Avaliação de Desempenho de Dois Padrões de Resiliência para Microsserviços: Retry e Circuit Breaker

Thiago M. Costa[1], **Davi M. Vasconcelos**[1], **Carlos M. Aderaldo**[2], **Nabor C. Mendonça**[2]

[1]Bacharelado em Ciência da Computação
[2]Programa de Pós-Graduação em Informática Aplicada
Universidade de Fortaleza (UNIFOR)
Fortaleza – CE

{thiago.maia971,davimelovasc,cmendesce}@gmail.com, nabor@unifor.br

Abstract. *Microservice developers increasingly use resiliency patterns, such as Retry, Fail Fast, and Circuit Breaker, to cope with failures during the invocation of remote services. However, there is still little research on the impact of those resiliency patterns on application performance. This paper presents an experimental evaluation of the performance impact of the Retry and Circuit Breaker resilience patterns, as implemented by two popular open-source resilience libraries: Polly, for C#, and Resilience4j, for Java. The evaluation results show that the Retry pattern can be more effective than the Circuit Breaker pattern in reducing the application's contention for external resources, with both patterns causing a slight to moderate impact on its execution time.*

Resumo. *Desenvolvedores de microsserviços cada vez mais utilizam padrões de resiliência, como Retry, Fail Fast, e Circuit Breaker, para lidar com falhas durante a invocação de serviços remotos. Porém, ainda há poucos trabalhos na literatura sobre o impacto do uso desses padrões no desempenho das aplicações. Este trabalho apresenta uma avaliação experimental do impacto de desempenho dos padrões de resiliência Retry e Circuit Breaker, tais como implementados por duas populares bibliotecas de resiliência de código aberto: Polly, para a linguagem C#, e Resilience4j, para a linguagem Java. Os resultados da avaliação mostram que o padrão Retry pode ser mais efetivo que o padrão Circuit Breaker na redução da contenção por recursos externos da aplicação, com ambos os padrões causando de leve a moderado impacto no seu tempo de execução.*

1. Introdução

Aplicações de microsserviços, como qualquer outro tipo de sistema distribuído, são propensas a diferentes tipos de falhas operacionais. Causas diversas, como falhas de hardware, de software, ou excesso de carga na rede ou em serviços externos, podem tornar os microsserviços da aplicação indisponíveis ou inacessíveis a seus clientes, a qualquer momento [Jamshidi et al. 2018]. Para mitigar o impacto da indisponibilidade temporária dos microsserviços, os desenvolvedores precisam equipar suas aplicações com mecanismos para lidar com falhas de forma robusta e resiliente [Birolini 2013]. Um abordagem cada vez mais adotada na indústria para lidar com certos tipos de falhas operacionais em aplicações de microsserviços é implementar a comunicação entre os microsserviços utilizando *padrões de resiliência* [Nygard 2007].

Padrões de resiliência, como Retry e Circuit Breaker, introduzem mecanismos capazes de aguardar passivamente a recuperação de serviços remotos cuja invocação falhou, evitando, assim, que a aplicação desperdice valiosos recursos de processamento e de rede re-invocando continuamente serviços que estejam sobrecarregados ou momentaneamente inoperantes [Microsoft Azure 2017]. No entanto, a documentação disponível sobre o uso desses padrões é, tipicamente, omissa quanto ao seu potencial impacto no desempenho das aplicações. Entender o impacto de padrões de resiliência no desempenho de aplicações distribuídas é importante porque, ao deliberadamente retardarem a re-invocação de serviços remotos em situações de falha, esses padrões podem aumentar significativamente o tempo total de execução dos serviços que os utilizam, podendo, por sua vez, levar a falhas em cascata de outros serviços da aplicação [Mendonca et al. 2020].

Apesar do crescente interesse da comunidade científica por microsserviços [Di Francesco et al. 2019], há ainda poucos trabalhos publicados na literatura com foco na avaliação dos padrões de resiliência utilizados neste contexto [Aquino et al. 2019, Mendonca et al. 2020, Jagadeesan and Mendiratta 2020, Saleh Sedghpour et al. 2022]. Porém, nenhum desses trabalhos investigou experimentalmente o impacto do uso de padrões de resiliência, em particular, dos padrões de resiliência implementados por populares bibliotecas de resiliência amplamente utilizadas pela indústria, no desempenho de aplicações distribuídas. Com o intuito de preencher esta lacuna de pesquisa, este trabalho apresenta uma avaliação experimental do impacto de desempenho de dois conhecidos padrões de resiliência, Retry e Circuit Breaker, tais como implementados por duas das mais populares bibliotecas de resiliência disponíveis atualmente: Polly [App vNext 2022], para a linguagem C#, e Resilience4J [Resilience4j 2021], para a linguagem Java.

A avaliação conduzida considerou diferentes opções de configuração de cada padrão, diferentes cargas de trabalho impostas pela aplicação cliente, e diferentes taxas de falha do serviço alvo. Os resultados obtidos mostram que ambos padrões de resiliência podem causar de leve a moderado impacto no tempo de execução da aplicação, e que o padrão Retry, quando configurado com uma política de incremento exponencial do tempo de espera entre as tentativas de invocação, pode ser bem mais efetivo que o padrão Circuit Breaker na redução da contenção por recursos externos da aplicação. Nesse sentido, o padrão Retry é a solução de resiliência recomendada quando a aplicação necessita obter uma resposta diretamente de um serviço que falhou, com o padrão Circuit Breaker sendo indicado para os casos nos quais é possível contornar a falha do serviço alvo, por exemplo, devolvendo uma resposta padrão à aplicação ou invocando um serviço alternativo. A principal implicação dos resultados deste trabalho para desenvolvedores e pesquisadores de microsserviços é que os padrões Retry e Circuit Breaker devem ser utilizados e configurados com cautela, de modo a viabilizar seus benefícios de resiliência sem comprometer o desempenho das aplicações que os utilizam.

O restante deste trabalho está organizado da seguinte forma: a Seção 2 descreve e ilustra o uso dos padrões Retry e Circuit Breaker; a Seção 3 apresenta o método de avaliação experimental utilizado; a Seção 4 analisa os resultados obtidos; a Seção 5 discute os principais achados e implicações do trabalho; a Seção 6 compara os resultados do trabalho com outros trabalhos relacionados; por fim, a Seção 7 oferece as considerações finais e sugere tópicos para trabalhos futuros.

