**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Luís Rafael Barbosa Correia

# Construção de uma plataforma de e-commerce: uma abordagem baseada numa arquitetura de microsserviços

Dezembro 2021

**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Luís Rafael Barbosa Correia

# Construção de uma plataforma de e-commerce: uma abordagem baseada numa arquitetura de microsserviços

Dissertação de Mestrado
Mestrado Integrado em Engenharia Informática

Dissertação supervisionada por
**António Manuel Nestor Ribeiro**

Dezembro 2021

# 2

# ESTADO DA ARTE

No presente capítulo será apresentado um estudo das arquiteturas monolítica e de microsserviços. Irá ser feita uma comparação destas duas arquiteturas, as vantagens e desvantagens de cada uma e quando se deve escolher uma ao invés da outra.

## 2.1 ARQUITETURA MONOLÍTICA

> "Any organization that designs a system... will inevitably produce a design whose structure is a copy of the organization's communication structure."
>
> (Conway's law)

A arquitetura monolítica é uma arquitetura onde um produto de *software* é concebido para funcionar como uma unidade só, estando num único processo. Os componentes neste tipo de arquitetura encontram-se conectados e dependentes uns dos outros, resultando num código mais acoplado, o que leva a que toda a aplicação tenha que ser instalada juntamente [Newman (2019)].

Uma aplicação monolítica é constituída maioritariamente por três camadas, a camada de apresentação, camada de negócio e camada de dados. Na Figura 1 temos um diagrama representativo de uma arquitetura monolítica, onde é possível observar a presença destas três camadas. Toda a funcionalidade, código, componentes e lógica formam uma única entidade [Javed (2019)].

Esta arquitetura é bastante usada porque é universal e bem conhecida. Outra razão é devido à forma como as equipas a desenvolver uma aplicação estão organizadas [Newman (2019)].

Nesta arquitetura as equipas estão normalmente organizadas conforme as competências de cada programador, sejam elas administração de base de dados, programação *backend* ou *frontend*. Desta forma, o desenvolvimento de uma aplicação, por parte de equipas que se organizam desta forma será organizada de acordo com a organização das equipas [Conway (1968)].

Normalmente numa aplicação monolítica existe apenas uma base de dados a servir toda a aplicação [Javed (2019)], sendo que depois por motivos de desempenho e de tolerância a faltas, se possa replicar a base de dados, para prevenir a perda de dados ou para acelerar os acessos a esta. As aplicações que seguem esta arquitetura possuem um código bastante extenso, muito acoplado e por vezes pouco modular.

É possível ter múltiplas instâncias de uma mesma aplicação monolítica por razões de robustez e de escalonamento, mas a aplicação continua basicamente a consistir numa única unidade [Newman (2019)].

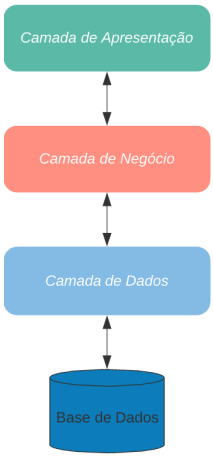



Figura 1: Arquitetura monolítica.

### 2.1.1 *Tipos de monolítico*

*Monolítico num único processo*

O exemplo mais comum de um monolítico é um sistema onde todo o código está instalado num único processo. É possível ter várias instâncias deste monolítico por questões de robustez e resiliência, mas fundamentalmente todo o código se encontra num único processo [Newman (2019)].

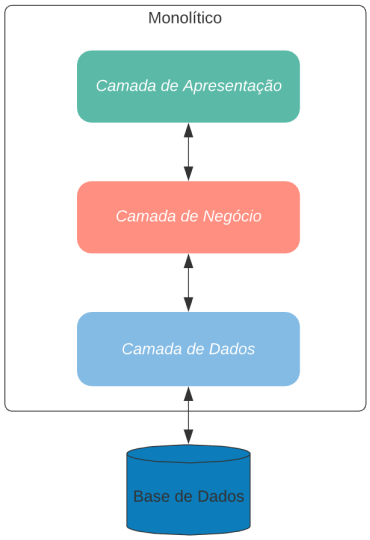



Figura 2: Monolítico num único processo.

*Monolítico Modular*

O monolítico modular é uma variação do monolítico num único processo, exceto que o sistema se encontra separado por módulos, que podem ser desenvolvidos independentemente, mas que precisam de estar juntos para a aplicação poder ser instalada [Newman (2019)].

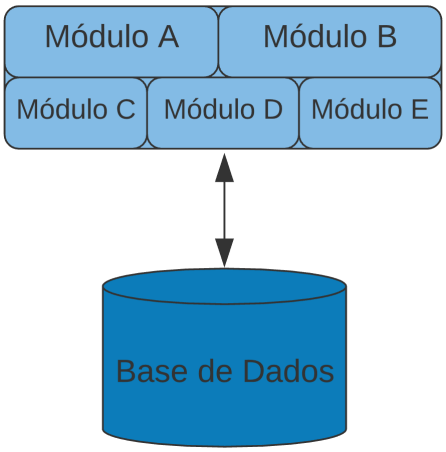



Figura 3: Monolítico Modular.

Esta forma de organização do código permite o trabalho em paralelo por várias equipas nos diferentes módulos, não necessitando de haver preocupação com os problemas de sistemas distribuídos adquiridos da adoção de microsserviços [Newman (2019)].

Um dos problemas deste tipo de monolítico é a base de dados tender a não ter o mesmo nível de decomposição que o código, o que pode dificultar um eventual processo de separação do monolítico [Newman (2019)].

*Monolítico Distribuído*

Um monolítico distribuído é um sistema composto por vários serviços, mas que por alguma razão necessita de ser instalado conjuntamente, isto é, os serviços não podem ser instalados independentemente [Newman (2019)].

### 2.1.2 *Vantagens*

A arquitetura monolítica possui vantagens como as apresentadas:

- **Fácil desenvolvimento**

  O desenvolvimento de uma aplicação monolítica é mais fácil, no sentido em que as ligações entre componentes não tem de ser estabelecidas sobre a rede, não havendo necessidade de preocupações com a perda de mensagens, nem com problemas adotados dos sistemas distribuídos [Richardson (2018)].

  Além disso, numa fase inicial do desenvolvimento de *software* não existe a necessidade de ter que pensar como deve ser a separação dos microsserviços [Newman (2019)].

- **Simples de resolver problemas**

  Um problema que cause a falha de uma funcionalidade ou da aplicação é mais facilmente encontrado e resolvido visto toda a funcionalidade se encontrar num único lugar.

- **Simples de testar**

  Um produto de *software* deve contemplar testes às suas funcionalidades, como testes unitários ou de integração, para que se determine a qualidade do código e que as funcionalidades desenvolvidas comportam-se como efetivamente pretendido.

  Num único artefacto é muito mais fácil de testar a aplicação na totalidade e a integração com cada componente [Richardson (2018)].

- **Instalação mais fácil**

  Uma aplicação monolítica é um único artefacto sendo só necessário copiar o artefacto para um servidor e pôr a correr [Richardson (2018)].

  Se a aplicação fosse composta por vários componentes que tivessem que ser instalados separadamente era necessário a configuração destes para a aplicação funcionar na totalidade.

  Com uma aplicação monolítica não é necessário ter a preocupação de configurar todos os componentes da aplicação, visto que eles já se encontram conectados. Evita-se também ter problemas normalmente associados aos sistemas distribuídos.

- **Simples de escalar**

  Um único artefacto é fácil de escalar horizontalmente, tendo várias máquinas a correr a mesma aplicação atrás de um *load balancer* [Richardson (2018)], tal como apresentado na Figura 4.

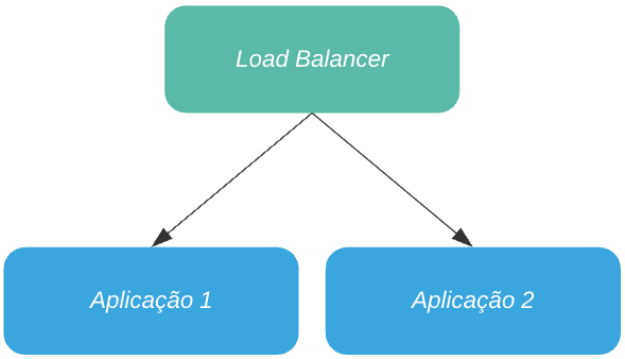



Figura 4: Escalonamento de uma Arquitetura Monolítica.

- **Desempenho**

  Todos os componentes encontram-se no mesmo processo, sendo a comunicação entre estes efetuada através de abstrações da tecnologia utilizada e recorrendo a memória partilhada. Isto é mais rápido e eficiente do que fazer pedidos a outros componentes que não residam no mesmo processo. Neste caso

os pedidos iriam ser feitos pela rede que pode ter inconsistências, tal como atrasos, perda de mensagens, entre outros, que leva a um aumento do tempo de resposta [Newman (2019)].

### 2.1.3 *Desvantagens*

Apesar de vantagens, uma aplicação monolítica possui também desvantagens, como as apresentadas:

- Uma aplicação monolítica acaba por ter um código bastante extenso e de difícil compreensão, o que leva a uma maior dificuldade e demora em eventuais alterações ou adições às funcionalidades da aplicação. A introdução de um novo elemento na equipa de desenvolvimento também será feita com algum custo. Este terá mais dificuldade em compreender o código, dado que o código estará muito acoplado, complexo e extenso [Richardson (2018)].
- Uma modificação a um componente da aplicação implica que toda a aplicação tenha que compilar de novo e voltar a ser instalada [Richardson (2018)].
- Não está adaptada para várias equipas trabalharem em simultâneo. Várias pessoas a trabalhar no mesmo produto de *software* pode levar a que umas se atrapalhem às outras, acabando por modificar partes que outros estavam a modificar, levando a conflitos no código [Newman (2019)].

  Leva a um processo de adição de novas funcionalidades para o utilizador mais demorado, podendo haver atrasos em várias fases da entrega do produto.
- A falha de num componente da aplicação leva à falha de toda a aplicação.

  A falha de um componente da aplicação tem impacto na disponibilidade da aplicação, pois mesmo que a aplicação seja escalada horizontalmente se existir um problema num componente este estará em todas as outras máquinas [Javed (2019)].
- Complexidade na adoção de novas tecnologias.

  A arquitetura monolítica dificulta a adoção de novas linguagens ou *frameworks*.

  A adoção de uma nova tecnologia implica a reescrita de grande parte do código da aplicação, o que pode ser problemático em termos de tempo, custo e eventuais problemas que possam surgir [Newman (2019)].
- Escalabilidade

  Quando se quer escalar a aplicação, tem que se escalar na totalidade. Não é possível escalar apenas componentes da aplicação, que necessitem de uma maior eficácia e desempenho [Richardson (2018)].

### 2.1.4 *Decisão arquitetural*

Desenvolver segundo uma arquitetura monolítica permite uma maior facilidade no início do desenvolvimento. Não existe a necessidade de pensar na divisão dos microsserviços, o que leva a uma maior rapidez no desenvolvimento do produto de *software* ou do MVP (minimum viable product) [Fowler (2015)].

A decisão de se desenvolver uma aplicação seguindo uma arquitetura monolítica deve ser tida em conta quando:

- A equipa de desenvolvimento é pequena.

  Uma equipa pequena ao se focar no desenvolvimento de uma aplicação que siga uma arquitetura de microsserviços acabaria por perder bastante tempo e recursos a lidar com as complexidades desta arquitetura. Uma aplicação monolítica acaba por ser a solução mais rápida e eficiente de se desenvolver a aplicação.

- A aplicação a desenvolver é simples.

  Uma arquitetura monolítica é simples de entender e desenvolver uma aplicação com poucas funcionalidades [Fowler (2015)].

- Falta de conhecimento com a arquitetura de microsserviços.

  Começar a desenvolver uma aplicação seguindo uma arquitetura de microsserviços sem ter conhecimento na área torna-se um processo dispendioso e frustrante. É necessário gerir os vários serviços, as ligações entre estes, conflitos e problemas que surjam, o que dificulta a resolução destes, dado que os microsserviços se encontram em processos diferentes [Fowler (2015)].

- A aplicação deve ser instalada rapidamente.

  Uma aplicação monolítica é mais fácil de ser instalada, visto que toda a aplicação se encontra num único artefacto, sendo apenas necessário colocar este artefacto num servidor e pôr a correr.

## 2.2 ARQUITETURA DE MICROSSERVIÇOS

A arquitetura de microsserviços é uma abordagem ao desenvolvimento de *software* definida como um conjunto de microsserviços autónomos que trabalham conjuntamente e comunicam entre si para o funcionamento de um produto de *software* [Fowler and Lewis (2014)].

Ao contrário do que se encontra na Figura 1 apresentada na secção 2.1, na Figura 5 podemos observar que a camada de negócio foi separada em vários microsserviços, independentes, que comunicam entre si e possuem a sua própria base de dados.

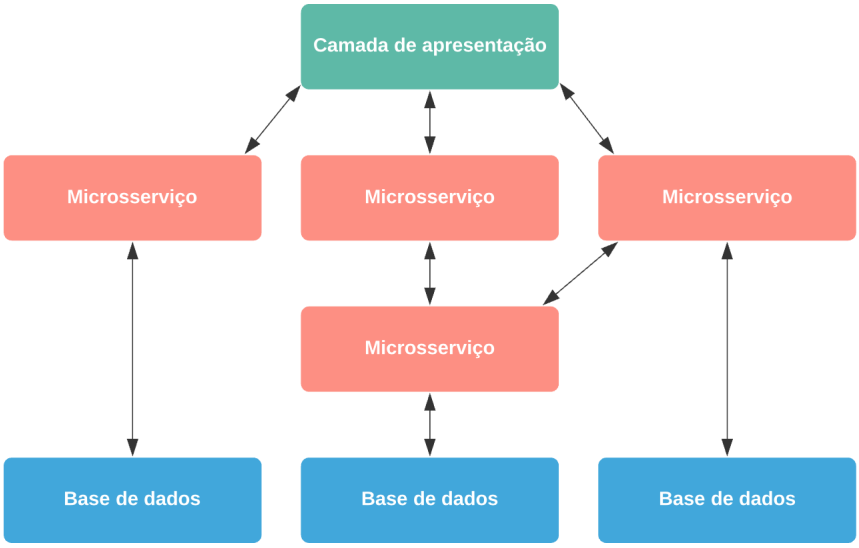



Figura 5: Arquitetura de Microsserviços.

### 2.2.1 *Definição de um microsserviço*

Numa arquitetura de microsserviços um microsserviço é uma unidade de *software* que é independente de todos os outros, capaz de ser substituído e atualizado sem que os outros microsserviços tenham que ter qualquer tipo de ação perante esta mudança [Newman (2019)].

Cada microsserviço corre num processo exclusivamente seu e de forma autónoma, encapsulando funcionalidade e comunicando com outros através da rede [Newman (2021)]. Como é possível observar pela Figura 5 cada microsserviço tem a sua própria base de dados, sendo este responsável por gerir os seus dados. Caso um microsserviço necessite de informação que outro possua, este realiza um pedido a esse microsserviço para que este lhe devolva a informação. Do exterior, um microsserviço é tratado como uma caixa negra e nenhum acede diretamente à base de dados de outro. No entanto, pode existir um microsserviço que não necessite de base de dados [Newman (2021)].

Um microsserviço pode representar o inventário, outro a gestão das encomendas e outro o envio das encomendas, mas juntos trabalham conjuntamente para constituir uma aplicação.

Ter microsserviços que possuem fronteiras claras e bem definidas que não são modificadas quando existe uma mudança numa implementação interna, resulta num sistema que possui pouco acoplamento e bastante coesão [Newman (2021)], algo a mencionar mais à frente na secção 2.2.3.

#### *Agregados*

Uma das dificuldades encontrada quando se desenvolve uma aplicação seguindo uma arquitetura de microsserviços é definir o que deve ser um microsserviço, com que funcionalidades este deve ficar encarregue e quais as suas fronteiras.

Um conceito apresentado por Eric Evans no seu livro *Domain-Driven-Design* [Evans (2003)] é o de *agregado*. Um agregado é um conjunto de objetos de domínio que podem ser tratados como uma unidade só. São representações de domínios reais. Um exemplo de um agregado pode ser uma encomenda e os produtos desta encomenda. Estes são objetos diferentes e separados, mas pode ser vantajoso tratá-los como um agregado só [Fowler (2013)].

Um agregado normalmente possui um ciclo de vida, o que permite que seja implementado como uma máquina de estados. O que se pretende é tratar agregados como unidades independentes, que todo o código que trata das transações de estado do agregado seja agrupado conjuntamente, bem como o próprio estado. Agregados podem ter relações com outros agregados [Newman (2019)].

Mapeando o conceito de agregados para microsserviços, um único microsserviço pode lidar com o ciclo de vida e armazenamento de dados de um ou mais agregados. Caso outro microsserviço queira alterar o estado de um agregado de outro, deve efetuar um pedido a este para efetuar a transferência de estado. Esta possibilidade de alteração de estado deve ser definida pelo microsserviço que gere o agregado, evitando mudanças de estado ilegais [Newman (2019)].

*Bounded Context*

> "Cells can exist because their membranes define what is in and out and determine what can pass."
>
> (Eric Evans)

*Bounded Context* [Evans (2003)] é outro conceito introduzido no livro *Domain-Driven Design* que define fronteiras numa aplicação, indicando vários sub-domínios que possuem funções e características diferentes. A ideia passa por indicar que qualquer domínio consiste num conjunto de *Bounded Context*, sendo que dentro de cada *Bounded Context* existem modelos que podem ou não ser partilhados com o exterior [Evans (2003)].

Eric Evans faz a analogia com as células, dizendo que estas existem porque as suas membranas definem o que se encontra dentro e fora da célula e o que entra e o que sai. *Bounded Context* encontra-se como uma boa definição de o que deve ser um microsserviço. Deve estar responsável pelos seus próprios dados e funcionalidades, receber pedidos e fazer pedidos a outros microsserviços, controlando o que se encontra dentro de si próprio [Newman (2015)].

