UNIVERSIDADE FEDERAL DE SANTA CATARINA

Análise Comparativa de Técnicas de Integração entre Microsserviços

Renato Pereira Back

Florianópolis - SC

2016/2

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE SISTEMAS DE INFORMAÇÃO

Análise Comparativa de Técnicas de Integração entre Microsserviços

Renato Pereira Back

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Frank Augusto Siqueira

Florianópolis - SC

2016/2

Renato Pereira Back

Análise Comparativa de Técnicas de Integração entre Microsserviços

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Frank Augusto Siqueira

Banca Examinadora

Prof. Fernando Augusto da Silva Cruz

Ivan Luiz Salvadori

Dedicatória:

À minha mãe, que sempre quis ver um filho formado na universidade e ao Tio Dão, Pedro Antônio da Rosa Medeiros (in memoriam), que com seus labirintos de Turbo Pascal feitos nas tardes chuvosas de sábados nos Açores fez despertar em mim o fascínio pela programação.

Agradecimentos:

Agradeço ao Professor Frank Augusto Siqueira, que soube entender minhas dificuldades temporais ao desenvolver este projeto. Aos meus pais, que sempre estimularam meus estudos, permitindo que eu chegasse ao final deste curso após tantas tentativas. Em especial à minha esposa Amanda, que por onze anos me incentivou a não largar os estudos e finalmente me graduar.

Sumário

Resumo

No mundo das aplicações distribuídas, Microsserviços são um assunto em evidência. O conceito existe há mais de dez anos, porém, apenas agora, após a popularização da computação na nuvem, dos contêineres, do amadurecimento das técnicas de integração e entrega contínuas e do renascimento da programação funcional, é que se torna cada vez maior o número de sistemas utilizando a arquitetura de microsserviços. O poder computacional obtido com essa abordagem traz consigo o aumento das responsabilidades que, se ignoradas, abre brechas para situações catastróficas. A integração dos microsserviços é uma dessas responsabilidades e, quando bem feita, propicia a autonomia, permitindo que os microsserviços sejam alterados e disponibilizados de forma independente do restante do sistema. Este trabalho de conclusão de curso avalia, de forma analítica, REST e AMQP como diferentes técnicas de integração entre microsserviços, mostrando as vantagens e desvantagens encontradas em cada uma delas, apresentando argumentos que permitam ao leitor escolher a abordagem mais adequada conforme o cenário apresentado. A análise foi feita com base numa aplicação hipotética, desconsiderando as regras de negócio e focando apenas na parte da integração dos serviços.

Palavras chave: Microsserviços, Integração, REST, AMQP.

Abstract

In the world of distributed applications, Microservices are a trending topic. The concept has been around for over ten years, however, only now, after concepts like cloud computing and containers became so popular, with the evolution of continuous delivery and integration techniques, as well as the rebirth of functional programming, is that the number of systems using the microservices architecture have been growing ever more. The computing power obtained with this approach brings with itself greater responsibilities that, if ignored, leave the gap open for catastrophic situations. Integrating microservices is one of such responsibilities and, when done right, enable autonomy, allowing microservices to be altered and deployed independently from the rest of the system. This final thesis evaluates, in an analytical way, REST and AMQP as different microservices integration techniques, showing the advantages and disadvantages found in each of them, presenting arguments that would allow the reader to choose the most appropriate approach according to the presented scenario. The analysis has been made based on a hypothetical application, ignoring business rules and focusing only in the services integration logic.

Keywords: Microservices, Integration, REST, AMQP.

1. INTRODUÇÃO

A Revolução Digital, simbolizada pela disseminação de computadores digitais e do armazenamento de dados na forma digital, estabeleceu o início do período histórico conhecido como Era da Informação. Um dos marcos deste período foi o surgimento e, posteriormente, a ampla adoção da Internet e da World Wide Web - WWW (daqui em diante denominada apenas Web). Na início da Web, o conteúdo fornecido era praticamente todo estático, como um repositório de documentos para serem acessados conforme sua URL. O surgimento de tecnologias e especificações como o *Common Gateway Interface* - CGI, permitiu que os servidores web fornecessem não apenas conteúdos estáticos, mas também conteúdos dinâmicos, gerados a partir de requisições recebidas via HTTP, dando origem a termos controversos, como: Web 1.0 referindo-se ao período dos conteúdos estáticos; Web 2.0, ou colaborativa, dito dos conteúdos gerados conforme interação com os usuários; e, posteriormente, Web 3.0, ou semântica, "a internet que sabe o que o usuário quer" (MORROW, 2014). No momento, estamos presenciando a explosão ou Internet das Coisas - IoT (sigla em inglês), chamada por alguns autores de Web 4.0, que envolve a comunicação de dispositivos diversos em constante comunicação com a rede, consumindo e gerando dados. Há relatos de que a Web 5.0 já está em desenvolvimento e que envolverá a conectividade, inteligência e emoção nos dados. Os termos são ditos controversos, pois a web já fora concebida com o intuito de ser colaborativa e dinâmica. Então, classificá-la com outros nomes é visto por muitos como sinônimos, ou apelidos, apenas para facilitar a compreensão de certas características da Web.

A evolução da Web teve repercussões em novas tecnologias de hardware, software e sistemas operacionais. Da mesma forma, metodologias e processos de

desenvolvimento foram evoluindo em paralelo para se adequar à nova realidade. A necessidade de atender um número cada vez maior de usuários e requisições tornou as arquiteturas de software do início da web obsoletas e insuficientes para atender esta nova realidade que exige sistemas responsivos e adaptáveis para incluir novas funcionalidades com baixo custo e maior velocidade de entrega, atender aumento de demanda de forma escalável e tornar os desenvolvedores mais produtivos e eficientes.

As primeiras técnicas para desenvolvimento de sistemas focavam em aplicações monolíticas, que consistiam de um único bloco agregador de funcionalidades, com grande necessidade de recursos humanos e computacionais, onde as maneiras para aumento da capacidade de atendimento consistiam no aumento de recursos no servidor, ou a replicação do sistema como um todo, tornando o processo caro e ineficiente.

Dentre as abordagens que surgiram para contornar essas dificuldades e problemas dos sistemas monolíticos, destacam-se os sistemas distribuídos, particularmente a Arquitetura Orientada a Serviços - SOA (sigla em inglês para *Service Oriented Architecture*). Porém, a falta de consenso em como aplicá-la de forma eficiente, as diferentes práticas adotadas pela indústria e as dificuldades técnicas encontradas pelos desenvolvedores, geraram obstáculos que diminuíram a adoção da SOA pela indústria. Foi justamente dessa realidade que os microsserviços emergiram. O conceito em si não é novo, porém um conjunto de fatores tecnológicos recentes proporcionou que os microsserviços surgissem, tendo a aceitação e a popularidade que estão tendo.

A análise realizada neste trabalho será feita com base na implementação de microsserviços com escopo limitado na aplicação MusicCorp, descrita por Sam Newman no livro *Building Microservices* (2015) e com foco na integração dos serviços, avaliando critérios de vazão, latência, complexidade de implementação, heterogeneidade tecnológica e atendimento aos conceitos que definem uma boa arquitetura de integração.

O resultado da análise proposta neste trabalho é de relevância para quem deseja adotar a arquitetura de microsserviços e busca mais informações sobre as técnicas de integração que deve adotar. Ao invés de propor um guia de regras, o objetivo é fornecer orientações, com base nos dados levantados, que auxiliem na escolha da arquitetura de integração.

1.1. PROBLEMA

A disseminação dos microsserviços foi beneficiada por fatores como os conceitos de entrega contínua, automação da infraestrutura, escalabilidade de sistemas, desenvolvimento ágil, concepção orientada ao domínio, do inglês *Domain-driven design* - DDD (EVANS, 2009). Entretanto, nem todos os conceitos são familiares ao programador normal: a modelagem do sistema, o comportamento do arquiteto de software, a arquitetura escolhida para integrações, o controle de versão das APIs, como testar, entregar, monitorar, assegurar e escalar os microsserviços são tarefas que exigem dedicação e aprimoramento contínuo para garantir melhores resultados.

A integração dos microsserviços é considerada um dos aspectos mais importantes da tecnologia a eles associados, pois, se bem feita, permite que seja mantida a sua autonomia e independência do sistema como um todo. A escolha da tecnologia de integração deve priorizar a prevenção de alterações que venham a desestabilizar o serviço. Desta mesma forma, desenvolver APIs que entreguem boa funcionalidade, usabilidade e forneçam boas experiências ao usuário (neste caso, desenvolvedores) está diretamente associado ao sucesso, adoção e a retenção de usuários. Outro foco inerente às boas práticas de integração é estar aberto a mudanças, pois esta é uma das únicas constantes quando o assunto é tecnologia: ela muda. A arquitetura das APIs também

deve ocultar dados internos de como ela foi implementada, ou seja, como num encapsulamento, isto deve ser imperceptível para quem consome o serviço, promovendo a diminuição do acoplamento entre os códigos (NEWMAN, 2015).

Fundamentado nestes quesitos, este trabalho de conclusão de curso focará especificamente na crucial etapa de integração dos microsserviços, propondo-se a analisar cenários para o uso de microsserviços e compará-los de modo a fornecer um embasamento sólido para a escolha da arquitetura de integração mais apropriada.

Desta forma, tem-se como pergunta de pesquisa "Qual técnica de integração apresenta um melhor retorno em desempenho, custo de desenvolvimento e consumo de recursos, dado um determinado conjunto de fatores?".

1.2. OBJETIVOS

1.2.1. Objetivo Geral

O objetivo geral deste trabalho é apresentar argumentos que balizem a decisão de um arquiteto de software quando este estiver escolhendo a técnica de integração de microsserviços.

1.2.2. Objetivos Específicos

Com o intuito de alcançar o objetivo geral, foram traçados objetivos específicos que direcionaram a pesquisa:

1. Revisar o referencial teórico sobre técnicas de desenvolvimento de microsserviços;
2. Elaborar métricas para comparar as diferentes técnicas de desenvolvimento;

3. Desenvolver um conjunto de microsserviços que permita a avaliação das métricas elegidas;
4. Comparar os resultados coletados a fim de verificar se é possível responder à pergunta de pesquisa.

1.2.3. Limitações da Pesquisa

Esta pesquisa será limitada pelas seguintes condições:

1. Os microsserviços desenvolvidos serão codificados em Java.
2. O conceito de dificuldade de desenvolvimento será considerado apenas com base no nível de conhecimento do autor.
3. As técnicas de integração avaliadas serão REST e AMQP.
4. Os cenários avaliados serão execução local e na plataforma Heroku.

1.3. METODOLOGIA

Este trabalho de conclusão de curso será desenvolvido com os seguintes critérios de pesquisa:

- Bibliográfica, sobre estudos existentes, visando ter conhecimento sobre o estado da arte em relação temas dos objetivos do trabalho;
- Experimental, através de testes com sistemas de microsserviços; e,
- Qualitativa, analisando métricas coletadas de cada técnica.

As etapas abaixo descrevem como serão aplicadas as metodologias de desenvolvimento deste trabalho:

1. Análise da literatura sobre microsserviços, técnicas de integração e métodos de comparação;
2. Definição das métricas de comparação e técnicas de integração que serão utilizadas;
3. Projeto e implementação de uma base de dados para armazenar os dados coletados das experiências;
4. Projeto e implementação dos microsserviços de modelo;
5. Execução dos testes em ambiente controlado;
6. Avaliação dos resultados obtidos com base nas métricas escolhidas.

1.4. ESTRUTURA DO TRABALHO

Este documento está dividido em sete capítulos, sendo o primeiro de carater introdutório, com definições do problema, apresentação dos objetivos geral e específicos, bem como as limitações da pesquisa, a metodologia adotada e a estrutura do trabalho.

Em seguida, no capítulo dois, inicia-se a fundamentação teórica, com revisão da literatura relacionada a sistemas distribuídos, aplicações web, SOA e microsserviços.

O capítulo três é composto de trechos e interpretações utilizados embasamento teórico para o assunto central do trabalho, a integração de microsserviços, discutindo estilos arquiteturais, interações entre sistemas e a análise de técnicas de integração.

No capítulo quatro é decorrido sobre a contextualização do escopo dos serviços utilizados na aplicação fictícia MusicCorp e como são definidas as métricas que serão analisadas durante a execução do experimento.

O quinto capítulo pode ser dividido em duas etapas. A primeira descreve a realização dos experimentos, listando as definições de lógica de negócio, integração,

cenários de execução, configuração dos projetos, formas de medição, implementação dos microsserviços. A segunda parte detalha os resultados obtidos nos cenários REST e AMQP nos cenários Local e Heroku. Ao final desta parte, é apresentada uma análise dos resultados obtidos.

O capítulo seis contém a conclusão do trabalho e a proposta de resposta para o problema apresentado no primeiro capítulo, bem como temas possíveis para trabalhos futuros.

Por fim, o sétimo capítulo lista as referências bibliográficas utilizadas na elaboração deste trabalho.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. SOBRE INTERNET E SISTEMAS DISTRIBUÍDOS

Originada de uma necessidade técnica da ARPANET - *Advanced Research Projects Agency Network*, do Departamento de Defesa dos EUA, onde criou-se um protocolo para comunicação entre redes, o *Internet Protocol* - IP, a Internet é um conjunto de redes de computadores interconectadas ao redor do mundo e que provê serviços como telefonia, redes *peer-to-peer*, correio eletrônico, documentos de hipertexto interconectados e aplicações da *World Wide Web* - WWW (HAFNER, LYON, 1999).

A WWW, também conhecida apenas como *Web*, foi criada em 1989 por Tim Berners-Lee como uma nova maneira de compartilhar e conectar informações, na forma de documentos eletrônicos, entre centros de pesquisa através de *hiperlinks*. Para dar suporte à Web, outros padrões foram definidos, como *HyperText Markup Language* - HTML, *HyperText Transfer Protocol* - HTTP, *Universal Resource Identifier* - URI, *Universal Resource Locator* - URL, Servidores Web, entre outros (McPHERSON, 2009).

Sistemas distribuídos, de acordo com Tanenbaum e Steen (2006), “são uma coleção de computadores independentes que aparentam aos usuários como um único e coerente sistema". Desta forma, este trabalho classifica a Internet como uma plataforma para a implementação e uso de sistemas distribuídos.

Em sua fase inicial, a Web era utilizada para compartilhar documentos estáticos, pois este era o conteúdo existente na época. Assim, com o aumento de sua adoção, outras funcionalidades foram surgindo e aproveitando a estrutura por ela fornecida. Alguns autores passaram a identificar cada conjunto de funcionalidades com números de versão, indicando que a Web estaria evoluindo. Entretanto, não há um consenso sobre o

assunto, havendo divergências entre os números das versões e as funcionalidades nelas existentes. O próprio criador da Web, Tim Berners-Lee é contraditório em suas visões, pois apesar de alegar que "Web 2.0" é apenas um jargão, ele denomina a Web Semântica como "Web 3.0". (WIKIPEDIA, 2016). Este trabalho utilizará a classificação demonstrada na **Tabela 1**, baseada nos conceitos de Fleerackers (2010).

VERSÃO	APELIDO	FUNCIONALIDADES
1.0	Estática	Leitura de documentos estáticos.
2.0	Colaborativa	Leitura e escrita de conteúdo dinâmico.
3.0	Semântica	Informação com contexto. Serviços Web.
4.0	Internet das Coisas	Tudo conectado à internet.
5.0	Simbiótica	Interação neurotecnológica.

Tabela 1: Versões da Web de acordo com Fleerackers

Embora não exista consenso na nomenclatura das versões da Web, a evolução do seu uso é evidente. A demanda pelos serviços fornecidos na WWW é cada vez maior, na medida em que cresce o número de usuários, sistemas e dispositivos alimentando e consumindo dados da Web. Dados da *International Telecommunications Union* indicam que o número de usuários na internet cresceu de 16 milhões de pessoas em 1995, para 3,336 bilhões em 2015. Com essa mudança de comportamento, empresas passaram a oferecer serviços através da Web na forma de conteúdo estático, com o que ficou conhecido como *Web Sites*, ou apenas *sites*.

2.1.1. Aplicações Web

Inicialmente, os *Web Sites* eram compostos de páginas estáticas escritas em HTML. Com o surgimento e adoção de tecnologias para geração de conteúdo dinâmico, como CGIs e Servlets entre outros, empresas passaram a identificar o potencial latente em um *Web Site* bem feito. Estudos de caso mostram um aumento de cerca de 34% no faturamento de uma empresa quando o *site* é bem feito (KIRKPATRICK, 2011). Desta maneira, do aperfeiçoamento da forma de entregar conteúdo estático e dinâmico emergiu o conceito de Aplicações Web.

Assim como as versões da Web, a diferença de significado entre Aplicações Web e *Web Sites* varia dentro da comunidade (BORODESCU, 2013; KURAMOTO, 2010; ROESTAMADJI, 2015; SHAPIRO, 2013), porém todas orbitam ao redor de conceitos que diferenciam *Web Sites* e Aplicações Web como:

- *Web Sites* são um conjunto de páginas, documentos, ou mídias (áudio, vídeo, imagens, etc) acessíveis através de navegadores web pela internet.
- Aplicações Web são sistemas que executam operações conforme requisições do usuário e também são acessíveis por navegadores web via internet.

A demanda pelo desenvolvimento de Aplicações Web botou à prova os paradigmas de programação e arquiteturas de sistemas existentes na época do surgimento da Web. Aplicações se tornaram grandes blocos de código agregadores de funcionalidades e conteúdo, inchadas e reféns do próprio tamanho. À esta abordagem em bloco deu-se o nome Arquitetura Monolítica.

De acordo com o padrão definido por Richardson (2014), aplicações desenvolvidas em um único bloco tendem a agregar as seguintes características iniciais:

- Desenvolvimento - as ferramentas atuais propiciam um ambiente adequado para desenvolver aplicações completas de modo simples;
- Deploy - a aplicação pode ser disponibilizada através de um único artefato;
- Escalabilidade - com um balanceador de carga, a execução de múltiplas cópias do sistema aumenta a capacidade de processamento das requisições.

Entretanto, na medida em que crescem, em tamanho de código e time de desenvolvimento, algumas obstáculos se formam:

- Linhas de código - em razão da constante evolução do sistema, na medida em que novas funcionalidades são adicionadas, é mais provável o aumento na quantidade de linhas de código;
- Manutenção - devido à quantidade de linhas de código, uma boa documentação é necessária para que se saiba onde cada alteração deve ser feita, do contrário, a atividade poderá ser mais difícil e propensa a postergação das modificações;
- Ferramentas de desenvolvimento - a quantidade de arquivos tende a exigir mais das ferramentas de desenvolvimento em funções como indexação de arquivos e mantê-los em memória;
- Servidores web - normalmente o *deploy* de aplicações é mais lento na medida em que o tamanho do arquivo aumenta;
- Escalabilidade - verticalmente, torna-se mais cara e limitada em função dos recursos computacionais. Já horizontalmente, nem sempre é possível, uma envolve codificação específica, além de haver desperdício, pois é necessário replicar toda a estrutura da aplicação;

- Tempo de reação à mudanças - alterar diferentes partes da aplicação por diferentes equipes é uma tarefa mais complexa e tende a gerar efeitos colaterais;
- Evoluções tecnológicas - análogo ao princípio da inércia, ou seja, quanto maior a "massa" do sistema, mais difícil alterá-lo.

Alternativas com foco na divisão do monólito surgiram visando contornar as limitações encontradas ao longo do uso prolongado dessa arquitetura. Dentre as soluções, podemos citar o uso de bibliotecas compartilhadas, a modularização e a arquitetura orientada a serviços.

Bibliotecas compartilhadas são uma maneira de dividir o código reutilizável de uma aplicação em arquivos isolados que podem ser distribuídos entre várias aplicações. De acordo com Newman (2015), o uso de bibliotecas, tanto particulares, quanto de terceiros, é prática comum na indústria por sua facilidade de adoção.

Modularização é uma técnica que permite o controle do ciclo de vida de um pedaço da aplicação. É como se uma biblioteca pudesse ser inicializada, interrompida e substituída em tempo de execução.

2.1.2. Arquitetura Orientada a Serviços

Também conhecida como *Service-Oriented Architecture* - SOA, é uma perspectiva da arquitetura de software que permite que um sistema distribuído seja flexível ao ponto de utilizar funcionalidades implementadas por diferentes equipes ou providas por terceiros, independentemente da tecnologia adotada.

Josuttis (2007) define SOA como um paradigma, ou conjunto de princípios, que guia a concepção e manutenção de processos de negócio inerentes a grandes sistemas

distribuídos. Desta forma, SOA não é uma solução pronta para ser implantada, mas algo que deve ser elaborado de acordo com o contexto e os requisitos de cada situação. Os princípios que capitaneiam SOA são preponderantemente baseados nos seguintes aspectos:

- Serviço - funcionalidade autocontida, simples ou complexa, que utiliza interfaces para conectar as necessidades do negócio à tecnologia de informação. O uso de interfaces formalmente definidas permite a interoperabilidade de forma heterogênea e encapsulada, independentemente da linguagem adotada.
- *Enterprise Service Bus* - ESB - é a infraestrutura que torna capaz a interoperabilidade entre os serviços, independentemente de plataformas e tecnologias.
- Baixo Acoplamento - como no conceito de engenharia de software, preza pela redução das dependências do sistema, resultando na redução da superfície de contato para efeitos de modificações, aumento na tolerância a falhas, escalabilidade e flexibilidade.

Para adotar uma arquitetura de sistemas distribuídos, uma empresa precisa ter um conjunto bem definido de papéis, políticas e processos permeado em todas as equipes que irão colaborar para entregar a solução final integrada. Esses conjuntos serão as ferramentas que conduzirão uma implementação bem sucedida da SOA.

2.1.3. Serviços Web

Se a SOA for considerada como um paradigma, Serviços Web são uma implementação de seus aspectos técnicos, ou seja, uma maneira de expor as funcionalidades de um sistema e disponibilizá-las através de recursos da Web.

A *World Wide Web Consortium* - W3C define Serviços Web como um sistema de software feito para apoiar interações entre máquinas através de uma rede e que deve possuir uma interface descrita em um formato processável por máquinas (no caso, WSDL). Outros sistemas interagem com um Serviço Web, na maneira detalhada em sua respectiva descrição, através de mensagens SOAP, normalmente transmitidas via HTTP com serialização de XML, em conjunto com outros padrões relacionados à Web.

