eventos publicados por produtores. A Figura 2 ilustra a diferença entre o relacionamento dos membros de uma coreografia com comunicação mediada e o relacionamento par a par de uma coreografia com comunicação não mediada.

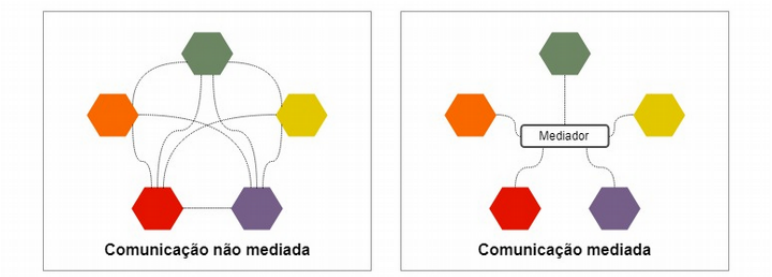

Figura 2. Diferença entre comunicação mediada e não mediada [D'Amore 2015].

Conforme apresenta a Figura 2, do lado esquerdo a comunicação não mediada conta com a comunicação direta entre os membros da coreografia. A ligação entre os membros se faz necessária sempre há a comunicação entre eles. Por outro lado, do lado direito há um mediador que conhece quais consumidores estão registrados para receber os eventos publicados pelos produtores.

Alguns mecanismos de comunicação mediada apresentam inteligência significativa. Um bom exemplo é o *Enterprise Service Bus*, que normalmente emprega implementações sofisticadas para roteamento, manipulação e filtro de mensagens e aplicação de regras de negócio. Embora soem como boa escolha, esses mecanismos ocultam detalhes de domínio e diminuem a coesão do microsserviço. Nesse sentido, a comunidade de coreografia de microsserviços recomenda uma abordagem alternativa: membros inteligentes e mediadores burros. Logo, ao projetar uma arquitetura de

comunicação mediada, todas as regras inerentes aos membros são contidas neles mesmos [Fowler and Lewis 2014].

Sistemas de mensagens despontam como uma excelente alternativa para mediar comunicações em coreografias baseadas no paradigma de notificação de eventos. Esses sistemas são serviços executados de forma independente do software e permitem a entrega e recepção de mensagens entre serviços distribuídos sem que esses se conheçam diretamente. Sistemas de mensagens são divididos em dois modelos [Garg 2013, Foundation 2018]: Filas e Publicação-Assinatura (*publish-subscribe*). No primeiro os eventos são recebidos pelo sistema e salvos em uma fila, para que então um conjunto de consumidores os consumam alternadamente. Essa característica permite a divisão do processamento de dados entre diversos consumidores sem duplicidade de leitura. No segundo, os eventos são recebidos pelo sistema e retransmitidos para todos os consumidores registrados para receber tais eventos. Essa característica permite que diversos membros executem seu papel na coreografia através de um único evento. Por outro lado, escalar o processamento deixa de ser trivial, pois todas os eventos vão para todos os assinantes. A plataforma Apache Kafka [Kreps et al. 2011] é uma solução para sistemas de mensagens que se enquadram no modelo publicação-assinatura e é empregada neste trabalho para realizar a coreografia baseada em eventos.

3.2. Programação Coreográfica

A coreografia pode ser comprometida simplesmente pela implementação incorreta de envio ou recebimento de eventos em um único membro. Entre os principais erros, os mais comuns são: *deadlock* e condição de corrida. O primeiro erro ocorre na espera indefinida por um recurso computacional e interfere no comportamento do sistema, levando-o a não responder. O segundo ocorre em cenários com acessos simultâneos a um recurso compartilhado e leva a computações com valores e retornos incorretos. Nesse contexto, afirma-se que a má comunicação entre os membros é uma falha de confiabilidade [Montesi 2013].

Linguagens de programação tradicionais não oferecem recursos para implementar a interação entre os membros de uma coreografia baseada na perspectiva global. Todas as comunicações modeladas são implementadas diretamente na perspectiva individual de cada microsserviço. Essa característica pode levar à construção de software não confiável. Por exemplo, um *deadlock* pode ocorrer quando um microsserviço produz determinado evento somente após o recebimento de um outro, no entanto nunca o recebe.

A Programação Coreográfica é um paradigma de programação recente e surge como estratégia de coreografia de microsserviços. Nesse paradigma, a especificação de coreografia é representada pela formalização das interações esperadas entre os membros a partir da perspectiva global [Montesi 2015]. Por exemplo, a formalização abaixo representa a interação entre dois papeis: Alice e Bob. Alice envia um livro ao Bob, que por sua vez retribui em dinheiro à Alice: $\text{Alice} \rightarrow \text{Bob} : \text{livro}; \text{Bob} \rightarrow \text{Alice} : \text{dinheiro}$.

A especificação de coreografia será sempre correta por *design*, pois descreve formalmente o fluxo de comunicação esperado em um software. E, por conseguinte, a coreografia pode ser compilada em implementações locais para cada microsserviço, provendo as ações de envio e recebimento. As implementações de microsserviços (*endpoints*) sempre corretas por construção. Como resultado, é obtido um sistema distribuído livre de *deadlocks* e as interações não são mais implementadas separadamente na perspectiva de

cada papel da coreografia. Após o processo de projeção, naturalmente, cada microsserviço pode ser editado e ter suas regras de negócio e de domínio implementadas. Atualmente há apenas uma linguagem de programação que implementa o novo paradigma, a linguagem Chor [Montesi 2013]. A linguagem foi proposta em conjunto com o paradigma pelo mesmo autor e ainda é classificada como um protótipo.

Na linguagem Chor, o recurso de especificação permite que coreografias sejam formalizadas e, principalmente, que as comunicações entre os membros respeitem protocolos previamente especificados. Caso existam inconsistências na especificação formalizada, erros são relatados utilizando realce de sintaxe. Há também o recurso de projeção que permite que, ao finalizar a especificação da coreografia, *endpoints* sejam projetados para a sequência das atividades de implementação.

Programas implementados pela linguagem Chor são compostos pelos seguintes elementos: identificação, preâmbulo e corpo. A identificação permite declarar um nome para a coreografia para aumentar a legibilidade do código. O preâmbulo, por sua vez, é a composição da definição do protocolo de comunicação e a criação do canal de comunicação. Por fim, o corpo é a especificação do comportamento de cada membro da coreografia. No Chor, todas as comunicações entre os membros da coreografia ocorrem através de sessões, assim como em serviços web tradicionais, e são modeladas como tipos básicos na linguagem. As sessões implementam protocolos, que especificam a ordem e o tipo as interações.

Apesar de ser uma linguagem, o Chor é disponibilizado atualmente apenas como um complemento para o ambiente de desenvolvimento integrado Eclipse. No entanto, a falta de uma versão *standalone* não impede a entrega de todos os recursos necessários para a programação de coreografias. Segundo os autores do projeto, há trabalhos em andamento para a disponibilização de uma versão própria em 2019 [Giallorenzo et al. 2018]. Chor ainda carece de mecanismos avançados para viabilizar a sua adoção na construção de um sistema distribuído de larga escala. Por exemplo, a linguagem ainda é limitada a tipos e estruturas de dados simples, como inteiros e *strings*, e não possui sistema de depuração integrado. Além disso, há somente suporte para a projeção de serviços com código fonte na linguagem de programação Jolie [Lanese et al. 2015].

Ainda que não seja uma alternativa completamente viável para a coreografia de grandes sistemas, a linguagem é promissora e está em plena evolução. O projeto possui código fonte aberto e é livre para uso. Além disso, a equipe acolhe bem novos colaboradores e possui um repositório no GitHub para facilitar a contribuição de terceiros.

4. Estudo de Caso

O estudo de caso proposto consiste de quatro microsserviços. A coreografia foi definida e modelada utilizando a linguagem de modelagem BPMN 2.0. Então, a partir do modelo gerado, a coreografia foi implementada em cada estratégia. A estratégia de notificação de eventos foi implementada através da Plataforma Apache Kafka. A programação no paradigma de Programação Coreográfica faz uso da linguagem Chor.

A Figura 3 apresenta quatro papeis que simulam a comunicação de acordo com a topologia de um anel. O microsserviço `Initiator` inicia a coreografia ao enviar uma mensagem `initiate` para o `Stream Processor A`, que ao recebê-la a envia para

o microsserviço `Stream Processor B`. Este, por sua vez, envia a mensagem para o `Returner`. O `Returner` então encerra a coreografia ao devolver a mensagem para o microsserviço `Initiator`. As subseções seguintes apresentam a implementação do modelo nas estratégias descritas.

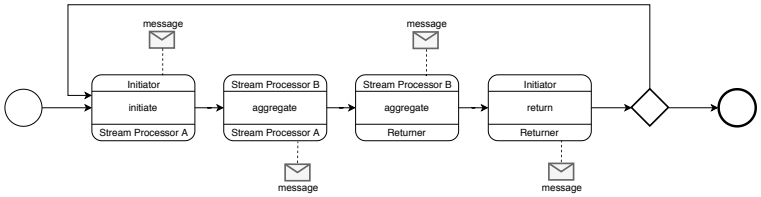

Figura 3. Coreografia para estudo de caso usando quatro microsserviços.

4.1. Implementação na Plataforma Kafka

Para a implementação dos membros da coreografia proposta foi utilizada a linguagem de programação JavaScript, seu ambiente de execução Node.js v10.15.1, o gerenciador de pacotes JavaScript NPM v6.8.0, a implementação kafka-node (pacote NPM) v4.0.2, uuid (NPM package) v3.3.2. Também, o Apache Kafka v2.1.0 gerenciado pelo Apache Zookeeper v3.4.13

O Apache Kafka é uma plataforma escalável de mensagens baseada no padrão de publicação-assinatura tendo como sua arquitetura principal um registro distribuído. A plataforma permite que os microsserviços sejam desenvolvidos em diversas linguagens de programação. Além disso, oferece recursos para a construção de sistemas distribuídos e em tempo real de forma confiável, tolerante a falhas e em alta performance. É possível também gerenciar serviços através de arquivos executáveis que são oferecidos junto com a distribuição padrão [Foundation 2018].

O Apache Kafka possui três conceitos chave: tópico, produtor e consumidor. Um tópico é um *stream* de mensagens de um tipo específico em ordem cronológica. Um produtor é uma aplicação, ou processo, que publica uma mensagem em um determinado tópico. Semelhantemente, um consumidor é uma aplicação, ou processo, que se inscreve em um ou mais tópicos e consome as mensagens lá publicadas.

Uma das desvantagens da arquitetura de microsserviços é a redundância de código fonte e essa característica pode ser observada no estudo de caso proposto. Dado que os papeis presentes na coreografia atuarão como consumidores e produtores, as implementações serão semelhantes para cada um. Para evitar esse problema uma única implementação para consumir e produzir mensagens na plataforma Apache Kafka foi especificada e compartilhada com cada serviço. Isso foi realizado através da implementação do `ServiceRunner` e do `BrokerCommons`. O primeiro implementa as funcionalidades para executar os serviços. E o segundo é uma coleção das funções utilizadas por todos os serviços.

Apesar do Algoritmo 1 apresentar apenas o código para o `Initiator`, código semelhante é empregado nos demais serviços apenas modificando as linhas 10, 15 e 16. O código inicia importando as bibliotecas descritas nas linhas 1 a 4. A linha 5 importa a implementação do comportamento do serviço. As linhas 7 e 8 instanciam o serviço

`Initiator` e o `ServiceRunner`. As linhas 9 a 17 contêm as configurações do `Initiator`. Por exemplo, as linhas 15 e 16 especificam o tópico que será publicado e o tópico que será consumido. As linhas 19 a 25 especificam o processo de execução do serviço pelo `ServiceRunner`. A linha 20 define as configurações escritas acima no escopo global do `ServiceRunner`. As linhas 21 a 24 adicionam os passos necessários para que o serviço se conecte ao Apache Kafka, verifique se os tópicos existem e instancia um consumidor e produtor. Por fim, a linha 25 executa os passos na ordem definida e executa o serviço.

```
1  import ServiceRunner from 'packages/service-runner';
2  import {
3   brokerConnection, consumerConnection, producerConnection, topicsSetup,
4  } from 'packages/broker-commons';
5  import Service from './service';
6
7  const service = new Service();
8  const runner = new ServiceRunner();
9  const config = {
10   group: 'initiators',
11   broker: {
12     host: '127.0.0.1',
13     port: '9092',
14   },
15   publishToTopics: ['queue.event-ia'],
16   consumeFromTopics: ['queue.event-ri'],
17 };
18
19 runner
20   .prepare(config)
21   .step(brokerConnection)
22   .step(topicsSetup)
23   .step(consumerConnection)
24   .step(producerConnection)
25   .run(service);
```

Listing 1. Código JavaScript para executar o microsserviço `Initiator`

4.2. Implementação na Linguagem Chor

A Chor versão Alpha e a linguagem Jolie v1.7.1 foram empregadas na coreografia do sistema distribuído descrito no estudo de caso. Importante observar que a linguagem Chor especifica a coreografia em um único arquivo, conforme apresenta o código do Algoritmo 2. Essa característica garante a perspectiva global das interações durante a implementação do sistema.

A definição do protocolo, que ocorre entre as linhas 3 e 8, formaliza quantos e quais serão os papeis na coreografia e como eles interagem entre si. Por exemplo, o `Initiator` se comunica com o `StreamProcessor1` através da função `initiate` ao enviar uma `string`. A construção da coreografia no corpo no algoritmo, entre as linhas 12 e 23, implementa de forma fixa o envio da mensagem a cada microsserviço e o seu retorno, também fixo. Essa construção deve, obrigatoriamente, implementar o protocolo definido no início. Esse código é então utilizado como parâmetro de entrada para o processo de projeção de *endpoint* e os respectivos microsserviços são gerados em Jolie.

```
1  program Serial;
2
3  protocol SerialProtocol {
4      Initiator        -> StreamProcessor1 : initiate  (string) ;
5      StreamProcessor1 -> StreamProcessor2 : aggregate (string) ;
6      StreamProcessor2 -> Returner         : aggregate (string) ;
7      Returner         -> Initiator        : return    (string)
8  }
9
10 public SerialChannel : SerialProtocol
11
12 main {
13     i[Initiator] start
14         sp1[StreamProcessor1] ,
15         sp2[StreamProcessor2] ,
16         r[Returner]
17         : SerialChannel (channel);
18
19     i   . "message" -> sp1 . payload : initiate  (channel) ;
20     sp1 . (payload) -> sp2 . payload : aggregate (channel) ;
21     sp2 . (payload) -> r   . payload : aggregate (channel) ;
22     r   . (payload) -> i   . payload : return    (channel)
23 }
```

Listing 2. Código Chor para especificar a coreografia

Ao gerar os *endpoints* as regras de negócio ou especificidades de domínio podem ser implementadas diretamente no código fonte do microsserviço. Por exemplo, a mensagem enviada pode ser lida em um arquivo presente no sistema de arquivos do sistema operacional ou consultada em um sistema gerenciador de banco de dados.

5. Resultados

Tendo como base o modelo gerado, a coreografia foi implementada por cada alternativa técnica de cada estratégia. A partir do estudo de caso, conclui-se que a mesma coreografia pode ser implementada e encenada perfeitamente em qualquer uma das duas estratégias introduzidas. No entanto, cada estratégia possui suas características e a sua seleção depende do cenário no qual ela será empregada. A Tabela 1 apresenta a comparação através de nove características observadas nas duas estratégias a partir das descrições apresentadas nas Seções 3.1, 3.2 e o estudo de caso implementado nas Seções 4.1 e 4.2. Além disso, também são apresentados o número de mensagens por minuto (vazão) obtidos a partir das implementações do estudo de caso. As implementações foram executadas em um máquina com sistema Linux Kernel 4.15.0-46-generic, distribuição Ubuntu 16.04.1 com 64 bits, processador Intel Core i5-3230M CPU @ 2.60GHz de quadro núcleos, memória DRAM Controller 7859MiB.

Analisando a comparação exposta na Tabela 1, considera-se que a estratégia baseada em eventos possui características importantes, como documentação completa, facilidade de manutenção, suporte a várias linguagens de programação e possui diversos estudos de casos. Essas características não se limitam a coreografia usando a ferramenta Apache Kafka. Também são encontradas essas características em outros sistemas de mensagens. As vantagens da programação coreográfica estão na perspectiva global e na construção por *design* que garante um sistema livre de *deadlocks*.

Característica	Baseada em eventos	Programação coreográfica
Mecanismo	Comunicação	Programação
Perspectiva	Individual	Global
Comunicação	Mediada	Ponto a ponto
Maturidade	Alta	Baixa
Facilidade de manutenção	Alta	Baixa
Documentação	Completa	Apenas a sua tese
Linguagens de programação	Mais de 15 implementações	Apenas Jolie
Estudo de casos	Diversos casos triviais e complexos	Quantidade limitada e trivial
Verificação por *design*	Não	Sim

Tabela 1. Comparação entre as estratégias de coreografia.

A principal vantagem na implementação de uma coreografia baseada em eventos é que uma interação não precisa ocorrer diretamente entre dois papéis. Basta que os microsserviços estejam conectados através do mesmo tópico, um produz e o outro consome. Essa característica permite escalar horizontalmente a aplicação apenas com a execução de um novo microsserviço interpretando o mesmo papel. Também, todas as tentativas de interação, eventos publicados, são registrados no mediador (Apache Kafka), assim não há perda de mensagem e é possível que o sistema continue uma ação distribuída mesmo após falha parcial ou integral dos microsserviços.

Em contrapartida, os microsserviços são implementados individualmente na sua própria perspectiva do todo. Logo, por não permitir visualizar globalmente os relacionamentos entre os membros da coreografia e dependendo da complexidade das interações, empregar modelagem, por exemplo através da BPMN 2.0, será essencial. Inevitavelmente os microsserviços serão iguais em diversos aspectos (por exemplo, a conexão com o *broker*, instanciação do produtor, consumidor e interação com os tópicos) e não há uma forma inteligente para projetá-los. Assim, será necessário implementar, gerenciar e distribuir bibliotecas para garantir a reusabilidade de código entre os microsserviços.

A implementação baseada na programação coreográfica destaca-se pela especificação em uma perspectiva global, ou seja, as comunicações são descritas sem quaisquer detalhes específicos de implementação de quem interpreta o papel. A implementação se concentra nas interações entre os membros. Em contrapartida, a especificação é utilizada como entrada para o processo de projeção de *endpoints* apenas uma vez. Ainda não é possível adotar o paradigma e sua linguagem, Chor, como ferramenta de trabalho numa metodologia de desenvolvimento de software incremental, por exemplo. Obrigatoriamente, cada alteração na especificação exige uma nova projeção de *endpoints*. A seguir são apresentados os resultados referentes à vazão para as implementações realizadas.

