Implantação da arquitetura de micro serviços: do monolítico ao distribuído.

Pedro Augusto Dias de Vasconcelos[1], César Augusto Borges de Andrade[1]

[1]Faculdade JK - Especialização em Gestão de TI na Administração Pública
QI 03, Lotes 1 a 4, Avenida Samdu Norte, Taguatinga/DF - CEP: 72.135-030

{vasconcelos.pedro5, caborges72}@gmail.com

Resumo. *Tradicionalmente temos construídos sistemas em arquiteturas monolíticas, porém este tipo de arquitetura passou a não atender adequadamente necessidades de negócio. A arquitetura de micro serviço surgiu para tentar resolver algumas das dores criadas por sistemas monolíticos. Este artigo apresenta uma visão sobre este modelo arquitetural distribuído, suas características, benefícios e desvantagens e estabelece uma discussão o sobre boas práticas de modelagem de serviços, e tópicos sobre integração entre sistemas com intenção de servir como um ponto inicial para aqueles que desejam trabalhar ou saber algo mais sobre esta arquitetura.*

Palavras-chave: *Micro serviços, Arquitetura de software, Sistemas distribuídos, REST*

Abstract. *Traditionally we have built systems in monolithic architectures, but this type of architecture started to not adequately meet business needs. The micro-service architecture emerged to try to resolve some the pain created by monolithic systems. This article gives an insight into this distributed architectural model, its features, benefits and drawbacks, and establishes an argument about good practices in service modeling, and topics of systems integration intended to serve as a starting point for those wishing to work or know something more about this architecture.*

Keywords*: Micro services, Software Architecture, Distributed Systems, REST*

1 INTRODUÇÃO

O termo micro serviços surgiu recentemente no mundo de desenvolvimento e arquitetura de software. A ideia de aplicações modeladas como pequenos serviços autônomos tem sido frequentemente mais adotada com o passar do tempo para se resolver problemas oriundas de softwares que adotam uma arquitetura mais antiga, chamada de arquitetura monolítica. Este estilo arquitetural mais moderno composto por micro serviços propõe aliviar as dores de arquitetos e desenvolvedores de software no que tange a implantação, a disponibilidade, a escalabilidade, e alguns outros aspectos que serão abordados com mais profundidade neste trabalho.

Tendo em vista todos os problemas que arquiteturas tradicionais podem causar, a tentativa de adotar uma arquitetura baseada em micro serviços tem aumentado nos últimos anos por todo o mundo. No entanto, ao tentar aderir este tipo de arquitetura, é comum esbarrar em alguns problemas.

Os interessados neste novo modelo arquitetural tem dificuldade a ter acesso à informação sobre o assunto pois o volume de bibliografia disponível ainda é pequeno; Boa parte da bibliografia atual sobre micro serviços está na língua inglesa, o que pode trazer dificuldades àqueles que não dominam o idioma; As melhores referências bibliográficas no assunto possuem algum conteúdo comum, mas cada uma possui um diferencial, uma visão particular do autor que passou pela experiência na adoção deste estilo arquitetural. Isso gera o problema de não haver algo que centralize essas experiências o que necessita a leitura de diversas fontes diferentes; Muitas vezes o conteúdo abordado nos trabalhos trata de questões muito introdutórias o que acabar por não ajudar muito aqueles que tem intenção de implantar a arquitetura de micro serviços nas empresas ou órgãos que trabalham;

Tendo em vista os problemas citados, este trabalho tem o intuito de introduzir a o conceito de arquitetura baseada em micro serviços, o por que adotar este modelo, evidenciando as suas vantagens e desvantagens. Em seguida, iremos discutir sobre tecnologias disponíveis hoje no mercado que permitem a integração entre serviços e apresentar algumas boas práticas sobre como realizar essa integração de forma sustentável. Por fim, será mostrada uma proposta sobre como tentar utilizar este modelo arquitetural para resolver problemas de arquiteturas monolíticas, decompondo o monolítico em serviços menores.

O restante deste trabalho estar organizado da seguinte forma: A seção 2 apresenta os conceitos básicos para compreender este trabalho de forma completa; A seção 3 apresenta os trabalhos relacionados; A seção 4 trata da ideia central deste artigo; e a seção 4 conclusões e considerações finais.

2 CONCEITOS BÁSICOS

Para entender pode introduzir a arquitetura baseada em micro serviços e aprofundar um pouco a discussão a respeito deste assunto é essencial termos alguns conceitos em mente. Micro serviços é um estilo arquitetural, mas o que é arquitetura no contexto de software? Segundo Fowler (2000) o termo arquitetura: "...transmite uma noção dos elementos centrais do sistema, as peças que são difíceis de mudar. Uma fundação em que o resto deve ser construído". Segundo Garlan e Shaw (1994) a arquitetura de software envolve uma série de questões que vão:

> além dos algoritmos e das estruturas de dados da computação. A projeção e a especificação da estrutura geral do sistema emergem como um novo tipo de problema. As questões estruturais incluem organização total e estrutura de controle global; protocolos de comunicação, sincronização e acesso a dados; atribuição de funcionalidade a elementos de design; distribuição física; composição de elementos de design; escalonamento e desempenho; e seleção entre as alternativas de design. (GARLAN; SHAW, 1994. p. 2)

O estilo arquitetural a vasta maioria dos softwares construídos até hoje segue o que chamados de arquitetura monolítica, descrita da seguinte forma:

> Ela deve suportar uma variedade de diferentes cliente incluindo browsers de desktops, browsers de dispositivos móveis e aplicações nativas de aplicações móveis. A aplicação pode também expor uma API para ser consumida por terceros. Ela pode também integrar com outras aplicações através de web services ou um broker de mensagens. A aplicação lida com requisições (requisições HTTP e mensagens) executando lógica de negócio; acessando uma base de dados; trocando mensagens com outros sistemas; retornando uma resposta HTML/JSON/XML. (RICHARDSON, 2014)

A arquitetura de micro serviços surgiu como uma alternativa a arquitetura monolítica como uma abordagem que propõe colaboração entre serviços coesos e de baixo acoplamento. Silveira (2012, p. 80) define acoplamento da seguinte forma: “Acoplamento é quando dois elementos estão amarrados entre si e quanto as alterações no comportamento de um afetam o de outro.”. Um serviço nada mais é do que um componente que segundo Silveira (2012, p. 80) deve possuir alta coesão: “... o que significa garantir que suas responsabilidades sejam relacionadas e façam sentido juntas, ou seja, que não haja responsabilidades desconexas dentro de um mesmo componente.”

A arquitetura de micro serviços depende muito de uma solução de plataforma como serviço (PaaS – Plataform as a service):

> Paas e é um conjunto de serviços direcionados a desenvolvedores que os ajudam a desenvolver, testar aplicações sem ter que se preocupar com a infraestrutura por baixo. Desenvolvedores não querem ter que se preocupar em provisionar servidores, armazenamento de dados e backup associados com o desenvolvimento e execução da aplicação. Eles querem escrever código, testar a aplicação, executar a aplicação, e ser capaz de realizar mudanças e corrigir bugs continuamente. Toda parte por traz que diz respeito a configurar servidores deve ser feito automaticamente em background, e essa é a promessa do PaaS. (BUTLER, Fev. 2013).

3 TRABALHOS RELACIONADOS

Uma das principais fontes para aqueles que nunca ouviram falar em micro serviços é o artigo escrito por Fowler (2014). Em seu trabalho, é apresentado um conceito de de micro serviço, suas características e uma visão futura sobre a arquitetura. A principal vantagem deste trabalho é que ele é curto, apenas com essa leitura é possível ter uma noção do que trata o assunto. No entanto, como o próprio título do trabalho diz, ele trata apesar de definir o termo e fornecer uma visão introdutória e teórica sobre o assunto, o que é insuficiente para aqueles que desejam realmente implementar esta nova abordagem arquitetural.

Um outro trabalho interessante é o artigo de Richardson (2014). Este artigo, assim como o trabalho de Fowler (2014) a respeito de micro serviços, também oferece uma visão introdutória sobre o assunto, mas diferentemente de Fowler (2014) , Richardson (2014) vai um passo além ao discutir questões interessantes sobre integração entre serviços e também discute sobre formas de se migrar de uma aplicação monolítica em uma aplicação distribuída baseada em micro serviços. Ainda assim, este trabalho é um pouco superficial para alguém que deseje por os conceitos apresentados em prática.

