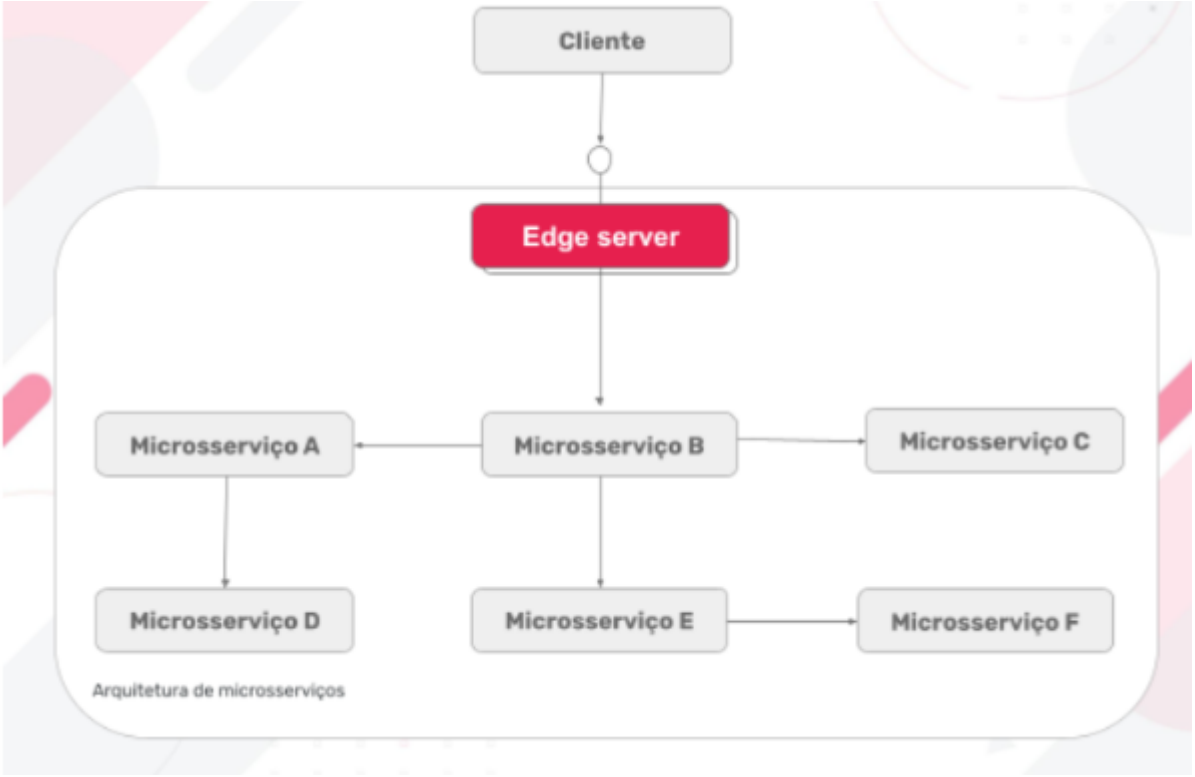
Edge server

Em muitos casos, em uma arquitetura de microsserviços, é necessário expor alguns microsserviços fora do contexto do nosso sistema e ocultar outros do acesso externo. Esses microsserviços expostos ao exterior devem ser protegidos contra solicitações de clientes mal-intencionados. É aqui que o componente do Edge server desempenha um papel fundamental, por onde passarão todas as solicitações externas.



Normalmente, um componente como este se comporta como se fosse um reverse proxy (proxy reverso) e pode ser integrado à descoberta de serviço para fornecer capacidade de balanceamento de carga dinâmico.