Cada microsserviço possui uma *interface*, onde expõe dados para que o exterior possa aceder. A base de dados de um microsserviço encontra-se encapsulada dentro das suas fronteiras, só podendo ser acedida por este. Cada um deve decidir que informação partilhar com outros *Bounded Context* [Newman (2015)].

Do ponto de vista da implementação de um microsserviço, um *Bounded Context* contém um ou mais agregados. Alguns agregados podem ser expostos para fora do contexto e outros serem escondidos internamente. Tal como agregados, *Bounded Context* podem ter relações para outros *Bounded Context* [Newman (2019)].

### 2.2.2 *Características de um microsserviço*

*Independentemente instalado*

Cada microsserviço deve poder ser modificado e instalado sem afetar os outros. Desta forma não existe necessidade de modificar os outros microsserviços por causa deste, o que diminui a quantidade de instalações necessárias. Para se conseguir tal, é necessário que os microsserviços sejam pouco acoplados, para se poder efetuar alterações sem ter obrigatoriamente que mudar outros [Newman (2019)].

*Modulado à volta de um domínio*

A realização de uma adição ou mudança que envolva vários processos é complicada e dispendiosa. Se for necessário a mudança a uma funcionalidade que envolva dois microsserviços, será necessária a compilação e instalação destes, acabando por se ter o dobro do trabalho que se teria se apenas fossem necessárias alterações a um microsserviço. Desenvolver uma funcionalidade que envolva a mudança em mais que um microsserviço é dispendioso, pois é necessário coordenar o trabalho entre estes, potencialmente entre equipas e gerir cuidadosamente a ordem com que devem ser instaladas as novas versões destes, o que leva a maior trabalho, do que desenvolver apenas a funcionalidade num único microsserviço [Newman (2021)].

*Possui os seus próprios dados*

Cada microsserviço deve ter a sua própria base de dados e não devem existir bases de dados partilhadas. Se um microsserviço necessita de dados de outro, não deve aceder diretamente à base de dados desse, mas sim pedir-lhe por esses dados [Newman (2019)].

Desta forma permite-se que cada microsserviço defina quais dados pretende partilhar e quais esconder do exterior. Além disso, também se reduz a acoplamento entre microsserviços.

*Ocultar informação*

Com microsserviços é possível ocultar informação, que significa ocultar o máximo de informação possível no microsserviço e expor o mínimo para o exterior. Permite que exista uma separação clara entre o que pode ser facilmente mudado e aquilo que é mais complicado e se deve ter mais cuidado. Implementações que se encontrem ocultas do exterior podem ser modificadas sem preocupação, dado que não devem afetar o exterior, pois estes não tem acesso a estas [Newman (2021)].

*Agnóstico à tecnologia*

Cada microsserviço pode ser implementado na tecnologia pretendida, podendo escolher a melhor tecnologia para cada situação [Newman (2019)].

### 2.2.3 *Acoplamento e Coesão*

> "A structure is stable if cohesion is high, and coupling is low."
>
> (Larry Constantine)

É importante quando se desenvolve microsserviços que se tenha em conta o acoplamento e coesão entre microsserviços aquando da definição das fronteiras de cada um [Newman (2019)].

Acoplamento é a medida de dependência que existe entre sistemas. Quanto mais os microsserviços se encontram acoplados, maior mudanças implicará nesses quando uma alteração tiver que ser realizada [Newman (2019)].

Uma boa definição de coesão é que o código, que precisa de ser modificado quando ocorre uma alteração, deve estar agrupado [Newman (2019)].

Com microsserviços o que se pretende é que quando seja necessário efetuar uma alteração não se tenha que modificar vários microsserviços, mas sim apenas num. Desta forma o que se pretende é que se tenha o código agrupado de certa forma a que quando for necessário efetuar alterações, estas sejam realizadas no menor número possível de microsserviços, alcançado através de microsserviços muito coesos e pouco acoplados [Newman (2019)].

A existência de microsserviços que sejam pouco coesos e muito acoplados leva a que seja muito dispendioso lidar com mudanças, visto que é necessário modificar microsserviços que residem em processos separados, levando à necessidade de ter que instalar estes microsserviços que são independentemente instaláveis [Newman (2019)].

O que se pretende com uma arquitetura de microsserviços é que se tenham microsserviços bastante coesos e estáveis, para se alcançar o conceito de cada microsserviço poder ser independentemente instalável. Para isto, cada um deve providenciar uma *interface* estável, que não seja constantemente mudada [Newman (2019)].

### 2.2.4 *Comunicação entre microsserviços*

James Lewis e Martin Fowler introduziram um conceito que explica como devem ser as comunicações entre microsserviços. Este conceito é originalmente conhecido por *"smart endpoints and dumb pipes"* [Fowler and Lewis (2014)], em tradução livre: "endpoints inteligentes e comunicação simples", que significa que não se deve colocar muita complexidade no transporte das mensagens, tal como roteamento, transformação de mensagens, entre outros. A comunicação entre microsserviços deve ser o mais simples possível, enquanto toda a lógica de tratamento das mensagens se deve encontrar dentro de cada um.

Os microsserviços podem comunicar sobre vários protocolos de comunicação, de entre os quais se destacam:

### *REST*

REST (REpresentational State Transfer) [Fielding and Taylor (2000)] define-se como um estilo arquitetural que consiste num conjunto de normas aplicadas a componentes e dados de um sistema, disponibilizando funcionalidades por parte deste.

Em REST todos os componentes apresentam a mesma *interface*, com um conjunto de operações fixas e universais, tais como GET, PUT, POST, DELETE, entre outras, que possibilitam a interação com outro sistema, possibilitando que os sistemas possam ser implementados independentemente uns dos outros, sem ser necessário estar a estabelecer protocolos entre eles [Doyle et al. (2021)].

REST possui a característica de não necessitar de estado indicando que cada sistema não necessita de saber nada sobre o outro que lhe faz o pedido, nem vice-versa [Fielding and Taylor (2000)].

Cada sistema a utilizar REST é constituído por um conjunto de recursos. Estes recursos representam qualquer objeto, documento ou informação que cada sistema armazena e que pode ser acedido por outros. Cada recurso tem um identificador único, podendo estes ser unicamente acedidos através da *interface* REST disponibilizada por cada sistema. Normalmente estes recursos nunca são disponibilizados, mas sim representações destes que podem ser devolvidas em vários formatos. A mais popular é JSON (Javascript Object Notation) devido à sua fácil compreensão e por ser agnóstico em termos de linguagem.

Uma das desvantagens do REST prende-se pela sua comunicação ser síncrona [Doyle et al. (2021)], o que faz com que o sistema que invoca o *endpoint* tenha que ficar à espera, bloqueado até receber a resposta. Pode ser problemático caso no sistema invocado tenha ocorrido uma falha ou se a comunicação através da rede estiver mais lenta.

### *RPC*

RPC (Remote Procedure Call) é uma técnica que permite realizar uma chamada local a uma função que na verdade é executada num outro sistema remoto. Em vez de se aceder remotamente a um sistema, faz-se a chamada localmente, sendo que esta esconde os detalhes da comunicação remota [Matturro (2020)].

Durante uma chamada remota acontecem os seguintes passos:

- O sistema invoca uma função local que é tratada por um *stub*, uma *interface* que permita a comunicação com o sistema remoto, que coexiste localmente com o sistema.
- Este *stub* transforma os parâmetros da função para uma mensagem compreendida pelo sistema remoto.
- Esta mensagem é passada através da rede para o sistema remoto.
- No sistema remoto esta mensagem é entregue ao *skeleton* do sistema que está encarregue de extrair a informação da mensagem e chamar no sistema remoto a função pretendida com os parâmetros que vieram na mensagem.
- Completada a execução é retornado o resultado ao *skeleton* que o transforma numa mensagem e envia através da rede para o *stub* do sistema que lhe fez o pedido.

- O *stub* do sistema recebe o resultado e extrai o conteúdo da mensagem, devolvendo o resultado ao sistema.

Este processo está apresentado na Figura 6.

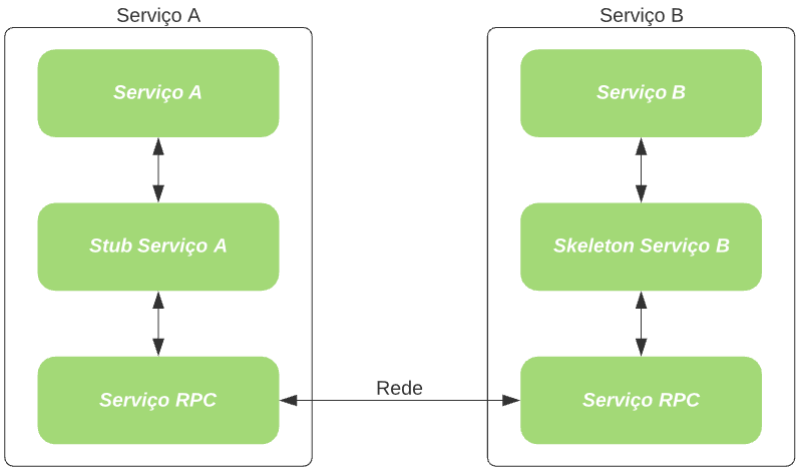



Figura 6: Comunicação RPC.

Como alguns exemplos de RPC temos Java RMI e Protocol Buffers .

### *Message Brokers*

Um *message broker* é um *middleware* que permite aplicações comunicarem entre si e trocarem informação. Este transforma uma mensagem que recebe do remetente para uma mensagem que o destinatário perceba, possibilitando que ambas as partes possam ser desenvolvidas em linguagens e tecnologias diferentes, atuando como intermediário entre aplicações. Este valida as mensagens, armazena-as, faz roteamento destas e entrega-as ao destino devido [IBM (2020b)].

Uma das vantagens dos *message brokers* deve-se ao facto de permitir uma comunicação assíncrona, o que faz com que o sistema a enviar uma mensagem não tenha que ficar bloqueado à espera da resposta. Estes permitem que o remetente envie uma mensagem para um destinatário sem saber a localização deste, se estão ou não ativos, ou quantos outros sistemas existem. O sistema apenas tem conhecimento do *message broker* sendo de lá que recebe e envia mensagens, permitindo um maior desacoplamento entre sistemas, podendo até ser substituído por uma nova versão e os sistemas que interagem com este continuam operacionais, não necessitando de nenhumas modificações [IBM (2020b)].

A um *message broker* está muitas vezes associada uma ou várias *message queues* que permitem o armazenamento e ordenação de mensagens até que os destinatários processem estas mensagens.

Existem alguns modelos de *message broker*, de entre os quais temos:

- **Ponto a Ponto:** Este modelo é utilizado em situações em que existe uma relação um para um entre o remetente e o destinatário. Cada mensagem na *queue* é enviada unicamente para um destinatário e consumida uma única vez.

- **Publicação/Subscrição:**

  Neste modelo, um remetente publica uma mensagem para um tópico, sendo que vários destinatários podem subscrever a esse tópico e consumir as mensagens para lá enviadas. Neste caso existe uma relação de um para muitos.

  Um sistema pode-se subscrever a vários tópicos e publicar para vários tópicos.

#### *Comunicação baseada em Eventos*

Outra forma de comunicação entre aplicações é uma comunicação baseada em eventos. Neste padrão arquitetural as aplicações atuam como produtoras ou consumidoras de eventos, podendo até desempenhar os dois papeis [IBM (2020a)].

Um evento é um registo de algo que aconteceu, de uma mudança de estado. Os eventos são imutáveis, não podem ser modificados nem eliminados e encontram-se por ordem de criação [Jansen and Saladas (2020)].

Quando uma aplicação executa localmente uma ação ou uma mudança de estado que pretende que outra aplicação tenha conhecimento, esta produz um evento, representando essa ação ou mudança, que a outra deva tomar conhecimento. Essa outra aplicação deve estar subscrita a estes eventos para os consumir e posteriormente processar esses eventos, executando uma ou mais ações [Jansen and Saladas (2020)]. As aplicações possuem um maior desacoplamento, visto que permite que comuniquem entre si de forma assíncrona sem que nenhuma delas necessite de qualquer conhecimento sobre a outra.

Existem duas formas de transmitir eventos:

- **Event Messaging (Publicação/Subscrição):**

  Este modelo é o apresentado na secção 2.2.4.

- **Event Streaming:**

  Neste modelo, existe uma *stream* onde os eventos são publicados. Um produtor gera um evento e coloca-o na *stream*. Um consumidor subscreve uma ou mais *streams*, contudo em vez de receber e consumir todos os eventos publicados na *stream*, este consegue apenas consumir os eventos que pretende [IBM (2020a)].

  A diferença desta forma para a anterior é que os eventos permanecem na *stream* mesmo após serem consumidos. Os consumidores podem subscrever-se a qualquer *stream* em qualquer altura do tempo, podendo consumir eventos que tenham ocorrido antes da sua subscrição.

### 2.2.5 *Vantagens sobre uma arquitetura monolítica*

A arquitetura de microsserviços possui várias vantagens, sendo muitas delas adquiridas de sistemas distribuídos. No entanto, muitas destas vantagens atingem um maior grau de benefício devido à definição dos microsserviços e das suas fronteiras. Combinando os conceitos de ocultação de informação, abordado na secção 2.2.1 e de *domain-driven design*, abordado na secção 2.2.1 e 2.2.1 com os de sistemas distribuídos, a arquitetura de

microsserviços consegue possuir mais vantagens do que um simples sistema distribuído [Newman (2021)], apresentando-se a seguir em comparação com uma arquitetura monolítica.

- **Resiliência**:

  Numa aplicação monolítica se um componente falhar toda a aplicação falha.

  Uma forma de melhorar este aspeto é escalar a aplicação horizontalmente, isto é, adicionar mais máquinas onde a aplicação corre para que no caso de falha da máquina primária uma destas assuma o cargo e continue com a aplicação disponível. O problema é que caso a máquina primária falhe devido a um erro de um componente, a aplicação irá sempre falhar mesmo que outras máquinas tomem o seu lugar.

  Com microsserviços temos unidades que são independentes uns dos outros. A falha de um microsserviço não leva à falha de toda a aplicação. A aplicação continua a correr, porém, o microsserviço que falhou estará temporariamente indisponível para os outros da aplicação. Na situação desse microsserviço ter um erro que faça com que este falhe repetidamente é muito mais fácil de o corrigir e não terá tanto impacto na aplicação, visto que apenas esse será compilado e instalado novamente [Newman (2015)].

  É importante mencionar que a adoção de microsserviços não implica que tenhamos necessariamente maior robustez. Separar as funcionalidades da aplicação em microsserviços mais pequenos e colocar estes em processos separados não aumenta por si só a robustez da aplicação. Para a alcançar, deve-se adotar mecanismos que diminuam ao máximo a falha destes microsserviços ou perda de mensagens. É possível colocar os vários microsserviços a correr atrás de um balanceador de carga e também adotar um mecanismo de comunicação entre microsserviços que permita que mensagens não sejam perdidas caso algum falhe [Newman (2019)].

- **Escalonamento**

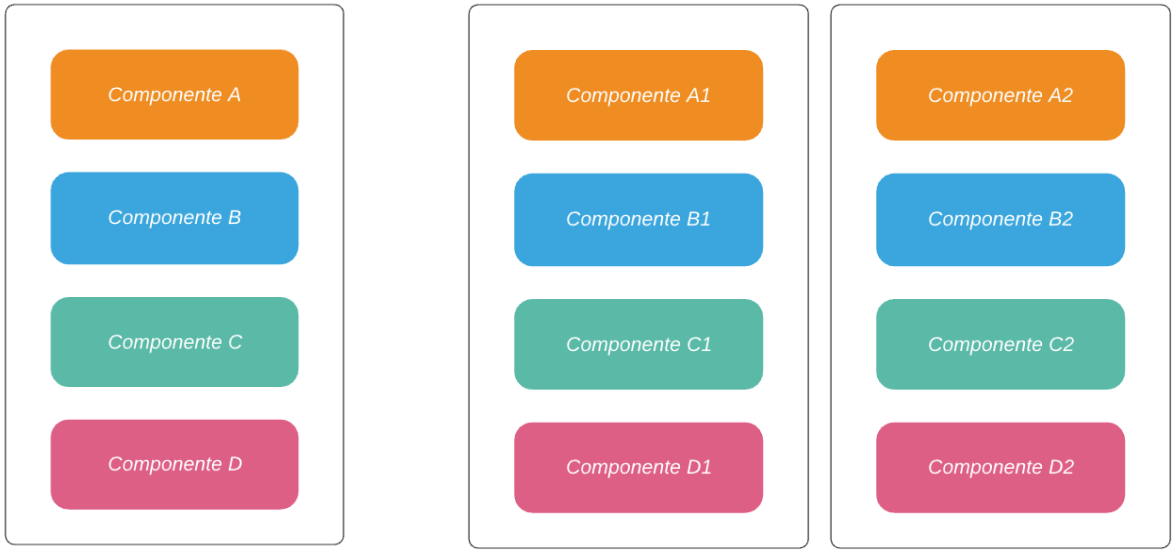
  Na Figura 7a consegue-se observar uma aplicação monolítica composta por quatro componentes que não está escalada. Visto que a única forma de escalar um monolítico é escalar a aplicação na totalidade, temos na Figura 7b um exemplo de escalonamento de uma aplicação monolítica. Neste exemplo colocou-se mais uma instância da aplicação.

  Numa aplicação em microsserviços é possível escalar independentemente [Bruce and Pereira (2018)].

  Imagine-se uma aplicação composta pelos quatro microsserviços apresentados na Figura 8 que comunicam entre si. Alguns destes podem estar a ter um tempo de resposta maior do que esperado, sendo necessário escalar.

  Como se pode observar pela Figura 9 cada microsserviço foi escalado conforme as suas necessidades. Nesta consegue-se observar que o microsserviço B foi escalado três vezes porque recebe muitos pedidos, ao invés do microsserviço D que só foi escalado uma vez, não recebendo provavelmente muitos pedidos ou que não possui muita carga computacional na execução destes.