Newman (2015) por sua vez, escreveu livro que, diferentemente dos trabalhos mencionados anteriormente, é bastante completo. Apresenta uma visão bem detalhada sobre várias questões relevantes sobre a arquitetura de micro serviços. O porém deste trabalho no entanto é sua disponibilidade apenas na língua inglesa e o fato de ser uma leitura bastante extensa, o que pode ser um obstaculo para alguns.

Este trabalho se propõe a escrever uma leitura introdutória a arquitetura de micro serviços, mas que seja completa o suficiente para começar a implementar projetos na prática sem que seja necessário um estudo muito intensivo do assunto.

4 IDEIA CENTRAL

4.1 SISTEMAS MONOLÍTICOS

Os primeiros softwares escritos foram projetados para rodar em grandes computadores chamados mainframes. Nesta época não existiam ainda computadores pessoais e todos os programas eram executados nesta grande máquina. Os sistemas que rodavam neste tipo de computador tinham uma arquitetura que pode ser chamada de monolítica. Segundo Stephens (2015, p. 94), em uma arquitetura monolítica, um único programa faz tudo. Ele é responsável pela execução da interface gráfica, acesso a dados, execução de regras de negócio e tudo mais que possa ser necessário.

Este tipo de arquitetura tem algumas vantagens. Como todas as partes do sistemas estão rodando juntas em um único processo, não há a necessidade de integração com outros sistemas através da rede utilizando de protocolos de comunicação complexos, nem preocupação com vários aspectos de segurança por exemplo. No entanto, este alto grau de acoplamento entre os módulos do sistema faz com que ele seja muito inflexível o que compromete significativamente sua manutenibilidade e o torna muito mais difícil de se evoluir.

Para resolver algumas das dificuldades encontradas em arquiteturas monolíticas os sistemas passaram a ser quebrados em duas grandes partes. A primeira que executa uma série de funções como a apresentação da interface gráfica e é cliente dos dados fornecidos pela segunda parte, que é o servidor. A comunicação entre os 2 blocos desta arquitetura é feita através de uma rede, hoje comumente a internet conforme ilustra a figura 1.

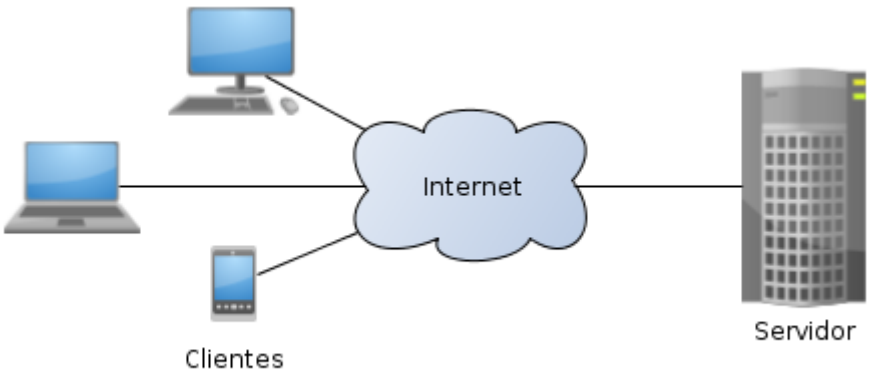

Figura 1. Arquitetura cliente servidor .

Fonte: (Wikipedia, 2010)

Essa arquitetura é conhecida como cliente/servidor em 2 camadas. Stephens (2015, p. 95) afirma que este modelo possibilita a execução de vários clientes simultaneamente compartilhando os mesmos dados. A grande desvantagem desta organização, contudo, é que ela cria uma dependência muito grande dos sistemas clientes com o servidor, no caso o sistema gerenciador de banco de dados (*SGBD*). Para resolver este problema foi adicionada mais uma camada física entre o cliente e o *SGBD*.

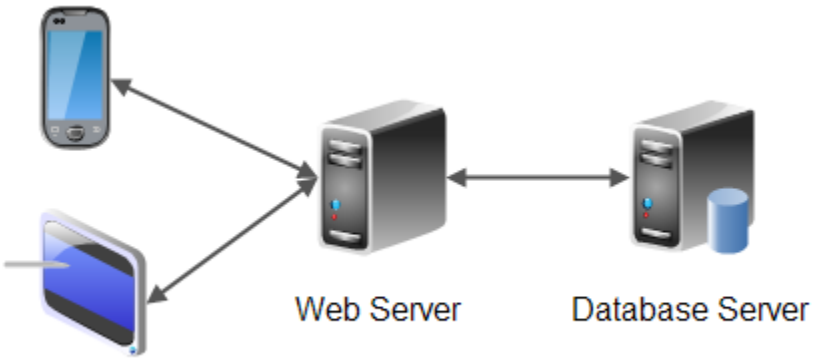

Figura 2. Arquitetura cliente servidor em 2 camadas.

Fonte: (JENKOV, 2014)

A figura 2 ilustra a arquitetura cliente/servidor em 3 camadas. Esta organização de camadas é a mais utilizada até hoje e resolve a maioria dos problemas de sistemas de pequeno, médio e grande porte, mesmo existindo alguns problemas.
Segundo Fowler (2014) aplicações corporativas atuais são frequentemente construídas seguindo esta arquitetura cliente/servidor em 3 camadas. A primeira camada, cliente, é composta de páginas HTML que utilizam CSS e Javascript para sua dinamização e que rodam no navegador na máquina do usuário. A segunda camada é uma aplicação do lado do servidor e é responsável por lidar com requisições HTTP, executar regras de negócio e fazer o acesso aos dados servidor pela terceira e última parte do modelo. A última camada executa frequentemente um banco de dados relacional onde todos os dados persistentes da aplicação ficam armazenados e podem ser compartilhados com diversas aplicações. A aplicação que roda no servidor é um sistema monolítico, um único executável lógico.

Fowler (2014) diz que construir uma aplicação monolítica como descrito acima é uma abordagem natural e pode ser bem sucedida. Muito embora todas as funcionalidades do sistema rodem em um único processo, recursos da linguagem de programação utilizada, como *namespaces,* módulos, classes e funções, são muito utilizadas para dividir a aplicação em partes lógicas menores.

> ...essas aplicações, apesar de terem um design modular, são implantadas como um único pacote (no caso de aplicações Java são utilizados arquivos de extensão WAR ou EAR) ou um único diretório (como é o caso de aplicações escritas em linguagens de script como Ruby, Python ou PHP) da aplicação em servidores de aplicação JEE e servidores Web (Apache, Nginx) respectivamente. (RICHARDSON, 2014)

Aplicações monolíticas tem como vantagens a facilidade de desenvolvimento, já que não é necessário realizar integração entre módulos e as IDEs facilitam a implementação de aplicações deste tipo. São fáceis de testar pois todas as funcionalidades estão contidas dentro do mesmo executável, e por fim, são fáceis de implantar pois a aplicação é composta por apenas um pacote ou diretório.

Como afirmado por Richardson (2014), este tipo de abordagem se mostra muito eficiente para aplicações de pequeno e médio porte, mas a medida que as aplicações se tornam maiores algumas dificuldades começam a surgir.

Aplicações grandes demais são mais difíceis de se entender e por isso a manutenibilidade pode ficar prejudicada. Outro aspecto é que sistemas de grande porte podem criar obstáculos para novas e frequentes implantações. Isso acontece por que caso seja necessário realizar uma mudança em apenas uma das várias funcionalidades que podem existir dentro deste sistema, será necessário realizar a construção e implantação de uma nova versão de toda a aplicação ao invés de uma unidade menor, o que pode ser complexo, demorado, evolver diversos papéis diferentes da equipe e ainda requerer longos ciclos de teste. Por fim, arquiteturas monolíticas desencorajam a adoção de novas tecnologias, pois na maioria das vezes não é possível integrar um novo *framework* ou tecnologia a um sistema existente sem causar grandes impactos e sem retrabalho excessivo, o que acaba por restringir os desenvolvedores e arquitetos do sistema às tecnologias escolhidas no início do projeto e assim impossibilitando sua evolução tecnológica.

4.2 ARQUITETURA DE MICRO SERVIÇOS

A busca por soluções para os problemas apresentados anteriormente levou a criação de um estilo arquitetural baseado em serviços conhecida hoje como arquitetura de micro serviços. Este estilo arquitetural pode ser definido seguinte forma:

> Arquitetura de micro serviços é um conceito arquitetural que tem como objetivo desacoplar a solução decompondo funcionalidades em serviços discretos. Pense nele como a aplicação do princípio SOLID em um nível arquitetural, ao invés de classes você tem serviços. (HUGHES, 2013).

