Jonathan Lincoln Gandhi Andrade Pires Brilhante

Uma Abordagem para Construção de Microsserviços Reativos Baseada em Filas Assíncronas

João Pessoa, Paraíba
Brasil

30 de Agosto, 2018

Jonathan Lincoln Gandhi Andrade Pires Brilhante

Uma Abordagem para Construção de Microsserviços Reativos Baseada em Filas Assíncronas

Proposta de dissertação junto ao Programa de Pós-Graduação em Informática da Universidade Federal da Paraíba como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Universidade Federal da Paraíba – UFPB

Centro de Informática – CI

Programa de Pós-Graduação em Informática – PPGI

Orientador: Rostand Costa

Coorientador: Tiago Maritan

João Pessoa, Paraíba
Brasil

30 de Agosto, 2018

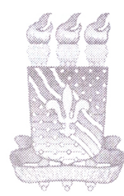
Catalogação na publicação
Seção de Catalogação e Classificação

Brla Brilhante, Jonathan Lincoln Gandhi Andrade Pires.
Uma Abordagem para Construção de Microsserviços Reativos Baseada em Filas Assíncronas / Jonathan Lincoln Gandhi Andrade Pires Brilhante. - João Pessoa, 2018.
96 f. : il.

Orientação: Rostand Costa.
Coorientação: Tiago Maritan.
Dissertação (Mestrado) - UFPB/PPGI.

1. Microsserviços, Programação Reativa, Refatoramento. I. Costa, Rostand. II. Maritan, Tiago. III. Título.

UFPB/BC



UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Ata da Sessão Pública de Defesa de Dissertação de Mestrado de Jonathan Lincoln Gandhi Andrade Pires Brilhante, candidato ao título de Mestre em Informática na Área de Sistemas de Computação, realizada em 30 de agosto de 2018.

1 Aos trinta dias do mês de agosto, do ano de dois mil e dezoito, às quatorze horas e trinta
2 minutos, no Centro de Informática da Universidade Federal da Paraíba, em Mangabeira,
3 reuniram-se os membros da Banca Examinadora constituída para julgar o Trabalho Final do
4 Sr. Jonathan Lincoln Gandhi Andrade Pires Brilhante, vinculado a esta Universidade sob a
5 matrícula nº 2016100065, candidato ao grau de Mestre em Informática, na área de "Sistemas
6 de Computação", na linha de pesquisa "Computação Distribuída", do Programa de Pós-
7 Graduação em Informática, da Universidade Federal da Paraíba. A comissão examinadora
8 foi composta pelos professores: Rostand Edson Oliveira Costa (PPGI-UFPB) Orientador e
9 Presidente da Banca, Tiago Maritan Ugulino de Araújo (PPGI-UFPB), Examinador Interno,
10 Raoni Kulesza (UFPB), Examinador Externo ao Programa, Francisco Vilar Brasileiro (UFCG),
11 Examinador Externo à Instituição. Dando início aos trabalhos, o Presidente da Banca,
12 cumprimentou os presentes, comunicou aos mesmos a finalidade da reunião e passou a
13 palavra ao candidato para que o mesmo fizesse a exposição oral do trabalho de dissertação
14 intitulado "Uma Abordagem para Construção de Microsservicos Reativos Baseada em Filas
15 Assíncronas". Concluída a exposição, o candidato foi arguido pela Banca Examinadora que
16 emitiu o seguinte parecer: "***aprovado***". Do ocorrido, eu, Clauirton de Albuquerque Siebra,
17 Coordenador do Programa de Pós-Graduação em Informática, lavrei a presente ata que vai
18 assinada por mim e pelos membros da banca examinadora. João Pessoa, 30 de agosto de
19 2018.

Prof. Dr. Clauirton de Albuquerque Siebra

Prof. Dr. Rostand Edson Oliveira Costa
Orientador (PPGI-UFPB)

Prof. Dr. Tiago Maritan Ugulino de Araujo
Examinador Interno (PPGI-UFPB)

Prof. Dr. Raoni Kulesza
Examinador Externo ao Programa (UFPB)

Prof. Dr. Francisco Vilar Brasileiro
Examinador Externo ao Programa (UFCG)

Agradecimentos

Primeiramente, tenho a agradecer a todos meus familiares que vêm me apoiando nesta empreitada. Em menções especiais, tenho minhas irmãs, Nadezhda e Christianne Brilhante, que, com respeito e paciência, abraçaram esta jornada ao meu lado; minha amada mãe Severina de Andrade que superou grandes transtornos nestes períodos, sendo minha inspiração e sustento nas horas tortuosas; a meu pai, Edvaldo Brilhante Filho, que cultivou em mim o desejo por aprender e ensinar; e a meu primo, amigo e irmão Thiago Reno, demonstrando com sua resiliência a capacidade para se levantar não importando a queda.

Não menos importante, esta pesquisa não seria possível sem a ajuda dos meus mentores, Rostand Costa e Tiago Maritan. Ambos me mostraram que ensinar vai muito mais além do conhecimento profissional, e que com paciência, confiança, oportunidades abertas e lições de vida reacenderam meu desejo por também vir a lecionar um dia. Por fim, sou grato a todos do Laboratório de Aplicações de Vídeo Digital (LAVID) que trabalham com determinação e excelência para elevar a qualidade da formação profissional na Universidade Federal da Paraíba.

Resumo

Atualmente, vivemos a consolidação da globalização, na qual a grande maioria das corporações atua com serviços fornecidos através da internet. Nestes, não apenas o acesso, mas também o provisionamento de recursos físicos, é feito de forma distribuída (na nuvem, por exemplo). Ainda que lentamente, a atividade de implementação e as estratégias de implantação começam a acompanhar tais tendências. Para alcançar um provisionamento dinâmico e escalável em grandes aplicações, emerge uma nova abordagem chamada de Microsserviços (MS). No entanto, as arquiteturas de MS estão em seu início, sendo ainda mais um conceito do que um padrão de design completamente maduro. Um dos tópicos mais difíceis nesta abordagem está em como migrar ou desenvolver adequadamente um único microsserviço em termos de escopo, eficiência e confiabilidade. Nesse sentido, este trabalho propõe um novo modelo de arquitetura baseado em padrões de alto nível da programação reativa para estruturar internamente um microsserviço. Este novo modelo de microsserviço é coordenado por filas assíncronas, o que permite preservar a compatibilidade com a maioria dos componentes monolíticos e fornecer um processo de encapsulamento para permitir a sua continuidade. Um estudo comparativo entre a abordagem padrão e a arquitetura proposta foi realizado a fim de mensurar os impactos da estratégia proposta em aspectos chave do serviço, como resiliência, elasticidade e desempenho.

Palavras-chaves: microsserviços; programação reativa; refatoramento; filas assíncronas.

Abstract

Nowdays we are living the era of globalization, where the vast majority of corporations work with services provided through the internet. In these, not only the access, but also the provisioning of physical resources, is done in a distributed way (like clouds). Even the form of development teams became spaced out geographically. Although slowly, implementation activity and deployment strategies begin to follow such trends. To achieve scalability and flexibility in larger aplications a new approach arises, named by Microservices (MS). However, MS architectures are at their inception and are even more a concept than a fully mature design pattern. One of the hardest topics in these approach is how to properly migrate or develop a single microservice, in terms of scope, efficiency and dependability. In this sense, this work proposes a new architectural model based on high-level architecture pattern of reactive programming to the internal structure of a new microservice. The new model of microservices are internally coordinated by asynchronous queues, which allowed to preserve compatibility with most monolithic components and provide an encapsulation process to enable its continuity. A comparative study between the standard approach and the proposed architecture was carried out in order to measure the impacts of the proposed strategy on key service aspects: dependability, elasticity and performance.

Key-words: micro services; reactive approach; refactoring; asynchronous queues.

Lista de ilustrações

Figura 1 – Categorias de Nuvens 26
Figura 2 – Nuvens Públicas e Privadas 27
Figura 3 – Comparação da arquitetura tradicional com a virtualizada 28
Figura 4 – Containerização e Virtualização - Comparação das Arquiteturas 30
Figura 5 – Modelo de Interação Síncrona e Assíncrona 32
Figura 6 – Protocolo de Troca de Mensagens - AMQP 33
Figura 7 – Arquitetura em 3 camadas 34
Figura 8 – Princípios da Programação Reativa 42
Figura 9 – Gráfico para Granularidade de Microsserviços 46
Figura 10 – Decomposição da Aplicação "Server Side as a Service- Antes e Depois . 48
Figura 11 – Representação Genérica de Aplicações Monolíticas 54
Figura 12 – Código Síncrono e Assíncrono 55
Figura 13 – Arquitetura de Microsserviço "Monolítico". 56
Figura 14 – Arquitetura de Microsserviço Reativo 57
Figura 15 – Arquitetura da Suíte Vlibras 65
Figura 16 – Microsserviço de Tradução Automática 67
Figura 17 – Microsserviço de Geração de Vídeos 68
Figura 18 – Exemplo de Vídeo Gerado 77
Figura 19 – Erro Médio(%) - Desempenho 81
Figura 20 – Vazão Média - Desempenho 82
Figura 21 – Consumo Médio da CPU - Desempenho 83
Figura 22 – Tempo Médio de Resposta - Elasticidade 84
Figura 23 – Erro Médio (%) - Elasticidade 86
Figura 24 – Vazão - Elasticidade 87
Figura 25 – Vazão Média - Resiliência 88
Figura 26 – Erro Médio (%) - Resiliência 89

Lista de tabelas

Tabela 1 – Comparação entre Principais Orquestradores de Contêineres 41

Tabela 2 – Resumo dos Trabalhos Analisados 50

Tabela 3 – Tradutores Automatizados Português – Libras 64

Tabela 4 – Configurações do Cluster Kubernetes 74

Tabela 5 – Distribuição das Requisições para Testes de Elasticidade 77

Tabela 6 – Distribuição das Requisições para Testes de Desempenho 78

Tabela 7 – Distribuição das Requisições para Testes de Elasticidade 79

Tabela 8 – Injeção de Falhas para Testes de Resiliência 80

Lista de abreviaturas e siglas

AMQP *Advanced Message Queuing Protocol*

API *Application Programming Interface*

CPU *Central Processing Unit*

DDD *Domain-Driven Design*

ESB *Enterprise Service Bus*

HTTP *HyperText Transfer Protocol*

IaaS *Infrastructure as a Service*

LAVID Laboratório de Aplicações de Vídeo Digital

MOM *Message Oriented Middleware*

MP Ministério do Planejamento, Desenvolvimento e Gestão

MS *Microsserviço*

PaaS *Plataform as a Service*

REST *Representational State Transfer*

SaaS *Software as a Service*

SLA *Service Level Agreement*

SOA *Service Oriented Architecture*

SSaaS *Server-Side as a Service*

STI Secretaria de Tecnologia da Informação

TI Tecnologia da Informação

UFPB Universidade Federal da Paraíba

VM *Virtual Machine*

VMM *Virtual Machine Manager*

VMPM Vazão Média por Minuto

Sumário

I Preparação da Pesquisa 19

1 Introdução . . . 21
1.1 Contextualização . . . 21
1.2 Motivação . . . 22
1.3 Organização do Documento . . . 23

2 Fundamentação Teórica . . . 25
2.1 Provisionamento de Serviços Distribuídos . . . 25
2.1.1 Virtualização . . . 28
2.1.1.1 Virtualização com Contêineres . . . 29
2.2 Protocolos de Troca de Mensagem . . . 31
2.3 Arquitetura Monolítica para Desenvolvimento e *Deployment* de Aplicações 34
2.4 Microsserviços . . . 35
2.4.1 Definindo o Escopo de Microsserviços . . . 36
2.4.2 Padrões de Projeto em Microsserviços . . . 37
2.4.3 Desafios para a Adoção de Microsserviços . . . 38
2.4.4 Conteinerização e Microsserviços . . . 40
2.4.5 Programação Reativa . . . 41
2.5 Considerações Finais . . . 43

3 Trabalhos Relacionados . . . 45
3.1 Escopo e Arquitetura de Componentes . . . 45
3.2 Estudos de Casos com Microsserviços . . . 47
3.3 Considerações Finais . . . 50

4 Uma Proposta de Arquitetura Interna para Microsserviços Reativos . . . 53
4.1 Caracterização do Problema . . . 53
4.2 A Abordagem Reativa . . . 55
4.3 Construindo Microsserviços Reativos usando Filas Assíncronas . . . 57
4.3.1 Adaptando *Workers* para um Microsserviço Elástico . . . 57
4.3.2 Resiliência com Redundância . . . 58
4.3.3 Paralelismo e Responsividade com Filas Assíncronas . . . 59

II Estudo Empírico 61

5 Metodologia . . . 63

5.1 Estudo de Caso: Tradutor Automático PORTUGUÊS-LIBRAS da Suíte VLibras 63
5.1.1 Tradutores Automáticos de Línguas de Sinais 63
5.1.1.1 Suíte VLibras 64
5.1.2 Transformando o Tradutor VLibras em um Microsserviço Reativo 66
5.1.2.1 Protótipo do Tradutor 67
5.1.3 Transformando o Gerador de Vídeos VLibras em um Microsserviço Reativo com Armazenamento 68
5.1.3.1 Protótipo do Gerador de Vídeos 69

6 Experimentos e Resultados 73
6.1 Ambiente de Testes 73
6.2 Projeto e Execução dos Experimentos 74
6.3 Cenários dos Experimentos 76
6.3.1 Avaliação da Responsividade 77
6.3.2 Avaliação da Elasticidade 78
6.3.3 Avaliação da Resiliência 79
6.4 Resultados Obtidos 81
6.4.1 Desempenho 81
6.4.2 Elasticidade 84
6.4.3 Resiliência 87
6.5 Considerações Finais 90

7 Conclusão 91

Referências 93

Parte I

Preparação da Pesquisa

1 Introdução

1.1 Contextualização

Desde o surgimento das plataformas dinâmicas para o provisionamento de serviços, a forma como as aplicações são estruturadas mudou drasticamente. Com a evolução do setor de TI de empresas de vendas online, focada na construção de um arcabouço cada vez mais eficiente de troca de dados e distribuição de servidores, surgiu o paradigma de Computação na Nuvem (*Cloud Computing*, o qual eclipsou sua área correlacionada de *Grid Computing* (ARMBRUST et al., 2010) no provisionamento de infraestrutura distribuída. Aproveitando a ideia original dos *grids* no aproveitamento e compartilhamento da capacidade computacional ociosa, a computação na nuvem utilizou-se amplamente do isolamento de recursos físicos através da virtualização para oferta das primeiras arquiteturas orientadas aos serviços (SOA, do inglês *Service Oriented Architectures*) (SCHROTH; JANNER, 2007).

A vantagem competitiva que o planejamento elástico proporciona é notável, permitindo às plataformas distribuídas de alocação dinâmica atenderem desde *startups* até serviços gigantescos, como a ***Netflix*** (MAURO, 2016) e o ***Google Drive*** (MISHRA et al., 2010). Entretanto, para atender à demanda crescente é preciso de um grande investimento em recursos. Ainda assim, novos serviços continuam sendo feitos por parte de novos *players*, assim como por parte das grandes e tradicionais empresas de tecnologia para prover tais serviços, como é o caso da ***Amazon***, ***Google***, ***Oracle***, ***IBM***, ***Microsoft***, o que vem por demonstrar a necessidade do mercado quanto a elasticidade dos serviços.

Contudo, talvez pelo caráter de mudança deste paradigma, esta nova abordagem de provisionamento ainda não dialoga plenamente com a forma tradicional de desenvolvimento e implantação de sistemas mais difundida (DRAGONI et al., 2016). Tem-se por um lado, uma infraestrutura elástica, que pode se adaptar dinamicamente às necessidades do serviço e da aplicação. Já por outro lado, as estratégias de distribuição e, principalmente, as metodologias de desenvolvimentos não evoluíram prontamente e continuam ainda vinculadas aos modelos de provisionamento anteriores, vigentes por décadas.

Gradualmente, alguns mecanismos estão sendo criados para adaptar as soluções desenvolvidas em blocos (ou monolíticas) para uma arquitetura orientada ao serviço. Inicialmente, alguns elementos externos foram adicionados para gerir múltiplas instâncias distribuídas do mesmo serviço, como uma central para mapeamento(*Service Discovery*) (HOSCHEK, 2002) e um serviço para gerenciamento de carga (*Load Balancer*). Na sequência, um conjunto de padrões evoluiu para ampliar o conceito da SOA para uma

abordagem mais alinhada com as necessidades correntes, sobretudo com uma modelagem mais focada em módulos de serviço do que em eventos (PAPAZOGLOU; HEUVEL, 2007).

De uma forma quase natural, os novos conceitos foram se expandindo em uma nova abordagem, a qual finalmente extrapolou os limites de um bloco de serviço, tornando o próprio serviço, também, em uma entidade modularizável e que pode ser estruturada em partes menores. As partes que compõem um serviço foram chamadas de **microsserviços** (***microservice ou MS***) (RICHARDSON, 2014).

Neste sentido, o surgimento de novas abordagens para a construção de aplicações vem trazendo uma série de benefícios com relação ao modelo arquitetural mais tradicional (RICHARDSON,). Juntamente com outras frentes da evolução metodológica e arquitetural, o modelo de microsserviços começa a emergir como uma alternativa relevante, baseada na proposta de transformar um sistema único em um conjunto de pequenos serviços, ou microsserviços, que cooperam e podem ser desenvolvidos, implantados e gerenciados de forma independente (NEWMAN, 2015). Com módulos menores e independentes, torna-se possível que equipes menores possam trabalhar em novos módulos sem a dependência de tecnologias ou da forma de implementação dos outros microsserviços.

Atualmente, começa a se consolidar o consenso de que a arquitetura de microsserviços é uma nova e poderosa abordagem para melhorar a escalabilidade, dependabilidade e manutenibilidade das aplicações computacionais. Uma vez que, ao surgir novas necessidades do negócio, novos microsserviços podem ser desenvolvidos, alterados, substituídos e multiplamente instanciados sem impactar o resto da aplicação. Entretanto, ainda é necessário que o novo paradigma evolua, sobretudo no amadurecimento dos novos conceitos em uma arquitetura mais formal, para que os seus benefícios possam ser mais amplamente difundidos e utilizados.

1.2 Motivação

Na arquitetura de microsserviços, cada MS é organizado para prover uma das necessidades do negócio e podem ser ativados diretamente entre si, de acordo com o *workflow* de cada operação. A independência decorrente deste tipo de organização e comunicação, permite que microsserviços de uma mesma aplicação possam ser codificados em linguagens de programação distintas assim como também utilizar do suporte das plataformas que melhor se adequem a sua natureza específica (FOWLER; LEWIS, 2014).

Entretanto, desenvolver novas aplicações baseadas em microsserviços ou migrar uma aplicação existente para microsserviços não é simples, e muitas questões surgem no momento de decompor as funcionalidades em novos serviços. Dentre elas, podemos citar:

- Qual deve ser o tamanho e escopo ideal de um microsserviço?

- Como transportar um serviço monolítico para uma arquitetura em microsserviços?
- Como estruturar, internamente, um microsserviço?

Neste sentido, a decomposição correta da aplicação é o principal alicerce para uma boa realização da arquitetura de MS. Apenas com uma estruturação adequada dos microsserviços integrantes e uma boa definição das interfaces de comunicação é possível alcançar um baixo índice de acoplamento e um alto grau de coesão, fundamentais para uma boa orquestração e gerenciamento (DANIEL; PERNICI, 2006).

Atualmente, começaram a surgir várias estratégias para evoluir de forma incremental uma aplicação monolítica em uma arquitetura de microsserviços (FRANCESCO, 2017). Além disso, módulos existentes podem ser gradativamente extraídos de uma aplicação monolítica e transformados em microsserviços (STUBBS; MOREIRA; DOOLEY, 2015). Desta forma, a expansão ou decomposição do serviço em bloco pode ser feita sem que seja necessária a sua interrupção.

Entretanto, ainda não há estudos suficientes sobre quais seriam as melhores práticas de projeto para a organização interna de um microsserviço. Neste trabalho, nós investigamos o uso de técnicas de decomposição e reestruturação de aplicações monolíticas na estrutura interna de microsserviços, adotando como diretriz as normas e padrões de serviços reativos (BONÉR et al., 2014). Os resultados obtidos apontam que alguns dos desafios para a adoção da arquitetura de MS, tais como complexidade, curva de aprendizado e desempenho, puderam ser amenizados com a abordagem proposta.

1.3 Organização do Documento

Este trabalho está estruturado em sete capítulos. Neste primeiro capítulo foram introduzidos os conceitos relevantes sobre o tema tratado, a motivação por trás desta linha de pesquisa e a importância do estudo desenvolvido.

No segundo capítulo é apresentada a teoria que fundamenta os conceitos necessários à compreensão desta pesquisa, tais como, o provisionamento de serviços de forma distribuída, aspectos das trocas de mensagens e principalmente o debate entre os modelos de arquitetura de serviços (Monolítico, Microsserviços e Programação Reativa).

O terceiro capítulo sumariza o conhecimento acumulado dos mais recentes trabalhos na área de microsserviços, em especial sua arquitetura.

O quarto capítulo detalha a nossa proposta para projetar/refatorar microsserviços de forma reativa.

O quinto capítulo contém a metodologia utilizada para aplicar a arquitetura reativa de microsserviços, assim como uma primeira experimentação empírica da proposta,

aplicada a um contexto real, e detalha a transformação do módulo de tradução automática de Português Brasileiro para LIBRAS da Suíte VLibras em um microsserviço reativo.

Por sua vez, o sexto capítulo destrincha o projeto, ambiente, cenários e resultados dos experimentos realizados.

Por fim, este documento se encerra com um resumo breve das contribuições desta pesquisa, no capítulo sete, ponderando-se sobre os pontos em aberto e trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo são relatados, de forma sucinta, alguns conceitos necessários ao entendimento geral da pesquisa em pauta. Inicialmente, na Seção 2.1) são discutidos os conceitos sobre o provisionamento de recursos de infraestruturas computacionais. Em seguida, na Seção 2.2, são listadas algumas arquiteturas distribuídas e são apresentados os conceitos básicos sobre a troca de mensagens em sistemas distribuídos. Por fim, as arquiteturas de aplicações monolíticas e baseadas em microsserviço são mostradas nas Seções 2.3 e 2.4, respectivamente.

2.1 Provisionamento de Serviços Distribuídos

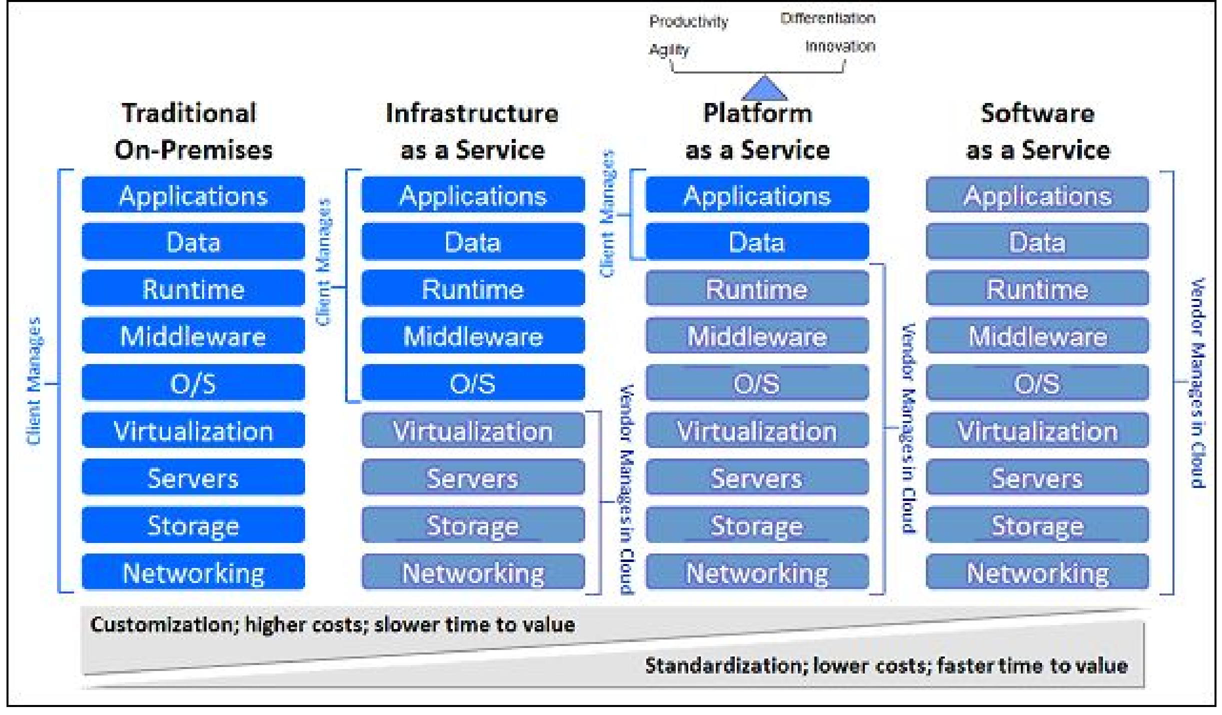
Nos dias atuais, os novos modelos de negócio exigem que as aplicações operem de forma distribuída afim de manter a competitividade. Tal abordagem, apesar de adicionar camadas de complexidade para a implementação, implantação e operação, acaba por agregar um potencial virtualmente ilimitado de recursos para expansão da infraestrutura computacional (DIKAIAKOS et al., 2009). Neste sentido, os grandes centros de processamento de dados privados vêm sendo, cada vez mais, substituídos por provedores públicos de computação em nuvem (do inglês *Cloud Computing*).

O conceito da computação na nuvem possui um significado amplo e abrangente, o qual, pode levar a interpretações distintas, dependendo da experiência e conhecimento de cada um. Apesar de possuir diversas formalizações, a ideia por trás da nuvem está na abstração de um recurso ou serviço, alocado a partir de uma demanda (BUYYA; YEO; VENUGOPAL, 2008). Estes recursos podem ser hardware, software, serviços e até o espaço de armazenamento. Então, em outras palavras, a computação na nuvem trata da disponibilização de recursos computacionais de forma dinâmica, escalável e flexível, um modelo quase intrínseco e imprescindível para as grandes aplicações da atualidade.

Com o novo paradigma houve a grande primeira ruptura entre a forma tradicional de provisionamento terceirizado, o qual se dava normalmente através do aluguel de servidores e comunicação, para uma nova abordagem, escalável e elástica, baseada em Acordos de Nível de Serviço (do inglês *Service Level Agreement - SLA*) (PATEL; RANABAHU; SHETH, 2009). Neste contexto, o isolamento dos recursos tem vital importância para fins de implantação, operação e segurança, enquanto que o balanceamento de carga é crucial para escalabilidade e elasticidade. Para exemplificar, se pensarmos nestes recursos como plataformas de hardware, têm-se na virtualização um modo eficiente para realizar o isolamento e a duplicação.

Há diversos tipos de nuvens computacionais que podem ser caracterizadas pelo tipo de recurso computacional que disponibilizam como serviço (LUO et al., 2011). A Figura 1 ilustra os mais comuns dos tipos de nuvem e compara a pilha de componentes de cada um com o modelo tradicional de TI. Como pode ser visto, à medida que a abstração do modelo cresce, aumenta também a quantidade de componentes gerenciados pelo provedor da nuvem.