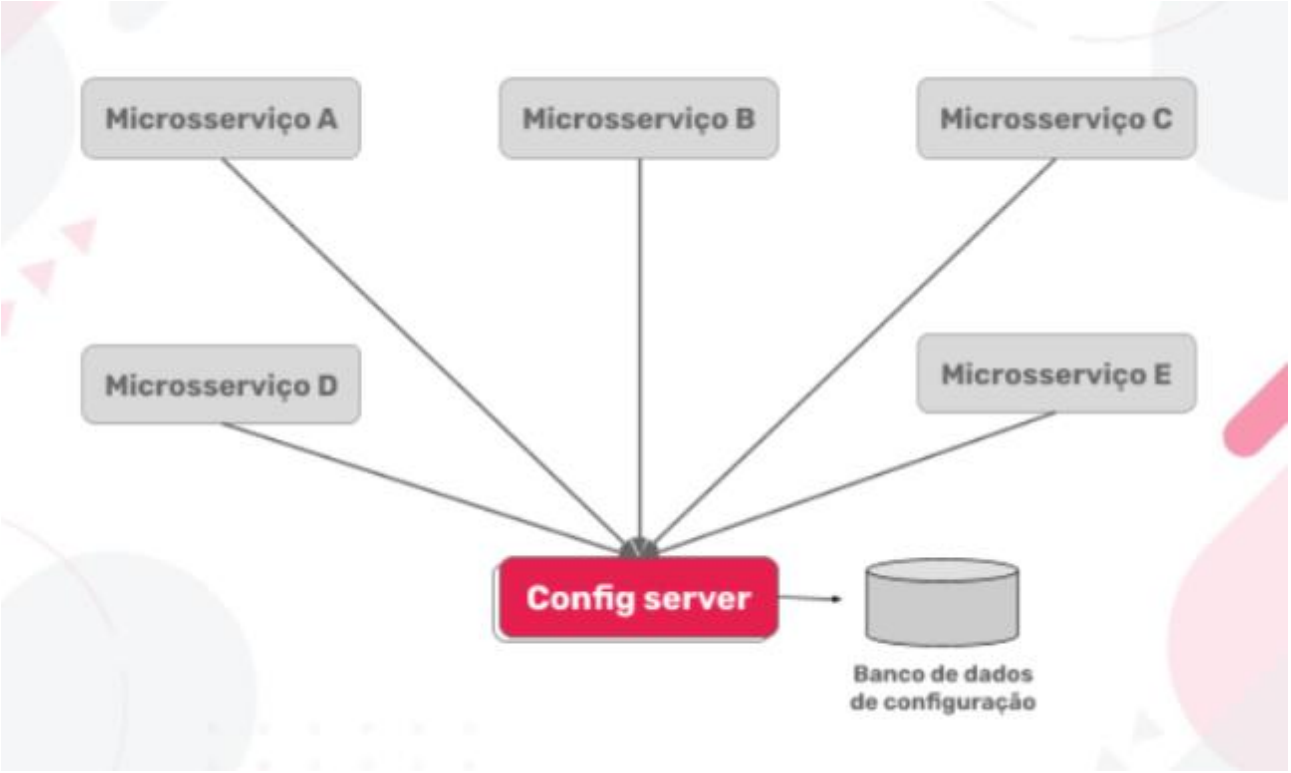
Portanto, para proteger contra solicitações maliciosas, podemos usar protocolos padrão e práticas recomendadas, como OAuth, OIDC, JWT y API keys, para você saber que eles vêm de clientes confiáveis. Da mesma forma, os microsserviços que precisam ter essa "camada de invisibilidade" podem se tornar "visíveis" para aqueles que requeiram seu serviço, tendo as rotas de acesso a eles configuradas em tempo hábil.

Central configuration

Normalmente, uma aplicação é implantada com sua configuração, onde podemos encontrar um conjunto de variáveis ??de ambiente e/ou arquivos que contêm informações de configuração. Diante de uma arquitetura baseada em microsserviços, ou seja, com um grande número de instâncias de microsserviços implantadas, surgem as seguintes questões:

- Como obtenho uma visão completa da configuração que existe para todas as instâncias de microsserviço em execução?
- Como atualizaremos a configuração e garantir que todas as instâncias de microsserviço afetadas adotem a mesma?

O padrão propõe uma resposta para isso adicionando um novo componente chamado configuration server (servidor de configuração), onde armazena a configuração de todos os microsserviços, como podemos observar no diagrama a seguir:

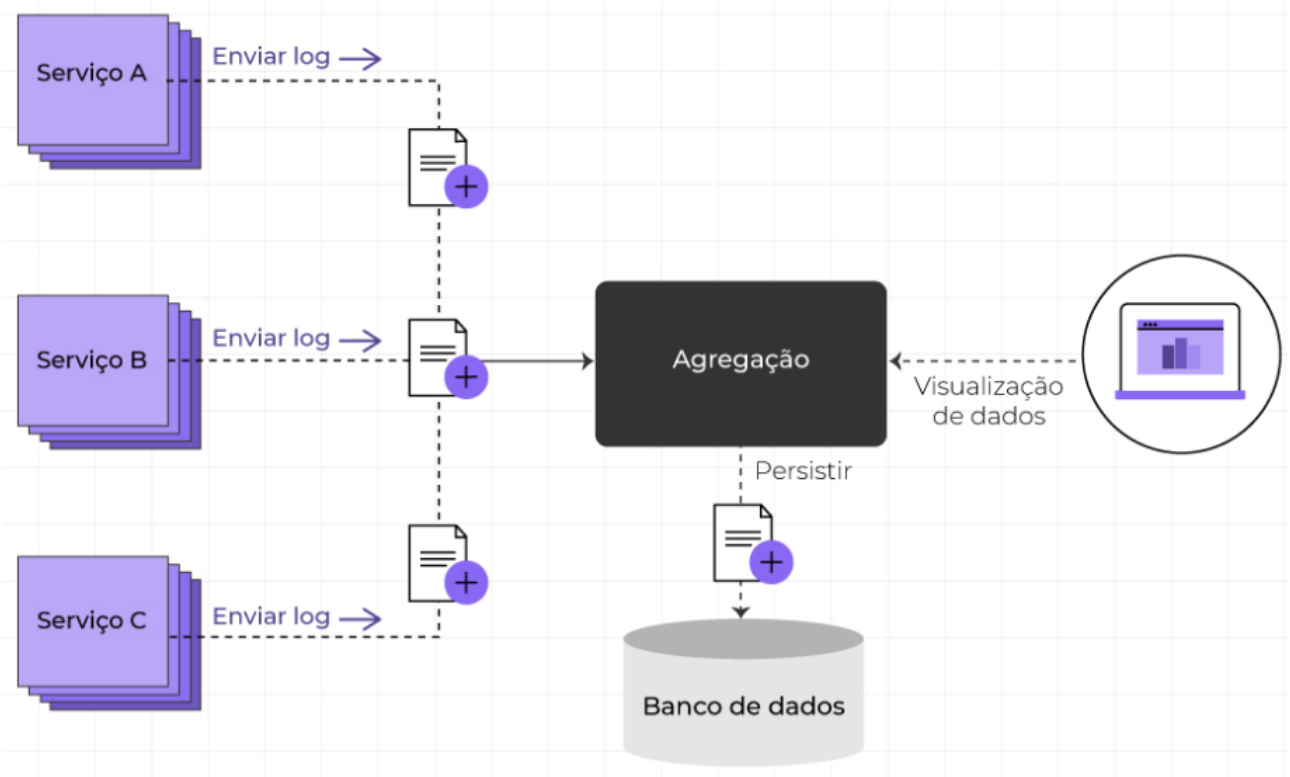


Ao mesmo tempo, isso permite centralizar todas as informações em um só lugar, mesmo com configurações diferentes dependendo do ambiente (desenvolvimento, testing, QA, produção, etc).

Log aggregation

Quando falamos em arquitetura distribuída, um dos problemas mais frequentes é obter rastreabilidade da execução de um serviço, pois nessa categoria de arquitetura a execução de um serviço passa por vários serviços, o que dificulta entender o que está ocorrendo e ter um registro detalhado do que está acontecendo. Especificamente, no caso de arquiteturas como microsserviços, espera-se ter várias instâncias do mesmo componente, o que torna ainda mais tedioso recuperar o rastreamento de execução.

Para resolver este problema, temos o padrão **Log aggregation, que por um componente externo nos permite concentrar todos os logs em uma única fonte de dados**, que podemos consultar posteriormente sem a necessidade de ter acesso físico aos servidores e sem se importar com quantas instâncias de cada componente temos. Desta forma, através deste padrão, podemos encontrar erros de forma mais eficiente, prestando um melhor atendimento aos nossos clientes.

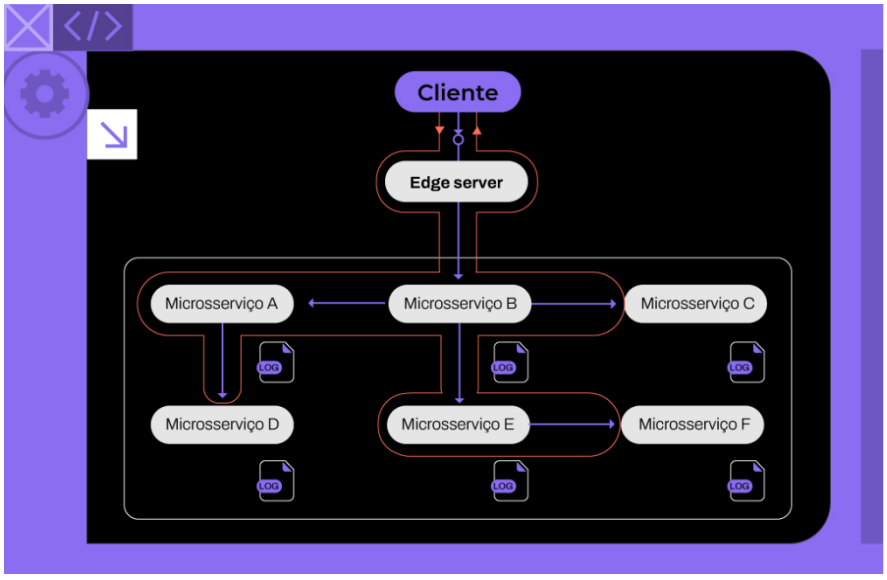


Distributed tracing

Como dissemos anteriormente, um dos principais problemas em uma arquitetura distribuída é a rastreabilidade da execução de um processo, pois uma única chamada pode afetar a chamada de vários serviços. Isso implica que temos que recuperar o log em partes, ou seja, temos que remover cada parte do log em cada microsserviço e depois juntá-los, o que pode ser uma tarefa titânica e complicada. Por isso, é importante implementar um **sistema de rastreamento distribuído que permita que os logs sejam unificados em um único ponto e agrupados por execução**.

Para analisar os atrasos em uma série de chamadas de microsserviços cooperantes, precisamos coletar carimbos de data e hora de quando solicitações, respostas e mensagens entram e saem de cada microsserviço.

