Client Resiliency Patterns with Spring Cloud and Netflix Hystrix

Chapter Content

1. What are client-side resiliency patterns

2. An example scenario

3. Introducing Hystrix

- When it comes to building resilient systems, most software engineers only take into account the **complete failure** of a piece of infrastructure or a key service. They focus on building redundancy into each layer of their application using techniques such as clustering key servers, load balancing between services, and segregation of infrastructure into multiple locations

- When a service crashes, it's easy to detect that it's no longer there, and the application can route around it. However, when a service is running **slow**, detecting that **poor performance** and routing around it is extremely difficult

- What's insidious about problems caused by poorly performing remote services is that they can trigger a **cascading** effect that can ripple throughout an entire application ecosystem

1. What are client-side resiliency patterns

- Client resiliency software patterns are focused on protecting a remote resource’s (another microservice call or database lookup) client from crashing when the remote resource is failing because that remote service is throwing errors or performing poorly.

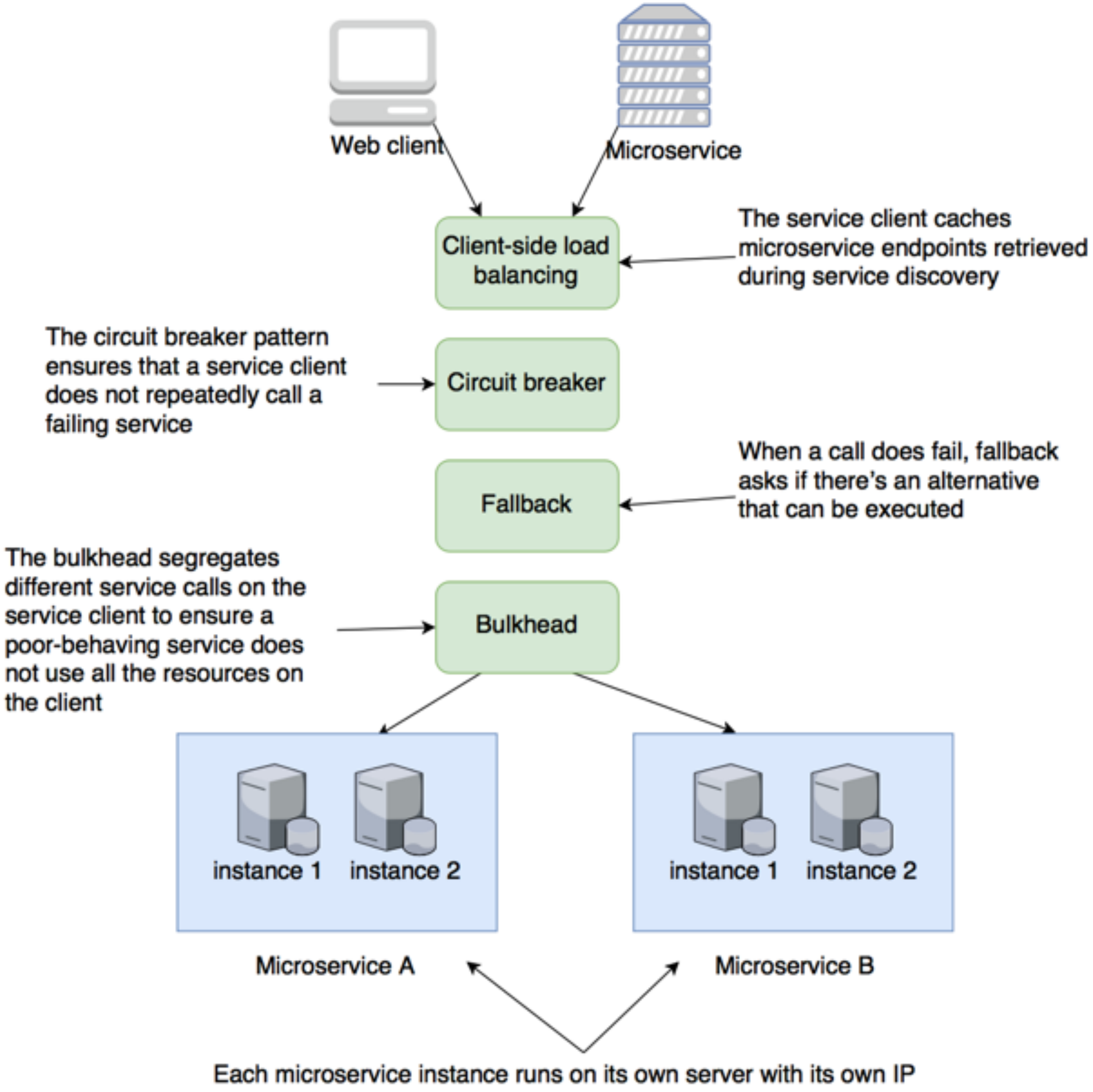
- The goal is to

    1. “fail fast”

    2. not consuming valuable resources such as database connections and thread pools,

    3. prevent from spreading “upstream” to consumers of client

- There are four client resiliency patterns:
    1. Client-side load balancing
    2. Circuit breakers
    3. Fallbacks
    4. Bulkheads



Web client

Microservice

Client-side load balancing

The service client caches microservice endpoints retrieved during service discovery

The circuit breaker pattern ensures that a service client does not repeatedly call a failing service

Circuit breaker

Fallback

When a call does fail, fallback asks if there's an alternative that can be executed

The bulkhead segregates different service calls on the service client to ensure a poor-behaving service does not use all the resources on the client