Figura 1 – Categorias de Nuvens



Fonte: TELFORD, 2011. [1]

Como listado na Figura 1, as principais categorias de nuvem são: **SaaS** (*Software as a Service*), com o Google Docs como um exemplo de serviço web; **PaaS**, representando as plataformas de *build-in* e provisionamento de aplicações(e.g, AppScale); e **IaaS** (*Infrastructure as a Service*), como por exemplo o serviço da Amazon (AMAZON, 2016) para aluguel de máquinas virtuais (VM), na qual o *deployment* do serviço é feito pelo cliente utilizando as VM como servidores instanciáveis e replicáveis.

Uma outra forma para classificação de uma nuvem trata do seu alcance e tipo de disponibilidade, como ilustra a Figura 2. Uma nuvem é dita privada (também referenciada por interna ou corporativa) quanto esta tem um escopo restrito para acesso, aplicações e clientes, sendo em geral uma nuvem para provisionamento de um conjunto de serviços de uma empresa ou corporação. Já com relação ao modelo de negócio padrão, "pay-as-you-go", existem as nuvens públicas, assim chamadas pelo fato de seus recursos poderem

[1] Disponível em: https://www.ibm.com/blogs/cloud-computing/2011/10/paas-comes-of-age-why-platform-as-a-service-needs-to-be-part-of-your-cloud-strategy-2/. Acessado em: 30-07-2017.

ser alugados por qualquer cliente e utilizados para prover qualquer aplicação (JADEJA; MODI, 2012).

Figura 2 – Nuvens Públicas e Privadas

Fonte: DATAVAULT, 2014.[2]

De forma geral, nuvens privadas tendem a ser menores e necessitam de gerência por parte da empresa, porém permitem um maior controle de segurança e acesso externo para aplicações críticas. Para aumentar a eficácia da utilização dos recursos, nuvens privadas podem ser integradas entre si e/ou juntas a nuvens públicas, para o provisionamento de uma determinada aplicação em larga escala. A este *pool* de recursos de distintas fontes denomina-se malhas dinâmicas, ou nuvem federada (KURZE et al., 2011).

Esta abordagem se mostra mais simples em ambientes sem competição, como nuvens institucionais, mas já são estudadas estratégias para cenários competitivos, como por exemplo a valoração dos recursos emprestados (GOIRI; GUITART; TORRES, 2010). Contudo, toda esta gerência de nuvens adiciona uma camada extra à arquitetura (VILLEGAS et al., 2012), e por consequência uma complexidade maior às aplicações destinadas a utilizar o máximo destes recursos.

Neste trabalho será dado ênfase nas nuvens IaaS, por tratar-se do ambiente em nuvem mais comum e flexível.

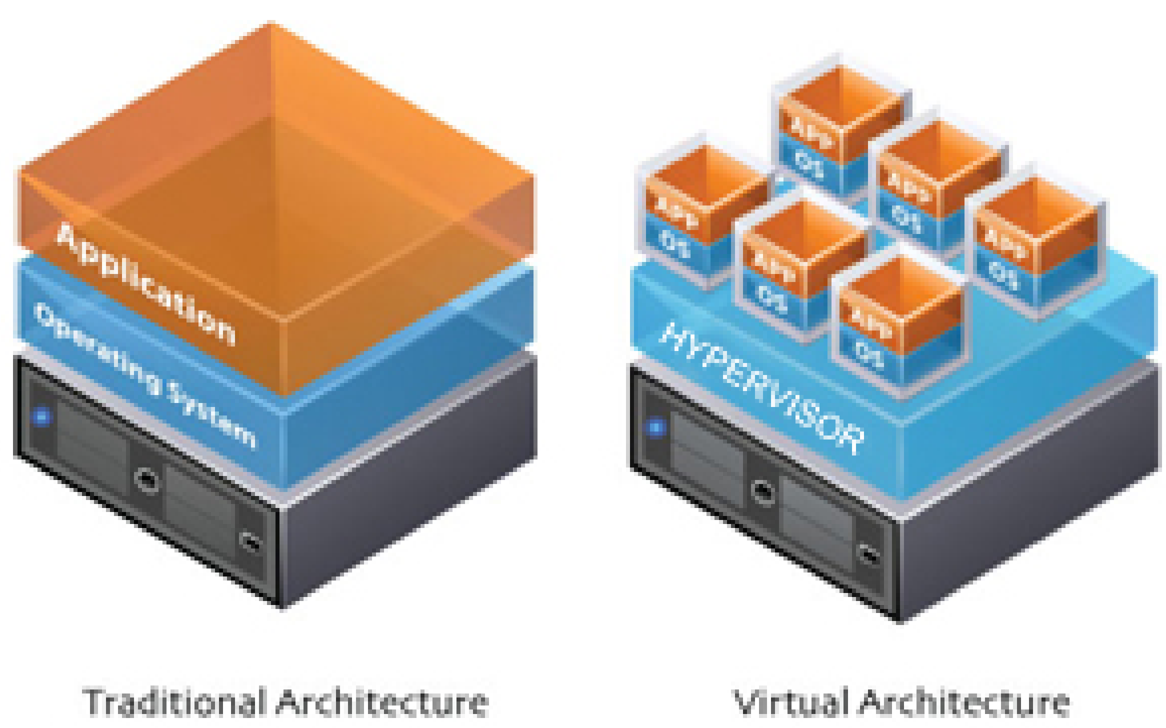
[2] Disponível em: https://thedatavault.com/public-vs-private-cloud/. Acessado em: 30-07-2017

2.1.1 Virtualização

No contexto das nuvens de IaaS, a virtualização trata da técnica de provisionamento na qual recursos de hardware, sistema operacional e dispositivos de entrada e saída são emulados. Popek e Goldenberg em 1974 (POPEK; GOLDBERG, 1974) definem uma máquina virtual como "uma forma eficiente e isolada de duplicar a máquina real".

A Figura 3 representa uma comparação entre um sistema computacional em uma arquitetura padrão e um sistema computacional virtualizado. Como pode ser visto, a grande contribuição da virtualização está no fato de que sobre um mesmo servidor(*host*), é possível executar vários sistemas operacionais de forma isolada.

Figura 3 – Comparação da arquitetura tradicional com a virtualizada



Fonte: Nevian, 2008. [3]

Para garantir a consistência e isolamento de tais ambientes sobre um mesmo hardware, um sistema virtual utiliza um Monitor de Máquinas Virtuais (*Virtual Machine Monitor* - VMM) (UHLIG et al., 2005), *hypervisor* ou programa controlador. De forma simplificada o papel do VMM é atribuir o acesso aos recursos físicos do servidor aos outros sistemas operacionais que agem como clientes, de forma que para estes sistemas apenas um subconjunto dos recursos do *host* seja transparente e acessível.

Três características centrais definem um sistema como um VMM, sendo estes:

[3] Modificado de: http://www.nevian.eu/virtualization. Acessado em: 30-07-2017

- **Fidelidade** : Um software rodando sobre um VMM executa de forma idêntica àquela que teria sobre hardware.

- **Performance** : As instruções do programa executando sobre o sistema virtual devem, em sua maioria, serem executadas diretamente pelo processador sem intermédio de instruções do VMM. Cabe assim ao VMM agir primariamente como um escalonador de recursos, sem grande interferência própria sobre a computação feita.

- **Segurança** : O monitor das máquinas virtuais deve se encarregar de gerenciar os dispositivos de hardware, garantindo assim que nenhum processo da máquina virtual seja acessado indevidamente.

Dentro deste contexto duas classificações de virtualizadores foram propostas : por Hardware e por Software (ADAMS; AGESEN, 2006). Uma virtualização é considerada por Hardware quando a arquitetura em si já provê, de forma pré-definida em hardware, suporte a operações que facilitam a instanciação do VMM (Hardware-assisted), assim como disponibilizam funções para que múltiplos sistemas operacionais executem de forma isolada (por exemplo, KVM (KIVITY et al., 2007)).

Apesar de possuírem uma eficiência superior, por terem parte da lógica implementada em hardware, nem todo sistema computacional permite a virtualização por Hardware, uma vez que requer dispositivos especiais. Ainda assim, existem outras técnicas de virtualização baseadas em simular dispositivos de hardware, incluindo memória e processador, sobre o software. Estes *hypervisors* utilizam de técnicas variadas, mas são classificados como VMM por Software (DESHANE et al., 2008).

2.1.1.1 Virtualização com Contêineres

Nos últimos anos a abordagem de virtualização tradicional vem sendo contestada em seu modo de operar. Por muitas vezes, principalmente em ambientes corporativos e infraestrutura mais complexas, a virtualização de sistemas acaba por gerar uma sobrecarga em termos de eficiência e processamento (SOLTESZ et al., 2007) significantemente considerável para pequenas empresas, podendo colocar em questionamento a utilização destes ambientes de *deployment* contra uma abordagem mais comum de servidores físicos.

Isto pode ser facilmente compreendido em um cenário com múltiplas máquinas virtuais por *host*. Uma vez que o número de VMs providas aumente, maior será o gasto de recursos (armazenamento e processamento) apenas com sistemas operacionais de cada VM, sendo estes muitas vezes iguais ou tendo partes semelhantes. Esta perda torna-se ainda mais visível uma vez que pelo modelo de negócio a maioria destes servidores tem que prover VMs com a mesma configuração, já pré-definidas em contratos.

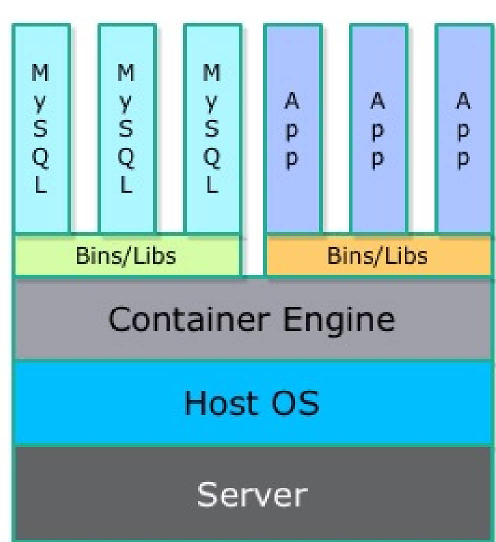
É visível então que existia um grande desperdício de recursos, pela falta de otimização da abordagem tradicional. A conteinerização (ou *Container-Based Virtualization*) emergiu como uma nova solução para tal problema. Nesta abordagem, a virtualização é feita a fim de prover a aplicação um sistema operacional emulado (VAUGHAN-NICHOLS, 2006), em contra ponto à abordagem tradicional que virtualiza um conjunto de recursos de hardware.

Com isto, múltiplos contêineres podem atender sistemas isolados, mas de forma mais eficiente, uma vez que exclui a repetição dos arquivos de sistemas operacionais entre os diversos contêineres (CHE et al., 2010). Por outro lado, a utilização desta abordagem depende de um sistema operacional comunitário, tanto para o provedor quanto para os contêineres. Com esta limitação atualmente apenas soluções para a arquitetura Unix foram desenvolvidas, tendo como exemplo mais comumente citados as soluções "*OpenVZ* e *Solaris Container*" (MATTHEWS et al., 2007).

Figura 4 – Containerização e Virtualização - Comparação das Arquiteturas

Virtual Machines

MySQL	MySQL	App
Bins/ Libs	Bins/ Libs	Bins/ Libs
Guest OS	Guest OS	Guest OS
Hypervisor		
Host OS		
Server		



Fonte: GOMEZ, 2015. [4]

Gomez *et al*, em seu trabalho sumariza a diferença entre as abordagens de virtualização e conteinerização (GOMEZ; LARA; KEBSCHULL, 2015). A Figura 4 ilustra o funcionamento de um conteinerizador genérico comparando-o a um virtualizador tradicional. Como pode ser visto, o isolamento feito em uma camada superior permite ao contêiner compartilhar um ambiente configurado, bibliotecas e o sistema operacional. Desta forma, utiliza-se menos recursos computacionais tanto de armazenamento, quanto

[4] Disponível em: http://iopscience.iop.org/article/10.1088/1742-6596/664/6/062017/pdf. Acessado em 30-07-2017.

processamento para criar, configurar e gerir novas instâncias de sistemas isolados (VM ou contêineres) (SOLTESZ et al., 2007).

Várias pesquisas foram e estão sendo feitas para comparar a performance da solução em contêiner e em máquinas virtuais, sendo esta questão estado da arte na academia. Por um lado, a maioria dos estudos demonstram que a adoção de contêineres promove uma melhora no consumo e eficiência dos recursos, como apontado por Xavier, Miguel (XAVIER et al., 2013). Por outro lado estudos com enfoque no isolamento e segurança provido por ambas as soluções, demonstram métricas inferiores da conteinerização (MATTHEWS et al., 2007).

No que se refere a IaaS privadas e/ou híbridas surge o *Docker* (MERKEL, 2014), uma solução promissora que utiliza de contêineres para prover um ambiente de *build* e *deployment* rápido e comum para aplicações. Adotando um cliente inspirando no padrão de facto AWS, os contêineres Docker dispõem de uma linguagem simplificada em YML (SHANG et al., 2009) para gerar e repassar arquivos de configuração(*DockerFile*), que podem ser instanciados em qualquer ambiente Docker (*docker-machine*). Pela fácil customização e arquivos simples para o provisionamento de serviços distribuídos, a tecnologia de conteinerização *Docker* vem se apresentando como um novo modelo dominante para deployment de serviços de forma distribuída.

2.2 Protocolos de Troca de Mensagem

Em um serviço computacional distribuído é imprescindível a troca de mensagens, uma vez que seus componentes podem estar dispersos fisicamente, independente da natureza do mesmo. A depender da estratégia planejada durante o desenvolvimento da aplicação, assim como da estratégia de deployment, métricas de desempenho, manutendibilidade, escalabilidade e segurança podem ser priorizadas. Desta forma, a estratégia para troca de mensagem é um grande fator para a configuração do deployment do mesmo. De forma geral, as troca de mensagens podem ser feitas de uma maneira direta entre os componentes, ou com a ajuda de um agente intermediário (*middleware*) encarregado da troca de mensagens.

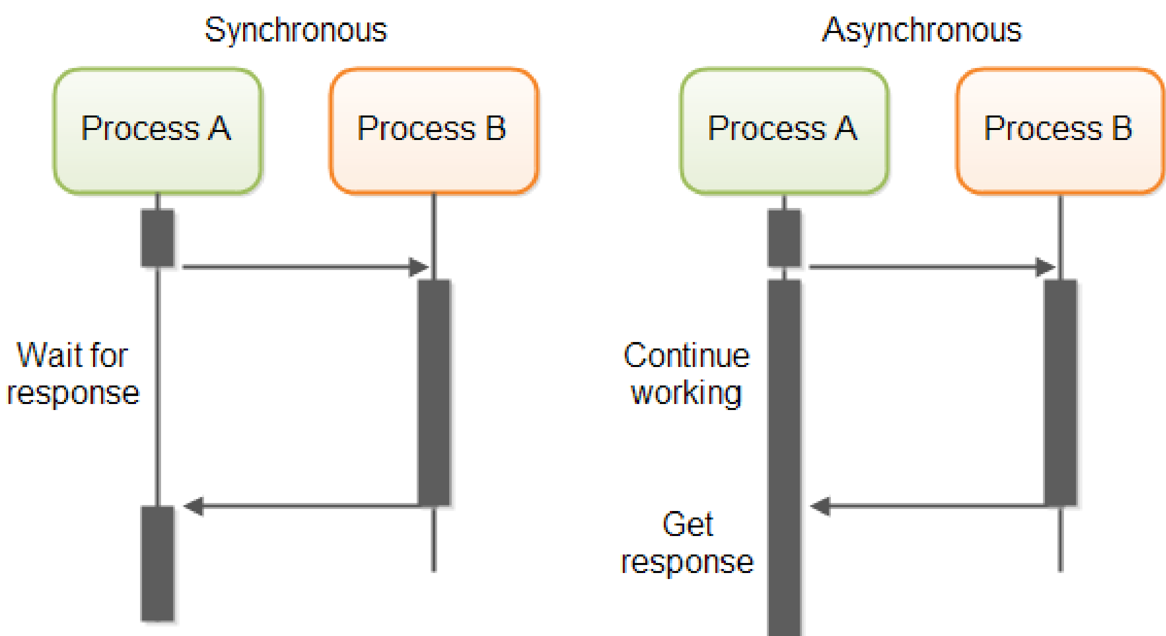
A comunicação direta tem como melhor atratividade a simplicidade. Nesta as trocas de mensagens são feitas diretamente entre os componentes da aplicação. Esta simplicidade reflete em uma melhor curva de aprendizado. Para muitas aplicações, em especial aplicações que adotam estratégias de provisionamento não distribuídas ou de escalonamento por replicação do código inteiro (arquiteturas monolíticas), tal abordagem é a mais comumente utilizada.

Contudo, os benefícios de isolar os componentes são grandes e para isto em geral utiliza-se um *middleware* para suporte a comunicação e chamadas de processo. Em pri-

meira instância, o desacoplamento de tais componentes permite uma liberdade maior de desenvolvimento e deployment, uma vez que separa toda a lógica de segurança e transferência de mensagem em um só serviço. Além disto, um *middleware* facilita a troca de mensagens entre mais de um componente simultaneamente, permitindo uma escalabilidade natural para os componentes da aplicação.

Outra forma de classificar a troca de mensagens trata do tempo envolvido no fluxo de dados: caso exista um pareamento ou bloqueio do fluxo de mensagens, haverá uma comunicação síncrona (e.g,*Corba's Internet Inter-ORB Protocol*, *Java Remote Method Invocation*); Por outro lado denomina-se um canal assíncrono quando a comunicação é feita sem sincronização ponto a ponto, liberando a rotina dos componentes enquanto a mensagem trafega pela rede de serviços (e.g, *Microsoft Message Queuing*, *Java Message Service e SOAP*) (VINOSKI, 2006).

Figura 5 – Modelo de Interação Síncrona e Assíncrona



Fonte: JENKOV, 2014. [5]

A Figura 5 representa o modelo de sequência de uma troca de mensagens síncrona. O sistema requerente (*Caller*) neste tipo de troca de mensagem irá iniciar a comunicação e então entrar em um estado de espera e bloqueio. Como pode ser visto no diagrama, neste tipo de comunicação o sistema requerente não tem nenhum controle ou independência no processamento.

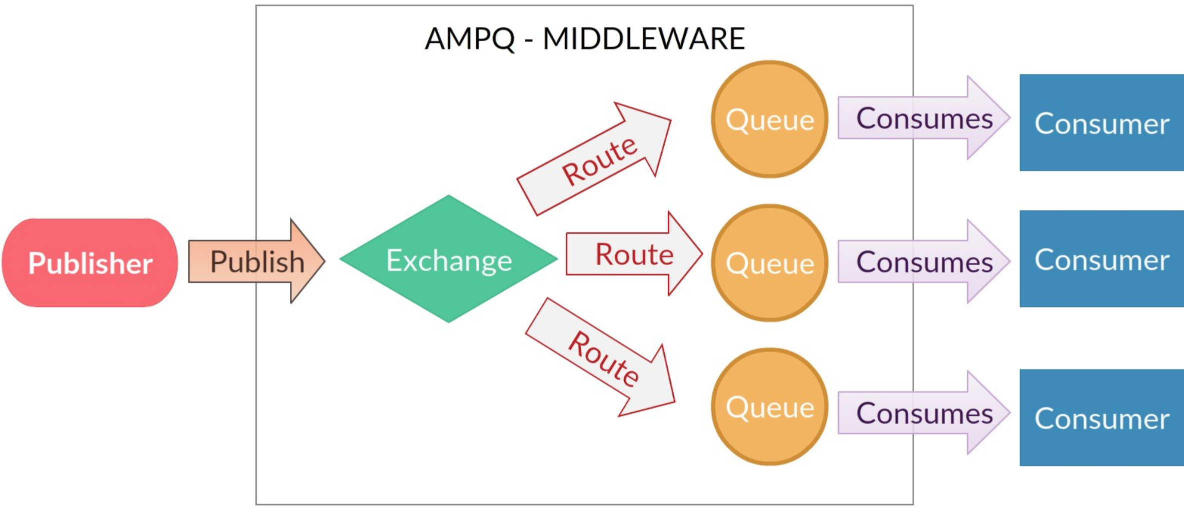
De forma análoga um modelo de sequência é apresentado para troca de mensagens assíncronas. Mesmo que sendo mais complexo em termos de implementação, este modelo

[5] Disponível em: http://tutorials.jenkov.com/software-architecture/index.html. Acessado em: 30-07-2017

permite que todos os participantes retenham a independência de seus processos. Neste modelo, uma vez que a chamada remota foi solicitada pelo sistema requerente, sua execução continua para outros trechos do código de forma independente. Isto possibilita, além de uma eficiência maior, uma melhor tolerância a falha e erros e em geral é feita com o auxílio de um agente intermediário.

Neste contexto, o *Advanced Message Queuing Protocol* (AMQP) surge como um novo protocolo baseado em filas para *message-oriented middleware* (MOM) de forma assíncrona (MARSH et al., 2008). O AMQP então define características de orientação das trocas de mensagens, estratégias de roteamento e os meios aonde são armazenadas e encaminhadas (formato de filas). A definição destes mecanismos em um protocolo permite que sejam analisadas isoladamente propriedades apenas das trocas de mensagens, como a confiabilidade e segurança da entrega e envio de mensagens (ROSTANSKI; GROCHLA; SEMAN, 2014).

Figura 6 – Protocolo de Troca de Mensagens - AMQP



A Figura 6 exemplifica o modelo arquitetural de um *middleware* que implemente o protocolo AMQP. Nestes, o envio de mensagens é feito através de um *publisher*, que por sua vez endereça uma mensagem a um *exchange*.

No *middleware* de troca de mensagens, o *exchange* carrega uma semântica e age como uma interface entre o *Publisher* e as filas (*queue*), redirecionando as mensagens corretamente. Do outro lado, clientes consomem as mensagens de forma assíncrona das filas que escutam (MARSH et al., 2008).

2.3 Arquitetura Monolítica para Desenvolvimento e *Deployment* de Aplicações

Antes de entender o quão significante vem sendo a arquitetura de microsserviços no desenvolvimento e implantação de grandes aplicações, faz-se necessário primeiro entender o modelo anterior. A arquitetura monolítica, recebeu este nome apenas com o advento das primeiras formalizações da proposta de MS. Rouse, Margaret em (ROUSE, 2016) define uma arquitetura monolítica como "O modelo unificado tradicional para design de programas de software, compostos todos em um bloco".

Entende-se então por aplicação monolítica, aquela em que a aplicação é auto-contida, ou seja, independente de outros programas que não estejam embutidos no seu código. É simples visualizar que tal modelo seja antagônico aos ambientes distribuídos de provisionamento, já apresentados anteriormente. De fato, tal modelo tem grandes limitações para este tipo de ambiente.

Figura 7 – Arquitetura em 3 camadas

Para melhor entender tais conceitos, temos na figura 7 representação de uma arquitetura comum em 3 camadas (KAMBALYAL, 2010). Na ponta, está o *client-side* da aplicação, constituído da apresentação para o cliente da lógica de negócio, seja através de um HTML para Web ou até uma aplicação móvel. Responsável por atender e processar as operações dos usuários está a camada do *server-side*. Na aplicação do lado do servidor que irá lidar com as comunicações vindas dos clientes, executar as lógicas e processamentos específicos do domínio da aplicação e repassar o resultado (como por exemplo em

HTML views). Ainda cabe ao servidor também acessar banco de dados e datacenters para atualizar e recuperar informações. Tal arquitetura é considerada monolítica pois todo o *server-side* age como um bloco único nesta estrutura de camadas.

Tal arquitetura permite com que o seja mais simples **implantar** e **escalar** tais aplicações (DRAGONI et al., 2017). Implantar é uma tarefa claramente mais fácil, uma vez que todo o código fora compilado e projetado para executar como um único bloco, fazer tal deployment exige apenas a submissão de um conjunto de arquivos (e.g, WAR) em um diretório ou *servlet* (HUNTER; CRAWFORD, 2001). Por último, a escalabilidade em tais sistemas só é possível através de múltiplas instanciações do mesmo código funcionando através de um *load-balancer* (BOWMAN-AMUAH, 2003).

Porém, por mais que seja simples, tal estratégia está longe de ser otimizada para o contexto de ambientes distribuídos. Um grande empecilho para tal abordagem está no *overload* gerado nos servlets, quanto maior a aplicação for, mais demorada e improdutiva é o tempo para o start-up. Este tempo impacta também testes e desenvolvimento para aplicações grandes e legadas, dificultando a manutenibilidade. Além disto, tal característica impacta pesadamente na implantação contínua. Alterar um componente ou corrigir um problema envolve interromper toda a aplicação e passar pelo overload de start-up novamente.

A escalabilidade de aplicações monolíticas também pode ser um ponto dificultador. Se por um lado é bastante simples apenas criar cópias da aplicação, funciona apenas na escalabilidade em uma dimensão (VAQUERO; RODERO-MERINO; BUYYA, 2011). Em outras palavras, cada cópia carrega todo o *overhead* da aplicação inteira, quando operações específicas podem ser o gargalo. Além de impactar a escalabilidade da própria aplicação, a arquitetura monolítica é um obstáculo para escalabilidade das equipes de desenvolvimento. Idealmente, quanto maior a aplicação mais modular devem ser as equipes de desenvolvimento. Contudo, uma abordagem monolítica não permite esta independência entre os componentes de sua arquitetura, sendo um obstáculo para o processo de desenvolvimento assim como implantação de grandes aplicações.

2.4 Microsserviços

Definimos uma arquitetura em microsserviços como uma metodologia de desenvolvimento de softwares independentes no deployment, pequenos e responsáveis por uma fração autônoma da lógica do negócio, que se comunicam através de um mecanismo leve. Pela natureza independente de suas partes, tal arquitetura é perfeita para projetos com equipes distribuídas (THÖNES, 2015). Atividades de implementação, testes, implantação e até manutenção podem ser feitos de forma isoladas para cada microsserviço e por terem um escopo reduzido tendem a ser mais simples e baratas.

Uma regra bem conhecida no estado da prática deste contexto é a das “duas pizzas” (GUCER; NARAIN et al., 2015). Esta regra define que o microsserviço deva ter o escopo de uma equipe em que apenas “duas pizzas"possam satisfazer (um grupo de não mais que doze pessoas), aonde o microsserviço seja algo pequeno o bastante para que as linhas de código e complexidade possam ser administradas por uma equipe diminuta. Contudo, uma vez que todos estes serviços individuais estejam se comunicando, uma lógica de negócio tão completa quanto uma aplicação monolítica pode ser provisionada.

2.4.1 Definindo o Escopo de Microsserviços

Um dos principais desafios para a refatoração de uma aplicação monolítica está em como extrair corretamente um microsserviço. E parte da dificuldade reside em ter um perfeito entendimento das características que distinguem um microsserviço de um componente de *software*. Basicamente, a principal diferença entre eles está no escopo e na forma de distribuição. Enquanto um componente, pacote ou biblioteca de software tem um enfoque em prover um conjunto isolado de funções para outras entidades internas da aplicação, um microsserviço tem como escopo a operacionalização integral de uma atividade ou rotina de negócio (NEWMAN, 2015).

Neste sentido, a principal propriedade de um microsserviço e que o classifica como tal é a atomicidade da sua funcionalidade. Esta precisa ser única, singular e autônoma, de forma que seja modular e independente de outros elementos externos para produzir as suas saídas e prover suas interfaces (DRAGONI et al., 2016).