A Figura 4 exibe a vazão (eixo y) obtida após diversas execuções das implementações do estudo de caso. As execuções consideram diferentes tamanhos de mensagens (eixo x): 2^5 até 2^{12} bytes. A linha roxa exibe a vazão obtida na execução via Chor/Jolie e a linha azul a vazão via Kafka/Node.js. Mesmo ao aumentar o tamanho da mensagem, o estudo de caso desenvolvido via Chor apresenta uma taxa de transferência de 2,5x acima da vazão presente via Kafka/Node.js. Esse é um ponto altamente favorável à abordagem de programação coreográfica. Importante destacar que as duas abordagens utilizam protocolo binário próprio para comunicação entre os serviços.

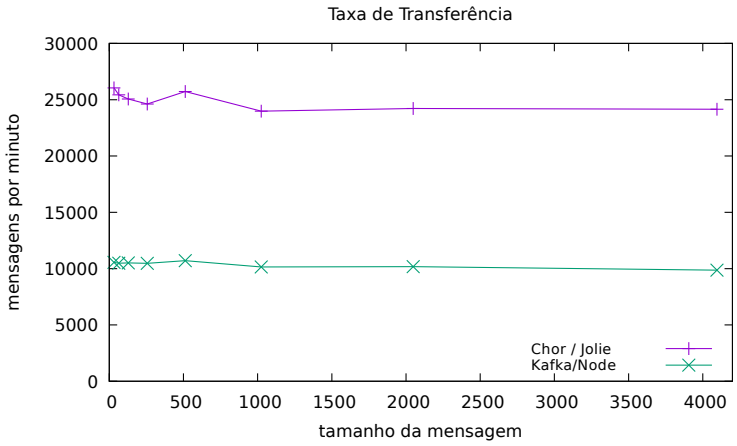

Figura 4. Taxa de transferência.

6. Conclusão

A arquitetura de microsserviços é de fato uma alternativa à arquitetura monolítica e garante diversos benefícios. Por exemplo, a execução independente de serviços, interoperabilidade e baixo acoplamento. No entanto, esse modelo arquitetural requer que a composição dos serviços distribuídos seja gerenciada de forma adequada. A coreografia de microsserviços é um dos modelos de composição de serviços e se destaca por ter valores semelhantes aos da arquitetura de microsserviços.

A coreografia de microsserviços pode ser aplicada através de duas estratégias. De um lado a estratégia baseada em eventos, que já é adotada em larga escada atualmente e possui diversos casos de uso para comprovar sua utilidade. Além disso, através da plataforma Apache Kafka é possível coreografar serviços em larga escala e manter um bom desempenho. Do outro lado, a estratégia baseada em programação coreográfica é um projeto pouco maduro. Ainda assim, possui valores intrinsecamente ligados à coreografia e desponta como uma alternativa promissora. Suas características garantem maior vazão quando comparada a estratégia baseada em eventos usando Apache Kafka.

Trabalhos futuros incluem a adoção do uso de *containers* como ambiente de execução dos microsserviços, distribuição em diferentes domínios, escalabilidade horizontal e tolerância a falhas. Outra direção para trabalhos futuros incluem a exploração da linguagem de microsserviços Jolie e, também, a contribuição para a evolução do paradigma de programação coreográfica.

Referências

Barker, A., Walton, C. D., and Robertson, D. (2009). Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166.

Bravetti, M. and Zavattaro, G. (2007). Towards a unifying theory for choreography conformance and contract compliance. In *Proceedings of the 6th International Conference on Software Composition*, SC'07, pages 34–50, Berlin, Heidelberg. Springer-Verlag.

Cerny, T., Donahoo, M. J., and Trnka, M. (2018). Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.*, 17(4):29–45.

Ciancia, V., Ferrari, G., Guanciale, R., and Strollo, D. (2010). Event based choreography. *Science of Computer Programming*, 75(10):848–878.

D'Amore, J. R. (2015). Scaling microservices with an event stream. Em 2019-03-20.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham.

Foundation, A. (2018). Apache kafka - a distributed streamin platform. Em 2019-03-20.

Fowler, M. and Lewis, J. (2014). Microservices. Em 2019-03-20.

Garg, N. (2013). *Apache Kafka*. Packt Publishing.

Giallorenzo, S., Montesi, F., and Gabbrielli, M. (2018). Applied choreographies. In *FORTE 2018*, pages 21–40. Springer.

Gomes, R. d. A. (2017). *Implantação eficiente de múltiplas coreografias de serviços em nuvens híbridas*. PhD thesis, Universidade Federal de Goiás, http://repositorio.bc.ufg.br/tede/handle/tede/7351.

Guidi, C., Lanese, I., Mazzara, M., and Montesi, F. (2017). Microservices: a language-based approach. *CoRR*, abs/1704.08073.

Indrasiri, K. and Siriwardena, P. (2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress.

Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182.

Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*.

Lanese, I., Montesi, F., and Zavattaro, G. (2015). The evolution of jolie - from orchestrations to adaptable choreographies. In Nicola, R. D. and Hennicker, R., editors, *Software, Services, and Systems*, volume 8950 of *Lecture Notes in Computer Science*, pages 506–521. Springer.

Montesi, F. (2013). *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen. urlhttp://www.fabriziomontesi.com/files/choreographic_programming.pdf.

Montesi, F. (2015). Kickstarting choreographic programming. In Hildebrandt, T. T., Ravara, A., van der Werf, J. M., and Weidlich, M., editors, *Web Services, Formal Methods, and Behavioral Types - 11th International Workshop, WS-FM 2014, Eindhoven, The Netherlands, September 11-12, 2014, and 12th International Workshop, WS-FM/BEAT 2015, Madrid, Spain, September 4-5, 2015, Revised Selected Papers*, volume 9421 of *Lecture Notes in Computer Science*, pages 3–10. Springer.

Stubbs, J., Moreira, W., and Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39.

Zhang, Y. and Chen, J.-l. (2015). Constructing scalable internet of things services based on their event-driven models. *Concurrency and Computation: Practice and Experience*, 27(17):4819–4851.

XVII Workshop em Clouds e Aplicações
SBRC 2019
Sessão Técnica 2

RW-Through: A Data Replication Protocol Suitable for Geo Distributed and Read-Intensive Workloads

Wagner R. M. Barretto, Ana Cristina B. Kochem Vendramin, Mauro Sérigo Pereira Fonseca

Informatics Department (DAINF) - Graduate Program in Applied Computing (PPGCA)
Federal University of Technology - Paraná (UTFPR)

wagner.rezende@gmail.com, {criskochem, maurofonseca}@utfpr.edu.br

Abstract. *In large-scale modern applications, users consume much more data than they create. To guarantee that data storage systems appropriately handle read-intensive workloads, application designers have been adopting data replication and caching techniques. These techniques face many challenges regarding data consistency, specially in faulty network conditions such as wide-area networks. In this work we use recent distributed consistency frameworks to identify the consistency models of the main solutions for read-intensive workloads used today in the industry. Additionally, we present RW-Through, a new data replication protocol suited for large-scale read-intensive workloads. Our new protocol provides stronger consistency guarantees than current solutions and is designed to tolerate unreliable network conditions, making it suitable for geo-distributed deployments.*

1. Introduction

Modern large scale applications manage high volumes of data. In these applications, users consume more data than they produce [Nishtala et al. 2013]. This behavior results in workloads dominated by reads and searches to the various storage systems that hold the application's data. Additionally, usage patterns of social networks indicates that the majority of the reads are concentrated on a small subset of the data [Bruner 2013].

Cloud providers and the industry have been adopting solutions such as database replication and caching in order to tackle read-intensive workloads scenarios. Database replication solutions for read intensive workloads generally consist of *Lazy Primary Copy Replication* (LPCR). LPCR is a common database replication technique [Gray et al. 1996, Pacitti et al. , Cecchet et al. 2008] where one node of a system is responsible for the write operations whereas a different set of nodes are responsible for the read operations. Caching solutions often consist of *Demman-filled read-aside caching* or, *Cache-aside*. Cache-aside is a common technique in web applications [Nishtala et al. 2013] where a read-optimized (often in-memory) data store is used to "cache" a subset of a system's data with a high chance of being accessed.

These solutions were born in single datacenter environments and face challenges when applied to geo-distributed cloud deployments. Specifically, both solutions are known to face distributed consistency problems. Such problems become more frequent when unreliable communication links are used.

Distributed consistency is an area that faces several advancements due to the emergence of large-scale distributed systems. Works in the area revolve around limitations inherent from distributed systems such as the CAP Theorem [Brewer 2000, Gilbert and Lynch 2002]. These limitations motivated the study of less restrictive consistency models such as eventual or causal consistency. These consistency models have relaxed requirements when compared to *Linearizability* [Herlihy and Wing 1990], the most restrictive of the models. Motivated by a large number of works introducing different consistency models, a work [Burckhardt 2014] introduced a mathematical framework capable of precisely defining the whole spectrum of consistency in distributed systems. This framework enabled the precise and formal identification of which consistency model a certain system is capable to provide.

In this paper we present two main contributions: we derive the general algorithms for LPCR and *Cache-aside* and formally analyze its consistency properties; and we present *RW-Through*, a new data replication protocol capable of addressing the same problems as the mentioned solutions while being more consistent, efficient in network usage and more suitable for geo-replicated deployments. The rest of the paper is organized as follows. Section 2 summarizes distributed consistency models and frameworks. Section 3 presents an analysis of the consistency model offered by LPCR and *Cache-aside*. Section 4 presents the *RW-Through* algorithms and their consistency analysis. Section 5 discusses how *RW-Through* compares to both LPCR and cache-aside. Section 6 presents future works. Finally, Section 7 presents concluding remarks.

2. Rationale and Background

This section briefly summarizes the main concepts behind consistency in distributed systems required in order to comprehend our analyses. The main focus is on the works done by [Burckhardt 2014] and [Viotti and Vukolić 2016]. It is important to note that the concept we are interested here is *Distributed Systems Consistency*, which is different from the concept of consistency commonly explored on the database literature. The concept of consistency in databases (i.e. the "C" in ACID) refers to the non violation of constraints the system's data may have. In distributed systems, consistency refers to how clients see a system's state when the system replicates such state among a group of different processes.

2.1. Distributed Consistency Specifications

We consider a distributed storage system as a finite set of processes connected through an asynchronous network. The system's purpose is to manage a collection of *Objects*. Clients are a separate set of processes that manipulate the objects by means of *Operations* submitted to the system's processes. A set of operations invoked during a given execution of the system is called a *History* (often referred as H). To represent notions of order or equivalence among the operations of a history, binary relations over the history are used. The notation $a \xrightarrow{\mathsf{rel}} b$ denotes that $(a, b) \in \mathsf{rel}$. The following relations are used in consistency models specifications [Burckhardt 2014].

- rb (*returns-before*): is an order relation based on real-time precedence. Denotes that an operation returned before other started. Formally: $\mathsf{rb} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.rtime < b.stime\}$ where stime and rtime are operations start and retur times respectively.

Ordering Guarantees	
EVENTUALVISIBILITY	$\forall a \in H, \forall [f] \in H/\approx_{ss}: \lvert\{b' \in [f] \lvert (a \xrightarrow{\mathsf{rb}} b) \wedge (a \not\xrightarrow{\mathsf{vis}} b)\}\rvert < \infty$
READMYWRITES	$\mathsf{so} \subseteq \mathsf{vis}$
MONOTONICREADS	$\forall a \in H, \forall b, c \in H\lvert_{rd} : a \xrightarrow{\mathsf{vis}} b \wedge b \xrightarrow{\mathsf{so}} c \Rightarrow a \xrightarrow{\mathsf{vis}} c$
CAUSALVISIBILITY	$\mathsf{hb} \subseteq \mathsf{vis}$
CAUSALARBITRATION	$\mathsf{hb} \subseteq \mathsf{ar}$
SINGLEORDER	$\exists H' \subseteq \{op \in H \mid op.oval = \nabla\} : \mathsf{vis} = \mathsf{ar} \setminus (H' \times H)$
REALTIME	$\mathsf{rb} \subseteq \mathsf{ar}$
Consistency Models	
BASICEVENTUALCONSISTENCY	EVENTUALVISIBILITY $\wedge$ NOCIRCULARCAUSALITY
CAUSALCONSISTENCY	CAUSALVISIBILITY $\wedge$ CAUSALARBITRATION
SEQUENTIALCONSISTENCY	READMYWRITES $\wedge$ SINGLEORDER
LINEARIZABILITY	SINGLEORDER $\wedge$ REALTIME

Table 1. Ordering guarantees and consistency models [Burckhardt 2014]

- ss (*same-session*): is an equivalence relation based on the invoking process. Two operations invoked by the same process are said to be in the same session. Formally: $\mathsf{ss} \stackrel{\text{def}}{=} \{(a, b) \mid a, b \in H \wedge a.proc = b.proc\}$ where proc is the client proccess that invoked the operation.
- so (*session-order*): is an order relation between operations of the same session. It can be expressed as the intersection of the two aforementioned relations. Formally: $\mathsf{so} \stackrel{\text{def}}{=} \mathsf{rb} \cap \mathsf{ss}$

Histories and their relations enable the representation of the observable behavior of a system. However they don't justify this behavior. Two other relations are used to explain why a history is the way it is. These relations can explain situations caused by non deterministic components of a system such as asynchronous networks and implementation-specific details. A history with these added relations is called an *Abstract Execution*. The relations are [Burckhardt 2014]:

- vis (visibility): an operation a is visible to an operation b ($a \xrightarrow{\mathsf{vis}} b$) if the effects of a are visible to the process that invokes b. For example, b reads a value written by a.
- ar (arbitration): denotes conflict resolution. Given two non-related operations a and b, $a \xrightarrow{\mathsf{ar}} b$ means that the system considers that a occurred before b.

An *Ordering Guarantee* is a logic predicate over an abstract execution. An abstract execution is said to be justified by a given ordering guarantee if the predicate is true for the execution. A *Consistency Model* is a collection of ordering guarantees. A system is said to provide a given consistency model if all of its histories can be justified by the ordering guarantees that compose the model.

With the notions of relations and ordering guarantees, the consistency models can be specified. Table 1 presents the formal definitions of the main ordering guarantees and consistency models. Next we describe the main characteristics for each model.

Informally, BASICEVENTUALCONSISTENCY can be defined as a guarantee that if additional operations stop being invoked on a given object, the state of the object on all

replicas will eventually be the same [Bailis and Ghodsi 2013]. This definition describes the *convergence* property of an eventually consistent system. Formally, convergence is expressed by the guarantee EVENTUALVISIBILITY. The guarantee states that a completed operation must eventually become visible to all other sessions. The guarantee EVENTUALVISIBILITY in conjunction with the guarantee NOCIRCULARCAUSALITY form BASICEVENTUALCONSISTENCY.

BASICEVENTUALCONSISTENCY offers very few guarantees and therefore allows a number of anomalies to be observed. The anomalies more easily observed by users are those that affect the sessions. To enforce that users can read objects that they wrote the guarantee READMYWRITES is defined. The guarantee can be formally defined by requiring that the session order be a subset of visibility. Another form of session anomaly is observed when users that read an object can not read it again in a future moment. The guarantee MONOTONICREADS prevents this.

Session guarantees do not rule out causality related anomalies. Causality in distributed systems is a notion that was explored by many past works [Lamport 1978, Schwarz and Mattern 1994, Ahamad et al. 1995]. Two operations are said to be causally related if the operations were issued in the same session or if one operation is visible by another. Using relations, causality can be expressed as the *happens-before* relation, which is the transitive union between session order and visibility. Formally: $\mathsf{hb} \stackrel{\text{def}}{=} \mathsf{so} \cup \mathsf{vis}$ [Burckhardt 2014]

The happens-before relation serve as a basis for two ordering guarantees that prevent cause-related anomalies: CAUSALVISIBILITY and CAUSALARBITRATION. The first one is when the system orders operations in a way that violates the happens-before relation. This can lead to situations where users see “answers” before “questions”. The second anomaly is when causally related operations are not visible to one another. This can lead to situations where users see chains of events with “holes” in them. The combination between the two causality guarantees form the consistency model CAUSALCONSISTENCY.

CAUSALCONSISTENCY fails to enforce that a single global order of operations exist within a system. The *Dekker Test* is a test designed to verify if a system enforces a single global order of operations. In the test, two processes write an object and right after read the value that the other process wrote. The pseudocode for the test is described in Table 2. If a system permits that both “A wins” and “B wins” get printed, the system does not pass the test and, consequently, does not guarantee a single order of operations.

To express the notion of a single global order of operations, the ordering guarantee SINGLEORDER is defined. A system that passes the Dekker test provide this guarantee. The guarantee SINGLEORDER combined with the guarantee READMYWRITES form SEQUENTIALCONSISTENCY. A system that provides this model, implicitly provides CAUSALCONSISTENCY [Burckhardt 2014].

SEQUENTIALCONSISTENCY still do not capture single copy semantics as it permits stale reads to occur. To capture single copy semantics, arbitration has to comply with real time. The ordering guarantee REALTIME defines that. Guarantees SINGLEORDER and REALTIME combined form the most restrictive consistency model, LINEARIZABILITY [Burckhardt 2014].

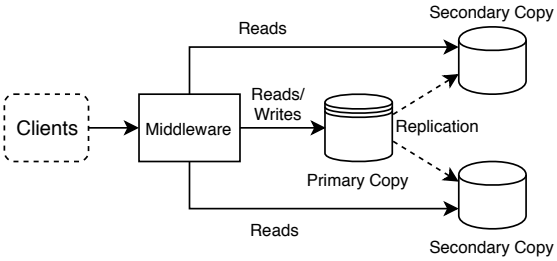

Figure 1. Lazy Primary Copy Replication

Program A	Program B

$x \leftarrow$ "active";
if $y = \varnothing$ **then**
  print "A wins";
end
	$y \leftarrow$ "active";
	if $x = \varnothing$ **then**
	  print "B wins";
	end

Table 2. Dekker test pseudocode [Burckhardt 2014]

With the implications among the consistency models, an objective "strength" order can be derived. It is said that a model A is "stronger" than B if every valid history of A is also valid over B. With this definition, the following hierarchy among the consistency models holds: LINEARIZABILITY $>$ SEQUENTIALCONSISTENCY $>$ CAUSALCONSISTENCY $>$ BASICEVENTUALCONSISTENCY [Burckhardt 2014]

3. Existing Solutions Consistency Analysis

In this section we use the distributed consistency definitions described previously to analyze the consistency models of LPCR and cache-aside.

3.1. Lazy Primary Copy Replication

Grey et. al. [Gray et al. 1996] categorizes database replication into four categories based on the combination of two properties: *Propagation* and *Ownership*. Propagation can be either *Lazy* (i.e. replication happens after the operation is returned to the client) or *Eager* (i.e. replication occurs before the operation is returned to the client). Ownership can be either *Primary Copy* (i.e. writes are centralized in one process) or *Update-Anywhere* (i.e. writes can happen in every process).