Tabela 1. Descrição dos padrões de resiliência Retry e Circuit Breaker.

	Padrão Retry
Propósito	O padrão Retry permite que uma aplicação trate falhas transitórias ao tentar invocar serviços remotos, reinvocando, de forma transparente, um serviço que tenha falhado.
Contexto	Um sistema distribuído deve ser resiliente a falhas temporárias que podem ocorrer em seu ambiente de execução. Essas falhas podem ser contornadas pela própria aplicação cliente, simplesmente reinvocando o serviço após um determinado tempo de espera.
Solução	Se uma aplicação detectar uma falha transitória ao tentar invocar um serviço remoto, esta deverá aguardar um período de tempo adequado (conhecido como *backoff*) antes de tentar novamente a invocação. Este processo é repetido até que a invocação seja bem-sucedida, ou que um limite máximo de falhas ou de tempo seja atingido, caso em que a aplicação considera que o serviço falhou definitivamente.
Aspectos de implementação	O intervalo de tempo entre as tentativas de invocar o serviço remoto deve ser definido de tal forma a evitar que múltiplas instâncias da aplicação reinvoquem o serviço ao mesmo tempo. Isto reduz a chance de que contínuas invocações a um serviço lento acabe por deixá-lo ainda mais lento. A aplicação também pode progressivamente aumentar o tempo de espera pelo serviço remoto, até obter uma resposta bem sucedida ou atingir um limite máximo de reinvocações. O aumento do tempo de espera pode ser feito de forma linear ou exponencial, dependendo do tipo de falha e da probabilidade do serviço se recuperar mais rápida ou lentamente quando falhar.
	Padrão Circuit Breaker
Propósito	O padrão Circuit Breaker permite que uma aplicação trate falhas que podem levar uma quantidade variável de tempo para se recuperar, quando invoca um serviço remoto.
Contexto	Em um ambiente distribuído, invocações a serviços remotos podem falhar devido a eventos imprevistos que podem durar por muito mais tempo que falhas transitórias. Nessas situações, é inútil para uma aplicação continuar tentando reinvocar um serviço que provavelmente não irá responder no curto prazo: em vez disso, a aplicação deve aceitar rapidamente que a operação falhou e lidar com essa falha de acordo.
Solução	O padrão Circuit Breaker atua como um *proxy* local para serviços sujeitos a situações de falha. O *proxy* deve monitorar o número de invocações mal-sucedidas recentes, e usar esta informação para decidir se deve invocar o serviço em nome da aplicação, ou simplesmente retornar um erro imediatamente. O *proxy* pode ser implementado como uma máquina de estado com os seguintes estados, que imitam os estados de um disjuntor elétrico: fechado, aberto, e semi-aberto [Fowler 2014]. Inicialmente, o disjuntor está no estado fechado, o que significa que o *proxy* vai encaminhar todas as requisições recebidas da aplicação ao serviço remoto. O disjuntor permanecerá neste estado até que um certo limiar de invocações mal-sucedidas seja alcançado, quando então passa para o estado aberto. Neste estado, o disjuntor inicia um mecanismo de *timeout* e imediatamente retorna uma mensagem de erro à aplicação a cada nova requisição ao serviço recebida, até que seu *timeout* expire, quando então o disjuntor passa para o estado semi-aberto. Uma vez neste estado, o disjuntor volta a invocar o serviço remoto em nome da aplicação cliente, passando para o estado fechado, se atingir um determinado limiar de invocações bem-sucedidas, ou retornando ao estado aberto, se atingir um determinado limiar de falhas.
Aspectos de implementação	O padrão Circuit Breaker é customizável e pode ser adaptado de acordo com o tipo e a expectativa de duração das falhas. Por exemplo, o disjuntor pode ser colocado no estado Aberto por alguns segundos, de início, e aí, a depender das falhas observadas, ajustar o seu valor de *timeout* conforme a necessidade. Outra alternativa é o *proxy* que implementa o disjuntor retornar à aplicação um valor padrão como resultado da invocação do serviço remoto, ao invés de uma mensagem de erro, sempre que estiver no estado Aberto.

2. Padrões de Resiliência

A Tabela 1 descreve os padrões Retry e Circuit Breaker em termos de seu *propósito*, *contexto*, *solução*, e *aspectos de implementação*. A notação adotada é baseada na documentação disponível em [Microsoft Azure 2017]. O comportamento de ambos os padrões para lidar com situações de falha durante a invocação de serviços remotos será ilustrado no contexto de um cenário ilustrativo, descrito a seguir.

Considere um cenário simples de interação remota, no qual uma aplicação cliente invoca sequencialmente uma operação oferecida por um serviço remoto, conforme ilustrado na Figura 1(a). Nesse cenário, uma requisição pode ser bem-sucedida, caso em que a aplicação cliente recebe uma resposta "OK" do serviço remoto, ou falhar devido ao serviço remoto estar indisponível ou muito lento, caso em que a aplicação cliente recebe uma mensagem de erro. Um importante desafio relacionado à implementação da