Nem sempre, a melhor estratégia para a delimitação de escopo é clara e/ou definitiva. Definir as fronteiras de um microsserviço é algo que depende do projetista, da natureza da aplicação e até mesmo da configuração e capacidades das equipes de desenvolvimento. De forma geral 3 diretrizes podem guiar a definição de um bom microsserviço (THÖNES, 2015):

- **Fraco Acoplamento**: Um bom microsserviço tem dependência mínima com outros serviços. A comunicação com as eventuais dependências necessárias deve ser feita unicamente através de interfaces públicas (API REST, eventos, troca de mensagens) e tais interfaces não devem expor nenhum processo interno.

- **Alta Coesão**: Funções associadas a um mesmo escopo devem ser agrupadas e disponibilizadas em um mesmo microsserviço. Com isto se minimiza a troca de mensagens desnecessárias entre microsserviços.

- **Único Contexto**: Um microsserviço deve cobrir os limites de apenas um contexto. Estes contextos delimitam detalhes internos de domínio e negócio.

2.4.2 Padrões de Projeto em Microsserviços

Assim como ocorre na definição do escopo de um microsserviço, a definição do modelo arquitetural de uma aplicação baseada em microsserviços também traz alguns desafios. As estratégias mais utilizadas atualmente foram inspiradas por *design patterns* (GUPTA, 2015) já adotados em webservices e são apresentadas a seguir:

- ***Aggregator***: Este padrão é o mais simples e intuitivo. Nele, um "agregador" invoca os microsserviços, baseado na operação requerida. Este componente central, age como um controlador do conjunto de microsserviços da aplicação, os quais são independentes entre si. Cabe ao agregador invocar e recuperar os resultados de cada um dos microsserviços envolvidos e por fim disponibilizar o resultado final para o usuário.

- ***Proxy***: Diferentemente do *Aggregator*, no padrão *Proxy* não é feito nenhum processamento no componente que concentra as requisições e ativa os microsserviços. Qualquer lógica de negócio deve ser tratada internamente nos microsserviços. Dessa forma, embora também exista um componente centralizador, o mesmo age apenas como um *proxy*, escolhendo e repassando as requisições para os microsserviços associados. Os microsserviços neste padrão também são independentes um dos outros.

- ***Chained***: Neste padrão, o cliente acessa apenas um microsserviço de entrada A, o qual, para concluir sua tarefa, ativará um segundo microsserviço B, que por sua vez invoca(se necessário) um novo microsserviço C e assim por diante. Essas conexões sequenciais dão a ideia de uma cadeia ou corrente, o que nomeia o padrão. Esta estratégia simplifica a resposta ao cliente a partir de apenas uma requisição feita a um único ponto de entrada. Uma vez que cada microsserviço depende temporalmente da resposta de sua subcadeia, todas as comunicações são feitas de maneira síncrona.

- ***Branch***: Na arquitetura em *Branch*, a requisição pode ser respondida por um ou mais microsserviços, sendo que a escolha depende do contexto e de critérios preestabelecidos. A grande diferença do padrão *Branch* em comparação ao *Aggregator* está no fato de que os microsserviços não precisam ser independentes e únicos. Em outras palavras, microsserviços em cadeia (*Chained*) podem ser acionados pelo componente central para processar uma determinada requisição.

- ***Shared Data***: Este padrão estipula algumas estratégias de compartilhamento de dados persistentes que precisam extrapolar o domínio interno de um microsserviço. De forma geral, a adoção dessa abordagem deve ser evitada para preservar a ideia de atomicidade e independência dos microsserviços. Entretanto, em alguns contextos como a refatoração de aplicações monolíticas, algumas lógicas de negócio quando distribuídas em diferentes microsserviços podem continuar necessitando acessar a

um mesmo banco de dados. Quando adotado, este padrão remete a mesma ideia de microsserviços em cadeia, pela forte dependência que acaba gerando entre eles.

- ***Asynchronous Messaging***: O padrão *Asynchronous Messaging* emerge como uma opção para as arquiteturas RESTful. Nesta arquitetura a comunicação é intermediada por uma fila que repassa as requisições ou chamadas remotas de processos. Desta forma, ao invés de uma cadeia de comunicações bloqueantes entre os microsserviços, passa a existir serviços mais independentes que não têm conhecimento direto um do outro. Tal abordagem facilita a escalabilidade e manutenção dos sistemas, mas em geral adiciona um componente extra para a aplicação: a fila de mensagens (*messaging queue*).

Na prática, um ou mais dos padrões citados podem ser combinados e/ou adaptados para a construção de uma aplicação baseadas em microsserviços.

2.4.3 Desafios para a Adoção de Microsserviços

Já começa a emergir um consenso de que o uso em microsserviços apresenta diversas vantagens na construção de aplicações, sobretudo em termos de escalabilidade, capacidade do reuso, facilidade de implantação, ampliação e extensão, isolamento, baixo acoplamento e etc (ESPOSITO; CASTIGLIONE; CHOO, 2016). Entretanto, como qualquer outra decisão de projeto, deve ter a sua utilização avaliada caso a caso, pois a adoção de serviços granulares autônomos e independentes para a construção de aplicações traz vários benefícios, mas também algumas desvantagens.

Um primeiro diferencial entre a arquitetura monolítica e a baseada em microsserviços está relacionado com um eventual impacto no **desempenho** (FAZIO et al., 2016). Embora não seja uma das principais preocupações quando se está avaliando tais arquiteturas, a questão existe pela própria natureza dos sistemas distribuídos, mas é acentuado quando se adota microsserviços. Uma das razões é que a troca de mensagens por rede é consideravelmente mais lenta e custosa do que chamadas diretas em memória. Deste modo, sistemas com enfoque máximo em desempenho precisam ter a sua construção ou migração para microsserviços avaliada com cautela.

A adoção de microsserviços também pode trazer impactos negativos na **disponibilidade** e na **confiabilidade** das aplicações (DRAGONI et al., 2016). Embora a natureza independente e autônoma dos microsserviços possa levar a crer que disponibilidade do sistema tende a crescer, isto não é sempre verdade. Como o funcionamento do sistema depende de que cada serviço autônomo execute corretamente, a disponibilidade total é um produto do valor individual de cada um destes serviços. Desta forma, o argumento de que serviços menores tendem a acarretar uma densidade menor de erros não é sempre verdade, como demonstrado por (HATTON, 1997) e (COMPTON; WITHROW, 1990). Por outro

lado, a redução do tamanho dos serviços (e, possivelmente, a sua complexidade) pode ter uma relevante influência na redução da propensão a falhas (EMAM et al., 2002). Assim, é fundamental planejar os microsserviços da aplicação considerando, além das estratégias de negócio envolvidas, as métricas de dependabilidade do sistema.

A desvantagem mais reconhecida e, possivelmente, a que mais afasta os programadores e projetistas dos microsserviços é um esperado aumento da **complexidade** do sistema (HASSAN; BAHSOON, 2016). Tal incremento da complexidade pode acontecer tanto no âmbito de *codificação*, *implantação*, *testes* e *logging*.

Além disso, a **programação** de microsserviços exige conhecimentos específicos, como os paradigmas de comunicação inter-processos, e a consequente curva de aprendizado de cada uma das técnicas e ferramentas necessárias. Se em um sistema monolítico a comunicação é feita de forma direta entre os seus componentes, em uma arquitetura de microsserviço é necessária uma camada adicional de coordenação (via orquestração ou coreografia (DANIEL; PERNICI, 2007; DANIEL; PERNICI, 2006), por exemplo) que conduza o fluxo da tarefa pelos diversos microsserviços. Em adição a isto, embora o grande poder de isolamento possa permitir escolhas livres de linguagens e padrões de projeto para cada microsserviço, também pode levar ao surgimento de equipes muito fragmentadas tecnicamente e exigir um esforço maior de integração e supervisão.

Em adição a tudo isto, a **implantação** de uma aplicação não é uma atividade trivial, mesmo no caso de sistemas monolíticos, e a implantação de inúmeros microsserviços tende a ser bem mais complexa (VILLAMIZAR; GARCÉS et al., 2015). Isso torna quase que obrigatório o uso de ferramentas de auxílio e virtualização, como *Docker Engine* (ANDERSON, 2015) e o uso de abordagens de distribuição mais elaboradas e modernas, como *DevOps* (BASS; WEBER; ZHU, 2015).

Embora os **testes** de microsserviços isolados possam ser mais simples e eficazes, garantir o funcionamento do conjunto como um todo adiciona etapas complementares no processo de validação da aplicação. A principal dificuldade disto está em identificar erros e comportamentos indesejados em microsserviços, pois estes normalmente exigem um esforço de análise de vários *logs*, em diferentes cenários e envolvendo múltiplos serviços independentes (ALSHUQAYRAN; ALI; EVANS, 2016).

Com relação à **persistência**, pela lógica de isolamento e autonomia dos microsserviços recomenda-se que os dados pertinentes a cada microsserviço sejam gerenciados internamente, incluindo a lógica de armazenamento e o **logging**. Embora esse seja um ponto positivo, permitindo assim que cada serviço isolado possa utilizar diferentes bancos de dados, linguagens e padrões para o atendimento das necessidades específicas, traz algumas novas preocupações. Dentre elas, está o aumento na complexidade da gerência da camada de persistência do projeto, uma vez que consultas anteriormente feitas de forma direta aos dados precisarão ser intermediadas por outros microsserviços.

2.4.4 Conteinerização e Microsserviços

Para implementar este novo conceito de arquitetura, contêineres vêm sendo utilizados como um paralelo semântico a um microsserviço. Isto se deve principalmente ao fato desta tecnologia ser responsável por reduzir drasticamente o tempo de implantação de aplicações, enquanto ainda garante um grande potencial de escalabilidade e clusterização (KRATZKE, 2015). Pesquisas na área de customização de microsserviços com contêineres vem sendo feitas com ótimos resultados, como a abordagem de *Serfnodes* para clusterização e *Service Discovery* proposta por Stubbs (STUBBS; MOREIRA; DOOLEY, 2015).

De certa forma o surgimento das novas propostas de arquiteturas em microsserviços coincidem com a difusão das principais tecnologias de virtualização por contêineres. Não por acaso, uma vez que estas tecnologias têm como foco prover ambientes facilmente replicados, tornando possível agilizar o processo de implementação, implantação e evolução de serviços. Desta forma, tecnologias de conteinerização revolucionaram o ciclo de vida de uma aplicação (desenvolvimento, implantação e integração), que passaram a ser feitas de forma gradativa e sem a interrupção do sistema, pré-requisito essencial para um bom microsserviço.

Em um área tão recente são poucas as soluções desenvolvidas até o momento no mercado. Em alguns casos tais tecnologias foram desenvolvidas para provedores e plataformas específicas, como é o caso da Cloud Foundry (PAHL, 2015). A Tabela 1 sumariza as principais tecnologias de orquestração para contêineres adotadas hoje em dia (LICHTIGSTEIN, 2017) (TRILOGIX, 2017).

Como demonstrado na Tabela 1, é possível verificar o quão impactante foram as novas tecnologias de *deployment* com contêineres para os principais *cases* de sucesso de arquiteturas utilizando microsserviços, como no caso dos serviços providos pela Netflix e Google. O *framework* Mesos (KAKADIA, 2015), por exemplo, serviu como base para microsserviços desenvolvidos pela Netflix (HARDESTY, 2016). Mas se por um lado a solução Mesos se apresenta dominante em cenários envolvendo *Big Data* com um número gigantesco de contêineres, por outro lado existe a dificuldade de codificar a orquestração dos mesmos. Para aplicações de outra natureza a Docker propôs o Docker Swarm (NAIK, 2016), um *framework* para orquestração de contêineres gerenciado diretamente no gerenciador Docker. Apesar disto, a solução ainda está em evolução, não tendo alcançado todos os *features* necessários para um provisionamento completamente auto escalável. Desta forma, aparece o *framework* Kubernetes (BERNSTEIN, 2014) dominando grande fatia do mercado, expandindo a abordagem BORG da Google (VERMA et al., 2015). Com uma grande comunidade, linguagem simplificada e única para customização do *framework* e *features* mais desenvolvidos para operações de *scalling*, a adoção de Kubernetes com Docker Images vem moldando a forma como aplicações são provisionadas nos dias

Tabela 1 – Comparação entre Principais Orquestradores de Contêineres

Plataforma	Mesos	Docker Swam	Kubernetes
Contribuidores	Mesosphere Microsoft	Docker	Linux Foundation Google
Ano de Lançamento	2009	2015	2014
Instalação do Cluster	Simples para infraestruturas pequenas	Muito Simples	Relativamente Complexa
Deployment	JSON	Nativo ao Docker	YAML
Recursos Mínimos	1 Master e 1 Slave	1 Server	1 Master e 1 Slave
Escalabilidade	Bem desenvolvida, especialmente para grandes clusters	Ainda em desenvolvimento	Bem desenvolvido. Descende da plataforma interna da Google (BORG)
Pontos Fortes	- Escalabilidade para demandas grandes - Ótimo Framework para Bigdata	- De fácil uso e mais próximo ao Docker Engine	- Escalonamento e escalabilidade com mais features. - Contribuições da comunidade -Nativo nos maiores provedores de cloud computing

atuais.

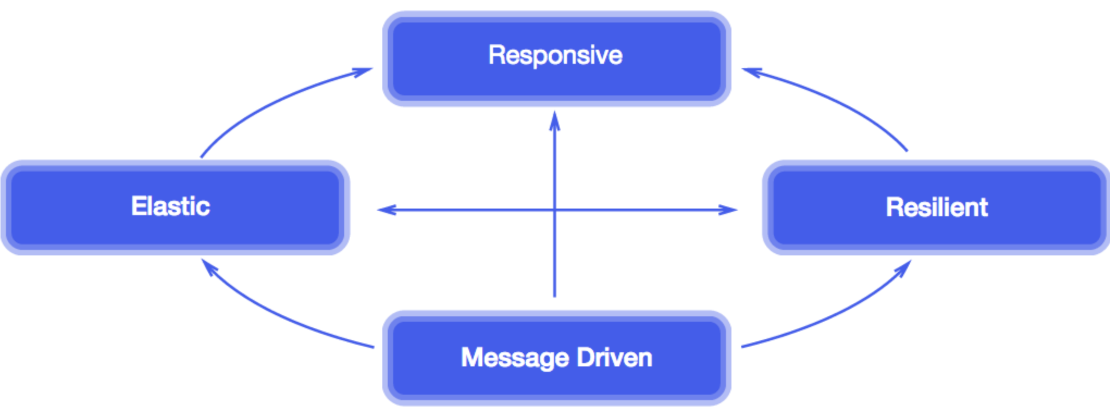
2.4.5 Programação Reativa

Uma corrente que começa a ganhar força na implementação de serviços é o conceito de **reatividade**, conforme descrito no Manifesto Reativo (BONÉR et al., 2014). Nesta vertente, um **serviço reativo**, ou seja, bem adaptado para as demandas correntes, deve priorizar a **responsividade**, a **resiliência**, a **elasticidade** e ser **dirigida a mensagens** (*message-driven*) (SHUKLA, 2014). Estes princípios em sua natureza se complementam, como demonstra a Figura 8. Um sistema mais responsivo, se torna mais elástico e resiliente. O melhor modo de alcançar tal responsividade está em mudar o foco das trocas de mensagens.

- **Responsividade**: A responsividade é o objetivo a ser alcançado com a programação reativa. Por responsividade então, entende-se que o serviço "empurra" (*push*) a resposta para os clientes em tempo real, ao invés do contrário. Mais do que apenas

[6] Disponível em: http://www.reactivemanifesto.org/. Acessado em: 30-07-2017.

Figura 8 – Princípios da Programação Reativa



Fonte: Reactive Manifesto, 2014. [6]

isto, um serviço responsivo deve ser rápido em sua resposta, sob qualquer condição. Deste modo, seja durante uma falha ou em condições de sobrecarga, um sistema responsivo deve garantir uma resposta consistente e ágil ao usuário.

- **Resiliência**: A resiliência é uma diretriz a ser seguida, caso um sistema anseie por ser realmente responsivo. Em geral, todos sistemas estão preparados para funcionar no cenário ideal. Contudo para garantir uma resposta ágil e proativa é preciso ir além. É preciso garantir que uma eventual falha ou um cenário inesperado não impacte na comunicação do servidor ao cliente, uma tarefa nem sempre fácil no contexto de aplicações distribuídas (LI et al., 2008).

- **Elasticidade**: Se a resiliência garante que um serviço irá responder mesmo sobre eventuais falhas, a elasticidade garante que o sistema seja capaz de responder independente do número de clientes. Podemos entender o grau de elasticidade de um sistema como o quão fácil é para que este expanda ou contraia sobre diferentes cargas, afim de garantir a responsividade da forma mais eficiente.

- **Dirigida a mensagens**: Pavimentando as fundações da arquitetura reativa está o princípio de *Message-driven*. Com “dirigida a mensagens", entende-se um sistema que utilize formas assíncronas e mensagens especiais para a troca de dados na arquitetura de cliente/servidor (HANSON, 2000). Em geral, é utilizado por sistemas distribuídos, modelos de troca dirigidos a eventos (*event-driven*) ou atores (*Actor-based*). É através de boas práticas de programação e design, fazendo uso destas técnicas que, é possível alcançar a elasticidade e resiliência necessárias para alcançar a responsividade de uma aplicação reativa.

Assim, a abordagem reativa encaixou perfeitamente com a proposta de microsser-

viços, uma vez que garante a elasticidade e fortalece sua atomicidade. Uma das principais formas para aplicar a abordagem reativa em microsserviços está na ideia de dividir para conquistar (BONER, 2016). Decompor um sistema monolítico em serviços distintos e isolados é a chave para alcançar a arquitetura reativa em microsserviços. Isolar os componentes (e também suas dependências e falhas) é o maior pré-requisito para resiliência, assim como para elasticidade (GUTIERREZ, 2017). Por sua vez, a melhor forma de alcançar tal isolamento é através de uma comunicação assíncrona e desacoplada, unindo perfeitamente a ideia de sistemas reativos a natureza dos microsserviços.

2.5 Considerações Finais

Este capítulo focou na contextualização do estado da arte, além de conceitos básicos na área de provisionamento de sistemas distribuídos, dado devido enfoque à arquitetura dos microsserviços.

A Seção 2.1 elenca os conceitos fundamentais no âmbito do *environment* e *deployment* de serviços distribuídos, tendo sido contextualizado e explorado em especial o modelo de infraestrutura como serviço. Foram discutidos, ainda, aspectos de funcionando do serviço destas arquiteturas nas Subseções 2.1.1,2.1.1.1, aonde foram apresentados alguns aspectos da virtualização e conteinerização, incluindo alguns pontos cruciais para o entendimento das limitações e possibilidades das arquiteturas de serviços distribuídos.

As Seções 2.3 e 2.4, por sua vez, expandiram a temática definindo as arquiteturas distribuídas no espectro do serviço. A Seção 2.3 apresentou a arquitetura vigente nos sistemas atuais, enquanto a Seção 2.4 contrastou a anterior com a abordagem de microsserviços reativos.

Uma vez traçada tal panorâmica, o capítulo seguinte apresentará uma revisão sistemática da literatura ao que diz respeito a soluções e arquiteturas para sistemas. Este compêndio objetiva identificar as principais contribuições da atualidade, assim como suas limitações na área de microsserviços.

3 Trabalhos Relacionados

Nesta seção é apresentado um levantamento geral sobre os estudos relacionados com esta dissertação. As pesquisas relacionadas com a área de interesse desta dissertação abrangem temas sobre microsserviços, migração de arquiteturas monolíticas, arquiteturas orientadas a serviço e à programação reativa. Os mais importantes destes trabalhos são detalhados no decorrer deste capítulo.

A metodologia empregada nesta revisão bibliográfica teve como foco um mapeamento de fontes de divulgação cientifica, a fim de identificar as pesquisas mais significativas da área. Em especial, foram utilizados os bancos de dados da biblioteca virtual do IEEE (IEEE. . . , 2017) e Springer (SPRINGER. . . , 2017). Além destas duas plataformas privadas, foi utilizado o agregador da Google (*Google Academics*). A relevância e o período (desde 2015) foram os critérios para classificação que direcionaram as pesquisas nos engenhos de busca em pauta.

Vale ressaltar que não é o objetivo desta seção apresentar uma revisão sistemática formal do estudo de microsserviços. Aqui tem-se o intuito de introduzir os principais trabalhos científicos feitos no âmbito de como projetar e definir um microsserviço, a migração de aplicações monolíticas, assim como estudos de caso que demonstrem suas conclusões sobre estes respectivos pontos. Caso o leitor deseje obter um conhecimento mais completo e formal sobre as pesquisas produzidas na área, aconselhamos, como ponto de partida, a leitura dos seguintes trabalhos: *"Architecting Microservices"* (FRANCESCO, 2017), *Microservices: A Systematic Mapping Study* (PAHL; JAMSHIDI, 2016), *"A Systematic Mapping Study in Microservice Architecture"* (ALSHUQAYRAN; ALI; EVANS, 2016) e *A Systematic Literature Review on Microservices* (VURAL; KOYUNCU; GUNEY, 2017).

3.1 Escopo e Arquitetura de Componentes

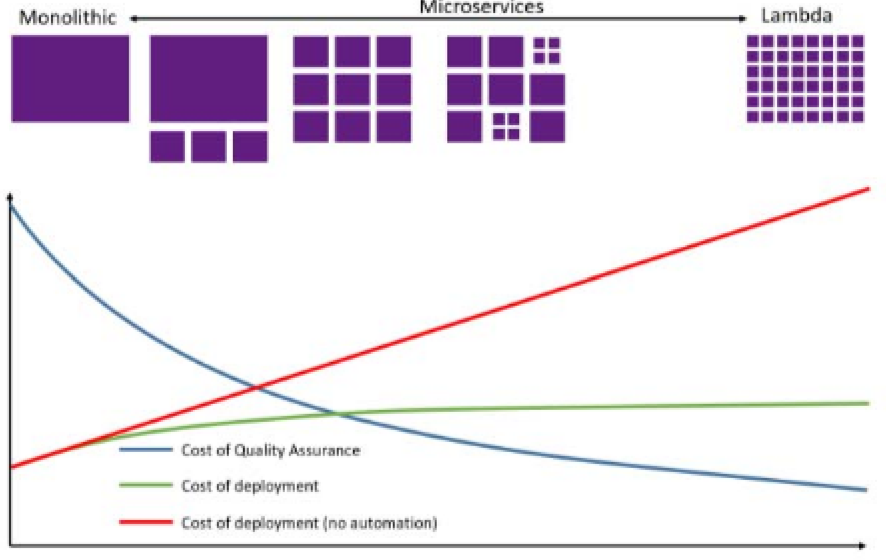
A preocupação em como arquitetar um sistema de microsserviços não surgiu agora. Ainda nos primórdios das ideias sobre microsserviço, Armin Balalaie *et al* (BALALAIE; HEYDARNOORI; JAMSHIDI, 2015a) iniciou a discussão de como deve ser feito um projeto de microsserviço. Seu trabalho apresentou um resumo da área aonde questões foram propostas e respondidas, como por exemplo : "Como decompor uma arquitetura de design orientado por domínio (*Domain-Driven Design*) para microsserviços", "O que fazer com a dependência de código entre módulos"e até "Como e Quando introduzir mecanismo de registro, configuração e descoberta?". Contundo, por seu pioneirismo na área, este trabalho conta apenas com a experiência dos autores para a elaboração de padrões de *design* na migração para microsserviços, carecendo assim de maiores avaliações empíricas

quanto a suas conclusões. Ainda, como em seu próprio trabalho apontava desafios para cada cenário discutido, este acabou servindo mais como um gatilho para novas pesquisas e discussões, do que uma formalização concreta em si.

Gouigoux e Tamzalit (em (GOUIGOUX; TAMZALIT, 2017)) de forma semelhante tentam responder em seu trabalho a problemas comuns na migração para arquiteturas de microsserviços. Como dividir uma aplicação monolítica, como fazer o *deployment* de partes separadas do serviço (microsserviços) e como fazer com que estas partes ajam em conjunto é o foco da pesquisa em questão.

Em seu trabalho, Gouigoux e Tamzalit reportam o conhecimento e estratégias adquiridos no trabalho de refatoramento junto à empresa MGDIS (MGDIS, 2017). A primeira conclusão apresentada pelos autores está quanto à dimensão dos microsserviços. Para eles, a granularidade de um serviço deve ser definida pelo balanço entre o custo necessário para a implantação e dispêndio com a validação dos novos microsserviços. Desse modo, no caso apresentado pelos autores, esta atividade faz parte de um processo difícil de tentativa e erro, aonde quanto maior o custo para validar novos microsserviços ou quanto menor o custo para novos deployments, mais granular deve ser arquitetada a solução, representado no gráfico Figura 9. Ainda assim, nenhuma métrica sobre grandezas mais específicas, como linha de códigos ou módulos incorporados, foi apresentada.

Figura 9 – Gráfico para Granularidade de Microsserviços



Fonte: GOUIX, 2017.

No âmbito do *deployment*, os autores apresentam assertivas mais conclusivas, aonde chegam no entendimento de que microsserviços possuem um forte relacionamento com a tecnologia de conteinerização. Para eles, tais virtualizadores, em específico a implementação do Docker, têm o ferramental necessário para minimizar custos de deployment para esta arquitetura. Para a integração, Gouigoux e Tamzalit relatam que a abordagem de *Enterprise Service Bus* (ESB) é grandiosa demais para aplicações que não sejam de

organizações muito grandes. Contudo, os autores não propõem uma solução definitiva para este aspecto, mas levantam alguns pontos e problemas a serem ponderados.

Afim de melhor organizar e formalizar o estado da prática, Levcovitz LEVCOVITZ; TERRA; VALENTE apresenta em seu trabalho uma primeira abordagem de como decompor aplicações monolíticas para o contexto de microsserviços. Nesta, a aplicação monolítica a ser considerada deve estar dividida no modelo de cliente e servidor. Além disto, presumisse que o servidor seja projetado numa arquitetura de 3 camadas. Satisfeitas tais restrições, Levcovitz formula como deve ser dividida a aplicação em microsserviços.

Para Levcovitz, cada microsserviço derivado desta arquitetura pode ser definido como uma tripla **(F,B,D)** aonde : **F** é um subconjunto das fachadas que compõem a camada de Apresentação, **B** é um subconjunto das funções que realizam a lógica de negócio para toda aplicação e **D** é um subconjunto dos módulos de armazenamento, em geral tabelas de um banco de dados. Toda a problemática, no que diz respeito a decompor um novo microsserviço, estaria então focada em encontrar um subconjunto dos módulos do serviço existente. Neste ponto os autores sugerem um mapeamento das dependências entre os módulos listados em um grafo.

O problema torna-se então em encontrar caminhos neste grafos de uma ponta a outra (da fachada ao banco de dados) e estes caminhos tornam-se as triplas para potenciais microsserviços. Desta forma, uma grande contribuição foi feita por este trabalho no que diz respeito a sistematizar e formalizar a arquitetura que um microsserviço deva ter. Além disto, o trabalho sugere como dividir e alocar os dados já armazenados para que não aja uma quebra na continuidade do serviço. Ainda assim, pouco foi discutido em como poderia ser a melhor forma de organizar os componentes internamente ao serviço.

