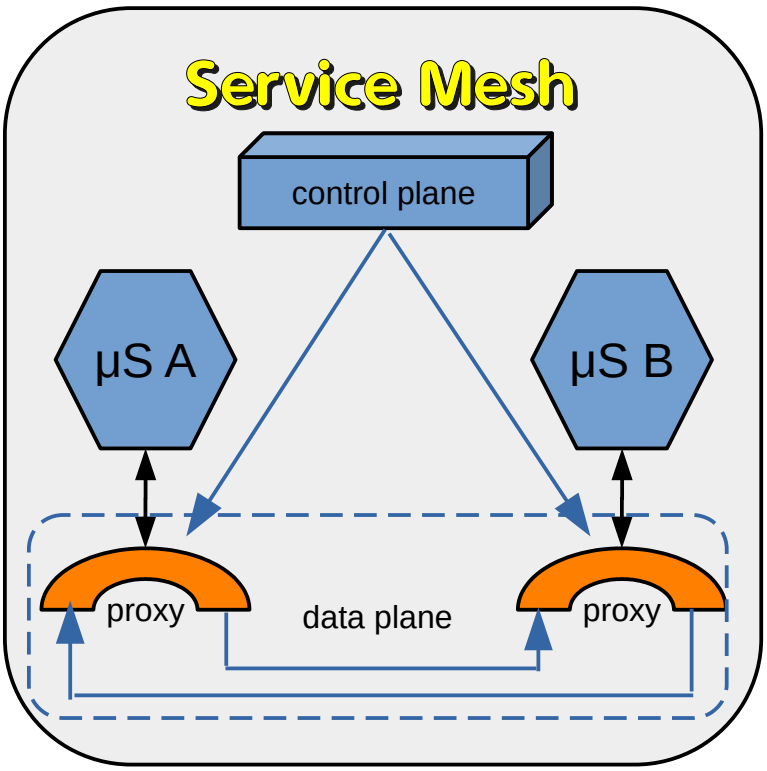


O que é Service Mesh?
Cleuton Sampaio

Na arquitetura de software, um **Service Mesh** (“malha de serviço”) é uma camada de infraestrutura dedicada para facilitar as comunicações serviço a serviço entre serviços ou microsserviços, usando um proxy. (Wikipedia em Inglês).

Vemos esse termo aparecer muito na mídia nos últimos anos, infelizmente sempre associado a determinados produtos. Mas o padrão arquitetural de Service Mesh existe independentemente de produtos.

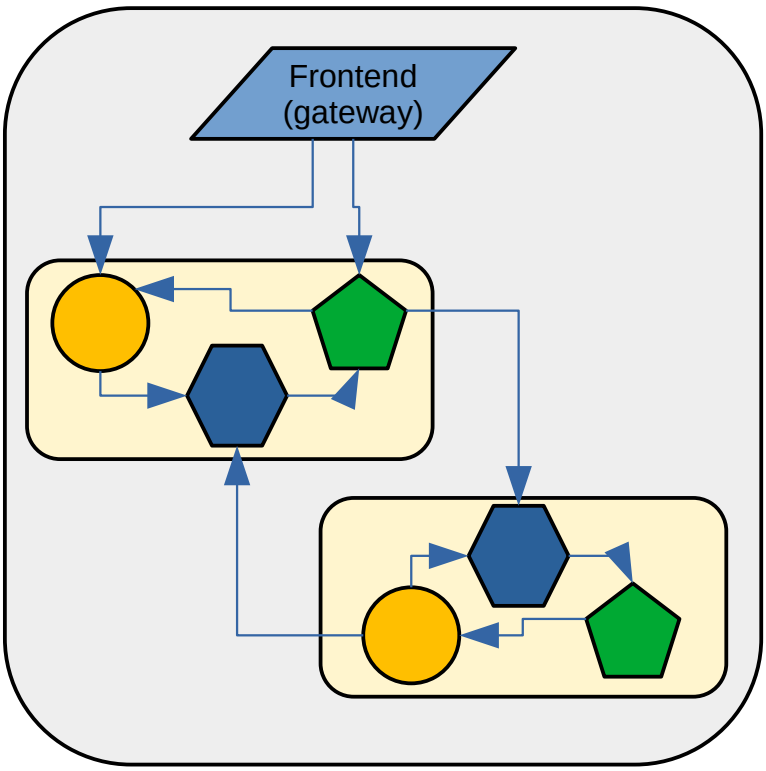
As **Big techs** (Netflix, Twitter, Facebook, Google etc) já estavam pensando neste problema há algum tempo e criaram algumas soluções para lidar com a complexidade da comunicação entre serviços.