Quando um microsserviço se comunica com outro, envia o ID de transação global e seu ID de transação em sua solicitação. Se um microsserviço não recebe esse ID, ele os gera. No protocolo HTTP, esses IDs são enviados e recebidos através dos cabeçalhos. Estes permitem correlacionar todos os rastreamentos emitidos pelos diferentes processos dos microsserviços de uma mesma solicitação na aplicação. Efetuando uma busca global pelo identificador global, obtém-se o conjunto de rastreamentos que foram emitidos pelos microsserviços pelos quais uma requisição passou.



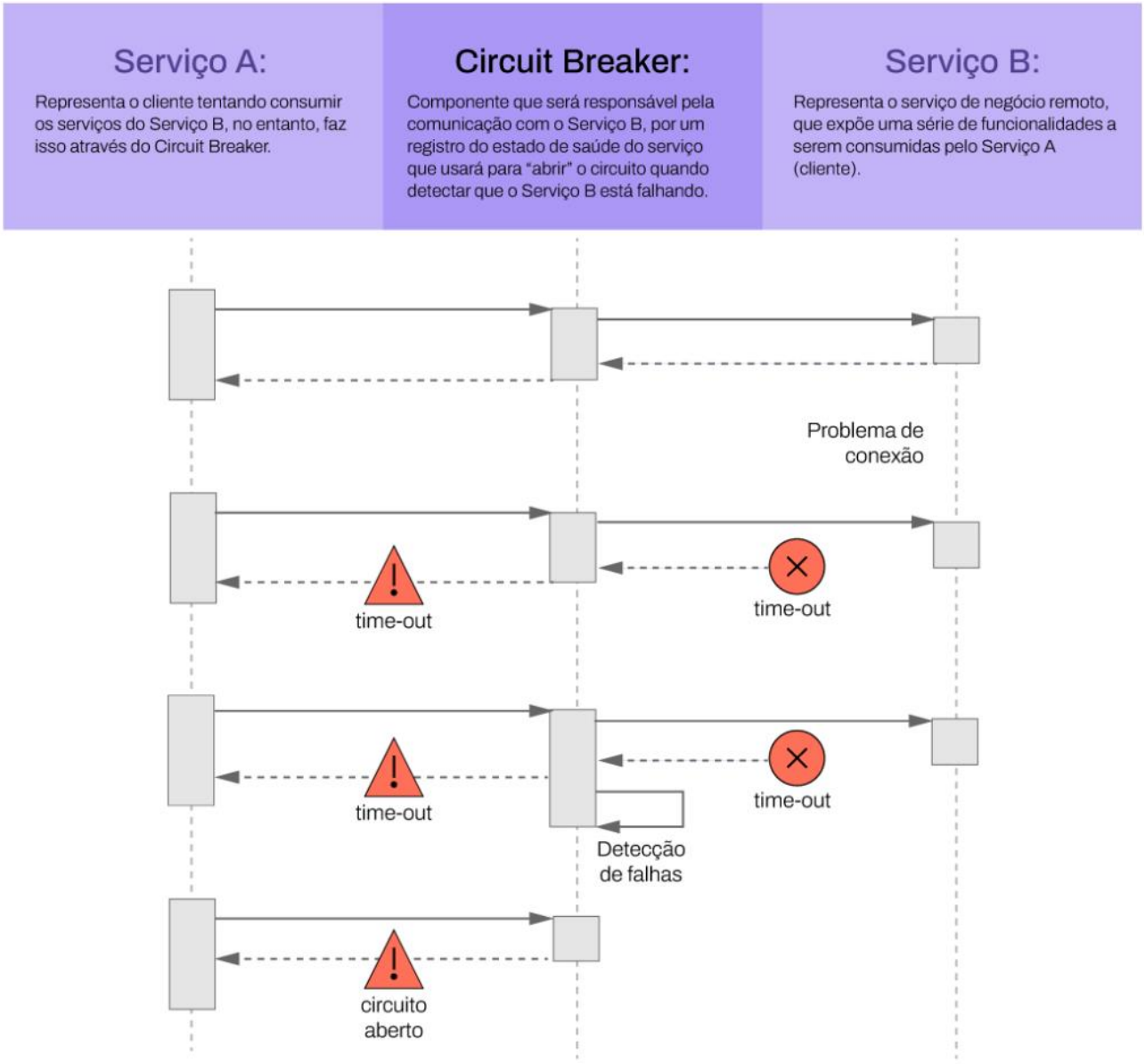
Como conclusão, precisamos garantir que todas as solicitações e mensagens relacionadas sejam marcadas com um ID de correlação comum e que o ID faça parte de todos os eventos de log. Com base nele, podemos usar o serviço de log centralizado para encontrar todos os eventos de log relacionados.

Circuit Breaker

Na prática, é comum que quando algo falhar, apenas mostramos ao usuário um erro de que algo não funcionou e que tente novamente mais tarde. Mas, e se o usuário estivesse tentando efetuar uma venda? Estamos dispostos a deixá-lo ir embora?