Partindo para uma abordagem mais concreta, já em 2016 Kecskemet Gabo e et (KECSKEMETI; MAROSI; KERTESZ, 2016) proporam uma metodologia para decomposição de um serviço monolítico baseado em máquinas virtuais. Utilizando do projeto ENTICE, esta metodologia foca em dividir a aplicação em os módulos isolados, através de uma análise feita nas VM(guiadas pelo usuário ou de forma automática) e não por uma questão arquitetural. Ainda que promova uma metodologia prática para a migração de sistemas monolíticos de forma semiautomática e no cunho do provedor da infraestrutura, não foram apontadas as métricas ou estratégias de implementação para essas rotinas. Ademais, não fora apresentado um estudo com testes empírico da metodologia em questão.

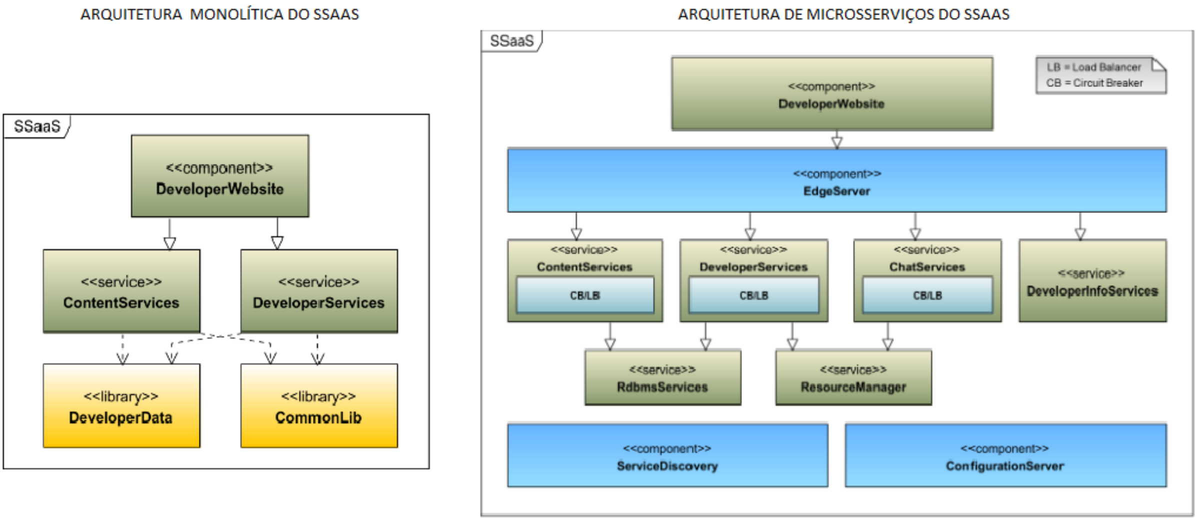
3.2 Estudos de Casos com Microsserviços

Esta seção apresenta os trabalhos relacionados a microsserviços que tiveram como foco o relato empírico de casos de uso na arquitetura de microsserviço. Vale ressaltar que

foram excluídos desta seção exposições de estudos nos quais no processo utilizado, não consta as ideias por trás do planejamento ou decomposição dos microsserviços.

Em 2015, Armin Balalaie (BALALAIE; HEYDARNOORI; JAMSHIDI, 2015b) relata uma das primeiras migrações que o inspirariam a debater sobre como arquitetar microsserviços. A aplicação Server-Side as a Service (SSaaS), iniciada pela Pegah-Tech (PEGAHTECH, 2017) teve o foco desta migração. Esta aplicação tem como objetivo simplificar a atividade de codificação do serviço, para desenvolvedores de aplicações móveis e foi desenvolvido em Java usando o framework Spring, Maven (para dependências e deployment) e Oracle 11g como gerenciador do banco. Motivados pela adição de um novo módulo para permitir a adição de um serviço de chat as aplicações dos usuários, culminou emfim na migração da SSaaS como um todo para os moldes dos microsserviços.

Figura 10 – Decomposição da Aplicação “Server Side as a Service- Antes e Depois



Fonte: BALALAIE, 2015. [1]

A Figura 10, adaptada do trabalho em questão, apresenta uma comparação entre a arquitetura anterior e posterior ao processo de migração. Uma vez que o design em questão já estava dirigido ao domínio, foi escolhido pelos autores que a transposição natural para microsserviços seria também relativa a cada domínio. Sendo assim, os módulos de “DeveloperServices", “ContentServices"e a aplicação html “DeveloperWebsite"tornaram-se microsserviços. Além disto, a biblioteca utilizada para chamadas ao banco se tornou também um serviço nesta arquitetura (“DeveloperData"). Ainda foi necessário que a biblioteca em comum de cada domínio, agora um serviço isolado, fosse incorporado a este.

Por fim, foram adicionados serviços para escalabilidade e consistência, como um *Service Discovery* (com Netflix OSS) e um *Configuration Service*. Apenas então a introdução do novo serviço (*Chat Service*) pode ser feita. Junto a esta, foi realizada a divisão das atribuições de gerência sobre os recursos do serviço de armazenamento, passando

[1] Disponível em: https://arxiv.org/pdf/1507.08217.pdf. Acessado em: 30-07-2017.

agora a serem feitas por uma nova entidade (*“Resource Mannager"*). Em 2016, Armin Balaleie (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016) revisita os mesmos padrões de sua pesquisa anterior, desta vez em uma aplicação de banco de dados como um serviço, Backtory. Esta pesquisa consuma a formalização de alguns *design patterns* para migração de aplicações monolíticas com foco ao provisionamento sem interrupção (*DevOps*).

Estas pesquisas foram importantes por demonstrar um paralelo semântico entre um design com foco no domínio e microsserviços, tornando todo o processo mais intuitivo. Ainda assim a abordagem prática em questão carece de uma metodologia mais apurada. Isto pode ficar claro, por exemplo, quando se tratando da divisão do domínio correspondente a biblioteca “DeveloperData"em duas entidades separadas. Neste ponto, foi necessária uma etapa gradual de desenvolvimento e principalmente da experiência para esta tomada de decisão. De forma geral, este trabalho não demonstra de uma maneira clara como deve ser feita a decomposição de microsserviços para serviços com arquiteturas não dirigidas ao domínio, e mesmo neste contexto, a decomposição de microsserviços dentro de um mesmo domínio acabou sendo mais uma atividade de experimentação.

Também no início das pesquisas sobre microsserviços, Johanson Arne dentre outros (JOHANSON et al., 2016) alcançaram um resultado análogo na aplicação OCEANTEA, um software de suporte às pesquisas oceânicas. Ao contrário do caso anterior, a metodologia escolhida para tratar o armazenamento dos dados foi diferente. Ao invés da customização de um microsserviço para prover operações sobre o banco, os autores decidiram incluir no escopo de cada microsserviço o próprio controle sobre os dados. Esta decisão reflete um impacto e diferença até mesmo em como foi esculpido os outros serviços extras. Infelizmente, nenhuma análise existiu sobre o impacto que tal decisão teve sobre a aplicação como um todo.

De fato, esta abordagem para decomposição de aplicações monolíticas é bastante difundida. Dragoni Nicola (DRAGONI et al., 2016), em seu relato, atesta um método semelhante no processo de migração do *FX Core System*. Este sistema bancário faz o registro, validação e gerenciamento das transações e das contas dos clientes. Para sua decomposição, a arquitetura monolítica já parcialmente baseada em domínios foi dividida como tal. Para tal, a comunicação através das API, tanto externa (com outros bancos) como interna (entre os componentes do serviço), passou a ser gerenciada por um middleware, neste caso o *RabbitMQ*. Por fim o banco de dados passou a se tornar um microsserviço isolado, assim como as bibliotecas compartilhadas. A maior contribuição desta pesquisa está no detalhamento técnico sobre o processo de deployment e reuso de bibliotecas, mas em questões arquiteturais não inova nem responde as assertivas anteriores.

3.3 Considerações Finais

Pesquisas na área de microsserviços têm se tornado muito populares nos últimos 3 anos, motivado principalmente pela crescente demanda do mercado por uma metodologia mais concreta. Contudo, devido à imaturidade da recente área de pesquisa, não há ainda um consenso sobre padrões arquiteturais, sendo até este momento uma linha de pesquisa norteada por experiências empíricas realizadas dentro e fora da academia.

Em conjunto a este fator, as pesquisas sobre microsserviços abrangem os mais variados escopos. É possível encontrar trabalhos com enfoque apenas em aspectos de infraestrutura e até outras motivadas pela de engenharia de software, tendo a organização distribuída e o modo de desenvolver como principal foco dos seus estudos. Nesta seção escolhemos apresentar os trabalhos mais relevantes, tendo em vista os aspectos de modelagem e projeto de microsserviços, com o intuito de mapear as metodologias que discernem sobre como deva ser implementado um microsserviço internamente.

Tabela 2 – Resumo dos Trabalhos Analisados

Aplicação	Primeiro Autor	Ano Publicado	Estratégia de Decomposição/ Migração	Insight sobre Arquitetura Interna de um Microsserviço	Tecnologias Utilizadas
Server-Side as a Server	Armin Balalaie	2015	Domain-Driven Design	Não	Docker e RESTful
MGDIS	Jean Gouigoux	2017	Custo x Validação	Não	Docker
Banco Brasileiro	Levcovitz	2016	Componentes de 3-camadas	Não	Docker e RabbitMQ
ENTICE	Kecskemet Gabo	2016	Síntese de VM	Não	Ferramental Proprio
Backtory	Armin Balalaie	2015	Domain-Driven Design (DevOPs)	Não	Docker, RESTful e Spring
OCEANTEA	Johanson Arne	2016	Domain-Driven Design (com bibliotecas internas)	Não	Docker e RESTful
FX Core System	Dragoni Nicola	2016	Domain-Driven Design	Não	Docker, RabbitMQ e Remote Procedure Calls

A Tabela 2 resume os trabalhos apresentados nesta seção. Como evidenciado, nenhum dos trabalhos teve a preocupação de ilustrar a arquitetura que um microsserviço tem internamente. Nestes, a responsabilidade maior cai sobre como toda a arquitetura é dividida, seja baseada em domínios ou em camadas.

Em outras palavras, não coube aos trabalhos até então avaliar o componente de um microsserviço e como organizar suas funcionalidades advindas do sistema monolítico. Pelos dados compilados também é possível mapear que, de maneira geral, aplicações monolíticas com designs dirigidos aos domínios (DDD) tornam o processo de decomposição mais simples e natural.

Apesar de não haver unidade entre as metodologias utilizadas para planejar a decomposição de microsserviços, é notável que há uma grande relação nas tecnologias utilizadas. Mesmo com diferentes padrões arquiteturais, a tecnologia de conteinerização (em específico a Docker Engine, Swam e Cluster) demonstrou sobressair no que diz respeito aos ambientes de implantação. Além desta, as principais técnicas para comunicação utilizadas foram a abordagem RESTful e de filas assíncronas. Tecnologias mais específicas para a criação de serviços de gestão (Service Discovery com a Netflix OSS por exemplo) também são recorrentes. Assim, é fácil enxergar que apesar de não haver um acordo em termos arquiteturais, ao menos existe boas práticas em termos de implementação no mercado.

Por fim, fica claro que não há uniformidade quanto a que abordagem deva ser tomada. Ainda mais, não há conformidade também em como realizar tais processos de migração mesmo entre abordagens semelhantes. Cabe então ao projetista ficar incumbido de tal atribuição, que em geral usa de suas experiências para o guiar. No mais também pouco é feito na tentativa de formalizar o processo de formação de um microsserviço. Menos esforços ainda foram feitos no quesito de comparação de diversas abordagens e metodologias para tal.

No próximo capítulo, será apresentada uma nova abordagem para projetar um novo microsserviço, tendo como foco principal manter sua arquitetura interna. A justificativa para tal detalhamento é que a solução proposta neste trabalho tem como objetivo melhorar a qualidade do microsserviço a partir de uma ótica da programação reativa.

4 Uma Proposta de Arquitetura Interna para Microsserviços Reativos

4.1 Caracterização do Problema

De maneira geral, um sistema monolítico é dividido em função de suas partes (SILL, 2016). Muito comumente, esta divisão é arquitetada em 3 camadas: **Apresentação**, **Lógica de Negócio** e **Armazenamento** (ver capítulo 2 para maiores detalhes). Nos sistemas baseados em microsserviços, por outro lado, a concepção da arquitetura do serviço normalmente é baseada na experiência pessoal e afinidade das equipes quanto às tecnologias, padrões de projeto (*design patterns*) e o domínio do negócio (*business logic*).

Frequentemente, as aplicações são organizadas em vários subprocessos de negócio. Nem sempre estas rotinas possuem a autonomia funcional necessária para constituir um microsserviço por inteiro, uma vez que possuem dependências e/ou escopos limitados. A principal questão durante o processo de migração para uma arquitetura de microsserviços está em como agrupar tais "microcomponentes". Contudo, ainda há pouco avanço no sentido de padronizar mecanismos para decomposição de arquiteturas monolíticas em microsserviços.

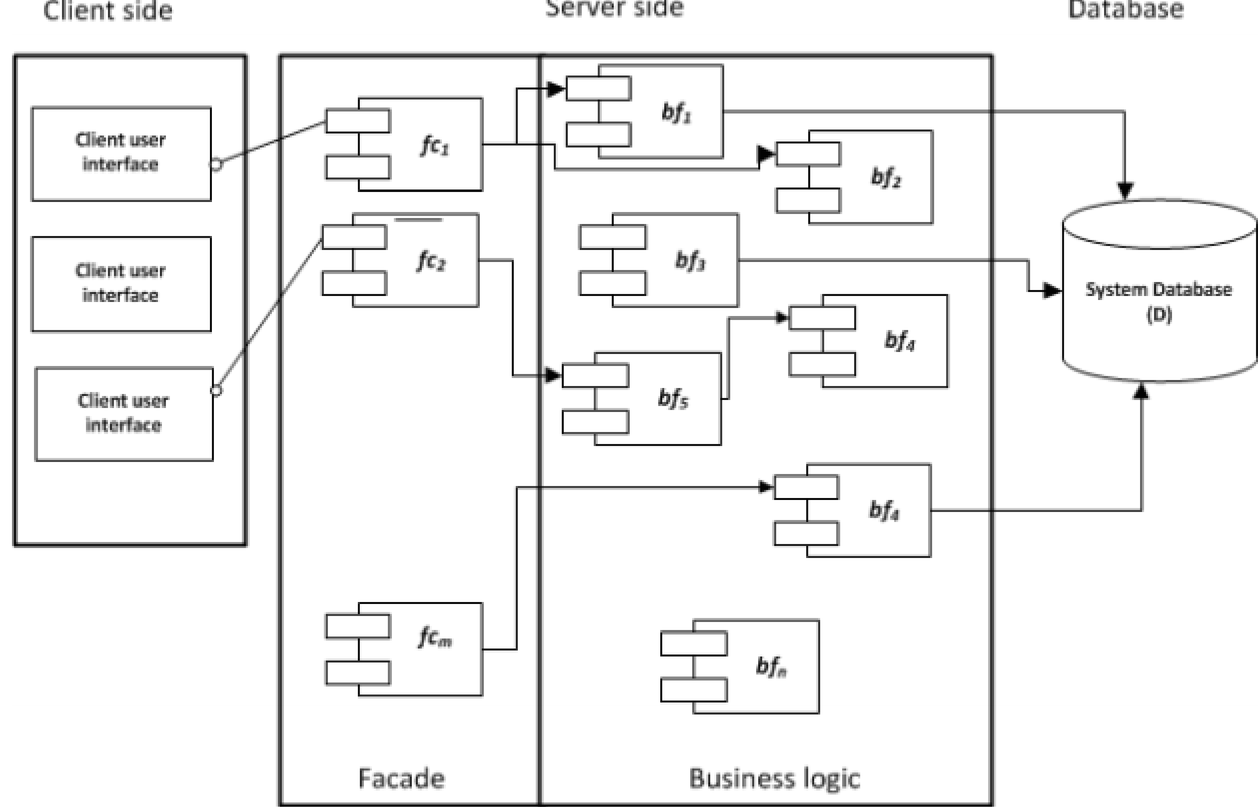
A Figura 11 apresenta o modelo genérico de uma arquitetura monolítica. Cada uma das camadas a seguir pode ser enxergada como um conjunto de funções, componentes ou entidades, mas que são codificadas e compiladas como um único bloco.

Levcovitz (LEVCOVITZ; TERRA; VALENTE, 2016) define uma metodologia para extração de microsserviços de tais sistemas e formaliza um passo a passo baseado nas práticas mais comuns do mercado. De acordo com Levcovitz, cada par de **"fachada de comunicação"** e **"rotina da lógica de negócio"** pode vir a ser um novo microsserviço, desde que haja dependências entre estes.

De forma complementar, um conjunto de objetos de persistência (ex. tabelas do banco de dados) pode ser incorporado a este microsserviço em potencial, assim como outras rotinas de negócio, quando estas forem relacionadas com o recorte da aplicação. Por exemplo, a fachada *"fc1"* e os componentes de negócio *"bf1"* e *"bf2"*(Figura 11) podem formar um subconjunto candidato a refatoração em um novo microsserviço. Da forma semelhante, a fachada *"fc1"* pode vir a ser restruturada junto a apenas um dos componentes de negócio *"bf1"* ou *"bf2"*, neste mesmo exemplo. Decidir qual abordagem seguir torna-se ainda mais complicado considerando as dependências internas de cada

[1] Disponível em: https://arxiv.org/pdf/1605.03175.pdf. Acessado em: 30-07-2017.

Figura 11 – Representação Genérica de Aplicações Monolíticas



Fonte:LEVCOVITZ, 2016.[1]

“microcomponente".

Agrupar estes microcomponentes em serviços isolados e independentes exige, em geral, grande restruturação da aplicação. Em adição a isto, a codificação de uma aplicação com enfoque numa arquitetura de microsserviços é uma tarefa completamente distinta da programação tradicional. É natural então que uma empresa que atua de uma maneira tradicional não esteja ainda capacitada internamente para conduzir alterações no seu paradigma de desenvolvimento, sendo este um dos maiores entraves para a adoção desta arquitetura pelo mercado.

Em geral, o processo para adaptar uma aplicação monolítica ao universo de microsserviços envolve, principalmente, a manutenção pontual e localizada do sistema. Em outras palavras, quando novas funcionalidades ou alterações são codificadas, estas são integradas como novos microsserviços, sem que seja feita uma restruturação completa do bloco monolítico principal.

Mesmo que a longo prazo tal operação possa convergir para uma arquitetura modular, técnicas e análises sobre estratégias de decomposição podem agilizar e simplificar todo este processo. Além disso, um bom projeto deve garantir que os novos microsserviços

sejam eficientes e alinhados com ambientes distribuídos e escaláveis.

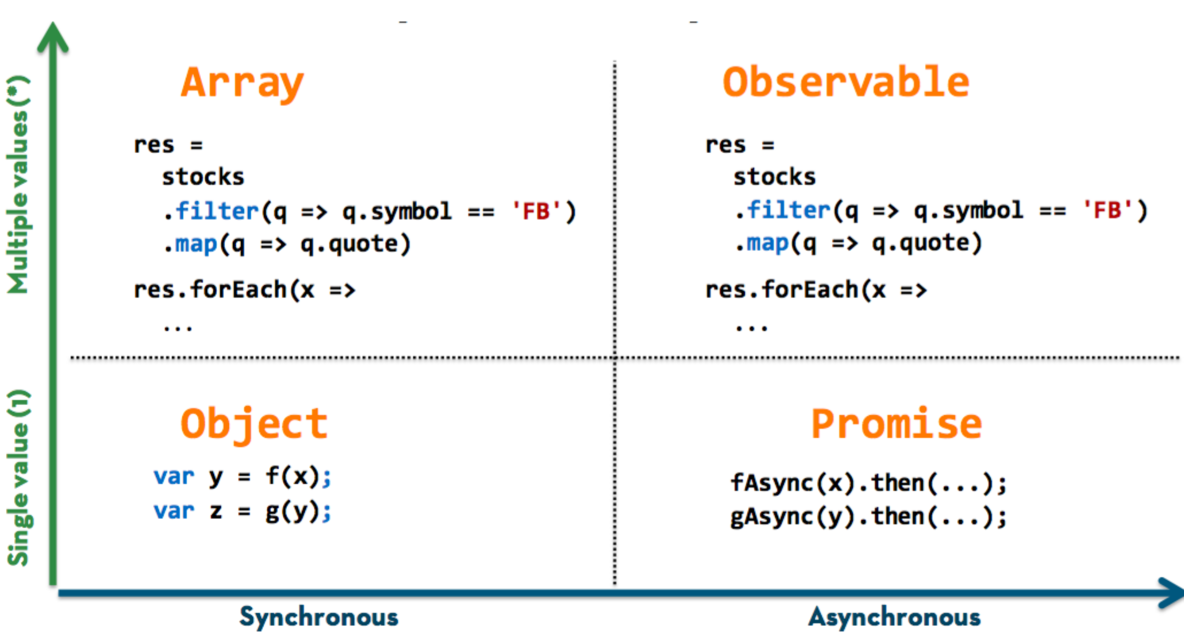
Neste sentido, nem sempre apenas reorganizar os blocos da aplicação monolítica em entidades separadas para configurar um bom microsserviço é suficiente. Caso fosse feito, potencialmente seriam replicados os mesmos problemas de uma arquitetura monolítica, porém, em escala menor. Em tal contexto, surge a seguinte questão: **como agilizar a refatoração de uma aplicação monolítica, e ainda garantir um microsserviço reativo?**

No decorrer deste trabalho será apresentada uma abordagem para a organização destes microcomponentes em novas entidades na arquitetura interna de um microsserviço. Esta abordagem inspira-se nas principais arquiteturas para programação reativa aplicada ao projeto microsserviços de maneira ágil, robusta e escalável, com o intuito de maximizar o reuso do código monolítico.

4.2 A Abordagem Reativa

A Figura 12 ilustra as diferenças entre códigos feitos para uma aplicação monolítica e para sistemas reativos e escaláveis. Em uma aplicação monolítica, as entidades são manipuladas diretamente em suas variáveis, há chamadas diretas de função e suas estruturas de dados são apontadores destes valores. Já no contexto de serviços reativos, as funções e os objetos são substituídos por “promessas"e chamadas assíncronas. Desta forma, o fluxo de dados é estruturado em “correntes"(do inglês, *stream*).

Figura 12 – Código Síncrono e Assíncrono

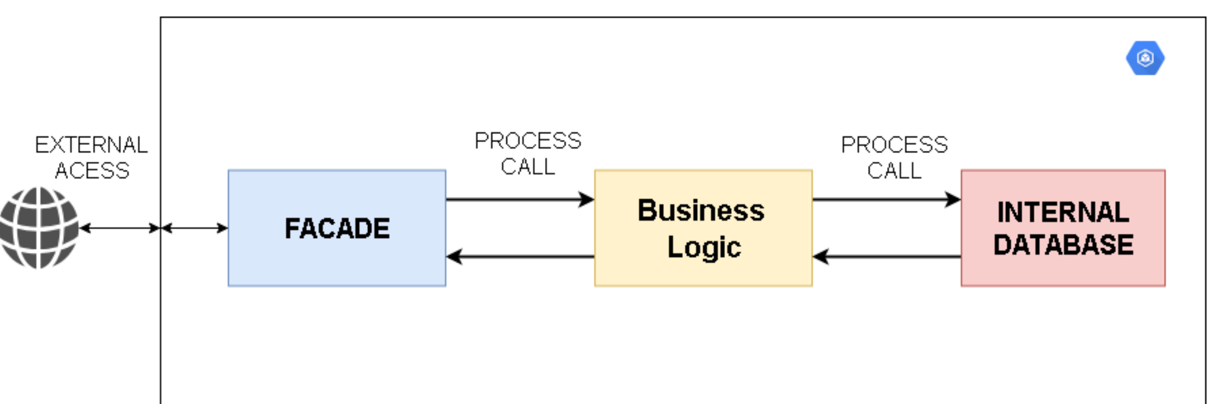


Fonte: ALVAREZZ, 2016. [2]

Como apresentado anteriormente, um microsserviço é composto internamente por entidades (microcomponentes). Temos como premissa que tais microcomponentes também possam ser organizados sob uma ótica de coreografia e/ou orquestração semelhante a da arquitetura global da aplicação.

A Figura 13 apresenta a decomposição de um sistema monolítico genérico em microsserviços com base em uma técnica arbitrária de fatoração.

Figura 13 – Arquitetura de Microsserviço "Monolítico".



Em linhas gerais, a forma mais intuitiva e simples para a concepção deste microsserviço é através de uma analogia direta com a adotada para a construção de uma aplicação monolítica. Neste sentido, toda a funcionalidade interna do microsserviço seria provida através de uma interface (*facade*) que encapsula e distribui uma lógica de negócio (*business logic*) e faz acesso a uma serie de dados e tabelas (*internal database*) através de comunicação direta entre cada componente.

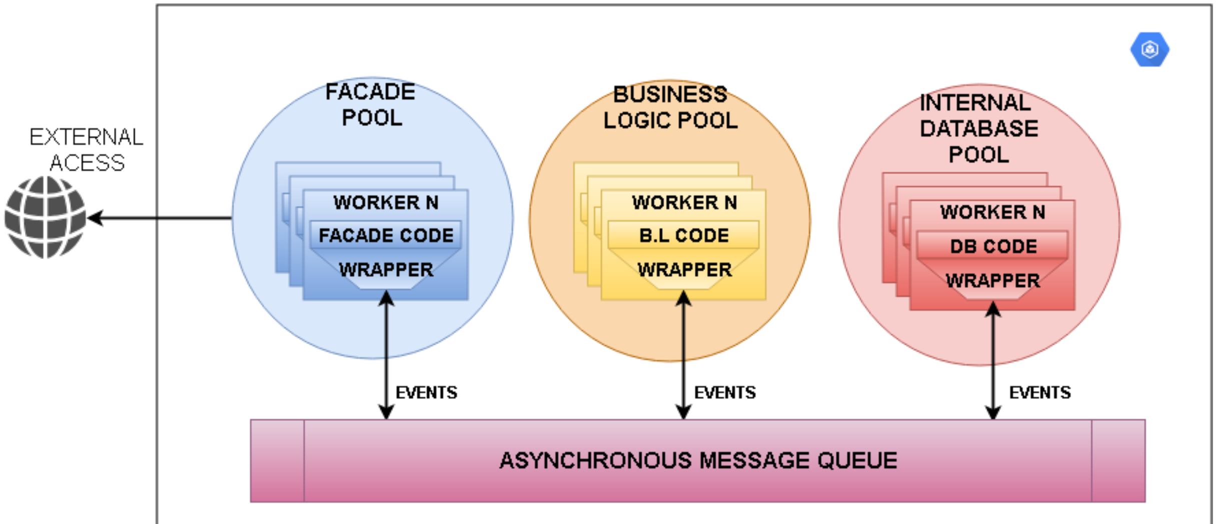
Para garantir as propriedades de um sistema reativo no contexto isolado de cada microsserviço que compõe a aplicação, esta abordagem utiliza de uma coreografia interna com filas assíncronas, para desacoplar as ligações diretas de cada microcomponente do MS. Como apresentada na Figura 14, um *middleware* orientado a mensagem (MOM) (CURRY, 2004) é adicionado para servir de canal de comunicação, do qual baseia-se as operações de mensagens, filas e rótulos do protocolo AMQP (VIDELA; WILLIAMS, 2012).

Uma vez que toda troca de mensagem passa por um canal separado, os microcomponentes do MS passam a agir como **workers**, que utilizam blocos de código padronizados ("wrappers") que encapsulam e facilitam a utilização das filas[3].