Newman (2015) conceitua "micro serviços" de forma mais simples: "serviços pequenos e autônomos que trabalham juntos.". Serviços devem ser pequenos de forma que cada um desempenhe apenas uma responsabilidade, como é recomendado pelo princípio da responsabilidade única. Este método de definição de escopo não é diferente para sistemas monolíticos. Uma classe deve ser coesa de tal forma que tenha apenas uma responsabilidade, caso uma classe tenha mais de uma responsabilidade:

> …então as responsabilidade ficam acopladas. Mudanças em uma responsabilidade podem dificultar ou inibir a habilidade da classe de se encaixar com as outras. Esse tipo de acoplamento leva a projetos frágeis que quebram de formar inesperadas quando eles mudam. (MARTIN, 2003, p. 110).

Este método de definição de escopo não é diferente para sistemas monolíticos. Nestes sistemas, desenvolvedores tentam constantemente manter o código coeso utilizando abstrações lógicas como módulos e classes para agrupar as responsabilidades. Da mesma forma, micro serviços utilizam este mesmo princípio para definir serviços independentes. A interface destes serviços é focada nas interfaces de negócio. Manter o serviço focado em uma interface explícita evita que ele cresça demais evitando todos os problemas que isso poderia acarretar.

Mesmo seguindo o princípio da responsabilidade única ainda pode ficar uma dúvida sobre o quão pequeno deve ser um micro serviço. Richardson (2014) afirma que pessoas da comunidade recomendam que micro serviços tenham entre 10 e 100 linhas de código, no entanto seguir tais regras estritamente podem trazer problemas. Um comportamento perfeitamente normal é que alguns serviços tenham seu tamanho muito reduzido enquanto outros acabarão ficando maiores. O tamanho dos micro serviços pode varias bastante, esse realmente não é o ponto principal da arquitetura de micro

serviços, o importante realmente é tentar focar na resolução dos problemas de sistemas monolíticos.
Para Newman (2015), micro serviços devem ser autônomos: “Nosso micro serviço é uma entidade separada. Ele pode ser implantado como um serviço isolado em uma plataforma como serviço (PAAS), ou ele pode ser seu próprio processo do sistema operacional.”. Embora isolados, existe comunicação entre serviços, mas ela deve acontecer pela utilização de chamadas pela rede para garantir a separação física entre os serviços e evitar os perigos do alto acoplamento. Micro serviços devem ser projetados de tal forma que suas mudanças não causem impacto em seus consumidores ou outros serviços. A API (application programming interface) dos serviços deve ser pensada como aquilo que queremos expor publicamente, de tal forma que o mínimo possível do funcionamento interno seja conhecido, para garantir o baixo acoplamento. Isto significa entre outras coisas, utilizar uma tecnologia que seja agnóstica, ou seja que não exponha detalhes de sua implementação na API. A meta final é conseguir mudar um serviço e implantá-lo sem que nada mais seja afetado.

4.3 BENEFÍCIOS

4.3.1 Heterogeneidade Tecnológica

A arquitetura de micro serviços possibilita heterogeneidade de tecnologias. Por serem pequenos e autônomos, cada micro serviço pode independentemente adotar uma pilha de tecnologia diferente do outro sem que isso se torne um problema. Desenvolvedores tem mais liberdade para adotar a tecnologia ideal para um determinado trabalho. Newman (2015) exemplifica que em casos em que é desejável adotar diferentes tipos de bancos de dados para cada tipo de dado a ser persistido. Por exemplo, para uma aplicação de rede social podemos persistir as interações de usuários em um banco de dados orientado a grafos, enquanto as postagens dos usuários podem ser armazenadas em um banco de dados orientado a documentos e as fotos destes usuários armazenados em um banco de dados onde seja possível armazenar dados binários. Tal exemplo pode ser divido em 3 micro serviços independentes trabalhando com pilhas de tecnologias completamente diferentes entre si conforme ilustrada a figura 3.

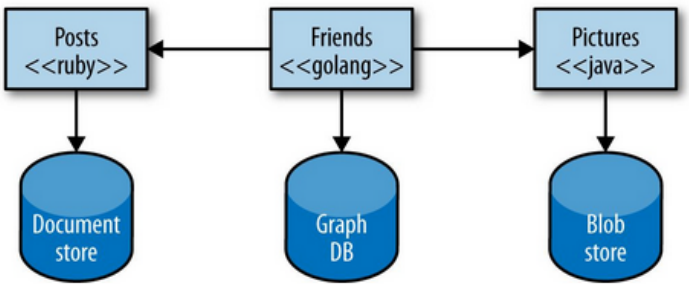

Figura 3. Persistência heterogênea.

Fonte: (NEWMAN, 2015, p 4)

4.3.2 Resiliência

Em um sistema monolítico, composto de várias funcionalidades diferentes, se uma delas falhar, o sistema inteiro falha, de tal forma que a única forma de evitar a falha do

sistema completo é a replicação dele em diversas máquinas diferentes. No caso de micro serviços, a independência entre eles o sistema fique mais tolerante a falha total do sistema, pois as falhas são restritas a funcionalidades isoladas. No caso de uma parte do sistema composto por um conjunto de micro serviços cair, a outra parte do sistema não é afetada, restringindo o problema ao conjunto de funcionalidades providas pelos micro serviços que ficaram fora do ar.

4.3.3 Escalabilidade

Suponha o caso de uma funcionalidade do sistema muito utilizada por usuários que precisa ser escalada. Se essa funcionalidade fizer parte de uma aplicação monolítica, para escalá-la horizontalmente deve-se fazer a cópia de toda a aplicação em máquinas diferentes, mesmo que exista a necessidade de escalar apenas uma funcionalidade. Ao utilizar micro serviços, como a funcionalidade é implantada de forma independente como um pacote ou diretório único, isso faz com que seja possível replicar a funcionalidade desejada em uma quantidade maior de máquinas ou mesmo implantá-la em máquinas mais potentes, enquanto as demais funcionalidades podem rodar em uma quantidade menor de máquinas ou em máquinas com hardware menos poderoso. A figura 4 exemplifica bem a diferença entre a implantação de uma aplicação monolítica (a esquerda) e de um sistema baseado em micro serviços (a direita).

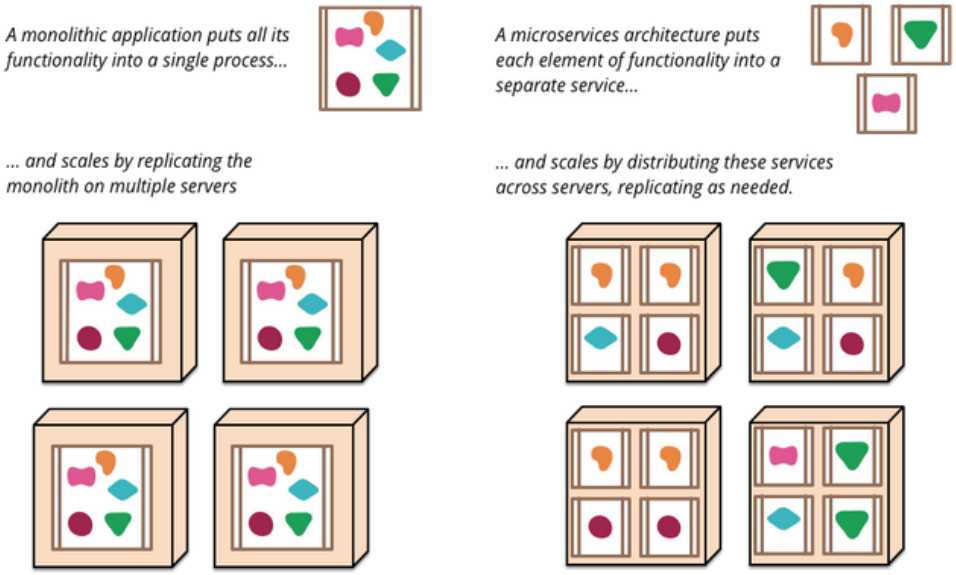

Figura 4. Implantação monolítica vs micro serviços

Fonte: (FOWLER, 2014)

4.3.4 Facilidade de Implantação

Realizar a implantação de grande aplicação monolítica pode ser complicado, pois uma atualização pequena no software pode implicar em um grande impacto na aplicação como um todo tornando assim a atividade muito arriscada. Por conta da possibilidade impacto e altos riscos, a implantação de uma aplicação monolítica acaba ocorrendo menos frequentemente do que deveria. A demora na implantação de uma aplicação pode

trazer uma série de problemas para o negócio. Em geral, quanto maior o intervalo entre lançamentos de um software, maior o risco de alguma coisa dar errada. A implantação de um micro serviço é feita de forma independente, por isso, uma alteração no serviço pode ser feita sem o medo de grandes impactos. No caso de um erro pode-se facilmente voltar o micro serviço em questão para versão anterior. Essa segurança faz com que as funcionalidades possam ser implantadas mais facilmente e assim sendo lançadas mais rapidamente para os clientes.