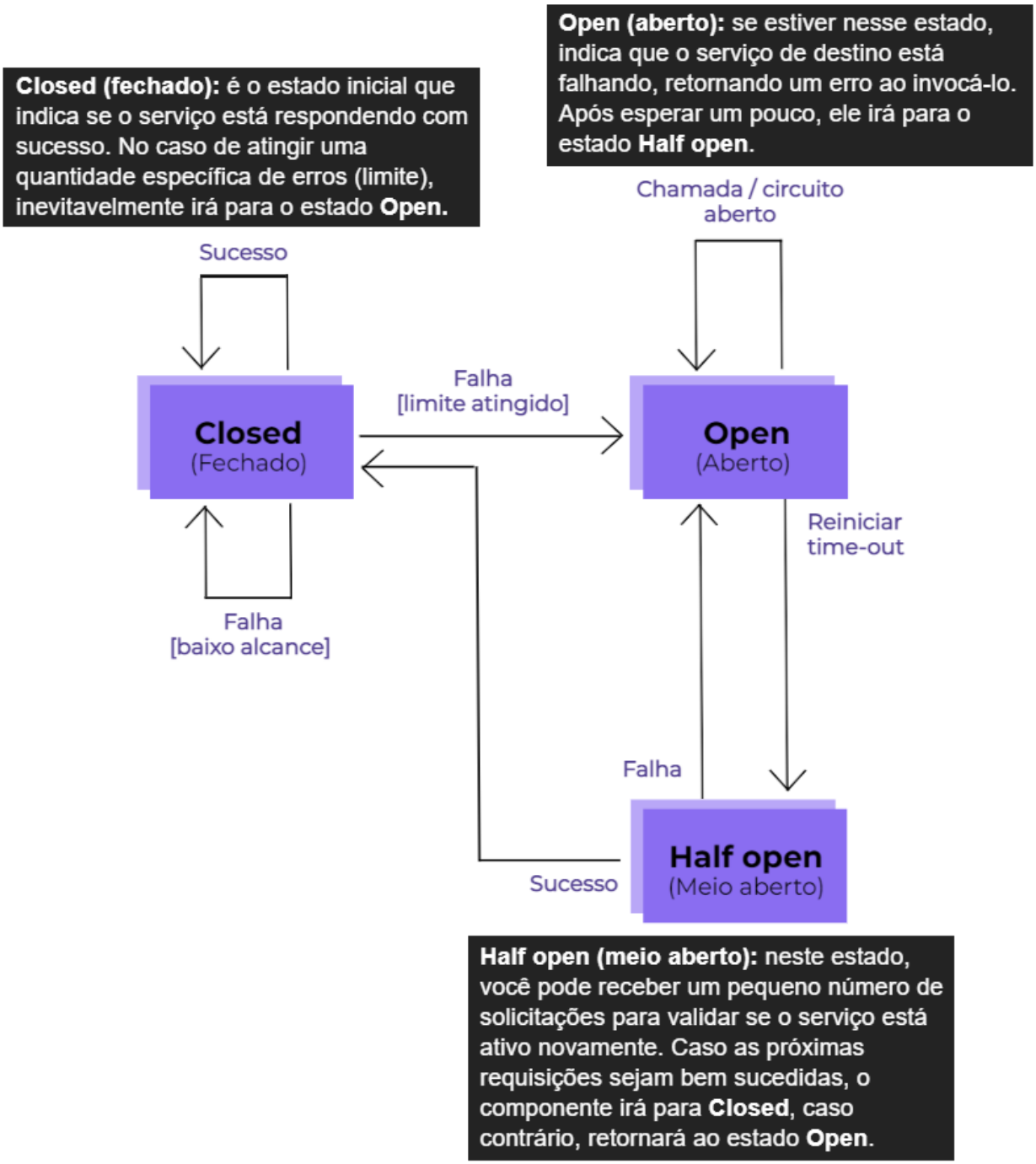
Com a chegada de novas arquiteturas distribuídas como os microsserviços, muitas vantagens vieram com ela, mas também surgiram novos problemas que são raramente conhecidos para se resolver com precisão. Um desses casos é **poder identificar quando um serviço parou de funcionar repentinamente** para então parar de esperar por solicitações e, ao mesmo tempo, fazer algo em consequência.

O padrão **Circuit Breaker** — disjuntor ou fusível (não deve ser confundido com "curto-circuito", em inglês short circuit) — propõe uma solução para o problema anterior e permite efetuar uma analogia como se fosse um fusível doméstico (elétrico), que derrete para evitar que uma descarga elétrica afete o circuito. Em outras palavras, **esse padrão permite cortar inteligentemente a comunicação com um determinado serviço quando é detectado que ele está falhando, evitando assim que o sistema continue falhando.**



Continuando com o exemplo, se fosse uma venda e se o serviço falhar, poderíamos pensar em parar de enviar pedidos e executar um plano B enquanto o serviço se recupera. Algumas alternativas poderiam ser tentar fazer novamente a venda em modo diferido, deixando passar alguns minutos que definimos explicitamente, ou enviá-los para uma fila para serem processados de forma assíncrona e depois podemos dar a confirmação ao consumidor final.