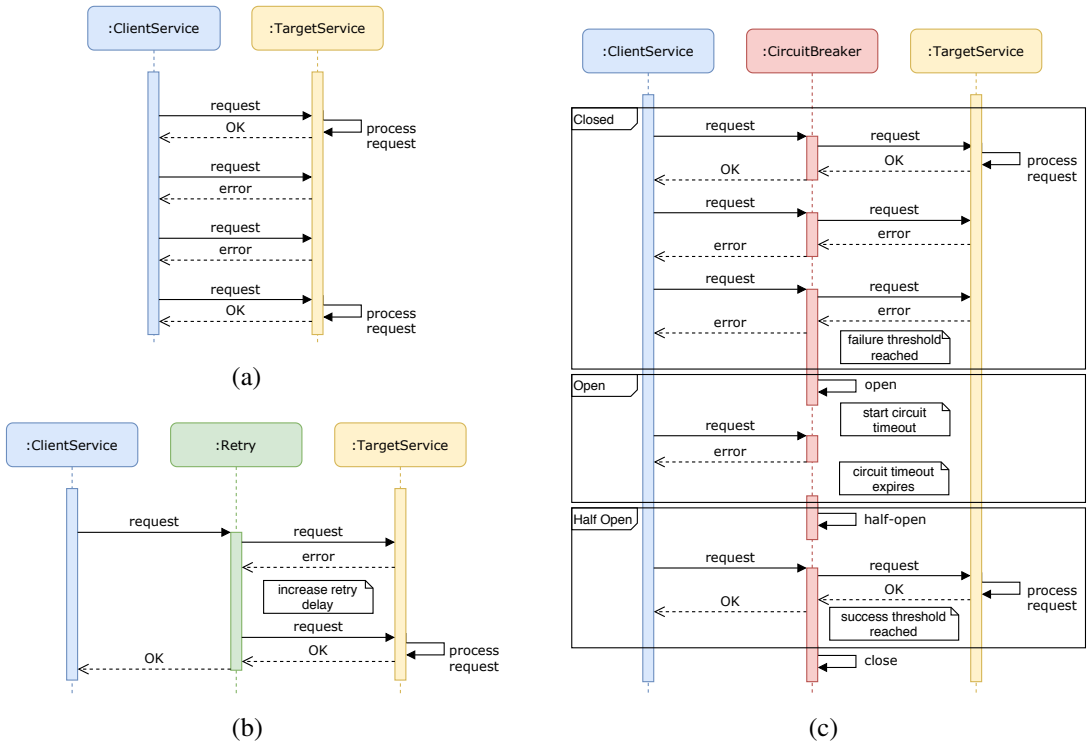

Figura 1. (a) Um cenário ilustrando a interação entre uma aplicação cliente e um serviço remoto sujeito a falhas operacionais; e exemplos de utilização dos padrões (b) Retry e (c) Circuit Breaker no contexto desse cenário.

aplicação cliente, neste cenário, é como lidar com um serviço remoto não responsivo. Basicamente, há dois casos indesejados e mutuamente conflitantes que devem ser evitados em caso de falhas do serviço remoto:

1. Se a aplicação cliente tentar minimizar seu tempo de execução reinvocando continuamente um serviço que falhou, isto inevitavelmente irá aumentar sua contenção por recursos externos, o que poderá contribuir para sobrecarregar ainda mais a rede ou o serviço remoto, além de representar um claro desperdício de recursos.
2. Se, por outro lado, a aplicação cliente tentar reduzir sua contenção por recursos externos retardando as re-invocações ao serviço que falhou, na expectativa de que este vai precisar de mais algum tempo para se recuperar da falha, isto poderá retardar o seu próprio tempo de execução, possivelmente retardando o tempo de execução de outros serviços que dependem da aplicação.

Os padrões Retry e Circuit Breaker oferecem soluções diferentes para tentar minimizar as consequências decorrentes dos dois casos descritos acima [Microsoft Azure 2017]. As Figuras 1(b) e 1(c) ilustram o uso desses dois padrões no contexto do cenário mostrado na Figura 1(a). No exemplo da Figura 1(b), o mecanismo que implementa o padrão Retry invoca o serviço remoto duas vezes antes de receber uma resposta bem sucedida. Note que o mecanismo aumenta o tempo de espera após a primeira tentativa de invocação do serviço, permitindo, assim, um maior tempo ao serviço para este se recuperar da falha. Já no exemplo da Figura 1(c), o mecanismo que implementa o padrão Circuit Breaker abre o circuito e inicia a contagem do intervalo de *timeout* após duas falhas consecutivas do serviço, devolvendo imediatamente um erro à

aplicação enquanto o circuito permanecer neste estado. Ainda nesse exemplo, o circuito muda para o estado semi-aberto após encerrado o intervalo de *timeout*, e então retorna ao estado fechado após uma nova invocação ao serviço alvo ser realizada com sucesso.

O objetivo deste trabalho é avaliar experimentalmente o impacto que diferentes configurações dos padrões Retry e Circuit Breaker podem causar no desempenho das aplicações que os utilizam, sob variadas cargas de trabalho e variadas taxa de falha do serviço remoto. As próximas seções descrevem o método de avaliação utilizado no trabalho e os resultados experimentais obtidos.

3. Método de Avaliação

Esta seção descreve as questões de pesquisa investigadas no âmbito deste trabalho, bem como o ambiente de teste utilizado, os experimentos realizados, e as métricas de desempenho coletadas.

3.1. Questões de Pesquisa

Este trabalho procura responder as seguintes questões:

RQ1 Qual é o impacto no desempenho do uso de diferentes configurações dos padrões de resiliência Retry e Circuit Breaker no desempenho de uma aplicação cliente que invoca continuamente um serviço remoto propenso a falhas?

RQ2 Como o impacto no desempenho da aplicação é influenciado por fatores como a biblioteca de resiliência utilizada, a carga de trabalho imposta pela aplicação cliente, e a taxa de falha do serviço remoto?

Para responder as questões acima, foi necessário desenvolver um novo ambiente de teste voltado especificamente para a avalização experimental de padrões de resiliência.

3.2. Ambiente de Teste

O ambiente de teste integra três componentes principais: uma aplicação cliente, um servidor HTTP, e um servidor *proxy* (ver Figura 2). A aplicação cliente atua como gerador de carga para o servidor HTTP ao criar múltiplas instâncias (linhas de execução), cada uma responsável por invocar continuamente o servidor HTTP até atingir um dado número mínimo de invocações bem sucedidas, utilizando uma das seguintes estratégias de invocação: (1) com o padrão Retry; (2) com o padrão Circuit Breaker; e (3) sem padrão de resiliência, referida neste trabalho como estratégia Baseline. O servidor *proxy*, por sua vez, atua como intermediário entre a aplicação cliente e o servidor HTTP, sendo responsável por injetar falhas no fluxo de requisições entre a aplicação cliente e o servidor HTTP de acordo com uma dada taxa de falha (por exemplo, uma taxa de falha de 0,25 significa que o *proxy* irá injetar uma falha a cada três invocações bem sucedidas ao servidor HTTP). A vantagem de se utilizar um servidor *proxy* como mecanismo de injeção de falhas é que isso permite configurar a taxa de falha do servidor independentemente do servidor HTTP utilizado nos testes.