4.3.5 Alinhamento Organizacional

Grandes equipes são mais difíceis de gerenciar e frequentemente são menos produtivas que equipes pequenas. Sistemas monolíticos com uma grande base de código necessitam grandes equipes para construir e dar manutenção a eles. Richardson (2015) afirma que utilizando micro serviços é mais fácil escalar o desenvolvimento, pois como um micro serviço só precisa de uma pequena equipe para mantê-lo é possível combinar vários pequenos times desenvolvendo cada time uma funcionalidade isolada. Cada time tem autonomia para escrever o seu serviço e implantá-los independente dos outros.

4.3.6 Composabilidade

Em uma corporação que possui um conjunto de grandes sistemas monolíticos, é muito comum ver duplicação de funcionalidades implementadas dentro destes sistemas. Esta duplicação acontece por uma série de motivos, mas um fator determinante é a dificuldade de integração de funcionalidades existentes. A arquitetura de micro serviços é baseada na integração de serviços heterogêneos. A facilidade de utilização de serviçõe existentes faz com que novas possibilidades de uso destes serviços surjam além daquelas que foram imaginadas originalmente. Um cliente para celular por exemplo prover novas funcionalidades aos usuários finais apenas compondo serviços já existentes que originalmente foram projetados para serem consumidos por outros clientes. Em arquiteturas monolíticas onde os serviços oferecidos possuem granularidade muito menor, tal tipo de composição é impossível.

4.3.7 Facilidade de Substituição

Outro caso muito comum em grandes empresas são sistemas monolíticos legados que ninguém conhece todas as regras de negócio. Não é raro encontrar desenvolvedores que tem medo de substituir ou mesmo dar manutenção a esses sistemas. Utilizando micro serviços este tipo de problema tende a não acontecer pois os serviços são pequenos e de fácil entendimento. Equipes se sentem mais a vontade em reescrever ou mesmo jogar fora todo um serviço pois os custos de substituição são bem menores quando comparado ao custo de substituição de um grande sistema monolítico.

4.4 DESVANTAGENS

4.4.1 Complexidade Adicional de Sistemas Distribuídos

Um sistema baseado na arquitetura de micro serviços é muito mais complicado de se escrever quado comparado a um sistema monolítico. Isso de dá por conta da complexidade inerente a sistemas distribuídos, como a necessidade de implementar a comunicação entre processos diferentes, geralmente separados pela rede, o pouco suporte de IDEs para desenvolver este tipo de sistemas e também a dificuldade de implementação de testes automatizados.

4.4.2 Complexidade Operacional

Um sistema monolítico que roda em apenas um processo do sistema operacional quando quebrado em micro serviços passa a rodar em vários processos diferentes, que devem ser administrados pela equipe de infraestrutura de uma empresa. O fato de existirem muitas aplicações diferentes dificulta o processo. O ideal é adotar uma ferramenta de PAAS (Plataform as a service) para prover automatização para simplificar a complexidade operacional gerada. Implantação de funcionalidade que abrangem vários serviços pode ser complexo. Embora a arquitetura de micro serviços possibilitar uma frequência maior de implantações, estas implantações podem se tornar complexas quando envolvem funcionalidades que permeiam vários micro serviços distintos. Se cada micro serviço for responsabilidade de equipes de desenvolvimento diferentes, deve-se envolver as equipes de cada micro serviço na atividade de implantação para ter um trabalho coordenado.

4.4.3 Momento Certo de Adotar este Estilo Arquitetural

Sistemas baseados em arquitetura de micro serviços são mais complexos, e por isso a produtividade de desenvolvimento, quando adotada esta arquitetura, geralmente é menor quando comparada com a produtividade de desenvolvimento de sistemas monolíticos. Em determinas situações é preferível adotar uma arquitetura convencional para que se possa entregar funcionalidades mais rápido. É importante frisar que o momento de se migrar de uma arquitetura monolítica para uma arquitetura distribuída é essencial para ter sucesso. Se houver muita demora ao se adotar a arquitetura de micro serviços, é possível que a decomposição de serviços grandes em serviços menores se torne muito mais custosa.

4.5 MODELANDO BONS SERVIÇOS

Agora que sabemos bem qual o conceito de arquitetura de micro serviços, quais os benefícios podemos alcançar implantando este modelo arquitetural e as desvantagens do modelo, podemos devemos discutir sobre o que como podemos fazer para modelar bons serviços.

É muito importante garantir que os nossos serviços tenham baixo acoplamento e alta coesão. Para obter baixo acoplamento devemos modelar os serviços de tal forma que mudanças em um determinado serviço possam ser feitas e o serviço seja implantado sem que nenhum outro serviço ou parte do sistema se quer fique sabendo. Além disso é importante que todos os comportamentos relacionados estejam agrupados em um local apenas, e seguindo o mesmo raciocínio, é importante que outros assuntos que não tenham relação com o assunto do de um serviço fiquem em outro local qualquer. O que acabamos de descrever é a alta coesão. Alta coesão é desejada pois o desejável é que alterações de um determinado comportamento sejam feitas em local centralizado. Fazer mudanças em muitos locais diferentes é mais lento, implica em maior é complexidade e maior risco, por isso é necessário que os serviços tenham baixo acoplamento e alta coesão.

Evans (2004) apresenta uma série de técnicas sobre como modelar sistemas que representam o mundo real de forma eficiente. Os conceitos sobre como modelar serviços não são diferentes. Dos vários conceitos apresentados por Evans um chama atenção, que é o conceito de contextos delimitados, ou em inglês B*ounded Contexts*. Conceituado por Fowler (2014) da seguinte maneira:

> Contexto delimitado é um padrão central em Domain Driven Design. É o foco da seção estratégica de projeto do DDD, que é sobre como lidar com grandes modelos e equipes. DDD lida com grandes modelos dividindo-os em contextos delimitados e sendo explícito sobre suas inter-relações. *(Fowler, 2014)*

É importante delimitar os diferentes contextos de um sistema. Por exemplo, um sistema que atende a um super mercado pode lidar com diversos contextos diferentes como estoque, ponto de venda, crediário, folha de pagamento, entre outros. Por mais que todos esses contextos estejam ligados ao conceito de super mercado, eles devem ser modelados como contextos delimitados diferentes onde cada um possui uma responsabilidade específica delimitada por uma fronteiras explícitas.

Os serviços devem ser organizados ao redor de capacidades de negócio. Para explicar como funciona esse tipo de organização é mais fácil começar explicando o que é um sistema que foge a essa organização, que é o caso mais comum. Imagine um sistema monolítico com uma arquitetura de multi-camadas. Esse sistema possui uma camada de apresentação que roda no browser utilizando html, css e javascript, uma camada negocial que roda em um servidor de aplicação Java EE, e uma camada de persistência que roda em um SGBD relacional. Para desenvolver e manter este sistema é comum vermos a criação de equipes diferentes, uma parada cada camada tecnológica. No nosso exemplo seriam então três equipes, a equipe que lida com a camada de apresentação, a equipe que lida com a lógica negocial e a equipe que lida com a persistência. A figura 5 demonstra essa divisão de equipes.

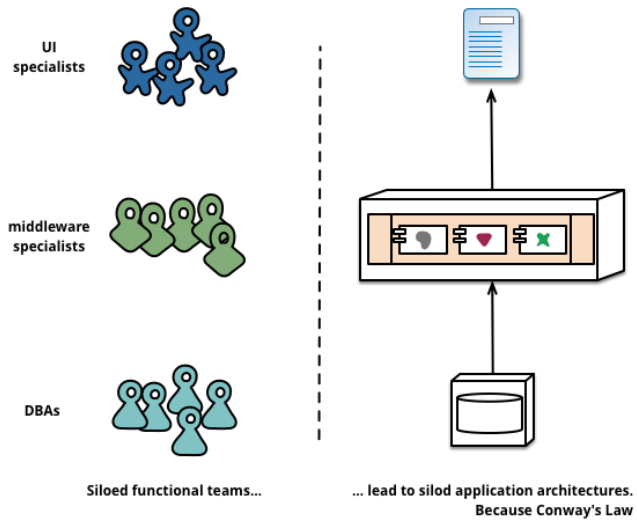

Figura 5. Equipes organizadas por critérios tecnológicos.