- **Instalação**:




(a) Sem escalonamento (b) Com escalonamento

Figura 7: Escalonamento de uma arquitetura monolítica.

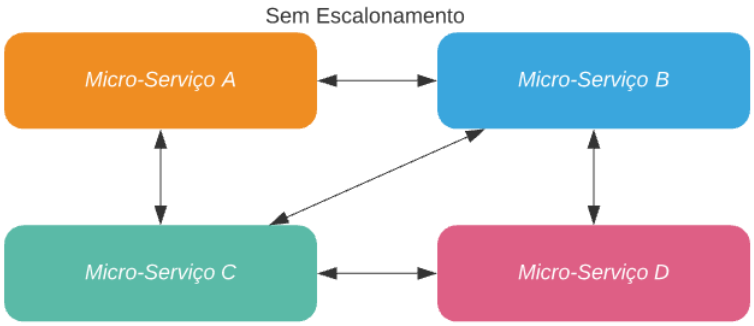



Figura 8: Arquitetura de microsserviços sem escalonamento.

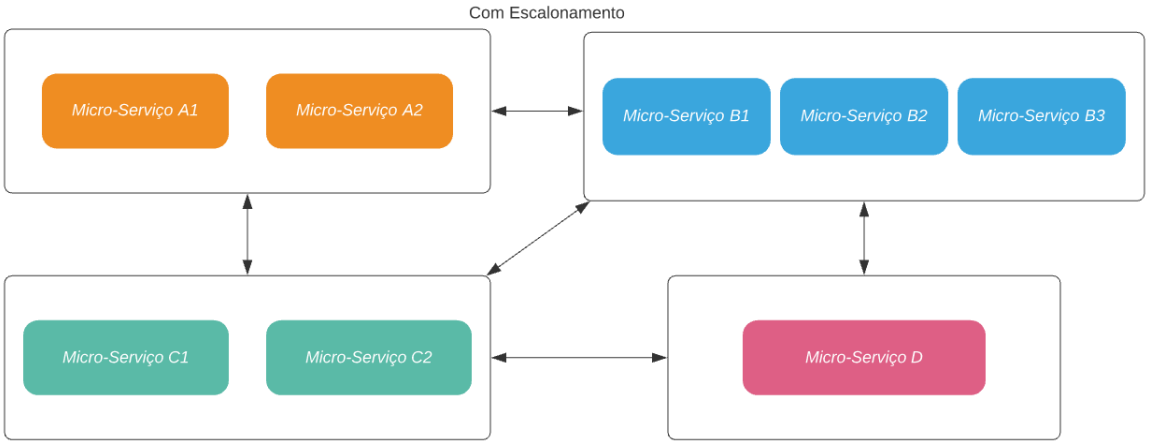



Figura 9: Arquitetura de microsserviços com escalonamento.

Numa aplicação monolítica a mudança de uma linha de código leva a que toda a aplicação precise de ser novamente compilada e instalada para que as mudanças sejam efetuadas, o que leva a uma instalação com um impacto enorme e também de alto risco, sendo que na eventualidade de a nova mudança ter um erro que tenha passado despercebido, será necessário despender mais tempo para corrigir o problema ou retroceder nas mudanças, levando a um maior tempo de inatividade da aplicação [Richardson (2018)].

Na arquitetura de microsserviços, a mudança num microsserviço não implica a instalação de toda a aplicação. Pode-se efetuar mudanças a um microsserviço e instalar sem ter impacto nos restantes, levando a uma instalação mais rápida e permitindo que novas funcionalidades sejam entregues ao cliente mais rapidamente [Newman (2015)].

Caso ocorra algum problema com a nova instalação, acontece apenas a um microsserviço isolado de todos os outros, sendo facilmente revertida caso necessário.

- **Substituível**:

  Numa arquitetura monolítica toda a funcionalidade encontra-se num mesmo lugar, levando a uma maior dificuldade e risco aquando da necessidade de querer modificar alguma parte do código [Richardson (2018)].

  Na arquitetura de microsserviços, o custo de trocar este microsserviço por uma implementação nova ou até apaga-lo é muito menor e mais fácil de gerir, não levando a conflitos com os outros [Newman (2015)].

- **Adoção de novas tecnologias**

  Como referido anteriormente cada microsserviço pode ser escrito numa tecnologia diferente da dos restantes, conseguindo assim tirar o melhor partido que uma tecnologia oferece para a sua implementação, de forma a ter o melhor desempenho possível [Nadareishvili et al. (2016)].

  Numa aplicação monolítica isto não acontece. Todos os componentes da aplicação devem ser escritos na mesma tecnologia [Richardson (2018)].

- **Estrutura da Equipa**

  A existência de microsserviços que são independentes e que estão em processos diferentes permite que várias equipas possam trabalhar em simultâneo para a mesma aplicação [Richardson (2018)].

  Para além disto, sendo as equipas mais pequenas e focadas num único propósito, torna-as mais autónomas, pois sendo pequenas em número aumenta a facilidade de se comunicarem e de não se atrapalharem em termos de escrita de código [Newman (2019)].

- **Entregas mais rápidas**

  A utilização de microsserviços permite maior rapidez na mudança de funcionalidades e na entrega de novas funcionalidades ao cliente [Newman (2019)].

  Se existirem muitas mudanças necessárias a efetuar à aplicação, tendo os microsserviços independentes entre si, é mais fácil fazer modificações e voltar a instalar sem que toda a aplicação tenha que reiniciar [Newman (2015)].

### 2.2.6 *Desvantagens*

A arquitetura de microsserviços traz várias vantagens, como se pode constatar na secção anterior. Contudo, possui também desvantagens, apresentadas a seguir.

- **Complexidade**

  Uma arquitetura de microsserviços é um sistema distribuído, composto por vários microsserviços em processos diferentes, que comunicam através da rede que pode ter atrasos e até falhas. É por isso mais complexo o desenvolvimento de aplicações onde é necessário ter preocupações com a entrega de mensagens, com as ligações entre microsserviços e a gestão de todas as base de dados [Richardson (2018)].

- **Difícil de Testar**

  Uma funcionalidade de uma aplicação em microsserviços pode implicar a interação entre vários microsserviços, o que pode dificultar o teste destas funcionalidades na totalidade, pois estes encontram-se separados.

- **Experiência de desenvolvimento**

  À medida que se têm mais microsserviços, a experiência de desenvolvimento começa a diminuir. A maioria dos ambientes de desenvolvimento, como JVM (Java Virtual Machine), limitam o número de processos que podem estar a correr numa única máquina. É possível correr talvez quatro ou cinco processos no mesmo computador sem problemas, mas dez ou vinte começa a ser mais complicado [Newman (2021)].

  Uma solução passa por começar a desenvolver na *cloud*, onde os programadores deixam de poder desenvolver localmente. Outra solução passa por limitar o número de microsserviços que cada programador necessita de trabalhar [Newman (2021)].

### 2.2.7 *Desafios*

A adoção de uma arquitetura de microsserviços para o desenvolvimento de um produto de *software* leva a alguns desafios que não apareceriam no caso de se ter adotado uma arquitetura monolítica.

Alguns desses desafios são:

- **Definição de um microsserviço**

  Quando se começa a desenvolver uma aplicação em microsserviços é necessário pensar que microsserviços vão existir, o que cada um deve ter, quais as suas fronteiras e o que expor destes [Bruce and Pereira (2018)].

  Esta dificuldade pode ser ultrapassada recorrendo à definição de *Bounded Context* como referido na secção 2.2.1. Contudo, mesmo com esta definição, a limitação das fronteiras de um microsserviço é sempre uma tarefa mais complicada e que se não for bem definida pode levar a complicações futuras.

- **Comunicação entre microsserviços**

  Sendo a arquitetura de microsserviços composta por microsserviços que se encontram instalados em processos diferentes é necessário que estes comuniquem entre si. Esta comunicação necessita de ser efetuada através da rede. A comunicação através da rede não é instantânea, o que significa que é necessário a preocupação com a latência e eventuais perdas de mensagens. Além disso, é necessário considerar que estas latências possam variar, o que leva a uma maior dificuldade na definição dos comportamentos dos microsserviços [Newman (2019)].

- **Falha de microsserviços**

  Um microsserviço pode falhar e desligar-se, deixando de responder a pedidos ou então começar a comportar-se de maneira diferente da que devia, podendo levar à indisponibilidade de algumas funcionalidades do sistema. É necessário prevenir a falha de um microsserviço, podendo-se escalar horizontalmente, onde se adicionam mais máquinas que disponibilizam o mesmo microsserviço [Newman (2019)].

- **Escalonamento dinâmico**

  Pode haver momentos do ciclo de vida de um produto de *software* em que este pode ter uma grande adesão e outros em que não. Por isso, um dos microsserviços pode ter a necessidade de em alguma situação ter mais máquinas, de escalar horizontalmente para fazer face ao crescimento de utilizadores. Mais tarde essa adesão pode diminuir não havendo necessidade de ter tantas máquinas, podendo assim libertar algumas.

  A complexidade aparece quando se acrescentam novas máquinas para um dos microsserviços e se quer que estas sejam efetivamente usadas, isto é, que estejam também a ser usadas pelo *load balancer*, de modo a que a carga seja distribuída igualmente por todas.

- **Visibilidade**

  A ocorrência de um problema na aplicação pode ser mais difícil de identificar, visto que temos vários microsserviços em processos separados [Bruce and Pereira (2018)].

  Um pedido à aplicação pode envolver vários microsserviços, por isso na ocorrência de uma eventual falha é necessário identificar qual o microsserviço que originou o erro e porquê, para que o problema possa ser corrigido.

  É necessária por isso a existência de um *log* centralizado onde se possa verificar qual a origem de um problema.

  Também pode ser necessário monitorizar todos os microsserviços de forma centralizada, para verificar se algum deles está em baixo e também o desempenho de cada um.

- **Microsserviço muito utilizado**

  Pode existir um microsserviço que seja muito utilizado por todos os outros, estando todos esses dependentes deste. É necessário prevenir que este microsserviço esteja inoperacional, porque pode levar à indisponibilidade de quase toda a aplicação.

É necessário dotar a aplicação de mecanismos de tolerância a faltas e também escalar estes microsserviços muito consumidos para aumentar a disponibilidade e tempo de resposta da aplicação.

Concluindo, percebe-se que a utilização de microsserviços implica ter alguns desafios no desenvolvimento, instalação e manutenção da aplicação, alguns deles adquiridos dos sistemas distribuídos.

Para conseguir as vantagens apresentadas é necessário saber como automatizar todo o processo de instalação, teste e monitorização.

### 2.2.8 *Decisão arquitetural*

Em algumas situações, escolher uma arquitetura monolítica para o desenvolvimento de uma aplicação não é a escolha mais acertada.

O início do desenvolvimento de um produto de *software* é o momento exato para se começar a pensar nos diferentes microsserviços necessários e na separação destes [Newman (2015)].

Quando se constrói um monolítico a pensar no futuro de separar para microsserviços é errado pensar que existem componentes dentro deste monolítico prontos a ser facilmente separados e colocados em microsserviços. Até pode ser verdade que os limites dos componentes estejam bem definidos e se saibam quais se devem separar, porém, estes estão normalmente muito acoplados e cheios de ligações entre eles. Estes componentes estão todos a comunicar através de abstrações da mesma tecnologia que usam, partilham objetos, modelos e fazem parte do mesmo modelo de persistência, partilhando a mesma base de dados [Tilkov (2015)].

A escolha de uma arquitetura de microsserviços deve ter em conta:

- A rapidez com que se pretende a aplicação desenvolvida.

  Com a arquitetura de microsserviços consegue-se um desenvolvimento mais rápido, visto que a equipa de desenvolvimento pode dividir-se em subgrupos e desenvolver os microsserviços de forma autónoma e independente dos outros [Bruce and Pereira (2018)].

  Todavia, esta vantagem só acontece se se contemplar o próximo ponto.

- A familiarização com microsserviços.

  Se se pretende um desenvolvimento rápido com esta arquitetura é necessário conhecimento e experiência em microsserviços. Se a equipa de desenvolvimento não possui conhecimentos nesta área, torna-se difícil e custoso o desenvolvimento da aplicação. O tempo de entrega da aplicação para o cliente é muito maior e a resolução de eventuais problemas que apareçam serão muito mais difíceis de alcançar [Bruce and Pereira (2018)].

- A necessidade de alguma parte da aplicação ser mais eficiente.

  Pode existir algum componente que se prevê que seja mais consumido pelos utilizadores e tenha assim um maior volume de acesso. Uma aplicação onde a funcionalidade principal seja a procura por algo que a aplicação oferece, pode ser problemático não ter esse componente de procura o mais eficiente possível.

A utilização de microsserviços neste caso é útil, dado que esse componente mais utilizado pode ser um microsserviço que pode ser escalado independentemente dos outros e assim aumentar o desempenho e a eficiência deste [Indrasiri and Siriwardena (2018)].

## 2.3 SERVICE-ORIENTED ARCHITECTURE (SOA)

Ao falar de microsserviços é necessário falar também sobre Service-Oriented Architecture (SOA), porque microsserviços pode ser considerado um tipo de *service-oriented architecture* [(Newman, 2021, p. 5)].

SOA é uma arquitetura de *software* que define o desenvolvimento de uma aplicação como uma composição de componentes de *software* que se encontram separados [IBM (2019b)]. Com esta abordagem pretende-se que se construa uma aplicação utilizando serviços já desenvolvidos, totalmente independentes que se encontram disponíveis a fornecer as suas funcionalidades, de modo a construir a nossa aplicação de forma mais desacoplada, sem necessidade de voltar a desenvolver outra vez um componente já existente [Josuttis (2007)]. O objetivo é promover a reutilização destes serviços por outras aplicações [Javed (2019)].

Imagine-se que já existe um serviço que está encarregue de fazer autenticação de um utilizador através de APIs externas, como Google Authenticator. Em SOA, é utilizado este serviço já implementado ao invés de reescrever um componente que faça essa funcionalidade.

Nesta arquitetura um serviço é um processo completamente separado encarregue por uma tarefa que comunica com outros serviços através da rede usando protocolos de comunicação como REST, SOAP, entre outros. Existem dois tipos de serviços, os que fornecem e os que consomem. Um serviço pode no entanto agir das duas formas [IBM (2019b)]. A comunicação entre estes serviços é feita através de *Enterprise Service Bus* (ESB).

*Enterprise Service Bus* é um *middleware* que permite a integração de vários sistemas de forma a estes poderem comunicar entre si [Josuttis (2007). ESB permite aos serviços conectarem-se ao *bus*, podendo subscrever para receber um tipo de mensagens e publicar mensagens neste, sendo que o ESB fica encarregue de fazer o roteamento das mensagens [Schmidt et al. (2005)]. Consegue-se o desacoplamento por parte dos serviços, permitindo que estes tenham uma comunicação entre eles sem necessitarem de conhecimento acerca do outro serviço [IBM (2019a)].

SOA surgiu como uma evolução dos sistemas distribuídos e uma alternativa às aplicações monolíticas que acabavam por ser difíceis de manter, promovendo a reutilização de serviços e funcionalidades [IBM (2020c)].

Tendo como objectivo facilitar a manutenção e escrita de *software*, é possível substituir um serviço por outro que disponibilize as mesmas funcionalidades sem que outros serviços o saibam, desde que a semântica do serviço não mude muito.

### 2.3.1 *Diferença entre SOA e Microsserviços*

A maior diferença entre SOA e microsserviços encontra-se na forma como as ligações entre serviços são feitas, o âmbito da sua utilização e o propósito de cada uma.

SOA é usado em contexto empresarial, permitindo que aplicações/serviços desenvolvidos em contexto empresarial possam ser expostas, cada uma correspondendo a um conjunto de funcionalidades [Erl (2007)]. Desta maneira permitimos que outras aplicações possam reutilizar as funcionalidades de outras aplicações/serviços.

Microsserviços é usado em contexto aplicacional, permitindo que uma aplicação possa ser particionada em microsserviços menores que são independentes de modo a atingir vantagens mencionadas anteriormente. Não define como as aplicações devem comunicar entre elas [Richards (2015)].

SOA está focado em tentar maximizar a reutilização de funcionalidades já implementadas noutros serviços, enquanto que microsserviços foca-se mais no desacoplamento entre componentes [IBM (2020c)].

Em SOA todos os serviços podem partilhar a mesma base de dados, sendo que a decisão é de quem constrói a aplicação, enquanto que em microsserviços cada microsserviço possui a sua base de dados [IBM (2020c)].

# 3

# MIGRAÇÃO PARA MICROSSERVIÇOS

A maioria das aplicações criadas acabam por nunca serem imutáveis. No processo de desenvolvimento de uma aplicação é necessário considerar, que mesmo quando a aplicação é entregue ao cliente, que esta precisará de ser mudada e adaptada [Newman (2015)].

Numa aplicação é quase impossível prever todas as funcionalidades e requisitos da aplicação. Por esta razão é necessário preparar a aplicação para que esta seja modificável sem grande impacto.

Um monolítico cresce ao longo do tempo, adquirindo novas funcionalidades e cada vez mais ligações entre componentes, o que torna o código mais acoplado. Quando houver a necessidade de mudar uma linha de código, esta modificação terá grande impacto na aplicação, pois leva à instalação de toda a aplicação [Fowler and Lewis (2014)]. A adoção de uma arquitetura de microsserviços pode facilitar na mudança e adição de novas funcionalidades, bem como uma entrega mais rápida de funcionalidades para o cliente. O maior problema em migrar de uma arquitetura monolítica para uma de microsserviços, prende-se na definição do que deve ser cada microsserviço e na mudança da forma de comunicação entre estes [Newman (2015)].

A adoção de uma arquitetura monolítica deve ser uma decisão consciente. Uma decisão que parte do princípio de se pretender algo que não se consegue com o sistema arquitetural que já se possui [Newman (2021)].

## 3.1 MIGRAÇÃO INCREMENTAL

> "If you do a big-bang rewrite, the only thing you're guaranteed of is a big bang."
>
> (Martin Fowler)

A migração para microsserviços não deve ser realizada abruptamente. É aconselhado realizar mudanças incrementais, extraindo funcionalidades uma a uma. Esta migração de forma incremental ajuda a que se aprenda mais sobre microsserviços à medida que se efetua esta migração e limita o impacto de se acabar por realizar algo mal [Newman (2019)].