A aplicação cliente foi implementada em duas versões funcionalmente equivalentes: uma em C#, que utiliza a biblioteca de resiliência Polly, e outra em Java, que utiliza a biblioteca de resiliência Resilience4j. Ambas versões são compostas de dois módulos: o *módulo de resiliência*, responsável por invocar o servidor HTTP utilizando uma das três

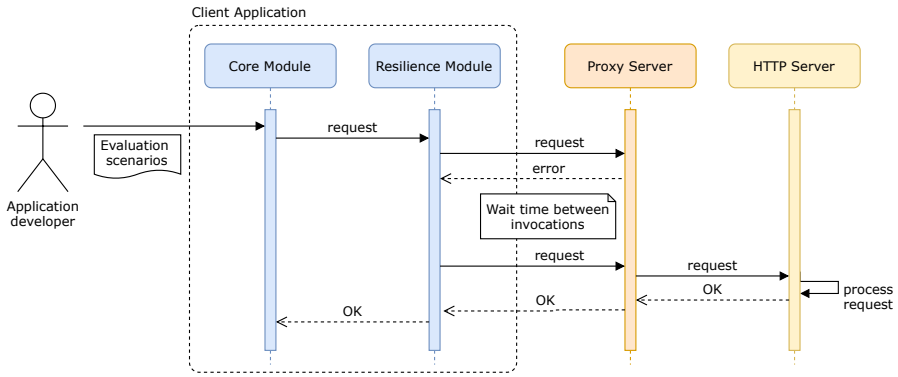

Figura 2. Ambiente de teste utilizado na avaliação dos padrões de resiliência.

estratégias descritas acima, e o *módulo principal*, responsável por processar os parâmetros do teste (descritos abaixo), instanciar o servidor HTTP e o servidor *proxy*, criar e executar o número desejado de instâncias clientes que irão invocar o servidor HTTP via módulo de resiliência, e coletar as métricas de desempenho.

Como servidor HTTP, foi utilizado o httpbin [Postman 2017], que implementa um serviço de eco muito usado para testes de clientes HTTP. Já como servidor *proxy*, foi utilizado o Vaurien [Mozilla 2012], um ferramenta de engenharia do caos [Rosenthal et al. 2017] muito utilizada para injetar falhas em fluxos de requisições HTTP. Por fim, dois contêineres Docker [Merkel 2014] foram utilizados para implantar os componentes do ambiente de teste, sendo um contêiner para a aplicação cliente e outro contêiner para o servidor *proxy* e o servidor HTTP. A decisão de alocar os dois servidores no mesmo contêiner foi feita de modo a minimizar o tempo de comunicação entre os dois servidores, e também para evitar que o mecanismo de injeção de falhas, ao barrar requisições enviadas pela aplicação cliente, reduzisse a carga de trabalho imposta sobre o contêiner do servidor do HTTP.

Todos os testes foram executados em um ambiente controlado, com os dois contêineres sendo implantados em um único computador com processador Core i5 da 7ª geração com quatro núcleos e 16GB de RAM. A decisão de implantar os dois contêineres na mesma máquina física justifica-se pela necessidade de minimizar a influência de eventuais oscilações da rede nos tempos de resposta do servidor HTTP observados pela aplicação cliente, o que poderia comprometer a confiabilidade dos resultados dos testes. Durante os testes, o Docker foi configurado para utilizar até 2 núcleos e 6GB de RAM. Estas restrições foram necessárias para reduzir o risco de algum dos contêineres exaurir os recursos da máquina hospedeira, o que também poderia comprometer os resultados.

3.3. Cenários de Teste

Um cenário de teste é composto por um conjunto de parâmetros definidos em um arquivo no formato JSON, o qual é passado como dado de entrada para a aplicação cliente no início de cada teste. Neste trabalho, foram definidos vários cenários de teste envolvendo múltiplas estratégias de resiliência, múltiplas cargas de trabalho, e múltiplas taxas de falha do servidor HTTP. A Tabela 2 apresenta os parâmetros de teste utilizados na definição dos cenários de teste e seus respectivos valores. Um cenário de teste corresponde a uma configuração dos parâmetros de teste descritos na Tabela 2, com cada parâmetro sendo

Tabela 2. Parâmetros utilizados nos testes.

Categoria	Parâmetro	Descrição	Valor(es) Utilizado(s)
Controle	*Rounds*	Número de testes executados em cada cenário	5
	Clients	Número de instâncias da aplicação cliente	25, 50, 100
	ServerUrl	Endereço do servidor	`http://localhost:5001`
	ServerFailRate	Taxa de falha do servidor	0, 0,25, 0,5
	ResiliencePolicy	Estratégia de resiliência utilizada pela aplicação cliente	Retry, Circuit Breaker, *Baseline* (sem padrão)
	MinSuccRequests	Número mínimo esperado de invocações bem sucedidas ao servidor em cada teste	25
	MaxRequests	Número máximo permitido de invocações ao servidor em cada teste (utilizado como condição de parada quando o número mínimo de invocações bem sucedidas não é atingido)	1000
Retry	*MaxRetries*	Número máximo de tentativas consecutivas de invocação do servidor	5
	WaitTime	Intervalo de tempo inicial (em ms) entre as tentativas de invocação do servidor	50, 75, 100, 200
	WaitTimeIncPolicy	Política de incremento do intervalo de tempo entre as tentativas de invocação do servidor, podendo ser *Constante* ou *Exponencial*	*Exponencial*
	Power	Fator de multiplicação da política de incremento *Exponencial*	1,5
Circuit Breaker	*FailThreshold*	Número de falhas consecutivas necessárias para abrir o circuito	2
	SuccThreshold	Número de invocações bem sucedidas consecutivas necessárias para fechar o circuito	1
	BreakingTimeout	Intervalo de tempo (em ms) durante o qual o circuito permanece aberto	50, 75, 100, 200

atribuído um único valor. Esses parâmetros são agrupados em três categorias: parâmetros de controle do teste; parâmetros do padrão Retry; e parâmetros do padrão Circuit Breaker. Os parâmetros de controle relativos ao número de instâncias da aplicação cliente e à taxa de falha do servidor foram definidos de modo a produzir cenários com crescentes cargas de trabalho e crescentes níveis de indisponibilidade do serviço httpbin. Já os parâmetros dos padrões Retry e Circuit Breaker relativos ao intervalo de tempo entre re-invocações foram definidos de modo a impor crescentes níveis de impacto no tempo total de execução da aplicação, considerando o tempo esperado de resposta do serviço httpbin diante das diferentes cargas de trabalho investigadas. Para facilitar a comparação do desempenho da aplicação cliente utilizando os dois padrões de resiliência, esse parâmetros foram definidos com o mesmo conjunto de valores. Os demais parâmetros de configuração de cada padrão foram definidos com seus respectivos valores padrão.