In this work we are interested in *Lazy Primary Copy Replication* since this is the replication strategy that suits better read-intensive workloads. A general architecture [Cecchet et al. 2008] of a system using LPCR is illustrated by Figure 1. The storage layer is composed by one *Primary Copy* (sometimes called *master*) and multiple *Secondary Copies* (sometimes called *slaves*). Clients submit operations to a *Middleware* layer responsible for sending write operations to the primary copy and read operations to the secondary copies. The primary copy is responsible for sending the write operations to the secondary copies. This process happens asynchronously, i.e. after the operations are returned to clients. Secondary copies hold a complete copy of the objects that the system manages, hence the strategy is said to be *full-replication*.

In LPCR, strong models can be immediately discarded. The reason is demonstrated by the execution of the Dekker test illustrated by Figure 2. I can be observed that a delay in the replication of the two writes from the primary copy to the secondary copy caused both reads to return $\varnothing$, which means a failure to provide the guarantee SINGLE-ORDER. If the execution is modified so that each program reads the value that it just wrote, the results would still be $\varnothing$. This shows that session order is not a subset of visibility ($\mathsf{so} \not\subset \mathsf{vis}$), which violates READMYWRITES. We can also infer that if $\mathsf{so} \not\subset \mathsf{vis}$,

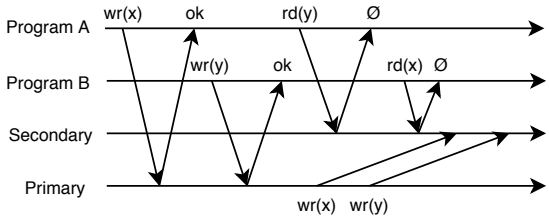

Figure 2. Possible Dekker test execution in an LPCR system

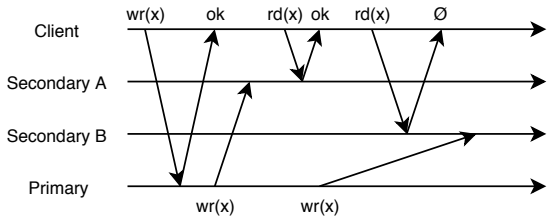

Figure 3. Execution with replication delay in an LPCR system

than $(\mathsf{so} \cup \mathsf{vis}) \not\subset \mathsf{vis}$, which violates CAUSALVISIBILITY. Since writes are centralized, the guarantee CAUSALARBITRATION is provided.

Figure 3 illustrates an execution that demonstrates that the guarantee MONOTONICREADS is not provided. The figure shows a client communicating with an LPCR system with two secondary copies A and B. After a write, two reads are routed to different secondary copies. A delay in the replication causes the second read to return an earlier version of the object. This situation violates the guarantee MONOTONICREADS.

The convergence property is achieved by LPCR since the primary copy needs to be always available for the system operation. Therefore replication to secondary copies never stops. Given the ordering guarantees provided by the solution, its consistency model can be defined as BASICEVENTUALCONSISTENCY plus the guarantee CAUSALARBITRATION.

3.2. Cache-aside

A common practice to increase performance of storage systems that handle read-intensive workloads is to use an in-memory data store alongside it. The in-memory data store acts as a cache to the system's objects and is managed by an application that uses the data. This is a popular technique in web applications where the web server is the main client of the storage system.

The solution is composed by two data stores, a "main" store that is used for general purpose and guarantees durability of the data (e.g. a relational database); and a "cache" store that is highly optimized for reads and often uses volatile memory to store the data (e.g. memcached). An application layer handles both stores and is responsible to fill the cache in an on-demand fashion. To handle possible out of space problems, an eviction mechanism is employed to free space as new objects are inserted into the store. When processing reads, the application first checks if the object exists in cache, if yes it is returned to the client, otherwise the application reads the object from the main storage, writes it in the cache, and returns it to the client. On writes the application writes the object in the main storage and cache sequentially.

As long as only one instance of application and cache exists and no other process writes to the main storage, no consistency anomalies occur. This happens because the application essentially serves as a single client to the storage system. This configuration is very limited and not suitable for geo-replicated scenarios. For the consistency analysis, we assume that the main storage can have multiple writers, e.g. multiple instances of applications and caches.

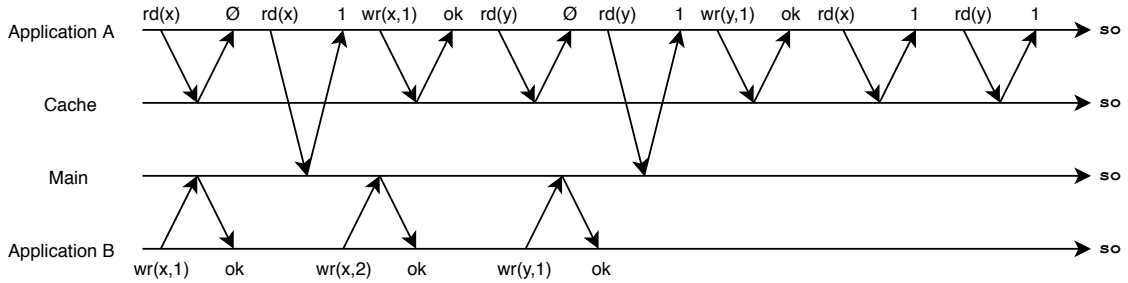

Figure 4. Cache-aside execution with two processes writing to the main storage

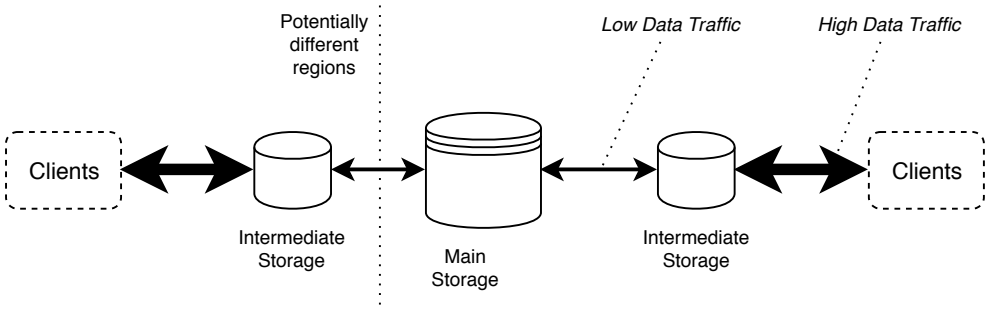

Figure 5. *RW-Through* general architecture

In configurations with multiple writers, strong models are not provided as illustrated by Figure 4. A second writer "Application B" issues three writes for two objects to the main storage. According to the happens-before relation, we have $wr(x,1) \xrightarrow{\text{hb}} wr(x,2) \xrightarrow{\text{hb}} wr(y,1)$. The process "Application A" keeps reading the key x with a stale value until an eviction occurs. This execution demonstrates that the guarantees SINGLE-ORDER and CAUSALVISIBILITY are not provided.

As the application accesses only a single cache and updates the cache after each write, both session guarantees are provided by the solution. As in LPCR, the centralized writes on cache-aside guarantee that CAUSALARBITRATION is provided. Convergence happens because the eviction mechanism prevents the cache from keeping the same objects forever. Given the ordering guarantees presented, we can conclude that the cache-aside solutions offers BASICEVENTUALCONSISTENCY plus the guarantees READMY-WRITES, MONOTONICREADS, and CAUSALARBITRATION.

4. RW-Through Specification

In this section we present a new algorithm for data replication suitable for read-intensive workloads and large scale applications. Similarly to LPCR and *Cache-aside*, our algorithm uses a centralized process for writes and a set of processes for reads. The main difference between our algorithm and LPCR and *Cache-aside* is the way clients interact with the system. In our algorithm, clients never communicate directly with the process responsible for writes. Instead all operations go to the set of processes responsible for reads. These processes act as an intermediate layer, processing some of the operations and making others pass *through* them as illustrated by Figure 5. For this reason we call the algorithm *Read Write Through* or simply *RW-Through*.

We present two variations of *RW-Through*: *full replication* and *cache*. The full replication version has similar properties to LPCR, where every process keeps a full copy of the data managed by the system. The cache version is similar to *Cache-aside* where the intermediate layer store a subset of the objects and employs an eviction mechanism. We refer to the two types of processes as *Main Storage* and *Intermediate*.

4.1. Full Replication Mode

The full replication mode of *RW-Through* are described by Algorithms 1 and 2. Inter-process communication is modeled as Remote Procedure Calls (RPC) exposed by the *Main Storage* process.

```
Data:
LS: local storage;
log: write log;
time: process current logical time;

RPC MAINGET(t):
    return log[t ↦];
end

RPC MAINPUT(key, val, t):
    LS[key] ← val;
    time++;
    log[time] ← (key, val, time);
    return log[t ↦];
end
```

Algorithm 1: Pseudocode of the Main Storage Process in Full Replication Mode

```
Data:
LS: local storage;
time: process current logical time;

Operation GET(key):
    return LS[key];
end

Operation PUT(key, val):
    log ← ORIGINPUT(key, val, time);
    APPLY(log);
    return;
end

Procedure APPLY(log):
    while log is not empty do
        (k, v, t) ← dequeue log;
        if t > time then
            LS[k] ← v;
            time ← t;
        end
    end
end
```

Algorithm 2: Pseudocode of the Intermediate Process in Full Replication Mode

The *Main Storage* process keeps three data structures in its internal state: LS, log, and $time$. LS is a map of keys and values and act as the main storage for the system objects. log is a record of all write operations performed on the system. Each operation is represented by an array of tuples (k, v, t), where k and v represent the key and value of object written and t the logical time associated with the operation. $time$ is the current logical time of the last write operation.

The *Main Storage* process exposes two RPCs: MAINGET and MAINPUT. MAINGET receives a logical time t as input and returns a subset of log containing the operations with logical $time > t$ (indicated with the notation $log[t \mapsto]$. MAINPUT receives as input key and val, which represent the object being written and t, which is the logical time of process that invoked the RPC. Upon receiving a MAINPUT request, the *Main Storage* process updates its local storage with the object received, increments its local variable $time$, appends the operation to its log, and returns a subset of log with the operations $time > t$.

The *Intermediate* process, similarly to MAIN STORAGE, keeps LS and $time$ in its local state. The process exposes two operations to clients: GET and PUT. GET receives as input a key and returns the value associated with this key in its LS. PUT receives as input a key and value pair and send it to the *Main Storage* process with its current logical time. With the write log resulted from the RPC call, the process invokes the auxiliary procedure APPLY. The procedure takes a write log as input and iterates over it similarly to a *First-In-First-Out* (FIFO) queue. For each dequeued object, the procedure checks if the object's logical time is higher than the process's current logical time; if it is, the key and value pair is applied in the local storage and process current time becomes the object's logical time. The process also executes a second auxiliary procedure GETLOG periodically. This procedure simply invokes the GETLOG RPC and applies the resulting log.

The algotithm has two main drawbacks: (1) high usage of space as the log grows on *Main Storage*. This can be solved by keeping track of current times for the *Intermediate* processes and implementing a cleanup routine that evitcs from the log every operation with logical time lower than the lowest of *Intermediate*'s logical time. (2) the high occurrence of stale reads when writes are not issued frequently on a particular *Intermediate*. This can be mitigated by periodically invoking MAINGET from *Intermediates*.

4.2. Cache Mode

RW-Through can also operates in cache mode supporting a partial replication scheme similar to *Cache-aside* where the *Intermediate* processes act as a demmand-filled cache. The general architecture is still the same, i.e. clients communicate only with the intermediate layer and contact the same *Intermediate* process instance for the duration of a session.

The algorithms for the *Main Storage* and *Intermediate* processes are described by Algorithms 3 and 4, respectively. The main idea of this mode is to keep on *Main Storage* state one write log for each *Intermediate* process instance. These write logs contain only operations on the objects stored by each *Intermediate*.

The cache mode of *RW-Through* is capable of using storage space and network more efficiently. Storage space efficiency is achieved by storing on the intermediate layer only the subset of data that has the highest chance of being required. Network usage efficiency is achieved by sending write logs with only the elements relevant to a particular *Intermediate* instance.

The *Main Storage* process needs to employ a mechanism to identify which objects are stored on each *Intermediate* instance. This can be achieved in a variety of ways such as hash tables or binary trees. In our implementation we choose *Cuckoo Filters* [Fan et al. 2014]. A cuckoo filter is a recent probabilistic data structure that enables set membership verification using space many times smaller than the set itself. The drawback is that the data structure allows a small number of false-positive results. The cuckoo filter is very similar to *Bloom Filters* [Bloom 1970]. The main difference between them is that cuckoo filter supports set member deletion while bloom filter does not.

We use the notation $\mathbb{I}$ to denote the set of *Intermediate* process instances. The *Main Storage* process keeps the following data structures in its internal state: A local storage LS; a set of *Cuckoo Filters* $F = \{f_i \mid i \in \mathbb{I}\}$ that maintains a compact registry of which key is currently stored in each *Intermediate* instance; a set of write logs $L = \{l_i \mid$

$i \in \mathbb{I}\}$ that represents the modifications to be executed in each *Intermediate* instance; a vector clock $T = \{t_i \in \mathbb{N} \mid i \in \mathbb{I}\}$ that keeps the highest logical time reported by each *Intermediate* instance.

```
Data:
LS: local storage;
L: log with all write operations performed;
T: set of logical times, one for each intermediate;

RPC MainGet(key, t, i):
    val ← LS[key];
    if val ≠ ∅ then
        add key to F_i;
        T_i++;
        enqueue (key, val, T_i) in L_i;
    return L_i;
end

RPC MainPut(key, val, t, i):
    add k to F_i;
    LS[key] ← val;
    foreach j ∈ I do
        if key exists in F_j then
            T_i++;
            enqueue (key, val, T_j) in L_j;
        end
    end
    CleanLog(L_i, t);
    return L_i;
end

RPC Evict(i, key):
    remove key from F_i;
end

Procedure CleanLog(log, t):
    while top of log has position ≤ t do
        dequeue from log;
    end
end
```

Algorithm 3: Pseudocode of the Main Storage Process in Cache Mode

```
Data:
LS: local storage;
i: process identifier;
time: process logical time;

Operation Get(key):
    if LS[key] = ∅ then
        log ← MainGet(key, time, i);
        Apply(log);
    end
    return LS[key];
end

Operation Put(key, val):
    log ← MainPut(key, val, time, i);
    Apply(log);
    return;
end

Procedure Apply(log):
    while exist elements in log do
        (k, v, t) ← dequeue from b;
        if t > time then
            e ← LS[k] ← v;
            time ← t;
            if e ≠ ∅ then
                Evict(k, i);
            end
        end
    end
end
```

Algorithm 4: Pseudocode of the Intermediate Process in Cache Mode

On cache mode, the *Main Storage* process follows a similar behavior to the full replication mode. The main difference lies in the *Main Storage* process returning the correspondent write log for each *Intermediate* instance. To keep track of which object is stored in each *Intermediate*, the *Main Storage* adds every key sent as input to MAINGET or MAINPUT RPCs to the *cuckoo filter* relative to the *Intermediate* instance that invoked the RPC. Whenever an object is evicted from an *Intermediate* local storage, an EVICT RPC is invoked. This RPC simply removes the object's key from the respective *cuckoo filter*. Before sending the write logs, they are submitted to a cleanup procedure CLEANLOG that removes every entry with logical time lower than the *Intermediate*'s current.

The *Intermediate* process in cache mode has two main differences from the full

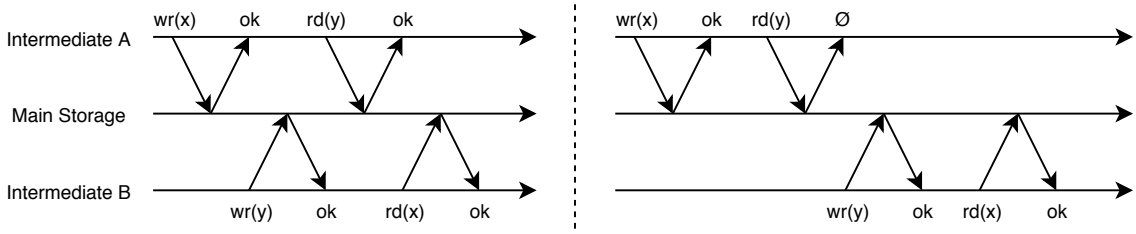

Figure 6. Dekker test executions on *RW-Through*

replication mode. The first is its GET operation. When a key is not found on local storage, the process submits a MAINGET RPC to fetch the requested object. The second is in the local storage that evicts items when its capacity is full.

4.3. Consistency Analysis

Since the full replication mode is a simplified version of cache mode, we only present an analysis for the cache mode. All properties observed in this mode are also valid on full replication mode. Initially, we proceed by defining the arbitration and visibility relations. Since writes are centralized in the *Origin* process, arbitration can be defined as the session order of that process: $\mathsf{ar} = \mathsf{eo}|_{orig}$. An *Intermediate* process instance has visibility to every operation that happened up until the process last communication with the *Main Storage* process. Formally, visibility can be defined as $a \xrightarrow{\mathsf{vis}} b \stackrel{\text{def}}{=} a \xrightarrow{\mathsf{ar}} \mathsf{lc}(b)$ where $\mathsf{lc}(op) : last_op$ (*last communication*) is a function that receives as input an operation op and returns the last operation that communicates with the *Main Storage* that precedes op.

To verify the ordering guarantees provided by *RW-Through*, we begin by analyzing the Dekker test execution. Figure 6 shows two executions of the test in a *RW-Through* system with two *Intermediate* processes, one for each test program. These two executions can be generalized to all possible executions of the test in an *RW-Through* system. For this, one only have to symmetrically invert the two programs. It can be seen that the solution passes the test, which means that *RW-Through* provides SINGLEORDER.

In a given session, writes will always be visible to subsequent operations in the same session. To demonstrate this, let a write operation a and an operation b that reads the object written by a such that $a \xrightarrow{\mathsf{so}} b$. After a completes, the *Intermediate* local storage will immediately contain the written object. The outcome of b can be either a cache-hit or a cache-miss. If the outcome is a cache-hit, the object is successfully retrieved, thus $a \xrightarrow{\mathsf{vis}} b$. If the outcome is a cache-miss, a communication with the *Main Storage* is required, thus $a \xrightarrow{\mathsf{ar}} b$. Given the visibility definition for *RW-Through*, $a \xrightarrow{\mathsf{ar}} b$ implies that $a \xrightarrow{\mathsf{vis}} b$. With these definitions we can conclude that the guarantee READMYWRITES is provided. This guarantee together with SINGLEORDER is enough to conclude that *RW-Through* provides SEQUENTIALCONSISTENCY.

5. RW-Through Compared to LPCR and Cache-aside

In this section we discuss how *RW-Through* compares to LPCR and cache-aside. Initially we analyze the ordering guarantees and consistency models provided by the three solutions. After, we discuss network usage and the applicability of the solutions in large-scale geo-replicated scenarios.