Todavia, nem todos os problemas tecnológicos são resolvidos com essa abordagem, pois ainda não existem padrões maduros o suficiente para garantir a interoperabilidade dos sistemas. Junto disso, a natureza dos Serviços Web é incapaz de atingir o baixo acoplamento requerido pela SOA. De acordo com CHAPPELL (2003), as interações entre Serviços Web são dependentes de padrões e soluções proprietárias, onde cada grupo de empresas dissemina o seu próprio entendimento como um padrão. Esta guerra de padrões leva ao acoplamento entre serviços conforme as definições de determinados grupos de fabricantes específicos são utilizadas.

2.2. MICROSSERVIÇOS

Newman (2015) considera microsserviços como serviços pequenos e autônomos que funcionam em conjunto. Devem ser pequenos em razão do princípio de responsabilidade única (FOWLER, 2008), que mantém agrupadas funcionalidades que são alteradas juntas, e separadas aquelas que sofrem alterações por motivos diferentes. O tamanho da equipe que desenvolve os serviços também deve ser pequeno, pois

equipes menores são mais independentes, descentralizadas e responsivas, propiciando mais autonomia para evoluir sem ser afetado, nem afetar outras partes do todo.

Para Jones (2014), microsserviços são uma maneira de entrega de conteúdo orientada a serviços (*Service Oriented Delivery* - SOD) para uma solução bem arquitetada e granularizada de SOA.

Fowler e Lewis (2014) definem que o estilo arquitetural dos microsserviços aborda como desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e utilizando mecanismos leves de comunicação, geralmente uma API de recursos HTTP. Estes serviços são construídos em torno de competências de negócio e são disponibilizados independentemente por processos de *deploy* automatizados. Há um gerenciamento centralizado mínimo desses serviços, que podem ser escritos em diferentes linguagens de programação e utilizar distintas tecnologias de armazenamento de dados.

2.2.1. Microsserviços x SOA

"Microsserviços são SOA, para aqueles que sabem o que é SOA." - Steve Jones, 2014.

Uma das dificuldades de associar ou diferenciar os dois conceitos, é que a SOA em si possui mais do que uma definição, inclusive definições conflitantes sobre quais as reais características que devem ser seguidas (FOWLER, 2014). As diferenças nas definições da SOA são mais acentuadas em razão das implementações de ESB disponíveis no mercado, onde cada fabricante utilizou sua própria interpretação da SOA para conceber e distribuir seus produtos (ROTEM-GAL-OZ, 2014).

A abordagem dos microsserviços surgiu das experiências reais dos usuários de SOA, que utilizaram destes conhecimentos para eleger as boas práticas essenciais. Desta forma, é correto dizer que os microsserviços são um subconjunto da SOA, assim como XP e Scrum são abordagens específicas do desenvolvimento Ágil (NEWMAN, 2015).

Segundo Wolff (2016), tanto SOA quanto Microsserviços dividem aplicações através de serviços disponíveis em rede e as mesmas tecnologias podem ser utilizadas por ambas as abordagens. SOA busca flexibilidade corporativa em nível de TI através da orquestração dos serviços, sendo esta uma tarefa complexa e que não funciona muito bem quando os serviços precisam ser modificados. Em contrapartida, Microsserviços focam em projetos individuais e buscam permitir que se trabalhe e facilite a entrega de diferentes serviços paralelamente.

2.2.2. Benefícios

Dentre os benefícios obtidos com microsserviços, podemos citar (FOWLER, LEWIS, 2014; NEWMAN, 2015; WOLFF, 2016):

- Liberdade para substituição e composição de serviços;
- Flexibilidade para lidar com sistemas legados de modo sustentável;
- Facilidade de entrega;
- Entrega contínua;
- Escalabilidade sob medida;
- Robustez e resiliência através da prática de tolerância a falhas;
- Liberdade para escolha de tecnologias heterogêneas;
- Independência do restante do sistema;
- Alinhamento organizacional com projetos e equipes menores;

- Desenvolvimento paralelo de funcionalidades;

2.2.3. Princípios

Newman (2015) recomenda que, para um microsserviço oferecer todos os benefícios esperados, as seguintes características sejam seguidas:

- Baixo acoplamento: um serviço deve saber o mínimo necessário dos serviços com os quais ele interage, de modo a garantir que um serviço não sofra mudanças em razão de outros serviços, nem cause alterações em outros serviços devido a modificações em suas funcionalidades.
- Alta coesão: seguindo o Princípio de Responsabilidade Única, comportamentos associados devem ser agrupados, enquanto que comportamentos desconexos não devem ser reunidos. Assim, alterações não são propagadas.
- Delimitação do contexto: utilizando o conceitos de DDD de contextos delimitados, os microsserviços encapsulam informações que são pertinentes ao seu domínio, expondo através de interfaces apenas o que deve ser compartilhado com os consumidores.
- Regras de negócio: ao delimitar o contexto de cada serviço, suas próprias funcionalidades não devem ser voltadas aos dados compartilhados com o mundo exterior, mas sim às operações que o serviço realiza e aos dados dos quais ele precisa para isso.
- Comunicação condicionada aos conceitos de negócio: a comunicação entre os serviços deve obedecer o modo em que os contextos que eles representam se comunicam na vida real. Isto facilita a implementação de mudanças em decorrência de alterações nas regras de negócio.

3. INTEGRAÇÃO DE MICROSSERVIÇOS

Microsserviços têm que ser integrados e precisam se comunicar. Na opinião de Newman (2015), o aspecto mais importante da tecnologia associada com microsserviços é esclarecer os fatos sobre a integração. Se bem feita, a autonomia é mantida, do contrário as chances de ocorrerem problemas aumentam drasticamente.

3.1. PONTOS CHAVE

Para Newman (2015), uma boa integração pode ser obtida se certos pontos chave forem observados:

- Alterações que causam impactos nos serviços consumidores devem ser as mais raras possíveis;
- As APIs utilizadas para comunicação entre microsserviços não devem ser limitadas, nem associadas à funcionalidades específicas de determinadas tecnologias;
- A utilização do serviço deve ser simples para os usuários;
- Detalhes internos da implementação devem ser encapsulados;
- A arquitetura de eventos, ao invés de chamadas diretas, deve ser colocada em foco.

Todos os pontos chave mencionados giram em torno de um princípio fundamental para os microsserviços: o desacoplamento.

3.2. ESTILOS ARQUITETURAIS

Existem dois tipos de estilo arquitetural que podem ser utilizados para realizar a integração de microsserviços: o Coreografado e o Orquestrado (NEWMAN, 2015).

O estilo **Coreografado** é normalmente associado à programação orientada a eventos, onde cada serviço monitora determinadas situações e executa seu processamento caso uma determinada condição seja atendida. Este estilo é utilizado principalmente em sistemas assíncronos e responsivos.

Em contrapartida, o estilo **Orquestrado** gira em torno de um ponto de comando central, onde um serviço é responsável por realizar chamadas aos demais serviços e aguarda o retorno dos devidos processamentos. Esta abordagem é normalmente associada a sistemas síncronos e quebra o princípio do baixo acoplamento.

3.3. INTERAÇÃO COM CONSUMIDORES

Wolff (2016) classifica a integração dos serviços em três níveis: Interface com o Usuário; Lógica; e, Via Banco de Dados.

3.3.1. Interface com o Usuário

Para Tilkov (2014), cada serviço deve prover sua interface com o usuário, pois desta forma, a integração se dá através de links e proporciona a adoção de *dumb pipes, smart endpoints*, característica descrita por Fowler e Lewis (2014), onde sugerem que a lógica de negócio deve ser mantida separada das vias de comunicação.

Há casos onde a interface gráfica com o usuário - GUI (em inglês) precisa ser única. Nessas situações, o aconselhado é utilizar os microsserviços como módulos da aplicação. Assim, a GUI se comunicaria apenas com as interfaces dos módulos.

Uma alternativa para reduzir as dependências e simplificar a divisão dos microsserviços numa aplicação web é a Arquitetura do Cliente Orientada a Recursos - ROCA (do inglês *Resource Oriented Client Architecture*), que prioriza o uso apenas de HTTP e HTML. Entretanto, o uso de um Servidor de *Frontend*, ou *Asset Server*, para facilitar e padronizar a implementação das GUIs é desaconselhado, pois aumenta o acoplamento entre as interfaces dos microsserviços.

Interfaces com o Usuário para aplicativos *Mobile*, *Desktop* e *Rich Client*, que dependem da instalação de um software no cliente, apresentam grande risco para a autonomia dos microsserviços devido ao risco de aumento de acoplamento e dependências tecnológicas. Porém, a adoção de uma camada de *backend* favorece a autonomia dos serviços, pois permite isolar a lógica de negócio (WOLFF, 2016).

3.3.2. Lógica

Na classificação definida por Wolff (2016), a integração lógica é a maneira em que os microsserviços chamam uns aos outros para trabalhar em conjunto. Apresentam-se como integrações lógicas tecnologias como REST, SOAP, RPC e mensageria.

3.3.2.1. *Representational State Transfer* - REST

Comumente guiado pelo Modelo de Maturidade de Richardson (FOWLER, 2010), o REST é baseado no vocabulário HTTP (ex.: GET, POST, PUT) para representar as ações que podem ser executadas pelo microsserviço, caracterizando-se por:

- Suportar o uso de *cache* e balanceamento de carga;
- *Hypertext Application Language* - HAL permite a implementação do *Hypermedia as the Engine of Application State* - HATEOAS, que possibilita uma maior autonomia e flexibilidade dos microsserviços;
- Suportar diferentes formatos para transporte de dados, como XML, HTML, JSON e *Protocol Buffer*.
- Ter processamento síncrono em razão do HTTP.

Newman (2015) cita três pontos negativos para o uso de REST com HTTP: não há modo simples para gerar o esboço de um cliente para o serviço; alguns navegadores web não suportam todos os os verbos HTTP; e é suscetível a problemas de performance e latência de rede.

3.3.2.2. Remote Procedure Call - RPC

A RPC, sigla em inglês para chamada remota de procedimentos, é um mecanismo que permite a chamada de métodos em outros processos transparentemente, como se fosse uma chamada local (BIRRELL e NELSON, 1984). É mais utilizada atualmente na forma das tecnologias SOAP e Thrift.

O protocolo SOAP funciona através do envio mensagens que são convertidas no servidor para a chamada de um método em um objeto. O transporte das mensagens SOAP é bastante flexível, pois funciona sobre diferentes tecnologias, como HTTP, JMS e

SMTP/POP. A grande variedade de protocolos de transporte disponíveis para o SOAP tende a aumentar a complexidade na camada de comunicação, afetando a interoperabilidade dos serviços. Em geral, isso tem impacto no princípio de baixo acoplamento, pois a solução mais comum é a adoção de uma camada de coordenação de serviços (WOLFF, 2016).

Thrift é uma solução RPC oferecida pela Apache Software Foundation que utiliza de forma eficiente a codificação binária, semelhante ao Protocol Buffer, desenvolvido pelo Google. A interface dos microsserviços é desenvolvida num formato específico para o Thrift, permitindo que clientes e servidores de tecnologias diferentes comuniquem-se entre si. Além disso, Thrift pode encaminhar requisições de um processo para outro com outra linguagem de programação (WOLFF, 2016). O suporte poliglota oferecido pelo Thrift traz em contrapartida restrições na interoperabilidade entre os serviços em razão do acoplamento de tecnologias (NEWMAN, 2015).

3.3.2.3. Mensageria

Sistemas de Mensageria são *middlewares* que servem como uma terceira opção para a integração lógica dos microsserviços. Wolff (2016) destaca as seguintes características:

- Mensagens podem ser entregues para mais de um recipiente;
- Garantia de entrega de mensagens contorna a falha de comunicação;
- O controle de entrega pode validar o resultado do processamento e reenviá-lo no caso de falhas;
- Mensagens são processadas de forma assíncrona;

- O desacoplamento é facilitado, pois as mensagens não são enviadas diretamente para um recipiente; e,
- Oferecem suporte transacional sem necessidade de uma coordenação global.

Alguns dos serviços de mensageria disponíveis, são: AMQP, Apache Kafka, JMS, Redis, ATOM Feeds, entre outros. Assim como as técnicas de integração, cada uma dessas tecnologias tem suas vantagens e desvantagens.

O uso de mensageria vai contra uma das recomendações dos microsserviços que é manter as conexões "burras", pois transfere boa parte da lógica da aplicação para o middleware. Outra desvantagem é que a programação assíncrona torna o tratamento das ações mais complexo (NEWMAN, 2015).

O AMQP, sigla para *Advanced Message Queuing Protocol* é um padrão aberto de um protocolo de nível de aplicação para *middlewares* de mensageria. A especificação AMQP define que o formato utilizado para o envio dos dados pela rede é um *stream* de *bytes*. Desta forma, qualquer ferramenta pode criar e interpretar mensagens que sigam o formato definido pela especificação.

3.3.3. Banco de Dados

É a forma mais comum de integração entre serviços encontrada na indústria, onde todos os serviços acessam uma mesma base de dados, leem e escrevem informações conforme suas responsabilidades. Esta abordagem é muito simples de implementar, porém quebra os dois principais princípios dos microsserviços: forte coesão e baixo acoplamento (NEWMAN, 2015).

Para Wolff (2016), mesmo as alternativas que evitam a quebra desses princípios possuem desvantagens, pois aumentam a complexidade de desenvolvimento do microsserviço e gestão da base de dados. Por exemplo:

- Utilizar replicação de dados para prover autonomia aumenta as chances de conflito de *schemas*. Uma possibilidade é aplicar os conceitos de contextos delimitados para que cada microsserviço represente os dados da maneira que lhe for adequada;
- Replicar os dados aumenta as chances de redundância e inconsistência entre os dados que cada microsserviço armazena;
- Implementação da redundância precisa ser uma funcionalidade nativa do banco, para não aumentar a complexidade do microsserviço;
- Operações em lote podem aumentar a latência do microsserviço inicialmente, mas trazem como benefício a possibilidade de corrigir inconsistências nos dados. Uma alternativa é utilizar eventos para sinalizar o momento de executar a replicação dos dados.

3.4. ANÁLISE DE TÉCNICAS DE INTEGRAÇÃO

As integrações entre sistemas e serviços podem ocorrer das mais diferentes formas. Cada maneira possui suas vantagens e desvantagens. Assim, este trabalho visa analisar o REST e a AMQP (via RabbitMQ). com base nas métricas de vazão, latência, complexidade e heterogeneidade. Não foram encontradas referências na bibliografia que ditassem um conjunto de métricas para balizar a comparação entre técnicas de integração. Desta forma, optou-se por coletar dados de vazão, latência, complexidade e

heterogeneidade por serem dados simples de coletar, contanto expressivos para comparar.

O referencial teórico nos mostra que ambas as técnicas de integração apresentam diferentes soluções para os mesmos tipos de problemas e se propõe a atender às demandas de integração de serviços tanto para estilos arquiteturais coreografados, quanto orquestrados.

Ambas as técnicas de integração necessitam de uma funcionalidade externa para gerenciar um estilo arquitetural coreografado. No caso do REST, é necessário um serviço como o ATOM *Feeds*, ou *Push Notifications*, por exemplo. Para o AMQP, é preciso que exista um *message broker* servindo como um ponto central para receber, armazenar e entregar as mensagens relacionadas aos eventos.

Caso seja optado por um estilo arquitetural orquestrado, a abordagem com REST é um pouco mais simples, pois não é necessário utilizar um serviço externo para distribuir as mensagens. Em contrapartida, para o AMQP, sempre é necessário que haja um *message broker*.

4. CONTEXTUALIZAÇÃO DO ESCOPO

Para realizar a análise das técnicas de integração, será utilizado o contexto e especificação de um sistema de vendas online de CDs para a empresa fictícia MusicCorp, descrito no livro *Building Microservices*, de Sam Newman (2015).

O domínio do sistema que a MusicCorp utiliza abrange toda a lógica de negócio e operacional que a empresa necessita para comercializar seus produtos e operacionalizar suas atividades internas, desde o almoxarifado à recepção, do departamento financeiro ao setor de pedidos.

4.1. ESTRUTURA DO SISTEMA MUSICCORP

Consideramos o departamento financeiro e o almoxarifado como dois contextos delimitados separados, onde ambos possuem detalhes internos que são disponíveis apenas dentro de cada contexto, bem como interfaces explícitas para acessos externos, utilizadas para comunicação formal com outros domínios. Os setores do sistema são os seguintes:

- *Catalog*

  Tudo o que diz respeito aos metadados dos itens à venda

- *Finance*

  Relatórios de contas, pagamentos, reembolsos, etc.

- *Warehouse*

  Envio e recebimento de pedidos de clientes, gestão ítens do inventário, etc.

- *Recommendation*

  Sistema de alta complexidade para recomendação de itens ao consumidor.

Apesar dos setores terem escopos separados, eles precisam trocar informação. Para isso, utiliza-se um modelo compartilhado com os contextos delimitados de cada setor e um conceito de objeto compartilhado, utilizado para que os setores se comuniquem. Parte do modelo da aplicação se encontra representado na **Imagem 1**, exemplificando o compartilhamento de dados entre o setor *Warehouse* e o *Finance*.

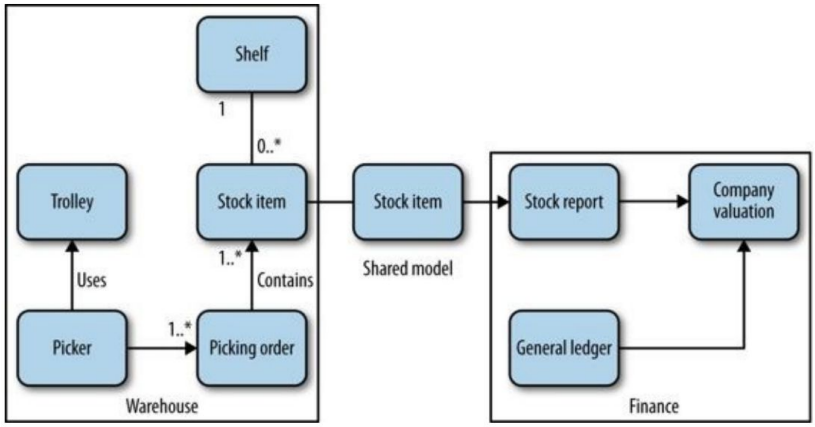


Imagem 1: Modelo compartilhado entre o departamento financeiro e o almoxarifado

4.2. ESCOPO DA ANÁLISE

Para a definição do escopo do trabalho, realizarmos os experimentos e, por fim, a análise das técnicas de integração, utilizamos a funcionalidade de Cadastro de Consumidor descrita na aplicação da MusicCorp, conforme ilustrado na **Imagem 2**.

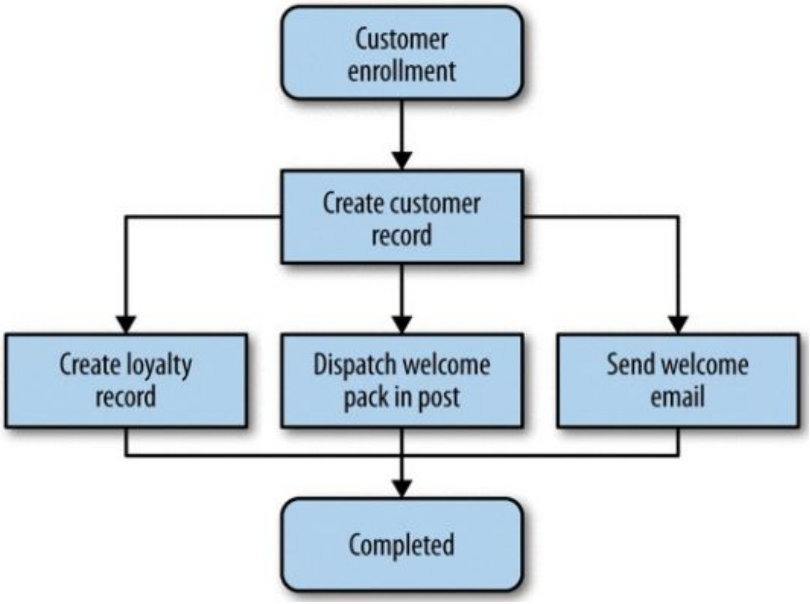


Imagem 2. Processo para criação de um novo Consumidor (NEWMAN, 2015)

A funcionalidade descrita na **Imagem 2** é realizada por quatro serviços:

- *Customer Service*

  Gerencia e inicializa a criação do consumidor

- *Loyalty Service*

  Responsável pela associação do consumidor com o serviço de fidelização de clientes

- *Post Service*

  Gerencia o envio de pacotes ao consumidor

- *Email service*

  Controla o envio de emails ao consumidor

Em detalhes, o início do processo se dá com o registro do Consumidor, realizado pelo Customer Service. Após a processamento do mesmo, os serviços Loyalty Service,

Post Service e Email Service devem realizar seus processamentos para criação do programa de fidelização, envio do pacote de boas vindas e envio de email de boas vindas, respectivamente. A execução de todas essas tarefas marca como concluído o processo em sua íntegra.

Para este trabalho de conclusão de curso, avaliar-se-ão apenas as técnicas de integração entre os serviços. Desta forma, não será feita a implementação da lógica de negócio de cada serviço, mas apenas as interfaces de comunicação (recebimento e chamada de mensagens) utilizando REST e AMQP como diferentes técnicas de integração.

4.2.1. Métricas Analisadas

Este trabalho analisará os seguintes critérios nas diferentes técnicas de integração:

- Vazão de Requisições

  Consiste na quantidade de requisições que os serviços conseguem realizar.

- Latência dos Serviços

  Identificado pelo tempo de resposta de cada serviço para iniciar o processamento após o recebimento da requisição.

- Facilidade de Implementação

  Leva em consideração a documentação disponível, bem como a complexidade para implementar cada serviço.

- Índice de Heterogeneidade de Linguagens de Programação

  Medido pela quantidade de diferentes linguagens que podem se comunicar com o serviço.