3.4. Métricas de Desempenho

Em cada teste, a aplicação cliente coleta duas métricas de desempenho, *tempo de execução* e *taxa de contenção*, cujo impacto no desempenho da aplicação foram discutidos na Seção 2. O tempo de execução corresponde ao tempo que cada instância da aplicação cliente leva para completar o número mínimo de invocações bem sucedidas ao servidor HTTP. No caso de uma instância não conseguir completar o número mínimo de invocações bem sucedidas desejado, o que acontece quando a instância atinge o número máximo de invocações permitidas, suas métricas são descartadas. Quanto menor o tempo de execução de uma instância da aplicação, maior sua eficiência na utilização de recursos do sistema, tanto externos quanto internos.

Já a taxa de contenção é calculada pela razão entre o tempo de contenção acumulado pela instância da aplicação e seu tempo de execução. O tempo de contenção

acumulado, por sua vez, é calculado pela soma de todos os intervalos de tempo durante os quais a instância da aplicação ficou esperando uma resposta do servidor HTTP, independentemente se a invocação foi bem sucedida ou falhou. O tempo de contenção acumulado corresponde a uma fração do tempo de execução, excluindo-se os tempos nos quais a instância da aplicação ficou passivamente aguardando entre uma e outra tentativa de invocação do servidor HTTP, no caso do padrão Retry, ou enquanto o circuito permanecia no estado aberto, no caso do padrão Circuit Breaker. Quanto menor a taxa de contenção de uma instância da aplicação, menor o nível de desperdício de recursos externos do sistema gerado pela aplicação cliente diante da falha do servidor.

4. Resultados

As Figuras 3 e 4 mostram os resultados obtidos para as métricas tempo de execução e taxa de contenção, respectivamente, nos diferentes cenários avaliados. Em ambas as figuras, as linhas da grade contêm os resultados referentes às duas bibliotecas de resiliência, enquanto as colunas contêm os resultados referentes às diferentes cargas de trabalho. Cada métrica é representada nos gráficos pelo valor da mediana calculado com intervalo de confiança de 95% considerando os valores coletados por todas as instâncias da aplicação em todos os testes de cada cenário. Os resultados da três estratégias de resiliência (Retry, Circuit Breaker, e Baseline) são representados nos gráficos em curvas com diferente cores e estilos, com a espessura das curvas representando os diferentes valores atribuídos aos parâmetros *WaitTime* ou *BreakingTimeout*, dos padrões Retry e Circuit Breaker, respectivamente. Para manter a consistência visual entre as curvas, considera-se que a estratégia Baseline possui um intervalo de espera entre as invocações de 0 ms.

4.1. Tempo de Execução

Como pode ser visto na Figura 3, o tempo de execução da aplicação cresce à medida que crescem a taxa de falha do servidor e a carga de trabalho. Este resultado era esperado porque uma maior taxa de falha do servidor significa que a aplicação tem que gastar mais tempo reinvocando o servidor, no caso da estratégia Baseline, ou esperando que este se recupere, no caso dos padrões Retry e Circuit Breaker. Além disso, com o aumento da carga de trabalho o servidor passa a demorar mais para responder as requisições da aplicação cliente, o que também impacta negativamente o seu tempo de execução.

Comparando-se as três estratégias de invocação, observa-se que ambos os padrões causam um ligeiro a moderado aumento no tempo de execução da aplicação em relação à estratégia Baseline. Os piores resultados são observados nos cenários com a maior taxa de falha do servidor e os maiores intervalos de espera entre as invocações. Isto acontece porque, com essas configurações, ambos os padrões bloqueiam a aplicação por mais tempo enquanto aguardam a recuperação do servidor HTTP, o que não acontece quando esta utiliza a estratégia Baseline. Também é possível observar que o impacto relativo no tempo de execução causado pelos dois padrões tende a diminuir conforme a carga de trabalho aumenta. A explicação é que, sob maiores cargas de trabalho, o tempo de resposta do servidor passa a dominar o tempo de execução da aplicação, com este sendo relativamente menos afetado pela taxa de falha do servidor.

Finalmente, observam-se resultados muito similares para as duas bibliotecas de resiliência, com a Polly produzindo tempos de execução ligeiramente maiores que aqueles obtidos com a Resilience4j na maioria dos cenários. A exceção foi o cenário com

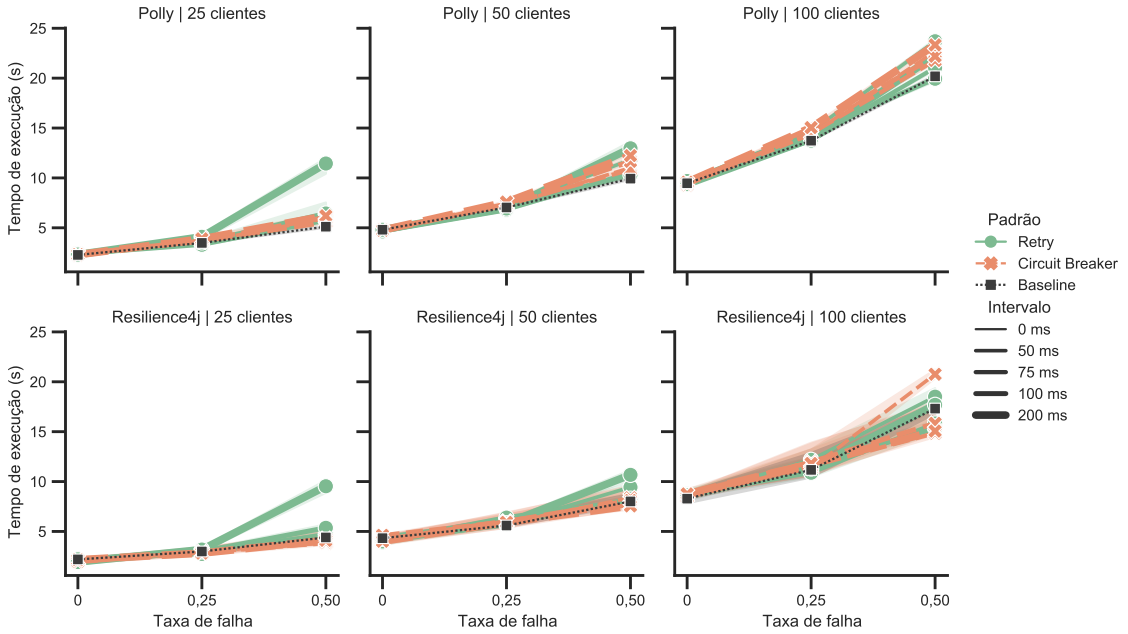

Figura 3. Impacto da taxa de falha do servidor HTTP no tempo de execução.