	LPCR	Cache-aside	RW-Through
EVENTUALVISIBILITY	✓	✓	✓
READMYWRITES		✓	✓
MONOTONICREADS		✓	✓
CAUSALVISIBILITY			✓
CAUSALARBITRATION	✓	✓	✓
SINGLEORDER			✓
REALTIME			

Table 3. Ordering guarantees provided by each solution

Table 3 summarizes the ordering guarantees provided by the three solutions. Based on Table 1, we can determine the consistency model for each solution. According to distributed consistency specifications, *RW-Through* offers SEQUENTIALCONSISTENCY which is stronger than the BASICEVENTUALCONSISTENCY offered by both LPCR and *Cache-aside*.

In practice, not offering session guarantees (i.e. READMYWRITES and MONOTONICREADS) can result in unintuitive situations to users. For example, a user writes a comment in a social network, refreshes the page, and can not see his comment. This is a possible situation on a system that does not provide READMYWRITES. Systems that do not provide causal guarantees may result in scenarios where answers are ordered before questions. These anomalies are possible in both LPCR and cache-aside. On the other hand, *RW-Through* provides all session and causal guarantees.

Due to its replication characteristics, *RW-Through* full replication mode can be an alternative to LPCR. In single datacenter scenarios where network partitions and delays are rare, LPCR consistency anomalies are also rare. *RW-Through* on the other hand, offers SEQUENTIALCONSISTENCY regardless of the network conditions, making it a viable option for geo-replicated deployments. LPCR can also be used as a fault tolerance solution for node crashes (e.g. a secondary can assume the primary role when the primary fails). *RW-Through* does not have this property. Therefore, we conclude that *RW-Through* is a solution for read-intensive workloads and unreliable network problems while LPCR is a solution for read-intensive workloads and unreliable nodes problems.

RW-Through cache mode can offer advantages when compared to cache-aside. Besides offering a stronger consistency model, *RW-Through* cache mode is more efficient in network usage and is less susceptible to stale-reads than cache-aside. Network efficiency is higher due to cache-misses being more efficient in *RW-Through*. While cache aside requires 6 messages to be exchanged in a cache-miss, *RW-Through* completes the operation with 4 messages exchanged. The reduction in stale reads occurs due to the write log mechanism employed by *RW-Through*. In cache aside an item is only refreshed in the cache when it expires. In *RW-Through*, every modified item in the main storage is refreshed on the *Intermediate* whenever a communication is made between *Intermediate* and *Main Storage* process. This reduces the window that an item already modified on the main storage remains active in the cache.

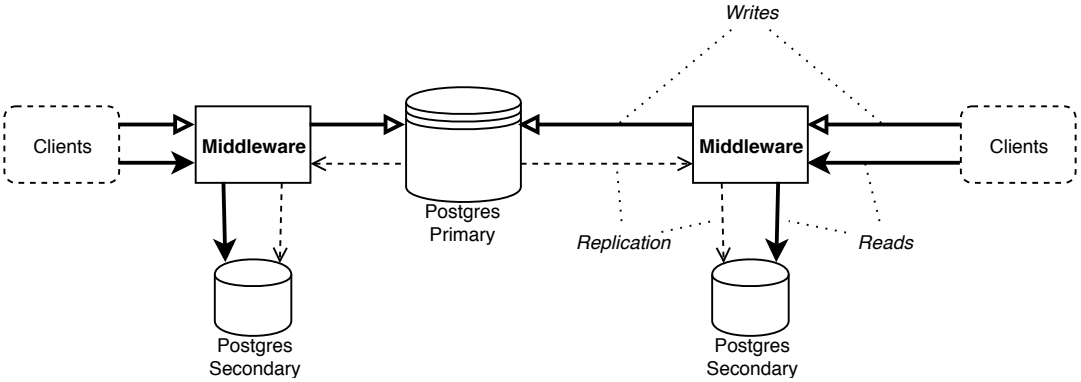

Figure 7. PostgreSQL middleware based solution based on *RW-Through*

6. Future Work

RW-Through can serve as a base for different types of projects involving data replication. In this paper we presented the algorithms and demonstrated its consistency and network usage properties. To validate its applicability in real world scenarios, we are currently developing projects that use *RW-Through*'s algorithms in different settings.

One such project is a middleware-based distributed PostgreSQL solution that implements *RW-Through*'s full replication mode. Figure 7 illustrates the general architecture implemented by the project. The main idea is to place a middleware between clients, a PostgreSQL instance configured as a primary copy, and a PostgreSQL instance configured as a secondary copy. One middleware exists for each secondary copy. All PostgreSQL instances are configured to use PostgreSQL's streaming replication. In this replication mode the primary sends write logs to secondaries in a way very similar to *RW-Through*. When a read operation is received, the middleware simply routes the query to the secondary. When writes are received, the middleware routes the query to the primary and waits for the write log containing the write operation. As soon as the log is applied to the secondary the operation is returned to the client. This scheme certainly adds an additional overhead to write operations when compared to other solutions. However, this solution is intended to be used when read throughput needs to be maximized and has the benefit of providing a higher consistency than LPCR.

7. Conclusion

In this work we conducted a formal consistency analysis on the main data storage solutions used by the industry for read-intensive workloads. We demonstrated that these solutions offer weak consistency models susceptible to many different consistency anomalies. The occurrence of such anomalies increases as the systems are exposed to faulty network conditions such as wide-area networks.

We presented *RW-Through*, a design for a data replication protocol capable of solving many of the problems faced by current solutions. *RW-Through* is capable of addressing the same issues as current solutions while offering a stronger consistency model. By using formal consistency verification frameworks, we demonstrated that our new protocol can maintain its consistency regardless of the network conditions. This makes *RW-Through* a suitable solution for large-scale geo-distributed deployments such as global

cloud applications.

In future works, we intend to use *RW-Through* as a basis for a PostgreSQL plugin. We aim to demonstrate that existing replication plugins can present consistency anomalies and compare the performance with our plugin on geo-distributed cloud deployments.

References

Ahamad, M., Neiger, G., Burns, J. E., Kohli, P., and Hutto, P. W. (1995). Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49.

Bailis, P. and Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Brewer, E. A. (2000). Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00.

Bruner, J. (2013). Tweets loud and quiet.

Burckhardt, S. (2014). *Principles of Eventual Consistency*, volume 1. now publishers.

Cecchet, E., Candea, G., and Ailamaki, A. (2008). Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08.

Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. (2014). Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59.

Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, New York, NY, USA. ACM.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. (2013). Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*.

Pacitti, E. p., Minet, P., and Simon, E. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. Technical report.

Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174.

Viotti, P. and Vukolić, M. (2016). Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34.

Impacto de LoRaWAN no Desempenho de Plataformas de IoT baseadas em Nuvem e Névoa Computacional

Ivan Zyrianoff, Alexandre Heideker, Dener Ottolini, João Kleinschmidt, Carlos Kamienski

Universidade Federal do ABC (UFABC)
{ivan.dimitry, alexandre.heideker, dener.silva, joao.kleinschmidt, cak}@ufabc.edu.br

Abstract. *LoRaWAN is a new technology that has been consolidating as a key data communication component to send data in IoT-based systems, due to its ability to send data over long distances with low energy costs. However, literature considers only wireless aspects, disregarding its computational aspects and its integration with IoT platforms, as well as ignoring the deployment possibilities that involve cloud and fog computing. In order to understand the computational impacts of the LoRa architecture we performed a careful performance evaluation study in a complex IoT scenario, exploring cloud and fog computing scenarios and integrating with the IoT FIWARE platform. The results show that the LoRaWAN architecture is scalable, but it has impacts on system performance.*

Resumo. *LoRaWAN é uma nova tecnologia de comunicação de dados que está se consolidando como uma das principais formas de se enviar dados em sistemas de Internet das Coisas (IoT), devido a sua capacidade de envio de dados a longas distâncias e baixo gasto energético. No entanto, trabalhos na literatura consideram apenas o desempenho da comunicação sem fio, desconsiderando seus aspectos computacionais e sua integração com plataformas de IoT, além de ignorar as possibilidades de instalação em nuvem e névoa computacionais. A fim de compreender os impactos computacionais da arquitetura LoRa realizamos uma avaliação de desempenho criteriosa em um cenário complexos de IoT explorando cenários de computação em nuvem e névoa e integrando com a plataforma FIWARE de IoT. Os resultados mostram que a arquitetura LoRaWAN é escalável, mas possui impactos no desempenho do sistema.*

1. Introdução

Atualmente há um crescimento exponencial de dispositivos conectados à Internet e espera-se que bilhões de sensores sejam conectados na próxima década [Ericsson, 2016]. Isso tem permitido o surgimento de uma nova geração de aplicativos e serviços inteligentes para o benefício da sociedade em diversos setores, como o urbano e o da agricultura. Um dos desafios é como obter os dados dos sensores, que possuem requisitos como: baixa taxa de transferência, longo alcance, baixo atraso e baixo consumo de energia; que não são totalmente atendidos pelas tecnologias de rede atuais.

Para atender essa nova demanda foram introduzidas as redes sem fio de baixa potência (LPWAN) como um conjunto de novas tecnologias adequadas para IoT [Raza

et al. 2017]. Dentre as tecnologias LPWAN, a tecnologia *Long Range* (LoRa) está se consolidando como uma das principais formas de se enviar dados em sistemas de Internet das Coisas (IoT), devido a sua capacidade de envio de dados a longas distâncias e baixo gasto energético. LoRa define a camada física que fornece o enlace de comunicação de rádio enquanto LoRaWAN estabelece o protocolo de comunicação das camadas superiores (como acesso ao meio e segurança) e a arquitetura do sistema.

Apesar de estar despontando como uma das principais tecnologias existentes, os trabalhos na literatura consideram apenas o desempenho da comunicação sem fio do LoRaWAN, desconsiderando seus aspectos computacionais além da sua integração com plataformas de IoT. Para implementar a tecnologia LoRaWAN, no mínimo dois módulos de software são necessários: o LoRa Network Server e o LoRa App (Application) Server, além de seus respectivos bancos de dados para persistência de informações. Esses elementos possuem potencial para tornarem-se gargalos e inviabilizarem o uso da tecnologia em determinados cenários. Outras implementações usam a modulação LoRa, mas não usam toda a arquitetura de referência LoRaWAN, que inclui segurança, configuração e gerenciamento.

Esse trabalho visa preencher essa lacuna, analisando o impacto computacional dos módulos de software necessários para se implementar uma arquitetura LoRaWAN e sua integração com a plataforma de IoT FIWARE[1]. Foram avaliadas diferentes configurações de uma comunicação entre sensores emulados e a nuvem computacional, envolvendo também o uso de névoa computacional. Para a emulação dos sensores se comunicando através de LoRaWAN foi necessário o desenvolvimento de um gerador capaz de implementar as opções de criptografia envolvendo a troca de chaves, acoplado à ferramenta SenSE [Zyrianoff, et. al. 2017], um gerador escalável de dados de sensores. A principal hipótese desse trabalho é que o uso da arquitetura LoRaWAN adiciona várias características importantes para o gerenciamento de redes IoT de larga escala e ao mesmo tempo adiciona uma sobrecarga computacional dentro de limites aceitáveis.

Os resultados obtidos esclarecem diversos pontos anteriormente nebulosos sobre a escalabilidade e o impacto computacional dos módulos de software da arquitetura LoRaWAN. A tecnologia é escalável e possui um bom desempenho. No entanto, há impactos no desempenho do sistema na presença desses módulos, em especial no tempo de processamento em cargas de trabalho baixas (aumento de aproximadamente 200 ms). Embora para a maioria das aplicações essa diferença no tempo de processamento não seja significativa, ela pode inviabilizar serviços que necessitem de processamento em tempo real com tempos de resposta muito baixos, como um serviço de semáforos inteligentes. No entanto, para a grande maioria das aplicações, vale a pena adicionar a arquitetura LoRaWAN à comunicação básica LoRa para obter as vantagens mencionadas de segurança, configuração e gerenciamento.

O restante do artigo está organizado como segue: na Seção 2 está presente o estado da arte, a seção 3 apresenta os trabalhos relacionados; na Seção 4 está descrita a nossa implementação da arquitetura LoRaWAN; a seção 5 trata da metodologia adotada no projeto enquanto os resultados são apresentados na seção 6 e discutidos na seção 7 e por fim são apresentadas as conclusões do trabalho.

---

[1] fiware.org

2. Estado da Arte

2.1. LPWAN e LoRaWAN

Nos últimos anos foram desenvolvidas várias tecnologias de comunicação sem fio para IoT conhecidas como LPWAN (*Low Power Wide Area Networs*) [Raza et al. 2017]. Estas redes possuem longo alcance de transmissão, baixo consumo de energia e baixa taxa de dados. Estas características tornam as redes LPWAN atrativas para aplicações de IoT que enviam poucas dezenas de bytes de dados a cada poucos minutos ou horas, como iluminação pública, monitoramento de poluição, *smart grids* e agricultura. Redes LPWAN possuem topologia estrela, em que vários sensores enviam dados diretamente a um concentrador, geralmente chamado de gateway, que possui conexão com a Internet. A tecnologia LoRa vem sendo bastante estudada na literatura e testada em aplicações práticas, já que a infraestrutura da rede pode ser instalada pelos próprios desenvolvedores das aplicações, ao contrário de outras tecnologias.

LoRa é um protocolo desenvolvido pela *Semtech Corporation* que opera nas faixas de frequência sub-GHz e permite atingir alcances de transmissão de até vários quilômetros com baixo consumo de energia. A camada física é chamada de LoRa e as camadas superiores formam o padrão LoRaWAN (padrão aberto desenvolvido pela LoRa Alliance) [Sornin et al. 2016].

2.3. Computação em Névoa

Computação em névoa (*fog computing*) é um paradigma relativamente novo, que visa lidar com desafios relacionados à enorme quantidade de dados que serão gerados com o aumento de dispositivos conectados à Internet [Bonomi, et. al. 2012]. A computação em névoa traz inteligência a dispositivos na borda da rede, solucionando sérios problemas como a diminuição da latência de sistemas de tempo real, diminuição do tráfego de dados entre a borda da rede e o núcleo, e diminui a carga de processamento na nuvem executando um processamento hierárquico dos dados.

2.4. Plataforma FIWARE

A plataforma FIWARE tem atraído a atenção de desenvolvedores de sistemas IoT no mundo inteiro por ser uma solução de código aberto financiada e incentivada pela Comissão Europeia. Ela é composta de uma série de componentes chamados de Generic Enablers (GE) que executam funções necessárias para uma variedade de aplicações, muitas voltadas para o uso de IoT em ambientes de sociedades inteligente.

O conceito do FIWARE implica em utilizá-lo para compor um sistema com uma arquitetura de micro-serviços, em que cada GE desempenha uma função e troca automaticamente mensagens estruturadas em um determinado padrão (OMA NGSI *standard*[2]) com outros GEs ou aplicações de terceiros por uma API REST bem definida. Dessa maneira, o FIWARE disponibiliza diversos desses blocos, os quais os desenvolvedores selecionam e os integram para criar o sistema que desempenhe as tarefas que necessitam. Entre os vários GEs disponíveis na plataforma FIWARE, alguns desempenham papel essencial para viabilizar aplicações inteligentes para IoT, como:

[2] openmobilealliance.org/release/NGSI

- Orion Context Broker[3]: Orion é um broker de informações de contexto baseado no padrão de comunicação *publish/subscribe*, sendo o principal GE da FIWARE. Ele oferece uma interface para clientes registrarem entidades de contexto e seus atributos, assim como produtores/consumidores dessas entidades. Na prática, o Orion é um grande distribuidor de dados que desacopla os transmissores dos receptores.
- IDAS e IoT Agent[4]: IDAS é uma implementação do GE chamado *Device Backend Management* que contém vários componentes IoT Agent que mapeiam dados vindos de dispositivos IoT no modelo de informação FIWARE NGSI. Componentes IoT Agent são conectados ao Orion para o qual enviam e do qual recebem dados.

O uso do FIWARE envolve a instalação de seus GEs numa infraestrutura apropriada para executá-los, que podem ser máquinas isoladas, nuvens públicas ou privadas usando diferentes controladores, como OpenStack[5] ou OpenNebula.

3. Trabalhos Relacionados

Diversos trabalhos recentes fizeram testes experimentais e analisaram o desempenho de enlaces LoRa em diferentes cenários, como ambientes internos e urbanos [Ayele et al. 2017] [Magrin et al. 2017] [Georgiou & Raza 2017] [Petäjäjärvi et al. 2017] [Van den Abeele et al. 2017]. Estes trabalham analisaram a influência dos parâmetros de camada física na taxa de entrega de pacotes, atraso e consumo de energia. Também há vários trabalhos na literatura que analisam o desempenho de redes LoRaWAN para verificar a escalabilidade e desempenho da rede com vários dispositivos na rede (nós sensores e gateways). Devido à dificuldade de implementar redes experimentais com dezenas ou centenas de dispositivos, estes trabalhos usam modelos analíticos e simuladores para a análise. Em geral, os trabalhos na literatura consideram apenas o desempenho da comunicação sem fio do LoRaWAN (dispositivos até o gateway), no entanto, a arquitetura LoRaWAN possui outros componentes, como o servidor de rede LoRa (*LoRa Network Server)* e aplicação LoRa (*LoRa App Server*), que não têm sido objeto de estudo na literatura em termos de desempenho e escalabilidade (podem agregar dados de dezenas de gateways, por exemplo).

Escalabilidade é uma característica fundamental para plataformas de IoT, já que visam gerenciar milhares de dispositivos enviando dados continuamente e o desempenho do FIWARE vem atraindo atenção da sua comunidade de usuários. Um estudo abrangente que propõe métricas qualitativas, quantitativas e avalia o desempenho de várias plataformas de IoT é apresentado por da Cruz et al. (2018). No entanto, eles adotaram uma abordagem genérica, sem considerar a integração com LoRaWAN e apenas uma única configuração de plataforma foi considerada.

Martínez et al. (2016) fornece uma descrição detalhada da arquitetura de um *testbed* da plataforma FIWARE configurada para o cenário de agricultura de precisão, no entanto, todos os aspectos de integração com sensores foram abstraídos, já que os

[3] fiware-orion.readthedocs.io
[4] catalogue-server.fiware.org/enablers/backend-device-management-idas
[5] openstack.org

dados eram enviados diretamente para o FIWARE utilizando usando a API REST do Orion Context Broker. Nosso trabalho se difere dos demais citados já que ele considera aspectos de encriptação, empacotamento e envio dos dados pelo sensor com o respectivo impacto gerado na infraestrutura computacional, considerando uma plataforma de IoT e conceitos de computação em névoa.

Resultados de experimentos de avaliação de desempenho da plataforma FIWARE em cenários de agricultura foram apresentados em Zyrianoff et al. (2018) e Kamienski et al. (2019). No entanto, esses artigos não incluem a avaliação de componentes da arquitetura LoRaWAN, nem sua comparação com o uso da comunicação LoRa básica, como é mostrado aqui.