Fonte: (Fowler, 2014)

O problema desta abordagem é que se for necessário o desenvolvimento de uma nova funcionalidade ou mesmo a manutenção de uma já isso levará a necessidade de um esforço que envolva necessariamente as 3 equipes. Em um primeiro momento essa pode parecer até uma boa abordagem de como se dividir equipes pode parecer adequada, mas ela pode levar a problemas.

O que a arquitetura de micro serviços prega é que equipes sejam divididas por capacidades de negócio. A ideia é que o serviço envolva toda pilha tecnológica necessária para que o valor de negócio possa ser entregue ao usuário. Isso envolve a camada de apresentação, lógica de negócio, possíveis integrações, persistência e qualquer outra questão necessária para implementar o serviço necessário. Isso implica que as equipe sejam multifuncionais, tendo membros de cada especialidade necessária como ilustrado na figura 6.

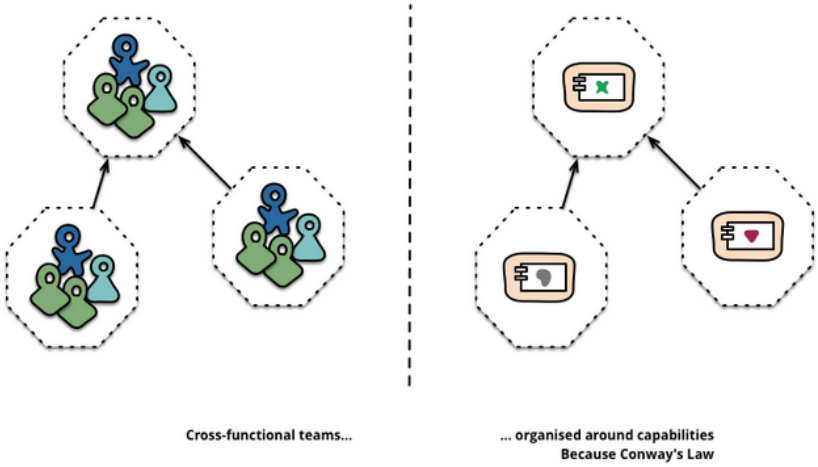

Figura 6. Equipes multifuncionais.

Fonte: (Fowler, 2014)

A organização que acabamos de citar também pode ser aplicada em sistemas monolíticos. Não há nada que nos impeça de modularizar nossa aplicação por capacidades de negócio, inclusive pode-se ver no mercado alguns sistemas que o fazem com sucesso e esses sitsemas acabam por se beneficiar desse tipo de modularização. A desvantagem de aplicar essa organização em arquiteturas monolíticas é que um grande sistema geralmente tem muitos contextos negociais diferentes, e por não haver divisão física entre esses contextos, exige muita disciplina para manter esses contextos isolados. Caso o isolamento entre estes contextos se perca e eles se, fica muito mais difícil para um membro de uma equipe trabalhar nesta base de código. Micro serviços exigem que as fronteiras sejam explícitas e o risco disto acontecer é bem menor.

4.6 INTEGRAÇÃO

4.6.1 Tecnologia

Com os serviços modelados de forma correta, devemos então arranjar uma forma deles comunicarem entre si. Nos temos visto nos últimos anos diferentes opções de tecnologias para integração de serviços. Podemos citar desde tecnologias mais antigas como CORBA, passando pelo RMI, até abordagens mais recentes como SOAP e o REST. O importante é poder escolher uma tecnologia que permita que mudanças feitas em um serviço gerem o mínimo de impacto possível nos clientes que o consomem.

Um aspecto que deve ser observado na tecnologia de integração adotada é se ela permite a criação de APIs tecnologicamente agnósticas. Isso significa escolher uma tecnologia de integração que não imponha a pilha tecnológica a ser adotada. Um exemplo de uma tecnologia que não é tecnologicamente agnóstica por exemplo é o

RMI, pois o serviços a ser consumido deve ser escrito em Java, assim como seu cliente e não é possível reutilizar este serviços.

Uma premissa muito importante é a facilidade que consumidores tem de utilizar um serviço disponível. De nada adiantar ter um serviço se por acado o trabalho para consumir este serviço for grande demais. O ideal é que os consumidores tenham liberdade para usar a tecnologia que bem entender sem que isso seja uma limitação para consumir o serviço disponível. Um serviço escrito em Java por exemplo deve oferecer a mesma facilidade integração para consumidores que rodam em Ruby, Python ou qualquer outra linguagem de programação. Uma forma de garantir este tipo de facilidade é a disponibilização de clientes. Algumas tecnologias como o SOAP por exemplo oferecem formas de se criar clientes de forma automatizada a partir e um serviço. Esse tipo de facilidade pode acelerar bastante o desenvolvimento em um primeiro momento, no entanto, devemos ter em mente que utilização de clientes causa aumento do acoplamento com o serviço.

Para manter o acoplamento baixo entre o serviço e seus consumidores, é muito importante expor em suas APIs apenas informações estritamente necessárias e que não estão ligadas com o funcionamento interno do serviço. Devemos esconder detalhes de implementação que são internas ao serviço para que mudanças na lógica interna do serviço não impacte consumidores. Devemos escolher uma tecnologia que nos ajude neste propósito.

4.6.2 Banco de dados como integrador

Vamos imaginar um cenário bastante comum nas corporações em que uma determinada empresa possui um conjunto de aplicações que tem como seu meio de persistência um banco de dados relacional. Todas elas utilizam a mesma instância do banco de dados. Vamos supor que a aplicação A seja responsável por manter os dados dos clientes da empresa. No caso da aplicação B precisar consumir os dados de clientes, a forma mais natural de fazê-lo seria por alguma forma de integração com a aplicação A.

Ao pensar nas formas de integração entre as duas aplicações nota-se que ambas acessam o mesmo banco de dados. Uma solução muito simples seria fazer a aplicação B ler os dados mantidos pela aplicação A diretamente do banco de dados. Esse tipo de integração, usando o banco de dados para fazer duas aplicações compartilharem informações, é uma solução muito simples e por isso se tornou muito popular. No entanto, essa abordagem possui uma série de problemas.

O primeiro problema que podemos notar é que se a aplicação A é responsável por manter os dados de clientes e nós permitimos que a tabela de clientes seja acessível por outras aplicações diretamente o que estamos fazendo é permitir que outras aplicações se atrelem a detalhes de implementação da aplicação A. Ao fazer isso as aplicações que acessam essa tabela ficam extremamente acopladas ao banco de dados. Se por acaso nós atualizarmos o *schema* da tabela clientes para comportar alguma manutenção ou evolução existe uma grande possibilidade de que essa mudança ira quebrar os consumidores da tabela. Para garantir que isso não aconteça é necessário executar uma grande bateria de testes de regressão.

Outro problema da estratégia adotada é que ao acessar diretamente o banco de dados, a aplicação se torna muito acoplada a este mecanismo de persistência. Suponha a empresa decide migrar todos os dados de clientes do banco de dados relacional para um banco de dados textual. Ao fazer isso, nós teremos que atualizar todas as aplicações que acessam

a tabela de clientes para acessar a banco de dados textual. As vezes o esforço de manutenção nas aplicações é tão grande que simplesmente a mudança não é realizada. Em oposição a isso, o que pode ser feito é projetar um serviço centralizado de manutenção de clientes que esconde os detalhes de implementação, como por exemplo a forma como a persistência é feita. Dessa forma nós podemos mudar a implementação deste serviço sem que os consumidores sejam afetados.

Por último, imagine que exista uma lógica de negócio associada a ação de salvar um cliente. Se as aplicações acessam diretamente a tabela que armazena os dados, onde pode ficar essa lógica negocial? Desta forma, o que acaba acontecendo é que lógica fica replicada em diferentes aplicações. Vamos lembrar que para obtermos alta coesão nós temos que ter tudo que trata sobre uma determinada funcionalidade em um local centralizado. Se nós temos uma responsabilidade que está espalhada por diversas aplicações diferentes, por definição nós não temos como ter coesão.