[2] Disponível em: http://jonalvarezz.github.io/presentation-reactive-programming/. Acessado em: 30-07-2017

[3] De forma semelhante, outras estratégias de troca de mensagens (como a coreografia de fluxo de eventos) podem ser adotadas para tal contexto, mas dependeriam de uma estruturação arquitetural mais elaborada e de maior adaptação do código já existente no caso de uma refatoração.

Figura 14 – Arquitetura de Microsserviço Reativo



4.3 Construindo Microsserviços Reativos usando Filas Assíncronas

Tendo traçado como norte os pilares da programação reativa, tem-se como principais objetivos a serem alcançados com o uso desta nova abordagem a **comunicação orientada a eventos**, **resiliência**, **responsividade** e **elasticidade** (já apresentados no Capítulo 2).

Esta arquitetura permite que um microsserviço tenha a unidade a mais para expansão em termos de **escalabilidade horizontal** (NAMIOT; SNEPS-SNEPPE, 2014). Em outras palavras, a escalabilidade do sistema pode ser feita também dentro do escopo interno de cada microsserviço, dividindo a demanda em um conjunto elástico de *workers*. Combinada com a replicação e balanceamento de carga dos microsserviços como um todo através de VMs, esta abordagem permite que gargalos específicos do sistema possam ser tratados em uma granularidade menor que a de um microsserviço.

A seguir, será explanado como a orquestração interna apresentada anteriormente impacta em cada um destes pontos.

4.3.1 Adaptando *Workers* para um Microsserviço Elástico

Tecnologias de balanceamento elástico de carga (*Elastic Load Balancers* ou EBL) permitem a um serviço gerenciar de forma dinâmica para qual(is) servidor(es) será direcionada a requisição do cliente, o que dependerá do seu histórico, localização e das condições de utilização da infraestrutura, dentre outros fatores. Desta forma, é possível que uma requisição seja atendida por uma instância do serviço em um dada ativação e, em outro momento, por outra instância espelho em um *host* completamente distinto, o qual pode, até mesmo, ser alocado em tempo de execução.

A arquitetura de um serviço em várias unidades por si só já evolui o potencial de elasticidade da aplicação. Uma vez que com a utilização do serviço como um bloco monolítico, todo o código deve ser espelhado para a instanciação de outro servidor. Neste caso, pode ocorrer desperdício de recursos, já que, em geral, o acesso às funcionalidades do sistema podêm ter picos distintos. Tendo adotado uma arquitetura reativa é possível dotar um sistema de **escalabilidade horizontal** (NAMIOT; SNEPS-SNEPPE, 2014), ou seja, permitindo aumentar a quantidade de recursos para cada funcionalidade de forma isolada. Todavia, diferente da escalabilidade horizontal tradicional, que é feita através da replicação de recursos computacionais, os microsserviços reativos podem também escalonar a quantidade de Workers internos a estes. A esta nova granularidade para a elasticidade de um serviço, denominamos de **escalabilidade horizontal interna**.

A proposta aqui apresentada expande esta ideia. A fatoração de microsserviço tradicional, normalmente, também segue as ideias do design monolítico e, com isto, tem a sua escalabilidade associada com o espelhamento de instâncias inteiras. A nova arquitetura aqui proposta permite que o microsserviço tenha a possibilidade de uma expansão ainda mais granular em termos de elasticidade. Em outras palavras, a escalabilidade do sistema pode ser configurada também dentro do escopo interno de cada microsserviço, dividindo a demanda interna em conjuntos elásticos de *workers*, cada conjunto é responsável por uma funcionalidade interna relevante do microsserviço reativo.

Combinada com a replicação e balanceamento de carga dos microsserviços como um todo, esta abordagem permite que gargalos específicos do sistema possam ser tratados em uma granularidade menor que a de um microsserviço. Por exemplo, se um microsserviço hipotético para gestão de contas de usuários está sendo atipicamente ativado por solicitações de novos cadastros, o número de *workers* relacionados com esta operação pode ser ampliado temporariamente. Em caso de contenção de recursos, um ajuste fino pode envolver a diminuição do tamanho do pool de *workers* responsáveis pelo cancelamento de contas, por exemplo. Com a possibilidade de serem melhor adaptados para cada contexto e cenário, até mesmo a replicação tradicional do microsservico reativo pode ser feita de forma mais otimizada e eficiente.

4.3.2 Resiliência com Redundância

Associada com a ideia de elasticidade, a aplicação da abordagem reativa na arquitetura de microsserviços também pode trazer incrementos para a resiliência do serviço como um todo. Aplicando na resiliência a mesma analogia feita para o aumento da granularidade de elasticidade do sistema, a arquitetura proposta também ajuda a evoluir a abordagem monolítica neste aspecto. Neste sentido, como os *workers* estão vinculados a uma funcionalidade específica do MS, falhas em tais componentes tendem a ter um impacto menor no funcionamento geral da aplicação, além de permitirem uma maior ca-

pacidade de recuperação. Já, para uma aplicação monolítica, uma falha desta natureza pode significar a quebra completa do sistema.

Assim, para permitir que os requisitos de resiliência sejam atendidos é preciso que um microsserviço possa ter mais de um componente capacitado para responder as requisições do usuário e que o chaveamento para qual componente deve atender cada operação pendente possa ser feita de maneira dinâmica, até mesmo sobrepondo uma alocação anterior.

Neste sentido, a arquitetura interna de microsserviços reativos também expande esta premissa. Isto se dá pelo fato que os diversos conjuntos de *workers* podem continuar operando mesmo que um destes *workers* falhe. Uma vez restaurado ou reiniciado, o *worker* pode ser integrado novamente ao *pool* e retornar ao fluxo de execução com facilidade, retomando o atendimento aos pedidos pendentes nas suas respectivas filas de entrada. Enquanto isso, a operação que estava com ele quando ocorreu a falha, contina na fila e será liberada para atendimento por outro *worker* após decorrer um timeout pré-estabelecido para a sua conclusão. Vale salientar que para estes ganhos na tolerância a falha do MS, é esperado uma independência entre os erros destes componentes internos.

4.3.3 Paralelismo e Responsividade com Filas Assíncronas

A responsividade e a resiliência são pontos chave da programação reativa. Alcançar esta característica em uma aplicação está intrinsecamente ligada com o potencial de elasticidade do sistema e o tipo de comunicação adotada (vide Capítulo 2). Um sistema resiliente, tende a ter um baixo tempo de resposta, unindo em um ciclo as diretrizes da programação reativa.

Como reflexo da escalabilidade horizontal interna apresentada anteriormente, uma das hipóteses deste trabalho é que ocorra um incremento na performance do microsserviço como um todo quando executado em ambientes *multithread*. Se, por um lado, as trocas de mensagens diretas possam ser ligeiramente mais rápidas do que as feitas por intermédio de filas assíncronas, uma abordagem de execução concorrente e independente traz ganhos na utilização de múltiplas CPUs.

Parte II

Estudo Empírico

5 Metodologia

5.1 Estudo de Caso: Tradutor Automático PORTUGUÊS-LIBRAS da Suíte VLibras

Nesta seção, é descrita uma atividade de decomposição realizada com abordagem proposta, a qual foi feita sobre um serviço de tradução automática de Português Brasileiro para a Língua Brasileira de Sinais (LIBRAS), chamado **Suíte VLibras** (PESSOA et al., 2015).

A **Suíte VLIBRAS** é o resultado de uma parceria entre o Ministério de Planejamento, Desenvolvimento e Gestão (MP), através da Secretaria de Tecnologia da Informação (STI), e a Universidade Federal da Paraíba (UFPB), junto ao Laboratório de Aplicações de Vídeo Digital (LAVID), e consiste de um conjunto de ferramentas para tradução automática de Português Brasileiro (texto, áudio e vídeo) para a Linguagem Brasileira de Sinais (LIBRAS), tornando computadores, dispositivos móveis e plataformas Web acessíveis para os surdos. Atualmente, o VLIBRAS é usado em vários sites públicos e privados, dentre eles os principais são: Governo Brasileiro (brasil.gov.br), Câmara dos Deputados (camara.leg.br) e Senado Federal (senado.leg.br)[1].

O nosso primeiro experimento tem como base uma primeira iteração da refatoração do código da **Suíte VLibras** através da abordagem proposta. O desempenho da nova versão, que passou a contar com um microsserviço no lugar do componente de tradução automática original, foi comparada com o desempenho da versão tradicional.

As subseções seguintes detalham mais sobre o funcionamento de serviços de tradução simultânea e também as modificações realizadas para customização da Suíte VLibras em uma arquitetura de MS.

5.1.1 Tradutores Automáticos de Línguas de Sinais

Hoje em dia, há consenso sobre a importância em prover condições para integração e acessibilidade das pessoas com qualquer grau de deficiência. Contudo, no âmbito computacional isto ainda precisa evoluir bastante. Mesmo antes da disseminação da web e a popularização dos computadores, já existia a preocupação de tornar estes novos ambientes virtuais acessíveis para pessoas com deficiências sensoriais.

Petrie(2004,p.1131) propôs alterações simples, como aumentar e customizar ícones, mas que mostraram ter um grande impacto para compreensão geral do conteúdo (PETRIE

[1] Mais informações podem ser obtidas em http://www.vlibras.gov.br.

et al., 2004). Todavia, desde lá os sistemas multimídia popularizaram-se e técnicas como esta não demonstram ser uma solução efetiva para estes novos paradigmas. Afim de contornar isto, é necessário traduzir o conteúdo a uma linguagem natural para o deficiente.

5.1.1.1 Suíte VLibras

No cenário Brasileiro, a tradução automática do Português para Libras representa um importante passo para a abertura das novas mídias aos deficientes auditivos. Grandes pesquisas vêm sendo feitas desde 2008 a fim de alcançar uma plataforma consistente para tradução Português-Libras, como por exemplo a FALIBRAS (TAVARES; CORADINE; BREDA, 2005), que define um sistema de tradução para trechos de áudio captado, ou o sistema SOTAC (BREDA et al., 2009), que tem como estratégia a comparação em dicionários estáticos. Contudo, apesar dos esforços, estas plataformas especializaram-se apenas em nichos e pouca experimentação prática fora concretizada com resultados satisfatórios.

Tabela 3 – Tradutores Automatizados Português – Libras

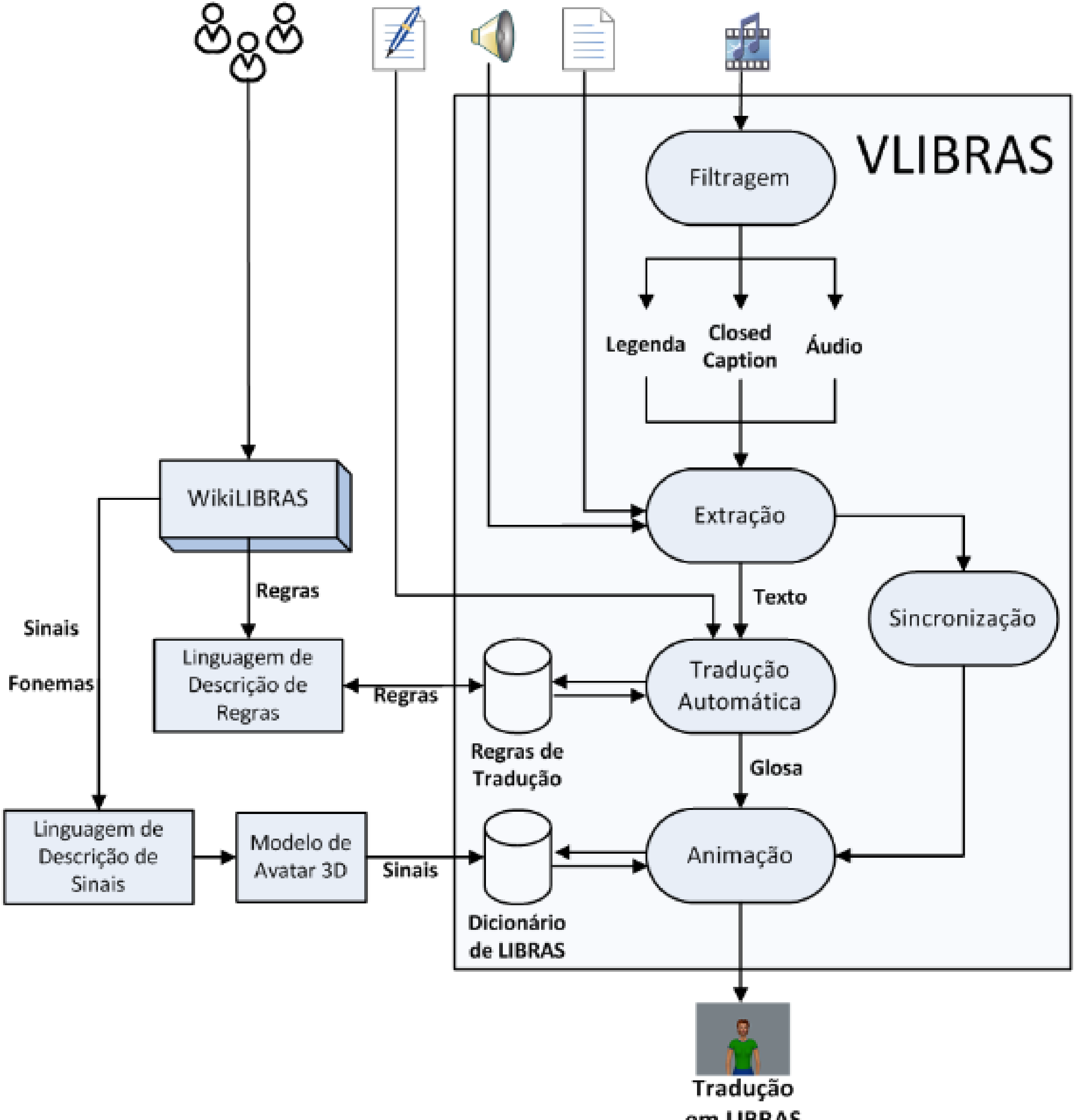
Tradutor	**Função**	**P/ Web**	**P/celular**	**Disponível**	**Desenvolvedor**	**Ano Publicação**
Rybená	Escrita/libras	X	X	Sim	CTS-DFJUG	2001- início
Veris	Oral/escrita/ libras	X	X	Não finalizado	Faculd. Veris	2009
Vlibras	Escrita/libras	X	X	Sim	UFPB	2014
FALIBRAS	Oral/escrita/ libras	X	X	Não finalizado	UFAL	2004
PUL0	Escrita/libras	X		Não finalizado	Unisinos	2005
SOTAC	Escrita/ libras	X		projeto inicial	UFES	2006
TLIBRAS	Oral/escrita/libras	X		Não finalizado	USP – S.Carlos	2004
POLI	Escrita/ libras	X		projeto inicial	E. Politéc. SP	2011

Adaptado de : Pivetta , 2011.

Pivetta em 2011 (PIVETTA; ULBRICHT; SAVI, 2011) compilou os resultados dos projetos em tradução automática Português-Libras em sua revisão sistemática. A Tabela 3 resume os dados apresentados Pivetta. Como é possível observar, nenhuma solução para a problemática de tradução da língua Portuguesa para Libras obteve expressivos sucessos no âmbito da tradução oral.

Com uma proposta de disponibilizar uma plataforma de tradução automática como um serviço adaptável a multíplas aplicações de apresentação, o Vlibras oferece todo o ferramental necessário para a tradução de Português-Libras em qualquer mídia.

Figura 15 – Arquitetura da Suíte Vlibras



Fonte: FALCÃO , 2014. (FALCÃO et al., 2014)

A Figura 15 referencia o modelo arquitetural do estudo em questão. Na arquitetura do Vlibras, podemos observar 3 grandes lógicas de negócio. A primeira, faz parte da **Extração** e filtragem das múltiplas mídias a serem traduzidas. Uma vez extraído o dado bruto, um processo de análise é feito sobre ele. Após a análise sintática e semântica, é possível então transcrever o texto extraído em uma linguagem auxiliar com correspondência direta em Libras, encerrando assim a etapa de **Tradução**.

Munido desta glosa traduzida, o último passo deste processo é a geração de vídeo (**Animação**). Neste, um mapeamento é feito junto a um banco de dados com anima-

ções previamente compiladas dos sinais em Libras. Finalmente então o processamento é encaminhado para o cliente (móvel ou web) e apresentado ao usuário na forma visual. Ainda, nesta arquitetura fora planejado um sistema auxiliar (*Wikilibras*) para que usuários fluentes em Libras possam expandir e atualizar o dicionário da Suíte Vlibras com seus *feedbacks*.

Contudo, em termos de elasticidade a Suíte Vlibras apresentava os mesmos obstáculos de qualquer sistema monolítico. A expansão de tal serviço se dava através de um balanceamento de carga feito através da replicação de todos os componentes (agregados em um bloco) da aplicação. O estudo de caso a ser apresentado neste trabalho evolui e incrementa a ideia em questão, adaptando o serviço Vlibras para o modelo arquitetural de microsserviços e contêineres.

Neste novo modelo proposto serão apresentados dois microsserviços que encapsulam as operações de cada diferente mídia. Em outras palavras, será detalhado a seguir um microsserviço com enfoque na tradução de textos da Língua Portuguesa para glosa e um segundo microsserviço para geração de vídeos com legendas em libras. Devido à natureza das operações de tradução de texto apenas um microsserviço foi extraído da Suíte Vlibras, caracterizando uma **orquestração simples**. Já para uma operação mais complexa que é a animação, múltiplos contêineres foram interligados numa **orquestração complexa** de microsserviços.

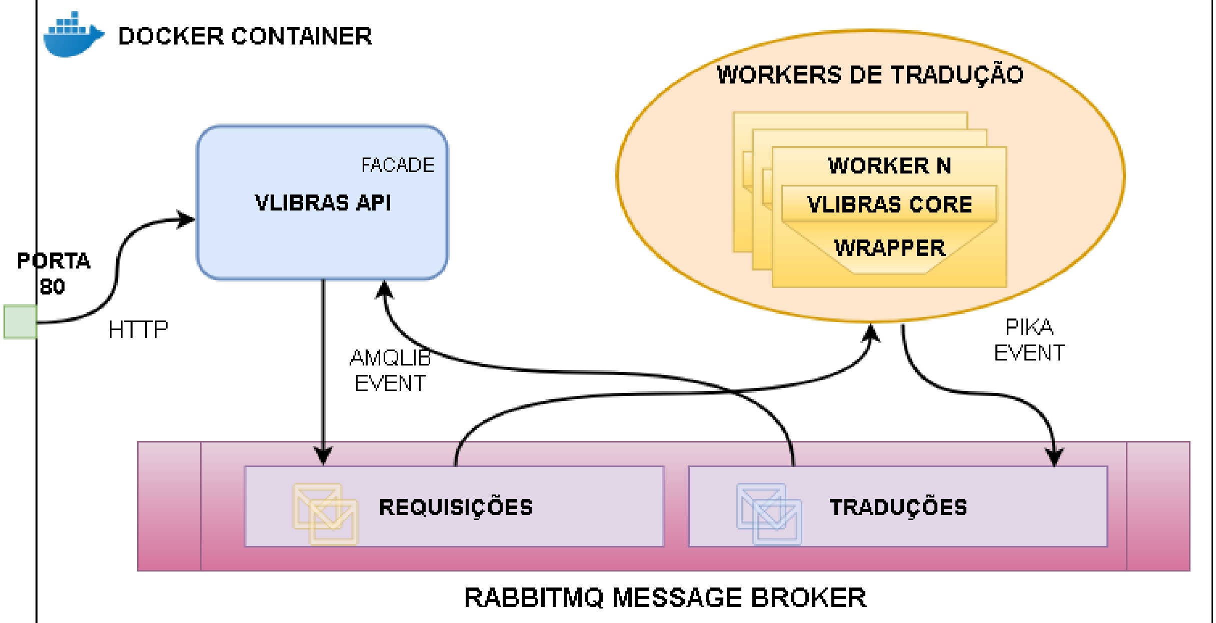
5.1.2 Transformando o Tradutor VLibras em um Microsserviço Reativo

O diagrama contido na Figura 16 ilustra a arquitetura de um novo microsserviço criado a partir da refatoração do código do **Tradutor** da Suíte VLibras. Modulado em um contêiner Docker (ANDERSON, 2015), o novo microsserviço oferece uma API *RESTful*, através da qual são feitos os pedidos de tradução por parte dos outros componentes da Suíte. Através de uma requisição GET é possível submeter um texto para tradução automática de português para LIBRAS, o qual, posteriormente, será sinalizado visualmente por um avatar através das diversas interfaces da Suíte VLibras.

Após receber uma requisição, a fachada do microsserviço enfileira a solicitação de tradução em uma fila assíncrona específica (**Requisições**) gerenciada pelo RabbitMQ (VIDELA; WILLIAMS, 2012). Por sua vez, um *pool* de *workers* de tradução, os quais fazem a conversão efetiva de português para LIBRAS, consomem as requisições da fila **Requisições**, processam-nas e encaminham as respostas para a fila de saída, chamada **Traduções**.

Esta organização possibilita o atendimento concorrente e simultâneo de várias solicitações de tradução ao mesmo tempo, sendo possível customizar o número de processos de tradução, ou seja, o tamanho do *pool* de *workers*) de acordo com a demanda e a capacidade do *hardware*.

Figura 16 – Microsserviço de Tradução Automática



Essa capacidade foi explorada nos nossos testes que consideraram vários tamanhos para o *pool* de *workers*.

5.1.2.1 Protótipo do Tradutor

Cada macro componente do modelo de conteinerização foi planejado e implementado de forma que funcione semelhante a um serviço *stand-alone*, como discutido anteriormente. A implementação a ser destrinchada a seguir foca apenas no serviço de tradução automática de texto utilizando contêineres Docker. Testes de performance em um contexto de orquestração simples foram planejados para este microsserviço, que sendo uma extensão do Vlibras faz uso das mesmas tecnologias apresentadas a seguir :

- **API**: A API é o ponto de controle e acesso ao serviço de tradução. Implementado através do framework *NodeJS/Express* (TILKOV; VINOSKI, 2010) para *JavaScript*, A API provê uma interface *RESTFUL*(*GET*) (RODRIGUEZ, 2008) na porta 80 aonde toda comunicação com o cliente é feita.

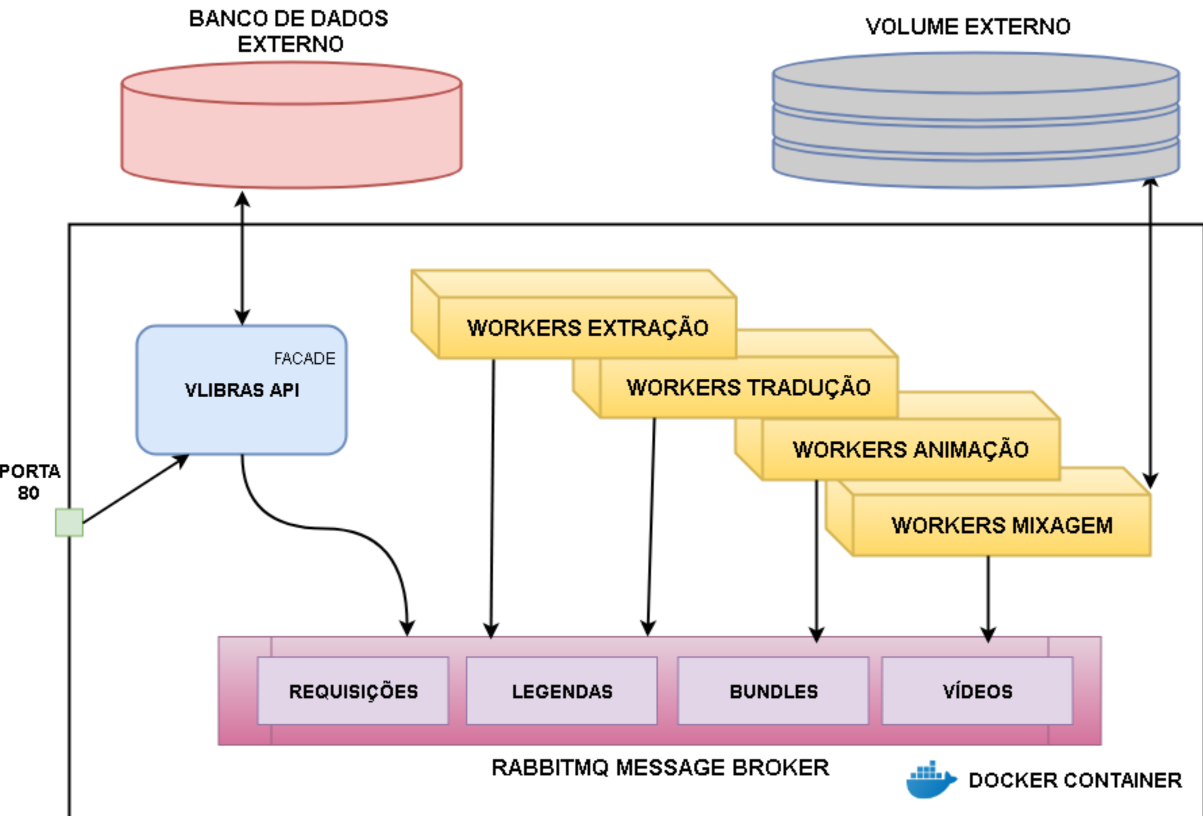
- **Filas Assíncronas**: Para tornar o sistema *Message-Driven*, a troca de mensagens(requisições) entre a API e os *N Workers* do contêiner, é utilizado um servidor de filas assíncronas *RabbitMQ* (VIDELA; WILLIAMS, 2012). Para cada categoria de *Worker* fora instanciado 2 filas, uma para leitura das requisições. Neste caso para o serviço de tradução são necessárias apenas 2 filas, nomeadas texto e libras respectivamente. A comunicação assíncrona acontece através de uma conexão TCP na porta 5672.

- ***Workers***: Principal componente no processo de tradução. Sendo implementado em *Python*, este componente encapsula a lógica de negócio interna do Vlibras(também em *Python*) que divide a semântica das frases e acessa dicionários e regras de tradução. Assim sendo, a pouca codificação necessária está em fazer a comunicação com as filas assíncronas. Por consumir e escrever destas filas, mais de um *Worker* pode ser instanciado sem nenhuma adaptação para a API e o serviço no geral.

5.1.3 Transformando o Gerador de Vídeos VLibras em um Microsserviço Reativo com Armazenamento

A arquitetura do microsserviço anteriormente apresentado específica uma orquestração simples de contêineres, ou seja, uma orquestração de apenas um tipo de microsserviço. Por outro lado a rotina para geração de vídeos (Animação) na Suíte VLibras possui quatro rotinas principais: a tradução de legendas, operações sobre dicionário de sinais, geração de vídeo em libras e mixagem. O diagrama da Figura 17 apresenta uma proposta de microsserviço reativo para cenário mais complexos como este.

Figura 17 – Microsserviço de Geração de Vídeos



Semelhante ao serviço Tradutor, o **Gerador de Vídeo** da Suíte Vlibras foi refatorado com base no código legado, utilizando um contêiner Docker. Internamente, este contêiner principal tem uma arquitetura semelhante ao serviço de tradução, com uma

API *RESTful* que se comunica através de uma fila assíncrona com um pool de *workers* do código vlibras. A primeira grande diferença dos dois serviços está na natureza destes *workers*. Enquanto no serviço de tradução existe apenas um tipo de worker, no serviço de geração de vídeo existem 4 tipos destintos, sendo estes: **extração** de legendas, **tradução** de legendas para glosa, **animação** das legendas e **mixagem** dos vídeos. Uma vez que estes componentes trabalham no mesmo conjunto de dados e dicionário internos, seria impossível refatorar cada uma dessas operações em um microsserviço separado.