4. Implementação da Arquitetura LoRaWAN

LoRaWAN define o protocolo de acesso ao meio para que vários dispositivos possam se comunicar com um gateway que usa a modulação LoRa, assim como define mecanismos de segurança, como autenticação e criptografia. Os dados em um pacote podem variar até 255 bytes, atingindo taxas de dados de até 50 Kbps. LoRaWAN define também o mecanismo ADR (*Adaptive Data Rate*) em que o servidor LoRa pode alterar parâmetros de configuração dos dispositivos para otimizar a taxa de transmissão, tempo no ar e consumo de energia [Sornin et al. 2016].

A implementação típica de uma rede LoRaWAN é mostrada na Figura 1. O *end node* possui uma aplicação desenvolvida para transmitir dados com a tecnologia LoRa, usando uma camada de abstração de hardware (HAL) e o protocolo SPI (*Serial Peripheral Interface*) para comunicação do microcontrolador com os diversos componentes do sensor. A camada física gera o pacote LoRaWAN e transmite para o gateway, que encaminha os dados para o servidor de rede via protocolo MQTT – um protocolo típico de IoT. O servidor de rede LoRaWAN envia as informações a um servidor de aplicação LoRaWAN.

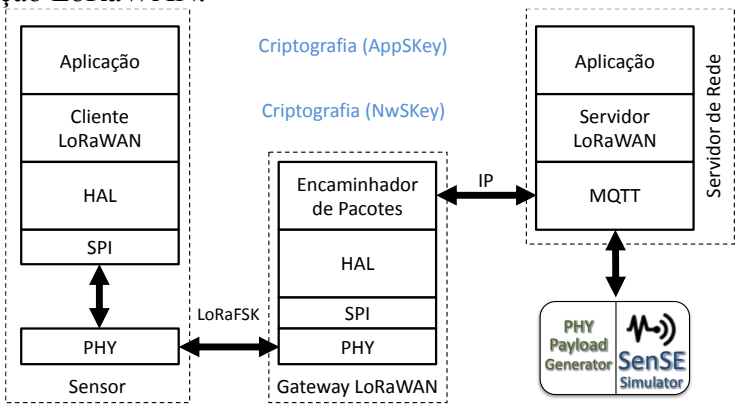

Figura 1. Arquitetura LoRaWAN e ponto de injeção dos dados gerados

LoRaWAN permite que pacotes sejam autenticados e encriptados com o algoritmo AES [Sornin et al. 2016]. A chave *Network Session Key (NwkSKey)* é conhecida pelo nó sensor e o servidor de rede LoRa, sendo usada para verificação de integridade das mensagens. A chave *Application Session Key (AppSKey)* deve ser conhecida pelo nó sensor e o servidor de aplicação LoRaWAN e é usada para encriptação e decriptação dos dados. Existem dois métodos para implementar estas chaves nos dispositivos LoRa:

ativação pelo ar (OTAA – *Over-the-Air Activation*) e ativação por personalização (ABP – *Activation by personalization*) [Sornin et al. 2016], utilizada nesse trabalho.

Para realizar a avaliação de desempenho da arquitetura LoRaWAN, é necessário gerar o tráfego de uma grande quantidade de dispositivos de IoT, enviados para um servidor de rede LoRa. Para este fim, foi utilizado o gerador de tráfego SenSE (*Sensor Simulating Environment*) e o programa PHYPayload para geração do *payload* de camada física. Como não existem geradores de *payload* para comunicação LoRaWAN disponíveis adequados à essa demanda, o programa PHYPayload foi desenvolvimento especialmente para esse artigo, mas pode ser utilizado em outras avaliações por terceiros. PHYPayload utiliza as mesmas chaves de criptografia utilizadas pelo sensor LoRaWAN para criar um *payload* com a mesma estrutura e integridade de dados de um *payload* real.

A implementação da arquitetura LoRaWAN com seus servidores de rede e de aplicação pode ser feita de algumas formas diferentes: a) Software de código livre que pode ser instalado na nuvem ou na névoa de acordo com o cenário desejado, como o LoRa Server Project[6]; b) The Things Network[7] (TTN), uma comunidade viabilizada por *crowdfunding* que implementa os servidores LoRaWAN e os disponibiliza através de um serviço web. No entanto, alguns serviços da TNN são pagos; c) Soluções proprietárias e pagas, como Loriot[8]; d) Desenvolvimento de nova implementação para LoRaWAN, que representaria um retrabalho, uma vez que existem outras opções disponíveis.

Neste trabalho foi utilizada a implementação disponibilizada pelo *Lora Server Project* (referenciado como módulos LoRaWAN ou loraserver no restante do trabalho), pois é de código aberto, gratuita e pode ser implementada em uma rede privada. Foram utilizados os seguintes componentes do LoRa Server Project: LoRa Server (servidor de rede) e LoRa App Server (servidor de aplicação). Para banco de dados, foi utilizado REDIS[9] e Postgres[10].

Nesse trabalho os *end-nodes* e *gateways* LoRa são abstraídos pela ferramenta SenSE com a adição do programa PHYPayload desenvolvido especialmente para essa finalidade. A abstração dos sensores e gateways não prejudica a comprovação da principal hipótese de pesquisa, uma vez que o objetivo é avaliar o comportamento e efeito dos componentes computacionais da arquitetura LoRaWAN numa implementação de IoT, e não os aspectos de modulação comunicação da camada física. Além disso, não existem dados disponíveis de milhares de sensores LoRaWAN para avaliar o desempenho e escalabilidade da solução.

5. Metodologia

O objetivo principal do trabalho é entender o impacto computacional da arquitetura de software do LoRaWAN, compará-la com o uso da comunicação LoRa básica (sem

[6] loraserver.io
[7] thethingsnetwork.org
[8] loriot.io
[9] redis.io
[10] postgresql.org

LoRaWAN) e sua integração com plataformas de IoT. Para isso, foram executados experimentos com diferentes configurações da plataforma de IoT distribuídos na nuvem e na névoa e com LoRaWAN e LoRa básico – ou seja, com e sem os módulos do loraserver. A Figura 2 ilustra todas as configurações exploradas.

5.1. Cenário e Ambiente dos Experimentos

Um dos objetivos principais desse trabalho é entender o impacto computacional da arquitetura de software do LoRaWAN e sua integração com plataformas de IoT. Para isso avaliamos duas configurações diferentes: utilizando LoRaWAN (arquitetura de software loraserver - configurações A e B da Figura 2) e sem LoRaWAN (configurações C e D da Figura 2), na qual o gateway LoRa envia dados diretamente para a aplicação. A importância de tal comparação se justifica devido ao pouco tempo de existência da tecnologia e a de avaliações em grande escala. Muitas instalações menores dispensam o uso da arquitetura LoRaWAN considerando-a um *overhead* desnecessário para a escala da implementação.

O ambiente desses experimentos é baseado em cenários reais de sociedades inteligentes, modelados com base em pilotos do projeto SWAMP[11], uma colaboração entre instituições e empresas do Brasil e da Europa que visa desenvolver métodos e abordagens baseados em IoT para a irrigação de precisão na agricultura, com um foco experimental [Kamienski et. al. 2019]. Apesar dos cenários avaliados serem baseados em ambientes de agricultura inteligente, acreditamos que não possuam características muito específicas, tornando os resultados potencialmente adequados para ambientes urbanos e sistemas baseados em IoT de modo geral que usem LoRaWAN. É possível fazer essa afirmação uma vez que LoRaWAN e névoa são tecnologias adequadas para diferentes cenários de IoT. Além disso, a camada física, que poderia trazer diferenças, por exemplo entre cidades e campo, não está sendo avaliada.

Além da diferença de configuração envolvendo LoRa, também avaliamos duas formas diferentes de implantação da plataforma de IoT FIWARE: 1) Configurações A e C da Figura 2: toda a plataforma FIWARE está apenas na nuvem computacional, sendo que o apenas módulos do loraserver estão na névoa; 2) Configurações B e D da Figura 2: parte da plataforma está na névoa computacional, dessa maneira, dados podem ser consumidos localmente sem a necessidade de ter que se conectar através de uma WAN. A Figura 2 mostra o fluxo de dados entre os componentes de software avaliados e as diferentes formas para sua instalação em gateway, névoa e nuvem são apresentadas nas figuras subsequentes.

O *trade-off* entre ter a plataforma de IoT somente na nuvem ou distribuída entre a névoa e a nuvem é particularmente interessante de ser explorado. Por um lado, adicionamos mais módulos de software, que potencialmente podem comprometer o desempenho do sistema, todavia, esses módulos na névoa são importantes em cenários de agricultura inteligente, pois a conexão entre a fazenda e a internet é muito instável e dessa maneira serviços locais ainda podem funcionar normalmente em caso de desconexões com a nuvem. Em ambientes urbanos, o nó da névoa configurado desse modo diminui consideravelmente a latência do serviço, possibilitando aplicações em

[11] swamp-project.org/

tempo real. Muitos dos módulos de software apresentados na Figura 2 já foram apresentados nas seções anteriores, no entanto, alguns são novos. São eles:

- Mosquitto: Eclipse Mosquitto é um *broker open source* de mensagens MQTT. Ele é utilizado inclusive internamente pelo LoRa Server Project para comunicação entre os servidores e pela The Things Network para instalação de redes privadas LoRaWAN[12];
- FIWARE-based IoT Agent: No presente momento, não há uma versão oficial do IoT Agent que suporte a comunicação LoRaWAN. Devido a isso, nossa equipe programou um novo IoT Agent, capaz de mapear as mensagens LoRaWAN enviadas pelo LoRa App Server até o Orion Context Broker. Esse módulo foi programado em JavaScript, executado pela *engine* do Node.js;
- Consumer: O Consumer é uma aplicação web Express.js que se inscreve no Orion e recebe dados dos sensores. Quando o Orion envia uma mensagem para a API do consumidor, um registro de data e hora é feito e é subtraído do *timestamp* de geração da mensagem.
- SenSE: um gerador de tráfego de sensores IoT em larga escala, capaz de abstrair dispositivos reais e modelar cenários complexos, como *smart farms* e *smart cities*. Embora os sensores sejam sintéticos, o tráfego é real [Zyrianoff, et. al. 2017].

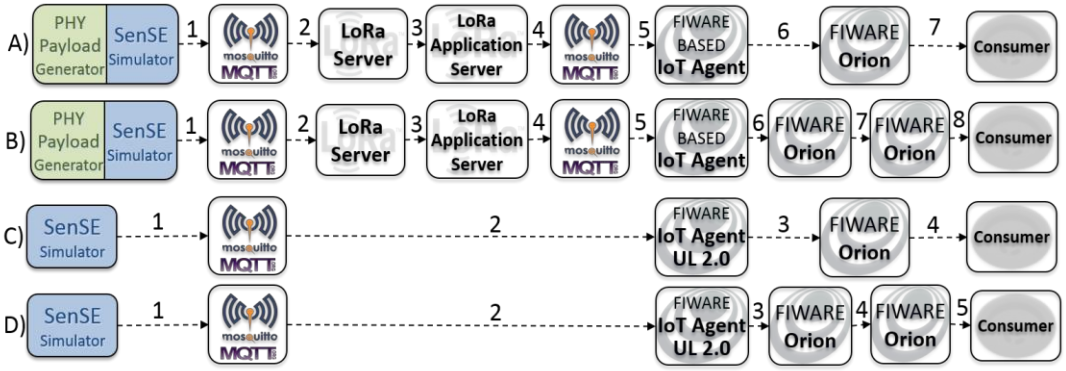

Figura 2. Diferentes fluxos de dados das configurações dos experimentos

Em todas as configurações, os módulos foram encapsulados em contêineres Docker. As Figuras 3, Figura 4, Figura 5 e Figura 6 mostram os cenários dos experimentos para cada configuração, em detalhes, respectivamente. Para emular um ambiente mais próximo do real, os experimentos foram feitos em um ambiente de nuvem privada (Openstack Queens), onde instanciou-se uma máquina virtual (VM) para o SenSE, outra representando o nó da névoa e uma última representando a nuvem computacional. Para simular a distância entre o nó da névoa e a nuvem, utilizou-se um emulador de rede, o WANem[13]. As VMs foram configuradas de acordo com os padrões de máquinas virtuais da Amazon AWS, sendo que para a nuvem foi usada uma máquina

[12].thethingsnetwork.org/article/setting-up-a-private-routing-environment

[13] wanem.sourceforge.net/

t2.medium (4GB e 2vCPUs) e para representar o nó da névoa *t2.small* (2GB RAM e 1vCPU), já que a névoa possui recursos computacionais limitados. Devido a quantidade de módulos na névoa na configuração B foi necessário dividir entre dois nós de névoa.

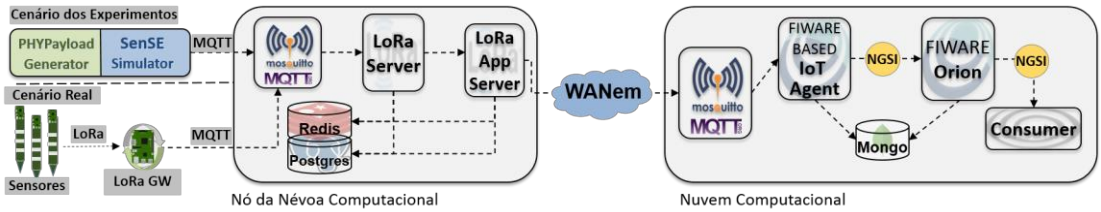

Figura 3. Cenário da configuração (a)

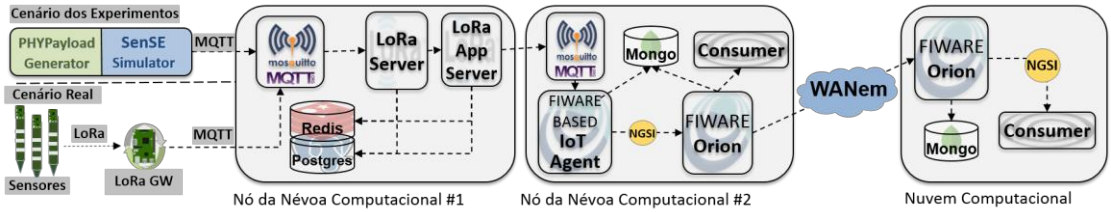

Figura 4. Cenário da configuração (b)

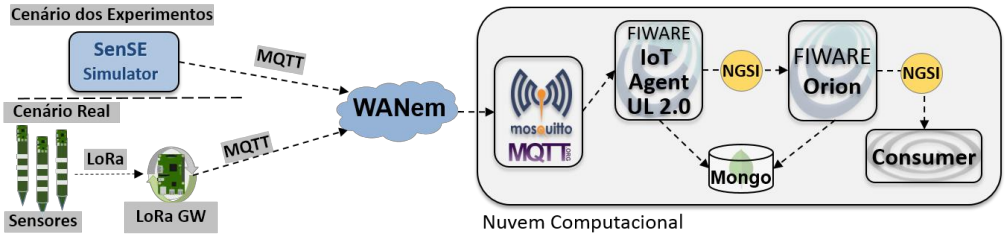

Figura 5. Cenário da configuração (c)

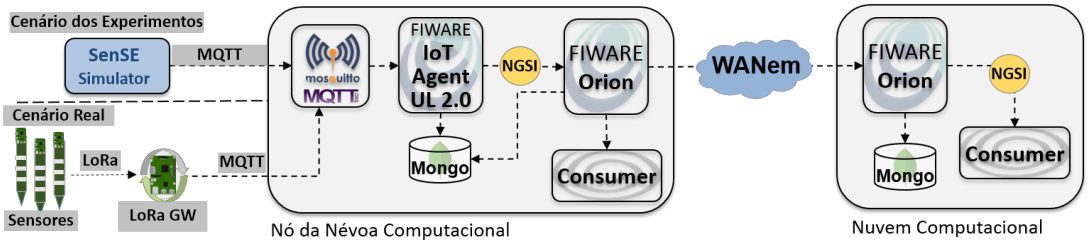

Figura 6. Cenário da configuração (d)

Em todos os experimentos, o SenSE emula os sensores e o LoRa gateway. Nos cenários utilizando a arquitetura de software LoRaWAN (Figura 3 e Figura 4), o SenSE utiliza o PHYpayload Generator para gerar os dados que enviará para o LoRa Server, que irá realizar a autenticação e o decriptação dos dados. Nos experimentos em que os dados são enviados diretamente para o FIWARE (Figura 5 e Figura 6), não é necessário gerar o PHYPayload. Ressalta-se mais uma vez que a motivação para cenários com e sem a arquitetura LoRaWAN é a comparação com implementações simples que usam apenas a comunicação LoRa na camada física entre o *end-node* (sensor) e o gateway, e o gateway envia dados diretamente para um *broker* MQTT, sem os recursos de controle de acesso ao meio, segurança e autenticação.

Para entender completamente o cenário de avaliação, é necessário compreender o modelo de dados das mensagens enviadas pelo SenSE em cada cenário. Para isso foi adotado o protocolo Ultralight 2.0 (UL) - um protocolo leve baseado em texto destinado a dispositivos e comunicações restritos onde a largura de banda e a memória do dispositivo podem ser escassos – para as mensagens enviadas diretamente para a arquitetura FIWARE. Nesse caso, cada sensor envia uma mensagem que consiste em sete valores (`t1|v|t2|v|t3|v|m1|v|m2|v|m3|v|c|v|ts|v`, no protocolo UL, sendo que o *v* representam valores gerados aleatoriamente), simulando uma sonda de solo para agricultura [Zyrianoff, et. al. 2018], compondo um *payload* de 65 bytes. No caso do envio para o loraserver, é necessário enviar muitos outros dados com informações dos *end-nodes*, como modulação LoRa, gateway e segurança, então, apesar dos sensores simulados apenas enviarem um único valor, o *payload* é muito maior do que nos experimentos sem o loraserver, consistindo de 314 bytes.

5.2. Métricas

Dois conjuntos diferentes de métricas foram utilizadas nos experimentos:

- Tempo médio de processamento: o tempo médio decorrido desde que o dado do sensor é gerado até atingir o Consumer. Aqui avalia-se o tempo decorrido desde que os dados são gerados até que os mesmos estejam prontos para serem usados por outro aplicativo - como um *dashboard*, um módulo de *analytics* ou um tipo de *reasoner* que irá atuar em algum dispositivo.
- Métricas de sistema: foi coletado uso de CPU e memória para cada container Docker. Os dados são registrados com uma periodicidade de 5 segundos.

A análise da escalabilidade das diferentes configurações decorre diretamente do efeito do aumento da carga de trabalho (descrita a seguir) sobre o tempo médio de processamento e a completude dos experimentos.