Tendo em vista os problemas acima, sabemos que não é possível ter um sistema que tenha alta coesão e baixo acoplamento utilizando integração através do banco de dados. Mesmo que essa abordagem seja muito simples vantajosa em um primeiro momento, ela começa a mostrar as suas desvantagens e se tornar muito cara em um médio prazo.

4.6.3 Comunicação síncrona e assíncrona

Outro aspecto que essencial para definir a forma como os serviços colaboram é a forma de comunicação. Se ela ocorrerá de forma síncrona ou assíncrona. Sam Newman diferencia o comportamento da comunicação síncrona e assíncrona da seguinte forma:

> Com comunicação, uma chamada a um servidor remoto é bloqueada até que a operação complete. Com comunicação assíncrona, aquele que chamou não espera a operação concluir antes de retornar, e pode até nem ligar se operação completa ou não. *(NEWMAN, 2015)*

Enquanto a comunicação pode ser mais simples, já que sabemos se uma determinada operação terminou com sucesso ou não, a comunicação assíncrona pode ser vantajosa em casos de tarefas de longa duração, onde o cliente não tem como esperar a resposta da requisição, ou quando é necessário ter latência baixa. Utilizar comunicação assíncrona por exemplo quando a rede está muito lenta pode melhorar muito a experiência da aplicação, já que com a comunicação síncrona existe a tendência de deixar tudo mais lento. É preciso observar no entanto que comunicação assíncrona pode ser mais complicada de se implementar.

Estas duas formas de comunicação nós possibilita duas formas de colaboração entre serviços: requisição/resposta e orientado a eventos. A modelo de requisição/resposta se alinha mais com a requisição síncrona, enquanto a o modelo orientado a eventos está mais alinhado com o a comunicação assíncrona. Com colaboração orientada a eventos nós mudamos a lógica convencional de se pedir por algo e aguardar a resposta para uma outra abordagem em que uma ação é feita e outros serviços são notificados do evento que acabou de ocorrer.

A vantagem da colaboração assíncrona é que a lógica de negócio não fica centralizada em um serviço que sabe quais serviços chamar. Ao invés disso, essa lógica é distribuída para os serviços que estão colaborando juntos. Isso garante um baixíssimo acoplamento já que o serviço que executa uma ação e portanto emite um evento, não tem conhecimento de quem será notificado por este evento. Dessa forma, podemos adicionar assinantes de determinados eventos sem que o cliente saiba disso.

Embora comunicação assíncrona orientada a eventos em geral possibilite um menor acoplamento entre os serviços que estão colaborando é preciso ter em mente qual a melhor abordagem para um problema complexo já que para cada situação pode ser melhor aplicar um modelo em detrimento de outro.

4.6.4 Orquestração e coreografia

Ao trabalhar com micro serviços é muito comum que para realizar uma lógica de negócio complexa seja necessário utilizar um conjunto de micro serviços que ao colaborar atinjam um objetivo maior do ponto de vista negocial. Existem basicamente duas formas de implementar este tipo de colaboração. Por orquestração ou coreografia.

Com orquestração, nós dependemos de um comando central para controlar e guiar o processo, como um maestro em uma orquestra. Esta analogia sugere que cada serviço é como um músico de uma orquestra, que espera as instruções de um maestro para poder colaborar com os outros músicos. A desvantagem da orquestração é que ao centralizar o comando da colaboração existe o risco de lógica negocial começar a surgir ali, o que pode levar os serviços individuais a ficarem anêmicos, muitas vezes realizando apenas operações de CRUD, enquanto a lógica negocial passa a ser executada nos serviços orquestradores que se tornam serviços extremamente inchados.

Com a abordagem da coreografia nós podemos fazer o primeiro serviço da cadeira de colaboração emitir um evento dizendo que terminou seu processamento, assim o serviço que assina este tipo de evento seria notificado de forma assíncrona e o outro passo do processamento seria iniciado e assim por diante. Este tipo de abordagem é muito mais desacoplado já que serviços que seguem a cadeia de colaboração só devem assinar o evento apropriado e não há um serviço central que controla essa inteligência. Um problema desta forma de colaboração, no entanto, a forma como esses serviços colaboram fica implícita. Isso implica em mais trabalho para monitorar e saber como está o andamento de uma colaboração assíncrona.

Assim como as questões associadas a comunicação síncrona e assíncrona, nós devemos também pesar nos prós e contras de colaboração por orquestração e por coreografia. A orquestração tente a ser mais simples no geral enquanto a coreografia tende a possibilidade menos acoplamento entre os serviços, com o preço de ser mais complexa.

4.7 REST

Existem diversas tecnologias que podem ser empregadas para integrar aplicações. Dentre elas podemos citar algumas como RMI da plataforma JAVA, o SOAP que roda encima do HTTP e o REST. REST assim como SOAP também utiliza o protocolo HTTP, porém pela sua facilidade de uso e leveza, está sendo amplamente adotado para implementar um arquitetura baseada em micro serviços.

> REST significa Representational State Transfer (Transferência de estado representacional). Ele se baseia em um protocolo de comunicações sem estado, cliente-servidor, cacheável, e em praticamente todos os casos o protocolo HTTP é utilizado.
>
> REST é um estilo arquitetural para projetar aplicações em rede. A ideia é que, ao invés de utilizar mecanismos complexos como CORBA, RPC ou SOAP para conectar as máquinas, simples HTTP é usado para fazer chamadas entre elas. *(ELKSTEIN, 2008)*

Tilkov (2008) cita alguns princípios que são fundamentais para as sua implementação:

- Dê a todas as coisas um identificador.
- Vincule as coisas,
- Utilize métodos padronizados,
- Recursos com múltiplas representações,
- Comunique sem estado;

4.7.1 Dê a Todas as Coisas Um Identificador

REST trabalha com recursos, que nada mais são do que coisas e essas coisas precisam ser identificadas. Essa identificação na web corresponde a URIs (Uniform Resource Identifier). A ideia é que todo recurso na web tenha uma forma única para ser acessada.

4.7.2 Vincule as coisas

Vincular as coisas significa dizer que esses recursos disponíveis na web devem estar ligados de alguma forma. No contexto de REST esse conceito está intimamente ligado com o conceito de HATEOAS, que é um acrônimo para *Hypermedia as the Engine of Application State*. A ideia é que os recursos possam estar ligados por links, assim como fazemos links de entre páginas html. Um exemplo disto seria um recurso pedido e suas ligações por exemplo com o recurso produto e cliente:

```
<pedido self="http://exemplo.com/pedido/1">
  <quantidade>23</quantidade>
  <produto ref="http://exemplo.com/produtos/2"></produto>
  <cliente ref="http://example.com/clientes/3"></cliente>
</pedido>
```

No exemplo acima podemos ver uma representação de um pedido e partir dele podemos ver a forma de acessar o produto e o cliente o qual o pedido referência. Essa abordagem oferece a vantagem de o cliente poder mudar de estado, mudando de um recurso para outro, a partir dos links que foram oferecidos. Estes links são muito úteis para tonar uma aplicação mais dinâmica.

4.7.3 Utilize Métodos Padronizados

Utilizar os métodos padrão significa dizer que todos os recursos devem possuir uma mesma interface. O HTTP possui verbos que são mais conhecidos como métodos. São o GET, POST, PUT, DELETE, HEAD e OPTIONS. O GET e POST são os mais conhecidos e estão muito utilizados por aplicações web de qualquer tipo. A definição destes verbos é definida na especificação do HTTP. O GET é um verbo utilizado para obter uma informação, ele é idempotente, ou seja é possível realizar a mesma operação quantas vezes for necessário sem que o valor do resultado se altere. O PUT também é idempotente, e tem a semântica de atualizar um recurso já existente. Já semantica do DELETE é a de excluir um recurso, caso ele exist enquanto o POST tem a característica de criar um novo recurso que até então não existia.

Implementar o protocolo de aplicação padrão (HTTP) corretamente, isto é, utilizar os métodos padrão: GET, PUT, POST e DELETE faz com que tenhamos uma interface uniforme o que permite que qualquer componente que entenda o protocolo de aplicação HTTP interaja com seu aplicativo.