5. REALIZAÇÃO DOS EXPERIMENTOS

Para a implementação dos serviços, foi utilizada a linguagem de programação Java (http://www.java.com) na versão **JDK 1.8.0_45-b14**, em conjunto com o Spring Boot (http://projects.spring.io/spring-boot/) na versão **1.4.2**.

As técnicas de integração avaliadas foram o REST e RabbitMQ como implementação do AMQP. Nos testes com REST, utilizou-se o estilo arquitetural orquestrado, com um serviço coordenando as chamadas. Para os testes com AMQP, o broker ficou responsável pelo registro e descoberta dos serviços.

A IDE utilizada para codificação dos serviços foi o Spring Tool Suite - STS (https://spring.io/tools) na versão **3.8.2.RELEASE**.

Uma das funcionalidades providas pelo Spring Boot é que o artefato gerado após a compilação é um arquivo JAR auto executável e que carrega todas as suas dependências.

Para os testes remotos, utilizou-se o serviço de *Platform as a Service* Heroku: *Cloud Application Platform* (http://www.heroku.com) para realizar os testes com serviços remotos. O Heroku também disponibiliza *add-ons* com o CloudAMQP, uma versão online e gratuita do RabbitMQ para servir de broker AMQP.

Para a chamada inicial dos serviços e execução dos testes, foi utilizado **Postman** (http://www.getpostman.com) para realizar requisições HTTP que simulam o cadastro de um Consumidor.

O pré-processamento dos logs foi feito com a ferramenta Sublime Text 3 (http://www.sublimetext.com).

5.1. LÓGICA DE NEGÓCIO

A primeira etapa do experimento foi analisar a real necessidade de desenvolvimento de projetos que representassem a lógica de negócio dos serviços *Customer Service*, *Loyalty Service*, *Post Service* e *Email Service*. Por não terem relação com o escopo deste trabalho, o de analisar as técnicas de integração, optou-se pela não implementação de projetos com lógica de negócio. Desta forma, não há projetos que realizam processamento propriamente dito, apenas os módulos relacionados à comunicação nas diferentes técnicas de integração.

5.2. LÓGICA DE INTEGRAÇÃO

Em seguida, os projetos responsáveis pela integração entre os serviços foram desenvolvidos, sendo divididos conforme a responsabilidade do serviço e a tecnologia analisada.

A complexidade para o estilo arquitetural coreografado em REST atualmente é maior, pois a infra-estrutura para registrar e descobrir os serviços, tratar as requisições e garantir sua entrega não possui instruções ou padrões de fácil utilização. Além disso, serviços como o ATOM *Feeds* não fornecem funcionalidades simples e básicas, como a garantia de entrega e leitura exclusiva, que já são padrão para filas no AMQP. Desta forma, os projetos foram implementados com o estilo arquitetural orquestrado. Para a análise do REST, temos:

- customer-service-rest (http://github.com/tioback/customer-service-rest)
- loyalty-service-rest (http://github.com/tioback/loyalty-service-rest)
- post-service-rest (http://github.com/tioback/post-service-rest)

- email-service-rest (http://github.com/tioback/email-service-rest)

Diferentemente do REST, para a análise da mensageria, focou-se no *Advanced Message Queuing Protocol* (AMQP) e escolheu-se o *broker* RabbitMQ (http://www.rabbitmq.com) para implementar, centralizar e distribuir as operações de maneira bem trivial. Portanto, o estilo arquitetural adotado foi o coreografado. Os projetos são:

- customer-service-mq (http://github.com/tioback/customer-service-mq)
- loyalty-service-mq (http://github.com/tioback/loyalty-service-mq)
- post-service-mq (http://github.com/tioback/post-service-mq)
- email-service-mq (http://github.com/tioback/email-service-mq)

5.3. CENÁRIOS DE EXECUÇÃO

Os cenários escolhidos para execução dos testes foram:

1. Todos os serviços rodando localmente na mesma máquina
2. Serviços rodando na plataforma Heroku através do *addon* **CloudAMQP**, no plano *Little Lemur*.

Para a mensageria, o serviço do RabbitMQ foi executado localmente no cenário 1, enquanto que no cenário 2 foi executado no Heroku.

Todos os testes seguiram os seguintes critérios:

- **Repetições** - quantas vezes um teste foi repetido.
- **Intervalo** - quanto tempo uma thread realiza requisições.
- **Quantidade** - quantas threads simultâneas realizam requisições.

- **Pausa** - que cada thread faz antes de realizar outra requisição.

5.4. CONFIGURAÇÃO DOS PROJETOS

Os projetos foram configurados para que utilizarem endereços padrão na ausência de uma parametrização durante a inicialização do serviço. Desta forma, a troca dos endereços não exige uma nova recompilação do projeto.

Nos testes com REST, utilizou-se uma funcionalidade do *Spring Boot* que são os *profiles*. Quando o profile *heroku* era informado, o sistema ativava a configuração que buscava pelos serviços nas URLs **http://<NOME DO SERVIÇO>.herokuapps.com/**. Caso não fosse fornecido o *profile*, o sistema opera utilizando portas distintas e pré-definidas para cada microsserviço.

Para os testes com AMQP também foi utilizada a funcionalidade dos *profiles*, mas de uma maneira diferenciada. Ao invés de passar o endereço dos serviços, o sistema buscava pelo parâmetro **CLOUDAMQP_URL** disponibilizado pelo Heroku quando é instalado o *addon* CloudAMQP. Caso o parâmetro não fosse encontrado, o serviço carregava as configurações padrão do RabbitMQ, conforme demonstrado na **Imagem 3**.

```
@Profile("heroku")
@Bean(name = "connectionFactory")
public ConnectionFactory herokuConnectionFactory() {
    final URI ampqUrl;
    try {
        ampqUrl = new URI(getEnvOrThrow("CLOUDAMQP_URL"));
    } catch (URISyntaxException e) {
        throw new RuntimeException(e);
    }

    logger.error("User AMQP config: " + ampqUrl);

    final CachingConnectionFactory factory = new CachingConnectionFactory();
    factory.setUsername(ampqUrl.getUserInfo().split(":")[0]);
    factory.setPassword(ampqUrl.getUserInfo().split(":")[1]);
    factory.setHost(ampqUrl.getHost());
    factory.setPort(ampqUrl.getPort());
    factory.setVirtualHost(ampqUrl.getPath().substring(1));

    return factory;
}
```

Imagem 3: Configuração dos projetos com *profiles* do Spring Boot para AMQP.

Uma das limitações encontradas ao executar os testes REST localmente foi o impacto causado na máquina utilizada. Dependendo do sistema operacional, diferentes configurações adicionais são necessárias para o correto funcionamento dos serviços sem a incidência de erros. A **Tabela 2** detalha as limitações por sistema operacional.

Sistema Operacional	**Limite de Threads**
64-bit Mac OS X 10.9	2.030
64-bit Ubuntu Linux	31.893
64-bit Windows 7	Ilimitado

Tabela 2: Limites dos Sistemas Operacionais para threads ativas por processo (PLUMBR, 2016)

Considerando as limitações dos sistemas operacionais, o Windows foi o que apresentou melhores condições para realização dos testes sem necessidade de configurações ou codificação específica. Entretanto, com o intuito de controlar o escopo dos testes, optou-se por realizar alterações no projeto de modo a padronizar a execução tanto no Linux, quanto no MacOS X e no Windows.

- **Memória**

Optou-se limitar em 256 MB a memória disponibilizada para a JVM de cada serviço, pois esta é a quantidade disponibilizada pelo Heroku em sua camada grátis e ao mesmo tempo permite executar o teste localmente em máquinas com menos recursos sem competir com processos do sistema operacional. Para limitar a quantidade de

memória utilizada pela JVM, adicionou-se o parâmetro **-Xmx256m** ao script de inicialização dos serviços.

- **Threads**

O uso de requisições assíncronas no REST provou utilizar muitas threads por padrão, extrapolando os limites pré definidos de fábrica do Linux e Windows. Este problema foi minimizado ao rodar os serviços remotamente, pois a escalabilidade obtida com o uso de outras máquinas proporcionou mais recursos para a execução dos testes. Contudo, se aumentarmos a quantidade de threads realizando requisições, mesmo o cenário remoto é comprometido. Por padrão, o Spring Boot utiliza o Apache Tomcat como servidor web e este vem configurado para permitir que 200 threads sejam criadas para tratar requisições, drenando os recursos da máquina quando os serviços são executados ao mesmo tempo. Para permitir que os testes rodassem adequadamente, reduziu-se este número para 75 threads através da configuração do Spring Boot **server.tomcat.max-threads**.

Outro impacto causado pela limitação de threads foi no modo padrão que o Spring Boot adota para gerenciar as requisições. Internamente, a biblioteca utiliza um cache de threads que, para cada requisição recebida, reutiliza threads inativas, ou cria uma nova thread caso não haja alguma disponível no pool. Esta configuração permite que o pool cresça indefinidamente, podendo extrapolar os limites impostos pelo SO e levando a erros de execução.

Dentre as alternativas para contornar a limitação das threads, optou-se por trocar a biblioteca de comunicação HTTP. As bibliotecas avaliadas foram a **async-http-client** e a **Apache HttpComponents HttpClient**. Após experimentos, a segunda mostrou-se mais adaptável e permitiu a configuração de um **java.util.concurrent.ExecutorService** para controlar o número máximo de threads e conexões HTTP ativas e inativas simultaneamente. Este controle foi feito através do uso de um

java.util.concurrent.ThreadPoolExecutor com capacidade do pool igual ao dobro da quantidade de threads simultâneas configuradas por teste.

- **Sockets**

Com o aumento do número de threads ativas realizando requisições simultaneamente, outro limite do sistema operacional foi atingido, foi a quantidade de arquivos/sockets abertos por sessão. Este problema fica mais evidente na medida em que as threads demoram para processar, seja porque estão bloqueadas por I/O esperando por resposta do servidor, ou porque aguardam tempo do processador para realizar suas operações. Quanto mais se aumentou o intervalo dos testes e a quantidade de threads, mais esse problema se mostrou presente. Desta forma, para realizar testes com um maior número de threads, foi preciso diminuir o intervalo de duração dos testes, enquanto que para testar numa maior duração de tempo, foi preciso diminuir a quantidade de testes.

5.5. FORMAS DE MEDIÇÃO

Os quesitos analisados foram: vazão de requisições, latência dos serviços, facilidade de implementação e índice de heterogeneidade de linguagens.

Para medir a **vazão**, foram contadas quantas requisições cada serviço conseguiu processar no intervalo de um minuto.

Para os testes com AMQP, o monitoramento ocorreu através do mecanismo de confirmação. Em resumo, cada vez que uma mensagem é enviada para o *broker*, este devolve uma confirmação de entrega para o serviço que produziu (produtor) a mensagem sempre que ela é processada por outro serviço (consumidor). Para cada confirmação recebida, incrementava-se o contador de vazão.

Para os testes com REST, por serem requisições assíncronas e orquestradas, optou-se por usar a programação orientada a aspectos, através do **AspectJ** (https://eclipse.org/aspectj), para monitorar de forma ortogonal o envio de requisições e recebimento das respostas. Assim, sempre que uma resposta era retornada, o contador de vazão era incrementado.

A **latência** foi considerada após a análise dos logs de todos os serviços e avaliado quanto tempo levou para que todos sinalizassem o recebimento. Nos testes com REST, a latência foi medida pelo intervalo de tempo entre o envio da requisição e o retorno da resposta. Nos testes com AMQP, a medição ocorreu com o uso de mensagens de confirmação enviadas pelo broker ao produtor quando uma mensagem é processada por um consumidor. Foi considerado, então, o intervalo de tempo entre o envio da mensagem de criação do usuário e a confirmação enviada pelo broker .

Tanto para as medições de vazão, quanto de latência, tomou-se como medida de precaução o uso de um identificador para cada iteração. Assim, requisições de iterações anteriores não foram contabilizadas de forma errada. Em trabalhos futuros, essas requisições descartadas podem constituir uma nova métrica de avaliação.

Para gerar os dados necessários para a análise comparativa das técnicas de integração, cada serviço imprime no log mensagens específicas, como as exibidas na **Imagem 4**, que posteriormente foram analisadas manualmente e inseridas em uma planilha para análise. Nos testes locais, o log era extraído do próprio console do STS e tratado no Sublime com o auxílio de expressões regulares para remover dados indesejados.

```
[STAT]-[Thread][Pausa][Registros][Latência]:   5   15  465 521436
[STAT]-[Thread][Pausa][Registros][Latência]:   5   15  885 563598
[STAT]-[Thread][Pausa][Registros][Latência]:   5   15  2800    455876
[STAT]-[Thread][Pausa][Registros][Latência]:   5   25  1730    814758
[STAT]-[Thread][Pausa][Registros][Latência]:   5   25  1650    373397
[STAT]-[Thread][Pausa][Registros][Latência]:   5   25  675 656792
[STAT]-[Thread][Pausa][Registros][Latência]:   5   25  1716    631342
```

Imagem 4: Exemplo de log gerado pela aplicação

Para cada iteração, foi feita a média aritmética simples e, ao final, o resultado era impresso no log. O resultado do log então foi limpo com o uso de expressões regulares no Sublime. Posteriormente, os valores tratados foram então adicionados à uma planilha no Google Sheets (http://sheets.google.com), onde houve o processamento estatístico com o cálculo dos quartis, variância, desvio padrão e distribuições. Enfim, elaboraram-se gráficos com o resultado dos dados estatísticos, conforme pode-se ver na **Imagem 5**.

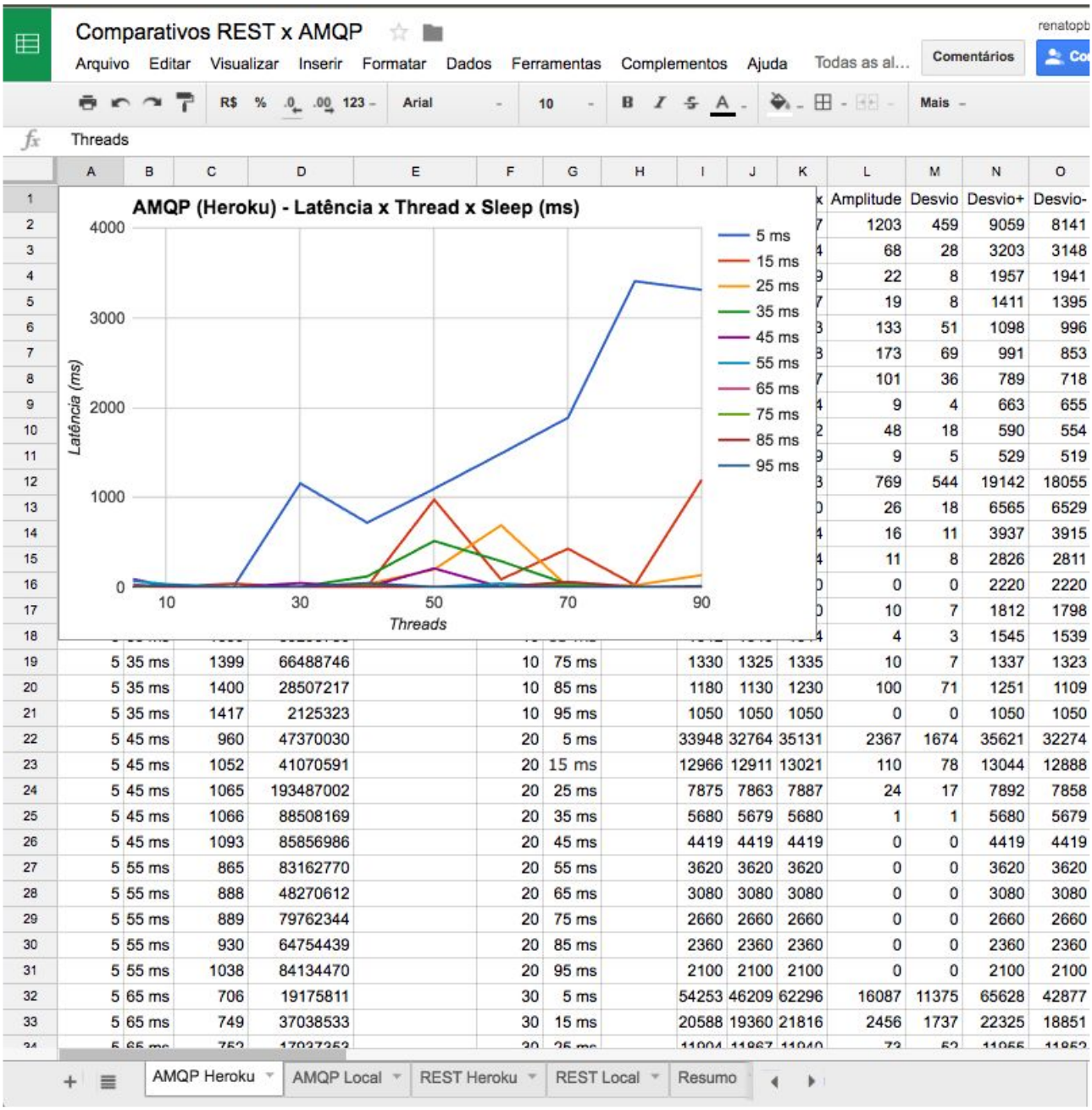


Imagem 5: Exemplo de dados processados no Google Sheets.

A **complexidade** ou facilidade de implementação levou em conta a experiência do autor, bem como o material disponível na literatura e na internet que desse apoio ao desenvolvimento dos serviços.

O índice de **heterogeneidade** levou em conta a informação disponível na documentação das ferramentas utilizadas.

5.6. IMPLEMENTAÇÃO DOS SERVIÇOS

O *Spring Boot* oferece muitas facilidades de implementação dos serviços, o que permite que os desenvolvedores foquem diretamente na lógica do negócio. Entretanto, algumas das integrações fornecidas por ele ainda são carentes de documentação.

A qualidade e quantidade de documentação disponível na forma de tutoriais e exemplos teve impacto direto no entendimento de complexidade de implementação dos serviços. Para os testes serem realizados, é necessário ter todas os serviços funcionando corretamente, inclusive os de terceiros, como o ATOM Feed e o RabbitMQ.

Desenvolver os projetos REST foi muito simples e o *Spring Boot* facilitou muitos dos aspectos do mapeamento das chamadas. Entretanto, a parte que levaria a um estilo arquitetural coreografado exige um maior conhecimento de *Polling* (busca ativa por modificações) e ferramentas que façam isso de forma passiva (*WebSocket*, *Streaming*, entre outros). Apesar de existir a possibilidade de integrar REST com outras técnicas (JMS, por exemplo), optou-se por não o fazer, limitando o foco da análise em uma única técnica.

Utilizar o Heroku como plataforma remota de testes auxiliou muito no experimento, pois permitiu que os testes fossem feitos sem a necessidade de adquirir ou realocar maquinário. A curva de aprendizado para utilizar a plataforma é suave e não tem grandes impactos no desenvolvimento. Além disso, mesmo no nível gratuito, a plataforma oferece opções de escalabilidade, administração logs e addons com diversas funcionalidades.

5.7. RESULTADOS OBTIDOS

5.7.1. REST - Local

Conforme se pode observar na **Imagem 6** , o número de registros segue crescendo conforme o número de requisições feitas pelas threads. Observa-se, também, que o tempo de pausa entre requisições impacta na ordem inversa o número de registros criados. Chamou a atenção a oscilação do número de registros cadastrados conforme o número de threads, pois esperava-se que o valor crescesse linearmente. A **Imagem 7** mostra o gráfico das latências encontradas durante os testes locais com REST. Observa-se picos esporádicos de latência, conforme se observa no histograma representado pela **Imagem 8**. Não obstante, os tempos médios de resposta foram em torno de 78,98 ms.

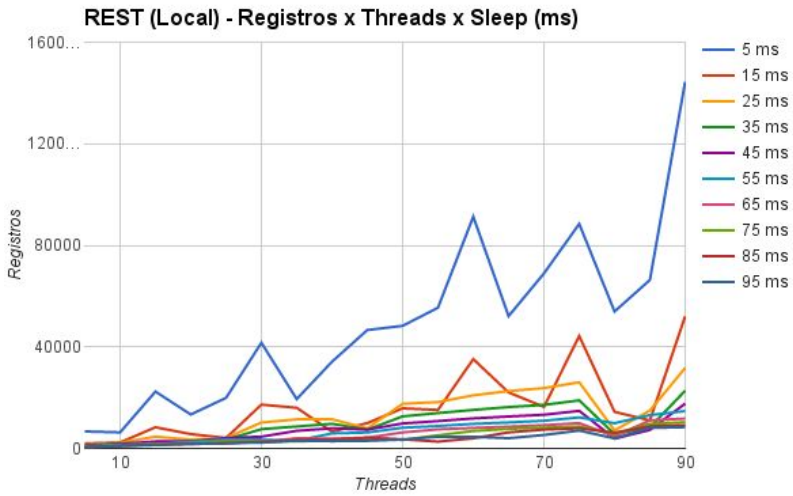


Imagem 6: Resultado do total de registros cadastrados no teste local com REST em relação ao número de threads ativas e a pausa entre cada requisição.

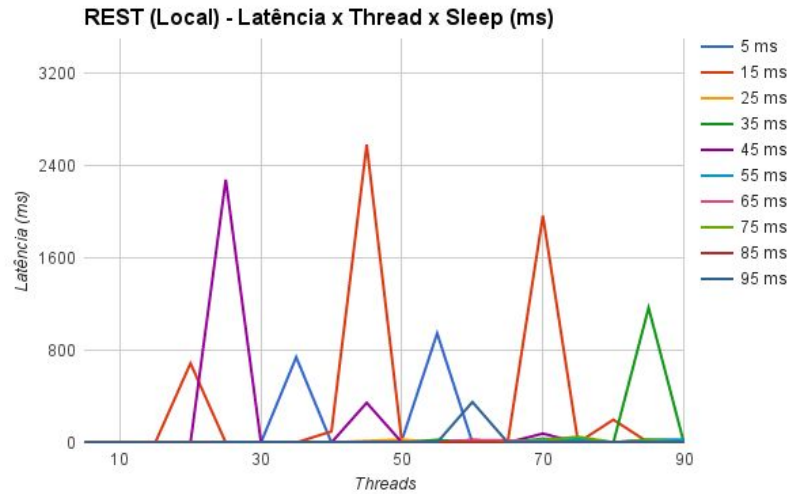


Imagem 7: Latência média do processamento das requisições no teste local com REST em relação ao número de threads ativas e a pausa entre cada requisição.