> "Think of our monolith as a block of marble. We could blow the whole thing up, but that rarely ends well. It makes much more sense to just chip away at it incrementally."[1]
>
> (Sam Newman)

1 Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2nd edition, August 2021

Transformar um monolítico numa arquitetura de microsserviços pode ser muito dispendioso, principalmente se este estiver bastante acoplado. Efetuar todas as mudanças em simultâneo, leva a que seja difícil de perceber se a migração está a ser concretizada de forma correta ou não. É muito mais fácil particionar esta mudança em pequenas etapas, sendo cada uma focada numa única ação. É preciso perceber que vão acontecer erros e que algo pode ser efetuado incorretamente. A adoção de uma migração incremental permite que sejam reduzidos os erros a efetuar e a dimensão destes [Newman (2019)].

É aconselhado começar por separar apenas um ou dois componentes da aplicação em microsserviços, proceder à sua instalação e avaliar se tudo correu bem e o impacto que esta mudança trouxe à aplicação [Newman (2019)].

O custo de modificar código e movê-lo de um lado para o outro é relativamente pequeno. Existem variadas ferramentas e editores de texto que nos ajudam nesta tarefa e que possibilitam retroceder nas ações realizadas. Contudo, modificar uma base de dados é muito mais trabalhoso e reverter estas ações ainda mais. O mesmo se aplica caso se tenha que reescrever uma API que é consumida por muitos outros microsserviços, visto que é necessário modificar todos os outros para ficarem atualizados com esta mudança [Newman (2021)].

Uma boa forma de começar a pensar numa migração parte pelo desenho da arquitetura de como se pretende a aplicação. Desta forma conseguimos visualizar os microsserviços que se pretende que existam e que ligações existem entre eles. É preciso imaginar alguns exemplos de funcionalidades, tal como o registo de utilizadores e perceber se existem dependências cíclicas. Caso existam é porque algo não está correto e deve ser pensado como remover estes ciclos. O mesmo se aplica no caso de se perceber que existam dois microsserviços que comunicam muito entre si, indicando talvez que estes devam formar um só [Newman (2019)].

## 3.2 SEPARAÇÃO DO CÓDIGO

O primeiro passo na transformação de um monolítico para microsserviços passa pela organização do código.

Na maioria das vezes a principal barreira em migrar de um monolítico para microsserviços é porque o código se encontra pouco coeso e não está organizado à volta de domínios [Newman (2019)].

*Seam*, um conceito introduzido no livro *Working Effectively with Legacy Code* [Feathers (2004)], define-se como uma parte do código que pode ser isolada do resto, sem ter impacto no resto do código. É uma parte onde se podem efetuar mudanças sem se terem que mudar outras partes devido a estas mudanças efetuadas [Feathers (2004)]. Estas *seams* podem ser vistas como alternativa para uma boa organização do código, consistindo na organização do código por *namespaces* ou *packages*, dependendo da linguagem utilizada, onde dentro de cada um se encontra código direcionado para um mesmo propósito [Newman (2019).

No monolítico o que se pretende é que se encontrem *seams*, conseguindo-se obter uma melhor organização do código, para que posteriormente se consiga definir mais facilmente os microsserviços [Newman (2015)]. Uma boa identificação destas *seams* pode ser efetuado através da identificação de *Bounded Context* como falado na secção 2.2.1.

Com estes *seams* identificados, o próximo passo é transformá-las em módulos, tornando o monolítico num monolítico modular, como apresentado na secção 2.1.1. Continua-se a ter uma única aplicação que tem que ser

instalada conjuntamente, porém esta unidade é composta por vários módulos que são bem visíveis e podem ser extraídos para microsserviços. A natureza destes módulos depende da tecnologia a ser utilizada. Para uma aplicação em Java o monolítico modular pode consistir em múltiplos ficheiros JAR [Newman (2019)].

No entanto, mesmo com estes *Bounded Context* definidos, o código continua acoplado. Existem *Bounded Context* que acedem a funcionalidades ou objetos de outro. A separação destes para microsserviços deve ser realizada de forma incremental, começando por aquele que possui menos dependências de outros *Bounded Context*.

## 3.3 PADRÕES DE MIGRAÇÃO - CÓDIGO

Existem várias formas para efetuar a separação do código. Nesta secção serão apresentados alguns padrões que facilitam esta tarefa.

### 3.3.1 *Strangler Fig Application*

Uma técnica frequentemente usada quando se pretende efetuar uma reescrita de um sistema é a de *Strangler Fig Application* [Fowler (2004)]. Martin Fowler inspirou-se para este padrão num tipo de figueira (*fig*), denominada *strangler fig*. Estas nascem nos ramos mais altos de uma árvore e à medida que vão crescendo, descem pela árvore até que chegam ao solo e se enraízam. Neste processo elas envolvem a árvore que se torna na estrutura de suporte desta figueira. A árvore vai gradualmente morrendo e por último apodrecendo, deixando apenas a nova figueira, que já possui o seu próprio suporte e independente da árvore antiga [Fowler (2004)].

Esta metáfora é uma boa forma de explicar uma técnica de efetuar uma reescrita a um sistema. Cria-se um novo sistema à volta do sistema antigo, deixando este ir crescendo gradualmente à volta do antigo até este ficar obsoleto e deixar de ser necessário [Fowler (2004)].

A ideia é que o sistema antigo e o novo podem coexistir em simultâneo, dando tempo ao novo sistema de crescer e de substituir por completo o antigo. Com isto, procede-se a uma migração incremental para o novo sistema, como referido na secção 3.1, adquirindo todas as vantagens desta. É possível retroceder alguma ação que se tenha feito [Newman (2021)].

A execução do padrão de *strangler fig* consiste em três passos, como se pode observar pela Figura 10. Primeiro é necessário identificar as partes do sistema que se pretende migrar. A seguir é preciso implementar estar partes em microsserviços. Por último, estando este implementado, é necessário redirecionar as chamadas a estas partes no monolítico para serem agora efetuadas ao novo microsserviço criado [Newman (2021)].

Caso esta parte extraída seja também utilizada por outras partes no monolítico é preciso também reescrever estas partes, visto que agora as chamadas não serão locais. Por outro lado, se este novo microsserviço necessitar de alguma funcionalidade que se encontre no monolítico, é preciso realizar mudanças a este para se expor esta funcionalidade ao novo microsserviço criado [Newman (2021)], algo mencionado na secção 3.8.1.

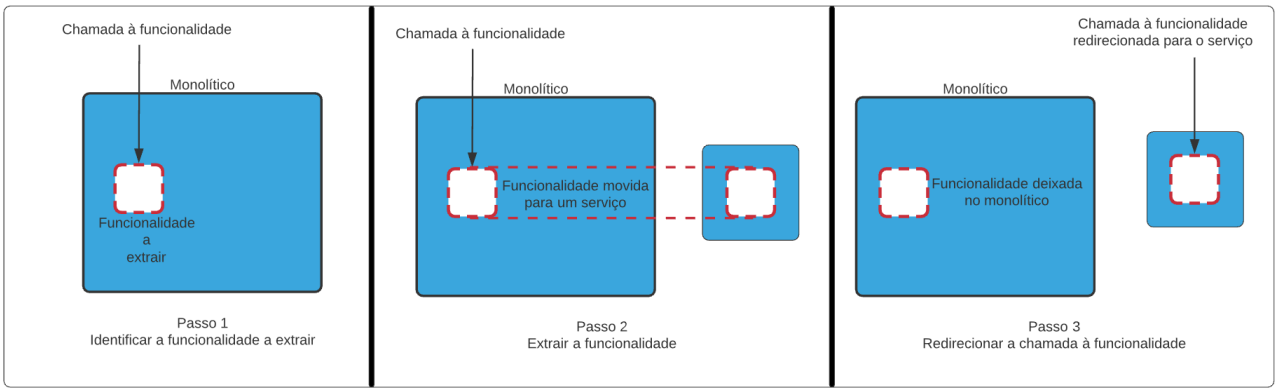



Figura 10: Strangler Fig Application.

Este padrão permite que seja movido e extraído funcionalidade do monolítico para microsserviços sem ter que o modificar nem alterar, sendo uma vantagem quando o próprio monolítico continua a ser desenvolvido e a ser adicionado com novas funcionalidades [Newman (2021)].

### 3.3.2 *Branch by Abstraction*

Para que o padrão *strangler fig application* possa ser aplicado e bem-sucedido, é preciso que a funcionalidade a extrair não esteja muito acoplada no monolítico, ou seja, devem ser partes que não são tão utilizadas por outras no monolítico [Newman (2019)].

Para extrair estas partes bastante utilizadas é preciso que modificações sejam feitas ao monolítico. Estas mudanças podem ser significantes e disruptivas para outras pessoas a trabalhar no mesmo código em simultâneo [Newman (2019)].

Muitas vezes, para mitigar esta disrupção são usadas *branchs* separadas para realizar estas mudanças. O desafio está quando o trabalho de migração a ser feito nesta *branch* é terminado e necessita de ser fundida com a *branch* principal, o que pode levar a conflitos e ainda mais trabalho. Quanto mais tempo a *branch* existir, maior serão os problemas a enfrentar. O que se pretende é realizar estas mudanças ao código com o menor impacto possível no trabalho que outros programadores estejam a realizar nele [Newman (2019)].

Para estes casos em que se pretende justamente começar por extrair aquelas partes mais enraizadas no monolítico, existe o padrão *branch by abstraction* [(Newman, 2019, p. 104)], que não necessita de recorrer à utilização de *branches* para a realização destas mudanças. Este padrão baseia-se em realizar mudanças ao código já existente, permitindo que várias implementações da mesma parte a modificar/extrair possam coexistir no mesmo código em simultâneo, sem se causar muita disrupção [Newman (2019)].

Este padrão consiste em cinco passos:

1. **Criar uma abstração para a funcionalidade a ser substituída**

   O primeiro passo é criar uma abstração que represente as interações entre o código a ser extraído e as partes que o chamam [Newman (2019)]. Como se pode observar pela Figura 11, a funcionalidade

que se pretende extrair é a representada pela letra C. É criado então uma abstração de C que a própria implementa.

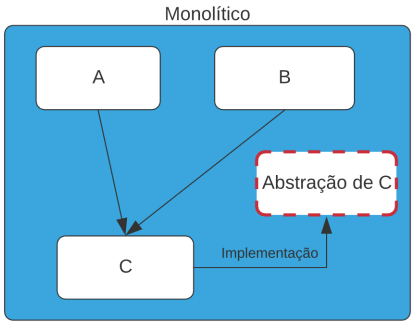



Figura 11: Criar uma abstração.

2. **Mudar as chamadas à funcionalidade para usarem a nova abstração**

   Com a nova abstração criada, é preciso agora redirecionar as chamadas que eram feitas a C para usarem agora a nova abstração, como se pode observar pela Figura 12.

   Este processo envolve uma análise ao código, para encontrar as chamadas efetuadas a esta funcionalidade. Neste ponto nenhuma mudança deve ter sido efetuada ao comportamento e código do sistema [Newman (2019)].

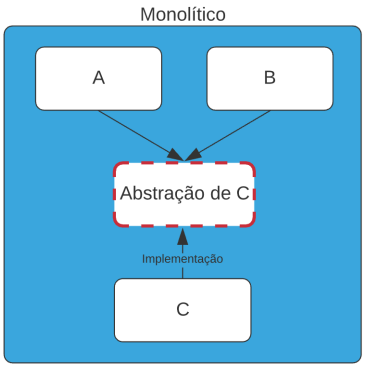



Figura 12: Redirecionar chamadas para a nova abstração.

3. **Criar uma nova implementação da abstração**

   Tendo agora a nova abstração criada é possível começar a desenvolver uma nova implementação desta [Newman (2019)]. No nosso caso, como se pretende uma migração para microsserviços, esta nova implementação realizará chamadas ao novo microsserviço que se desenvolverá, como se pode observar na Figura 13.

   É importante perceber que neste ponto apenas uma das implementações é usada pelo sistema, mesmo existindo duas implementações em simultâneo, da mesma abstração, o que permite ir-se desenvolvendo o novo microsserviço, sendo que este só é ativo e utilizado quando estiver completo [Newman (2019)].

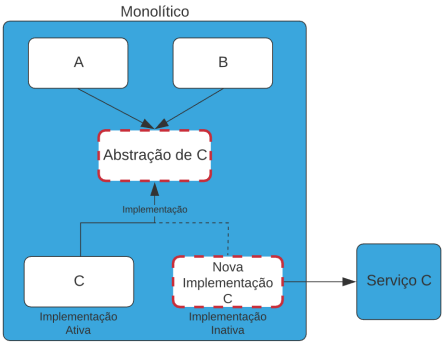



Figura 13: Nova implementação da abstração.

4. **Mudar a abstração para usar a nova implementação**

   Com o novo microsserviço implementado pode-se alterar a abstração para utilizar a nova implementação, ao invés da antiga [Newman (2019)], tal como mostra a Figura 14.

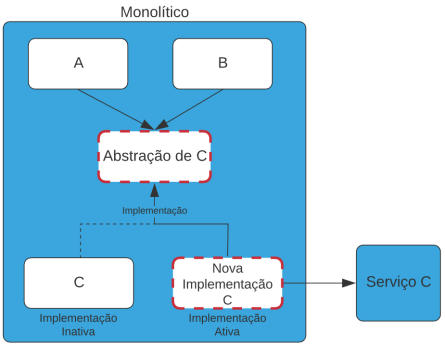



Figura 14: Mudar para usar a nova implementação.

   Idealmente, pretende-se ter um mecanismo que nos permita facilmente mudar de uma implementação para outra. Desta forma pode-se facilmente mudar para a implementação antiga caso algum problema aconteça com a nova [Newman (2019)].

5. **Remover a implementação antiga**

   Com o novo microsserviço a fornecer toda a funcionalidade que a antiga implementação fornecia, pode-se proceder à remoção desta. Visto que esta já não é usada e tem-se a certeza que o novo microsserviço funciona como esperado é possível eliminar esta implementação antiga [Newman (2019), como se pode observar pela Figura 15.

   Por último, com a implementação antiga removida, também temos a possibilidade de remover a abstração criada para este processo, como se pode observar na Figura 16.

   É possível, no entanto, que esta nova abstração tenha melhorado a estrutura e organização do código ao ponto de não se querer remover.

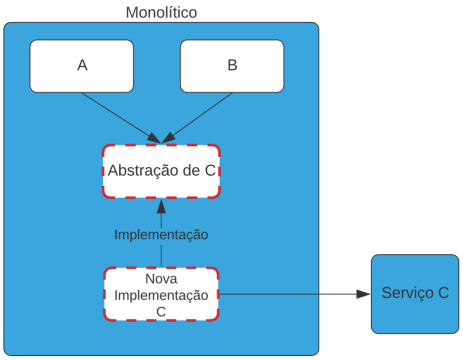



Figura 15: Remover implementação antiga.

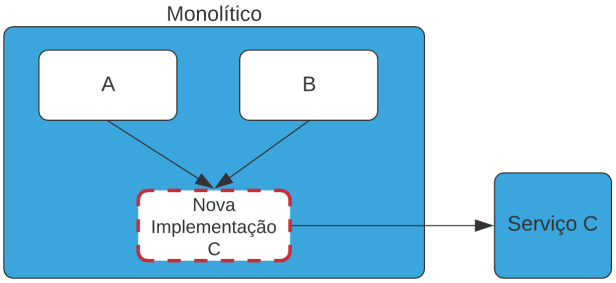



Figura 16: Remover a abstração.

Com este padrão consegue-se progredir e ir adicionando novas funcionalidades à aplicação sem haver disrupções entre a equipa de desenvolvimento, sendo possível instalar as modificações para produção em qualquer altura, dado que a aplicação se encontra sempre funcional. Também não existe a pressão de terminar rapidamente esta extração para um microsserviço, pois é possível modificar a funcionalidade a ser extraída em simultâneo com a migração [Hammant (2007)].

### 3.3.3 *Verify Branch by Abstraction*

A habilidade de se ter um mecanismo que nos permita escolher a implementação que se pretende, seja ela a antiga ou a nova implementação, é muito útil [Newman (2019)]. Contudo, existe algum risco de haver alguma falha da aplicação quando se muda para a nova implementação, que pode não ter sido detetada antes de se ter efetivamente mudado para esta e eliminado a antiga [Smith (2013)].

O padrão *Verify Branch by Abstraction* é uma variante do padrão *Branch by Abstraction* apresentado na secção 3.3.2. Este é igual, à exceção que adiciona uma fase de verificação por detrás da abstração criada. Desta forma, é possível que no caso da falha da nova implementação se reverta as chamadas para a antiga de forma dinâmica. Além disso, sabemos que a abstração obriga a cada implementação a possuir o mesmo comportamento, porém não consegue garantir a mesma implementação do comportamento em ambas. Esta fase de verificação chama ambas as implementações com os mesmos parâmetros para as várias funcionalidades que expõe e verifica que o resultado é o mesmo. Desta forma, pretende-se mitigar as incompatibilidades entre implementações e saber na

totalidade quando a nova implementação se encontra terminada [Smith (2013)]. Este padrão encontra-se na Figura 17.

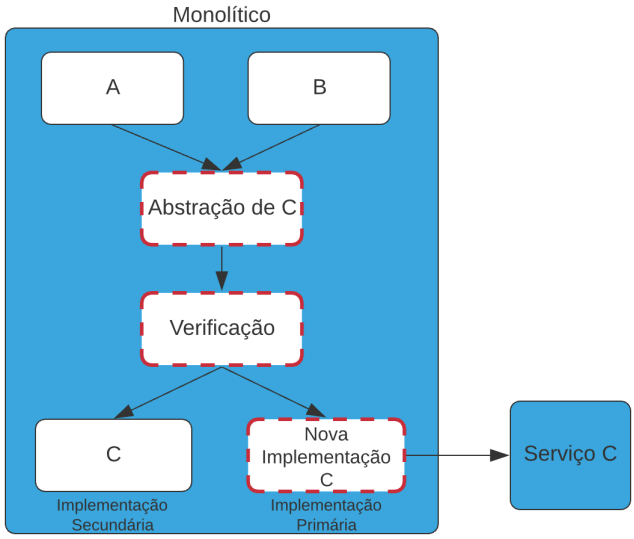



Figura 17: Verify Branch by Abstraction.