4.7.4 Recursos com múltiplas representações

Uma possibilidade que REST nos oferece é a de fornecer diferentes formatos para os recursos. Serviços web tradicionalmente trabalham com resultados em formato HTML. Raggett (2005) define HTLM como "... um tipo especial de documento texto que é usado por navegadores web para apresentar textos e gráficos". Um recurso cliente por exemplo pode trabalhar com recursos em diferentes representações como HTML, que pode ser interpretado por um navegador web e assim apresentado para o usuário, ou em formatos como XML, JSON ou YAML que são formas de se representar dados e são usualmente utilizados para transportar dados através da internet e podem ser consumidos diretamente por outras aplicações. A vantagem de se implementar serviços que suportam diferentes formatos é permitir que uma maior diversidade de clientes possam consumir nossos serviços.

4.7.5 Comunique sem estado

Serviços REST não devem manter estado. Isso quer dizer que o serviço não deve ter conhecimento de qualquer detalhe sobre requisições anteriores, mas apenas a requisição que está ocorrendo no momento. Isso implica em não utilizar recursos como a sessão por exemplo para armazenar informações transitórias. Fazer o serviço não manter estado tem benefícios como a escalabilidade e o isolamento do cliente de possíveis mudanças no servidor. Quando um serviço não mantém estado sobre os clientes, o consumo de recursos como memória tende a ser menor e além disso é possível ter escalabilidade horizontal já que podemos replicar as instâncias daquele serviço.

4.8 API GATEWAY

Em uma aplicação monolítica os seus clientes realizam requisições para a aplicação diretamente ou, no caso de uma aplicação em cluster, as requisições são enviadas para um balanceador de carga e então direcionado para uma instância da aplicação que foi replicada como ilustrado na figura 7.

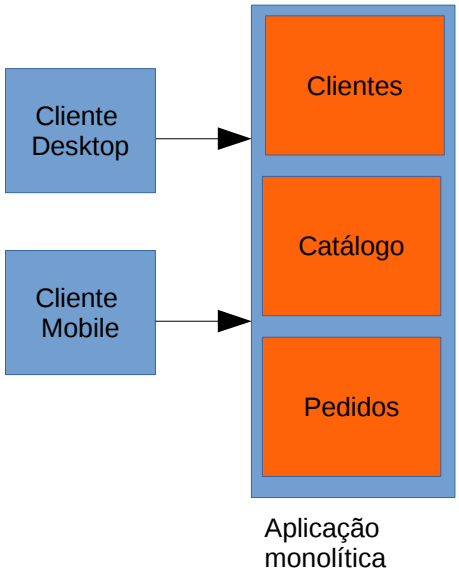

Figura 7. Comunicação em uma arquitetura monolítica.

Fonte: (Elaborada pelo autor)

Imagine o caso desta aplicação monolítica ser quebrada em micro serviços. Como ficaria a comunicação desses clientes com os serviços. A forma mais simples de realizar esta comunicação é bolando um esquema como ilustrado na figura 8.

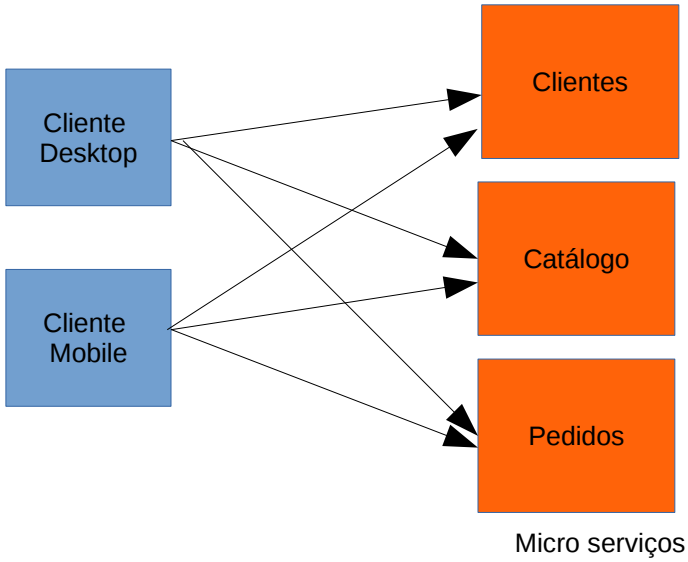

Figura 8. Comunicação em uma arquitetura baseada em micro serviços.

Fonte: (Elaborada pelo autor)

Em um primeiro momento este esquema pode parecer uma boa solução para a comunicação dos clientes com os serviços, no entanto Richardson (2014) levanta alguns problemas para este tipo de comunicação:

- A granularidade das APIs providas pelos micro serviços é diferente do que o cliente pode precisar. No exemplo acima, um cliente desktop pode consultar dados de clientes e pode precisar de mais detalhes do que um cliente mobile. Desta forma cabe o cliente ter que filtrar, rearranjá-los ou mesmo consumir dados de vários serviços.
- A performance da rede para diferentes clientes. Geralmente um cliente mobile, por exemplo, possui menos largura de banda que um cliente desktop. Deixar que o cliente mobile por exemplo tenha que fazer inúmeras requisições para o servidor pode tornar a experiência de usuário ruim, enquanto uma aplicação do lado do servidor pode realizar vários requisições para os serviços na mesma rede sem impactar a experiência do usuário.
- A quantidade de instâncias dos serviços, seus endereços como IP e porta mudam dinamicamente, o que pode impactar os clientes que os consomem.
- Eventualmente pode ser necessário quebrar um serviço em serviços menores e essas mudanças devem ser transparentes para os clientes.

Para resolver estes problemas pode-se implementar uma camada na frente dos micro serviços chamada de API Gateway proposto por Richardson (2014) da forma ilustrada na figura 9.

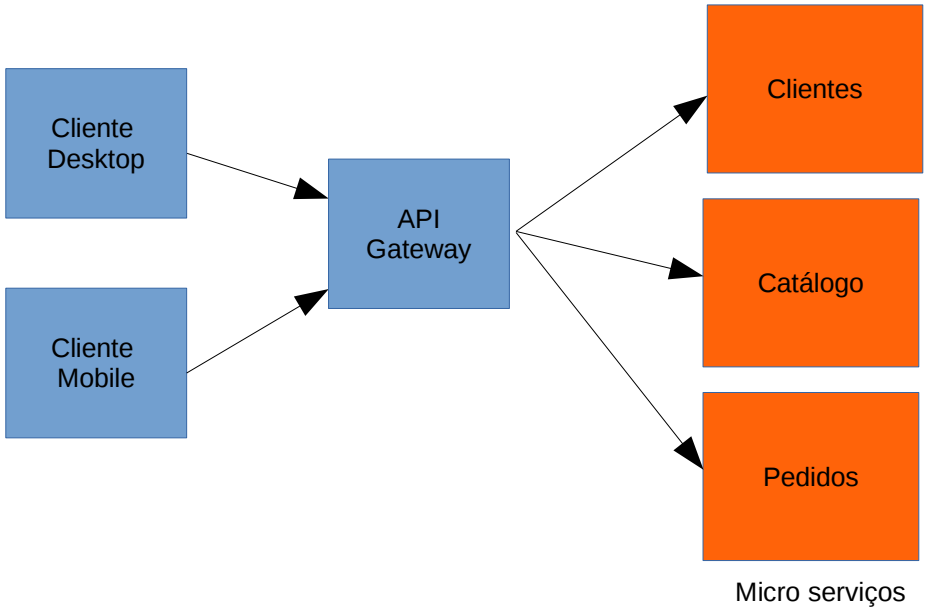

Figura 9. Comunicação com API Gateway.

Fonte: (Elaborada pelo autor)

Com o API Gateway nós podemos prover APIs específicas para diferentes clientes de tal forma que eles não precisem acessar diferentes micro serviços de granularidade baixa. Essa estratégia já ajuda na performance de rede pois reduz o numero de requisições ao servidor. Outra vantagem deste padrão é que o acesso as APIs expostas para os clientes ficam todas centralizadas nesta camada. Assim, caso hajam mudanças nos serviços que estão atrás do API Gateway as mudanças terão impacto apenas no centralizador, evitando impacto nos consumidores.

Em sistemas de todos os estilos arquiteturais existem comportamentos comuns implementados compartilhados entre diferentes serviços. Esses comportamentos comuns são chamados aspectos transversais. Exemplos desses aspectos bastante comuns são: implementação de log de sistemas, controle transacional, tratamento de erros, entre outros.

Implementar um aspecto transversal em um sistema monolítico é relativamente simples. Estes aspectos são tratados por uma camada acima da pilha do software. Em uma arquitetura baseada em micro serviços não é diferente. Quem faz o papel de desta camada é o API Gateway. O API Gateway além de resolve os problemas apresentados acima ele também pode atuar para tratar questões de segurança, balanceamento de balanceamento de carga ou transformação de dados de transporte por exemplo como citado por Peyrott (2015).