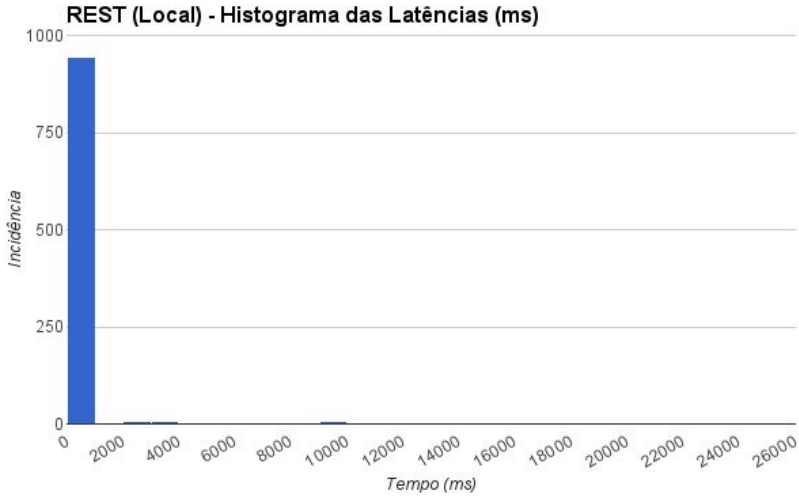


Imagem 8: Histograma das latências no teste REST realizado no ambiente local.

5.7.2. REST - Heroku

Inicialmente não fora possível executar os testes no Heroku, pois por se tratar de um plano gratuito, as limitações de memória e processamento da máquina não suportaram a execução dos testes, resultando em erros do servidor que interrompiam o serviço e os testes. Porém, com os controles de memória e threads adotados, foi possível executar os testes e extrair os resultados.

A **Imagem 10** ilustra a quantidade de registros cadastrados durante os testes. Assim como nos testes locais, nota-se uma aumento representativo na quantidade de requisições realizadas na medida em que se aumenta o número de threads e diminui o tempo de pausa. Observa-se, no entanto, como a limitação dos recursos disponíveis na versão gratuita do Heroku impactam no desempenho do sistema quando há uma maior necessidade de processamento (muitas threads, pouca pausa).

As latências medidas na plataforma Heroku e representadas na **Imagem 11**, apresentam comportamento semelhante ao identificado nos testes realizados no ambiente local. Os picos esporádicos de latência aumentada são mais frequentes, principalmente

nas iterações com menor tempo de pausa (5 ms). Apesar disso, a média da latência foi de 79,27 ms, muito semelhante aos testes com rest no cenário local. A distribuição dessas incidências pode ser vista na **Imagem 12**.

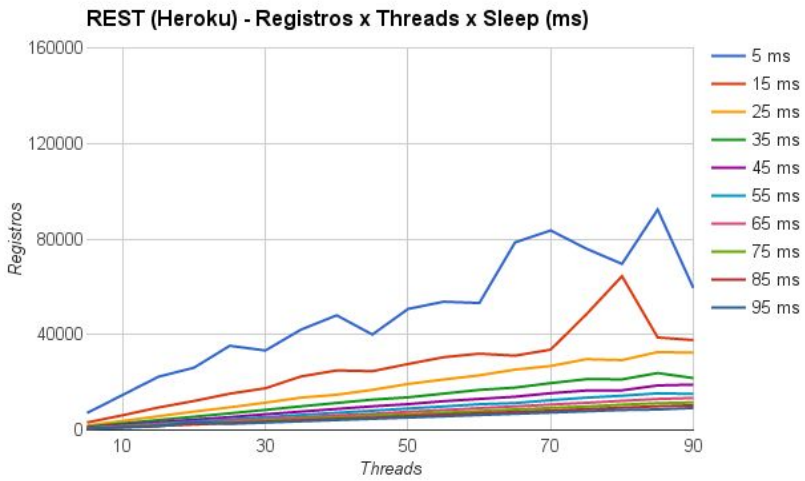


Imagem 10: Resultado do total de registros cadastrados no teste com REST na plataforma Heroku em relação ao número de threads ativas e a pausa entre cada requisição.

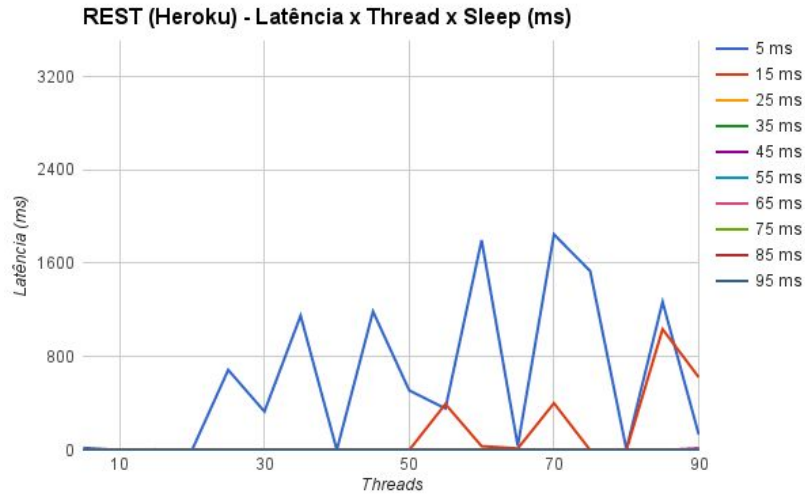


Imagem 11: Latência média do processamento das requisições no teste na plataforma Heroku com REST em relação ao número de threads ativas e a pausa entre cada requisição.

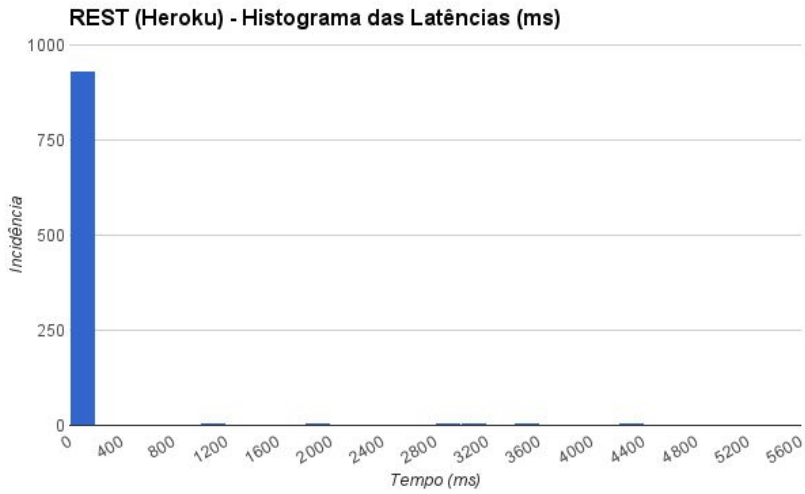


Imagem 12: Histograma das latências no teste REST realizado na plataforma Heroku.

5.7.3. Mensageria - Local

Conforme se pode observar na **Imagem 13** , o número de registros cadastrados com o uso da mensageria apresentou resultados mais constantes e lineares. Apesar de o número máximo de requisições realizadas com sucesso ser muito próximo do obtido nos testes com REST local, observou-se que a quantidade total de registros processados foi cerca de 24,66% maior nos testes com AMQP. A **Imagem 14** mostra o gráfico das latências mensuradas, diferentemente dos testes com REST, a oscilação parece mais evidente, porém é preciso considerar que a amplitude é ínfima, se comparada aos demais cenários de teste. O tempo médio de resposta foi bem inferior aos obtidos nos outros experimentos, 1,57 ms. A distribuição demonstrada pelo histograma na **Imagem 15** também exibe uma maior variação das latências.

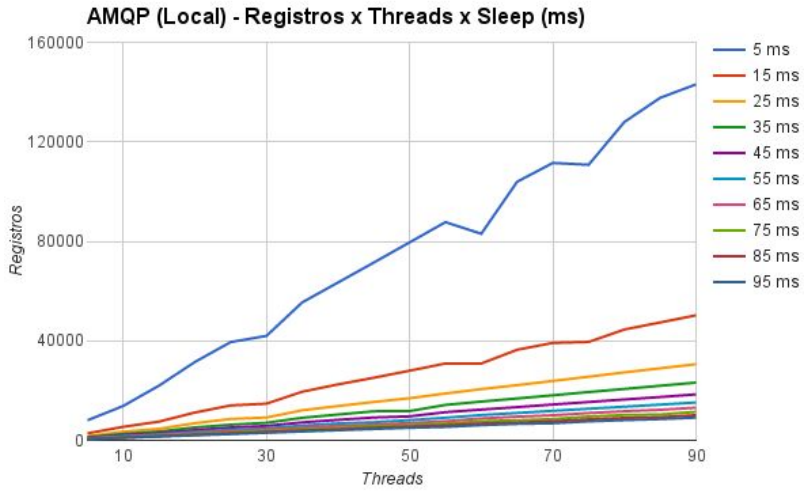


Imagem 13: Resultado do total de registros cadastrados no teste com AMQP no ambiente local em relação ao número de threads ativas e a pausa entre cada requisição.

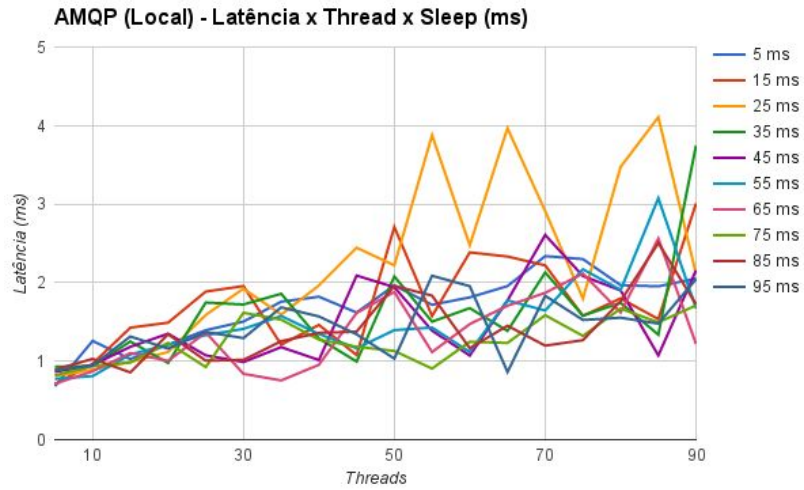


Imagem 14: Latência média do processamento das requisições no teste no ambiente local com AMQP em relação ao número de threads ativas e a pausa entre cada requisição. As latências foram tão baixas, que utilizou-se outra escala.

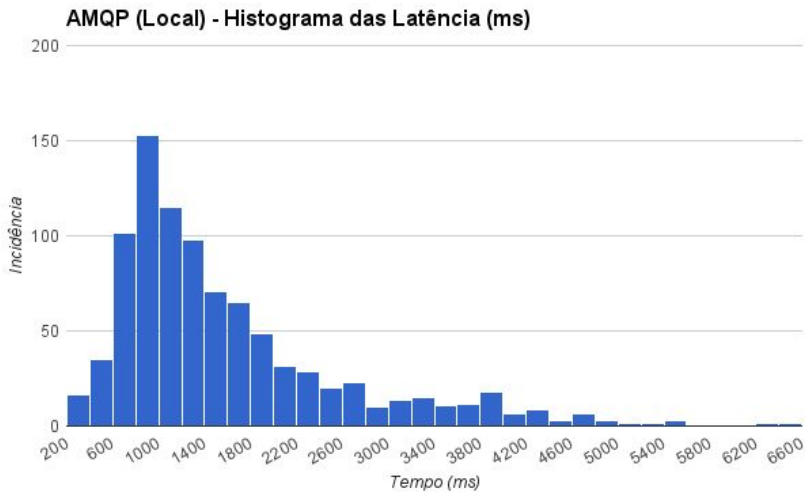


Imagem 15: Histograma das latências no teste AMQP realizado no ambiente local.

5.7.4. Mensageria - Heroku

A **Imagem 16** ilustra o número de registros cadastrados com o uso da mensageria e observa-se que o comportamento apresentado é muito semelhante ao que se obteve nos testes com REST na plataforma Heroku. A **Imagem 17** mostra o gráfico das latências mensuradas, diferentemente dos demais cenários, a variação do tempo de resposta parece estar mais diretamente associado à quantidade de threads e ao tempo de pausa entre as requisições. A oscilação, no entanto, não fica tão aparente ao se observar o histograma apresentado na **Imagem 18.** O tempo médio de resposta foi bem superior aos obtidos nos demais cenários: 258,23 ms.

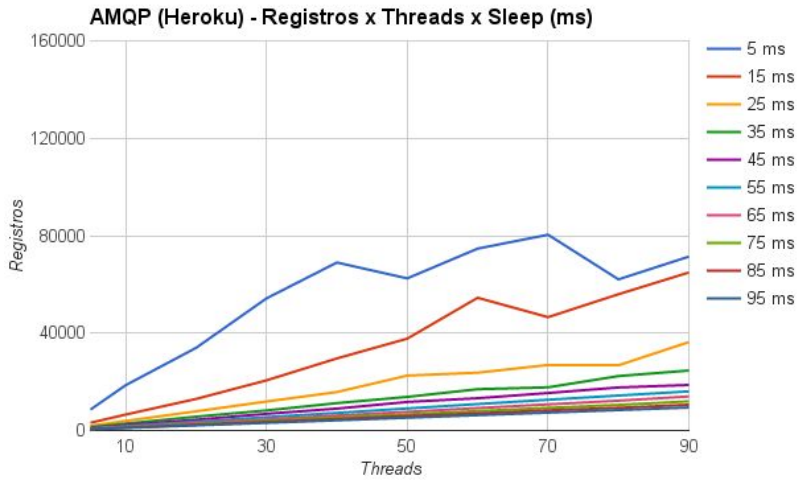


Imagem 16: Resultado do total de registros cadastrados no teste com AMQP na plataforma Heroku em relação ao número de threads ativas e a pausa entre cada requisição.

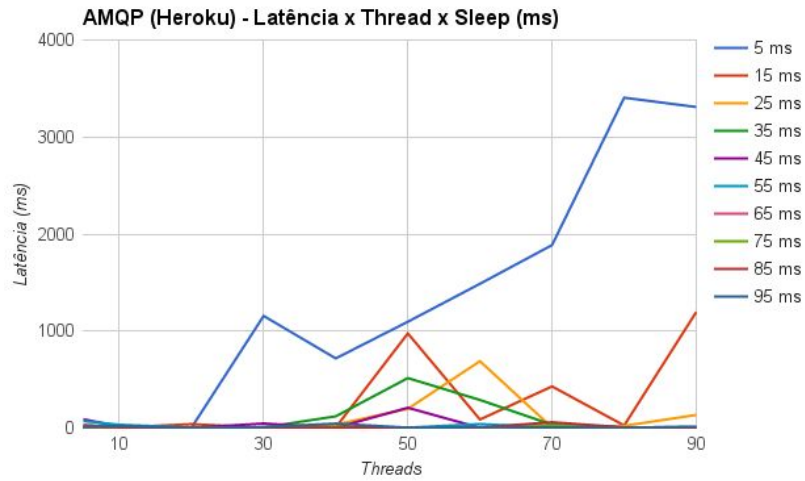


Imagem 17: Latência média do processamento das requisições no teste na plataforma Heroku com AMQP em relação ao número de threads ativas e a pausa entre cada requisição.

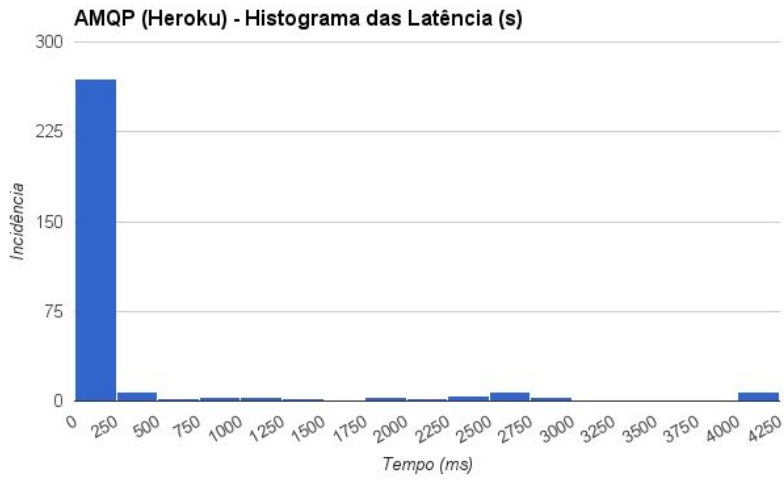


Imagem 18: Histograma das latências no teste AMQP realizado na plataforma Heroku.

5.8. ANÁLISE DOS RESULTADOS

A **Tabela 3** sumariza os dados obtidos ao comparar os resultados obtidos pelos testes dos cenários local e Heroku, para as técnicas de integração REST e AMQP.

Critério	Mensageria (AMQP)		REST	
	Local	Heroku	Local	Heroku
Latência média (ms)	1,57	258,23	78,98	79,27
Vazão máxima	143.801	99.692	154.283	150.283
Vazão total	15.209.959	13.684.285	11.459.350	14.412.824

Tabela 3: Resumo dos Testes

Conforme os dados apresentados na **Tabela 3** evidenciam, no cenário da mensageria na plataforma Heroku tanto a **latência**, quanto a **vazão** foram muito afetados pelas limitações da plataforma. No entanto, os testes com REST não demonstraram sofrer

um impacto tão relevante com tal limitação. A latência obtida nos testes locais com AMQP se destacaram, pois parecem sofrer menos o impacto do compartilhamento dos recursos ao executar os serviços em uma mesma máquina.

Os testes com REST foram muito semelhantes, sugerindo que o ambiente remoto oferecido pela plataforma Heroku possua uma rede interna de alta velocidade e que integre as máquinas com disponibilidade idêntica à local. A latência obtida no teste AMQP com Heroku pode ser justificada pelo fato do broker não estar hospedado na mesma rede onde rodam os serviços produtor e consumidor, sofrendo o impacto da latência de rede.

Para o quesito **complexidade**, observou-se os seguintes critérios: tempo de desenvolvimento do projeto, técnicas de integração e dependência de serviços externos. A **Tabela 4** mostra os resultados.

Critério	**Mensageria (AMQP)**	**REST**
Tempo de desenvolvimento	30 horas	18 horas
Estilo arquitetural	Coreografado e Coordenado	Coreografado e Coordenado
Dependência de serviços externos	Sim	Não necessariamente, pois pode-se implementar um serviço específico

Tabela 4: Complexidade das técnicas

O índice de **heterogeneidade** foi obtido com base na quantidade de linguagens de programação que a documentação das técnicas/ferramentas indicam que suportam e permitem integração. A **Tabela 5** lista algumas linguagens de programação e o **X** na coluna da técnica indica se ela suporta a linguagem.

A coluna AMPQ foi obtida do site do RabbitMQ (http://rabbitmq.com/), enquanto que a coluna REST veio de pesquisa na documentação das linguagens de programação

quanto à sua capacidade de comunicação de rede e possibilidade de implementar requisições HTTP.

Linguagem	**Mensageria (AMQP)**	**REST**
C/C++	X	X
Clojure	X	X
COBOL	X	X
Erlang	X	X
Go	X	X
Groovy on Grails	X	X
Haskell	X	X
Java	X	X
Lisp	X	X
.Net	X	X
Node.js	X	X
Objective-C/Swift	X	X
OCaml	X	X
Perl	X	X
PHP	X	X
Python	X	X
Ruby	X	X
Scala	X	X

Tabela 5: Suporte à linguagens de programação

6. CONCLUSÃO

A existência de diferentes técnicas de integração entre serviços é justificada com os testes realizados neste trabalho de conclusão de curso, pois como se pode observar com os dados apresentados, cada técnica possui aplicações conforme suas vantagens e desvantagens que devem ser ponderadas em função das necessidades do sistema que se deseja implementar. Analisando os quesitos avaliados, é possível concluir situações onde cada técnica se aplicaria apresentando melhores resultados dentre os cenários avaliados.

6.1. VAZÃO

O comportamento observado na comparação das vazões exibido na **Imagem 19** leva a crer que a concorrência por recursos tem impacto direto na quantidade de requisições processadas. Em ambos os cenários executados na plataforma Heroku, observou-se uma maior quantidade de registros processados.

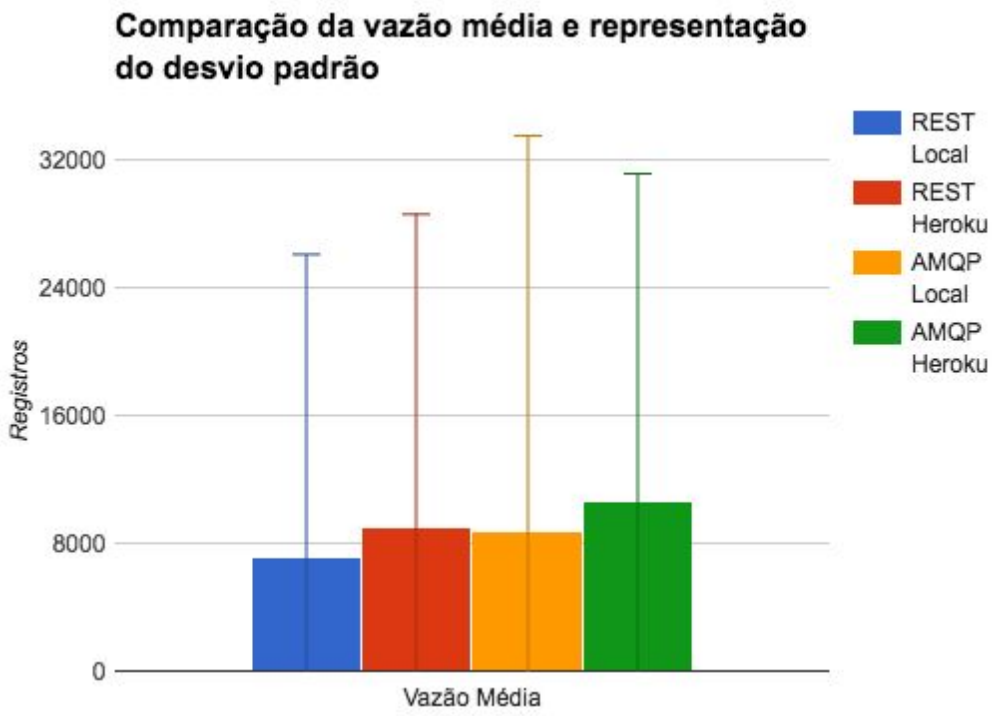


Imagem 19: Comparação da vazão média, com desvio padrão, entre os cenários de teste.