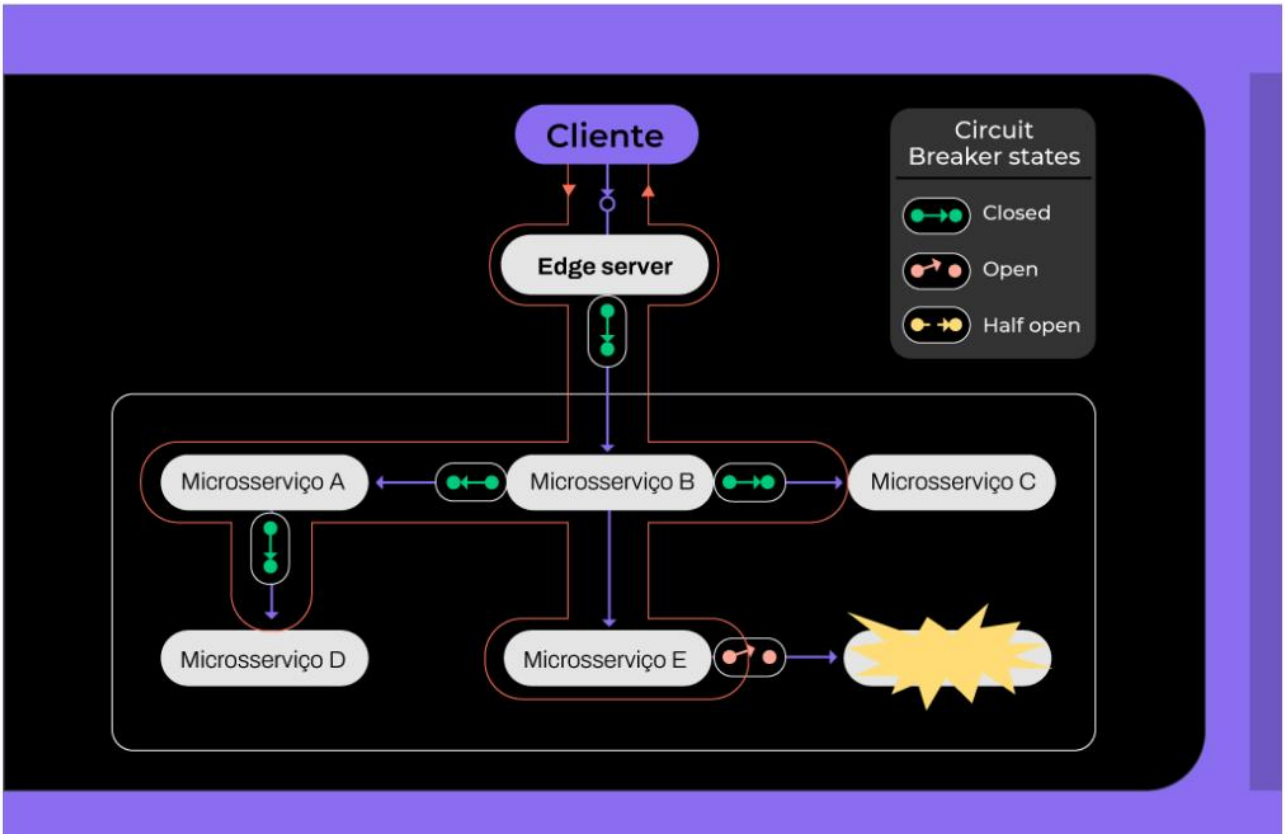
Finalmente, o padrão do Circuit Breaker pode passar por três estados diferentes. Cada um destes afetará como ele funciona. Estes são: fechados, abertos e meio abertos. Vamos prosseguir para saber mais.



Em geral, o padrão do Circuit Breaker é aplicado quando um serviço depende de outro. Se outro serviço falhar ou demorar demasiado tempo a responder (time-out), ele volta a tentar um determinado número de vezes. Uma vez ultrapassado este número, é devolvido um erro. Caso contrário, o funcionamento normal é retomado. Por sua vez, este padrão geralmente revela problemas de design. Suponha que o nosso serviço depende de um serviço de terceiros que falha continuamente. Uma opção é chamar o proprietário e pedir-