Com este mecanismo, diminui-se o custo de falha aquando da mudança para a utilização da nova implementação, visto que quando ocorre a falha de uma chamada à nova implementação é usada a antiga ao invés desta [Newman (2019)].

### 3.3.4 *Parallel Run*

Os padrões de migração *Strangler Fig Application* 3.3.1 e *Branch by Abstraction* 3.3.2 permitem que tanto a implementação antiga como a nova coexistam, permitindo que se execute ou a antiga do monolítico, ou a nova que já se encontra num microsserviço. Estas também permitem que se possa reverter para a implementação antiga em caso de falha da nova implementação, diminuindo assim o risco desta extração [Newman (2021)].

Com o padrão *parallel run*, em vez de se usar apenas uma implementação, usam-se as duas, permitindo que se compare os resultados para se garantir que ambas as implementações são equivalentes. Apesar de ambas as implementações estarem a receber chamadas às suas *interfaces*, apenas uma delas é considerada como primária num determinado momento e é desta a resposta a enviar da chamada. Normalmente, a implementação antiga é considerada como primária até que a nova implementação se verifique ser de confiança [Newman (2021)]. Os resultados das respostas dos pedidos feitos a ambas as implementações são guardados, para serem analisados e se verificar que são idênticos. Este processo pode ser observado pela Figura 18.

Com este padrão, não se garante apenas que a nova implementação retorna a mesma resposta que a antiga, mas que também apresenta o desempenho pretendido, tal como se demora muito a responder ou se não devolve resposta. Este padrão é tipicamente utilizado nos casos onde a funcionalidade a ser mudada é considerada de risco [Newman (2019)].

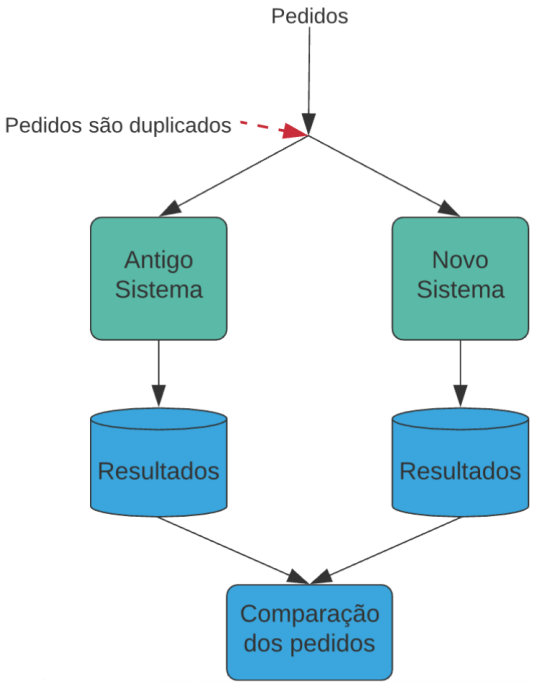



Figura 18: Parallel Run.

Existem situações em que uma ação desencadeia outra, tal como o envio de um correio eletrónico de boas-vindas quando um utilizador se regista numa aplicação. Neste caso, não se quer que o utilizador receba dois correios eletrónicos. Para este caso é preciso que se simule a ação no sistema que não é o primário [Newman (2019)].

Este padrão é uma forma de implementar uma técnica denominada *dark launching*. Com esta técnica, a nova implementação é instalada e testada, sem estar visível nem acessível aos utilizadores [Fowler (2020)].

É importante referir que este padrão não é o mesmo que *canary release* [(Newman, 2019, p. 118)]. Uma *canary release* é uma técnica para reduzir o risco de introduzir uma nova implementação do sistema em produção. Nesta, apenas alguns utilizadores têm acesso ao novo sistema antes de ficar disponível para todos os utilizadores [Sato (2014)]. A ideia é que se a nova implementação tiver um problema, apenas um pequeno conjunto de utilizadores sofre o impacto destes problemas [Newman (2019)].

## 3.4 BASE DE DADOS PARTILHADA

Antes de se começar a pensar na separação da base de dados, vão ser abordados alguns padrões arquiteturais onde microsserviços partilham o mesmo esquema de base de dados. Apesar de não ser uma boa prática na arquitetura de microsserviços, é uma opção que pode ser temporária.

### 3.4.1 *Esquema de base de dados partilhado*

Aquando da migração para uma arquitetura de microsserviços, pode-se optar por não se particionar o esquema da base de dados e ter um partilhado por todos os microsserviços, como se pode observar na Figura 19. Esta opção traz menor complexidade, dado que não se separa o esquema da base de dados nem se lida com as dificuldades que possam ocorrer.

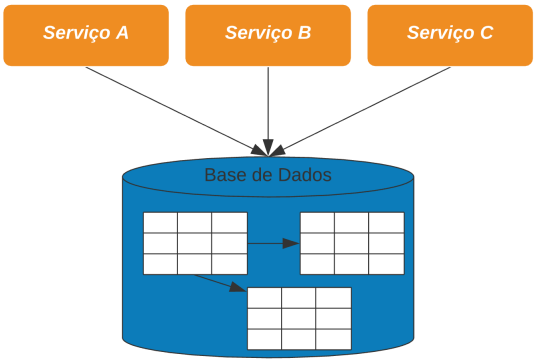



Figura 19: Esquema da base de dados partilhada por vários microsserviços.

No entanto, existem alguns problemas de se usar um esquema de base de dados partilhado entre vários microsserviços. Permite-se que todos tenham acesso à informação que não lhes pertence, perdendo assim a oportunidade de decidir que dados devem ser partilhados e ocultos para outros microsserviços, o que significa que fica complicado de perceber que partes do esquema da base de dados podem ser modificadas de forma segura [Newman (2019)].

É problemático também no sentido que fica confuso de saber quem possui certos dados, implicando uma falta de coesão, pois se na Figura 19 os três microsserviços puderem fazer alguma operação sobre uma mesma tabela, pode acontecer de este comportamento estar diferente em cada um. Mesmo que não esteja diferente, a mudança de comportamento de uma mesma ação que estes façam sobre essa tabela, implica a alteração deste comportamento em mais do que um microsserviço [Newman (2019)].

O uso de um esquema de base de dados partilhados é apropriado para uma arquitetura de microsserviços quando se considera dados estáticos e apenas de leitura. Neste casos onde a estrutura de dados é bastante estável e provavelmente só é modificada por partes administrativas, uma base de dados partilhada pode ser uma boa opção [Richardson (2018)].

### 3.4.2 *Base de dados com vistas*

Ainda abordando a possibilidade da utilização de um mesmo esquema de base de dados, o uso de vistas pode ajudar a mitigar preocupações relacionadas com acoplamento. Com o uso de vistas, é apresentado a um microsserviço um esquema que é uma projeção restrita de um esquema base. Esta projeção, resultado de uma *query*, consegue limitar os dados visíveis para o microsserviço, ocultando informação que este não deve ter

acesso, permitindo que se possa modificar a base de dados desde que se consigam manter as vistas [Newman (2019)]. Esta opção é apresentada na Figura 20.

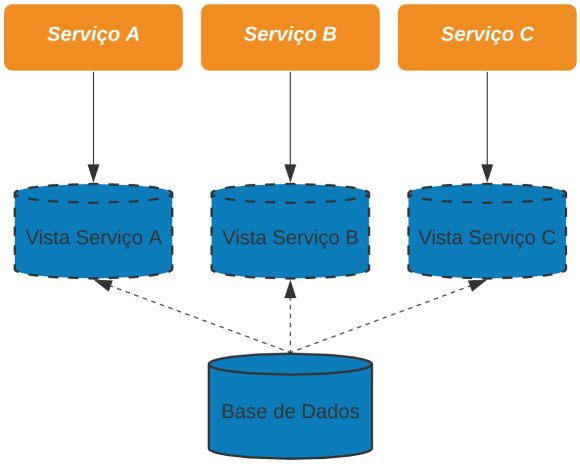



Figura 20: Utilização de vistas para cada microsserviço.

Dependendo da base de dados a utilizar, é possível a opção de criar vistas materializadas. Com uma vista materializada, a vista é previamente calculada, significando que uma leitura a este tipo de vistas não necessita de realizar uma leitura à base de dados, podendo assim ter mais desempenho. É necessário, que estas vistas materializadas estejam atualizadas com a base de dados, de modo a não serem lidos dados desatualizados [Newman (2019)].

Existem, no entanto, limitações ao uso de vistas. Dado que estas são o resultado de *queries* feitas à base de dados, estas permitem apenas leituras. Por outro lado, nem todos os motores de base de dados suportam vistas e ainda existem aqueles que não permitem vistas fora do mesmo esquema da base de dados, o que leva a um maior acoplamento aquando da instalação de algum microsserviço e também a um ponto único de falha [Newman (2019)].

### 3.4.3 *Encapsulamento da base de dados num microsserviço*

> "Sometimes, when something is too hard to deal with, hiding the mess can make sense."[2]
>
> (Sam Newman)

Pode acontecer de a base de dados ser bastante complicada de modificar, acabando por ser mais vantajoso de a esconder por detrás de um microsserviço [Newman (2019)].

Com este padrão, o que se pretende é encapsular a base de dados num microsserviço, movendo todos os acessos à base de dados para este [Newman (2019)].

Desta forma consegue-se controlar os dados que cada microsserviço pode aceder e quais estão ocultos. Posteriormente é preciso que todos os acessos à base de dados sejam agora redirecionados para o microsserviço,

2 Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2nd edition, August 2021

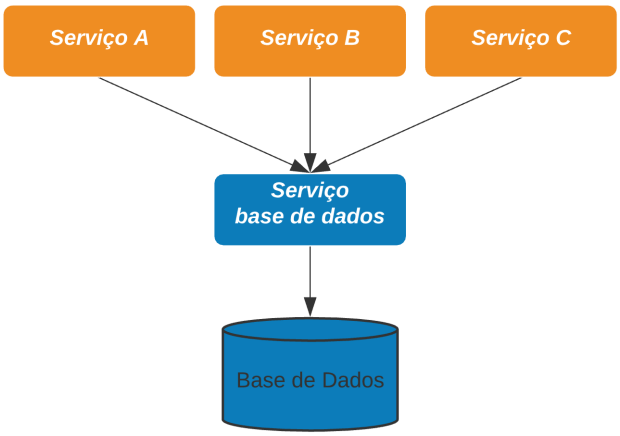



Figura 21: Encapsulamento da base de dados num microsserviço.

mudando de acessos diretos à base de dados para chamadas à API deste. Este padrão possui mais vantagens que o uso de vistas na base de dados, porque pode-se escrever código de forma a se apresentar projeções mais sofisticadas. Além disso, permite também escritas, algo que o uso de vistas não permitia [Newman (2019)].

### 3.4.4 *Base de dados como um microsserviço*

Em algumas situações, os microsserviços precisam apenas de realizar leituras à base de dados, seja para ir buscar enormes quantidades de dados ou apenas para ir buscar métricas sobre como se desempenha outro sistema. Nestas situações, permitir que outros microsserviços visualizem dados que outro gere na sua base de dados é vantajoso, desde que se separe a base de dados exposta e a gerida pelo próprio microsserviço [Newman (2019)].

Uma das abordagens é criar uma base de dados dedicada para ser exposta apenas para leituras, sendo esta populada quando existem mudanças na base de dados principal [Newman (2019)], como se apresenta na Figura 22.

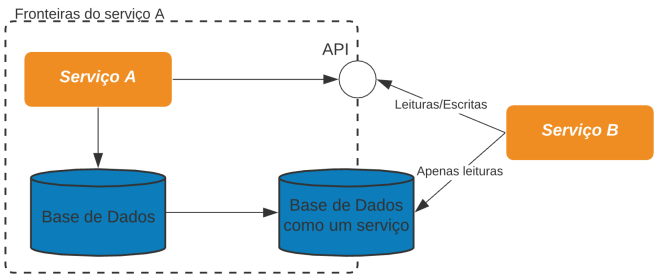



Figura 22: Base de dados como um microsserviço.

Como se pode ver, este padrão é vantajoso para microsserviços que necessitem apenas de realizar leituras.

Este padrão arquitetural tem algumas vantagens. É possível alterar a base de dados principal sem ter que se alterar a base de dados como um microsserviço. As leituras realizadas a esta base de dados como um

microsserviço não implica maior carga à base de dados principal e é possível ter várias para diferentes propósitos [Fowler (2014)]. Contudo, também possui desvantagens. Uma base de dados como um microsserviço necessita de estar atualizada com a base de dados principal [Fowler (2014)].

Em relação ao uso de vistas, este padrão é mais flexível, no sentido que não obriga a que seja usado a mesma tecnologia, podendo-se usar uma base de dados totalmente diferente da base de dados principal [Newman (2019)].

## 3.5 SEPARAÇÃO DA BASE DE DADOS

Assim como no código, na base de dados também é necessário encontrar partes que possam ser separadas [Newman (2015)]. É preciso perceber que partes efetuam leituras da base de dados e quais efetuam escritas.

Tal como ter o código separado por *namespaces* ou *packages*, o código que representa o acesso à base de dados deve também ser separado e estar junto do código a que pertence. Desta maneira consegue-se perceber que partes da base de dados são acedidas por certas partes de código [Newman (2015)].

A arquitetura de microsserviços funciona melhor quando cada microsserviço possui a sua informação, encapsulando-a e só permitindo certos acessos a esta. Apesar do que se apresentou na secção 3.4, é preciso ter em mente que quando se pretende migrar para uma arquitetura de microsserviços, é preciso separar a base de dados para que cada microsserviço tenha a sua própria, de modo a se obter as vantagens desta transição [Newman (2019)].

Um dos maiores problemas na separação de um monolítico é a base de dados, dado que é preciso considerar consistência entre os dados das várias bases de dados, transações, *joins*, latências, entre outros [Newman (2015)].

### 3.5.1 *Separação Física e Lógica*

Existem duas formas de realizar a separação do esquema da base de dados.

Existe a separação lógica, em que um único motor de base de dados possui mais do que um esquema de base de dados, como se apresenta na Figura 23.

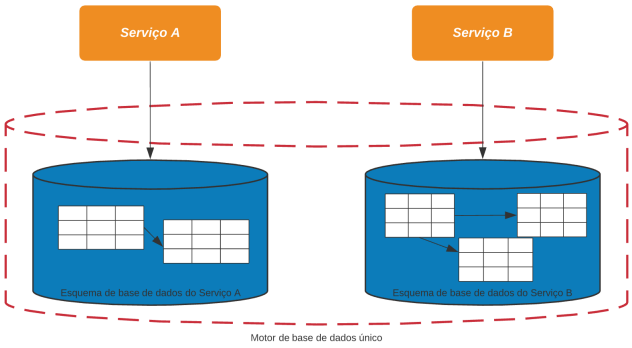



Figura 23: Separação lógica da base de dados.

O uso de uma separação lógica permite um processo mais simples e direto [Newman (2019)]. Contudo, fica-se sujeito a um ponto único de falha, dado que se o motor da base de dados falhar, ambos os microsserviços vão ser afetados.

Por outro lado, podemos ter separação física, tendo estes esquemas da base de dados em motores separados, como se observa na Figura 24.

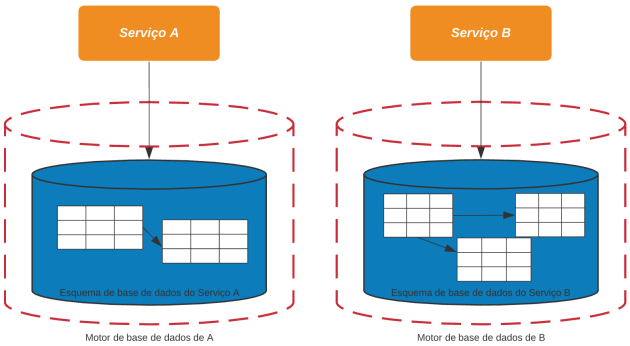



Figura 24: Separação física da base de dados.

O uso de uma separação física permite uma maior robustez ao sistema e ajuda a diminuir a contenção de recursos, melhorando o tempo de resposta e diminuindo a latência [Newman (2019)].

### 3.5.2 *Organização do código da base de dados*

Existem alguns desafios complexos quando se pensa em separar o esquema da base de dados.

Imagine-se que temos uma aplicação onde utilizadores podem fazer a gestão dos seus carros e realizar encomendas de componentes para os seus carros, tal como gerir informação sobre estes, datas em que algo é preciso ser feito ao carro, tal como uma mudança do óleo ou fazer uma encomenda de produtos para um certo carro, como um rádio, jantes, entre outros. Para isto existem vários componentes, dos quais se destacam um componente que trata de gerir os utilizadores e outro que trata de gerir os carros, como se pode observar na Figura 25.

Estando já o código organizado em componentes separados, pretende-se que o código de acesso à base de dados fique separado por repositórios como se apresenta na Figura 25.

Todavia, continuamos a ter componentes que acedem a tabelas que não lhes pertencem e existem relações entre tabelas, representadas por chaves estrangeiras.

Voltando ao exemplo mencionado anteriormente, consideremos que precisamos de saber quais são os carros de um certo utilizador. Para isso, dado que o componente do utilizador tem acesso à base de dados dos carros, pode ir diretamente à tabela dos carros buscar todos os que tenham como chave estrangeira o identificador do utilizador, como se pode observar pela Figura 26.

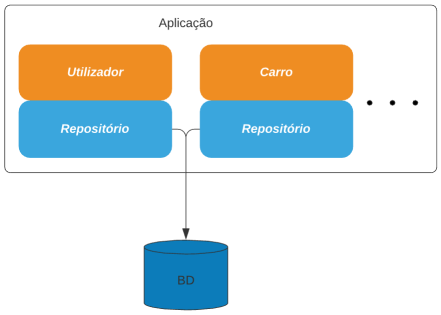



Figura 25: Separação por acesso à base de dados.