O estilo arquitetural orquestrado utilizado para implementar os serviços REST aumentou o número de requisições necessárias para realizar a operação de cadastro do consumidor e concentrou o processamento no **customer-service-rest**, reduzindo a vazão. Fica evidente o quanto a concorrência por recursos afetou o desempenho nos cenários locais.

Destacando o intenso número de mensagens processadas pelo *message broker* em uma máquina com recursos compartilhados, fica como ponto a ser avaliado em trabalhos futuros a diferença de desempenho caso o broker tivesse mais recursos a seu dispor, pois como se pode observar no cenário AMQP Local, este apresentou o maior pico de vazão.

6.2. LATÊNCIA

O cenário AMQP com Heroku apresentou as maiores e mais variadas latências entre todos os cenários. As limitações associadas ao plano gratuito tanto do Heroku, quanto do CloudAMQP, tiveram grande impacto na performance dos serviços, conforme se observa na **Imagem 20**. Ainda neste cenário, nota-se que nem a infra-estrutura dedicada foi o suficiente para dar vantagens ao teste.

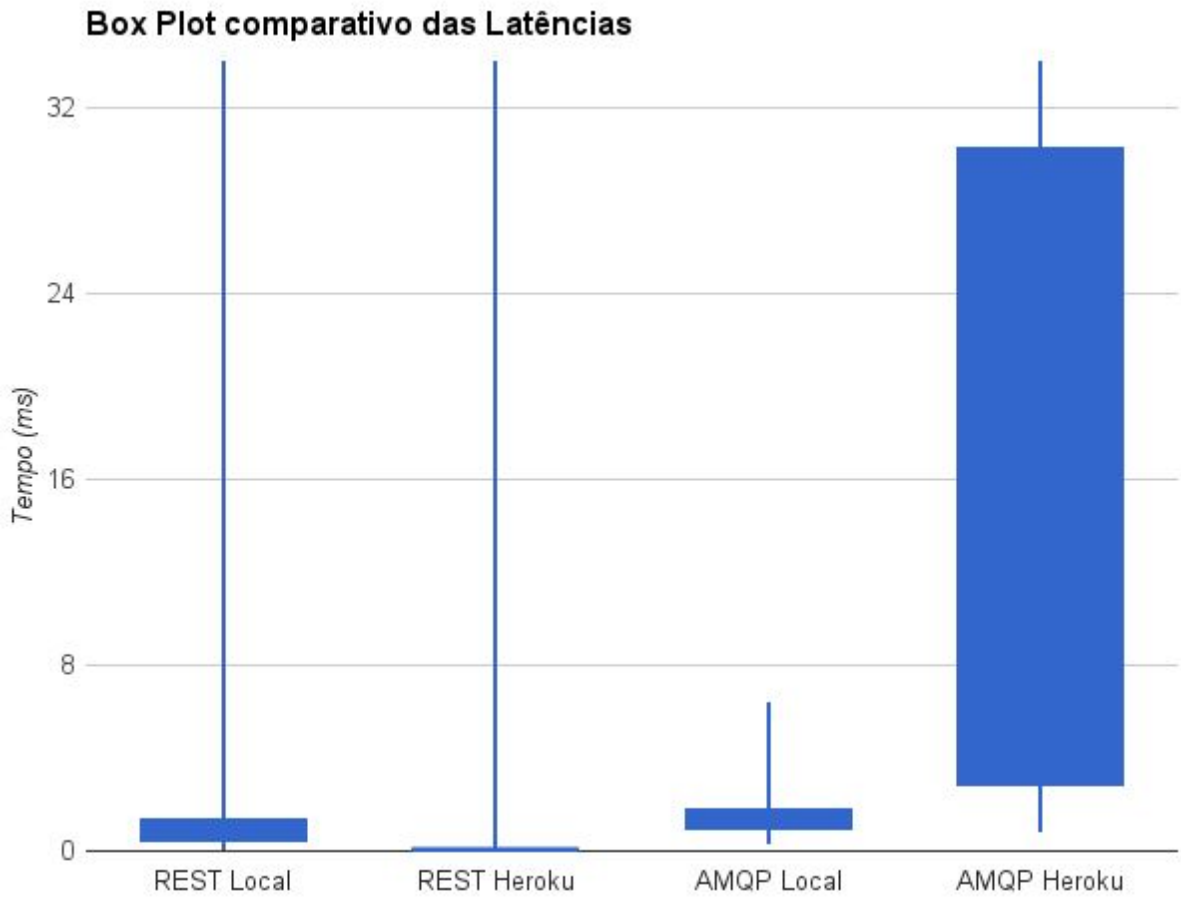


Imagem 20: Box Plot das latências entre os cenários de teste.

A média dos latências com REST foi interessante, pois apresentaram os menores patamares. Entretanto, é preciso levar em consideração a existência de pontos fora da curva em ambos os cenários Local e Heroku.

Chamou a atenção a consistência dos resultados com AMQP local, indicando uma maior confiabilidade nos resultados.

6.3. COMPLEXIDADE

A experiência do autor foi o principal parâmetro para avaliar a complexidade das técnicas de integração dos microsserviços. Entretanto, a quantidade de material disponível na literatura e internet tiveram grande influência no desenvolvimento dos serviços e nos testes dos mesmos. Nesse quesito, a mensageria apresentou maior vantagem, pois além de existirem diversos produtos que provêem a funcionalidade, a documentação contida neles é rica em exemplos.

No caso do REST, o problema foi encontrar soluções ou instruções suficientemente claras sobre como configurar ambas as formas de arquitetura de integração dos microsserviços.

Com o AMQP nos testes remotos, a restrição da quantidade de mensagens que o servidor gratuito disponibiliza serviu de fardo para a realização dos testes. Cada conta gratuita pode emitir um número máximo de 1.000.000 de mensagens ao mês e este valor é insignificante perto da quantidade de mensagens utilizadas nos cenários de teste descritos neste trabalho. Foram necessárias 19 contas diferentes para permitir que os testes fossem realizados em um número aceitável para os cálculos estatísticos.

Outro fator que reduziu muito a complexidade foi o uso do Spring Boot para criar os serviços e a vasta disponibilidade de material na internet para a codificação dos sistemas.

6.4. HETEROGENEIDADE

No quesito de heterogeneidade, ambas as técnicas apresentam larga adoção pelo mercado e oferecem suporte às principais linguagens de programação utilizadas atualmente, bem como para linguagens existentes há mais tempo.

Além dos quesitos planejados no escopo do projeto, outros foram identificados no andamento dos testes e também auxiliam na classificação das técnicas:

6.5. REQUISIÇÕES NÃO ATENDIDAS

Os gráficos não evidenciam, mas ao analisar os logs, observou-se que muitos registros não são atendidos a tempo. Em especial no caso do AMQP, onde o broker serve de pool de mensagens, permitindo que um produtor não seja bloqueado caso consiga produzir mensagens mais rapidamente do que os consumidores conseguem processar.

6.6. CONFORMIDADE COM PRINCÍPIOS DE MICROSSERVIÇOS

Em razão da escassez de material de pesquisa com informações sobre como configurar os serviços no estilo arquitetural coreografado com REST, concluiu-se que a mensageria apresenta maior conformidade com os princípios dos microsserviços. Contudo, é preciso cuidado, pois por se tratar de um sistema a parte, deve-se evitar que uma ferramenta de comunicação incorpore lógica de negócio, o que iria contra o princípio de "*dumb pipes, smart endpoints*".

6.7. TRABALHOS FUTUROS

Na continuidade deste trabalho, seria importante comparar diferentes linguagens de programação, bem como outras alternativas de integração, como Apache Thrift, WebSockets, SOAP, etc.

Um cenário não abordado nos testes é o uso de computadores em diferentes redes, o que certamente teria impacto na latência do processamento das requisições.

Adicionar outras métricas de comparação, como o tamanho da mensagem enviada, o índice de perda de mensagens, uso de CPU, sequencialização dos eventos, entre outros.

Aprimorar as análises estatísticas, de modo a permitir uma melhor análise e comparação entre os diferentes aspectos dos testes e dados coletados.

7. REFERÊNCIAS BIBLIOGRÁFICAS

ALONSO, Gustavo et al. **Web Services:** Concepts, Architecture and Applications. Palo Alto, Califórnia: Springer, 2004.

APACHE HttpComponents HttpClient. Disponível em: <https://hc.apache.org/httpcomponents-client-ga/index.html>. Acesso em: 10 nov. 2016.

ASYNC Http Client. Disponível em: <https://github.com/AsyncHttpClient/async-http-client>. Acesso em: 10 nov. 2016.

BIRRELL, Andrew D.; NELSON, Bruce Jay. Implementing remote procedure calls. **Acm Transactions On Computer Systems,** [s.l.], v. 2, n. 1, p.39-59, 1 fev. 1984. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/2080.357392. Disponível em: <http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf>. Acesso em: 06 dez. 2016.

BOOTH, David et al. **Web Services Architecture:** W3C Working Group Note 11 February 2004. 2004. Disponível em: <https://www.w3.org/TR/ws-arch/#whatis>. Acesso em: 12 out. 2016.

BORODESCU, Ciprian. **WEB SITES VS. WEB APPS: WHAT THE EXPERTS THINK.** 2013. Disponível em: <http://www.visionmobile.com/blog/2013/07/web-sites-vs-web-apps-what-the-experts-think/>. Acesso em: 12 jul. 2016.

CHAN, Lloyd. **Resolving 'Java OOM: Unable to Create New Native Thread' Errors on Heroku.** 2013. Disponível em: <https://beachape.com/blog/2013/09/12/resolving-java-oom-unable-to-create-new-native-thread-errors-on-heroku/>. Acesso em: 10 nov. 2016.

CHAPPEL, David. **The Myth of Loosely Coupled Web Services.** 2003. Disponível em: <http://www.davidchappell.com/HTML_email/Opinari_No5_3_03.html>. Acesso em: 14 jul. 2016.

COCKBURN, Alistair. **Hexagonal architecture.** 2005. Disponível em: <http://alistair.cockburn.us/Hexagonal+architecture>. Acesso em: 15 jul. 2016.

CONTEMPORARY history. 2016. Disponível em: <https://en.wikipedia.org/wiki/Contemporary_history>. Acesso em: 20 jun. 2016.

DIGITAL Revolution. 2016. Disponível em: <https://en.wikipedia.org/wiki/Digital_Revolution>. Acesso em: 20 jun. 2016.

EVANS, Eric. **Domain-driven Design:** Tackling Complexity in the Heart of Software. Boston: Addison-wesley Professional, 2004. 529 p.

FLEERACKERS, Tom. **Web 1.0 vs Web 2.0 vs Web 3.0 vs Web 4.0 vs Web 5.0:** A bird's eye on the evolution and definition. 2010. Disponível em: <https://flatworldbusiness.wordpress.com/flat-education/previously/web-1-0-vs-web-2-0-vs-web-3-0-a-bird-eye-on-the-definition/>. Acesso em: 4 jul. 2016.

FOWLER, Martin. **Richardson Maturity Model:** steps toward the glory of REST. 2010. Disponível em: <http://martinfowler.com/articles/richardsonMaturityModel.html>. Acesso em: 05 nov. 2016.

FOWLER, Martin; LEWIS, James. **Microservices**: a definition of this new architectural term. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em: 01 maio 2016.

FREEMAN, C; LOUÇÃ, F. **As Time Goes By:** From the Industrial Revolutions to the Information Revolution. United States Of America (): Oxford University Press, 2002. 432 p.

GAMMA, Erich et al. **Design Patterns Elements of Reusable Object-Oriented Software.** United States Of America: Addison Wesley, 1994. 361 p.

HEFNER, Katie; LYON, Matthew. **Where wizards stay up late::** the origins of the Internet.. Nova Iorque: Simon & Schuster, 2006.

JACOBSEN, Alessandra de Linhares. **METODOLOGIA CIENTÍFICA:** (ORIENTAÇÃO AO TCC). Florianópolis: Universidade Federal de Santa Catarina, 2016. Disponível em: <http://cursodegestaoelideranca.paginas.ufsc.br/files/2016/03/Apostila-Orientação-ao-TCC.pdf>. Acesso em: 07 jul. 2016.

JAVA.LANG.OUTOFMEMORYERROR: Unable to create new native thread. Disponível em: <https://plumbr.eu/outofmemoryerror/unable-to-create-new-native-thread>. Acesso em: 10 nov. 2016.

JONES, Steve. **Microservices is SOA, for this who know what SOA is.** 2014. Disponível em: <http://service-architecture.blogspot.com.br/2014/03/microservices-is-soa-for-those-who-know.html>. Acesso em: 14 jul. 2016.

JOSUTTIS, Nicolai M.. **SOA in Practice:** The art of distributed system design. Sebastopol, Ca, United States Of America: O'reilly Media, Inc., 2007.

KELLY, Mike. **HAL - Hypertext Application Language.** 2013. Disponível em: <http://stateless.co/hal_specification.html>. Acesso em: 15 jul. 2016.

KIRKPATRICK, David. **Case Study Website Redesign Leads to 34% Increase in Revenue.** Disponível em: <https://www.marketingsherpa.com/article/case-study/website-redesign-leads-to-34>. Acesso em: 12 jul. 2016.

KURAMOTO, Jake. **Website vs. Web App.** Disponível em: <http://theappslab.com/2010/11/12/website-vs-web-app/>. Acesso em: 12 jul. 2016.

LANINGHAM, Scott. **DeveloperWorks Interviews: Tim Berners-Lee.** Disponível em: <http://www.ibm.com/developerworks/podcast/dwi/cm-int082206txt.html>. Acesso em: 12 jul. 2016.

LEIA mais: Pesquisa Quantitativa e Pesquisa Qualitativa: Entenda a diferença [atualizado] | Blog Instituto PHD Pesquisa Quantitativa e Pesquisa Qualitativa: Entenda a diferença [atualizado]. Disponível em: <http://www.institutophd.com.br/blog/pesquisa-quantitativa-e-pesquisa-qualitativa-entenda-a-diferenca/>. Acesso em: 07 jul. 2016.

LUZ, Camila. **TCC SEM SOFRIMENTO: COMO DEFINIR METODOLOGIA E PRIMEIROS PASSOS.** Disponível em: <https://www.freetheessence.com.br/unplug/inspire-se/tcc-metodologia/>. Acesso em: 05 jul. 2016.

MARTIN, Robert C.. **The Single Responsibility Principle.** Disponível em: <http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle>. Acesso em: 14 jul. 2016.

MCPHERSON, Stephanie Sammartino. **Tim Berners-Lee: Inventor of the World Wide Web.** Minneapolis: Twenty-first Century Books, 2010.

MICROSERVICES. Intérpretes: Martin Fowler. Berlin, 2014. Streaming, son., color. Série GOTO. Disponível em: <https://www.youtube.com/watch?v=wgdBVIX9ifA>. Acesso em: 14 jul. 2016.

MORROW, Kim. **Web 2.0, Web 3.0, and the Internet of Things.** 2014. Disponível em: <http://www.uxbooth.com/articles/web-2-0-web-3-0-and-the-internet-of-things/>. Acesso em: 14 out. 2014.

MÜLLER, Josiane. **Paradigma e Sintagma.** Disponível em: <http://meuartigo.brasilescola.uol.com.br/gramatica/paradigma-sintagma.htm>. Acesso em: 11 jul. 2016.

NEWCOMER, Eric; LOMOW, Greg. **Understanding SOA with Web services.** Boston: Addison-wesley Professional, 2005.

NEWMAN, Sam. **Building Microservices:** Designing fine-grained systems. [s.l.]: O'reilly Media, Inc., 2015. 473 p.

NORMATIZAÇÃO de Trabalhos Acadêmicos: MATERIAL E MÉTODOS OU METODOLOGIA. MATERIAL E MÉTODOS OU METODOLOGIA. Disponível em: <http://www.fio.edu.br/manualtcc/co/7_Material_ou_Metodos.html>. Acesso em: 07 jul. 2016.

ORNBO, George. **API design for an event-driven world.** 2014. Disponível em: <https://shapeshed.com/api-design-for-an-event-driven-world/>. Acesso em: 25 out. 2016.

PROTOCOL Buffers. Disponível em: <https://developers.google.com/protocol-buffers/>. Acesso em: 15 jul. 2016.

RICHARDSON, Chris. **Microservices: Decomposing Applications for Deployability and Scalability.** 2014. Disponível em: <https://www.infoq.com/articles/microservices-intro>. Acesso em: 14 jul. 2016.

RICHARDSON, Chris. **Pattern: Monolithic Architecture.** 2014. Disponível em: <http://microservices.io/patterns/monolithic.html>. Acesso em: 12 jul. 2016.

ROCA: Resource-oriented Client Architecture. Disponível em: <http://roca-style.org>. Acesso em: 15 jul. 2016.

ROESTAMADJI, Philip. **Website cs. Web Application: What's the difference?** Disponível em: <https://lionandpanda.com/website-vs-web-application-whats-the-difference/>. Acesso em: 12 jul. 2016.

ROTEM-GAL-OZ, Arnon. **Services, Microservices, Nanoservices - oh my!** 2014. Disponível em: <http://arnon.me/2014/03/services-microservices-nanoservices/>. Acesso em: 14 jul. 2016.

S.TANENBAUM, Andrew; VAN STEEN, Maarten. **DISTRIBUTED SYSTEMS:** Principles and Paradigms. 2. ed. Amsterdam: Pearson Education, 2006. 705 p.

SERVICE Oriented Architecture. Disponível em: <https://en.wikipedia.org/wiki/Service-oriented_architecture>. Acesso em: 12 jul. 2016.

SHAPIRO, Ben. **Website vs. Web Application: What's the Difference?** 2013. Disponível em: <http://www.seguetech.com/website-vs-web-application-whats-the-difference/>. Acesso em: 12 jul. 2016.

TILKOV, Stefan. **Web-based frontend integration.** 2014. Disponível em: <https://www.innoq.com/blog/st/2014/11/web-based-frontend-integration/>. Acesso em: 15 jul. 2016.

TIM Berners-Lee: The next web. Intérpretes: Tim Berners-lee. Long Beach, Ca, Eua: Ted, 2009. (16 min.), son., color. Série TED2009. Disponível em: <http://www.ted.com/talks/tim_berners_lee_on_the_next_web>. Acesso em: 4 jul. 2016.

TONY RUSSELL. **MLA In-Text Citations: The Basics.** Disponível em: <https://owl.english.purdue.edu/owl/resource/747/02/>. Acesso em: 12 jul. 2016.

WILDE, Erik; PAUTASSO, Cesare. **REST: From Research to Practice.** New York: Springer-verlag New York, 2011. 528 p.

WOLFF, Eberhard. **Microservices**: Flexible Software Architectures. [s. L.]: Createspace Independent Publishing Platform, 2016. 338 p.

APÊNDICE I - ARTIGO SBC

Análise Comparativa de Técnicas de Integração entre Microsserviços

Renato Pereira Back

Departamento de Informática e Estatística (INE) - Universidade Federal de Santa Catarina (UFSC)
Campus Reitor João David Ferreira Lima, s/n, Trindade – Florianópolis – SC – Brasil

renatopb@gmail.com

Abstract. *Microservices became mainstream amongst distributed systems after the maturing of concepts and technologies like Cloud Computing, Containers Continuous Integration, Continuous Delivery and the rebirth of Functional Programming. Microservices are made to be integrated, so choosing the right technology and integration techniques is a major step when working towards success. This article evaluates REST and AMQP as different integration techniques, showing their pros and cons by analysing performance on a hypothetical scenario and presenting test results for latency and outflow.*

Resumo. *Microsserviços entraram em evidência entre os sistemas distribuídos após o amadurecimento de conceitos e tecnologias como Computação na Nuvem, Contêineres, Integração Contínua, Entrega Contínua e o renascimento da Programação Funcional. Microsserviços foram feitos para serem integrados, portanto escolher a tecnologia e as técnicas de integração corretas é um importante passo para garantir o sucesso da aplicação. Este artigo avalia REST e AMQP como diferentes técnicas de integração, mostrando seus prós e contras ao analisar sua performance em um cenário hipotético e apresentando os resultados de testes de latência e vazão.*

1. Introdução

A Revolução Digital, simbolizada pela disseminação de computadores digitais e do armazenamento de dados na forma digital, estabeleceu o início do período histórico conhecido como Era da Informação. Um dos marcos deste período foi o surgimento e, posteriormente, a ampla adoção da Internet e da World Wide Web - WWW (daqui em diante denominada apenas Web). Na início da Web, o conteúdo fornecido era praticamente todo estático, como um repositório de documentos para serem acessados conforme sua URL. O surgimento de tecnologias e especificações como o Common Gateway Interface - CGI, permitiu que os servidores web fornecessem não apenas conteúdos estáticos, mas também conteúdos dinâmicos, gerados a partir de requisições recebidas via HTTP, dando origem a termos controversos, como: Web 1.0 referindo-se ao período dos conteúdos estáticos; Web 2.0, ou colaborativa, dito dos conteúdos gerados conforme interação com os usuários; e, posteriormente, Web 3.0, ou semântica, "a internet que sabe o que o usuário quer" [MORROW 2014]. No momento, estamos presenciando a explosão ou Internet das Coisas - IoT (sigla em inglês), chamada por alguns autores de Web 4.0, que envolve a comunicação de dispositivos diversos em constante comunicação com a rede, consumindo e gerando dados. Há relatos de que a Web 5.0 já está em desenvolvimento e que envolverá a conectividade, inteligência e

emoção nos dados. Os termos são ditos controversos, pois a web já fora concebida com o intuito de ser colaborativa e dinâmica. Então, classificá-la com outros nomes é visto por muitos como sinônimos, ou apelidos, apenas para facilitar a compreensão de certas características da Web.