5.3 Experimentos

Cada experimento consiste do SenSE simulando sensores e os enviando a um *broker* MQTT, que pode estar conectado a uma arquitetura LoRaWAN ou diretamente a uma plataforma de IoT. Para entender a escalabilidade do LoRaWAN, foram realizados experimentos com uma grande quantidade de sensores enviando dados simultaneamente, cada sensor envia uma mensagem a cada 10 minutos. Foram aplicadas em cada configuração (da Figura 6) quatro diferentes cargas de trabalhos: 1.000, 5.000, 10.000 e 15.000 sensores enviando dados, totalizando 16 experimentos, sendo que cada experimentos foi replicado 30 vezes. O intervalo de confiança assintótico calculado foi de 99%. As condições de rede emuladas pelo WANem nos experimentos foram obtidas replicando uma conexão de uma rede 4G até uma nuvem computacional, os parâmetros configurados foram de uma conexão de 10 Mbps com uma latência de 45 ms e 5 ms de *jitter*. O uso de 4G em ambientes rurais é comum, principalmente em casos onde a comunicação vai diretamente do campo (gateway LoRa) até a nuvem, sem passar pela sede da fazenda. A tecnologia 4G também é usado em cidades, quando sensores e gateways estão instalados em locais sem o auxílio de outras conexões com ou sem fio à Internet.

6. Resultados

Os resultados para o tempo médio de processamento em todas as configurações apresentadas na Figura 2, estão apresentadas na Figura 7. Em geral, os resultados mostram que utilizando ou não a arquitetura LoRaWAN o sistema permanece escalável e com bom desempenho, exceto nos experimentos com 15.000 sensores enviando dados continuamente. Sendo que apenas a configuração A não apresentou falhas quando submetida a essa carga de trabalho, as configurações C e D caíram durante esses experimentos e a configuração B caí logo após o início dos experimentos. O valor numérico máximo do eixo y foi definido como 600 para melhor visualização, no entanto, as configurações C e D apresentaram tempos de: 676,80 ± 457,23ms e 17429.89 ± 36165,70ms, respectivamente.

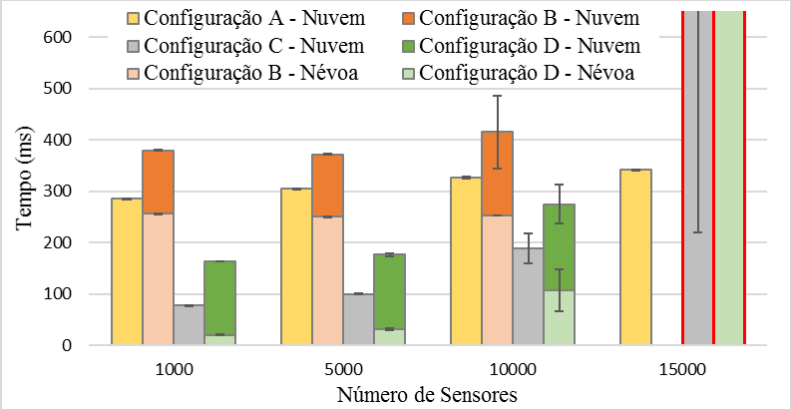

Figura 7. Tempo de processamento de cada configuração

Com relação a comparação dos experimentos utilizando o LoRaWAN (configuração A e B) e sem utilizá-lo (configuração C e D), observa-se que, nas cargas de trabalho baixas, a arquitetura LoRa adiciona cerca de 200ms ao processamento. Esse aumento no tempo de processamento era esperado, pois se adiciona módulos ao fluxo de dados que tratam questões de acesso ao meio, segurança e autorização - que demandam processamento.

Um resultado não intuitivo é a estabilidade e escalabilidade da configuração A, a única que não apresentou problemas na mais alta carga de trabalho. Isso acontece porque a arquitetura LoRaWAN controla melhor o fluxo de dados do que o IoT Agent UL. Nas configurações em carga alta em que os dados são enviados diretamente para IoT Agent UL em a plataforma caí, podemos observar o aumento exponencial do uso de recursos do IoT Agent UL na Figura 9. A plataforma também apresenta falhas na configuração B devido quantidade de módulos computacionais sendo executadas na névoa (que possui recursos computacionais limitados).

As configurações B e D são as únicas nas quais é possível consumir dados diretamente da névoa, possibilitando que aplicações processem parte dos dados antes da nuvem computacional e, portanto, responder a mudanças em tempo real. Na Figura 7 vemos que a maior parte do tempo processamento da configuração B está concentrado na névoa – devido aos módulos LoRaWAN – enquanto o oposto ocorre para a configuração D. Para determinadas aplicações em que um baixo tempo de resposta é essencial a arquitetura LoRaWAN pode não atender as restrições.

A Figura 8 mostra os resultados para recursos de sistema (uso de CPU e memória) para a configuração B. Por limitações de espaço, só iremos apresentar as métricas de sistema para essa configuração, mas não há grandes variações entre as configurações. O FIWARE Orion é o módulo que mais usa recursos do sistema, o que é esperado pois ele lida com diversas operações sobre os dados na forma mais verbosa do fluxo de dados (NGSI JSON). Não vemos crescimento do uso de memória de 10.000 para 15.000 sensores pois o fluxo de dados está limitado a taxa de serviço dos módulos LoRaWAN, sendo que o resto das mensagens são enfileiradas, dessa maneira, o uso desse recurso se mantém o mesmo.

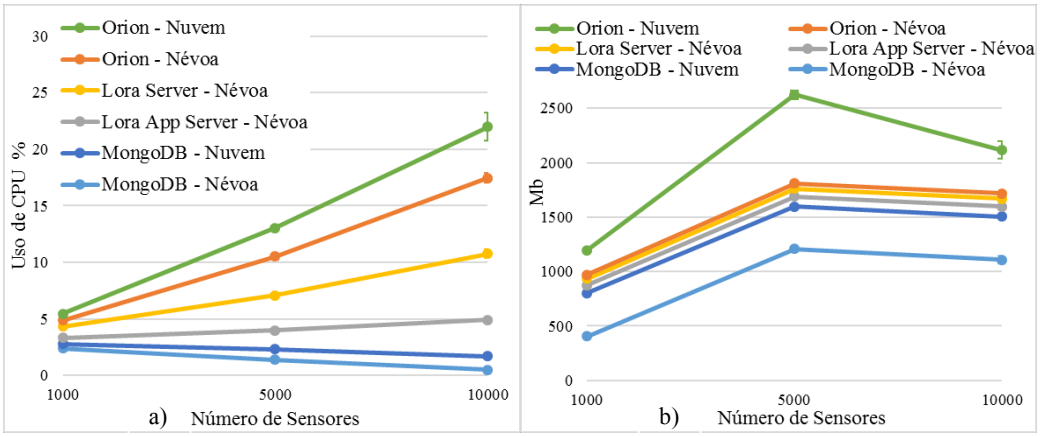

Figura 8. Recursos do sistema da configuração B; a) uso de CPU b) uso de memória

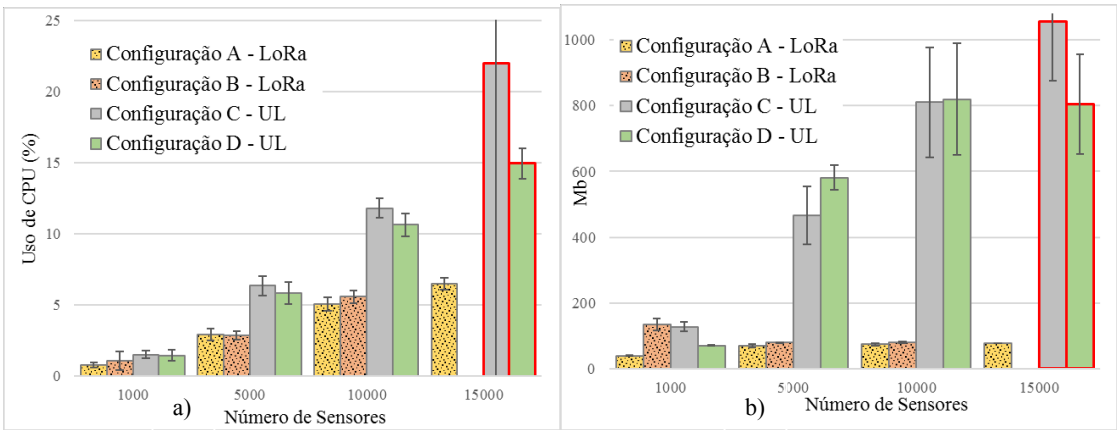

Figura 9. Uso de recursos de sistema do IoT Agent; a) uso de CPU; b) uso de memória

Um comparativo interessante é entre o IoT Agent LoRa (desenvolvido pelos autores) e o IoT Agent UL oficial do FIWARE. A Figura 9 mostra o uso de recursos computacionais desse módulo para cada configuração, podemos observar que o IoT Agent UL é menos escalável, aumentando o uso de recursos à medida que sobe o processamento, sendo que cai em cargas mais altas (15.000 sensores). Em contrapartida, o IoT Agent LoRa mantém o consumo de recursos estável para as cargas de trabalho apresentadas. Esse resultado é uma contribuição importante, pois mostra que a integração dos módulos LoRaWAN com o FIWARE foi realizada de maneira eficiente e pode ser utilizada em outros projetos.

7. Discussão

A partir da cuidadosa análise dos resultados dos experimentos observa-se que o loraserver se mostrou escalável, mantendo o tempo de processamento estável mesmo com o aumento da carga de trabalho. Existe um impacto computacional que a arquitetura LoRaWAN adiciona ao sistema, principalmente em cargas de trabalho baixas quando comparado a uma situação sem eles, um resultado esperado e intuitivo que pode significar que para algumas aplicações que necessitem de tempo real, a implantação da arquitetura LoRaWAN não seja viável. No entanto quando se trata do gerenciamento de uma rede com muitos sensores, é necessário possuir uma camada de controle de acesso ao meio, além de tratar de questões como segurança e autenticação.

Contra intuitivamente, a configuração mais escalável foi a com apenas módulos LoRaWAN executados na névoa e o FIWARE na nuvem. Isso é devido ao tratamento de fluxo de dados do LoRaWAN e a falta de escalabilidade do IoT Agent UL do FIWARE.

Os experimentos revelaram que o gargalo do sistema como um todo é o IoT Agent – um módulo da plataforma FIWARE de IoT – e não os módulos LoRaWAN, já que esse módulo é o primeiro que apresenta falhas quando submetido a uma carga de trabalho alta.

A integração dos módulos do loraserver com a plataforma de IoT foi mais trabalhosa do que foi previsto pois ainda não existem muitas aplicações que conseguem se comunicar com o LoRa Application Server de forma transparente. Tanto que para viabilizar os experimentos, foi necessário desenvolver um IoT Agent que conseguisse realizar essa integração, sendo a ponte entre os módulos LoRaWAN e o FIWARE. Os resultados mostram que o IoT Agent desenvolvido por nós possui melhor desempenho do que o IoT Agent UL do FIWARE, nas condições analisadas nos experimentos.

Há também a discussão na implementação de um nó de névoa e nas possíveis configurações e usos desse. Em alguns cenários, somente foram executados módulos referentes ao LoRa, e em outros, parte da plataforma de IoT para que os dados pudessem ser consumidos localmente. No primeiro caso, há o processamento hierárquico da informação, no entanto, os dados processados na névoa não são utilizados localmente, sendo necessário envia-los para nuvem para que possam ser consumidos por outras aplicações. Na segunda configuração, os dados podem ser usados por aplicações locais próximas de onde os dados são gerados, possibilitando aplicações tempo real e robustez do sistema a possíveis desconexões. Os resultados obtidos mostram que todas essas características podem ser acrescentadas sem grandes perdas no desempenho geral do sistema. A relevância desse trabalho está na avaliação de um cenário IoT complexo fim-a-fim, desde a geração do dado até ser consumido por alguma aplicação. Considerando aspectos de rede e sistema, além da introdução da névoa computacional e comparando esses resultados com diversas possíveis configurações.

9. Conclusão

Esse artigo investigou o impacto computacional no desempenho e escalabilidade da arquitetura LoRaWAN em cenários complexos de IoT e sua integração com uma plataforma de IoT (FIWARE) em diferentes configurações que exploram *trade-offs* entre a computação de névoa e a computação em nuvem. Os resultados obtidos mostram

que a arquitetura LoRaWAN possui impacto no sistema, entretanto é escalável e factível de ser implementada em ambientes alta carga de trabalho, tanto em nuvem quanto em névoa. Como trabalhos futuros deseja-se realizar experimentos explorando outras possíveis configurações além de comparar a arquitetura LoRaWAN com outras tecnologias LPWAN que necessitem de algum módulo de software.

8. Agradecimentos

Essa pesquisa foi parcialmente financiada pelo projeto SWAMP [Kamienksi et. al. 2018], uma colaboração entre Brasil e União Europeia.

Referências

Ayele, E. D., Hakkenberg,C., Meijers, J. P. Zhang, K. Meratnia,N., and Havinga, P. J. M., (2017) "Performance Analysis of LoRa Radio for an Indoor IoT Applications," in Internet of Things for the Global Community (IoTGC).

Bonomi, F. Milito, R., Zhu, J., Addepalli, S., (2012). "Fog Computing and its role in the Internet of Things", IEEE Workshop on Mobile Cloud Computing (MCC), pp. 13-16.

da Cruz, M.A., Rodrigues, J.J., Sangaiah, A.K., Al-Muhtadi, J., Korotaev, V., (2018) "Performance Evaluation of IoT Middleware, Journal of Network and Computer Applications, 109, pp.53-65.

Ericsson, "Cellular Networks for Massive IoT", White Paper, UEN 284 23-3278, (2016) [Online], www.ericsson.com/res/docs/whitepapers/wp_iot.pdf, acesso em 10/10/2018.

Georgiou, O. and Raza, U. (2017). "Low Power Wide Area Network Analysis: Can LoRa Scale?" *IEEE Wireless Communications Letters*, vol. 6, no. 2, pp. 162–165.

Kamienski, C., Soininen, J.P., Taumberger, M., Dantas, R., Toscano, A., Cinotti, T.S., Maia, R.F. Torre Neto, A., "Smart Water Management Platform: IoT-Based Precision Irrigation for Agriculture", Sensors, 19, p.276, Janeiro 2019.

Magrin, D., Centenaro, M. e Vangelista, L. (2017). "Performance evaluation of LoRa networks in a smart city scenario", in 2017 IEEE International Conference on Communications (ICC), p. 1–7.

Martínez, R., Pastor, J.Á., Álvarez, B., Iborra, A., (2016) "A Testbed to Evaluate the FIWARE-based IoT Platform in the Domain of Precision Agriculture, Sensors, 16(11).

Petäjäjärvi, J., Mikhaylov, K., Yasmin, R., Hämäläinen, M. e Iinatti, J. (2017). "Evaluation of LoRa LPWAN Technology for Indoor Remote Health and Wellbeing Monitoring", *Int J Wireless Inf Networks*, vol. 24, nº 2, p. 153–165.

Raza, U., Kulkarni, P. e Sooriyabandara, M. (2017). "Low Power Wide Area Networks: An Overview", *IEEE Communications Surveys Tutorials*, vol. 19, nº 2, p. 855–873.

Sornin, N., Luis, M., Eirich, T., Kramp, T. e Hersent, O. "LoRaWAN Specification v1.0.2," (2016). [Online]. Available: www.loraalliance.org/portals/0/specs/. Acesso em 04/12/2018.

Van den Abeele, F., D., Haxhibeqiri, J., Moerman, I., Hoebeke,J. (2017). "Scalability Analysis of Large-Scale LoRaWan Networks in ns-3", *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 2186-2198.

Zyrianoff, I.; Borelli, F.; Kamienski, C., (2017), "SenSE – Sensor Simulation Environment: Uma ferramenta para geração de tráfego IoT em larga escala", SBRC 2017. Salão de Ferramentas.

Zyrianoff, I.; Heideker, A.; Otollini, D; Kamienski, C., (2018), "Scalability of an Internet of Things Platform for Smart Waer Management for Agriculture", FRUCT 2018.

Heap Allocator: Uma Polıtica de Alocação de Máquinas Virtuais em Ambiente de Computação em Nuvens baseada em Heap com Prioridade

Matheus Magalhães de Carvalho[1], Francisco Mardônio Vieira Filho[1], Filipe Fernandes S B de Matos[2], Rafael Lopes Gomes[1], Joaquim Celestino Júnior[1]

[1]Universidade Estadual do Ceará (UECE)
Av. Dr Silas Munguba, 1700, Fortaleza-CE

[2]Universidade Federal do Ceará (UFC)
BR 226, KM 4 - Venâncios, Crateús-CE

{matheus.magalhaes,celestino}@larces.uece.br

{mardoniovf,rafaellgom}@gmail.com,filipe.fernandes@crateus.ufc.br

Abstract. *Cloud Computing represents a paradigm that provides computing resources on virtual machines that are grouped and allocated according to customer requests. The increasing seeking for this type of service caused an increasing demand for electric energy. Several types of research aim to meet the demand for resources in Cloud Computing, and at the same time, reduce the energy consumption in the data center. Within this context, this paper presents a virtual machine allocation policy called Heap Allocator, which mitigates the power consumption of the data center. A comparative analysis of the proposed solution with other existing mechanisms showed the proposed solution reduces the data center power consumption by approximately* 1.5%.

Resumo. *Computação em Nuvens representa um paradigma computacional onde recursos físicos são agrupados e alocados em máquinas virtuais de acordo com as solicitações dos clientes. A crescente busca por esse tipo de solução ocasionou uma maior demanda por energia elétrica. Diversas pesquisas visam atender a contínua demanda de recursos em Computação em Nuvens, e ao mesmo tempo, melhorar o consumo energético no data center. Dentro deste contexto, este artigo apresenta uma Política de Alocação de Máquinas Virtuais chamada* Heap Allocator*, que busca mitigar o consumo de energia de um data center. Uma análise comparativa da proposta com outras políticas existentes mostrou uma redução do consumo de energia do data center em aproximadamente* $1,5\%$.

1. Introdução

Atualmente, muitos sistemas computacionais e sistemas de informação estão substituindo seu modelo tradicional de computação por um novo conhecido como Computação em Nuvens. O modelo tradicional compreende a divisão das máquinas entre clientes e servidoras. Os servidores possuem grande capacidade de processamento e armazenamento e disponibilizam seus recursos e serviços às máquinas clientes através de uma rede como,

por exemplo, a Internet. O papel das máquinas clientes é interagir com os usuários e consumir tudo aquilo que é ofertado pelos servidores. Um *data center* representa um conjunto de servidores e de equipamentos de rede, que interliga esses servidores e os conectam aos usuários através da Internet [Rong et al. 2016]. Um problema decorrente do modelo tradicional de computação é o desperdício de recursos pela infraestrutura do *data center*. Neste modelo, cada servidor hospeda uma única aplicação ou serviço que, devido a variação constante de demanda, pode, muitas vezes, subutilizar os recursos computacionais da máquina em questão. Este sub-uso pode levar a uma ociosidade dos recursos e ocasionar gastos e desperdícios para o *data center*.

Para solucionar esse problema, o modelo de Computação em Nuvens emprega a técnica da Virtualização. Tal mecanismo consiste na virtualização dos recursos físicos do *hardware* e os oferece como recursos virtuais às aplicações de alto nível [Zhang et al. 2010]. Dessa forma, é possível implementar várias aplicações dentro de um mesmo servidor físico. Cada aplicação é inserida em uma Máquina Virtual (VM) para a execução. Uma VM consiste de um subconjunto de recursos virtuais do servidor físico de forma que não exista sobreposição entre elas.