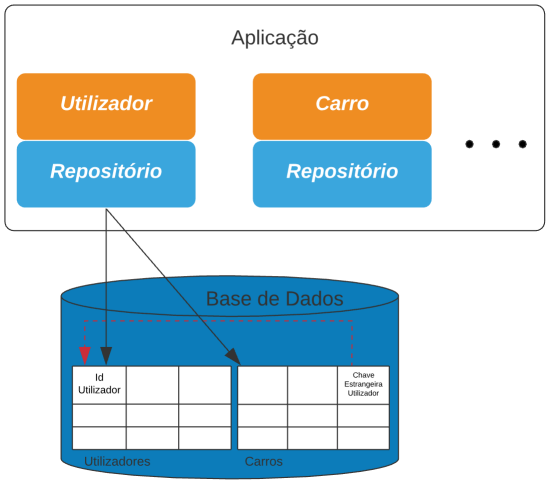



Figura 26: Chave estrangeira entre Utilizador e Carro.

### 3.5.3 *Remover Chaves Estrangeiras*

Antes de começar a pensar na separação do esquema de base de dados, seja esta separação física ou lógica, é preciso realizar mudanças no próprio esquema. É preciso não permitir que um componente tenha acesso a uma tabela da base de dados que não lhe pertence [Newman (2015)].

Caso se queira saber quais são os carros de um certo utilizador, é possível juntar as duas tabelas através da chave estrangeira e retirar apenas aqueles que são do utilizador pretendido. Como se constata na Figura 26 o componente do utilizador tem acesso à tabela dos carros e existe uma ligação entre a chave estrangeira da tabela dos carros e a chave primária dos utilizadores. Com esta relação entre tabelas, o motor de base de dados assegura a consistência dos dados, porque se existir um registo na tabela dos carros que se refere a um utilizador, é garantido que esse utilizador existe [Newman (2019)].

Se se pretender separar estes dois componentes para microsserviços independentes, é preciso que cada componente tenha apenas acesso às tabelas que lhe pertencem da base de dados e expor os seus dados através de uma API para que outros componentes possam aceder a esses sem ser diretamente à tabela [Newman (2015)], como é demonstrado na Figura 27.

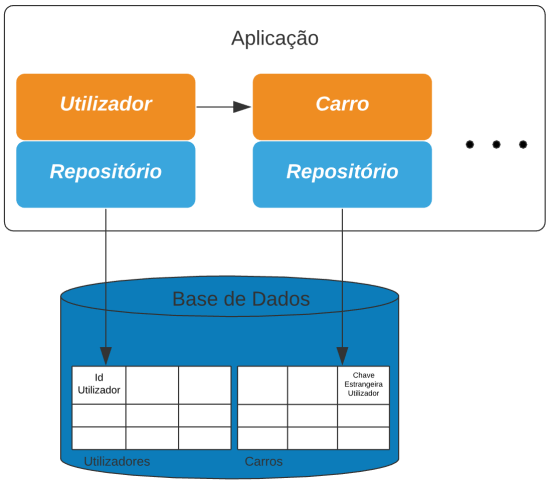



Figura 27: Remoção da chave estrangeira.

Voltando ao exemplo mencionado anteriormente, para saber quais são os carros de um certo utilizador, o componente dos utilizadores tem que encontrar o utilizador pretendido e quando o tiver, tem que ir buscar os carros deste. Nesta situação, em vez de ir diretamente à tabela dos carros, faz um pedido ao componente responsável por gerir os carros, fornecendo-lhe o identificador do utilizador. Este irá processar o pedido e devolver quais os carros relativos ao utilizador fornecido, como se observa na Figura 27.

Neste caso, a chave estrangeira que estaria presente na tabela dos carros relativa ao utilizador deixa de estar ligada à tabela dos utilizadores. Pode-se observar que a seta a vermelho na Figura 26 que representa uma ligação entre esta chave e a chave primária da tabela dos utilizadores, deixa de estar representada na Figura 27. Todavia, a coluna continua a existir, para se saber qual o utilizador de um certo carro, o que acrescenta alguma complexidade, visto que será necessário gerir estas colunas que não são chaves estrangeiras, mas que vão atuar como tal, para que a consistência entre microsserviços permaneça [Newman (2019)].

Voltando ao exemplo da aplicação mencionada anteriormente, caso um utilizador que possua carros seja removido, ficamos com linhas na tabela dos carros com informação inválida. Com um único esquema de base de dados, não seria possível eliminar um utilizador caso este tivesse carros associados, assegurando assim a consistência dos dados. Contudo, esta restrição já não existe, pois, separou-se estes dados para esquemas de bases de dados diferentes. Neste caso, existem algumas opções que dependem de situação para situação.

- Verificar antes de eliminar

  Uma das opções é quando se quer eliminar um utilizador, verificar se não existem referências na tabela dos carros. Um problema com esta opção, é que cria uma dependência para todos os microsserviços que possam ter referências para os utilizadores, sendo agora preciso verificar com todos os outros se estes possuem referências para um utilizador que se eliminará [Newman (2019)].

- Lidar com a eliminação

Outra opção é permitir que se possa eliminar utilizadores sem preocupações de este estar a ser utilizado por outros microsserviços. Neste caso, o microsserviço dos carros pode admitir que pode ter referências para utilizadores que já não existem.

Nesta situação, se se imaginar num contexto de *backoffice* em que se quer visualizar todos os carros na plataforma, pode haver carros que não tenham referência a utilizadores. Para esta situação pode-se indicar que o utilizador já existiu, mas que já não está mais presente [Newman (2019)].

- Eliminação em cascata

  Outra possibilidade é notificar os microsserviços aquando da eliminação de um utilizador, o que pode ser realizado através da subscrição de eventos ou outro mecanismo. Com isto, podem eliminar os registos que possuem referências a este utilizador eliminado [Newman (2019)].

- Não permitir a eliminação

  Uma última alternativa para assegurar que não se introduz inconsistência de dados no sistema é simplesmente não permitir que utilizadores possam ser eliminados. Uma possibilidade pode ser marcar o utilizador como eliminado, mas este permanecer na base de dados, podendo ser conseguido através de uma coluna que indica o estado do utilizador. Esta opção tem o problema de se poder ter vários registos deixados na base de dados que deviam estar eliminados [Newman (2019)].

Concluindo, é preciso sublinhar que nem todas as chaves estrangeiras devem ser removidas, pois podem ser tabelas que necessitam de estar juntas, fazendo parte de um mesmo agregado. Nestas situações visto que provavelmente irão pertencer ao mesmo microsserviço, não há necessidade de remover a restrição associada à chave estrangeira, dado que vão pertencer ao mesmo esquema de base de dados [Newman (2019)].

### 3.5.4 *Dividir uma tabela*

Podem existir situações em que se tem dados numa única tabela que necessitam de ser divididos entre dois ou mais microsserviços [Newman (2019)]. Na Figura 28 vemos do lado esquerdo uma tabela partilhada por dois contextos, o *A* e o *B*. O que se pretende é extrair *A* e *B* para novos microsserviços, contudo os dados destes estão numa única tabela. É preciso dividir estes dados em duas tabelas separadas, como se apresenta na Figura 28 no lado direito.

O ideal é separar primeiro as tabelas no esquema de base de dados do monolítico antes de os separar para bases de dados diferentes. Se estas tabelas existissem no mesmo esquema da base de dados, faria sentido declarar uma chave estrangeira numa das tabelas para a outra. Contudo, visto que estas vão estar em motores de bases de dados diferentes, não faz sentido esta chave estrangeira [Newman (2019)].

Este exemplo é bastante direto, separaram-se os dados para os microsserviços conforme os acessos que tinham às colunas, não tenho nenhum deles a aceder a colunas do outro. Contudo, podemos ter contextos a aceder à mesma coluna como se pode observar na Figura 29 do lado esquerdo.

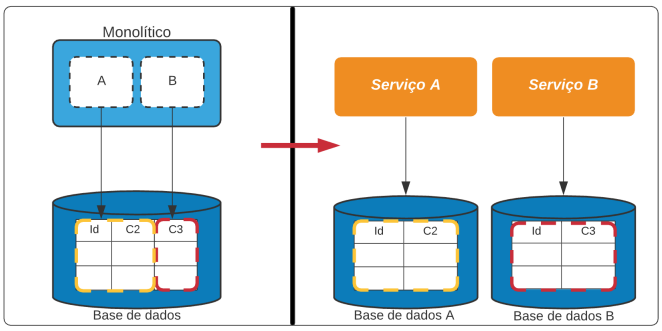



Figura 28: Dividir uma tabela.

Nesta situação apenas um dos microsserviços deve gerir esses dados, tendo o outro de aceder a estes para poder fazer leituras e escritas a esta coluna, como se demonstra na Figura 29 do lado direito. Nesta Figura o microsserviço *A* ficou encarregue dos dados, mas poderia ter sido ao contrário. Como referido na secção 2.2 o ideal é que cada microsserviço faça a gestão dos seus dados e que as mudanças de estado ocorram dentro destas, por isso nesta situação, tem que se perceber qual o microsserviço que de facto deve possuir estes dados.

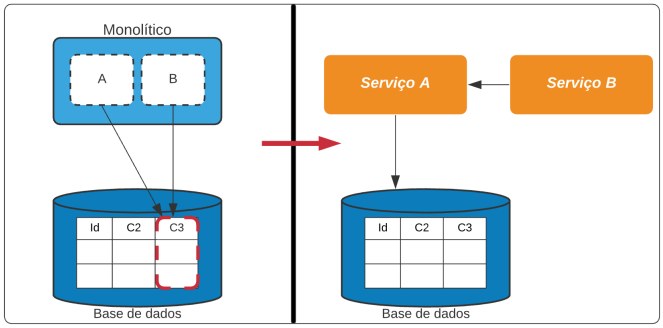



Figura 29: Dividir uma tabela com acessos à mesma coluna.

### 3.5.5 *Dados estáticos*

Existe em algumas aplicações conteúdo que é estático, conteúdo este que pode ser acedido por vários microsserviços. Muitas vezes este conteúdo estático é armazenado numa tabela na base de dados.

Imaginemos no exemplo da aplicação de gestão de carros. Existem várias marcas de carros e dentro das várias marcas existem vários modelos. Esta informação pode ser armazenada numa ou mais tabelas da base de dados.

Existem algumas opções a considerar.

- Duplicar os dados

  Uma das hipóteses é duplicar as tabelas em cada um dos microsserviços que a aplicação possui. Existem preocupações devido à duplicação e inconsistência de dados entre microsserviços, o que leva a que estes possam ter uma visão diferente destes dados [Newman (2019)].

Contudo, pode ser vantajoso se forem dados que muito dificilmente serão alterados [Newman (2015)]. Imagine-se ter que guardar os dados dos países que existem. Muito raramente existirá um novo país a ser criado e mesmo que aconteça vai provavelmente ser necessário apenas adicionar um novo registo a estes dados estáticos.

- Base de dados partilhada

  Outra hipótese seria ter uma base de dados partilhada responsável por estes dados estáticos.

  É, no entanto, necessário considerar todos os desafios de ter uma base de dados partilhada por vários microsserviços, algo abordado na secção 3.4.

  Esta opção tem a vantagem de evitar a duplicação de dados [Newman (2015)].

- Biblioteca estática

  Outra opção bastante mais simples seria guardar esta informação estática em ficheiros, tal como enumerações [Newman (2015)].

  De modo a não se ter que duplicar estas bibliotecas em todos os microsserviços, podia-se colocar estes dados numa biblioteca que possa ser acedida por todos. Contudo, o uso de uma biblioteca partilhada por todos os microsserviços, implicaria que não se pudesse desenvolver cada um em tecnologias diferentes e que se tivesse que voltar a instalar os microsserviços que a usam em caso de atualização desta biblioteca [Newman (2019)].

  Uma opção seria cada microsserviço ter a sua biblioteca estática, mas implica aceitar que nem todos precisam de ter a mesma versão desta biblioteca, porque caso precisem, voltamos outra vez à situação de terem todos que ser instalados aquando de uma alteração [Newman (2019)].

  Esta opção é vantajosa quando temos dados estáticos pequenos em volume e quando estes dados podem ter versões diferentes nos vários microsserviços.

- Microsserviço dedicado

  Uma última opção passa por colocar estes dados estáticos para um microsserviço independente e dedicado, sendo que os microsserviços que necessitem dessa informação a pedem a este [Newman (2015)].

  Com esta opção, está-se a adicionar mais um microsserviço que aumentará as chamadas feitas pela rede, levando a um aumento da latência. Uma solução seria não armazenar estes dados estáticos numa base de dados, mas sim em memória, o que depende do volume de dados estáticos, mas caso não seja muito grande, é uma solução viável e estando os dados em memória acaba por ser muito mais rápido do que ir buscar à base de dados [Newman (2019)].

  Por outro lado, caso estes dados não sejam facilmente mudados, cada microsserviço que necessite destes pode guardá-los na sua própria memória. Estes podem subscrever a eventos deste microsserviço dedicado para serem notificados quando existem novos dados ou alguma alteração [Newman (2019)].

Esta opção é vantajosa quando temos um valor enorme de conteúdo estático, onde ter um microsserviço que faça a gestão deste conteúdo valha realmente a pena. Se estes dados podem ser mudados com maior frequência é vantajoso ter apenas um local onde seja possível realizar estas alterações. É, no entanto, necessário ter em conta o custo associado com a criação de mais um microsserviço [Newman (2015)].

## 3.6 TRANSAÇÕES

Esta separação da base de dados leva a dificuldades aquando de operações que envolvam transações.

Normalmente depende-se muito da base de dados para assegurar a consistência e para poder executar várias operações em simultâneo, contando que estas são realizadas de forma atómica. Contudo, ao separar estes dados por vários microsserviços e bases de dados, perde-se a vantagem de realizar transações para realizar modificações de forma atómica [Newman (2019)]. Esta falta de atomicidade pode causar problemas especialmente se o sistema a migrar depender bastante desta propriedade.

Uma transação é um conjunto de operações que devem ser executadas juntas e que garante que ou todas elas foram executadas, ou que nenhuma delas foi executada, permitindo que o sistema continue num estado consistente, visto que caso alguma das operações falhe, as anteriores a elas são anuladas.

### 3.6.1 *Transações ACID*

ACID é um acrónimo que indica as propriedades-chave de transações da base de dados que levam a que um sistema seja confiável e que assegure a durabilidade e consistência dos dados [Newman (2019)]. ACID significa atomicidade, consistência, isolamento e durabilidade.

- Atomicidade

  Assegura que todas as operações de uma transação são realizadas na totalidade com sucesso ou nenhuma é. Caso alguma das mudanças a ser realizada falhar, toda a transação é abortada e revertem-se quaisquer mudanças realizadas durante a transação [Richardson (2018)].

- Consistência

  Quando alterações são realizadas à base de dados é assegurado que esta é deixada num estado válido e consistente. Os dados estão num estado consistente quando a transação começa e quando acaba [Richardson (2018)]. Por exemplo, se uma aplicação transferir dinheiro de uma conta para a outra, a propriedade de consistência assegura que o número total de dinheiro nas duas contas é o mesmo no início e no fim da transação.

- Isolamento

  Permite que várias transações sejam realizadas em simultâneo, sem que nenhuma interfira na outra. O estado intermediário de uma transação é invisível para outras transações. O resultado, de executar várias transações concorrentemente, é o mesmo caso elas fossem executas numa ordem arbitrária [Richardson

(2018)]. Por exemplo, na aplicação que transfere dinheiro de uma conta para outra, a propriedade de isolamento assegura que outra transação que esteja a ser efetuada, veja este dinheiro na primeira ou na segunda conta, nunca nas duas, nem em nenhuma.

- Durabilidade

  Assegura que após o término de uma transação, os dados permanecem na base de dados e não são perdidos, nem desfeitos, mesmo que exista alguma falha [Richardson (2018)].

É importante referir que nem todas as bases de dados asseguram transações ACID. As que não garantem as propriedades ACID são normalmente denominadas BASE que significa *Basically Available, Soft State and Eventual Consistency*, numa tradução livre "Basicamente disponível, estado delicado e eventual consistência"[Newman (2019)].

- Basicamente disponível

  Operações de escrita e leitura estão disponíveis tanto quanto possível, mas sem qualquer garantia de consistência, o que pode causar a que escritas não sejam persistidas devido a conflitos e que leituras não devolvam o último estado [Kleppmann (2017)].

- Estado delicado

  Sem a garantia de consistência, a probabilidade de saber o estado mais recente é diminuída, dado que as mudanças podem não ter sido convergidas.

  Indica que o estado do sistema pode mudar temporalmente, mesmo sem haver novas alterações realizadas, devido a não se garantir consistência [Kleppmann (2017)].

- Consistência Eventual

  Indica que o sistema ficará consistente eventualmente e se possam realizar leituras que representam o estado consistente do sistema, desde que, entretanto, não sejam efetuadas novas alterações Kleppmann (2017).

## 3.7 SAGAS

Uma saga é um algoritmo que consegue coordenar várias mudanças de estado, sem a necessidade de bloquear os recursos por um longo período de tempo, modulando os passos envolvidos nesta mudança de estado como atividades discretas que podem ser executadas independentemente [Newman (2019)].

Transações que são de longa duração, detém recursos da base de dados por períodos relativamente longos, acabando por atrasar outras transações mais curtas, o que causa problemas caso outros processos estejam a tentar aceder a estes recursos bloqueados para efetuar leituras ou escritas.

A ideia passa por partir estas transações numa sequência de transações mais pequenas, podendo ser cada uma tratada independentemente [Garcia-Molina and Salem (1987)]. Uma saga é definida como uma sequência

de transações locais, tendo cada transação local as propriedades ACID. A conclusão de uma transação local aciona a execução da próxima transação local [Richardson (2018)]. Com isto, a duração de cada transação será curta e vai apenas modificar uma parte dos dados. Como resultado, tem-se menos contenção de dados [Garcia-Molina and Salem (1987)].