A evolução da Web teve repercussões em novas tecnologias de hardware, software e sistemas operacionais. Da mesma forma, metodologias e processos de desenvolvimento foram evoluindo em paralelo para se adequar à nova realidade. A necessidade de atender um número cada vez maior de usuários e requisições tornou as arquiteturas de software do início da web obsoletas e insuficientes para atender esta nova realidade que exige sistemas responsivos e adaptáveis para incluir novas funcionalidades com baixo custo e maior velocidade de entrega, atender aumento de demanda de forma escalável e tornar os desenvolvedores mais produtivos e eficientes.
As primeiras técnicas para desenvolvimento de sistemas focavam em aplicações monolíticas, que consistiam de um único bloco agregador de funcionalidades, com grande necessidade de recursos humanos e computacionais, onde as maneiras para aumento da capacidade de atendimento consistiam no aumento de recursos no servidor, ou a replicação do sistema como um todo, tornando o processo caro e ineficiente.

Dentre as abordagens que surgiram para contornar essas dificuldades e problemas dos sistemas monolíticos, destacam-se os sistemas distribuídos, particularmente a Arquitetura Orientada a Serviços - SOA (sigla em inglês para Service Oriented Architecture). Porém, a falta de consenso em como aplicá-la de forma eficiente, as diferentes práticas adotadas pela indústria e as dificuldades técnicas encontradas pelos desenvolvedores, geraram obstáculos que diminuíram a adoção da SOA pela indústria. Foi justamente dessa realidade que os microsserviços emergiram. O conceito em si não é novo, porém um conjunto de fatores tecnológicos recentes proporcionou que os microsserviços surgissem, tendo a aceitação e a popularidade que estão tendo.

A análise realizada neste trabalho será feita com base na implementação de microsserviços com escopo limitado na aplicação MusicCorp, descrita no livro Building Microservices [NEWMAN 2015] e com foco na integração dos serviços, avaliando critérios de vazão, latência, complexidade de implementação, heterogeneidade tecnológica e atendimento aos conceitos que definem uma boa arquitetura de integração.
O resultado da análise proposta neste trabalho é de relevância para quem deseja adotar a arquitetura de microsserviços e busca mais informações sobre as técnicas de integração que deve adotar. Ao invés de propor um guia de regras, o objetivo é fornecer orientações, com base nos dados levantados, que auxiliem na escolha da arquitetura de integração.

2. Microsserviços

Newman (2015) considera microsserviços como serviços pequenos e autônomos que funcionam em conjunto. Devem ser pequenos em razão do princípio de responsabilidade única [FOWLER 2008], que mantém agrupadas funcionalidades que são alteradas juntas, e separadas aquelas que sofrem alterações por motivos diferentes. O tamanho da equipe que desenvolve os serviços também deve ser pequeno, pois equipes menores são mais independentes, descentralizadas e responsivas, propiciando mais autonomia para evoluir sem ser afetado, nem afetar outras partes do todo.
Para Jones (2014), microsserviços são uma maneira de entrega de conteúdo orientada a serviços (Service Oriented Delivery - SOD) para uma solução bem arquitetada e granularizada de SOA.

Fowler e Lewis (2014) definem que o estilo arquitetural dos microsserviços aborda como desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um executando em seu próprio processo e utilizando mecanismos leves de comunicação,

geralmente uma API de recursos HTTP. Estes serviços são construídos em torno de competências de negócio e são disponibilizados independentemente por processos de deploy automatizados. Há um gerenciamento centralizado mínimo desses serviços, que podem ser escritos em diferentes linguagens de programação e utilizar distintas tecnologias de armazenamento de dados.

2.1. Benefícios

Dentre os benefícios obtidos com microsserviços, podemos citar [FOWLER, LEWIS 2014] [NEWMAN 2015] [WOLFF 2016]:

- Liberdade para substituição e composição de serviços;
- Flexibilidade para lidar com sistemas legados de modo sustentável;
- Facilidade de entrega;
- Entrega contínua;
- Escalabilidade sob medida;
- Robustez e resiliência através da prática de tolerância a falhas;
- Liberdade para escolha de tecnologias heterogêneas;
- Independência do restante do sistema;
- Alinhamento organizacional com projetos e equipes menores;
- Desenvolvimento paralelo de funcionalidades.

2.2. Princípios

Newman (2015) recomenda que, para um microsserviço oferecer todos os benefícios esperados, as seguintes características sejam seguidas:

- Baixo acoplamento: um serviço deve saber o mínimo necessário dos serviços com os quais ele interage, de modo a garantir que um serviço não sofra mudanças em razão de outros serviços, nem cause alterações em outros serviços devido a modificações em suas funcionalidades.
- Alta coesão: seguindo o Princípio de Responsabilidade Única, comportamentos associados devem ser agrupados, enquanto que comportamentos desconexos não devem ser reunidos. Assim, alterações não são propagadas.
- Delimitação do contexto: utilizando o conceitos de DDD de contextos delimitados, os microsserviços encapsulam informações que são pertinentes ao seu domínio, expondo através de interfaces apenas o que deve ser compartilhado com os consumidores.
- Regras de negócio: ao delimitar o contexto de cada serviço, suas próprias funcionalidades não devem ser voltadas aos dados compartilhados com o mundo exterior, mas sim às operações que o serviço realiza e aos dados dos quais ele precisa para isso.
- Comunicação condicionada aos conceitos de negócio: a comunicação entre os serviços deve obedecer o modo em que os contextos que eles representam se comunicam na vida real. Isto facilita a implementação de mudanças em decorrência de alterações nas regras de negócio.

3. Integração

Microsserviços têm que ser integrados e precisam se comunicar. Na opinião de Newman (2015), o aspecto mais importante da tecnologia associada com microsserviços é esclarecer os fatos sobre a integração. Se bem feita, a autonomia é mantida, do contrário as chances de ocorrerem problemas aumentam drasticamente.

Os pontos chave para os microsserviços são: alterações que impactam externos devem ser as mais raras possíveis; APIs de comunicação não devem ser associadas às tecnologias utilizadas; serviço deve ser simples de usar; encapsulamento de detalhes internos; foco na integração responsiva, através de eventos. Todos os pontos-chave giram em torno de um dos princípios fundamentais para microsserviços: o desacoplamento.

3.1. Estilos

Existem dois tipos de estilo arquitetural que podem ser utilizados para realizar a integração de microsserviços: o Coreografado e o Orquestrado [NEWMAN 2015].

O estilo **Coreografado** é normalmente associado à programação orientada a eventos, onde cada serviço monitora determinadas situações e executa seu processamento caso uma determinada condição seja atendida. Este estilo é utilizado principalmente em sistemas assíncronos e responsivos.

Em contrapartida, o estilo **Orquestrado** gira em torno de um ponto de comando central, onde um serviço é responsável por realizar chamadas aos demais serviços e aguarda o retorno dos devidos processamentos. Esta abordagem é normalmente associada a sistemas síncronos e quebra o princípio do baixo acoplamento.

3.2. Tecnologias

Wolff (2016) classifica como integração lógica quando os microsserviços chamam uns aos outros para trabalhar em conjunto. Apresentam-se como integrações lógicas tecnologias como REST, SOAP, RPC e mensageria.

Comumente guiado pelo Modelo de Maturidade de Richardson [FOWLER 2010], o Representational State Transfer (REST) é baseado no vocabulário HTTP (ex.: GET, POST, PUT) para representar as ações que podem ser executadas pelo microsserviço, caracterizando-se por suportar o uso de cache e balanceamento de carga; uso do Hypertext Application Language (HAL) que permite a implementação do Hypermedia as the Engine of Application State (HATEOAS), que possibilita uma maior autonomia e flexibilidade dos microsserviços; suportar diferentes formatos para transporte de dados, como XML, HTML, JSON e Protocol Buffer; e ter processamento síncrono em razão do HTTP.

A Remote Procedure Call (RPC), inglês para chamada remota de procedimentos, é um mecanismo que permite a chamada de métodos em outros processos transparentemente, como se fosse uma chamada local [BIRRELL e NELSON 1984]. É mais utilizada atualmente na forma das tecnologias SOAP e Thrift.

Sistemas de Mensageria são middlewares que servem como uma terceira opção para a integração lógica dos microsserviços. Wolff (2016) destaca as seguintes características: mensagens podem ser entregues para mais de um recipiente; garantia de entrega de mensagens contorna a falha de comunicação; controle de entrega pode validar o resultado do processamento e reenviá-lo no caso de falhas; mensagens são processadas de forma assíncrona; o desacoplamento é facilitado, pois as mensagens não são enviadas diretamente para um recipiente; e, oferecem suporte transacional sem necessidade de uma coordenação global.

4. Experimentos

Para a definição do escopo do trabalho, realizarmos os experimentos e, por fim, a análise das técnicas de integração, utilizamos a funcionalidade de Cadastro de Consumidor

descrita na aplicação da MusicCorp, descrita no livro Building Microservices [NEWMAN 2015]. A **Imagem 1** ilustra a funcionalidade descrita.

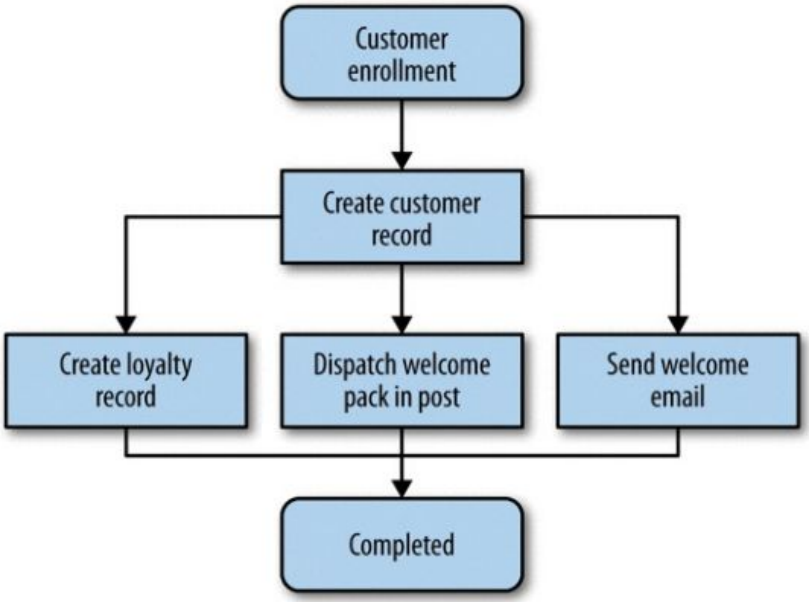


Imagem 1. Processo para criação de novo Consumidor [NEWMAN 2015]

Em detalhes, o início do processo se dá com o registro do Consumidor, realizado pelo Customer Service. Após a processamento do mesmo, os serviços Loyalty Service, Post Service e Email Service devem realizar seus processamentos para criação do programa de fidelização, envio do pacote de boas vindas e envio de email de boas vindas, respectivamente. A execução de todas essas tarefas marca como concluído o processo em sua íntegra. Este artigo avaliou apenas as técnicas de integração entre os serviços, ignorando quaisquer lógicas de negócio. As métricas avaliadas foram a Vazão das requisições, a Latência dos serviços, a Facilidade de implementação e o Índice de Heterogeneidade.

Para a implementação dos serviços, foi utilizada a linguagem de programação Java (http://www.java.com) na versão JDK 1.8.0_45-b14, em conjunto com o Spring Boot (http://projects.spring.io/spring-boot/) na versão 1.4.2. As técnicas de integração avaliadas foram o REST e RabbitMQ como implementação do AMQP. Nos testes com REST, utilizou-se o estilo arquitetural orquestrado, com um serviço coordenando as chamadas. Para os testes com AMQP, o broker ficou responsável pelo registro e descoberta dos serviços. A IDE utilizada para codificação dos serviços foi o Spring Tool Suite - STS (https://spring.io/tools) na versão 3.8.2.RELEASE. Uma das funcionalidades providas pelo Spring Boot é que o artefato gerado após a compilação é um arquivo JAR auto executável e que carrega todas as suas dependências.

Para os testes remotos, utilizou-se o serviço de Platform as a Service Heroku: Cloud Application Platform (http://www.heroku.com) para realizar os testes com serviços remotos. O Heroku também disponibiliza add-ons com o CloudAMQP, uma versão online e gratuita do RabbitMQ para servir de broker AMQP. Para a chamada inicial dos serviços e execução dos testes, foi utilizado Postman (http://www.getpostman.com) para realizar requisições HTTP que simulam o cadastro de um Consumidor. O pré-processamento dos logs foi feito com a ferramenta Sublime Text 3 (http://www.sublimetext.com).

Dois cenários foram escolhidos para execução dos testes: todos os serviços rodando localmente na mesma máquina; e cada serviço rodando em uma instância na plataforma Heroku. Para a mensageria, o serviço do RabbitMQ foi executado localmente

no cenário 1, enquanto que no cenário 2 foi executado no Heroku, com auxílio do addon CloudAMQP, no plano Little Lemur. Todos os testes seguiram os seguintes critérios:

- Repetições - quantas vezes um teste foi repetido.
- Intervalo - quanto tempo uma thread realiza requisições.
- Quantidade - quantas threads simultâneas realizam requisições.
- Pausa - que cada thread faz antes de realizar outra requisição.

5. Análise dos Resultados

A **Tabela 1** sumariza os dados coletados ao comparar os resultados obtidos pelos testes nos cenários local e Heroku para as técnicas de integração REST e AMQP.

Tabela 1. Resumo dos Testes

Critério	Mensageria (AMQP)		REST	
	Local	Heroku	Local	Heroku
Latência média (ms)	1,57	258,23	78,98	79,27
Vazão máxima	143.801	99.692	154.283	150.283
Vazão total	15.209.959	13.684.285	11.459.350	14.412.824

Em maiores detalhes, os resultados são ilustrados pela **Imagem 2**, **Imagem 3** e a **Imagem 4**.

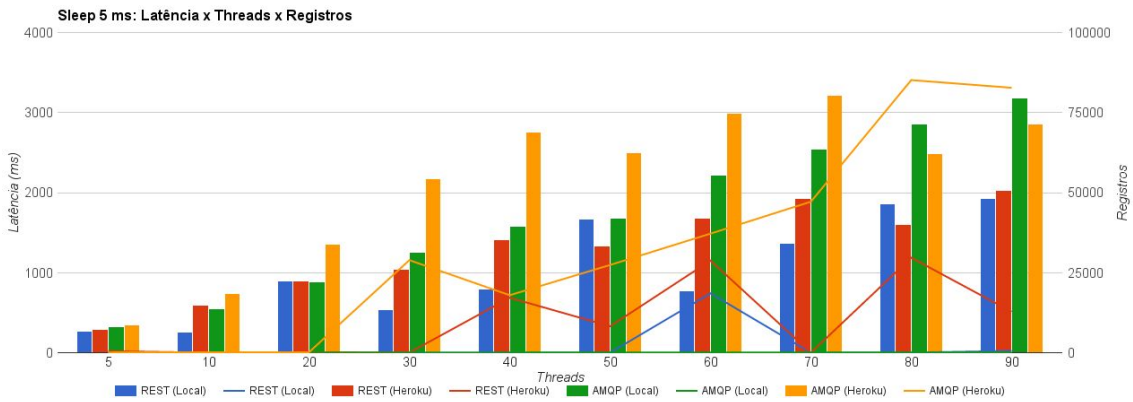


Imagem 2. Gráfico da Vazão e das Latências registradas nos testes com pausas de 5 ms entre cada requisição.

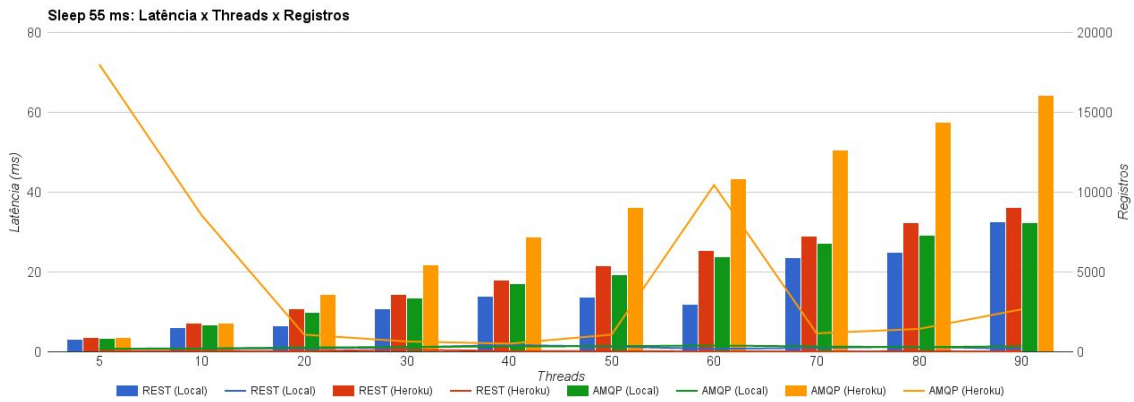


Imagem 3. Gráfico da Vazão e das Latências registradas nos testes com pausas de 55 ms entre cada requisição.

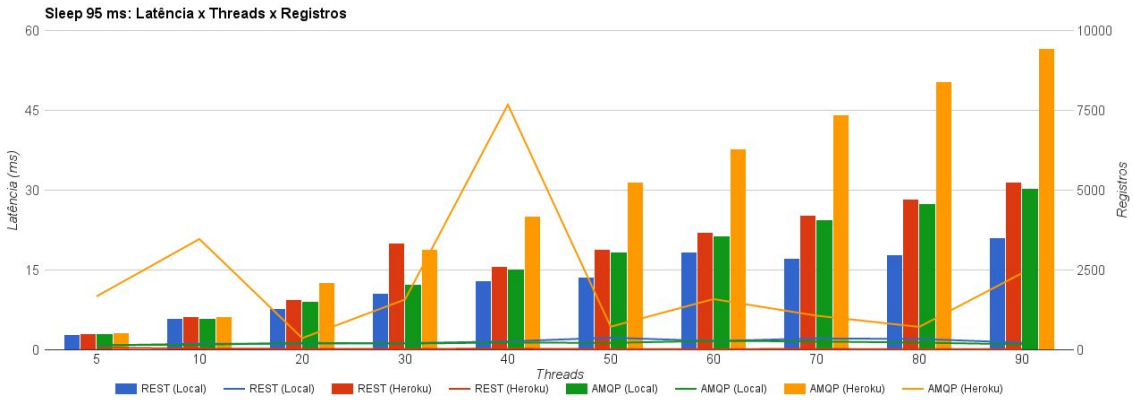


Imagem 4. Gráfico da Vazão e das Latências registradas nos testes com pausas de 95 ms entre cada requisição.

Na **Imagem 5**, observa-se a diferença entre a vazão média entre os diferentes cenários. Nota-se como o cenário AMQP Heroku se destacou em praticamente todas as combinações de Threads x Pausas, dando indícios de que o estilo arquitetural adotado e a disponibilidade de recursos possibilitaram tal vantagem.

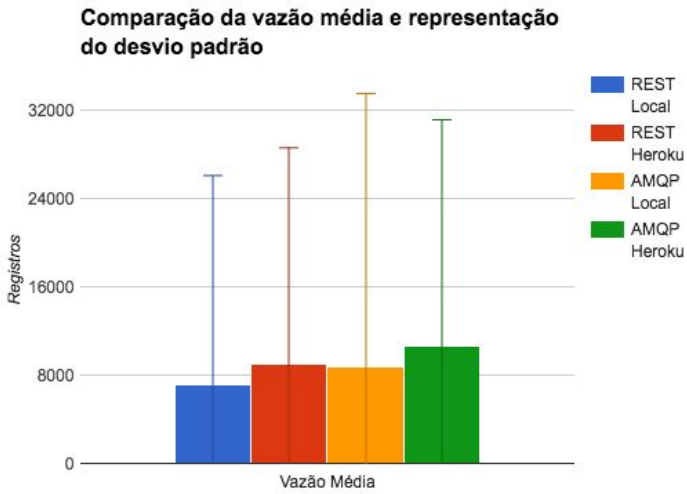


Imagem 5. Comparação da vazão média, com desvio padrão, entre os cenários de teste.

A **Imagem 6** mostra um gráfico de Box Plot das latências médias obtidas nos testes. Fica visível o contrapeso da vazão apresentada no cenário AMQP Heroku, pois dentre os cenários analisados, este foi o que apresentou as maiores e mais distribuídas latências. Nota-se como os cenários com REST apresentaram uma maior concentração dos resultados, mas também uma vasta amplitude, ao passo que o cenário AMQP Local inspira maior consistência e confiança pelas latências não variarem tanto serem relativamente baixas, em comparação aos cenários estudados.

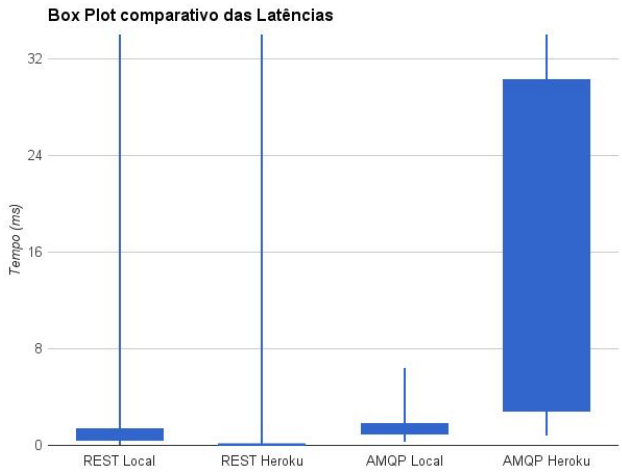


Imagem 6. Box Plot das latências entre os cenários de teste.

A **Tabela 2** apresenta o tempo de desenvolvimento das aplicações utilizadas nos testes e configuração dos servidores; o suporte aos estilos arquiteturais coreografado e orquestrado; e, a dependência que cada cenário tem de serviços externos.

Para a **heterogeneidade**, ambas as técnicas se sobressaem, pois suportam e são suportadas linguagens, como C/C++, COBOL, Erlang, Go, Groovy, Haskell, Java, Lisp, .Net, Node.js, Objective-C, OCaml, Perl, PHP, Python, Ruby, Scala, Swift.