Um gateway funciona como um ponto único de entrada de uma API. Nestes casos, eles lidam com a segurança e encaminham a requisição para outros micro serviços usando outro canal seguro ou mesmo removendo a segurança se não for necessário, dentro de

uma rede interna por exemplo. Um API Gateway Restful por exemplo, pode implementar comunicação HTTP via SSL, de tal forma que a conexão entre os clientes e o gateway seja segura, e após receber a requisição ele a repassa para o micro serviço interno em uma conexão HTTP sem SSL.

Em casos onde existe uma carga muito alta sobre os serviços, o API Gateway pode atuar distribuindo as requisições entre diferentes instancias de um mesmo micro serviços de acordo com uma lógica específica. Cada serviço pode ter limites diferenciados de escalabilidade. Os Gateways distribuem a carga entre os serviços levando em conta essas limitações.

Como vimos anteriormente, uma das características de um de um serviço Restful é o fato dele trabalhar com diversos formatos. Isso faz com que clientes de diferentes tecnologias possam se comunicar com os serviços de forma mais fácil. No entanto, se o serviço não foi projetado para ter essa flexibilidade, pode ser o formato utilizado não seja conveniente para clientes implementados em uma plataforma específica. Para resolver este problema um API Gateway pode fazer as transformações necessárias na representação destes dados para que diferentes cliente possam se comunicar com os serviços por trás dele.

5 Conclusão

Desde o surgimentos dos primeiros computadores, nós temos construído sistemas com arquiteturas monolíticas. Utilizando esta abordagem arquitetural, os sistemas concebidos foram capazes de resolver uma gama enorme de problemas, mas ao mesmo tempo estes sistemas foram se tornando maiores e mais complexos o que acabou por dificultar muito a sua manutenção, a realização de implantações mais frequentes, a evolução das tecnologia envolvidas e a escalabilidade destes.

A arquitetura baseada em micro serviços, uma abordagem de sistemas distribuídos, surgiu com a proposta de resolver estes problemas através da colaboração de serviços pequenos e autônomos. Um serviço pequeno deve ser coeso de forma que resolvam apenas um problema respeitando o princípio da responsabilidade única e para ser autônomo um micro serviço deve idealmente ser auto contido, com baixo acoplamento e tecnologicamente agnóstico.

Por serem pequenos e autônomos micro serviços tem a vantagem de possibilitarem uma maior heterogeneidade, composabilidade e maior facilidade de substituição. Sistemas baseados nesta arquitetura também possuem maior resiliência, escalabilidade e são mais fácil de se implantar e ainda por cima levam a um maior alinhamento organizacional por permitir uma organização de equipes mais adequada. Esses benefícios no entanto tem um custo. A implementação de uma arquitetura baseada em micro serviços adiciona considerável complexidade ao desenvolvimento, pois se tratar de sistemas distribuídos, e também complexidade operacional.

Para ter sucesso nesta abordagem é importante modelar bons serviços observando os limites delimitados (bounded contexts) envolvidos afim de obter alta coesão e baixo acoplamento, além disso é importante modelar os serviços determinados por capacidades de negócio ao invés de questões tecnológicas.

Um critério de extrema importância para o sucesso da arquitetura de micro serviços é a escolha da tecnologia de integração. Uma tecnologia de integração ideal deve permitir que clientes das mais diversas tecnologias possam se integrar a ela e também que o

mínimo de impacto será causado aos clientes em caso de atualização dos serviços. Um exemplo de tecnologia de ampla adoção para a implementação de micro serviços são os web services *RESTful*, que rodam sobre o HTTP e idealmente devem implementar uma série de princípios para o garantir o seu bom funcionamento.

Por fim vimos o padrão API Gateway, que visa que resolver problemas de diferentes granularidades de APIs para clientes distintos, melhorar a performance quando existe muita comunicação entre micro serviços e abstrair a localização dos micro serviços centralizando a API. Além de resolver os problemas citados, vimos que o API Gateway pode oferecer serviços de segurança, balanceamento de balanceamento de carga ou transformação de dados de transporte.

Este trabalho tem como resultado uma introdução ao estudo da arquitetura baseada em micro serviços, apresentando seus benefícios, desvantagens, boas práticas de como se modelar serviços e questões tecnológicas envolvidas neste estilo arquitetural. A partir daqui é possível iniciar trabalhos práticos utilizando esta abordagem, mas é importante frisar que por ser um campo de estudo amplo e relativamente recente, o envolvimento com o assunto demandará algum aprofundamento a aqueles que quiserem se aventurar nesta área.

Pode-se observar vários casos de sucesso entre empresas que adotaram arquitetura de micro serviços em cenários completamente desafiadores, mas este estilo arquitetural é muito novo. Muitas consequências de decisões arquiteturais só ficam evidentes depois de alguns anos que elas foram, tomadas por isso o conhecimento a respeito desta arquitetura deve crescer muito nos próximos anos, o que nos leva a necessidade de realizar estudos sobre tópicos específicos que não foram realizados neste trabalho.

6 Referências

BUTLER, Brandon. PaaS Primer: What is platform as a service and why does it matter? Fev. 2013. Disponível em : <https://yobriefca.se/blog/2013/04/29/micro-service-architecture/>. Acesso em: 15 fevereiro 2016.

ELKSTEINS, M. Learn REST: A Tutorial. Fev. 2008. Disponível em: <http://rest.elkstein.org/2008/02/what-is-rest.html/>. Acesso em: 03 fevereiro 2016.

EVANS, Eric. Domain Driven Design, 2004.

FOWLER, Martin. Is Design Dead? Jul. 2000. Disponível em: <http://martinfowler.com/articles/designDead.html>. Acesso em: 03 fevereiro 2016.

FOWLER, Martin. Bounded Context. Jan. 2014. Disponível em: <http://martinfowler.com/bliki/BoundedContext.html/>. Acesso em: 03 fevereiro 2016.

FOWLER, Microservices: a definition of this new architectural term. Mar. 2014. Disponível em: <http://martinfowler.com/articles/microservices.html>. Acesso em: 03 fevereiro 2016.

GARLAN, David; SHAW, Mary. An Introduction to Software Architecture,1994.

HUGHES, James. Micro Service Architecture. Abr. 2013 Em: <https://yobriefca.se/blog/2013/04/29/micro-service-architecture/>. Acesso em: 03 fevereiro 2016.

JENKOV, Jakob. Client-Server Architecture. Out. 2014. Disponível em: <http://tutorials.jenkov.com/software-architecture/client-server-architecture.html>. Acesso em: 05 fevereiro 2016.

MARTIN, Robert C. Agile Software Development, Principles, Patterns, and Practices. 2003. p. 110

NEWMAN, Sam. Building Microservices, 2015.

PEYROTT, Sebastián. An Introduction to Microservices, Part 2: The API Gateway. Set. 2015. Disponível em: <https://auth0.com/blog/2015/09/13/an-introduction-to-microservices-part-2-API-gateway/>. Acesso em: 05 fevereiro 2016.

RAGGETT, Dave. Getting started with HTML. Maio 2005. Disponível em: <http://www.w3.org/MarkUp/Guide/>. Acesso em: 05 fevereiro 2016.

RICHARDSON, Chris. Pattern: Monolithic Architecture. 2014. Disponível em: <http://microservices.io/patterns/monolithic.html>. Acesso em: 03 fevereiro 2016.

RICHARDSON, Chris. Pattern: API Gateway. 2014. Disponível em: <http://microservices.io/patterns/apigateway.html>. Acesso em: 05 fevereiro 2016.

RICHARDSON, Chris. Microservices: Decomposição de Aplicações para Implantação e Escalabilidade. Out. 2014 Disponível em: <http://www.infoq.com/br/articles/microservices-intro>. Acesso em: 03 fevereiro 2016.

SILVEIRA, Paulo et al. Introdução à Arquitetura e Design de Software: uma visão sobre a plataforma Java. 2012 p. 80.

STEPHENS, Rod. Beginning Software Engineering, 2015. p 94-96.

TILKOV, Stefan. Uma rápida Introdução ao REST. Out. 2008. Disponível em: <http://www.infoq.com/br/articles/rest-introduction/>. Acesso em: 03 fevereiro 2016.

WIKIPEDIA, Cliente-servidor. Abr. 2010. Disponível em: <https://pt.wikipedia.org/wiki/Cliente-servidor>. Acesso em: 05 fevereiro 2016.