100 instâncias da aplicação cliente e 50% de taxa de falha do servidor, no qual as configurações dos padrões Retry e Circuit Breaker implementados pela Resilience4j com os menores intervalos entre invocações obtiveram tempos de execução ainda menores que os da própria estratégia Baseline. Uma possível explicação para esse resultado é que o uso dessas configurações dos dois padrões com a Resilience4j, neste cenário específico, acabou por reduzir a carga de trabalho efetivamente imposta ao servidor em relação à estratégia Baseline, implicando em menores tempos de resposta do servidor e, consequentemente, menores tempos de execução da aplicação cliente. O fato deste mesmo resultado não ter sido observado com os mesmos padrões implementados pela biblioteca Polly pode sugerir que a implementação desses dois padrões é mais eficiente na biblioteca Resilience4j. No entanto, como essas duas bibliotecas são implementadas em diferentes linguagens de programação, com cada linguagem tendo seu próprio ambiente de execução, não é possível concluir categoricamente que as diferenças de desempenho observadas entre elas devam-se unicamente a diferenças na implementação de seus respectivos padrões de resiliência.

4.2. Taxa de Contenção

A Figura 4 mostra que ambos os padrões de resiliência conseguem efetivamente reduzir a taxa de contenção da aplicação em relação à estratégia Baseline. Este resultado explica-se pelo fato de que ambos os padrões bloqueiam as ações da aplicação diante da falha do servidor, evitando, assim, que esta reinvoque o servidor imediatamente. Note-se que a taxa de contenção da estratégia Baseline é praticamente constante e muito próxima a 1 em todos os cenários avaliados. Isso acontece porque essa estratégia continuamente invoca o servidor mesmo quando este falha.

Também é possível observar, na Figura 4, que o padrão Retry obtém ganhos na taxa de contenção da aplicação significativamente maiores que aqueles obtidos pelo

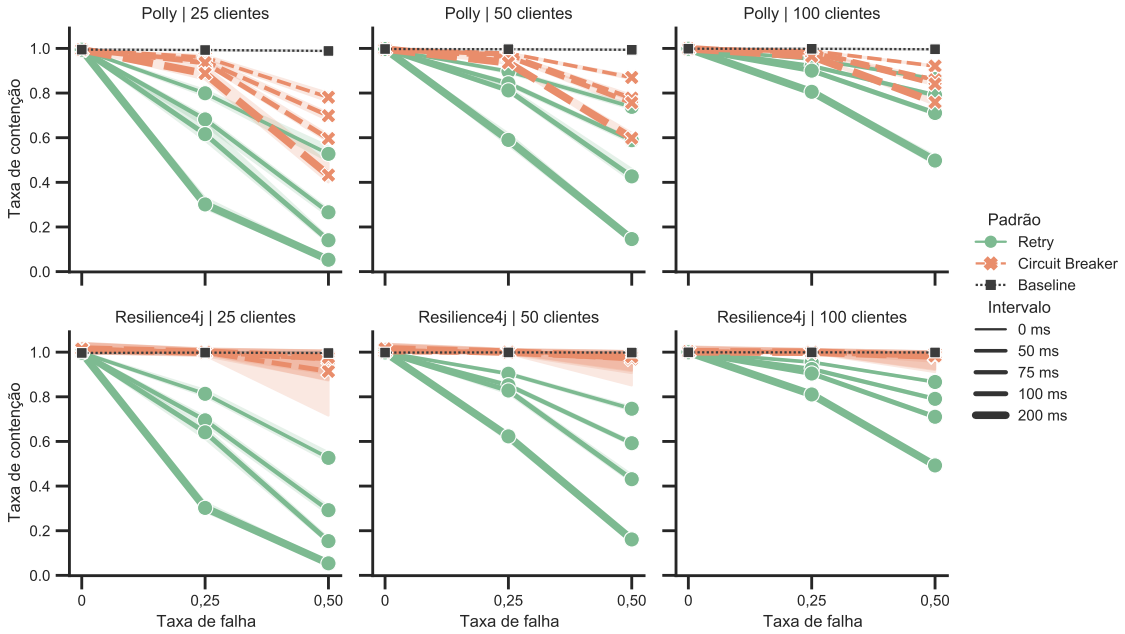

Figura 4. Impacto da taxa de falha do servidor HTTP na taxa de contenção.

padrão Circuit Breaker, e que essa diferença cresce à medida que cresce a taxa de falha do servidor. A explicação para este resultado é que, ao adotar uma política de incremento exponencial do intervalo de tempo entre as tentativas de invocação do servidor, o padrão Retry consegue bloquear as ações da aplicação cliente por mais tempo que o padrão Circuit Breaker, cujo intervalo de espera no estado aberto permanece constante durante todo o teste.

Outro resultado interessante é que o impacto na taxa de contenção da aplicação obtido pelos dois padrões de resiliência diminui à medida que cresce a carga de trabalho. Isso acontece porque, como explicado anteriormente, o impacto relativo dos dois padrões tende a diminuir à medida que o tempo de execução da aplicação passa a ser dominado pelo tempo de resposta do servidor.

Finalmente, a Figura 4 mostra que ambas as bibliotecas de resiliência obtiveram resultados muito similares em termos do impacto de seus respectivos padrões na taxa de contenção da aplicação, assim com já havia sido observado para o tempo de execução. A exceção são os resultados do padrão Circuit Breaker com a biblioteca Polly, que obteve ganhos na taxa de contenção ligeiramente maiores que aqueles obtidos com o mesmo padrão pela biblioteca Resilience4j. Uma possível explicação para essa diferença é que a implementação do padrão Circuit Breaker na Polly seja ligeiramente mais eficiente que a implementação deste mesmo padrão na Resilience4j. Essa hipótese, no entanto, precisaria ser investigada mais a fundo para ser confirmada, conforme discutiu-se anteriormente, durante a análise dos resultados do tempo de execução.