Uma saga consiste em três tipos de transações locais.

- Transação compensável - Transações que podem ser potencialmente revertidas. Para isso, usa-se uma transação de compensação, algo a ser apresentado na secção 3.7.2.
- Transação pivô - Uma transação pivô é uma transação que define o sucesso ou não da saga. Se esta terminar com sucesso, é assegurado que a saga vai também terminar com sucesso. Esta pode ser a última transação compensável ou a primeira.
- Transação bem-sucedida - Transações após a transação pivô, garantidas de ter sucesso e que não precisam de transações de compensação.

Embora sagas tenham sido originalmente previstas como um mecanismo para ajudar a resolver os vários problemas das transações de longa duração sobre uma única base de dados, o modelo funciona perfeitamente para coordenar mudanças ao longo de vários microsserviços [Newman (2019)].

### 3.7.1 *Exemplo de uma Saga*

Considerando como exemplo uma aplicação de gestão de carros e encomendas de produtos para carros, imagine-se que um utilizador pretende realizar uma encomenda que contém produtos para o seu carro. É necessário verificar que existe inventário suficiente, retirar dinheiro relativo à encomenda do utilizador, retirar do inventário estes produtos e por fim submeter a encomenda, como se pode observar pela Figura 30.

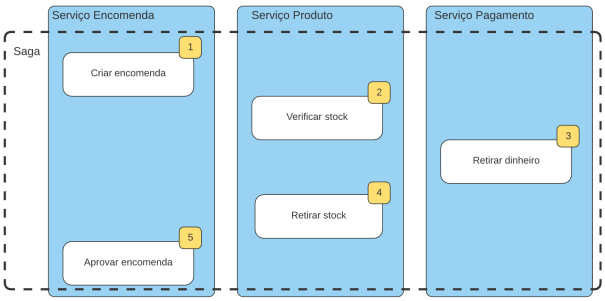



Figura 30: Exemplo de uma saga.

Cada participante da saga comunica de forma assíncrona, usando-se mecanismos de comunicação como *message broker* ou comunicação por eventos. Quando uma transação local termina, este microsserviço publica uma mensagem que desencadeia o próximo passo da saga. Desta forma garante-se que os participantes estejam pouco acoplados e que cada transação da saga é executada, mesmo que algum participante esteja temporariamente indisponível [Richardson (2018)].

### 3.7.2 *Lidar com Falhas*

Um dos desafios das sagas é recuperar de uma falha. Numa saga é necessário considerar que possa haver erros e que uma delas possa falhar [Richardson (2018)].

Existem duas formas de recuperação, para trás e para a frente. Uma recuperação para trás implica reverter a falha e tudo o que foi feito para trás. É preciso escrever transações de compensação para se poderem desfazer as alterações já realizadas pelas transações locais [Garcia-Molina and Salem (1987)]. Uma recuperação para a frente permite que se parta do ponto onde a falha ocorreu e continuar o processamento [Newman (2019)].

- Transações de compensação

  As transações que garantem as propriedades ACID conseguem facilmente retroceder caso algum problema aconteça, porque o retrocesso ocorre antes de se ter submetido as alterações, como se nada tivesse sido realizado [Newman (2019)].

  Com sagas é mais complicado e não é possível efetuar isto de forma automática, porque cada transação é local a um microsserviço e as modificações são submetidas para a sua base de dados. Para esta situação existem as transações de compensação [Richardson (2018)].

  Imagine-se que na Figura 30, a transação número quatro falha porque, entretanto, foi realizada outra encomenda com os mesmos produtos e deixou de existir inventário para esta encomenda. É preciso retroceder no que já se realizou, tal como ter retirado dinheiro ao utilizador pela encomenda.

  Para se retroceder nesta saga, é necessário a escrita de transações de compensação. Uma transação de compensação é uma operação que desfaz uma transação que foi já previamente submetida. Para retroceder neste exemplo de efetuar uma encomenda é preciso ter uma transação de compensação para cada uma que efetuou alterações [Newman (2019)].

  Uma saga executa as transações de compensação por ordem reversa das transações originais. Caso a transação $(n+1)$ falhar, é preciso que as $n$ transações anteriores sejam revertidas. Para cada uma destas transações $T_i$, existe uma transação de compensação $C_i$ que desfaz as mudanças da transação $T_i$. Para reverter os efeitos das $n$ transações, a saga tem que executar cada $C_i$ por ordem inversa, da $C_n$ para a $C_1$ [Richardson (2018)].

  É preciso sublinhar que estas transações de compensação não tem o mesmo efeito que um *rollback* de uma transação normal de base de dados. O *rollback* é executado antes de a transação ter submetido as mudanças para a base de dados, como se nada tivesse acontecido, enquanto qu nestas transações de compensação, é criada uma nova transação para reverter as mudanças realizadas pela transação original [Newman (2019)].

  Todavia, nem sempre é possível reverter totalmente uma transação [Newman (2019)]. Como um exemplo, um dos passos de realizar uma encomenda pode envolver o envio de um correio eletrónico que indique que a sua encomenda foi efetuada com sucesso. Caso se tenha que retroceder nas ações realizadas, não é possível desfazer o envio de um correio eletrónico. Nestas situações, a transação de compensação

pode ser o envio de um segundo correio eletrónico que indica que ouve um problema com a encomenda e que esta foi cancelada.

- Reordenar os passos da saga

  Uma reordenação dos passos envolventes numa saga pode significativamente diminuir os cenários de se ter que recuperar de uma falha [Newman (2019)]. Na Figura 30, a transação quatro pode ser reordenada e ser efetuada aquando da transação dois, como se observa na Figura 31. Desta forma, previne-se o possível erro de não conseguir retirar o inventário relativo à encomenda, porque já outra encomenda o retirou enquanto era executada a transação três de retirar o dinheiro.

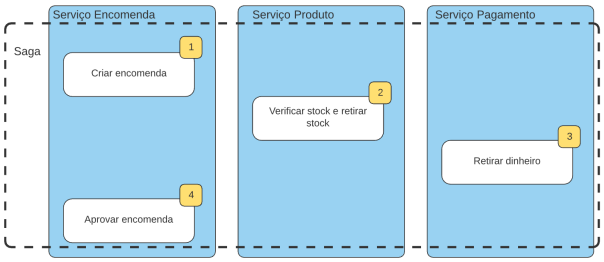



Figura 31: Ordenação dos passos de uma saga.

  Desta forma, evita-se ter que efetuar um retrocesso da transação três e simplifica-se em muito os cenários de recuperação [Newman (2019)]. Deixa de ser necessário escrever uma transação de compensação para a transação três.

- Conjugar recuperação para a frente e para trás

  Existem situações em que pode ser apropriado ter uma mistura dos tipos de recuperação de falhas, porque algumas falhas podem necessitar de ser recuperadas para trás e outras para a frente [Newman (2019)].

  Imagine-se que no processo de uma encomenda o último passo é o envio da encomenda. Se por alguma razão, ocorrer um problema no envio da encomenda, seja porque não existe espaço nas carrinhas de entrega ou por outra razão qualquer, não é correto reverter todo o processo da realização de uma encomenda. Neste caso, o melhor a fazer é voltar a tentar executar a ação de envio da encomenda.

### 3.7.3 *Implementação de Sagas*

Na implementação de uma saga é necessário ter em conta a coordenação dos passos desta. Quando uma saga inicia, é necessário selecionar o primeiro participante e indicar-lhe para executar a sua transação local. Quando este participante terminar a sua execução, a saga tem que selecionar o próximo participante e indicar para este executar a sua transação local e assim por adiante até a saga terminar todos os seus passos. Caso alguma transação local falhar, a saga deve executar as transações de compensação por ordem reversa [Richardson (2018)].

Para implementar esta lógica existem duas alternativas:

- Orquestração

  A lógica de coordenação é centralizada. Um orquestrador está encarregue de enviar as mensagens para cada participante de uma saga indicando que operações executar.

- Coreografia

  A decisão de coordenação e sequência está distribuída entre os participantes da saga.

### 3.7.4 *Orquestração*

Sagas que usam orquestração, possuem um coordenador central denominado orquestrador, que está encarregue de definir a ordem de execução, de indicar quando um microsserviço deve executar a sua transação local e de despoletar as transações de compensação caso necessário. Este controla o que acontece e quando conseguindo-se ter uma boa visibilidade do que acontece durante uma saga [Newman (2019)].

Em contraste com a saga apresentada na Figura 31, na Figura 32 é possível ver como se comporta uma saga que usa orquestração.

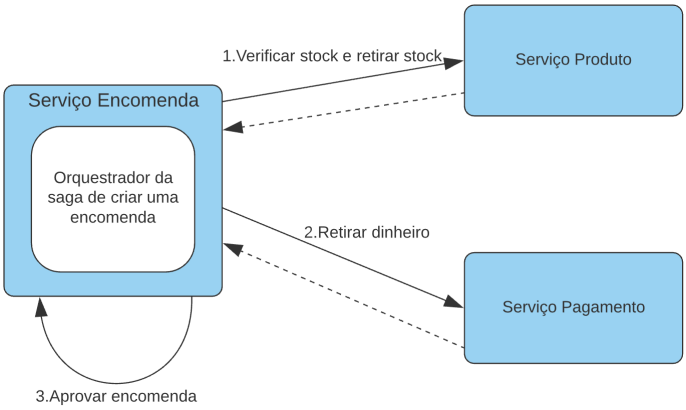



Figura 32: Saga usando orquestração.

Na Figura 32 pode-se observar o orquestrador desta saga de criar uma encomenda no microsserviço encomenda. Este sabe que microsserviços são necessários para que a saga possa ser realizada e decide que chamadas efetuar. Estes orquestradores tendem a ficar à espera da resposta do microsserviço que pediram para efetuar a sua chamada local, para saberem o que fazer a seguir, seja efetuar o próximo passo da saga ou então recuperar de uma falha.

De notar que na Figura 32, no último passo, o orquestrador envia uma mensagem ao próprio microsserviço onde se encontra. O orquestrador, visto que se encontra já no microsserviço encomenda, pode diretamente realizar este passo, porém para ser consistente, este trata o próprio microsserviço como qualquer participante [Richardson (2018)].

Uma saga orquestrada apresenta algumas vantagens. As dependências são mais simples, não existindo dependências cíclicas. O orquestrador invoca os participantes da saga e estes não invocam o orquestrador nem

sabem dos outros participantes, simplificando a lógica de negócio, dado que esta lógica de coordenação se encontra na totalidade no coordenador e não espalhada pelos participantes, como se verá na secção 3.7.5. A lógica de negócio é mais simples e não tem conhecimento das sagas em que participam [Richardson (2018)].

Esta alternativa possui, no entanto, algumas desvantagens. Com orquestração, os microsserviços vão estar mais acoplados, pois o orquestrador necessita de ter conhecimento dos que participam na saga. Outro problema é que como esta alternativa possui a lógica de coordenação centralizada, pode acontecer que exista lógica que devia estar nos microsserviços que acaba por ficar no orquestrador, levando a que estes tenham pouco comportamento e apenas recebam ordens do orquestrador [Newman (2019)].

### 3.7.5 *Coreografia*

Sagas que usam coreografia distribuem a responsabilidade e lógica de coordenação pelos participantes da saga [Newman (2019)]. Neste tipo de sagas, não existe nenhum coordenador central que diz o que cada participante da saga deve fazer. Cada participante subscreve eventos de outros participantes e reagem a estes [Richardson (2018)]. Na Figura 33 é possível ver um exemplo de como seria uma saga usando coreografia para a criação de uma encomenda.

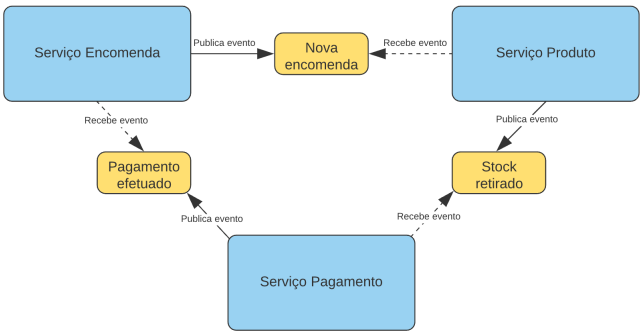



Figura 33: Saga usando coreografia.

É possível constatar na Figura 33 que os microsserviços reagem a eventos que recebem. Conceptualmente, estes eventos são publicados para todo o sistema, não apenas para um microsserviço em específico, sendo que os interessados subscrevem e os recebem [Newman (2019)].

Observando com detalhe para a Figura 33, é possível ver que o microsserviço encomenda publica um evento indicando que foi efetuada uma nova encomenda. O microsserviço produto que está subscrito a este evento, sabe que foi efetuada uma nova encomenda. Com isto, sabe que é preciso verificar se existe inventário para esta ser efetuada e subtraí-lo. Caso tudo corra com sucesso emite um novo evento no sistema indicando que este foi retirado e o processo continua. Contudo, caso não seja possível retirar, o microsserviço produto tem que emitir um evento que informe que não foi possível esta remoção e começa a recuperação desta falha.

Uma saga que usa coreografia apresenta vantagens como simplicidade e menos acoplamento entre microsserviços, porque os participantes publicam e subscrevem a eventos e não tem conhecimento direto uns dos outros [Richardson (2018)].

Também apresenta algumas desvantagens. É mais complicado de compreender o processo, porque ao contrário da orquestração não existe um único lugar onde a saga está definida. Com coreografia, a lógica de coordenação está espalhada pelos microsserviços, o que torna mais complicado a perceção do funcionamento desta. Uma segunda desvantagem é que pode criar dependências cíclicas entre microsserviços, pois cada participante subscreve aos eventos de outros participantes. Apesar de não ser necessariamente problemático, dependências cíclicas são um problema de arquitetura da aplicação [Richardson (2018)].

### 3.7.6 *Conjugar as duas implementações*

Apesar de sagas que usam orquestração e as que usam coreografia diferirem, é possível misturar estas duas implementações. Pode existir uma saga que funcione melhor com orquestração e outra que funciona melhor com coreografia. É possível até ter uma única saga que possui as duas implementações [Newman (2019)].

### 3.7.7 *Falta de isolamento*

Uns dos problemas das sagas é que estas são ACD (atomicidade, consistência, durabilidade), faltando-lhes a propriedade de isolamento. A propriedade de isolamento, como abordado na secção 3.6.1, assegura que o resultado, de executar várias transações concorrentes, é o mesmo caso estas fossem executadas numa ordem qualquer [Richardson (2018)].

A falta de isolamento nas sagas existe porque mal uma transação local de uma saga seja completada, outra saga qualquer pode visualizar estas modificações mesmo que esta não tenha terminado,o que pode causar dois problemas. Outras sagas podem acabar por modificar os dados acedidos por uma saga enquanto esta é executada e outras sagas podem realizar leituras de dados de uma saga que ainda não terminou a sua execução [Richardson (2018)].

Esta falta de isolamento pode causar uma anomalia, que pode levar a um funcionamento inesperado da aplicação. Uma anomalia é quando uma transação realiza leituras ou escritas de dados de maneira diferente da que acontecia caso fossem feitas uma de cada vez. Quando uma anomalia ocorre, o resultado de executar sagas concorrentemente difere caso estas fossem executadas por uma certa ordem [Richardson (2018)].

Existem três anomalias que podem ocorrer.

- *Lost updates* - Uma saga modifica dados sem ter visto as mudanças feitas por outra saga nestes dados.
- *Dirty reads* - Uma saga ou transação efetua leituras de dados modificados por uma saga que ainda não acabou esta atualização.
- *Fuzzy/nonrepeatable reads* - Dois passos diferentes de uma saga fazem leituras dos mesmos dados e obtém resultados diferentes porque, entretanto, outra saga realizou alterações a estes dados.

Como resultado desta falha de isolamento, a aplicação deve usar medidas, para prevenir ou reduzir o impacto de anomalias causadas por acessos concorrentes. Deve-se escrever sagas de modo a prevenir estas anomalias

ou minimizar o impacto destas. Algumas medidas implementam isolamento ao nível da aplicação, outras reduzem o risco da falta de isolamento [Richardson (2018)].

Existem algumas medidas, apresentadas a seguir, para colmatar esta falha de isolamento.

- Bloqueio semântico

  Quando se usa a medida de bloqueio semântico, as transações compensáveis colocam um identificador em qualquer registo criado ou atualizado. Este indica que o registo não acabou ainda de ser atualizado e pode vir a sê-lo. Este identificador pode ser um *lock* que previne que outras transações acedam a este registo ou um aviso que indica a outras transações que devem tratar este registo com alguma cautela. Este identificador é removido por uma transação bem-sucedida, visto que se tem garantias de que a saga terminará com sucesso ou então por uma transação de compensação, quando a saga está a recuperar de uma falha [Richardson (2018)].

  Além disso, é preciso decidir como outras sagas vão lidar com estes registos bloqueados.

  Uma opção pode passar por a saga falhar e informar o cliente para tentar de novo mais tarde. É de fácil implementação, mas acrescenta complexidade no cliente. Outra opção seria a saga ficar bloqueada até este identificador deixar de existir. Desta forma remove-se a complexidade no cliente de ter que voltar a tentar executar a operação relativa à saga. O problema é que a aplicação necessita de gerir estes identificadores e implementar um algoritmo de deteção de *deadlocks* que os resolva [Richardson (2018)].

- Atualizações comutativas

  Outra medida é desenvolver as operações de atualização para serem comutativas, significando que podem ser executadas numa ordem qualquer. Esta medida é útil porque elimina as anomalias de *lost updates* [Richardson (2018)].

- Visão pessimista

  Na medida de visão pessimista, os passos de uma saga são reordenados para minimizar o risco de acontecerem *dirty reads* [Richardson (2018)]. É semelhante ao que se constatou na secção 3.7.2 sobre a reordenação de passos de uma saga.

- Reler o valor

  Esta medida de reler o valor previne a anomalia de *lost updates*. Uma saga que use esta medida efetua uma nova leitura de um valor antes de o atualizar, verificando que não foi, entretanto, modificado. Caso tenha sido modificado, a saga é abortada e executada novamente [Richardson (2018)].