A segunda diferença entre as arquiteturas reativas está ligada ao armazenamento das operações. O Tradutor Vlibras é um microsserviço *stateless*, ou seja, que não guarda estado sobre as operações. Uma vez processado o texto de entrada, nenhuma mídia sobra desta requisição, sendo apenas encaminhado para o cliente o texto de resposta (em glosa). O mesmo não é valido para operações de geração de vídeo. Uma vez mixado o vídeo com sua legenda em libras, este precisa ser **armazenado** por certo tempo até ser requisitado pelo cliente, dado ao alto custo de processamento e tempo destas operações. Assim sendo, foram necessários dois microsserviços de auxílio para esta arquitetura mais complexa.

Primeiramente um banco de dados foi introduzido a orquestração que se comunica diretamente com a API. Este **Banco de dados** guarda os estados das operações, informações dos clientes e endereçamento dos vídeos. Os arquivos em si dos vídeos gerados são armazenados por um **Volume Externo**. Esse serviço de volume age como uma memória compartilhada e controla acessos e alocações a disco por parte dos *workers* e contêineres de geração de vídeo.

5.1.3.1 Protótipo do Gerador de Vídeos

O processo de implementação do protótipo para o gerador de vídeos foi feito de forma semelhante ao protótipo do microsserviço de tradução de textos. A maior diferença entre as implementações está na complexidade da orquestração dos contêineres. Enquanto para orquestração simples serviço de tradução apenas um contêiner foi implantado, para orquestrar múltiplos contêineres de forma complexa isto não seria o suficiente. Desta forma a primeira diferença entre as abordagens está na utilização de um framework para orquestração de contêineres Docker. Dado ao seu amplo suporte da comunidade, fácil linguagem de específicação e o acesso gratuito para experimentos, a combinação da plataforma da Google (PENG et al., 2009) para máquinas virtuais com a tecnologias de Kubernetes se mostrou a opção ideal para implantação deste serviço.

Desta forma além do microsserviço principal, de geração de vídeo, que refatora o código legado da Suíte Vlibras, ainda foram preparados dois outros microsserviços como apresentado anteriormente. A implementação dos componentes da orquestração do protótipo do Gerador de Vídeos pode ser definida como se segue:

- **Contêiner do Gerador de Vídeos**:

  O processo para implementação do contêiner principal na orquestração do Gerador de Vídeos, foi feita usando as mesmas tecnologias e metodologias já discutida anteriormente na Seção 5.1.2.1. Assim sendo, a API de comunicação com o microsserviço de Geração de Vídeos foi implementada utilizando NodeJS e possui internamente filas assíncronas (em RabbitMQ) que são consumidas por *workers* em Python que encapsulam o código legado da Suíte Vlibras. A única diferença na implementação deste componente para a feita com o tradutor de textos está na quantidade de filas e consequentemente no tipo de microcomponentes encapsulados.

  Para cada uma das 4 filas utilizadas nesta implementação, uma rotina do processo de geração de vídeos traduzidos foi encapsulada em um tipo distinto de *worker* Python, caracterizando assim uma orquestração interna mais complexa que a apresentada anteriormente. Cada um destes *workers* Python podem ser escalados internamente, com exceção do micro-componente que gerência as operações de captura de imagens junto à biblioteca do "unity" e "ffmpeg", devido a restrições destas ferramentas. Em geral, este tipo de restrição é um forte indicio de que tal micro-componente possa ser separado como um microsserviço único. Contudo, devido ao forte acoplamento dos dados compartilhados, neste caso o isolamento desta micro-rotina em um serviço isolado não se fez possível.

- **Volume Externo**:

  O serviço de armazenamento externo de dados é a segunda peça fundamental para implementação do microsserviço de geração de vídeos. Este tipo de serviço vem se mostrando indispensável para aplicações que geram dados armazenáveis (neste caso, vídeos) e que sejam providas de forma distribuídas e escaláveis. A arquitetura da solução escolhida para este protótipo foi a de um **Sistema de Arquivos em Rede** (SHEPLER et al., 2003) (Network File System, ou NFS). A implementação deste microsserviço no contexto de Kubernetes utilizou-se da versão de código aberto desenvolvida pela comunidade do próprio framework[2]. Este componente consiste de um serviço(nfs-server) que serve de ponte entre o contêiner de geração de vídeo e o sistema de arquivos do *cluster*, através de um Kubernetes Persistent Volume.

- **Banco de Dados**:

  O último dos microsserviços providos para esta solução se trata do **Banco de Dados Relacional**. Este serviço serve diretamente o contêiner central que processa as operações requisitadas e armazena informações de clientes, suas requisições e o status destas operações. Devido à sua alta performance em armazenar dados, fácil

[2] Disponível em: https://github.com/kubernetes/examples/blob/master/staging/volumes/nfs/README.md. Acessado em: 2-1-2017

linguagem de consulta e uma arquitetura voltada para ambientes distribuídos, o MongoDB (CHODOROW, 2013) foi a escolha natural para a orquestração do gerador de vídeos. A comunicação entre o microsserviço do Vlibras foi feita através de forma direta através de chamadas a biblioteca nativa do NodeJS. Por fim, para implantação deste banco de dados na Kubernete, foi utilizado uma imagem Docker configurada com este serviço feito através da especificação da própria comunidade da plataforma[3].

[3] Disponível em: http://blog.kubernetes.io/2017/01/running-mongodb-on-kubernetes-with-statefulsets.html Acessado em: 7-1-2017

6 Experimentos e Resultados

Uma vez estruturada a refatoração do serviço VLibras para a arquitetura de microsserviços, o último passo da pesquisa em questão foi a implementação, execução e análise de experimentos mais completos para aferir a eficácia da arquitetura proposta. Estes testes, tiveram como principal foco a comparação entre a abordagem tradicional do Vlibras com a versão da mesma em microsserviços gerada com base na metodologia anteriormente apresentada, afim de verificar o impacto que nova arquitetura tem em relação aos principais pontos da programação reativa: **Desempenho**(ou *Responsiveness*), **Elasticidade** e **Resiliência**.

Como métricas para comparação, foi medido o **desempenho** de ambas as soluções em diferentes cenários e ambientes. O ambiente utilizado apresenta uma **orquestração complexa** de contêineres, estruturada em um provedor de nuvem pública. Estes testes foram feitos sobre o microsserviço de **Geração de Vídeo** do VLibras (ver Seção 5.1.3), o qual, pelo seu maior grau de complexidade, permitiu uma análise mais acurada dos efeitos da aplicação da arquitetura reativa no VLibras.

6.1 Ambiente de Testes

Tratando-se de arquiteturas de microsserviços, experimentos com enfoque em medir a elasticidade e a resiliência não devem ser realizados com apenas um servidor e em um ambiente completamente isolado. Em um cenário real, serviços desta natureza são em geral distribuídos em provedores de nuvem (públicos ou privados) que provisionam outros serviços distintos. Em outras palavras, um ambiente isolado de orquestração simples pode não representar fielmente o comportamento esperado do serviço quando em produção.

Desta forma, para os testes mais complexos de elasticidade e resiliência, um provedor de *cloud computing* foi escolhido para hospedar o serviço durante a execução dos testes. Como introduzido anteriormente na Seção 5.1.3.1, o provedor escolhido para isto foi o **Google Cloud Platform**. Uma vez que o microsserviço foi implementado utilizando a tecnologia Docker, foi necessário que o ambiente a ser escolhido provesse um framework com suporte a este tipo de contêiner e opções de escalabilidade.

A plataforma da Google provou-se a mais acessível considerando tais requisitos. Disponibilizando um ferramental nativo com suporte a *Kubernetes*, tanto o framework em questão quanto o provedor de nuvem apresentam a menor curva de aprendizagem das opções comparadas, além de um incentivo em desconto para primeiros testes. Além do cluster hospedado na nuvem, uma máquina fisíca(**Ubuntu, Intel i7-4790K/4GHz 16**

GB de memória) na **infraestrutura da UFPB** foi utilizado para o envio dos vídeos e legendas, agindo assim como um cliente.

Tabela 4 – Configurações do Cluster Kubernetes

Tipo	n1-standard-2
Região	us-central
Sistema Operacional	Ubuntu 17.04
CPUs	2 vCPUs
Memória	7,5 GB
Disco de Inicialização	200 GB
Escalonamento(VMs)	2 $\sim$ 4 VMs

A Tabela 4 apresenta as configurações utilizadas para configurar o Kubernetes Cluster. As máquinas virtuais alocadas foram do tipo “n1-standard-2”[1], com sistema operacional Ubuntu, 2 CPUs virtuais e 200 gigas de armazenamento, afim de aproximar ao máximo tais experimentos de uma infraestrutura replicável na UFPB. A cada execução dos testes o cluster alocado com os contêineres (em Pods) é completamente removido, para que a próxima execução parta do mesmo estado inicial, ou o mais próximo disto.

Ainda, o *cluster* escalonará suas máquinas virtuais de acordo com o consumo de processamento e Pods, variando de no mínimo 2 máquinas virtuais para até 4, todas com as mesmas configurações. No âmbito dos conteineres, as operações de redirecionamento e balanceamento de carga foram gerenciadas automaticamente pelo orquestrador Kubernetes. Neste, a variação foi de no minimo 1 a até 4 Pods. Já para o processo de escalonamento, o procedimento de “*scale-up*"ou “*scale-down*"foi configurado de maneira semi-automática, aonde o grupo de *Pods* de cada serviço a ser testado teve suas configurações mínimas e máximas descritas ao orquestrador Kubernetes e testes foram realizados em cada uma destas situações.

6.2 Projeto e Execução dos Experimentos

Afim de comparar as distintas abordagens(reativa e monolítica), o sistema monolítico do Vlibras também precisou ser expandido para deployment em uma orquestração complexa e escalável. De forma semelhante à solução proposta, o sistema monolítico foi encapsulado em um contêiner *Docker*, tendo sua arquitetura também dividida em serviços separados: Um contêiner principal contendo o serviço **monolítico do Vlibras**, um contêiner contendo um serviço de **armazenamento de dados** (MongoDB) e um serviço de **armazenamento e sincronização de arquivos em rede**, provido também por um contêiner. Este microsserviço sem refatoramento do código em micro-componentes, uso de filas e compilado de forma monolítica(serviço de Vídeos, Texto e Dicionário) foi chamado

[1] Disponível em : https://cloud.google.com/compute/pricing?hl=pt-br

de **microsserviço monolítico** e foi utilizado como comparação a arquitetura reativa proposta.

Para executar testes com este enfoque dada uma configuração distribuída, o ferramental anteriormente utilizado (*Jmeter*) para envio e controle do fluxo dos experimentos não se fez suficiente. Desta forma, além da implementação feita sobre a arquitetura monolítica para migrá-la em um contexto de microsserviços, também foi preciso a implementação de um novo controlador de testes especialmente planejado para estes experimentos.

Este novo cliente de testes foi desenvolvido na linguagem de programação interpretada Python versão 2.7, que contém um tipo nativo de variável para datas, além de suporte nativo a *Threads* e tratamento de mensagens em *JSON*. Além do core Python, este cliente fez uso da biblioteca "*Requests*" (REITZ, 2018) para o envio de requisições HTTP (GET/-POST) e auxílio da biblioteca "TQDM" (COSTA-LUIS; LARROQUE, 2018) para tratar com fluxo de arquivos sendo transferidos. Com o auxilio destas bibliotecas foi possível implementar todo o procedimento de envio e recebimento de vídeos, de forma customizável a necessidade do cenário a ser abordado.

Vale ressaltar entretanto, que devido a restrições físicas de rede e hardware, nem todos os vídeos gerados como resposta foram integralmente baixados. Para cenários de testes com um grande volume de vídeos, foi implementada uma operação de confirmação remota do status do vídeo, feita entre a API do microsserviço do Vlibras e o cliente externo. Nestes casos, a certeza da geração do vídeo de forma correta foi feita avaliando o tamanho e extensão dos arquivos gerados, além da confirmação remota do microsserviço.

O trecho de código abaixo demonstra o algoritmo utilizado pelo cliente para administrar o fluxo de um experimento. A função "vazaoTeste" cria os arquivos de log e administra as *Threads* referentes a uma repetição dos testes. Os parâmetros "ip", "port" e "proxy" são referentes ao microsserviço Vlibras mantido pela *Kubenertes/Google Cloud*. Já o "video" e "legenda" são referentes ao *path* dos arquivos a serem submetidos para geração do vídeo com legendas em libras.

```python
def ExperimentoVazao(ip, port, proxy, video, legenda, rep, exp, number,
    minute):
    for teste in exp:
        for n in number:
            for i in range(rep):
                runKubectlFile("apply","vlibras.yaml")
                time.sleep(20*60)
                vazaoTeste(ip, port, proxy, video, legenda, minute, teste, n, i)
                runKubectlFile("delete","vlibras.yaml")
                time.sleep(20*60)
```

Portanto, a execução de um experimento envolve então um número de repetições (“rep” no código acima) customizável, que para estes testes foram padronizadas em **5**. Cada fluxo destas repetições, levantará a infraestrutura de *Pods* no Cluster Kubernetes definidos nos arquivos “*.yaml*” para leitura e escalonamento no cliente Kubernetes(kubectl(VOHRA, 2017)).Além disto, a duração dos envios dos vídeos foi definida em **1 hora**, representada na variável “minute’. Por fim, o tempo máximo de espera por uma resposta positiva também é igual a duração de envios.

Deste modo, uma entrada que não tenha sido concluída com êxito pelo microsserviço, não tenha tamanho ou extensão esperada ou demore mais de 1 hora para ser concluída será considerada uma falha. Logo, a duração máxima de cada repetição é de 2 horas de atividade e 40 minutos de *set-up* dos microsserviços na infraestrutura. Todas as medições foram feitas em intervalos de 1 minuto e armazenadas no cliente junto dos vídeos processados. Segue abaixo um exemplo de um log gerado por este cliente.[2].

```
16427... 02:02.9 ...12:20.3 ...14:23.1 True Thread-1689
a46f8... 02:03.4 ...13:20.4 ...15:23.8 False Thread-1699
f1958... 04:05.4 ...12:20.3 ...16:25.7 True Thread-1688
```

Cada entrada de um vídeo submetido em uma repetição é armazenada em um arquivo “.csv”, contendo um id único criado pela “Vlibras API” do servidor. No cliente do experimento, o tempo total para conclusão da requisição é computada e armazenada junto as datas completas do inicio e do fim da requisição. Após as avaliações no cliente, o status da requisição (True/False) e o identificador único da Thread interna que gerenciou a operação é adicionada a linha e inserida no log na ordem de recebimento de resposta do servidor. A análise dos logs foi feita de forma semi-automática, também codificada em Python e utilizando das bibliotecas *Numpy* e *Pandas* (MCKINNEY, 2012).

6.3 Cenários dos Experimentos

Para ajustar os cenários dos experimentos, uma batéria de testes foi feita utilizando de uma orquestração simples (um servidor e um cliente). Porém, estes testes de ajustes tiveram como foco o serviço de **tradução**. Já para os experimentos reais o componente de estudo foi o **gerador de vídeos** do VLibras em uma orquestração complexa, usando como base os resultados preliminares obtidos para ajustar a variação do número de Workers.

A orquestração interna dos serviços também foi ajustada para estes novos experimentos, O número de *workers* testados em cada um dos cenários foi ajustado para variar entre 1, 4 e 8, de acordo com a Tabela 5. Estes valores foram estipulados em decorrência dos testes de orquestração simples já mencionados. Para comparação com o modelo

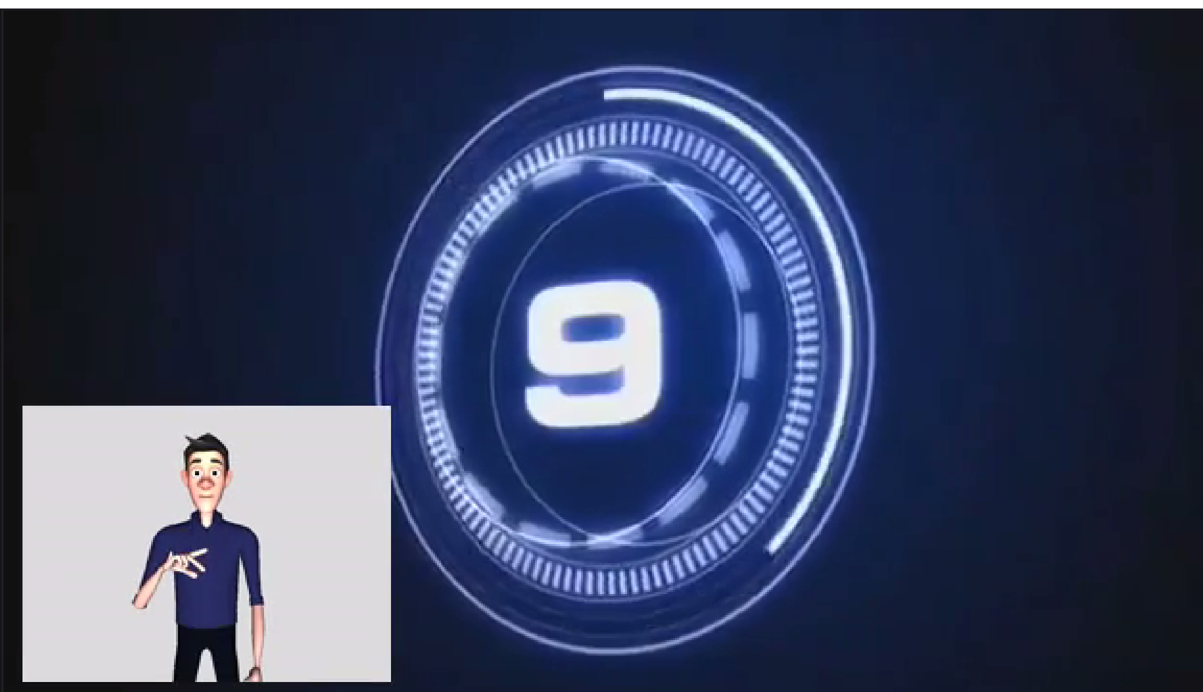
[2] Código, Experimentos e Logs disponíveis em : **https://goo.gl/QZbJTk**

Tabela 5 – Distribuição das Requisições para Testes de Elasticidade

Configurações	Pool de Workers
A	1 Worker
B	4 Workers
C	8 Workers
D	Vlibras Core

padrão, também foi testado o "microsserviço" da Suíte Vlibras sem nenhuma alteração exceto a adaptação para um contêiner Docker.

Figura 18 – Exemplo de Vídeo Gerado



Vale ainda salientar que o tipo de mídia a ser inserido foi "vídeo (formato mp4) com legenda". Uma vez que não foi demonstrada nenhuma diferença estatística relevante sobre o tamanho das entradas e o desempenho da arquitetura proposta, foram descartadas cenários variando esta grandeza. Para todos os cenários seguintes, o vídeo e a legenda se referem a uma contagem regressiva de 10 segundos como representado na Figura 18

6.3.1 Avaliação da Responsividade

A avaliação da responsividade do serviço Vlibras teve como principal foco a análise do eventual ganho em desempenho que a abordagem proposta pode proporcionar em uma orquestração complexa com o serviço de **Vídeo** da Suíte Vlibras. Neste sentido, o projeto dos experimentos considerou as seguintes variáveis de controle: a) tamanho do *pool* de Workers; b) quantidade de solicitações.

Tabela 6 – Distribuição das Requisições para Testes de Desempenho

Cenário	Intervalo entre Requisições	Taxa de Chegada
Carga Leve	12 minutos	5 vídeos/hora
Carga Baixa	8 minutos	7 vídeos/hora
Carga Normal	4 minutos	15 vídeos/hora
Carga Moderada	2 minutos	30 vídeos/hora
Sobrecarga	1 minuto	60 vídeos/hora
Stress	30 segundos	120 vídeos/hora

Tendo a vazão como uma das principais métricas a ser obtidas, foram levantadas cargas que devam variar do caso de **Stress** limitante para a carga **Leve**. Foram adicionados quatro outros cenários intermediários além dos extremos. A Tabela 6 enquadra os cenários planejados para o contexto do componente de geração de vídeo.

Nos cenários de “Sobrecarga e Stress"é esperado que ajam rejeições de requisições por parte do serviço, dada a tão grande demanda. Já para a “Carga Leve”, foi estipulada uma taxa de entrada que resulte em aproximadamente nenhuma rejeição, dado o pouco volume dos dados e grande espaço de tempo das requisições. Os cenários de carga “Baixa” e “Moderada” representam picos abaixo ou acima do comportamento padrão. Por outro lado “Carga Normal” é esperado o atendimento de todos clientes sem atrasos e com baixo consumo de recursos.

6.3.2 Avaliação da Elasticidade

No cenário anterior, apesar de se fazer uso de uma infraestrutura distribuída e uma orquestração complexa, o enfoque dado estava em medir a responsividade de uma única instância do microsserviço “Vlibras”. Temos neste caso o piso, ou seja, a menor configuração deste microsserviço.

Contudo, para avaliar a elasticidade do sistema é preciso não só avaliar o piso, como toda a capacidade de expansão do mesmo. Para avaliação da elasticidade, novos cenários de testes foram projetados, levando em conta específicadamente o número de instâncias do microsserviço. Apesar de suporte nativo a um processo automático de escalonamento, na prática este tipo de atividade acaba sendo seguida de uma revisão manual, em especial o processo de encerramento de instâncias com carga ociosa.

Desta forma, para simplificar e automatizar a rotina de testes, os cenários planejados para estes tipos de experimentos tiveram um número de **instâncias fixado em 4** (*Pods*). Este acréscimo no número de instâncias foi feito apenas no microsserviço do “Vlibras”. Os microsserviços de armazenamento de dados e arquivos, por não serem o objeto de estudo em questão, continuaram com a quantidade mínima de 1 *Pod*. Além disto, uma vez que o ponto principal está em avaliar as métricas de desempenho em relação a

um ambiente escalonado, a variação dos diferentes cenários passa a ser a quantidade de **vídeos submetidos em paralelo**. Deste modo, o intervalo da submissão das requisições para este tipo de medição foi fixado em **1 minuto**.

Tabela 7 – Distribuição das Requisições para Testes de Elasticidade

Cenário	Submissões Paralelas por Requisição	Taxa de Chegada (vídeo/hora)
Cliente Único	1	60 vídeos/hora
Carga Baixa	2	120 vídeos/hora
Carga Moderada	4	240 vídeos/hora
Stress	8	480 vídeos/hora

Na Tabela 7 é possível ver os quais cenários foram realizados para avaliar o potencial ganho da adoção de microsserviços reativos para elasticidade do sistema. Com base nos resultados experimentos de responsividade, quatro diferentes cenários foram projetados.

Como comparativo, o cenário de "cliente único", envia um único vídeo para ser processado a cada intervalo das requisições. Assim sendo, este caso corresponde ao mesmo *input* feito na "Sobrecarga" no contexto de responsividade e serviu para comparar a influência que os múltiplos *Pods* acarretaram ao sistema. Seguindo então a mesma progressão anterior, a "Carga Baixa" e "Carga Moderada" têm o dobro e o quadruplo de vídeos submetidos por minuto. Estes casos tem como objetivo extrapolar os limites do stress suportado por um único *Pod*, mas têm em teoria cargas aceitáveis para toda infraestrutura.

O último dos cenários em questão se refere ao estado crítico do sistema. Para caso de "Stress", foi planejada uma taxa de chegadas que provoque rejeição do sistema, mesmo considerando as múltiplas instâncias dos *Pods*.

6.3.3 Avaliação da Resiliência

O último dos aspectos desejados com a programação reativa e um dos principais pontos da adoção de microsserviços está na resiliência. Contudo, avaliar sistemas com falhas ou faltas não é uma atividade trivial, em especial quando se trata de ambientes em nuvem(BRILHANTE et al., 2014). Muitas são as técnicas para estimar a disponibilidade de um sistema, dentre estas podemos citar modelos analíticos, simulações de computador e medições reais. Dado o escopo desta pesquisa, a abordagem adotada foi a prototipagem e injeção de falhas.

Nesta, um segundo contêiner do protótipo do microsserviço "Vlibras" foi desenvolvido para injetar falhas controladas internamente ao sistema. Este script shell faz chamadas diretas ao sistema e interrompe os processos em execução dos *workers*, ou do *core* no caso do "microsserviço monolítico". Durante o tempo em que a falha está sendo

emulada no microsserviço, o injetor de falhas impede artificialmente a recuperação destes processos. Quando por fim o sistema deva retornar ao seu estado de normalidade, este controlador reinicia os processos que parou e continua a repetir este ciclo.

Desta forma, para este tipo de experimento as variáveis foram a **taxa de falha** e a **taxa de reparo**. Estas duas taxas juntas podem ser expressas em uma razão conhecida como **Disponibilidade** (Availability) (MALHOTRA; TRIVEDI, 1995), e serem utilizadas para calcular a média de tempo em que o sistema deve estar indisponível. Como uma extensão dos experimentos de elasticidade, os testes de resiliência foram executados utilizando a mesma configuração de **4 instâncias** de microsserviços "Vlibras" em uma taxa de chegada constante de **4 vídeos por minuto** em paralelo. A Tabela 8 denota os cenários projetados para cada disponibilidade esperada.

Tabela 8 – Injeção de Falhas para Testes de Resiliência

Cenário	Taxa de Falhas (em min)	Taxa de Reparo (em min)	Disponibilidade Aproximada (porcentagem)
Falhas Mínimas	8	1	90
Falhas Graves	8	2	80
Estado Crítico	8	4	50

Como de costume para este tipo de avaliação, em todos os 3 cenários planejados, tanto a taxa de falha quanto a taxa de reparo seguem uma distribuição exponencial (BALAKRISHNAN, 1996), por caracterizarem de forma geral, o comportamento de eventos distintos. Então, por exemplo, o cenário de "Falhas Mínimas" foi submetido a carga de stress relativa a 8 vídeos por minuto, enquanto seus *workers*, "tinham em média", 1 falha a cada 8 minutos e um reparo de uma eventual falha a cada minuto. De forma análoga, os cenários de "Falhas Graves" e "Estado Crítico" também sofreram em média uma falha a cada 8 minutos, mas demoraram o dobro e quádruplo, respectivamente, para se recuperar.

A taxa de falhas adotada em questão tem como base a duração máxima das submissões (60 minutos). Desta forma, é esperado que com 90% de certeza aja ao menos uma falha durante toda duração das requisições dos clientes. O último ponto a ser ressaltado é que devido a natureza da operação de geração de vídeo, o *worker* responsável por gerenciar as operações no *Unity Vídeo* não pode ser escalonado. Desta forma, prevendo um possível ponto único de falha, os mesmos cenários de testes apresentados na Tabela 8 foram executados com as falhas focadas neste gargalo. Com isto, esperasse avaliar qual o desempenho dos microsserviços reativos quando submetidos a restrições desta natureza.

6.4 Resultados Obtidos

A análise dos resultados obtidos em cada um dos experimentos levou em consideração duas métricas distintas. A primeira destas métricas se refere a vazão total do sistema, ou seja, o número médio de vídeos computados em uma fração de tempo qualquer. A segunda destas métricas computadas é percentual médio de erros encontrados. Esta Seção está dividida entre a análise dos resultados nos experimentos para cada um dos aspectos reativos : **Responsividade Elasticidade Resiliência**.