5. Discussão e Implicações

Um primeiro ponto a ser discutido é o fato do padrão Retry ter apresentado resultados significativamente melhores que os do padrão Circuit Breaker, tanto em termos de tempo de execução quanto em termos de taxa de contenção, em todos os cenários investigados.

Este resultado contrasta, pelo menos em princípio, com as informações e guias de uso disponíveis publicamente sobre o uso de padrões de resiliência, as quais recomendam o padrão Circuit Breaker como sendo o mais apropriado para lidar com falhas de natureza não transitória [Microsoft Azure 2017].

Uma análise mais atenta do modo como o padrão Circuit Breaker foi utilizado neste trabalho pode explicar essa aparente contradição entre a documentação dos dois padrões e os resultados obtidos experimentalmente. Durante os testes, o módulo de resiliência da aplicação cliente sempre retornava um erro para o módulo principal durante todo o tempo que o circuito permanecia no estado aberto. Isso fazia com que o módulo principal imediatamente reinvocasse o servidor, forçando o módulo de resiliência a barrar essa e todas as invocações subsequentes, até que o circuito retornasse ao estado fechado. Na prática, essa forma de tratar os erros gerados pelo padrão Circuit Breaker produz um comportamento da aplicação muito similar ao do padrão Retry, com o tempo que o circuito permanece no estado aberto correspondendo ao tempo de espera entre as reinvocações do padrão Retry. Porém, diferentemente do padrão Circuit Breaker, o padrão Retry permite que o intervalo de espera entre as invocações seja aumentado exponencialmente a cada sucessiva falha do servidor, o que torna seu mecanismo de bloqueio da aplicação cliente bem mais efetivo que aquele utilizado pelo padrão Circuit Breaker, especialmente em cenários com alta probabilidade de falha do servidor. Portanto, a partir dos resultados experimentais obtidos no contexto deste trabalho, recomenda-se: (i) *utilizar o padrão Retry quando houver a necessidade da aplicação sempre reinvocar um serviço remoto que falhou*; e (ii) *utilizar o padrão Circuit Breaker quando for mais conveniente mascarar completamente a falha do serviço remoto, por exemplo, retornando um valor padrão à aplicação ou redirecionando a invocação para outro serviço alternativo*.

Outro ponto a discutir é a definição do intervalo de espera entre as invocações, um fator crucial para o comportamento de ambos os padrões, e que pode impactar significativamente as métricas de desempenho da aplicação. Por um lado, aumentar o intervalo de espera entre as invocações tende a reduzir a taxa de contenção da aplicação, como mostram os resultados da taxa de contenção obtidos pelos padrões Retry e Circuit Breaker com intervalos de 100 ms e 200 ms (Figura 4). Por outro lado, longos intervalos de espera entre as invocações tendem a aumentar o tempo total de execução da aplicação, como mostram os resultados do tempo de execução obtidos com essas mesmas configurações dos padrões (Figura 3). Esta dicotomia entre os resultados obtidos para essas duas métricas evidencia a importância de se avaliar experimentalmente o potencial impacto de padrões de resiliência no desempenho das aplicações que os utilizam.

Por fim, acredita-se que os resultados deste trabalho possam ser de grande valor aos pesquisadores e desenvolvedores interessados no uso de padrões de resiliência como mecanismo de melhoria da qualidade de aplicações de microsserviços. Os desenvolvedores podem se beneficiar dos resultados aqui apresentados como evidência concreta de que padrões de resiliência precisam ser escolhidos e configurados com muita cautela, de modo a melhor explorar seus ganhos de contenção sem comprometer o tempo de execução das aplicações. Já os pesquisadores podem se beneficiar da nossa metodologia e dados de pesquisa para continuar investigando a relação entre os padrões de resiliência e outros atributos de qualidade relevantes para o desenvolvimento de microsserviços.

6. Trabalhos Relacionados

Apesar do crescente interesse pelo estudo do estilo arquitetural de microsserviços na academia [Di Francesco et al. 2019], até o momento relativamente poucos trabalhos foram publicados com foco em seus padrões de resiliência. Alguns dos trabalhos mais relevantes nesse tema são discutidos a seguir.

[Montesi and Weber 2016] implementaram vários padrões de resiliência no contexto da linguagem de microsserviços Jolie [Jolie Language 2020], incluindo três variações do padrão Circuit Breaker. [Preuveneers and Joosen 2017] propuseram um arcabouço para implementar o padrão Circuit Breaker estendido com a noção de *qualidade de contexto*, de modo a melhorar a resiliência de aplicações distribuídas cientes de contexto. [Aquino et al. 2019] discutiram o uso do padrão Circuit Breaker no contexto de aplicações de Internet das Coisas e avaliaram seus potenciais benefícios em um experimento envolvendo um protótipo de um sistema de sinal de trânsito. Mais recentemente, [Mendonca et al. 2020] propuseram uma abordagem baseada na verificação automática de modelos para analisar o comportamento dos padrões Retry e Circuit Breaker, ambos formalmente especificados como cadeias de Markov de tempo contínuo (*continuous-time Markov chains*—CTMC) [Kwiatkowska et al. 2007]. Um abordagem similar também foi proposta por [Jagadeesan and Mendiratta 2020] para analisar o comportamento do padrão Circuit Breaker. Por fim, [Saleh Sedghpour et al. 2022] estudaram o impacto dos padrões Retry e Circuit Breaker no contexto de uma tecnologia de malha de serviços específica, Istio [Istio.io 2022], com a qual a configuração dos dois padrões é feita pelo operador da malha de serviços, externamente ao código das aplicações.

Este trabalho difere dos trabalhos descritos acima por ser o primeiro a avaliar experimentalmente o impacto de desempenho causado por dois padrões de resiliência, Retry e Circuit Breaker, implementados em duas linguagens de programação, C# e Java, utilizando duas das mais populares bibliotecas de resiliência atuais, Polly e Resilience4j.