Com a adoção do padrão arquitetural de microsserviços, alguns complexidades passaram a serem embutidas neles, por exemplo: Controle de acesso, resistência a falhas, balanceamento de carga, service discovery, monitoração e várias outras. Isso complica muito o código-fonte dos serviços.

Foram criadas várias bibliotecas e frameworks, como: Eureka, Spring cloud, Hystrix e outras, para facilitar e gerenciar estas complexidades e todas servem como alternativas para implementar um Service Mesh.

O propósito de adotar Service Mesh é desacoplar o código funcional do código de gestão de infraestrutura de comunicação, deixando os serviços mais simples e permitindo gerenciamento externo da sua malha de microsserviços.

É isso o que você precisa entender sobre o assunto.



No hay problema

Vai por mim: A vida é bem melhor quando temos um sistema convencional, com alguns monólitos se comunicando diretamente. Tudo é mais simples e podemos gerenciar sem problemas. Você tem um **frontend**, um **backend** que serve de **gateway**, e alguns monólitos que expõem serviços.

Pode até ser que você implemente algum tipo de escalabilidade nos seus serviços ou mesmo nos seus monólitos. Por exemplo, utilizando um pool de threads (ou processos), ou mesmo um balanceador de carga. Para isso, a única coisa necessária é que o serviço (ou monólito) a ser balanceado seja **stateless**.

A segurança pode ser apenas na "**fronteira**" e uma vez autenticado, o controle de acesso pode ser feito apenas entre os monólitos, mas não é necessário em chamadas internas.

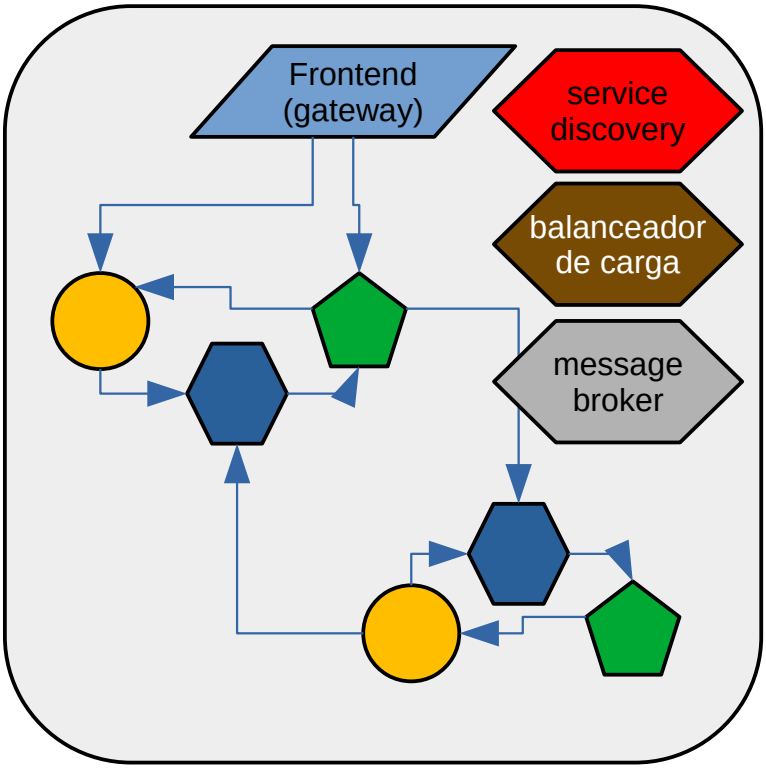
Caiu um monólito? Sem problemas! É fácil perceber e reiniciar o componente. Monitoração? Basta redirecionar os logs, seja para uma suíte **ELK** (ElasticSearch, Logstash e Kibana) ou apenas para um **graylog** server e analisar.

Não! Não estou debochando! De forma alguma! Se você já leu meus artigos, sabe que eu tendo a ser agnóstico tecnologicamente, e também pragmático: A solução boa é a que funciona para o seu problema. Não acredito em soluções miraculosas e belos artigos.

A granularidade dos componentes é grossa (não gosto de usar "alta" e "baixa" porque confunde). Ou seja são poucos componentes com muitas responsabilidades.

Portanto, há menos canais de comunicação a gerenciar e monitorar.

Por que você mudaria isso? Para afinar a granularidade dos seus componentes! E por que? Para diminuir custo, risco e prazo de manutenções. Portanto, você partiria para uma arquitetura de microsserviços.