Tabela 2. Complexidade das Técnicas de Integração

Critério	**Mensageria (AMQP)**	**REST**
Tempo de desenvolvimento	30 horas	18 horas
Estilo arquitetural	Coreografado e Orquestrado	Coreografado e Orquestrado
Dependência de serviços externos	Sim	Não necessariamente, pois pode-se implementar um serviço específico

6. Conclusão

A existência de diferentes técnicas de integração entre serviços é justificada com os testes realizados neste trabalho de conclusão de curso, pois como se pode observar com os dados apresentados, cada técnica possui aplicações conforme suas vantagens e desvantagens que devem ser ponderadas em função das necessidades do sistema que se deseja implementar. Analisando os quesitos avaliados, é possível concluir situações onde cada técnica se aplicaria apresentando melhores resultados dentre os cenários avaliados.

O comportamento observado na comparação das **vazões** exibido na **Imagem 5** leva a crer que a concorrência por recursos tem impacto direto na quantidade de requisições processadas, ficando ainda mais evidenciado nos testes com REST. Em ambos os cenários executados na plataforma Heroku, observou-se uma maior quantidade de registros processados. Destaca-se o cenário AMQP Local, que apresentou o maior pico de vazão, mesmo trabalhando com recursos compartilhados.

Em relação às **latências**, o cenário AMQP Heroku apresentou as maiores e mais variadas entre todos. As limitações associadas ao plano gratuito tanto do Heroku, quanto do CloudAMQP, tiveram grande impacto na performance dos serviços, conforme se observa na **Imagem 6**. Ainda neste cenário, nota-se que nem a infra-estrutura dedicada foi o suficiente para dar vantagens ao teste. Os testes com REST são interessantes, pois apresentaram baixa latência média, apesar de uma alta amplitude. Chamou a atenção a consistência e confiabilidade do cenário AMQP local, devido à sua concentração e baixa latência mesmo com recursos compartilhados.

A experiência do autor foi o principal parâmetro para avaliar a **complexidade** das técnicas de integração dos microsserviços. Entretanto, a quantidade de material disponível na literatura e internet tiveram grande influência no desenvolvimento dos serviços e nos testes dos mesmos. Nesse quesito, a mensageria apresentou maior vantagem, pois além de existirem diversos produtos que provêem a funcionalidade, a documentação contida neles é rica em exemplos.

No quesito de **heterogeneidade**, ambas as técnicas apresentam larga adoção pelo mercado e oferecem suporte às principais linguagens de programação utilizadas atualmente, bem como para linguagens existentes há mais tempo.

Os gráficos não evidenciam, mas ao analisar os logs, observou-se que muitos registros não são atendidos a tempo. Em especial no caso do AMQP, onde o broker serve de pool de mensagens, permitindo que um produtor não seja bloqueado caso consiga produzir mensagens mais rapidamente do que os consumidores conseguem processar..

Em razão da escassez de material de pesquisa com informações sobre como configurar os serviços no estilo arquitetural coreografado com REST, concluiu-se que a mensageria apresenta maior conformidade com os princípios dos microsserviços. Contudo, é preciso cuidado, pois por se tratar de um sistema a parte, deve-se evitar que uma ferramenta de comunicação incorpore lógica de negócio, o que iria contra o princípio de "*dumb pipes, smart endpoints*".

Referencias

BACK, Renato P. Análise Comparativa de Técnicas entre Microsserviços. Florianópolis. UFSC, 2016.

BIRRELL, Andrew D.; NELSON, Bruce Jay. Implementing remote procedure calls. Acm Transactions On Computer Systems, [s.l.], v. 2, n. 1, p.39-59, 1 fev. 1984. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/2080.357392. Disponível em: <http://www.cs.virginia.edu/~zaher/classes/CS656/birrel.pdf>. Acesso em: 06 dez. 2016.

FOWLER, Martin. Richardson Maturity Model: steps toward the glory of REST. 2010. Disponível em: <http://martinfowler.com/articles/richardsonMaturityModel.html>. Acesso em: 05 nov. 2016.

FOWLER, Martin; LEWIS, James. Microservices: a definition of this new architectural term. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em: 01 maio 2016.

JONES, Steve. Microservices is SOA, for this who know what SOA is. 2014. Disponível em: <http://service-architecture.blogspot.com.br/2014/03/microservices-is-soa-for-those-who-know.html>. Acesso em: 14 jul. 2016.

MORROW, Kim. Web 2.0, Web 3.0, and the Internet of Things. 2014. Disponível em: <http://www.uxbooth.com/articles/web-2-0-web-3-0-and-the-internet-of-things/>. Acesso em: 14 out. 2014.

NEWMAN, Sam. Building Microservices: Designing fine-grained systems. [s.l.]: O'reilly Media, Inc., 2015. 473 p.

WOLFF, Eberhard. Microservices: Flexible Software Architectures. [s. L.]: Createspace Independent Publishing Platform, 2016. 338 p.

APÊNDICE II - CÓDIGOS-FONTE

O código fonte dos sistemas elaborados para este trabalho estão disponíveis no GitHub, através das seguintes URLs.:

- **customer-service-mq**: http://github.com/tioback/customer-service-mq/
- **customer-service-rest**: http://github.com/tioback/customer-service-rest/
- **email-service-mq**: http://github.com/tioback/email-service-mq/
- **email-service-rest**: http://github.com/tioback/email-service-rest/
- **loyalty-service-mq**: http://github.com/tioback/loyalty-service-mq/
- **loyalty-service-rest**: http://github.com/tioback/loyalty-service-rest/
- **post-service-mq**: http://github.com/tioback/post-service-mq/
- **post-service-rest**: http://github.com/tioback/post-service-rest/

APÊNDICE III - APRESENTAÇÃO TCC

Análise Comparativa de Técnicas de Integração de Microsserviços

Aluno: Renato Pereira Back
Orientador: Prof. Frank Siqueira
Banca: Ivan Salvadori
Prof. Fernando Cruz

1

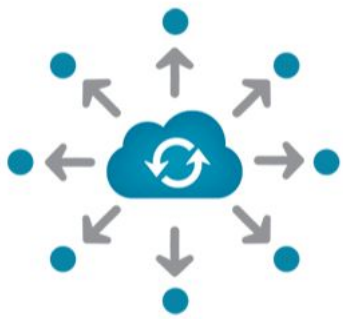
Introdução

Sistemas distribuídos

Microsserviços

Monolito x Microsserviços

Arquiteturas de integração



2

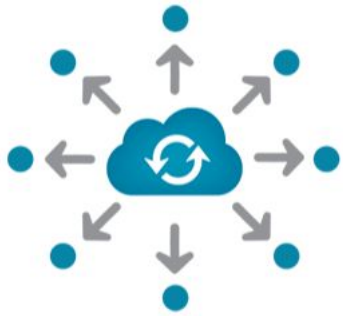
Introdução:
Sistemas distribuídos

Sistemas distribuídos

Microsserviços

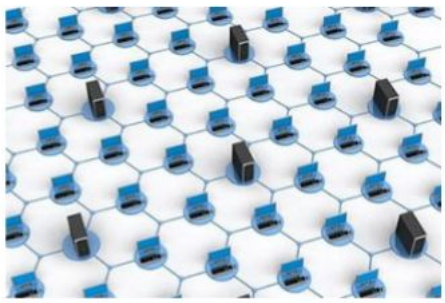
Monolito x Microsserviços

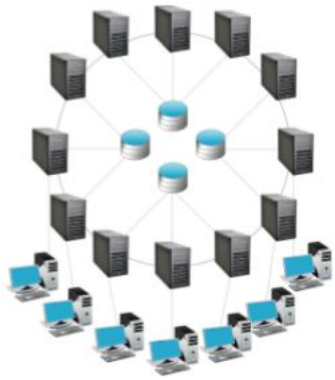
Arquiteturas de integração



3

Introdução:
Sistemas distribuídos





4

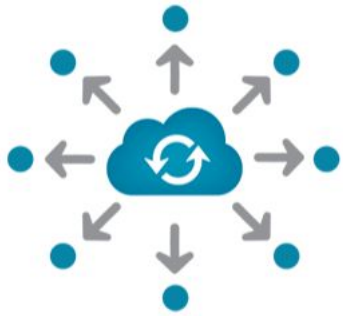
Introdução: Microsserviços

Sistemas distribuídos

Microsserviços

Monolito x Microsserviços

Arquiteturas de integração



5

Introdução: Microsserviços

"São SOA para aqueles que sabem o que é SOA."
- Steve Jones, 2014

"Pequenos, autônomos e fazem bem uma coisa." - Newman, 2015.

"Pequenos em razão do princípio de responsabilidade única" - Fowler, 2008.

Substituíveis, Escaláveis, Robustos, Resilientes, Autônomos, Alinhados à Organização, Agnóstico de Tecnologia

Princípios

- Baixo acoplamento
- Alta coesão
- Delimitação de contexto
- Funcionalidades de negócio
- Comunicação condicionada aos conceito de negócio

6

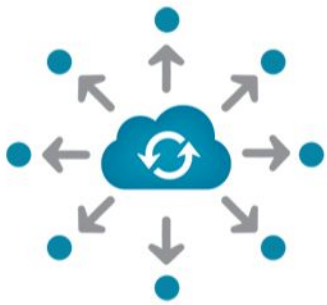
Introdução:
Monolito x Microsserviços

Sistemas distribuídos

Microsserviços

Monolito x Microsserviços

Arquiteturas de integração



7

Introdução:
Monolito x Microsserviços

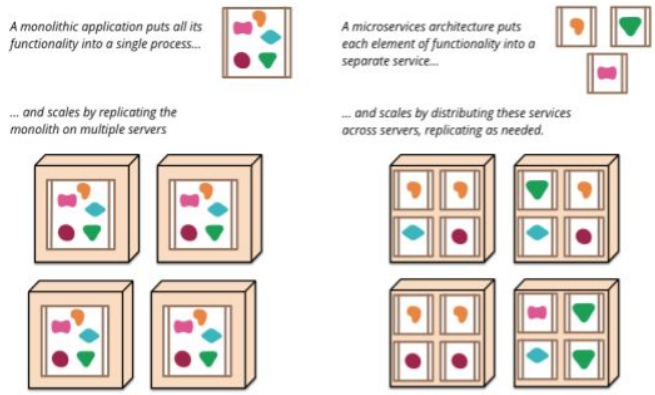


Figure 1: Monoliths and Microservices
http://martinfowler.com/articles/microservices.html

8

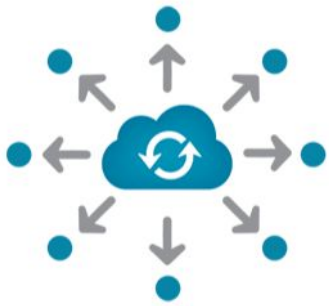
Introdução:
Arquiteturas de Integração

Sistemas distribuídos

Microsserviços

Monolito x Microsserviços

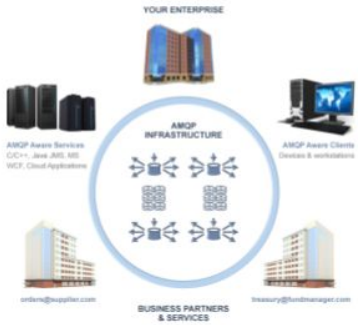
Arquiteturas de integração



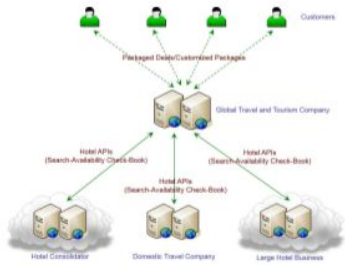
9

Introdução:
Arquiteturas de Integração

DB



AMQP



REST/SOAP/RPC

10

Problema da Pesquisa

- Integração é um dos aspectos mais importantes
  - Permite aos serviços manter autonomia e independência
- Escopos delimitados
- Suscetível a mudanças
- Facilidade de adoção
- **Qual arquitetura utilizar?**

11

Objetivo Geral

O objetivo geral deste trabalho é apresentar argumentos que balizem a decisão de um arquiteto de software quando este estiver escolhendo a arquitetura de integração de microsserviços.

12

Objetivos Específicos

- Revisar o referencial teórico
- Elaborar métricas
- Desenvolver microsserviços
- Comparar os resultados coletados

13

Limitação do Escopo

{REST}

14

Desenvolvimento: Ferramentas

AspectJ

RabbitMQ

heroku

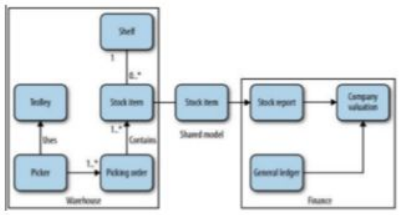
{REST}

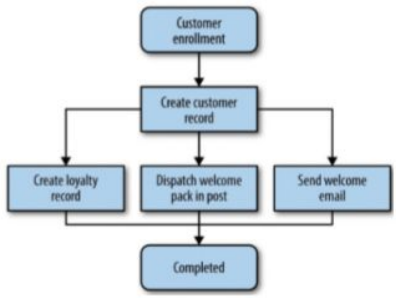
spring boot

CloudAMQP

15

Desenvolvimento: Escopo





16

Desenvolvimento: Serviços

- Customer Service
  - Criação do consumidor
- Loyalty Service
  - Associação consumidor à fidelização de clientes
- Post Service
  - Envio de pacotes físicos
- Email service
  - Envio de emails

17

Desenvolvimento: Implementações

REST:

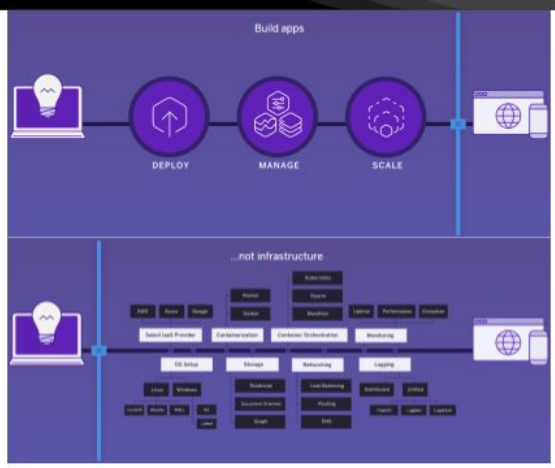
- customer-service-rest
  https://github.com/tioback/customer-service-rest
- email-service-rest
  https://github.com/tioback/email-service-rest
- loyalty-service-rest
  https://github.com/tioback/loyalty-service-rest
- post-service-rest
  https://github.com/tioback/post-service-rest

AMQP:

- customer-service-mq
  https://github.com/tioback/customer-service-mq
- email-service-mq
  https://github.com/tioback/email-service-mq
- loyalty-service-mq
  https://github.com/tioback/loyalty-service-mq
- post-service-mq
  https://github.com/tioback/post-service-mq

18

Desenvolvimento: Heroku



19

Desenvolvimento: Obstáculos

Limites

- Uso de memória
- Arquivos x Conexões abertos
- Threads ativas
- Quantidade mensagens
- $$$

Dificuldades

- Experiência com as ferramentas
- Intermitência dos problemas
- Divergências entre Sistemas Operacionais
- Conhecimento das Arquiteturas

20

Desenvolvimento: Obstáculos

Sistema Operacional	Limite de Threads	Limite de Arquivos/Sockets	RAM
64-bit MacOS X 10.9	2.030	10.240	8 GB
64-bit Heroku Linux	256	1.024	512 MB
64-bit Windows 10	Ilimitado	*	16 GB

21

Desenvolvimento: Coleta de Dados - AMQP

```
private void signalUserCreation(int userId, CorrelationData correlationData) {
    rabbitTemplate.correlationConvertAndSend(String.valueOf(userId), correlationData);
}
```

```
private void setUpConfirmation() {
    rabbitTemplate.setConfirmCallback((receivedCorrelationData, ack, cause) -> {
        if (currentCorrelationDataKey == null) return;
        String[] parts = receivedCorrelationData.getId().split("-");
        if (currentCorrelationDataKey.equals(String.format("%s-%s-%s", parts[0], parts[1], parts[2]))) {
            average.accumulateAndGet(System.nanoTime() - Long.parseLong(parts[3]),
                    (n, m) -> (n + m) / (n == 0 || m == 0 ? 1 : 2));
            recCounter.incrementAndGet();
        } else {
            refCounter.incrementAndGet();
        }
    });
}
```

22

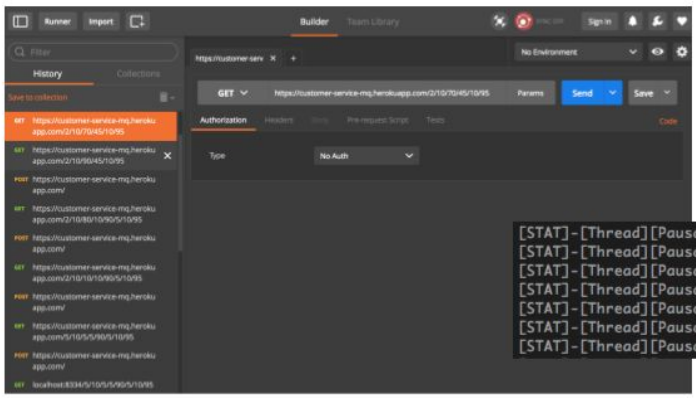
Desenvolvimento:
Coleta de Dados - REST

```
49    public void create_Customer() {
50        _doProcessing(); signalUserCreation(counter.incrementAndGet());
51    }
```

```
172   public void printStatistics(
173           int threads, int sleep) {
174   }
```

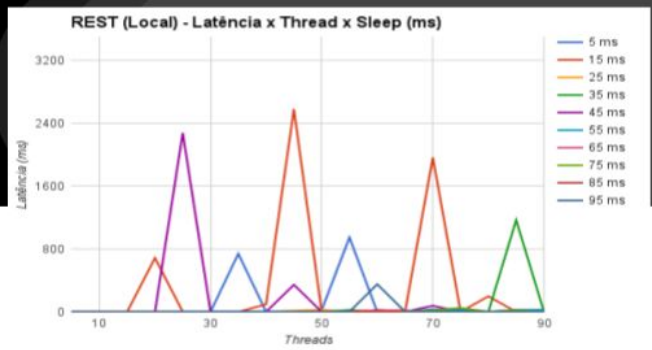
```
25    @Around("execution(* SpringCustomerService.create_Customer(..))")
26    public Object log(ProceedingJoinPoint pjp) throws Throwable {
27        long init = System.nanoTime(); counter.incrementAndGet();
28        try { return pjp.proceed(); } finally {
29            average.accumulateAndGet(System.nanoTime() - init, new LongBinaryOperator() {
30                public long applyAsLong(long n, long m) { return (n + m) / (n == 0 || m == 0 ? 1 : 2); }
31    }); } }
32
33    private final String LOG_FORMAT = "[STAT]-[Thread][Pausa][Registros][Latência]:\t%d\t%d\t%d\t%d";
34    @Before("execution(* SpringCustomerService.printStatistics(..)) && args(threads,sleep)")
35    public void print(JoinPoint jointPoint, int threads, int sleep) throws Throwable {
36        logger.info(String.format(LOG_FORMAT, threads, sleep, counter.getAndSet(0), average.getAndSet(0)));
37    }
```

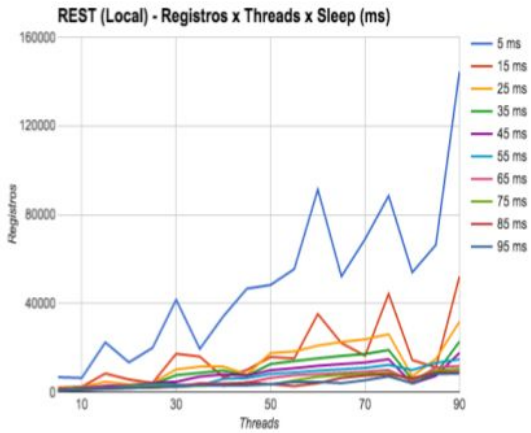

Desenvolvimento:
Execução dos Testes

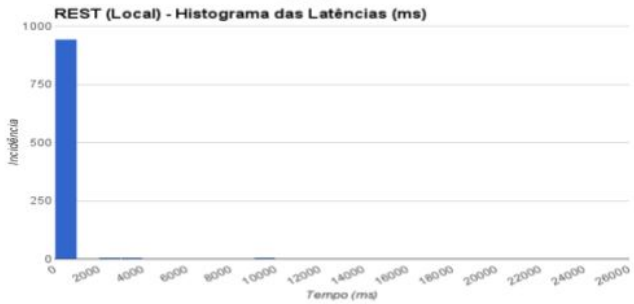



```
[STAT]-[Thread][Pausa][Registros][Latência]:    5    15   465 521436
[STAT]-[Thread][Pausa][Registros][Latência]:    5    15   885 563598
[STAT]-[Thread][Pausa][Registros][Latência]:    5    15  2800    455876
[STAT]-[Thread][Pausa][Registros][Latência]:    5    25  1730    814758
[STAT]-[Thread][Pausa][Registros][Latência]:    5    25  1650    373397
[STAT]-[Thread][Pausa][Registros][Latência]:    5    25   675 656792
[STAT]-[Thread][Pausa][Registros][Latência]:    5    25  1716    631342
```