A crescente procura por serviços em Nuvem motivou o aumento no número e no tamanho dos *data centers*. Um dos problemas relacionados a essa expansão é o alto consumo de energia do *data center*. Para se ter uma idéia, em 2013, segundo [Delforge 2014], estima-se que apenas os *data centers* dos Estados Unidos consumiram 91 bilhões kilowatts-hora de eletricidade e que tal valor seria suficiente para abastecer todas as residências da cidade de Nova York por dois anos seguidos. Outros estudos revelam que o índice de crescimento no consumo energético dos *data centers* é de 18% ao ano e que, até 2020, eles serão os responsáveis por 2% do consumo mundial de energia [Sharma and Reddy 2015]. Um estudo mais recente da Forbes [Forbes 2017] aponta que todos os *data centers* do planeta consumiram cerca de 416 terawatts de energia. Tal valor representa cerca de 40% a mais do consumo energético de todo o Reino Unido.

O presente trabalho propõe uma nova Política de Alocação de Máquinas Virtuais com o objetivo de reduzir o consumo de energia elétrica de um *data center* em ambiente de Computação em Nuvens. A política em questão utilizará uma estrutura de dados conhecida como MaxHeap para definir o servidor com a maior eficiência energética e escolhê-lo para receber uma VM em processo de alocação.

O restante deste trabalho está organizado da seguinte forma: A Seção 2 abordará os principais trabalhos relacionados com a técnica aqui proposta. A Seção 3 apresentará os principais conceitos associados a estrutura de dados Heap. A Seção 4 descreverá a proposta deste trabalho: a Política de Alocação HeapAllocator. A Seção 5 apresentará as configurações do cenário utilizado na simulação e as métricas usadas para a avaliação de desempenho das Políticas. A Seção 7 apresentará as conclusões obtidas e possíveis trabalhos futuros.

2. Trabalhos relacionados

Em [Beloglazov and Buyya 2012], o problema de alocação de máquinas virtuais, considerando o consumo de energia, é dividido em duas partes: 1) Escolha do servidor que receberá a VM; 2) Otimização da localização das VMs alocadas. No primeiro caso, é realizada a modelagem como um problema do empacotamento e é utilizado o algoritmo

Modified Best Fit Decreasing. Nesse algoritmo, a lista de máquinas virtuais a serem alocadas são ordenadas em ordem decrescente de acordo com a taxa de utilização de CPU. Em seguida, cada máquina virtual é alocada no servidor no qual haverá o menor aumento de consumo de energia após a alocação. No segundo caso são utilizadas políticas de migração de VMs onde são estabelecidos o limite inferior e o superior de utilização dos recursos de um servidor. O objetivo é manter todos os servidores ativos dentro da faixa estabelecida pela política. Apesar de apresentar bons resultados para os cenários testados, essa abordagem não diferencia os servidores em relação a sua capacidade de processamento e seu consumo de energia. Desse modo, em um cenário heterogêneo o algoritmo pode consolidar as máquinas virtuais em um servidor onde a eficiência energética é baixa, ocasionando um consumo de energia maior do que o esperado.

Em [Nasim et al. 2016], os autores apresentam uma técnica para realizar a alocação de máquinas virtuais considerando a energia elétrica consumida e os recursos utilizados por elas: processamento, memória principal e memória secundária. Sua estratégia aloca máquinas virtuais que necessitam intensamente de um mesmo tipo de recurso em servidores diferentes. O autor recorreu ao conceito de sensibilidade formulado por [Taheri et al. 2017]. A sensibilidade de uma máquina virtual por um recurso computacional é diretamente proporcional ao nível que ela necessita desse recurso, ou seja, o quanto seu desempenho é degradado caso esse recurso lhe seja disponibilizado de maneira insuficiente. Os autores afirmam que essa técnica não é uma heurística rápida, por isso realizaram os experimentos com pequenas instâncias para o problema de alocação de máquinas virtuais, especificamente com 45 máquinas virtuais e 20 máquinas físicas. Todas as máquinas físicas possuem a mesma configuração, o que representaria um data center homogêneo.

Em [Nasim and Kassler 2017], os autores acreditam que para realizar uma consolidação energeticamente eficiente deve-se conhecer as demandas de cada máquina virtual por recursos computacionais. Eles afirmam também que esses dados são difíceis de se obter, uma vez que a carga de trabalho dos serviços em execução nas máquinas virtuais sofrem variação com um curto intervalo de tempo. Uma estratégia robusta de alocação deve realizar a consolidação de forma que o servidores suportem tal variação sem violar demasiadamente o SLA. A heurística utilizada foi baseada na Busca Tabu (TS). O autor acrescentou em sua estratégia ideias de Otimização Robusta, que trabalha com problemas onde não se tem dados precisos, mas que podem ser limitados. Para isso a função de avaliação foi modificada adicionado um ruído R na solução atual, onde R representa as alterações esperadas nos dados de entrada. Ela observa quanto as máquinas virtuais podem variar para continuar atendendo a restrição de capacidade máxima definida por sua política. Observe que se a variação for maior que o limite definido, a política terá seu desempenho degradado.

Em [Okada et al. 2015], são propostos três algoritmos para a alocação de máquinas virtuais em um *data center*: um baseado no *First Fit Decreasing (FFD)* [Yue 1991] e outros dois derivados do *Power Aware Best Fit Decreasing* (PABFD) [Beloglazov 2013]. O FFD opera alocando VMs em servidores com base em uma lista ordenada (decrescente) de uso de CPU. Primeiramente, a lista de VMs é ordenada de forma descrescente com base no uso de CPU. Posteriormente, tal lista é percorrida, bem como a lista de servidores disponíveis. Cada VM é alocada no primeiro servidor com

recursos suficientes para armazená-la. A contribuição do trabalho de [Okada et al. 2015] é suspender todos os servidores que não receberam nenhuma máquina virtual durante o processo inicial de alocação.

O PABFD, por sua vez, trabalha com uma estimativa da energia antes da alocação efetiva da VM em um servidor. Assim, como no FFD, inicialmente, a lista de VMs é ordenada de forma decrescente de acordo com a utilização de CPU. As listas de VMs e de servidores também são percorridas. Entretanto, ao invés de alocar a VM no primeiro servidores com capacidade para armazená-la, é feito uma análise e é escolhido o servidor que sofrerá o menor aumento de energia após a alocação. Além da suspensão dos servidores que não possuem máquinas virtuais, a outra contribuição de [Okada et al. 2015] foi criar o GPABFD (*Global Power Aware Best Fit Decreasing*). O GPABFD é uma modificação na função que estima o aumento energértico da VM a ser alocado em um servidor. Ao contrário da proposta original, o GPABFD avalia o efeito energético em uma escala "global", ou seja, em relação a todo *data center*. Dessa maneira, a alocação é feita visando reduzir o consumo energético de todo o *data center* e não apenas do servidor local.

O primeiro problema de ambas as abordagens é que a eficiência energética dos servidores durante o processo de alocação não é analisada, o que pode ocasionar à uma resposta não ótima do problema, mesmo com a modificação sugerida pelo GPABFD que não utiliza a eficiência energética para escolher o melhor servidor para uma VM. Em segundo lugar, devido à natureza dinâmica de um *data center*, ordenar uma lista de VMs é algo difícil de ser realizado, pois tal lista estará sempre em constante modificação, devido à chegada ou eventuais saídas de VMs do *data center*. O *HeapAllocator* busca mitigar esses dois problemas.

3. Fundamentação Teórica

Uma *Heap* é uma estrutura de dados que permite a implementação de uma fila de prioridades [Lafore 2017]. Uma fila de prioridade é um tipo abstrato de dados que armazena uma coleção de elementos organizados de acordo com uma prioridade [Goodrich and Tamassia 2013]. Em uma fila de prioridade, os elementos são armazenados de maneira ordenada baseado nos seus níveis de prioridades.

A fila de prioridade pode ser classificada como de máximo ou de mínimo. A fila de máximo é aquela cujo a prioridade de cada pai é maior ou igual que qualquer um de seus filhos. Já no caso da fila de mínimo, ocorre o contrário, ou seja, a prioridade de cada pai é menor ou igual que qualquer de seus filhos. Dessa maneira, o primeiro elemento da fila será o maior e o menor dos valores da estrutura (para as filas de máximo e de mínimo, respectivamente).

Os dois principais métodos em uma fila de prioridade são: inserção e remoção de elementos. A operação de inserção consiste em adicionar um elemento na fila de prioridade, mantendo a ordenação dos níveis de prioridade. A operação de remoção remove e retorna o elemento com maior ou menor prioridade, de acordo com o tipo da estrutura implementada.

A estrutura *Heap* pode ser visualizada como uma árvore binária e representada através de um vetor. Cada elemento do vetor corresponde a um nó da árvore binária e o nó raiz é o maior ou o menor elemento de acordo com o nível de prioridade estabelecido.

Uma *Heap* implementa os métodos de inserção e remoção de uma fila de prioridades em tempo logarítmico [Cormen 2009]. No método de inserção da *Heap*, o novo elemento é sempre inserido no final da estrutura. Após isso, são realizadas comparações entre a prioridade do novo elemento com as prioridades dos elementos já inseridos na estrutura. Considerando uma *Heap* de prioridade máxima, então nesse caso, se o elemento possuir uma prioridade maior do que seu pai, eles serão trocados de posição. Esse comportamento é mantido até que a prioridade do nó pai seja maior do que a prioridade do novo elemento.

Já no procedimento de remoção, o topo da estrutura, que pode conter ou o maior ou o menor elemento de acordo com a prioridade previamente estabelecida, é sempre retirado. Após a remoção, a estrutura se reorganiza através de comparações. Ao remover o topo da *Heap*, o último elemento da estrutura é posto no lugar do elemento excluído. Assim, o novo topo será comparado com os seus nós filhos. Nesse caso, o elemento será permutado com o filho de maior prioridade. Esse comportamento é mantido até que os nós filhos do elemento tenham prioridades menores do que ele.

4. A Política Heap Allocator

A Política *HeapAllocator* utiliza uma estrutura *Heap* com prioridade máxima, cujo objetivo principal é reduzir o consumo de energia elétrica de um *data center*. Uma vantagem da *Heap* com prioridade é que suas operações de inserção e remoção possuem uma complexidade computacional na ordem $O(nlogn)$. Assim, espera-se que a *HeapAllocator* seja computacionalmente rápida. A *HeapAllocator* é uma Política de Alocação monocritério, sendo que neste trabalho, é considerada apenas a CPU (em MIPS) como critério para determinar se um servidor físico é capaz ou não de receber uma máquina virtual.

A estrutura *Heap* da *HeapAllocator* é responsável por ordenar informações de servidores que já possuem máquinas virtuais. A estrutura em *Heap* é ordenada de acordo com a métrica de eficiência energética (*M*) definida na Fórmula 1. Tal métrica é o resultado da divisão entre a eficiência energética do servidor em análise (E_f) e a taxa de CPU ociosa no mesmo servidor ($MIPS_{disp}$) naquele instante.

$$M = \frac{E_f}{MIPS_{disp}} \tag{1}$$

De acordo com [Lago et al. 2012], a eficiência energética (E_f) de um servidor é obtida através da divisão entre a quantidade máxima de MIPS por ele disponibilizada ($MIPS_{max}$) e a quantidade máxima de energia consumida por esse mesmo servidor (E_{cons}). A fórmula da eficiência energética pode ser visualizada na Fórmula 2.

$$E_f = \frac{MIPS_{max}}{E_{cons}} \tag{2}$$

Como o valor da métrica é diretamente proporcional à eficiência energética, quanto maior o seu valor, mais apropriada é a alocação de uma máquina virtual no servidor em questão. Dessa maneira, em uma *Heap* com prioridade máxima, o melhor candidato para receber uma nova máquina virtual, em termos de eficiência energética, estará sempre no topo da estrutura.

Os passos da política de alocação de máquinas virtuais *HeapAllocator* são descritos em Algoritmo 1.

Algoritmo 1: Política Heap Allocator

```
   Entrada: Lista de VMs, Lista de Hosts Vazios e Heap de Hosts
   Saída: VMs alocadas nos servidores
1  início
2      enquanto Lista de VMs não é vazia faça
3          VM atual recebe a primeira VM da Lista de VMs;
4          enquanto heap não está vazia faça
5              Host atual recebe o host do topo da heap;
6              se Host atual possui CPU disponível então
7                  Alocar VM atual no Host atual selecionado;
8                  Retirar VM atual da Lista de VMs;
9                  Atualizar Host atual na heap com prioridade;
10             fim
11             senão
12                 Retirar topo da heap;
13             fim
14         fim
15         se heap está vazia então
16             Retirar Host atual da Lista de Host Vazios;
17             Alocar VM atual no Host atual selecionado;
18             Inserir Host atual na heap com prioridade;
19             Retirar VM atual da Lista de VMs;
20         fim
21     fim
22 fim
```

O algoritmo recebe três estruturas como entradas: 1) Uma lista com máquinas virtuais passíveis de alocação no servidor (*Lista de VMs*); 2) Uma lista, ordenada com base na eficiência energética (E_f), com todos os servidores vazios, ou seja, sem nenhuma máquina virtual alocada (*Lista de Hosts Vazios*); 3) a *Heap de Hosts* com todos os servidores ativos, ou seja, que possuem ao menos uma máquina virtual alocada.

O algoritmo ainda possui duas variáveis relacionadas ao processo de decisão sobre a alocação de uma máquina virtual em um servidor. A variável *Host atual* refere-se a um servidor candidato a receber uma das máquinas virtuais. Já a variável *VM atual* está relacionada a uma máquina virtual da *Lista de VMs* que está em processo de alocação.

O objetivo do algoritmo é primeiro tentar alocar a *VM atual* em servidores ativos com base na ordem definida pela *Heap de Hosts* (Linhas 4-14). Caso não encontre algum apto a recebê-la, o algoritmo deverá alocá-la no primeiro servidor da *Lista de Hosts Vazios* (Linhas 15-20), ou seja, no servidor inativo que apresenta a maior eficiência energética. Tal comportamento é motivado pela tentativa da Política em procurar, inicialmente, consolidar as máquinas virtuais nos servidores ativos que se encontram na *Heap*. Esta consolidação possibilita a economia de energia, pois todos os servidores que estão

vazios podem ser postos em um modo de economia de energia, por exemplo, em modo de hibernação.

O primeiro passo do algoritmo é verificar se a *Lista de VMs* não está vazia (Linha 2), ou seja, se existem máquinas virtuais a serem alocadas nos servidores. Caso positivo, retira-se a primeira máquina virtual da lista e a armazena em *VM atual* (Linha 3). Feito isso, é verificado se a *Heap* também não está vazia (Linha 4). Caso positivo, o servidor no topo da *Heap* é escolhido e armazenado na variável *Host atual* (Linha 5). Na Linha 6, é verificado se o servidor tem capacidade de receber a máquina virtual em análise. Caso positivo, a máquina virtual é alocada (Linha 7) e a *Heap* atualizada (Linha 9). Caso negativo, retira-se o topo da *Heap* e continua-se a busca por um servidor com capacidade suficiente para alocar a máquina virtual (Linha 12). Esta busca acontece sempre obedecendo a ordem definida pela *Heap*.

Caso a *heap* esteja vazia (Linha 15), o *HeapAllocator* tem de escolher um servidor que ainda não possui nenhuma máquina virtual. Assim, a variável *Host atual* é preenchida com o primeiro servidor com capacidade para receber a máquina virtual (Linha 16), a *VM atual* é alocada no *Host atual* (Linha 17) e o servidor é inserido na *Heap* (Linha 18). O processo continua até que todas as VMs sejam alocadas em um servidor (Saída do algoritmo).

5. Cenário de Simulação

Neste trabalho, foi utilizado o simulador CloudSim [Calheiros et al. 2011] pelo fato de já possuir algumas Políticas de Alocação de VMs implementadas em sua versão nativa, além de sua ampla aceitação pela comunidade acadêmica.

Com o objetivo de aproximar os ambientes simulados a um *data center* real, buscou-se projetar um cenário com servidores e máquinas virtuais com configurações utilizadas em um ambiente real e tarefas com perfis de consumo de recursos físicos equivalentes à realidade. Para este trabalho foi utilizado o cenário proposto em [Beloglazov and Buyya 2012], pois atende ao perfil desejado de cenário e possui um conjunto de Políticas de Alocação que serão usadas na validação dessa proposta.

O *data center* simulado é composto por 800 servidores, sendo metade deles HP ProLiant G4 e a outra metade HP ProLiant G5. Os dois tipos de servidores são constituídos por um processador de dois núcleos, 4 gigabytes de memória RAM, 1 Gbit/s de largura de banda e 1 terabyte de armazenamento secundário. Quanto a capacidade de processamento, o HP ProLiant41 G4 possui 1860 MIPS por núcleo e o HP ProLiant G5 tem 2660 MIPS por núcleo.

A Tabela 1 define o modelo de consumo de energia dos servidores supracitados. O consumo de energia de cada servidor é calculado em função do seu nível de carga de trabalho para atender as máquinas virtuais em um determinado tempo. Os percentuais intermediários são obtidos através da interpolação dos valores piso e teto mais próximos ao percentual em análise.

As configurações das máquinas virtuais também seguem modelos reais comercializados. A empresa Amazon disponibiliza quatro instâncias de máquinas virtuais: Microinstância (*Micro Instance*), Instância Pequena de propósito geral (*Small Instance*), Instância Extra Grande de propósito geral (*Extra Large Instance*) e Instância Média Oti-

mizada para computação (*High-CPU Medium Instance*). Tais instâncias serão retratadas na simulação com as configurações descritas na Tabela 2.

Um componente importante para a simulação de uma tarefa no CloudSim é a Cloudlet. Uma Cloudlet é responsável pelo gerenciamento dos recursos necessários para a execução de uma tarefa entregue à ela. Para o cenário proposto por [Beloglazov and Buyya 2012], foi utilizado o projeto CoMon [Park and Pai 2006] com dados sobre o consumo de CPU de máquinas virtuais em dez dias aleatórios entre Março e Abril de 2011. Cada histórico de consumo da CPU coletado deverá ser entregue a uma Cloudlet, a qual será atribuída a uma única máquina virtual para processamento.

O CloudSim foi configurado para simular cada um dos dez dias. O projeto CoMon colheu os dados de consumo de CPU das máquinas virtuais a cada 5 minutos. Portanto, na simulação as máquinas virtuais terão uma variação no consumo de CPU a cada 5 minutos em todos os 10 dias.