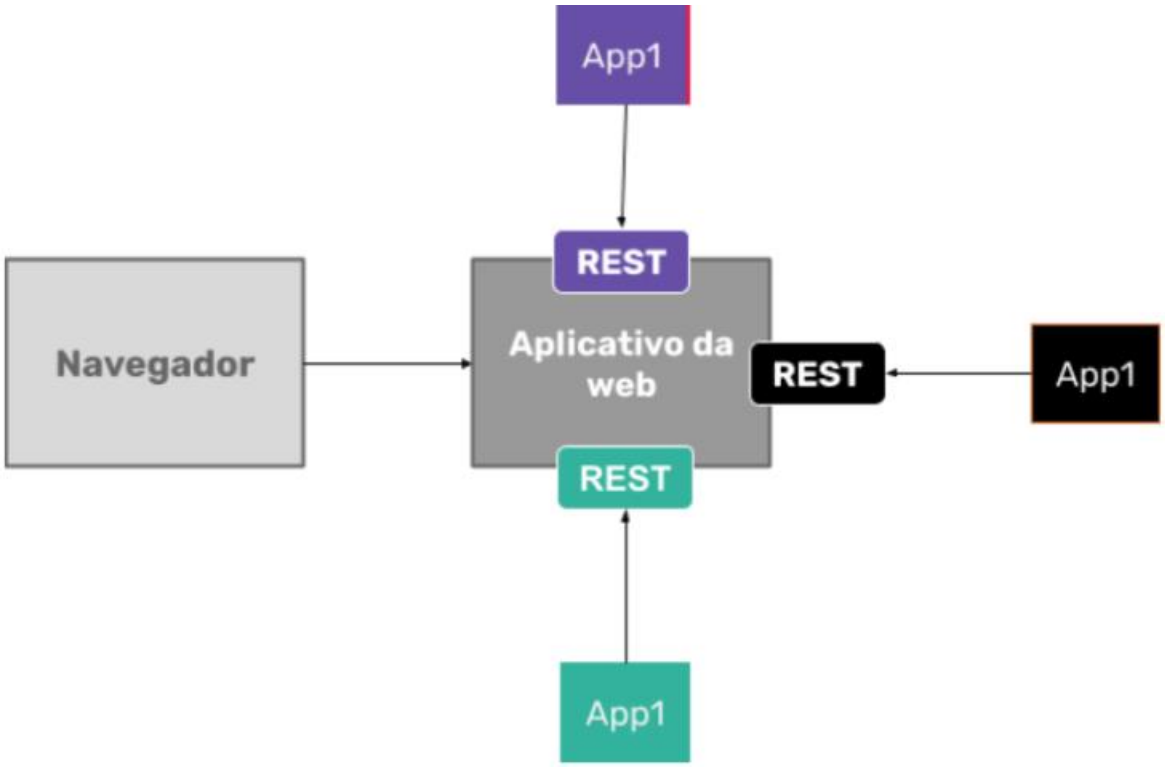
lhe para "consertá-lo", mas por vezes isto não é possível. Assim, a solução — em vez de uma chamada de serviço direta — deve ser utilizar uma fila de mensagens e tornar a operação assíncrona.

Em conclusão, o padrão do Circuit Breaker é amplamente utilizado em processos críticos, evitando antecipadamente que a nossa aplicação esteja envolvida num grande número de pedidos que sabemos que irão falhar, e tomando medidas em conformidade. Deve-se priorizar a resposta do serviço, mesmo que de uma forma diferente.