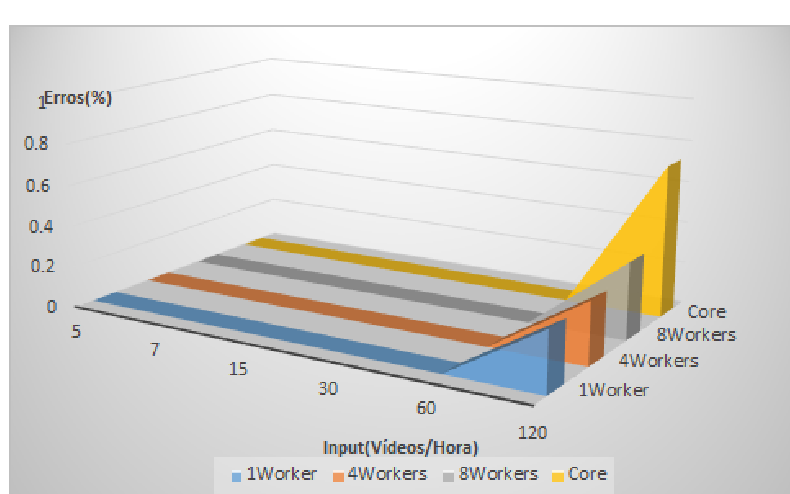
Para os experimentos de elasticidade e resiliência, por possuírem cenários de múltiplas *threads* clientes, também foi levantada uma métrica referente ao tempo médio para conclusão de uma requisição. Como múltiplas conexões esperam ativamente pelo retorno dos vídeos traduzidos, apenas a vazão do sistema não é capaz de representar o comportamento, como por exemplo caso aja *deadlock* ou interrupções a *threads* anteriores em decorrência de novos clientes.

De forma semelhante, nestes experimentos a vazão total ao longo do tempo pode não representar de forma mais assertiva o comportamento do sistema. Para estes casos, também foi levantada a vazão média do sistema ao decorrer do tempo, uma vez que devida a existência de clientes.

6.4.1 Desempenho

Como já visto anteriormente, sistemas reativos tem grande foco na resposta ao usuário. Neste contexto um sistema responsivo(**Responsive**), referente a métricas de performance e capacidade de responder de forma ágil *"on demand"*. Os resultados observados com os experimentos de desempenho da arquitetura proposta serão apresentados a seguir.

Figura 19 – Erro Médio(%) - Desempenho



O gráfico de área apresentado na Figura 19 representa a comparação entre a média

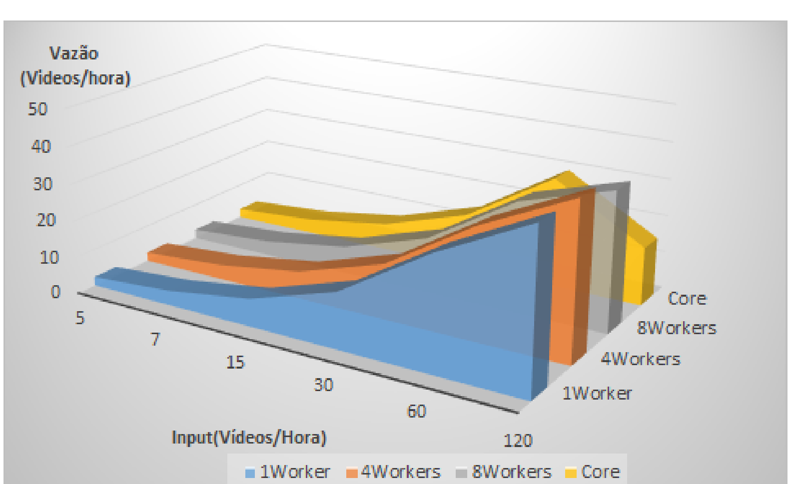
de erros de cada orquestração interna dos serviços em cada um dos cenários planejados em vídeos por hora (v/h). De imediato é notável o ganho de performance com a adoção da arquitetura distribuída com Kubernetes. Para os experimentos com a "carga leve" (5 v/h), "baixa" (7 v/h), "normal"(15 v/h), "moderada"(30 v/h) e "sobrecarga"(60 v/h), nenhuma das execuções ocasionou no impedimento do provisionamento do serviço de forma integral. Em outras palavras, duplicar a taxa de chegada nestas configurações não acarretou em falhas na computação dos vídeos e todas as requisições foram entregues.

Por outro lado, para uma carga de "stress"(120 v/h), foi possível verificar o surgimento de erros em todas as configurações do serviço. O pior resultado encontrado foi para configuração tradicional do "Vlibras Core". Para este microsserviço "monolítico", a porcentagem de erros médios chegou a **72.8%** com o desvio padrão de 23.8%. Já para as configurações de microsserviços que adotaram a arquitetura proposta, a média de erros obtidas foram de menos da metade. O pior caso dos microsserviço reativos para estes testes foi a orquestração de 8 micro-componentes internos. Dado ao uso excessivo dos recursos, tal distribuição não se mostrou a mais efetiva tendo o percentual médio de erros de **36.7%** (± 17.1).

Já para as configurações com 1 e 4 *workers*, os percentuais médios de erro foram de **28.8%** (± 09.2) e **29.8%** (± 10.5) respectivamente. Para as três configurações de microsserviços reativos, uma vez que seus limites são interpostos entre si, não se pode aferir estatisticamente uma diferença entre eles. Contudo, quando comparando estes casos com a versão do gerador de vídeo monolítico, fica comprovado o incremento na efetividade da arquitetura proposta.

Este percentual de erros médio, também pode ser visualizado em um número médio de requisições concluídas ao final do experimento. Ao número dos vídeos concluídos sobre a duração total do experimento têm-se a vazão geral do sistema ao final de 2 horas. A Figura 20 ilustra estes dados normalizados para a unidade de vídeos por hora.

Figura 20 – Vazão Média - Desempenho

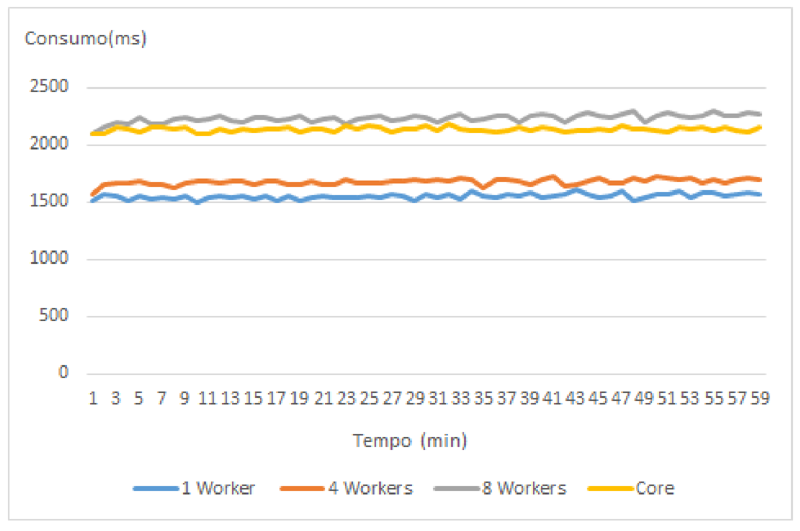


De forma semelhante ao percentual de erro, todas as arquiteturas propostas tiveram **resultados ideais** para os cenários de 5, 7, 15 , 30 e 60 vídeos por hora. Isto reflete a ociosidade do sistema quando submetido a estas cargas, conseguindo assim responder todas requisições de forma quase que imediata. A vazão destes cenários foram 0.042, 0.067, 0.125 e 0.25 vídeos por minuto ou **2.5** , **4**, **7.5**, **15** e **30** vídeos por hora.

A grande diferença entre as propostas se deu quando sobre stress. Para este cenário, a refatoração do Vlibras com a abordagem monolítica para microsserviços conclui, em média, apenas **15.72** (±13.68) vídeos por hora ou aproximadamente **1 vídeo a cada 4 minutos**. Por outro lado, o pior desempenho dos microsserviços reativos, obtido pela orquestração de 8 *workers* alcançou a entrega de **37.2** ($\pm 10,08$) vídeos por hora. Estes resultados foram ainda melhores para as orquestrações de 1 e 4 *workers*, com médias de **42.18** (±5.7) e **41.58** (±5.76) vídeos por hora. Desta forma, para cenários de cargas massivas, a solução reativa quase triplicou a responsividade do sistema, alcançando cerca de **2,4 vídeos a cada 4 minutos**.

Finalizando a avaliação do desempenho das arquiteturas, os recursos computacionais utilizados também foram medidos ao decorrer dos testes. Dada restrições de ambiente, o único recurso computacional disponível para medição neste caso foi o uso das vCPU. Nesta seção será abordada o consumo destes recursos para a carga de stress, que apresentou as maiores discrepâncias de erro e vazão.

Figura 21 – Consumo Médio da CPU - Desempenho



Os gráficos de linha na Figura 21, exibem os resultados ao longo do tempo da variação média da utilização das vCPUs, em microssegundos alocados. Por este motivo, o resultado da soma destes podem superar 1 segundo (ou 1000m), uma vez que o ambiente configurado contém ao todo 4 vCPUs. O comportamento demonstrado pelas curvas indica fortemente que a todos os cenários seguem a mesma função, deslocado apenas a média. Para a orquestração com 1 e 4 "workers" obteve-se uma média geral de **"1600"**

microssegundos ou **40%** da capacidade das 4 vCPUs alocadas.

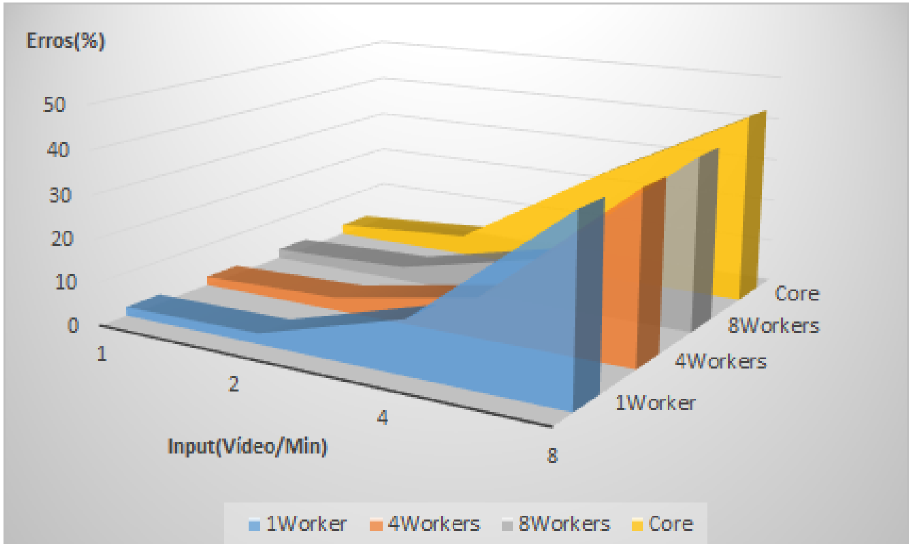
Por outro lado, o microsserviço monolítico obteve resultados ligeiramente melhores que a orquestração com 8 *workers*. Para o Core Vlibras, a média obtida foi de aproximadamente **2180**m (ou **54,5%**) em comparação aos aproximadamente **2240**m (ou **56%**). É notável então que a orquestração com 8*workers* obteve os piores resultados entre as abordagens reativas, sendo a menos indicada para uma orquestração simples. Contudo, de forma geral, o projeto reativo sinalizou um grande incremento na utilização dos recursos computacionais, que em um ambiente real acarretaria por baratear o custo do provisionamento e tornando o processo de *scale* mais eficiente.

6.4.2 Elasticidade

Uma vez comprovado o impacto que a arquitetura reativa de microsserviços tem em um contexto isolado, foram planejados e executados experimentos para avaliar este mesmo impacto agora no contexto elástico. "Elasticidade"se refere ao potencial do serviço crescer, para atender uma maior demanda e/ou reduzir para economizar gastos.

Utilizando os experimentos anteriores (Desempenho) como piso do processo de escalonamento de recursos é possível comparar o menor estado de recursos alocados do sistema, com a maior composição do serviço em uma orquestração complexa no ambiente apresentado. Desta forma, contrapondo os resultados logrados previamente com as medidas apresentadas a seguir, foi possível inferir sobre o impacto da orquestração interna para elasticidade de um microsserviço.

Figura 22 – Tempo Médio de Resposta - Elasticidade



A Figura 22 explica a variação do tempo até a entrega do vídeo (**Tempo Médio de Resposta**). Reafirmando os resultados já alcançados, o tempo médio de resposta com o ambiente em 4 *Pods* também exibiu resultados favoráveis aos microsserviços reativos.

Com o tempo médio de **1 vídeo a cada 2 minutos**, as configurações do *pool* de *workers* com 1 e 4 se mostraram novamente superiores para taxa de chegada de 1 vídeo por minuto. Neste mesmo cenário, o desempenho do Core Vlibras ficou em **1 vídeo a cada 2,5** minutos e o *pool* de 8 *workers* obteve resultados intermediários com **1 vídeo a cada 2,25 minutos**.

Todavia, uma vez que o desvio padrão destes intervalos sobrepõem-se, não é possível afirmar com certeza matemática uma diferença neste caso. O mesmo já não pode ser dito quando submetido a uma carga baixa (2 vídeos/minuto). Neste caso, o a utilização de apenas 1 *worker* obteve os melhores resultados, sendo capaz de processar **1 vídeo a cada 2,449**$(\pm 0,73)$ **minutos**, enquanto para os set de 4 e 8 *workers*, o tempo médio de resposta ficou em **2,99**$(\pm 1,10)$ e **3,30**$(\pm 1,20)$ minutos respectivamente. Mas a maior diferença obtida foi no microsserviço monolítico. Com **4,65**$(\pm 2,01)$ minutos para o processamento de um vídeo, ficou claro o ganho que os padrões reativos de programação têm sobre o sistema.

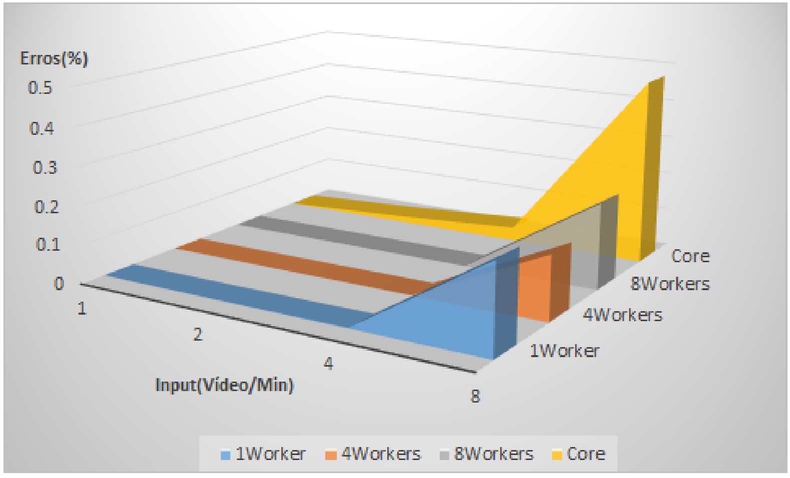
A fragilidade do microsserviço Vlibras sem refatoração também é evidenciada para taxas de chegada de 4 e 8 vídeos por minuto. Nestes casos, o Core Vlibras também apresentou os piores resultados, com tempo de **25,248**$(\pm 15,05)$ minutos para uma carga moderada e **43,501**(± 25) quando sobre stress. O principal a se observar para estas cargas está na configuração do *pool* de *workers*. Se para os experimentos até então os melhores resultados foram observados para uma configuração de apenas 1 *worker* por tipo de microcomponente, para as cargas moderada e de stress a situação se inverte.

Diferente do ocorrido em um ambiente de orquestração simples (1 *Pod*), nestes testes, a configuração que obteve os melhores resultados foi o conjunto de 4 *workers*, seguido pelo de 8 *workers* como segunda melhor composição. Com um tempo de entrega médio de apenas **9,248**$(\pm 5,2)$ minutos, a solução com 4 *workers* foi a única com uma média de tempo inferior a 10 minutos para uma carga moderada. As arquiteturas com 1 e 8 *workers* obtiveram médias de tempo semelhante, sendo respectivamente **12,242**(± 8) e **11,05**$(\pm 7,81)$ aproximadamente. No cenário de stress, a diferença entre os tempos das configurações reativas não foi tão grande, estando em **40.11**, **38.524** e **39.59** para 1, 4 e 8 workers nesta ordem.

De forma semelhante ao tempo médio de resposta, segue-se abaixo(Figura 23) o gráfico de superfície com resultados sobre o percentual de erros médio.

Como esperado, a taxa de 1, 2 e 4 vídeo por minuto não foram suficiente para gerar erro em nenhuma das abordagens reativas. Já para o microsserviço monolítico Core, a porcentagem de falhas variou de 0 (carga baixa) a **3%**(carga moderada) com desvio padrão contendo o 0. Já para a taxa de chegada de 8 vídeos em paralelo, 4 *workers* demonstraram novamente os melhores resultados.

Figura 23 – Erro Médio (%) - Elasticidade



Com o melhor de *trade-off* entre paralelismo interno e a utilização de recursos, a configuração com 4 *workers* obteve um percentual de apenas **16.5%** de falhas, contra os **22%** das configurações de 1 e 8 workers. O maior percentual de erros neste hipotético *"scale-up"* ficou por parte da abordagem tradicional, com **47.77%** de falhas dos vídeos recebidos.

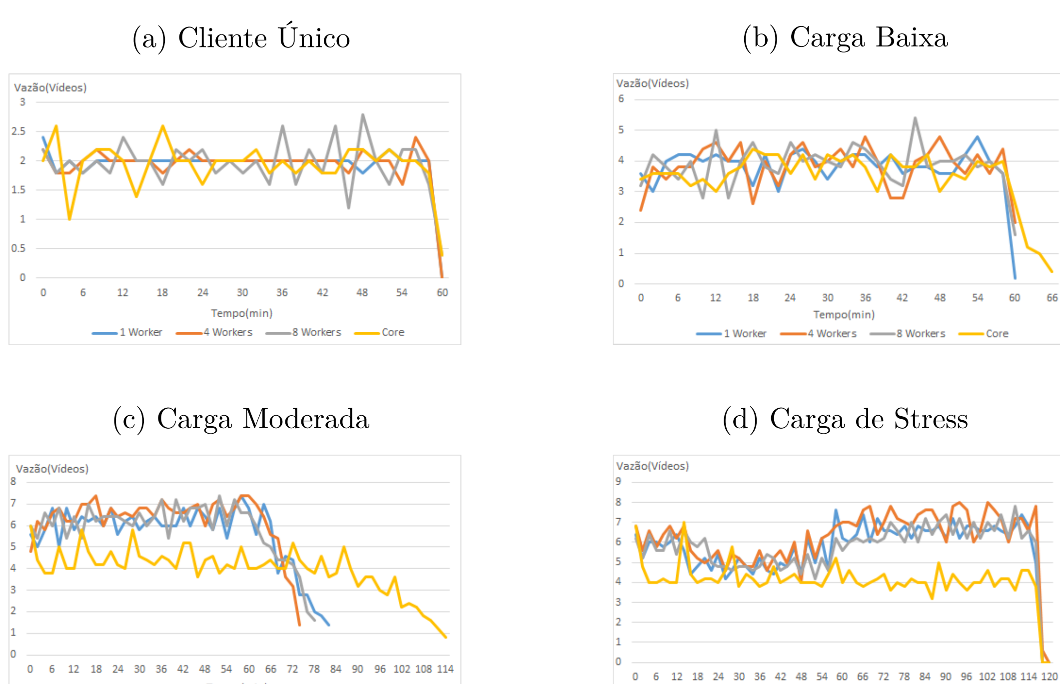
Ainda assim, apenas minimizar a quantidade de erros não caracteriza em si uma melhoria da escalabilidade do sistema. É desejado também que a velocidade da entrega dos vídeos gerados não seja prejudicada com o incremento na capacidade de atendimento. Tendo isto em mente, a comparação entre a vazão das abordagens pode ser vista nos gráficos da Figura 24.

Os gráficos 24a, 24b, 24c e 24d mostram a vazão média ao decorrer do tempo para os cenários experimentados. A uma taxa de 1 vídeo por minuto (Cliente Único), a vazão média de todas as configurações testadas seguem uma função aproximadamente constante de média **2** vídeos por minuto. Apesar das configurações com 1 e 4 *workers* apresentarem uma menor oscilação, todos as configurações seguem uma distribuição semelhantes.

De forma semelhante, este comportamento se repete para a "Carga Baixa", tendo uma média da vazão oscilando em torno de **4** vídeos por minuto, como visto na Figura 24b. Isto se deve ao fato de que, por submetidos a uma taxa de chegada inferior ao limite que os sistemas podem responder (taxa de saída média), não são esperados grandes atrasos e/ou perdas em filas.

Em contrapartida, a vazão média para as cargas "Moderada" e de "Stress" evidenciaram grandes diferenças entre a abordagem reativa da versão "monolítica". Em um cenário de "Stress", submetido a 8 vídeos por minuto, a abordagem tradicional oscilou de forma constante em **4,5** vídeos por minuto. Já para a proposta reativa, a vazão média cresceu após a primeira hora do experimento, conseguindo alcançar uma média de aproximadamente **7** vídeos por minuto durante a segunda hora do experimento, alcançando

Figura 24 – Vazão - Elasticidade



picos de até **8** vídeos para a melhor composição interna (4 *workers*).

Mas o caso que melhor evidenciou a superioridade da arquitetura reativa foi o cenário de "Carga Moderada"(4 vídeos/minuto). Atingindo a capacidade máxima dos recursos alocado na orquestração complexa, a abordagem de microsserviços conseguiu computar a bateria de testes em muito menos tempo. Neste cenário, a versão monolítica do Vlibras, limitada a uma vazão de até **5** vídeos/minuto, demorou quase toda duração do experimento para finalizar os vídeos, durando em média **114** minutos. Por sua vez, para o *pool* reativo de micro-componentes, a mesma carga pode ser finalizada em até **74**(4 *workers*), **78**(8 *workers*) e **82**(1 *worker*) minutos.

Desta forma, todas as métricas(vazão, tempo de resposta e erro médio) levantadas corroboram com a utilização de práticas e designs reativos na arquitetura interna de um microsserviço. Uma vez que, além de um desempenho superior ou igual em orquestrações simples, a adição da escalabilidade interna dos micro-componentes incrementou a performance do processo de *"scale"* de todo microsserviço.

6.4.3 Resiliência

A resiliência, a última das diretrizes reativas a ser analisadas neste trabalho, se trata da capacidade de um sistema de tolerar uma eventual falha e recuperar-se. Dada a orquestração interna de um microsserviço reativo, uma falha pode ser injetada em qualquer um dos seus micro-componentes. específicamente para o objeto de estudo em questão (Gerador de Vídeo Vlibras), um destes *workers* não permite replicações.

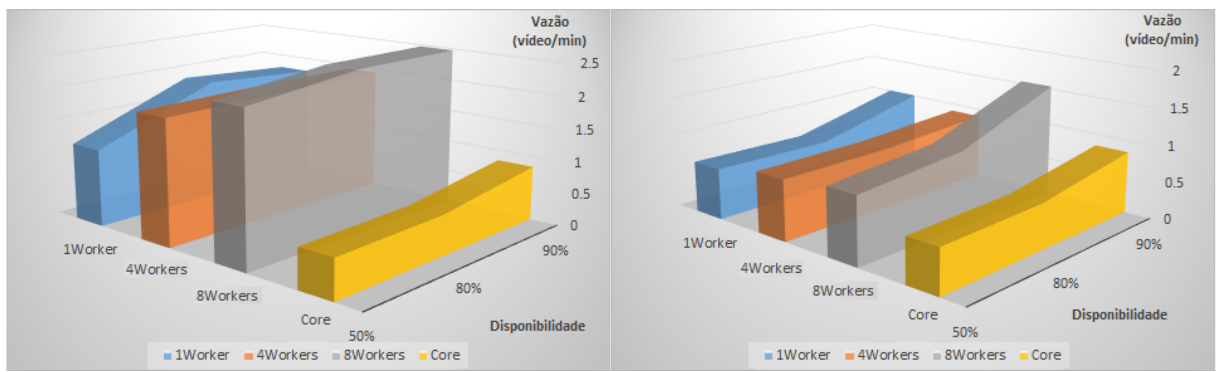
Deste modo, os resultados a seguir foram divididos entre os casos com falhas geradas em qualquer um dos componentes (**Falhas Gerais**) e aqueles em que as falhas injetadas foram direcionadas ao componente único do MS(**Apenas Renderizador**). Os cenários projetados variam a disponibilidade (razão entre as falhas e reparos) em 90%, 80% e 50%. Análogo aos cenários anteriores, as métricas de interesse deste caso foram a vazão, tempo médio e a porcentagem de erros.

A Figura 25 explicita a vazão média obtida nestes experimentos. Representado pela superfície amarela, a vazão média do Core Vlibras obteve os piores resultados, com uma vazão média de **0.838**, **0.612** e **0.593** para os cenários de "Falhas Mínimas"(90%), "Falhas Graves"(80%) e "Estado Crítico"(50%) respectivamente. Mesmo considerando as falhas apenas no renderizador(Figura 25b, a solução reativa com 1 *worker* já demonstrou melhores resultados, tendo uma vazão média de **1.007**, **0.657** e **0.688** nesta ordem. Porém a diferença fica ainda maior quando se considerando o aumento do número de *workers* no *pool* interno.

Figura 25 – Vazão Média - Resiliência

(a) Falhas Gerais

(b) Apenas Renderizador



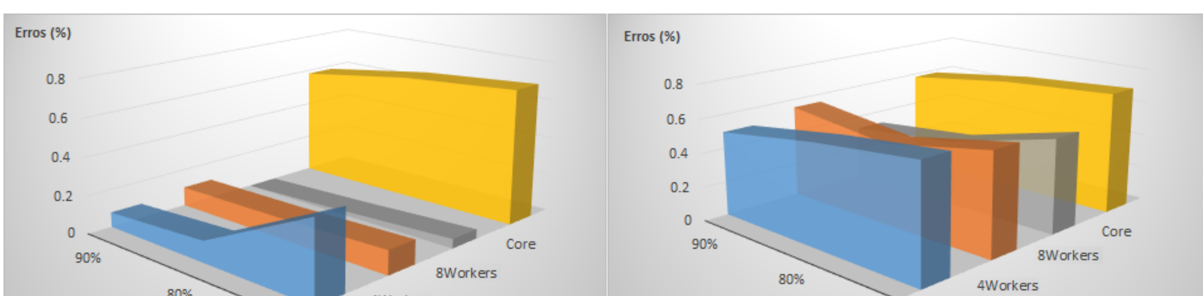
Ainda que a falha injetada não atinja diretamente os micro-componentes replicados, o paralelismo das filas assíncronas acarretam em fluxo de execução mais ágil. Apesar de, sobre "Falhas Mínimas" a orquestração de 4 *workers* ter uma média(**0.895**) inferior a com 1 *worker*, a progressão da degeneração da vazão com o crescimento da taxa de reparo é bem menor, variando para aproximadamente **0.818** nos casos de falhas graves e críticas. Já a arquitetura com 8 *workers* teve os melhores desempenhos sobre falhas, conseguindo uma vazão média de **1.535** quando a 90% de disponibilidade e até **0.902** para 50%, superando de longe a arquitetura monolítica e a configuração com apenas 1 *worker*.