Agora, que retiramos o “encapsulamento” em **monólitos**, cada **µS** (microsserviço) tem apenas uma responsabilidade, necessitando se comunicar com outros para poder complementar o que foi pedido. Você pode ter alguns serviços façade para facilitar o acesso à sua API, formada por miríades de serviços de granularidade fina.

Os **µSs** (microsserviços) são componentes isolados, portanto, para um deles invocar outro provavelmente você utilizará algum tipo de chamada **interprocessos** (talvez entre hosts diferentes). Se você simplesmente importar o serviço “B” dentro do “A” e fizer uma chamada **inprocess**, você estará acoplando os dois.

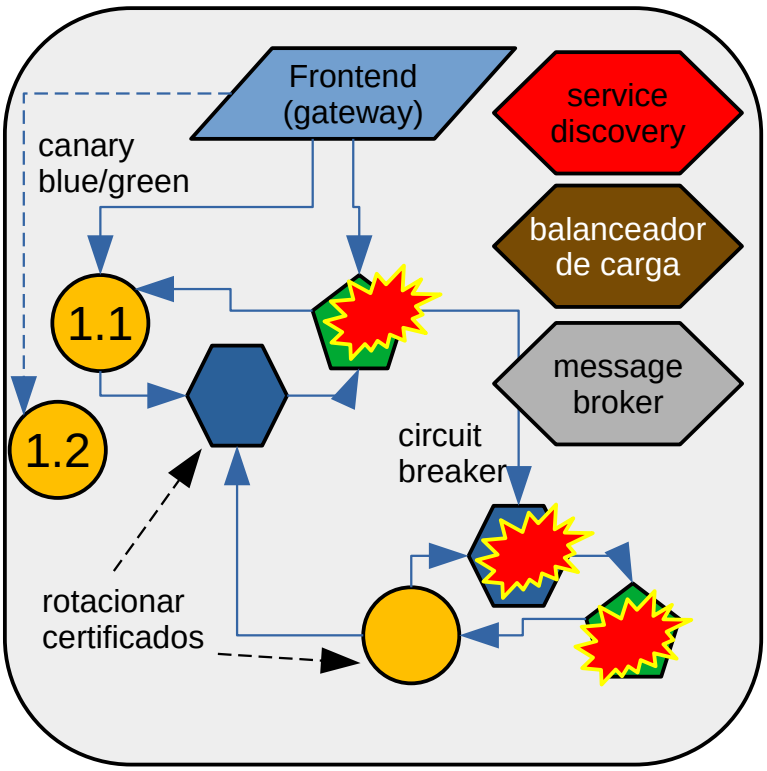
E tem mais! Imagine que o serviço “B” seja demorado, você deixaria o serviço “A” esperando? Não. Então você pode lançar mão de criar várias instâncias do serviço “B” e distribuir a carga, certo? Pode usar um **balanceador de carga** para isso.

Para informar o IP ou o nome do serviço “B” ao serviço “A”, você pode utilizar variáveis de ambiente… Ou então, utilizar um componente de **service discovery**, que desacoplaria de vez os dois serviços.

Uma alternativa seria utilizar mensagens assíncronas entre os serviços “A” e “B”, através de um **message broker.** Desta forma, você poderia ter várias instâncias do serviço “B” escutando uma fila.

Já notou que acrescentamos mais 3 componentes à sua infraestrutura? O que isso significa? Mais 3 coisas para gerenciar, mas vários canais de comunicação diferentes, mais **SPOFs** (Single Point of Failure – Ponto único de falha).

Se você tem muitos µSs, gerenciar isso tudo pode ser um grande problema. Mas vamos botar mais “sal” na ferida: Imagine que alguns serviços passaram a utilizar protocolos diferentes, como gRPC, por exemplo…



Mamma mia!

A vida é cruel! E se um serviço apresentar erro? Como você vai gerenciar isso? Vai implementar um mecanismo de retry? Uma falha em um serviço pode ocasionar falha em cascata não gerenciável. Você precisa de algum tipo de **circuit breaker** para represar isso. Ele ajuda a isolar falhas, interceptando os requests e implementando controle de comunicação com os serviços.

O circuit breaker ajuda a controlar a propagação de falhas, permitindo aos serviços re-tentar as chamadas posteriormente ou retornar algum tipo de mensagem que possa ser tratada pela aplicação.