Tabela 1. Padrão do consumo de energia dos servidores

Servidor	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
HP G4 (W)	86	89.4	92.6	96	99.5	102	106	108	112	114	117
HP G5 (W)	93.7	97	101	105	110	116	121	125	129	133	135

Tabela 2. Resumo das configurações das VMs

Recurso	High-CPU Medium	Extra Large	Small	Micro
Número de núcleos	1	1	1	1
Processamento	2500 MIPS	2000 MIPS	1000 MIPS	500 MIPS
Memória Principal	870 MB	1.7 GB	1.7 GB	613 MB
Tamanho da VM	2.5 GB	2.5 GB	2.5 GB	2.5 GB
Lagura de Banda	10 Mbit/s	10 Mbit/s	10 Mbit/s	10 Mbit/s
Número de VMs	$\frac{\text{Nº de Cloudlets}}{4}$			

Visando comparar os resultados obtidos neste trabalho, foram utilizadas as políticas de alocação de máquinas virtuais definidas em [Buyya et al. 2010] e [Beloglazov and Buyya 2012]:

1. ***Interquatile Range* (IQR):** Determina o limiar máximo do estado do servidor com base na diferença entre o primeiro e o terceiro quartil do histórico de consumo de CPU do referido servidor [Jain 1991];
2. ***Local Regression* (LR):** Determina uma linha de tendência com base no histórico de consumo de CPU e procura estimar o valor da próxima observação. O servidor é considerado sobrecarregado se tal estimativa satisfazer uma inequação presente em [Beloglazov and Buyya 2012]

3. ***Local Robust Regression* (LRR):** Semelhante à política anterior, mas os termos da inequação são obtidos através de técnicas um pouco mais robustas. A LRR tem como objetivo mitigar o problema que os outliers trazem para a política LR [Beloglazov and Buyya 2012].
4. ***Median Absolute Deviation* (MAD):** Determina o limite máximo do estado do servidor com base na mediana dos desvios absolutos de cada dado obtido do histórico de consumo de CPU do servidor e da mediana deste conjunto [Jain 1991];
5. ***Static Threshold* (THR):** Define estaticamente o limite superior de utilização para servidores e mantém a utilização total da CPU abaixo desse limiar.

Para realizar a simulação, é necessário a presença de políticas de seleção de VM. Quando um servidor está sobrecarregado, uma política de seleção determina quais entre suas máquinas virtuais devem ser migradas para outro servidor, com o intuito de aliviar a carga de trabalho do servidor em questão. Após uma análise dos resultados apresentados em [Beloglazov and Buyya 2012], a política de Máxima Correlação (MC) foi escolhida para os testes. A política de seleção MC possui como base a ideia proposta por [Verma et al. 2009], onde a VM que possui maior correlação de utilização CPU em relação às outras VMs é selecionada para ser migrada. Tal correlação é obtida com a aplicação da técnica de Regressão Linear Múltipla (RLM) e sua qualidade mensurada através do Coeficiente de Regressão.

Para validar a eficiência da política de alocação *HeapAllocator*, foram adotadas quatro métricas baseadas no trabalho em [Beloglazov and Buyya 2012].

5.1. Energia total consumida

A referida métrica calcula a quantidade de energia consumida pelo *data center* ao final de cada dia simulado. O CloudSim calcula a energia consumida por cada servidor com base na sua utilização de CPU em um determinado instante de tempo, segundo definido na Tabela 1. Ao final de cada dia simulado, o próprio CloudSim soma a quantidade total de energia consumida por cada servidor e obtém o gasto total do *data center* (referente apenas aos servidores e não a toda a infraestrutura de TI).

5.2. Número total de migrações

Esta métrica registra a quantidade de migrações de máquinas virtuais realizadas pelo *data center* ao longo do dia simulado. Dessa forma, a cada migração realizada, o CloudSim incrementa uma variável referente ao número de migrações. Ao final do dia, o conteúdo da variável é apresentado.

5.3. *Performance degradation due to migration* (PDM)

Para cada máquina virtual em processo de migração, a quantidade de CPU atribuída a ela sofre uma degradação de 10% pelo próprio simulador CloudSim. Tal penalidade tem como finalidade simular o tratamento de eventos relacionados ao procedimento de migração da máquina virtual. De acordo com [Beloglazov and Buyya 2012], essa degradação é representada como uma violação de SLA, pois uma parte da quantidade de CPU da máquina virtual não está sendo utilizada por ela. A métrica PDM calcula o percetual médio de MIPS indisponível às máquinas virtuais durante a migração e é definida pela Fórmula 3, onde M é o número de VMs, C_{d_j} é a quantidade de processamento

perdido devido a migração pela VM *j*, e C_{r_j}, a quantidade de processamento exigido durante a migração da VM *j*.

$$PDM = \frac{1}{M}\sum_{j=1}^{M}\frac{C_{d_j}}{C_{r_j}} \tag{3}$$

5.4. *SLA violation time per active host* (SLATAH)

Nos cenários de Computação em Nuvens, as máquinas virtuais podem utilizar somente a quantidade de *hardware* atribuída a elas para executar suas tarefas. No trabalho de [Beloglazov and Buyya 2012], todos os casos onde um servidor sobrecarregado está com a utilização de CPU em 100% são considerados como violação de SLA, uma vez que servidores nesse estado não conseguem oferecer mais unidades de processamento às máquinas virtuais neles hospedadas. A métrica SLATAH calcula o percetual médio do tempo em que os servidores estavam nessa situação (consumindo toda a CPU disponível) enquanto ativos. A métrica SLATAH é definida na Fórmula 4, onde N é o número de servidores, T_{s_i} é a quantidade de tempo onde o servidor *i* consumiu 100% do seu CPU, e T_{a_i}, a quantidade de tempo onde o servidor *i* estava ativo.

$$SLATAH = \frac{1}{N}\sum_{i=1}^{N}\frac{T_{s_i}}{T_{a_i}} \tag{4}$$

6. Resultados

Os resultados foram obtidos através da análise do dados referentes as métricas de consumo total de energia, número de migrações, SLATAH e PDM. Estas métricas foram descritas nas seções anteriores. O resultado de cada política, em cada métrica, foi obtido através da mediana dos valores encontrados nos dez dias de simulação.

Quanto ao consumo de energia, a política *HeapAllocator* (H) alcançou os melhores resultados quando comparados aos resultados das propostas sugeridas por [Buyya et al. 2010] and [Beloglazov and Buyya 2012]. Como pode ser visualizado na Figura 1, a política HeapAllocator (H), proposta nesse trabalho, foi a primeira colocada e apresentou uma economia de aproximadamente 21% em relação a última colocada (a política THR) e 1% em relação as segundas colocadas (as políticas LR e LRR). Esse comportamento se deve a estrutura de Heap proposta pela *HeapAllocator* que ordena os servidores com base na sua eficência energética. Assim, a *HeapAllocator* escolhe sempre os servidores mais eficientes em termos energéticos e isso reflete nos resultados dessa métrica. A tentativa de alocar a maior quantidade de máquinas virtuais por servidor (consolidação) também contribui para este resultado positivo.

Em relação à quantidade de migrações, mais uma vez, a política *HeapAllocator* obteve bons resultados. Como pode ser visto na Figura 2, a *HeapAllocator* foi mais uma vez a primeira colocada e apresentou uma redução de 4% em relação a última colocada (a política THR) e obteve redução de 2% comparada ao da segunda colocada (a política IQR). Tal comportamento se deve à consolidação satisfatória de máquinas virtuais proposta pela *HeapAllocator* que minimiza a necessidade de realizar migrações de máquinas virtuais.

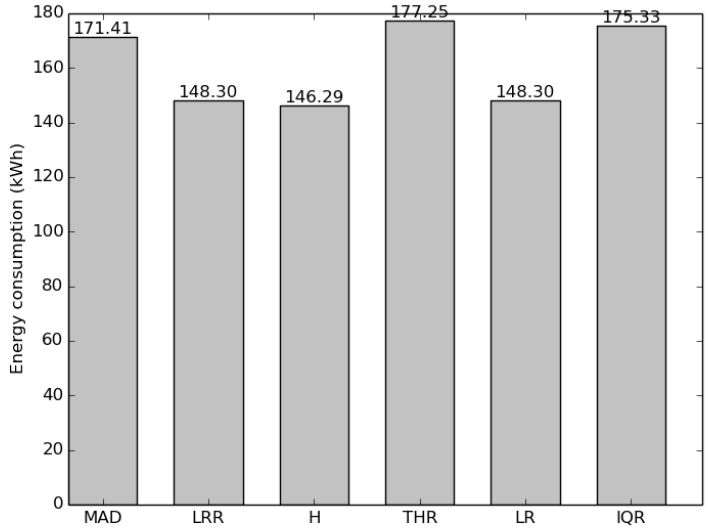

Figura 1. Resultados obtidos com a métrica consumo de energia

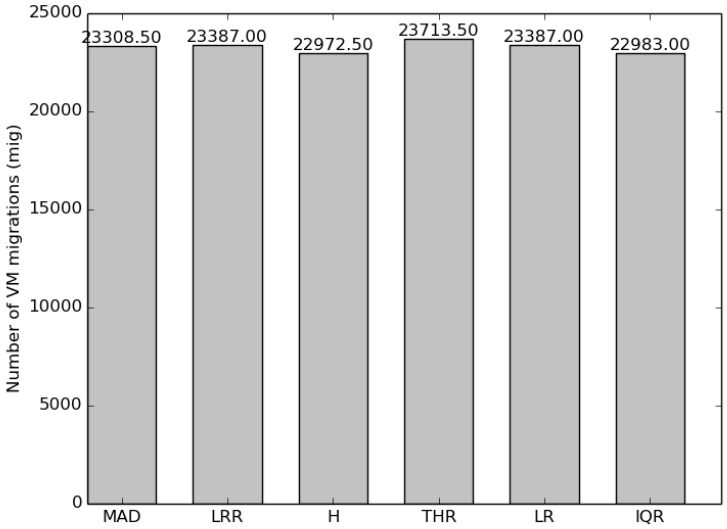

Figura 2. Resultados obtidos com a métrica número de migrações

Os resultados gerados pela métrica PDM foram os mesmos em relação a todas as políticas quando utilizadas em conjunto com a política de seleção MC (Figura 3). A política de seleção MC exige uma quantidade de tempo maior para verificar e retornar se um servidor está sobrecarregado ou não em relação às demais políticas de seleção, o que ocasiona um atraso maior para liberação de recursos que estão sendo utilizados pelas VMs que precisam realizar migração.

Em relação a métrica SLATAH, a política *HeapAllocator* apresentou resultados não satisfatórios em comparação com as demais propostas (Figura 4). A política *HeapAllocator* apresentou um dos piores resultados em relação às outras políticas de alocação. A

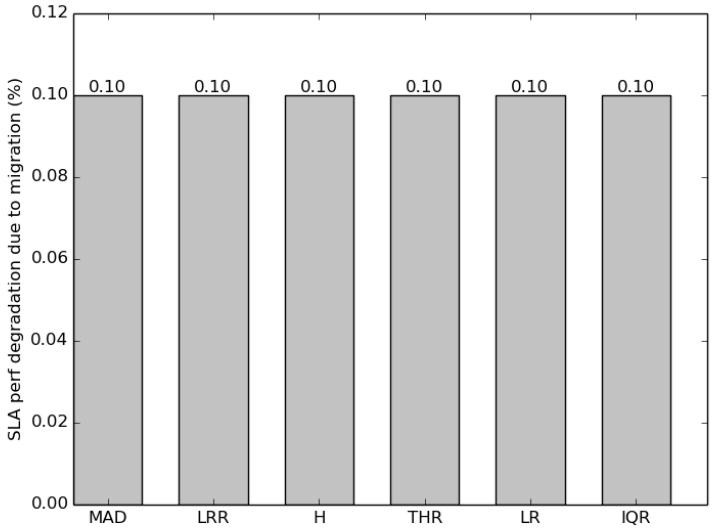

Figura 3. Resultados obtidos com a métrica PDM

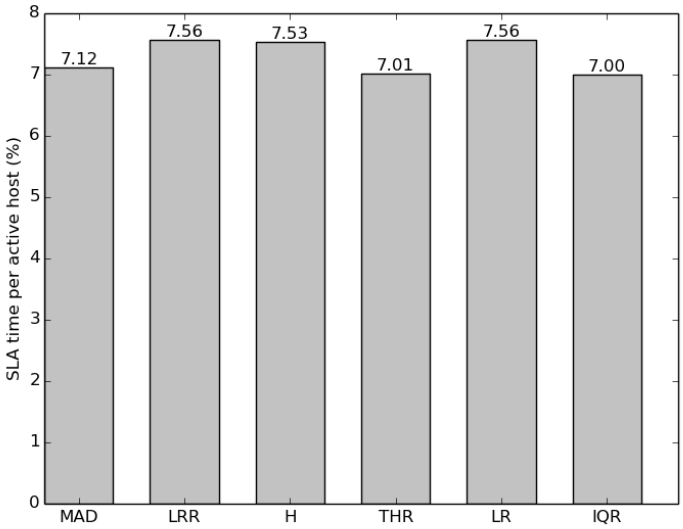

Figura 4. Resultados obtidos com a métrica SLATAH

HeapAllocator apresentou um diferença de aproximadamente 0.5% em relação a primeira colocada (política IQR). Isso é ocasionado pelo longo período necessário para verificação de sobrecarga do servidor, cujo método é importado das políticas de alocação LR e LRR, como explicado na seção anterior. Esse método de verificação de sobrecarga é mais custoso e representa um atraso em políticas relacionadas a regressão linear, deixando o servidor sobrecarregado por mais tempo mais rapidamente até a tomada de decisão, enquanto que a política IQR analisa se um servidor está sobrecarregado mais rapidamente, visto que o método possui complexidade $O(nlogn)$ [Ali and Smith 2006].

7. Conclusão

O principal objetivo deste trabalho foi definir uma política de alocação de máquinas virtuais em ambiente de Computação em Nuvens, denominado *HeapAllocator*, que usa a estrutura de dados *heap* com prioridade para definir o melhor servidor apto a receber uma máquina virtual em processo de alocação ou realocação. O processo de escolha do servidor é realizado visando reduzir o consumo de energia total do *data center*. Esta análise energética representa a contribuição desse trabalho.

A partir dos resultados obtidos, foi constatado que a política *HeapAllocator* alcançou desempenho satisfatório em comparação as outras políticas existentes em relação ao consumo de energia. A *HeapAllocator* conseguiu reduzir o consumo de energia durante a alocação de máquinas virtuais, graças ao mecanismo de tentar, inicialmente, consolidar as máquinas virtuais em poucos servidores e também por buscar usar apenas servidores energeticamente mais eficientes. Os resultados alcançados nas métricas relacionadas ao desempenho (PDM e SLATAH) apresentaram um pequeno acréscimo que indicam um maior número de violações de SLA. A política também reduziu o número total de migrações de máquinas virtuais.

Como trabalhos futuros, é sugerido um estudo para calibrar a fórmula usada na política *HeapAllocator* com o objetivo de reduzir ainda mais o consumo de energia. Além disso, é sugerido um estudo sobre a aplicação da *HeapAllocator* em conjunto com outras políticas de seleção, mas voltadas para a economia de energia também com o objetivo de reduzir ainda mais o consumo de energia. Por fim, é sugerido realizar testes com a política em ambientes reais ou emulados com o intuito de aumentar a credibilidade dos resultados obtidos via simulação.

Referências

Ali, S. and Smith, K. A. (2006). On learning algorithm selection for classification. *Applied Soft Computing*, 6(2):119–138.

Beloglazov, A. (2013). *Energy-efficient management of virtual machines in data centers for cloud computing*. PhD thesis, University of Melbourne, Australia.

Beloglazov, A. and Buyya, R. (2012). Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurr. Comput.:Pract. Exper.*, 24(13).

Buyya, R., Beloglazov, A., and Abawajy, J. H. (2010). Energy-efficient management of data center resources for cloud computing: a vision, architectural elements, and open challenges. *Computing Research Repository*.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R. (2011). Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1).

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Delforge, P. (2014). New study: America's data centers consuming – and wasting – growing amounts of energy. Disponível em: https://www.nrdc.org/experts/pierre-delforge/new-study-americas-data-centers-consuming-and-wasting-growing-amounts-energy. Acesso em: 26-09-2018.

Forbes (2017). Why energy is a big and rapidly growing problem for data centers. Disponível em: https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/3531b0835a30. Acesso em: 21-03-2019.

Goodrich, M. and Tamassia, R. (2013). *Estruturas de Dados e Algoritmos em Java - 4.ed.:*. Bookman Editora.

Jain, R. (1991). *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley.

Lafore, R. (2017). *Data structures and algorithms in Java*. Sams publishing.

Lago, D. G., Madeira, E. R., and Bittencourt, L. F. (2012). Escalonamento com prioridade na alocaçao ciente de energia de máquinas virtuais em nuvens. *XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 508–521.

Nasim, R. and Kassler, A. J. (2017). A robust tabu search heuristic for vm consolidation under demand uncertainty in virtualized datacenters. In *Cluster, Cloud and Grid Computing (CCGRID), 2017 17th IEEE/ACM International Symposium on*, pages 170–180. IEEE.

Nasim, R., Taheri, J., and Kassler, A. J. (2016). Optimizing virtual machine consolidation in virtualized datacenters using resource sensitivity. In *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, pages 168–175. IEEE.

Okada, T. K., Vigliotti, A. P. M. D. L. F., Batista, D. M., and vel Lejbman, A. G. (2015). Consolidation of vms to improve energy efficiency in cloud computing environments. pages 150–158.

Park, K. and Pai, V. S. (2006). Comon: A mostly-scalable monitoring system for planetlab. *SIGOPS Oper. Syst. Rev.*, 40(1).

Rong, H., Zhang, H., Xiao, S., Li, C., and Hu, C. (2016). Optimizing energy consumption for data centers. *Renewable and Sustainable Energy Reviews*, 58:674–691.

Sharma, N. K. and Reddy, G. R. M. (2015). Novel energy efficient virtual machine allocation at data center using genetic algorithm. In *Signal Processing, Communication and Networking (ICSCN), 2015 3rd International Conference on*, pages 1–6. IEEE.

Taheri, J., Zomaya, A. Y., and Kassler, A. (2017). vmbbprofiler: a black-box profiling approach to quantify sensitivity of virtual machines to shared cloud resources. *Computing*, 99(12):1149–1177.

Verma, A., Dasgupta, G., Nayak, T. K., De, P., and Kothari, R. (2009). Server workload analysis for power minimization using consolidation. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, San Diego, California.

Yue, M. (1991). A simple proof of the inequality ffd (l) 11/9 opt (l) + 1, l for the ffd bin-packing algorithm. *Acta Mathematicae Applicatae Sinica*, 7(4):321–331.

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.

Realização

SBC Sociedade Brasileira de Computação

LARC

Organização

.inf INSTITUTO DE INFORMÁTICA UFRGS

Apoio Institucional

RNP 30 anos

Patrocínio

Diamante

CAPES

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico

cgi.br Comitê Gestor da Internet no Brasil

nic.br Núcleo de Informação e Coordenação do Ponto BR

BEDU TECH

Ouro

SAP

Prata

HUAWEI

DATACOM

acm sigcomm

@dyLNET TELECOM

BRDigital Telecomunicações

Bronze

FAPERGS

Google