Até por terem sido originalmente propostos por profissionais da indústria, padrões de resiliência continuam sendo frequentemente abordados em diversos *blogs* de tecnologia (por exemplo, [Ibryam 2017, Scott 2018, Minkowski 2020]). Embora ofereçam uma rica fonte de informação sobre o uso de padrões de resiliência em produção, esses fóruns, via de regra, limitam-se a discutir exemplos pontuais e boas práticas sobre o uso de alguns padrões. Nesse sentido, este trabalho contribui para avançar o estado-da-prática no uso de padrões de resiliência ao complementar as informações disponíveis publicamente em *blogs* de tecnologia com resultados experimentais obtidos de forma mais rigorosa e sistemática.

7. Conclusão

Este trabalho mostrou experimentalmente que os dois padrões de resiliência avaliados, Retry e Circuit Breaker, podem efetivamente reduzir a contenção por recursos externos de uma aplicação cliente, com um leve a moderado impacto no seu tempo de execução. Outro resultado do trabalho foi que os ganhos de contenção obtidos pelo padrão Retry são significativamente maiores que aqueles obtidos pelo padrão Circuit Breaker, e que a diferença entre os padrões é maior em cenários com baixas a moderadas cargas de trabalho e alta taxas de falha do servidor. Por fim, observou-se que os resultados obtidos com os

dois padrões são, em grande parte, independentes das duas linguagens de programação (C# e Java) e das duas bibliotecas de resiliência (Polly e Resilience4j) utilizadas.

Alguns dos principais tópicos sendo considerados para pesquisas futuras incluem: avaliar outras configurações dos padrões Retry e Circuit Breaker, assim como de outros conhecidos padrões de resiliência, como Fail Fast e Bulkhead, sob maiores cargas de trabalho e maiores taxas de falha do servidor; conduzir novos experimentos envolvendo aplicações de microsserviços mais realistas, contendo múltiplos serviços de *upstream* e *downstream*; permitir a avaliação de padrões de resiliências implementados no contexto de malhas de serviços [Minkowski 2020, Saleh Sedghpour et al. 2022]; e, por fim, evoluir o ambiente de teste desenvolvido no contexto deste trabalho, disponibilizando-o à comunidade de pesquisa na forma de um *benchmark* de código aberto [Aderaldo et al. 2017] voltado para a avaliação de padrões de resiliência para microsserviços.[1]

Referências

Aderaldo, C. M. et al. (2017). Benchmark Requirements for Microservices Architecture Research. In *1st Int. Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13.

App vNext (2022). Polly. `https://github.com/App-vNext/Polly`. [Último acesso em 23 de Abril de 2022].

Aquino, G. et al. (2019). The circuit breaker pattern targeted to future IoT applications. In *Int. Conf. Service Oriented Computing (ICSOC)*, pages 390–396. Springer.

Birolini, A. (2013). *Reliability Engineering: Theory and Practice*. Springer Science & Business Media.

Di Francesco, P., Lago, P., and Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97.

Fowler, M. (2014). CircuitBreaker. `https://martinfowler.com/bliki/CircuitBreaker.html`. [Último acesso em 23 de Abril de 2022].

Ibryam, B. (2017). It takes more than a Circuit Breaker to create a resilient application. `https://developers.redhat.com/blog/2017/05/16/it-takes-more-than-a-circuit-breaker-to-create-a-resilient-application/`. [Último acesso em 23 de Abril de 2022].

Istio.io (2022). The Istio Service Mesh. `https://istio.io/`. [Último acesso em 23 de Abril de 2022].

Jagadeesan, L. J. and Mendiratta, V. B. (2020). When failure is (not) an option: Reliability models for microservices architectures. In *IEEE Int. Symp. Software Reliability Engineering Workshops (ISSREW)*, pages 19–24.

Jamshidi, P. et al. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

[1]Uma nova versão do ambiente de teste utilizado neste trabalho, chamado ResilienceBench, está disponível publicamente em `https://github.com/ppgia-unifor/resiliency-pattern-benchmark`.

Jolie Language (2020). Jolie: The first language for Microservices. https://www.jolie-lang.org/. [Último acesso em 23 de Abril de 2022].

Kwiatkowska, M., Norman, G., and Parker, D. (2007). Stochastic Model Checking. In Bernardo, M. and Hillston, J., editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer.

Mendonca, N. C. et al. (2020). Model-based analysis of microservice resiliency patterns. In *IEEE Int. Conf. Software Architecture (ICSA)*, pages 114–124.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

Microsoft Azure (2017). Resiliency patterns. https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency. [Último acesso em 23 de Abril de 2022].

Minkowski, P. (2020). Circuit breaker and retries on Kubernetes with Istio and Spring Boot. Piotr's TechBlog, https://piotrminkowski.com/2020/06/03/circuit-breaker-and-retries-on-kubernetes-with-istio-and-spring-boot/. [Último acesso em 23 de Abril de 2022].

Montesi, F. and Weber, J. (2016). Circuit breakers, discovery, and api gateways in microservices. *arXiv preprint arXiv:1609.05830*.

Mozilla (2012). Vaurien, the Chaos TCP Proxy. https://vaurien.readthedocs.io/. [Último acesso em 23 de Abril de 2022].

Nygard, M. (2007). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

Postman (2017). httpbin(1): HTTP Request & Response Service. https://github.com/postmanlabs/httpbin. [Último acesso em 23 de Abril de 2022].

Preuveneers, D. and Joosen, W. (2017). QoC^2 Breaker: intelligent software circuit breakers for fault-tolerant distributed context-aware applications. *Journal of Reliable Intelligent Environments*, 3(1):5–20.

Resilience4j (2021). Fault tolerance library designed for functional programming. https://github.com/resilience4j/resilience4j. [Último acesso em 23 de Abril de 2022].

Rosenthal, C. et al. (2017). *Chaos Engineering: Building Confidence in System Behavior through Experiments*. O'Reilly.

Saleh Sedghpour, M. R., Klein, C., and Tordsson, J. (2022). An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In *ACM/SPEC Int. Conf. Performance Engineering (ICPE)*, pages 17–27.

Scott, C. (2018). Designing Resilient Systems: Circuit Breakers or Retries? (Part 1). Grab Tech Blog, https://engineering.grab.com/designing-resilient-systems-part-1. [Último acesso em 23 de Abril de 2022].