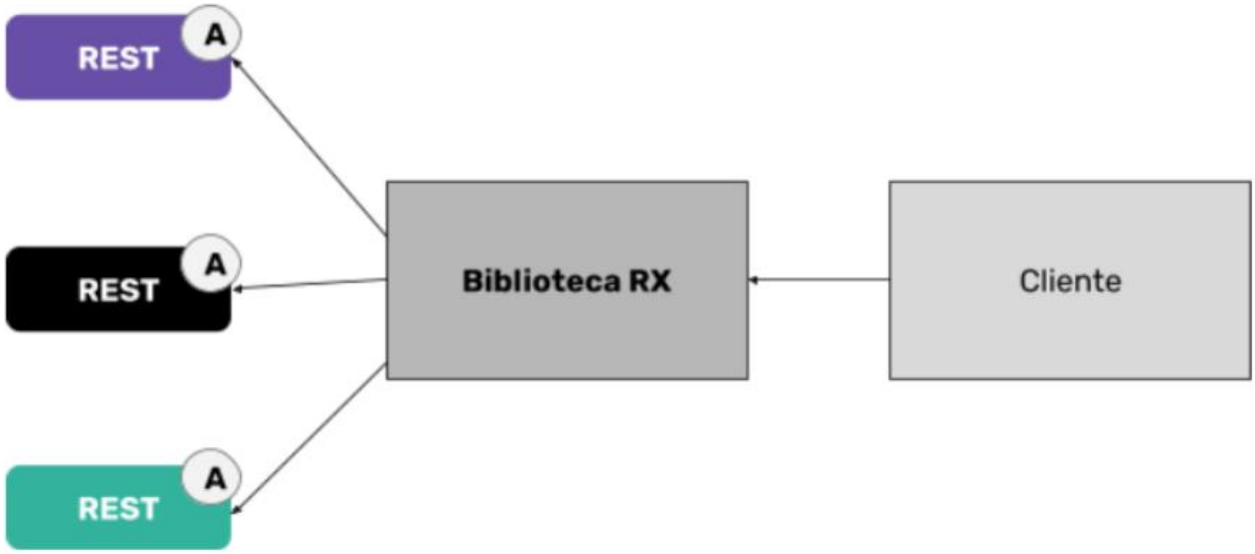


Reactive microservices

Como desenvolvedores, estamos acostumados a implementar a comunicação síncrona usando blocos de I/O (blocos de entrada/saída), por exemplo, para o caso de uma API RESTful JSON sobre HTTP. O servidor que recebe alguma requisição e, de acordo com ela, espera dar uma resposta atribui (no sistema operacional) uma thread pelo tempo que a requisição leva. Nesse cenário, se o número de solicitações simultâneas aumentar, esse servidor poderá ficar sem threads disponíveis. Isso pode causar uma variedade de problemas, desde tempos de resposta mais longos até falhas no servidor. Em uma arquitetura de microsserviços, esse problema geralmente é ainda pior, pois vários microsserviços cooperativos estão geralmente envolvidos para atender a uma solicitação. E quanto mais microsserviços estiverem envolvidos no atendimento de uma solicitação, mais rápido as threads disponíveis serão esgotadas.



Então, o que esse padrão oferece é usar chamadas assíncronas sem bloqueio sempre que possível. Isso inclui as chamadas usuais para recursos muito lentos na rede, as chamadas ao banco de dados, tratamento de solicitações e — em geral — todo o fluxo de chamadas. Dessa forma, precisaremos usar programação reativa para realizar uma comunicação assíncrona ideal e correta entre as diferentes API REST dos diferentes microsserviços que criamos.



Os sistemas construídos como sistemas reativos são mais flexíveis, pouco acoplados e escaláveis. Isso os torna mais fáceis de desenvolver e mais suscetíveis a mudanças. Eles são significativamente mais tolerantes a falhas e, quando ocorrem, lidam com uma solução pragmática em vez de resultar em desastre.

Vamos pensar em um exemplo de um microsserviço de bloqueio e não bloqueio

Suponha que temos um e-commerce com um processo de checkout de pedidos que consiste em checar ofertas, checar estoque e checar delivery. Cada etapa leva 1 segundo (com o servidor ocioso) e o servidor consegue manter 10.000 sessões abertas simultaneamente. Se nossa operação for síncrona de bloqueio, o usuário enviará sua solicitação e, após três segundos, todas as etapas serão executadas. Nesses três segundos tudo será verificado, mas, o que acontece se ocorrerem 10.000 pedidos simultâneos? O primeiro roda corretamente, o segundo demora um pouco mais (pois tem que esperar o anterior terminar) e, quando chegamos a 5.000, o sistema degrada e os erros começam a aparecer à medida que o servidor está saturado.

Agora, suponha que nossa operação seja "assíncrona". O servidor recebe o checkout do usuário e envia o evento para uma fila de mensagens para que os serviços de ofertas, estoque e delivery possam retirar os pedidos. O usuário recebe a mensagem "seu pedido está sendo processado" em um milésimo de segundo e o servidor é liberado. Então, se tivermos 10.000 pedidos e mais 10.000 chegarem no próximo segundo, o sistema não se

degradará. Talvez alguns pedidos "em andamento" não sejam atendidos por falta de estoque, mas na prática esse caso é muito melhor porque:

1. O servidor não está bloqueado e continuamos a receber pedidos.
2. A janela de estoque desatualizada é muito pequena.
3. Os fornecedores têm uma margem de estoque de segurança.
4. Podemos paralelizar processos.

Se você quiser saber mais sobre isso, em 2013 os princípios-chave para o projeto de sistemas reativos foram definidos em um documento chamado “The Reactive Manifesto” (O manifesto reativo). Você pode ler no link a seguir: https://www.reactivemanifesto.org

Centralized monitoring and alarms

Se os tempos de resposta observados e/ou o uso de recursos de hardware se tornarem inaceitavelmente altos, pode ser muito difícil descobrir a causa do problema. Precisamos conseguir analisar o consumo de recursos de hardware por microsserviço. Para isso, podemos adicionar um novo componente: Monitor service (um serviço de monitoramento), capaz de coletar métricas sobre o uso de recursos de hardware para cada nível de instância do microsserviço. Isso nos permitirá:

- Coletar métricas de todos os servidores usados pelo ambiente do sistema, incluindo servidores de escalonamento automático.
- Descobrir novas instâncias de microsserviço à medida que são lançadas em servidores disponíveis e começar a coletar métricas delas.
- Fornecer uma API e ferramentas gráficas para consultar e analisar as métricas coletadas.
- Definir alertas acionados quando uma determinada métrica excede um valor limite específico.

Em direção à sala de aula ao vivo

Já estamos chegando ao fim. Sabemos que foi uma aula longa, mas é o ponto de partida para o resto da matéria. Não se preocupe, se houver algo que você não tenha entendido, será estudado ao longo do curso. Ao mesmo tempo, à medida que avançamos no conteúdo, veremos quais ferramentas (a serem implementadas) do mercado temos para facilitar o uso de tais tarefas.

Para validar o conhecimento aprendido, sugerimos que você realize a seguinte atividade.