Este caso se repete também quando as falhas simuladas foram distribuídas por todos micro-componentes e não apenas no renderizador(Figura 25a). Neste cenário, o

impacto das falhas sobre a vazão do sistema caiu drasticamente, com a orquestração interna de 1 *worker* ainda detendo os menos promissores resultados (**1.777**, **1.892** e **1.183** para 90%, 80% e 50% respectivamente). Já a configuração com 4 *workers* deteve as menores variações entre os cenários de falha, atingindo uma média de **1.927** para uma disponibilidade esperada de 90% e a média de **1.909** para 50%. Entretanto, os melhores mesmo considerando falhas em qualquer componente foram por parte do *pool* com 8 *workers*, que obteve uma média de **2.474** para 90% e 80%, decaindo para **2.282** no "Estado Crítico".

Figura 26 – Erro Médio (%) - Resiliência

(a) Falhas Gerais (b) Apenas Renderizador



Reforçando as conclusões da análise da vazão média, a porcentagem média de erros(Figura 26) também retratou um melhor funcionamento com MS reativos. No gráfico 26a é possível observar com clareza os benefícios de uma comunição orientada a mensagens. Com um percentual de erros variando de **58.1%** (falhas mínimas) a **71%** (falhas críticas), o MS monolítico teve um percentual de erros **15 vezes maior** do que a melhor orquestração reativa(8 *workers*, com apenas **4.9%** de falhas em um cenário crítico. Mesmo para os *pools* contendo 1 e 4 *workers*, o percentual de erros foi bem menor.

A configuração com 4 *workers*, por exemplo, obteve um percentual de apenas **10.1 à 12.5%** de erros, estatisticamente não sendo impactado pela taxa de reparo das falhas injetadas. Contudo, com apenas 1 *worker*, esta taxa de reparo impactou mais fortemente nos resultados. Para uma disponibilidade de 90%, a orquestração com 1*worker* apresentou menos erros(**7.8%**) que o MS reativo com 4 destes. Para um percentual de 80% de disponibilidade, ambas as configurações(1 e 4) tiverem resultados semelhante estatisticamente (**11.2** e **10.7**), mas para uma taxa de reparo de 4 minutos (50% de *availability*) o pool sem replicações falhou em média **40.8%**, demonstrando que a escalabilidade interna de micro-componentes impactou positivamente na resiliência do sistema.

Mesmo com falhas focadas apenas no renderizador (Figura 26b) este comporta-

mento se repetiu. A orquestração com 8*workers* continuou com os melhores resultados obtidos, com **32.2%**, **38.4%** e **54.9%** em ordem de maior disponibilidade para menor. De forma semelhante à configuração com 4*workers* obteve um percentual de falhas quase constante entre os cenários, variando de **55.2%** à **59.4%** entre as falhas mínimas e falhas críticas. Isto se deve, em geral, a não otimizado número de *workers* deste cenário.

Ainda que aja paralelismo interno suficiente para diminuir os erros e aumentar a vazão do sistema, ainda assim não houve processamento paralelizado o suficiente para desengarrafar as filas assíncronas nos períodos que o renderizador funcionava normalmente. Como esperado também, com erros gerados apenas no renderizador os erros médios para 1 *worker* quase coincidiram com o Core Vlibras, manifestando **49.7%**, **59%** e **65.6%** de erro médio para as taxas de reparo de 1(90%), 2(80%) e 4(50%) minutos respectivamente.

6.5 Considerações Finais

Analisando a proposta de MS reativos com uma abordagem monolítica "padrão", fica claro que a adoção de filas assíncronas para comunicação interna tem o potencial de melhorar consideravelmente a eficiência do sistema, não só levando-se em conta as diretrizes da programação reativa. Mesmo nos piores cenários, a arquitetura proposta demonstrou rendimento no mínimo semelhante à versão Core do Vlibras em contêineres, comprovando a eficiência da metodologia apresentada no processo de refatoramento de um serviço monolítico em microsserviços.

7 Conclusão

Neste trabalho foi proposto uma nova metodologia para refatoração ou desenvolvimento de novos microsserviços baseada em boas práticas de design para serviços web, denominada de programação reativa. Com o intuito de minimizar a codificação e o projeto de microsserviços, desacoplar os "módulos" das macro-funcionalidades de um sistema monolítico exibiu promissores resultados. Como contribuições deste trabalho, destacam-se: um passo a passo para se definir o escopo de um novo microsserviço; uma metodologia de como arquitetar ou refatorar novos microsserviços com filas assíncronas; a migração de uma aplicação monolítica (Vlibras) para MS; e uma abordagem para testes e avaliações de microsserviços.

O presente trabalho teve o objetivo de melhorar o processo de refatoramento e design para MS, que ainda é pouco discutido academicamente e acaba por ser muitas vezes baseado puramente na intuição e conhecimento empírico da equipe de desenvolvimento. Este trabalho dá um primeiro passo a conjuntura de *design patterns* focados em microsserviços. Alguns testes computacionais foram realizados e mostraram através das métricas de desempenho, elasticidade e resiliência que dada a devida atenção de como restruturar um sistema monolítico, é possível incrementar demasiadamente sua performance sem a necessidade alterações maiores no código fonte.

No que tange o desempenho do microsserviço, não faziam parte do escopo destes testes a sugestão de uma configuração ideal para ser adotada pelo estudo de caso (Vlibras) ou qualquer outro ferramental. Como demonstrado nos resultados obtidos, o escalonamento interno de micro-componentes acarreta em um *trade-off*: por um lado, o menor número de recursos em *loop* acarreta por um desempenho superior para uma orquestração pouco complexa; por outro, um maior número de *workers* aperfeiçoa a tolerância a falhas do sistema ao custo de mais recursos computacionais. Desta forma, a melhor configuração entre o número de micro-componentes escalonados acaba também por depender da natureza da aplicação e ambiente de provisionamento.

Contudo , esta limitação pode ser investigada em trabalhos futuros. Assim como ocorre um *auto-scale* com microsserviços, esta ideia análoga pode ser aplicada para os *workers* internos a ele. Como trabalho futuro também se faz necessária a comparação da metodologia proposta com um sistema inteiramente projetado como microsserviço. Excluindo códigos legados e restrições monolíticas de modelagem, talvez seja possível aperfeiçoar a metodologia proposta. Por questões de escopo, testes desta natureza não foram objetivo desta pesquisa.

Além dos pontos ressaltados anteriormente, a adição de uma coreografia interna

baseada em filas pode vir a endereçar um dos principais entraves quanto à adoção de microsserviços: a **curva de aprendizado** na migração e incorporação de trechos de sistemas monolíticos aos novos microsserviços. Na abordagem proposta, o código inteiro do trecho desejado pode ser reaproveitado sem grandes mudanças em termos de codificação.

Neste caso, os microcomponentes candidatos devem ser identificados e agrupados em *workers* que processarão mensagens das filas de acordo com a coreografia definida. O código do *worker* pode ser encapsulado em um *wrapper* e integrado ao resto do microsserviço através do uso da API específica ao *middleware* de fila assíncrona adotado.

Desacoplando os componentes internos em um microsserviço, é facilitado todo o procedimento para alteração e/ou adição de um novo componente ou funcionalidade. Isto também se reflete em pontos como a gerência de logs e debug, que uma vez tendo auxílio de um *middleware* de trocas de mensagem, já possui um ponto central para a obtenção destes dados. Contudo, vale salientar que a facilidade para loggificação e debug está restrito apenas ao contexto interno do microsserviço.

Por fim, um efeito colateral esperado é que o microsserviço, como um todo, pode se tornar um pouco mais complexo, uma vez que a sua comunicação interna também passa a ser dirigida por mensagens. Entretanto, acreditamos que a adoção de um padrão arquitetural comum, encapsulado em um *framework* e bem documentado, possa minimizar tal problema. Todos estes benéficios adicionais da adotação de filas assíncronas para migração de sistemas legados ainda precisam de estudos para comprovar estas suposições.

Referências

ADAMS, K.; AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, ACM, v. 40, n. 5, p. 2–13, 2006. Citado na página 29.

ALSHUQAYRAN, N.; ALI, N.; EVANS, R. A systematic mapping study in microservice architecture. In: IEEE. *Service-Oriented Computing and Applications (SOCA)*. [S.l.], 2016. p. 44–51. Citado 2 vezes nas páginas 39 e 45.

AMAZON. *Amazon EC2*. 2016. Disponível em: <http://aws.amazon.com/ec2>. Citado na página 26.

ANDERSON, C. Docker. *IEEE Software*, v. 32, n. 3, 2015. Citado 2 vezes nas páginas 39 e 66.

ARMBRUST, M. et al. A view of cloud computing. *Communications of the ACM*, ACM, v. 53, n. 4, p. 50–58, 2010. Citado na página 21.

BALAKRISHNAN, K. *Exponential distribution: theory, methods and applications*. [S.l.]: CRC press, 1996. Citado na página 80.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. *Microservices migration patterns*. [S.l.], 2015. Citado na página 45.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Migrating to cloud-native architectures using microservices: an experience report. In: SPRINGER. *European Conference on Service-Oriented and Cloud Computing*. [S.l.], 2015. p. 201–215. Citado na página 48.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016. Citado na página 49.

BASS, L.; WEBER, I.; ZHU, L. *DevOps: A Software Architect's Perspective*. [S.l.]: Addison-Wesley Professional, 2015. Citado na página 39.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, IEEE, v. 1, n. 3, p. 81–84, 2014. Citado na página 40.

BONER, J. Reactive microservices architecture. O'Reilly Media, 2016. Citado na página 43.

BONÉR, J. et al. *The reactive manifesto*. [S.l.]: url: http://www.reactivemanifesto.org/pdf/the-reactivemanifesto-2.0.pdf, 2014. Citado 2 vezes nas páginas 23 e 41.

BOWMAN-AMUAH, M. K. *Load balancer in environment services patterns*. [S.l.]: Google Patents, 2003. US Patent 6,578,068. Citado na página 35.

BREDA, W. et al. Sotac: a software for knowledge-based automatic translation. In: *Proceedings of the 9th IFIP World Conference on Computers in Education, Springer, Bento Gonçalves, RS, Brazil*. [S.l.: s.n.], 2009. v. 1, p. 1–10. Citado na página 64.

BRILHANTE, J. et al. Eucabomber 2.0: A tool for dependability tests in eucalyptus cloud infrastructures considering vm life-cycle. In: IEEE. *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*. [S.l.], 2014. p. 2669–2674. Citado na página 79.

BUYYA, R.; YEO, C. S.; VENUGOPAL, S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In: IEEE. *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. [S.l.], 2008. p. 5–13. Citado na página 25.

CHE, J. et al. A synthetical performance evaluation of openvz, xen and kvm. In: IEEE. *Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific*. [S.l.], 2010. p. 587–594. Citado na página 30.

CHODOROW, K. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. [S.l.]: "O'Reilly Media, Inc.", 2013. Citado na página 71.

COMPTON, B. T.; WITHROW, C. Prediction and control of ada software defects. *Journal of Systems and Software*, Elsevier, v. 12, n. 3, p. 199–207, 1990. Citado na página 38.

COSTA-LUIS, C. da; LARROQUE, S. *Github - Python TQDM*. 2018. Disponível em: <https://github.com/tqdm/tqdm>. Citado na página 75.

CURRY, E. Message-oriented middleware. *Middleware for communications*, John Wiley & Sons, p. 1–28, 2004. Citado na página 56.

DANIEL, F.; PERNICI, B. Insights into web service orchestration and choreography. *International Journal of E-Business Research (IJEBR)*, IGI Global, v. 2, n. 1, p. 58–77, 2006. Citado 2 vezes nas páginas 23 e 39.

DANIEL, F.; PERNICI, B. Web service orchestration and choreography: Enabling business processes on the web. *E-Business Models, Services, and Communications-Advances in E-Business Research Series*, v. 2, p. 251–274, 2007. Citado na página 39.

DESHANE, T. et al. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, p. 1–2, 2008. Citado na página 29.

DIKAIAKOS, M. D. et al. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet computing*, IEEE, v. 13, n. 5, 2009. Citado na página 25.

DRAGONI, N. et al. Microservices: Migration of a mission critical system. *arXiv preprint arXiv:1704.04173*, 2017. Citado na página 35.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036*, 2016. Citado 4 vezes nas páginas 21, 36, 38 e 49.

EMAM, K. E. et al. The optimal class size for object-oriented software. *IEEE Transactions on software engineering*, IEEE, v. 28, n. 5, p. 494–509, 2002. Citado na página 39.

ESPOSITO, C.; CASTIGLIONE, A.; CHOO, K.-K. R. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing*, IEEE, v. 3, n. 5, p. 10–14, 2016. Citado na página 38.

FALCÃO, E. d. L. et al. Deaf accessibility as a service: uma arquitetura escalável e tolerante a falhas para o sistema de tradução vlibras. Universidade Federal da Paraíba, 2014. Citado na página 65.

FAZIO, M. et al. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, IEEE, p. 81–88, 2016. Citado na página 38.

FOWLER, M.; LEWIS, J. Microservices a definition of this new architectural term. *URL: http://martinfowler.com/articles/microservices.html*, 2014. Citado na página 22.

FRANCESCO, P. D. Architecting microservices. In: IEEE. *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. [S.l.], 2017. p. 224–229. Citado 2 vezes nas páginas 23 e 45.

GOIRI, I.; GUITART, J.; TORRES, J. Characterizing cloud federation for enhancing providers' profit. In: IEEE. *2010 IEEE 3rd International Conference on Cloud Computing*. [S.l.], 2010. p. 123–130. Citado na página 27.

GOMEZ, A.; LARA, C.; KEBSCHULL, U. Intrusion prevention and detection in grid computing-the alice case. In: IOP PUBLISHING. *Journal of Physics: Conference Series*. [S.l.], 2015. v. 664, n. 6, p. 062017. Citado na página 30.

GOUIGOUX, J.-P.; TAMZALIT, D. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In: IEEE. *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. [S.l.], 2017. p. 62–65. Citado na página 46.

GUCER, V.; NARAIN, S. et al. *Creating Applications in Bluemix Using the Microservices Approach*. [S.l.]: IBM Redbooks, 2015. Citado na página 36.

GUPTA, A. *Microservice Design Patterns*. 2015. `http://blog.arungupta.me/microservice-design-patterns/`. Citado na página 37.

GUTIERREZ, F. Microservices. In: *Spring Boot Messaging*. [S.l.]: Springer, 2017. p. 179–192. Citado na página 43.

HANSON, M. D. The client/server architecture. *Server Management*, CRC Press, p. 3, 2000. Citado na página 42.

HARDESTY, L. *Netflix Describes its Use of Apache Mesos in 3 Use Cases*. 2016. Disponível em: <https://www.sdxcentral.com/articles/news/netflix-desribes-use-apache-mesos/2016/08/>. Citado na página 40.

HASSAN, S.; BAHSOON, R. Microservices and their design trade-offs: A self-adaptive roadmap. In: IEEE. *Services Computing (SCC), 2016 IEEE International Conference on*. [S.l.], 2016. p. 813–818. Citado na página 39.

HATTON, L. Reexamining the fault density component size connection. *IEEE software*, IEEE, p. 89–97, 1997. Citado na página 38.

HOSCHEK, W. The web service discovery architecture. In: IEEE. *Supercomputing, ACM/IEEE 2002 Conference*. [S.l.], 2002. p. 38–38. Citado na página 21.

HUNTER, J.; CRAWFORD, W. *Java Servlet Programming: Help for Server Side Java Developers*. [S.l.]: "O'Reilly Media, Inc.", 2001. Citado na página 35.

IEEE Xplore Digital Library. 2017. Disponível em: <http://ieeexplore.ieee.org/Xplore-/home.jsp>. Citado na página 45.

JADEJA, Y.; MODI, K. Cloud computing-concepts, architecture and challenges. In: IEEE. *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. [S.l.], 2012. p. 877–880. Citado na página 27.

JOHANSON, A. et al. Oceantea: Exploring ocean-derived climate data using microservices. NCAR, 2016. Citado na página 49.

KAKADIA, D. *Apache Mesos Essentials*. [S.l.]: Packt Publishing Ltd, 2015. Citado na página 40.

KAMBALYAL, C. 3-tier architecture. *Retrieved On*, v. 2, 2010. Citado na página 34.

KECSKEMETI, G.; MAROSI, A. C.; KERTESZ, A. The entice approach to decompose monolithic services into microservices. In: IEEE. *High Performance Computing & Simulation (HPCS), 2016 International Conference on*. [S.l.], 2016. p. 591–596. Citado na página 47.

KIVITY, A. et al. kvm: the linux virtual machine monitor. In: *Proceedings of the Linux symposium*. [S.l.: s.n.], 2007. v. 1, p. 225–230. Citado na página 29.

KRATZKE, N. About microservices, containers and their underestimated impact on network performance. *Proceedings of CLOUD COMPUTING*, v. 2015, 2015. Citado na página 40.

KURZE, T. et al. Cloud federation. *CLOUD COMPUTING*, Citeseer, v. 2011, p. 32–38, 2011. Citado na página 27.

LEVCOVITZ, A.; TERRA, R.; VALENTE, M. T. Towards a technique for extracting microservices from monolithic enterprise systems. *arXiv preprint arXiv:1605.03175*, 2016. Citado 2 vezes nas páginas 47 e 53.

LI, M.-L. et al. Understanding the propagation of hard errors to software and implications for resilient system design. In: ACM. *ACM SIGARCH Computer Architecture News*. [S.l.], 2008. v. 36, n. 1, p. 265–276. Citado na página 42.

LICHTIGSTEIN, A. *DevOps Kubernetes vs. Docker Swarm vs. Apache Mesos: Container Orchestration Comparison*. 2017. Disponível em: <https://www.loomsystems.com-/blog/single-post/2017/06/19/kubernetes-vs-docker-swarm-vs-apache-mesos-container-orchestration-comparison>. Citado na página 40.

LUO, J.-Z. et al. Cloud computing: architecture and key technologies. *Journal of China Institute of Communications*, China International Book Trading Corporation, P. O. Box 399 Beijing 100044 China, v. 32, n. 7, p. 3–21, 2011. Citado na página 26.

MALHOTRA, M.; TRIVEDI, K. S. Dependability modeling using petri-nets. *IEEE Transactions on reliability*, IEEE, v. 44, n. 3, p. 428–440, 1995. Citado na página 80.

MARSH, G. et al. Scaling advanced message queuing protocol (amqp) architecture with broker federation and infiniband. *Ohio State University, Tech. Rep. OSU-CISRC-5/09-TR17*, 2008. Citado na página 33.

MATTHEWS, J. N. et al. Quantifying the performance isolation properties of virtualization systems. In: ACM. *Proceedings of the 2007 workshop on Experimental computer science*. [S.l.], 2007. p. 6. Citado 2 vezes nas páginas 30 e 31.

MAURO, T. *Adopting Microservices at Netflix: Lessons for Architectural Design, 2015*. 2016. Disponível em: <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>. Citado na página 21.

MCKINNEY, W. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. [S.l.]: "O'Reilly Media, Inc.", 2012. Citado na página 76.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, Belltown Media, v. 2014, n. 239, p. 2, 2014. Citado na página 31.

MGDIS. 2017. Disponível em: <https://www.mgdis.fr/>. Citado na página 46.

MISHRA, A. K. et al. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, ACM, v. 37, n. 4, p. 34–41, 2010. Citado na página 21.

NAIK, N. Building a virtual system of systems using docker swarm in multiple clouds. In: IEEE. *Systems Engineering (ISSE), 2016 IEEE International Symposium on*. [S.l.], 2016. p. 1–3. Citado na página 40.

NAMIOT, D.; SNEPS-SNEPPE, M. On micro-services architecture. *International Journal of Open Information Technologies*, v. 2, n. 9, p. 24–27, 2014. Citado 2 vezes nas páginas 57 e 58.

NEWMAN, S. *Building microservices*. [S.l.]: "O'Reilly Media, Inc.", 2015. Citado 2 vezes nas páginas 22 e 36.

PAHL, C. Containerization and the paas cloud. *IEEE Cloud Computing*, IEEE, v. 2, n. 3, p. 24–31, 2015. Citado na página 40.

PAHL, C.; JAMSHIDI, P. Microservices: A systematic mapping study. In: *CLOSER (1)*. [S.l.: s.n.], 2016. p. 137–146. Citado na página 45.

PAPAZOGLOU, M. P.; HEUVEL, W.-J. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal—The International Journal on Very Large Data Bases*, Springer-Verlag New York, Inc., v. 16, n. 3, p. 389–415, 2007. Citado na página 22.

PATEL, P.; RANABAHU, A. H.; SHETH, A. P. Service level agreement in cloud computing. 2009. Citado na página 25.

PEGAHTECH. *Pegahtech Co.* 2017. Disponível em: <http://www.pegahtech.ir/>. Citado na página 48.

PENG, J. et al. Comparison of several cloud computing platforms. In: IEEE. *Information Science and Engineering (ISISE), 2009 Second International Symposium on*. [S.l.], 2009. p. 23–27. Citado na página 69.

PESSOA, G. et al. Qd framework: Using dynamic catalogs to organize mirror servers on scalable and resilient service meshes. In: ACM. *Proceedings of the 21st Brazilian Symposium on Multimedia and the Web*. [S.l.], 2015. p. 49–52. Citado na página 63.

PETRIE, H. et al. Augmenting icons for deaf computer users. In: ACM. *CHI'04 Extended Abstracts on Human Factors in Computing Systems*. [S.l.], 2004. p. 1131–1134. Citado na página 64.

PIVETTA, E. M.; ULBRICHT, V.; SAVI, R. Tradutores automáticos da linguagem português oral e escrita para uma linguagem visual-espacial da língua brasileira de sinais. *Artigo publicado em evento-Conapha-Pelotas–RS*, 2011. Citado na página 64.

POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, ACM, v. 17, n. 7, p. 412–421, 1974. Citado na página 28.

REITZ, K. *Python Requests: HTTP for Humans*. 2018. Disponível em: <http://docs-.python-requests.org/en/master/>. Citado na página 75.

RICHARDSON, C. Introduction to microservices. *URL: https://www.nginx.com/blog/introduction-to-microservices*. Citado na página 22.

RICHARDSON, C. Pattern: Microservices architecture. *Microservices. io. http://microservices. io/patterns/microservices. html*, 2014. Citado na página 22.

RODRIGUEZ, A. Restful web services: The basics. *IBM developerWorks*, 2008. Citado na página 67.

ROSTANSKI, M.; GROCHLA, K.; SEMAN, A. Evaluation of highly available and fault-tolerant middleware clustered architectures using rabbitmq. In: IEEE. *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. [S.l.], 2014. p. 879–884. Citado na página 33.

ROUSE, M. *Monolithic Architecture*. 2016. `http://whatis.techtarget.com/definition/monolithic-architecture`. Citado na página 34.

SCHROTH, C.; JANNER, T. Web 2.0 and soa: Converging concepts enabling the internet of services. *IT professional*, IEEE, v. 9, n. 3, 2007. Citado na página 21.

SHANG, L. et al. Extending yml to be a middleware for scientific cloud computing. In: SPRINGER. *IEEE International Conference on Cloud Computing*. [S.l.], 2009. p. 662–667. Citado na página 31.

SHEPLER, S. et al. Network file system (nfs) version 4 protocol. *Network*, 2003. Citado na página 70.

SHUKLA, R. T. A. *Architecting Reactive Applications on AWS*. [S.l.]: AWS re:Invent 2014, 2014. Citado na página 41.

SILL, A. The design and architecture of microservices. *IEEE Cloud Computing*, IEEE, p. 76–80, 2016. Citado na página 53.

SOLTESZ, S. et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: ACM. *ACM SIGOPS Operating Systems Review*. [S.l.], 2007. v. 41, n. 3, p. 275–287. Citado 2 vezes nas páginas 29 e 31.

SPRINGER Link. 2017. Disponível em: <https://link.springer.com/>. Citado na página 45.

STUBBS, J.; MOREIRA, W.; DOOLEY, R. Distributed systems of microservices using docker and serfnode. In: IEEE. *Science Gateways (IWSG), 2015 7th International Workshop on*. [S.l.], 2015. p. 34–39. Citado 2 vezes nas páginas 23 e 40.

TAVARES, O. d. L.; CORADINE, L. C.; BREDA, W. L. Falibras-mt–autoria de tradutores automáticos de textos do português para libras, na forma gestual animada: Uma abordagem com memória de tradução. In: *XXV Congresso da Sociedade Brasileira de Computação*. [S.l.: s.n.], 2005. p. 2099–2107. Citado na página 64.

THÖNES, J. Microservices. *IEEE Software*, IEEE, p. 116–116, 2015. Citado 2 vezes nas páginas 35 e 36.

TILKOV, S.; VINOSKI, S. Node. js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, IEEE, v. 14, n. 6, p. 80–83, 2010. Citado na página 67.

TRILOGIX, C. *Cloud Management Platforms: Software Applications, Integration, Packages, Orchestration*. 2017. Disponível em: <http://crmtrilogix.com/Cloud-Blog/Dev-Test-Optimisation/Cloud-Management-Platforms:-Software-Applications-Integration-Packages-Orchestration/177>. Citado na página 40.

UHLIG, R. et al. Intel virtualization technology. *Computer*, IEEE, v. 38, n. 5, p. 48–56, 2005. Citado na página 28.

VAQUERO, L. M.; RODERO-MERINO, L.; BUYYA, R. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, ACM, v. 41, n. 1, p. 45–52, 2011. Citado na página 35.

VAUGHAN-NICHOLS, S. J. New approach to virtualization is a lightweight. *Computer*, IEEE, v. 39, n. 11, 2006. Citado na página 30.

VERMA, A. et al. Large-scale cluster management at google with borg. In: ACM. *Proceedings of the Tenth European Conference on Computer Systems*. [S.l.], 2015. p. 18. Citado na página 40.

VIDELA, A.; WILLIAMS, J. J. *RabbitMQ in action*. [S.l.]: Manning, 2012. Citado 3 vezes nas páginas 56, 66 e 67.

VILLAMIZAR, M.; GARCÉS, O. et al. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: IEEE. *Computing Colombian Conference (10CCC), 2015 10th*. [S.l.], 2015. p. 583–590. Citado na página 39.

VILLEGAS, D. et al. Cloud federation in a layered service model. *Journal of Computer and System Sciences*, Elsevier, v. 78, n. 5, p. 1330–1344, 2012. Citado na página 27.

VINOSKI, S. Advanced message queuing protocol. *IEEE Internet Computing*, IEEE Computer Society, v. 10, n. 6, p. 87, 2006. Citado na página 32.

VOHRA, D. Scheduling pods on nodes. In: *Kubernetes Management Design Patterns*. [S.l.]: Springer, 2017. p. 199–236. Citado na página 76.

VURAL, H.; KOYUNCU, M.; GUNEY, S. A systematic literature review on microservices. In: SPRINGER. *International Conference on Computational Science and Its Applications*. [S.l.], 2017. p. 203–217. Citado na página 45.

XAVIER, M. G. et al. Performance evaluation of container-based virtualization for high performance computing environments. In: IEEE. *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. [S.l.], 2013. p. 233–240. Citado na página 31.