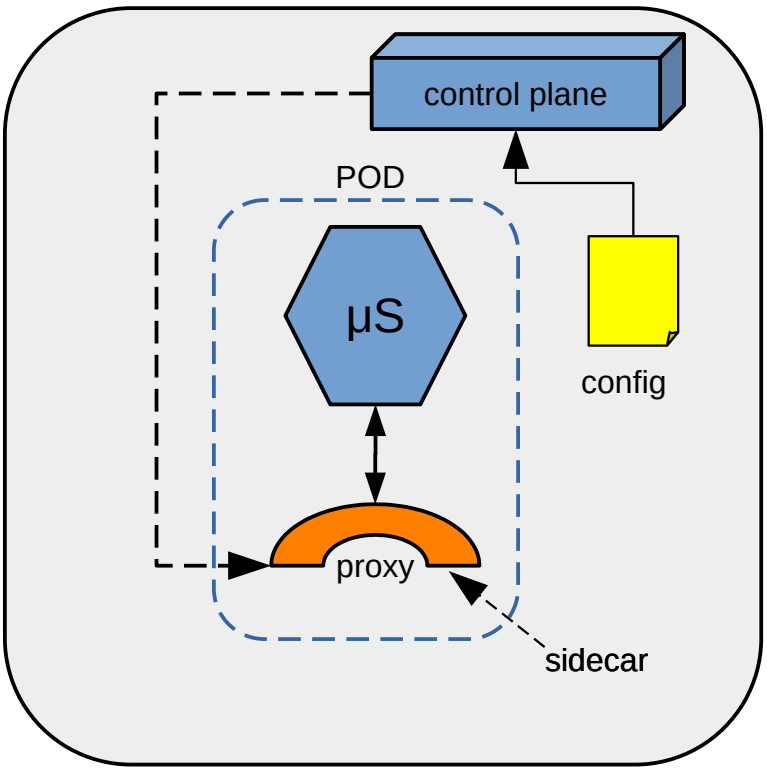
Outro problema é a segurança dos dados em trânsito. Ao partirmos um monólito em um “zilhão” de µSs, como implementaremos essa segurança? Precisaremos emitir e **gerenciar certificados** para cada serviço, de modo a utilizar **TLS**. Expiração e rotação de certificados são uma prática comum para implementar segurança em ambientes distribuídos.

E mais um problema surge ao experimentarmos novas práticas de **deploy** de serviços, como: **blue/green**, na qual subimos a versão nova junto com a antiga e, depois de testar viramos a chave, ou **canary**, na qual direcionamos uma parcela dos clientes (10%, 15% etc) para a nova versão.

Se você não utiliza nenhum recurso de Service Mesh, terá que implementar essas funcionalidades no código (funcional ou de infraestrutura), gerando mais problemas a serem administrados.

Em uma conversa rápida, levantamos 6 novos componentes que você deveria adicionar à sua infraestrutura para suportar uma grande variedade de µSs, porém, o problema é muito maior do que isso.

Nem falamos sobre gestão e documentação de API, e sequer vimos em detalhes as necessidades de monitoramento…



Em um padrão de **Service Mesh**, delegamos todas essas funções de infraestrutura de comunicação dos serviços a um componente separado, como um **proxy**, que intercepta todas as comunicações do **µS** direcionando-as de acordo com as regras configuradas.

Podemos automatizar essa configuração através de um componente control plane, que lê as configurações de maneira centralizada e as distribui entre os proxies de cada µS.

Geralmente, as bibliotecas e frameworks de Service Mesh provêm serviços de: Segurança, monitoração, tratamento de erros, balanceamento de carga, service discovery e muitos outros.

Um dos ambientes ideais para implementar isso é o **Kubernetes** (**k8s**), devido às suas características especiais de abstração. Em um **cluster k8s** podemos ter o **proxy** carregado no mesmo **pod** que o µS utilizando o padrão **sidecar**, no qual um segundo contêiner é executado junto com o contêiner principal, no mesmo pod, estendendo a funcionalidade deste.

Todo o código que lida com problemas de comunicação, discovery, tolerância a falhas, balanceamento de carga e outros, é movido do µS para o proxy sidecar, diminuindo a complexidade e utilizando código já testado.

É possível implementar Service Mesh utilizando vários tipos de software. O mais popular é o **Istio**, em conjunto com o proxy **Envoy,** mas há várias soluções, desde implementar tudo manualmente com bibliotecas (**Eureka**, **Spring cloud**, **Hystrix** e outras), como soluções completas (**Istio**, **Linkerd**, **Kuma** e o **Open Service Mesh**).

O importante é você entender o que é Service Mesh e qual problema ele resolve, antes de partir seguindo dicas de marketing.