Resultado: REST Local

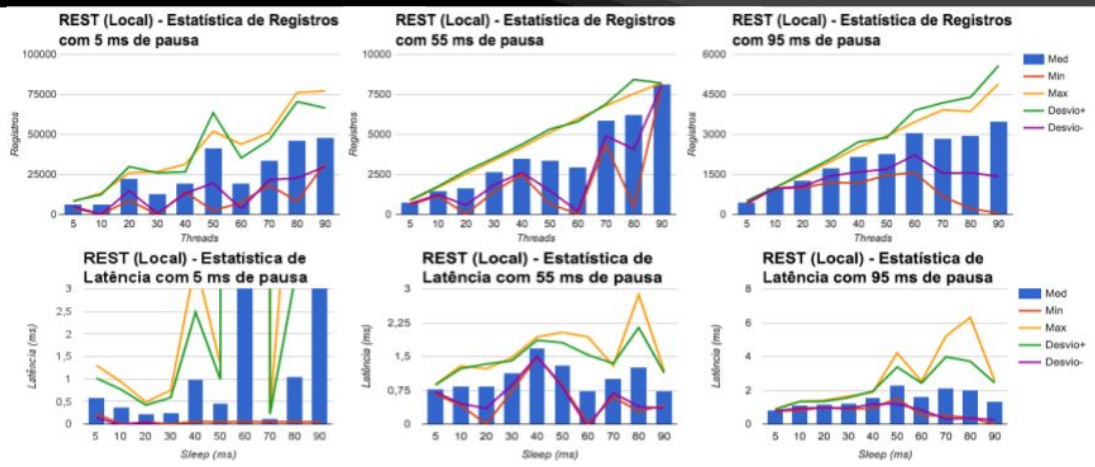






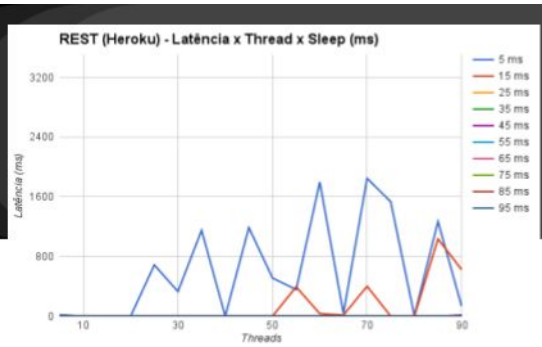
26

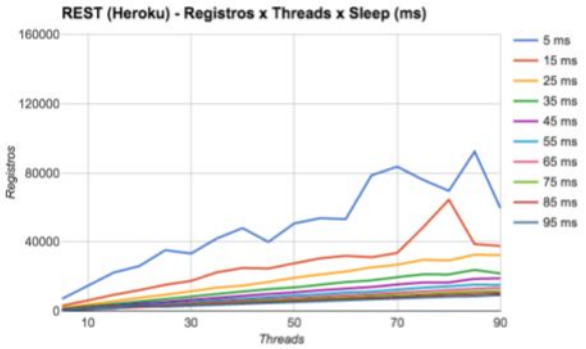
Resultado: REST Local

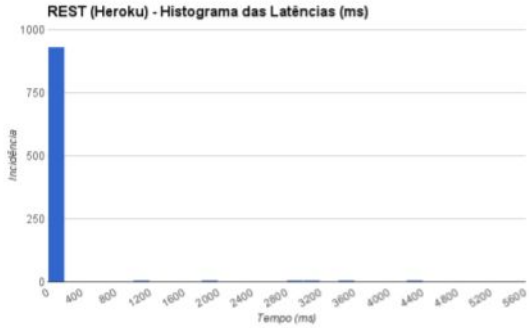


27

Resultado:
REST Heroku

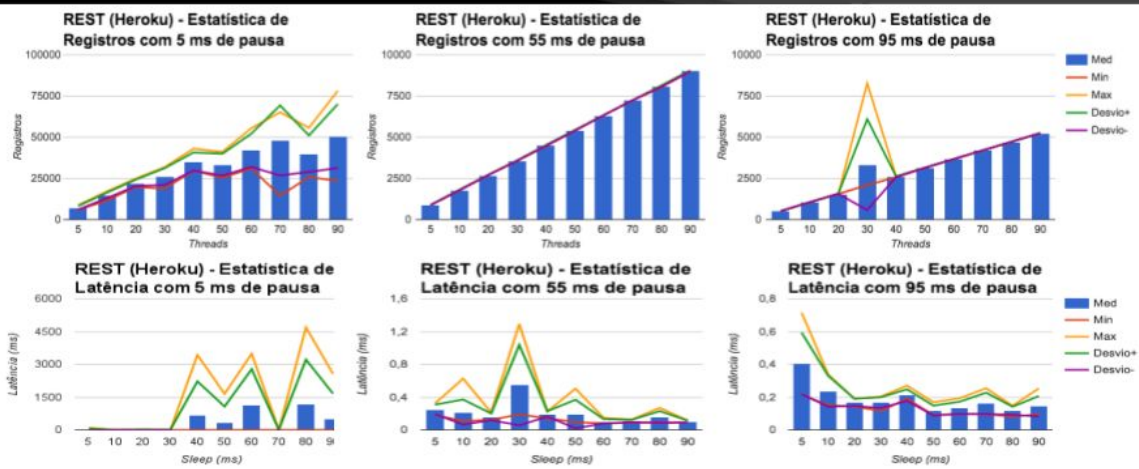






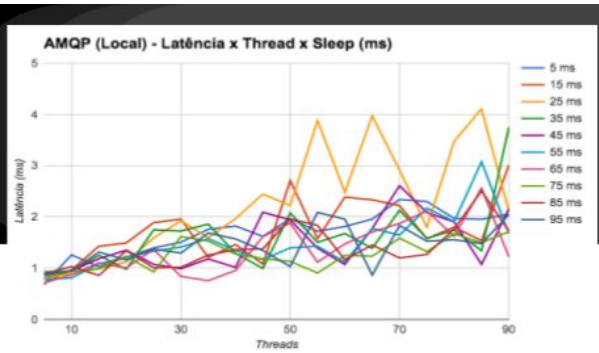
28

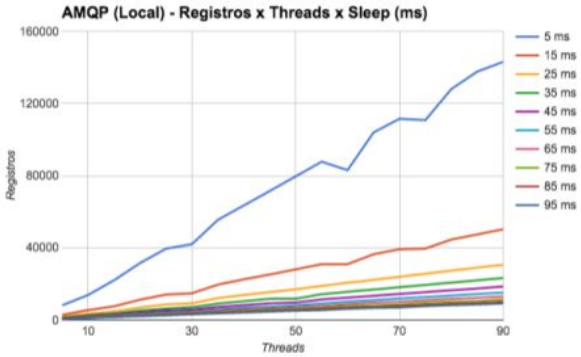
Resultado:
REST Heroku

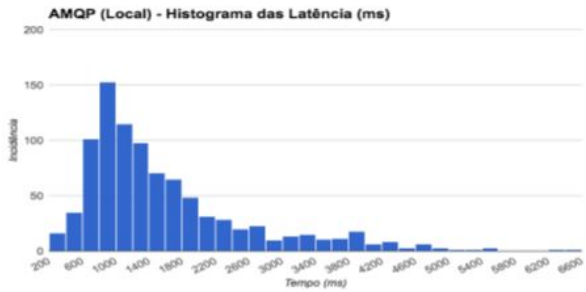


29

Resultado: AMQP Local

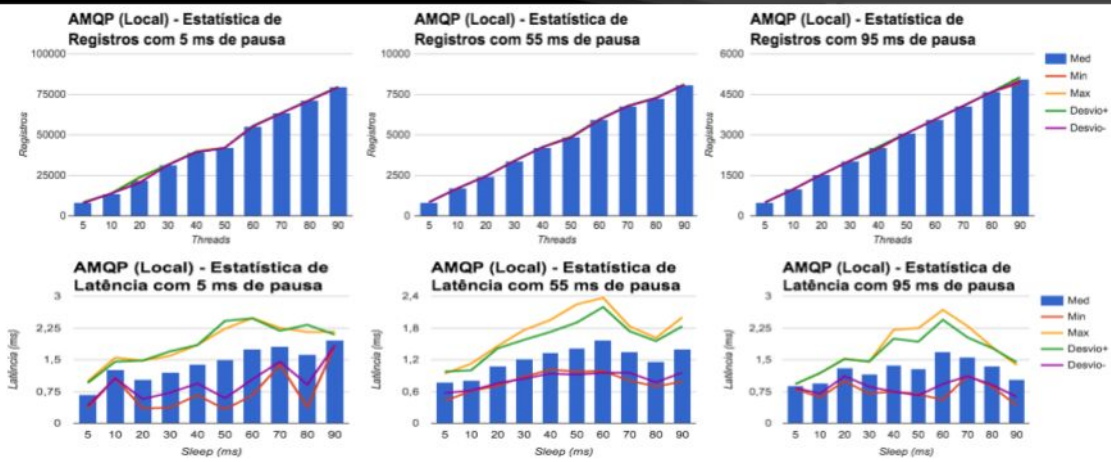






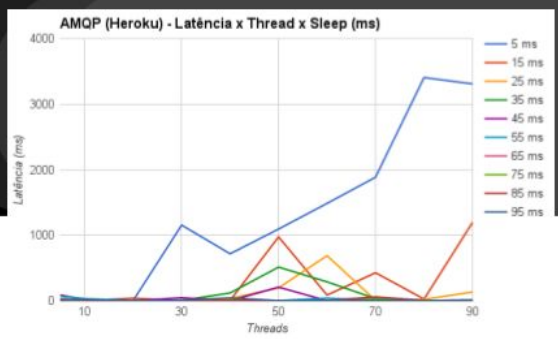
30

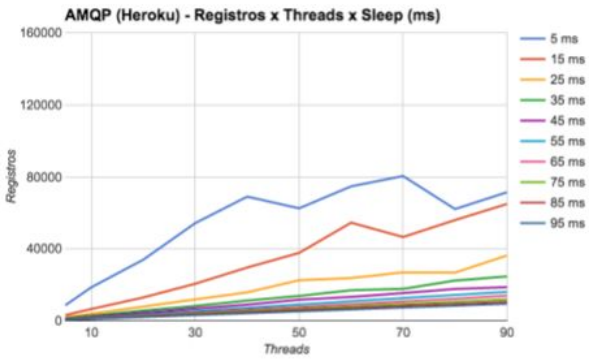
Resultado: AMQP Local

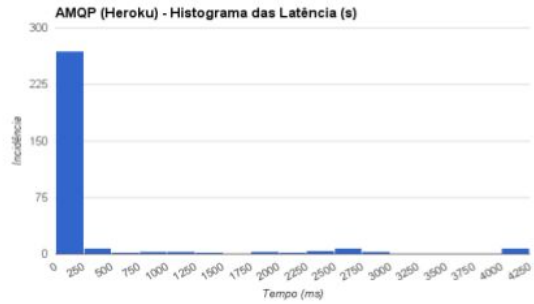


31

Resultado: AMQP Heroku

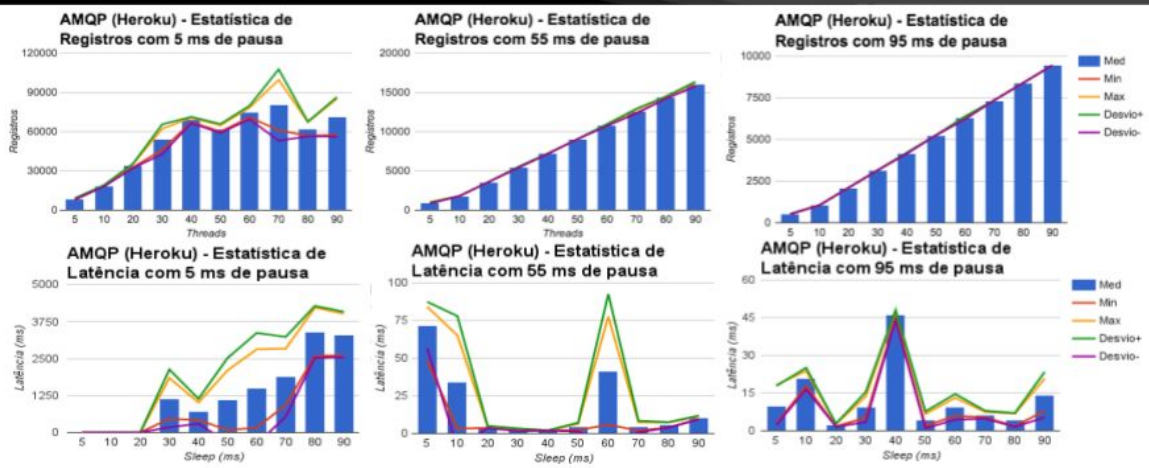






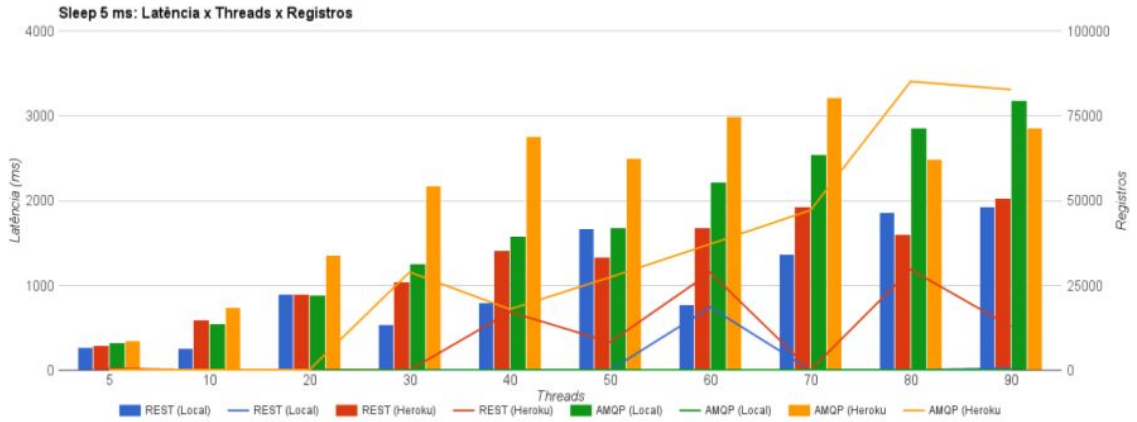
32

Resultado: AMQP Heroku



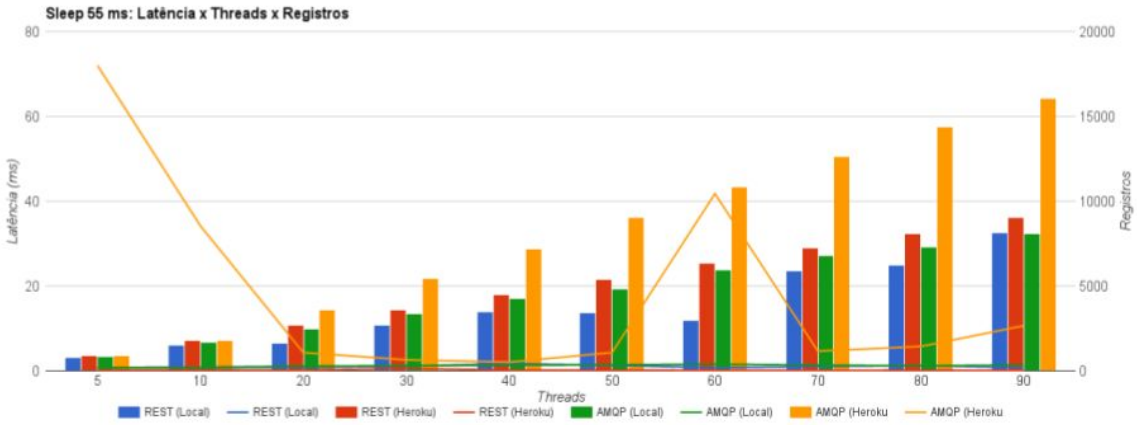
33

Resultado: Comparação



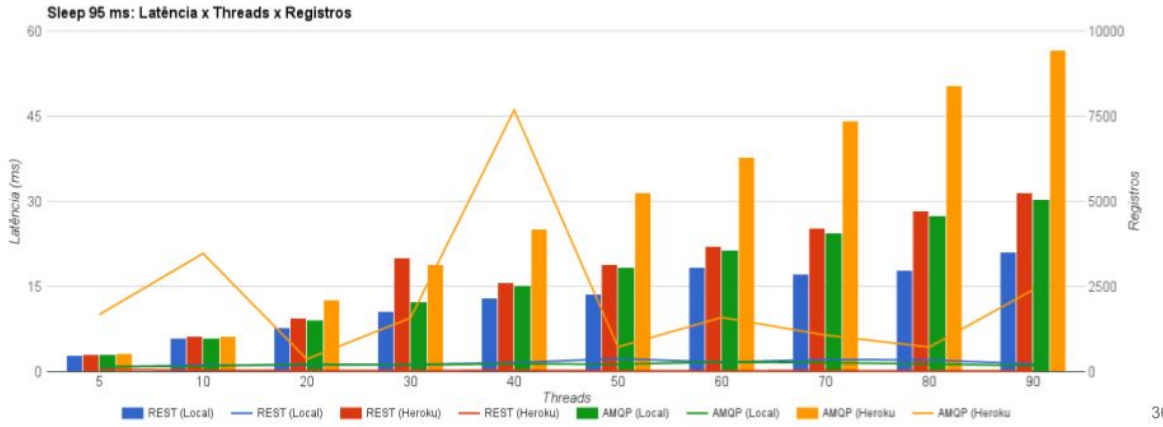
34

Resultado: Comparação



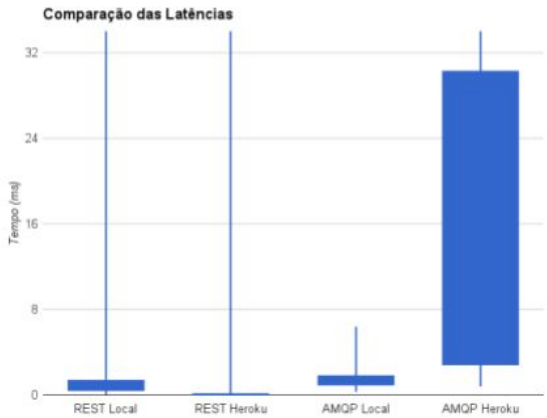
35

Resultado: Comparação



36

Resultado: Comparação



37

Resultado:
Comparação da Vazão e Latência

Critério	Mensageria (AMQP)		REST	
	Local	Heroku	Local	Heroku
Latência média (ms)	1,57	258,23	78,98	79,27
Vazão máxima	143.801	99.692	154.283	150.283
Vazão total	15.209.959	13.684.285	11.459.350	14.412.824

Tabela 3: Resumo dos Testes

38

Resultado:
Comparação da Complexidade

Critério	Mensageria (AMQP)	REST
Tempo de desenvolvimento	30 horas	18 horas
Arquiteturas de integração	Coreografada e Coordenada	Coreografada e Coordenada
Dependência de serviços externos	Sim	Não necessariamente, pode-se implementar um serviço específico

Tabela 4: Complexidade das técnicas

39

Resultado:
Comparação da Heterogeneidade

Tanto REST quanto AMQP suportam diversas linguagens:

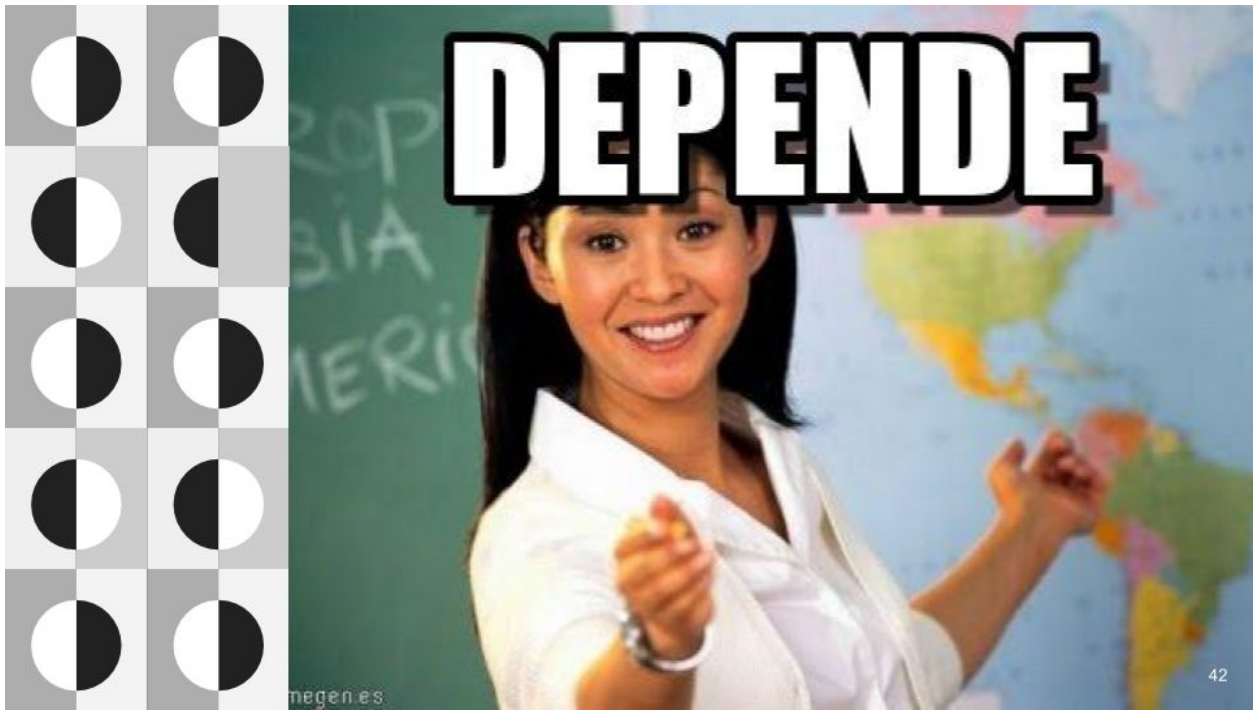
C/C++ Clojure COBOL Erlang Groovy on Grails
Haskell Java Lisp .Net Node.js Go
Objective-C/Swift OCaml Perl PHP Python Ruby Scala

40

Conclusão:

Qual arquitetura de integração apresenta um melhor retorno em desempenho, custo de desenvolvimento e consumo de recursos, dado um determinado conjunto de fatores?

41



42

Conclusão

- Vazão
  - AMQP = maior
- Latência
  - REST = menor
  - AMQP = mais consistente
- Heterogeneidade
- Requisições não atendidas
- Conformidade com Princípios de Microsserviços
- Complexidade

43

Trabalhos Futuros

- Utilizar outras linguagens (Go, Scala, Python, PHP, .Net, etc);
- Avaliar outras arquiteturas (SOAP, WebSockets, Apache Thrift);
- Avaliar outras combinações de rede;
- Avaliar outras métricas (tamanho do pacote, perda de dados, ociosidade do sistema, recuperação de erros, tamanho de código, tamanho de binário)
- Aprimorar análises estatísticas

95%

Perguntas?

45

FIM

Renato Pereira Back

renatopb@gmail.com

46