- Registar versões

  A medida de registar versões, regista as operações realizadas a um registo, para poderem ser reorganizadas. É uma forma de tornar operações que são não comutativas em comutativas. Desta forma, os pedidos realizados para alterar este registo são ordenados por ordem de chegada e são depois executadas pela ordem correta [Richardson (2018)].

## 3.8 PADRÕES MIGRAÇÃO - BASE DE DADOS

Os padrões mencionados na secção 3.3 abordam várias estratégias para extrair funcionalidade para microsserviços. Contudo, é preciso pensar também na base de dados.

Vão ser abordados alguns padrões arquiteturais relacionados com a base de dados.

### 3.8.1 *Monolítico com agregados expostos*

Os padrões mencionados anteriormente na secção 3.4 não procedem a nenhuma separação do esquema da base de dados, apenas apresentam várias formas de vários microsserviços acederem a uma mesma base de dados partilhada.

Se se considerar um microsserviço como um encapsulamento lógico associado a um ou mais agregados, como mencionado na secção 2.2.1, é preciso considerar também que este deve possuir o código relacionado com as mudanças de estado destes agregados, bem como os dados deste que devem estar na sua própria base de dados. Contudo, se o nosso microsserviço extraído necessitar de interagir com um agregado que ainda se encontra no monolítico, é preciso que este seja exposto para poder ser acedido [Newman (2019)].

Para isso existe um padrão que permite que o monolítico possua agregados que expõem uma API para o exterior do monolítico.

Como se observa na Figura 34, o microsserviço *B* foi extraído do monolítico. Todavia, necessita de dados do agregado *A* que se encontra no monolítico. Neste caso, através de uma API, ou outro mecanismo, o agregado *A* expõem uma API que permite ao microsserviço *B* aceder à informação que precisa [Newman (2019)].

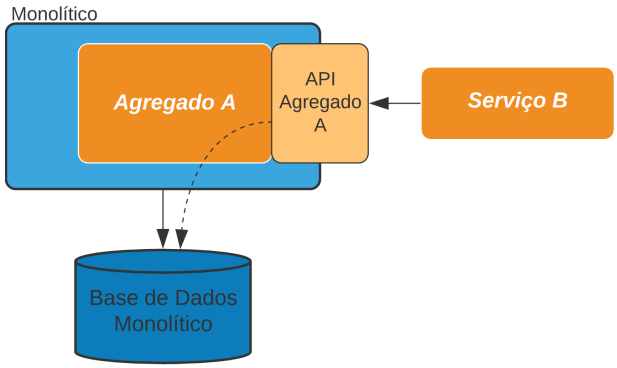



Figura 34: Monolítico com agregados expostos.

Desta forma está-se a permitir que microsserviços exteriores ao monolítico possam efetuar leituras dos dados daquele agregado e até alterar ou inserir. É possível, no entanto, restringir a informação exposta e as ações que se podem realizar [Newman (2019)].

Um ponto interessante deste padrão, é que ao se expor estas APIs do monolítico para microsserviços exteriores, está-se potencialmente a descobrir um futuro microsserviço. Na Figura 34 temos o agregado *A* a expor uma API para que o microsserviço *B* possa aceder a dados deste. Pode-se concluir que provavelmente o

melhor a fazer é extrair posteriormente este agregado *A* para um microsserviço. É evidente que se ao extrair este agregado *A* do monolítico, alguma parte do monolítico precisar de dados deste, este terá que novamente ser mudado para usar este novo microsserviço [Newman (2019)].

Este padrão é vantajoso para situações em que novos microsserviços necessitam de dados que ainda se encontram na base de dados do monolítico. Ao extrair um microsserviço do monolítico, ter este a realizar chamadas ao monolítico leva a um maior trabalho e complexidade do que se fosse diretamente à base de dados deste, mas no futuro prova-se benéfico e vantajoso, pela razão de que se realiza chamadas a um agregado que no futuro pode vir a se tornar num microsserviço que possuirá a sua própria base de dados [Newman (2019)].

### 3.8.2 *Mudança de domínio dos dados*

No padrão anterior, falou-se como abordar quando um novo microsserviço precisa de aceder a dados que o monolítico possui. Mesmo assim, existem casos em que temos dados que estão no monolítico que certamente deviam estar no novo microsserviço extraído, como se mostra na Figura 35 no passo 1.

Nestes casos, é preciso transferir os dados deste novo microsserviço para que deixem de ser do domínio do monolítico e sejam dele, visto que é lá que estes devem estar e serem geridos. É necessário mudanças no monolítico para passar agora a tratar o novo microsserviço como quem possui estes dados [Newman (2019)]. Quando o monolítico necessitar de realizar leituras ou alterações destes, deve agora interagir com o microsserviço, como se pode observar na Figura 35 no passo 2.

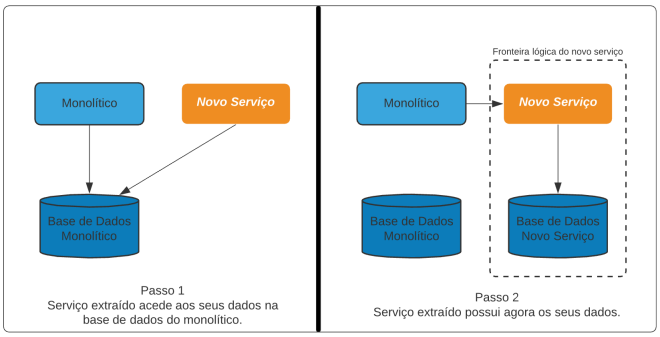



Figura 35: Mudança de domínio dos dados.

Separar estes dados do já existente esquema de base de dados pode ser algo bastante complexo. É preciso considerar a remoção de chaves estrangeiras, o uso de transações, entre outros, algo já mencionado na secção 3.5.

Este padrão deve ser utilizado quando um novo microsserviço extraído realiza alterações a dados que lhe deviam pertencer e que este não os possui. Neste caso, os dados devem ser movidos para o novo microsserviço criado, sendo este responsável pela sua manutenção e gestão [Newman (2019)].

### 3.8.3 *Sincronização dos dados*

Muitos dos padrões de migração abordados na secção 3.3 possuem a vantagem de ser possível retroceder para a implementação antiga após se ter mudado para a nova. Ainda assim, existe o problema de quando os dados que o novo microsserviço gere terem que estar sincronizados com os que estão no monolítico [Newman (2019)].

Na Figura 36 temos um exemplo desta situação onde se extrai o microsserviço *A* do monolítico. Todavia, tanto o novo microsserviço, como o monolítico, gerem os mesmos dados. Esta situação pode acontecer quando se pretende que seja possível um retrocesso para a implementação antiga. É preciso então que estes dados estejam sincronizados e também considerar o quão importante é ter os dados consistentes entre eles [Newman (2019)].

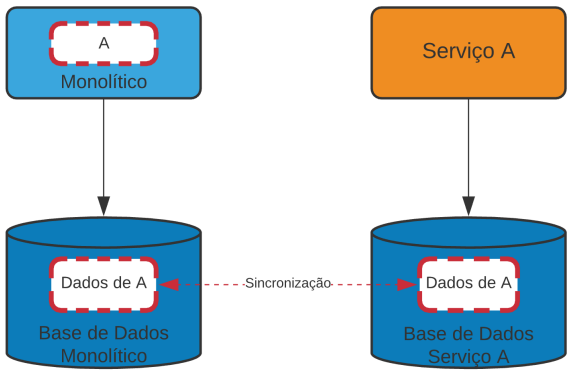



Figura 36: Sincronização dos dados.

Caso estes devam estar totalmente consistentes e sincronizados, o melhor é manter os dados num único lugar, tendo provavelmente o novo microsserviço a aceder diretamente à base de dados do monolítico, podendo recorrer ao padrão de base de dados com vistas, mencionado na secção 3.4.2. Quando se pretender proceder à mudança para o novo microsserviço, pode-se usar o padrão de mudança de domínio dos dados, mencionado anteriormente na secção 3.8.2.

Outra possibilidade seria considerar ter estas duas bases de dados em sincronia, como se tem na Figura 36. Leva a maior complexidade, pois tanto o monolítico, como o microsserviço, podem realizar escritas sobre os seus dados e terem ambos que estar sincronizados [Newman (2019)].

### 3.8.4 *Sincronização dos dados na aplicação*

Transferir dados de um lugar para outro é um processo bastante delicado, dependendo do quão valiosos são [Newman (2019)]. Quando se tratam de transferir dados relativos a contas bancárias ou registos médicos, por exemplo, é importante pensar com cuidado como se procederá a esta transferência.

Este processo pode realizar-se nos seguintes passos:

- **Cópia total de dados:**

  O primeiro passo consiste em realizar uma cópia total dos dados da base de dados para a nova.

Se for possível parar a aplicação, a cópia dos dados é mais fácil e direta. Caso não seja possível parar a aplicação é preciso realizar um *snapshot* do estado da base de dados e aplicá-lo na nova base de dados. Cria um desafio, pois enquanto foi realizada esta cópia, podem ter sido introduzidos novos dados na base de dados que não estavam no *snapshot*. É necessário registar as mudanças realizadas enquanto foi efetuada a cópia, para poderem ser aplicadas após esta [Newman (2019)]

- **Sincronizar escritas em ambas as bases de dados:**

  Estando agora as duas bases de dados em sincronia, é necessário assegurar que a aplicação escreva corretamente em ambas as bases de dados [Newman (2019)], como se mostra na Figura 37.

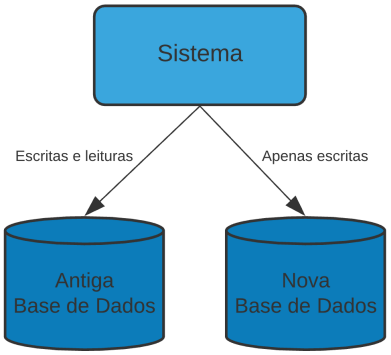



Figura 37: Sincronizar escritas em ambas as bases de dados.

- **Redirecionar escritas e leituras:**

  Encontrando-se a nova base de dados num correto funcionamento, pode-se alterar a aplicação para considerar a nova base de dados como aquela para onde devem ser realizadas as escritas e leituras. É evidente que as bases de dados devem continuar em sincronia, que continuem a realizar as escritas em ambas as bases de dados para que caso ocorra algum problema se tenha uma reserva [Newman (2019)].

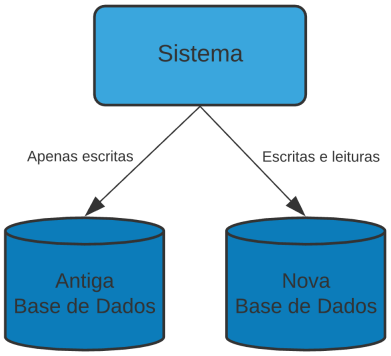



Figura 38: Redirecionar escritas e leituras.

Este padrão é vantajoso quando se considera separar primeiro o esquema da base de dados antes de separar o código [Newman (2019)].

### 3.8.5 *Rastrear escritas*

Este padrão de rastreio de escritas é uma variação do padrão apresentado na secção 3.8.4. Com este padrão, a base de dados principal é movida de forma incremental, permitindo que existam duas durante a migração [Newman (2019)].

Neste padrão, o novo microsserviço possui os mesmos dados que o monolítico. O monolítico continua a registar e a manter os novos dados localmente, mas quando existem mudanças assegura que estes novos dados são escritos no novo microsserviço através da API do microsserviço, como se pode observar pela Figura 39. É necessário efetuar mudanças ao código do monolítico para que aceda ao novo microsserviço para realizar esta sincronização de dados. Quando toda a funcionalidade estiver a usar o novo microsserviço como quem possui estes dados, é possível removê-los do monolítico [Newman (2019)].

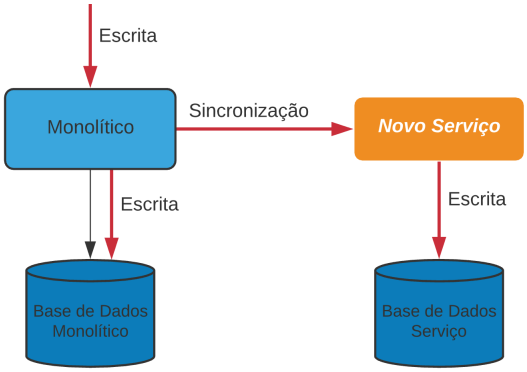



Figura 39: Rastrear escritas.

Este padrão permite que se proceda a uma migração incremental, reduzindo o impacto de cada mudança realizada. Este permite que se comece por manter apenas um pequeno conjunto de dados sincronizados e incrementar este conjunto temporalmente, enquanto se altera e se incrementa gradualmente as ligações a este novo microsserviço [Newman (2019)], como está exemplificado na Figura 40.

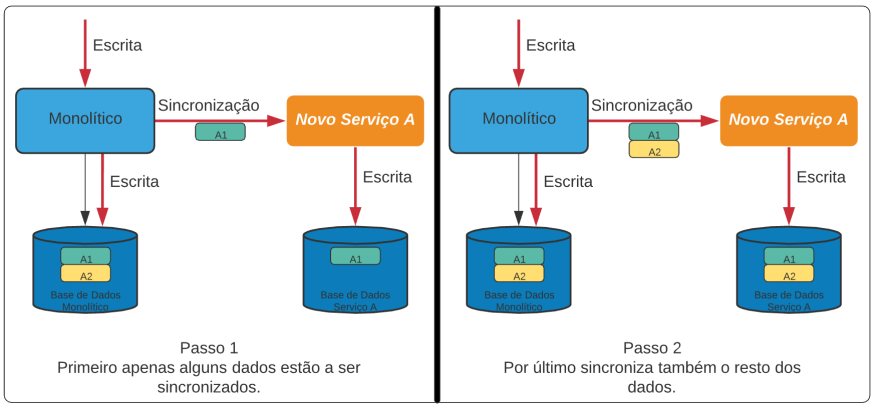



Figura 40: Sincronizar incrementalmente os dados do monolítico para o novo microsserviço.

Outros microsserviços que necessitem destes dados, podem ir buscar tanto ao monolítico como a este novo microsserviço, dependendo da informação que precisam. Se precisam de informação que está ainda apenas no monolítico, tem que ir obrigatoriamente a este, podendo apenas ir ao novo microsserviço quando este estiver a sincronizar os dados que estes pretendem [Newman (2019)].

Um problema que é preciso ter em conta neste padrão e em outras situações de dados duplicados é a inconsistência destes. Existem algumas hipóteses como:

- Realizar escritas para uma das bases de dados

  Todas as escritas são realizadas para uma das bases de dados. Os dados são sincronizados com a outra quando as escritas terminarem [Newman (2019)].

- Enviar as escritas para ambas as bases de dados

  Todos os pedidos de escrita realizados são enviados para ambas as bases de dados. Para isto, quem realiza os pedidos envia para ambas ou existe um intermediário que trata do envio [Newman (2019)].

- Enviar escritas para qualquer base de dados

  Os pedidos de escrita são enviados para qualquer base de dados, sendo que depois estas tratam de sincronizar os dados entre os sistemas [Newman (2019)].

  Esta hipótese traz mais complexidade, pois é preciso implementar sincronização nas duas bases de dados e não apenas numa.

Em todas as hipóteses apresentadas acima irá existir algum atraso na consistência dos dados, levando assim a uma eventual consistência, isto é, eventualmente os dados ficarão consistentes [Newman (2019)].

## 3.9 DEPENDÊNCIAS ENTRE COMPONENTES

Até agora, falou-se sobre como extrair microsserviços do monolítico e como lidar com a base de dados, porém é preciso saber por onde começar e quais dos microsserviços devem ser os primeiros a serem extraídos. Quando se tiver o código e o código de acesso à base de dados organizado e separado para cada componente, deve-se começar a pensar quais módulos extrair primeiro para microsserviços. Como mencionado anteriormente na secção 3.1, sugere-se seguir uma migração incremental de modo a diminuir o número de problemas e erros possíveis.

Usando como exemplo uma aplicação de gestão de carros e encomendas de produtos para carros, podemos ter vários componentes, representados por *packages* na Figura 41. Componentes para gerir os utilizadores, os carros, as encomendas realizadas, os produtos que existem e respetivo inventário e um que gere as notificações que tem que ser apresentadas aos utilizadores, tal como quando tem que efetuar uma mudança de óleo no carro ou levar o carro à inspeção.

Desenhando um diagrama de *packages*, apresentado na Figura 41 é possível obter uma visão da estrutura do sistema, que permite concluir que módulos são mais fáceis e mais difíceis de extrair. Se se pretender começar

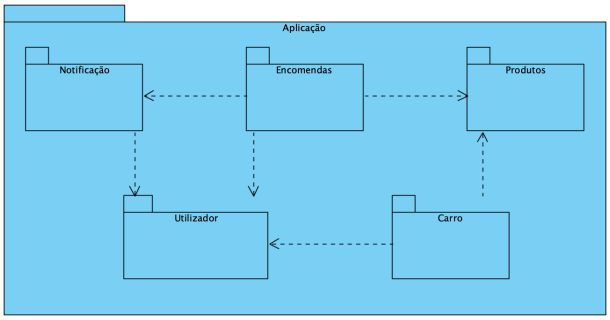



Figura 41: Dependências entre componentes.

por extrair o componente utilizador pode-se perceber que este recebe várias chamadas de outros componentes, o que implica que várias partes dos outros componentes teriam que ser modificadas. Seria preciso modificar bastante código de todos os outros componentes que realizam chamadas locais a este e que teriam que passar a ser chamadas remotas.

É mais fácil começar por extrair aqueles componentes que contém menos dependências, de modo a modificar o mínimo de outros componentes no início da separação para microsserviços [Newman (2019)].

Como se pode constatar na Figura 41, o componente do carro não possui outros componentes dependentes dele, o que o torna num componente mais fácil de extrair para um microsserviço. Desta forma reduz-se a quantidade de mudanças a ter que realizar noutros módulos [Newman (2019)].