Bulkhead

instance 1

instance 2

Microservice A

instance 1

instance 2

Microservice B

Each microservice instance runs on its own server with its own IP

- Client-side load balancing
  - Client-side load balancing involves having the client **look up** all of a service's **individual** instances from a service **discovery agent** (like Netflix Eureka) and then **caching** the physical location of said instances.
  - Whenever a service consumer needs to call that service instance, the client-side load balancer will return a location from the pool of service locations it's maintaining
  - Because the client-side load balancer sits between the service client and the service consumer, the load balancer can detect if a service instance is throwing errors or behaving poorly. If the client-side load balancer detects a problem, it can remove that service instance from the pool of available service locations and prevent any future service calls from hitting that service instance
  - This is exactly the behavior that Netflix's **Ribbon** libraries provide out of the box with no extra configuration

- Circuit breaker

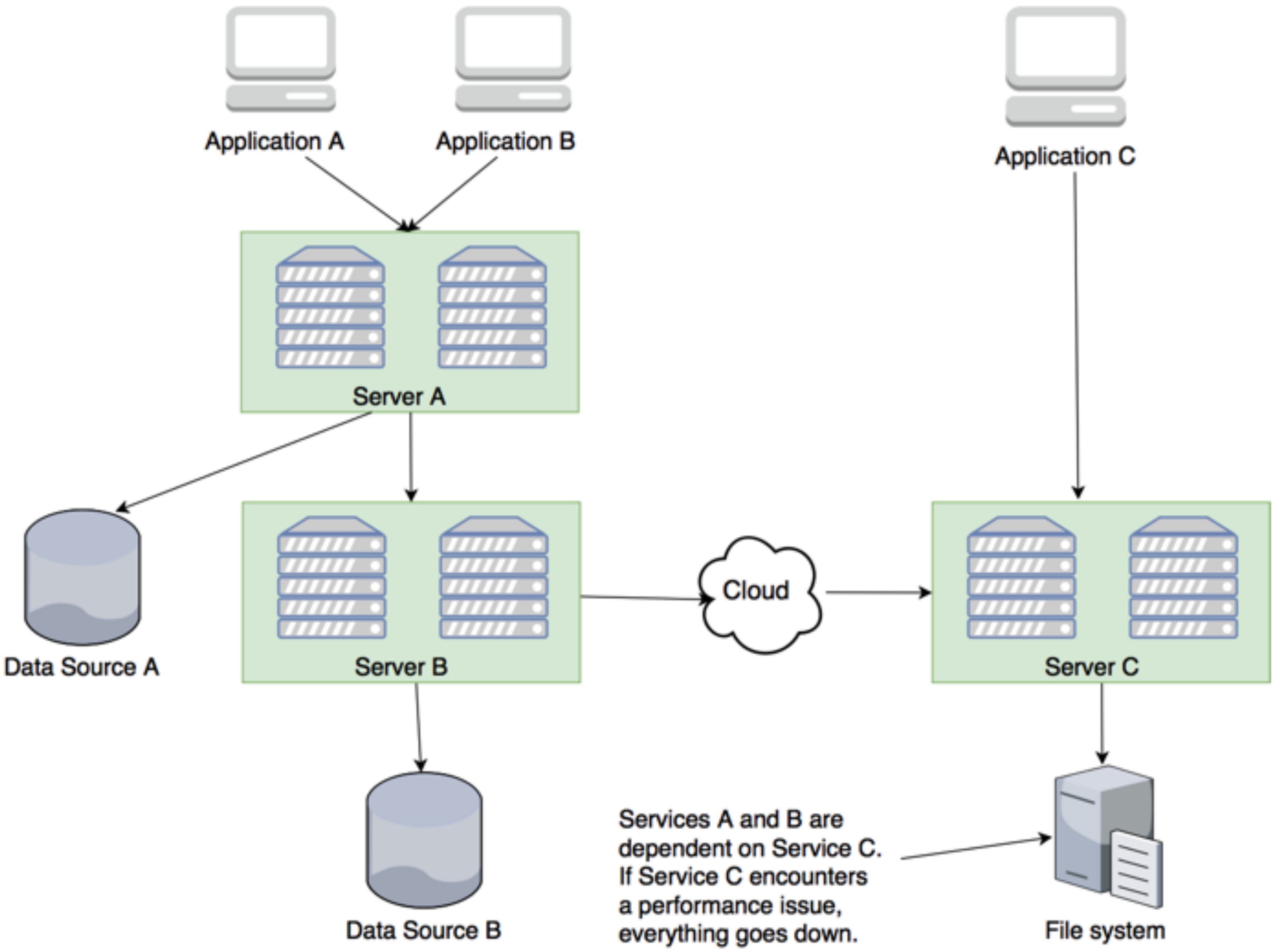
  - When a remote service is called, the circuit breaker will **monitor** the call. If the call takes too long, the circuit breaker will intercede and kill the call

  - In addition, the circuit breaker will monitor all calls to a remote resource and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.

- Fallback processing
  - With the fallback pattern, when a remote service call fails, rather than generating an exception, the service consumer will execute an **alternative** code path and try to carry out an action through another means
  - This usually involves looking for data from another data source or queueing the user's request for future processing. The user's call will not be shown an exception indicating a problem, but they may be notified that their request will have to be fulfilled at a later date

- Bulkheads
    - By using the bulkhead pattern, you can break the calls to remote resources into their **own thread pools** and reduce the risk that a problem with one slow remote resource will take down the entire application
    - The thread pools act as the bulkheads for your service. Each remote resource is **segregated** and assigned to the thread pool. If one service is responding slowly, the thread pool for that one type of service call will become saturated and stop processing requests. Service calls to other services won't become saturated because they're assigned to other thread pools

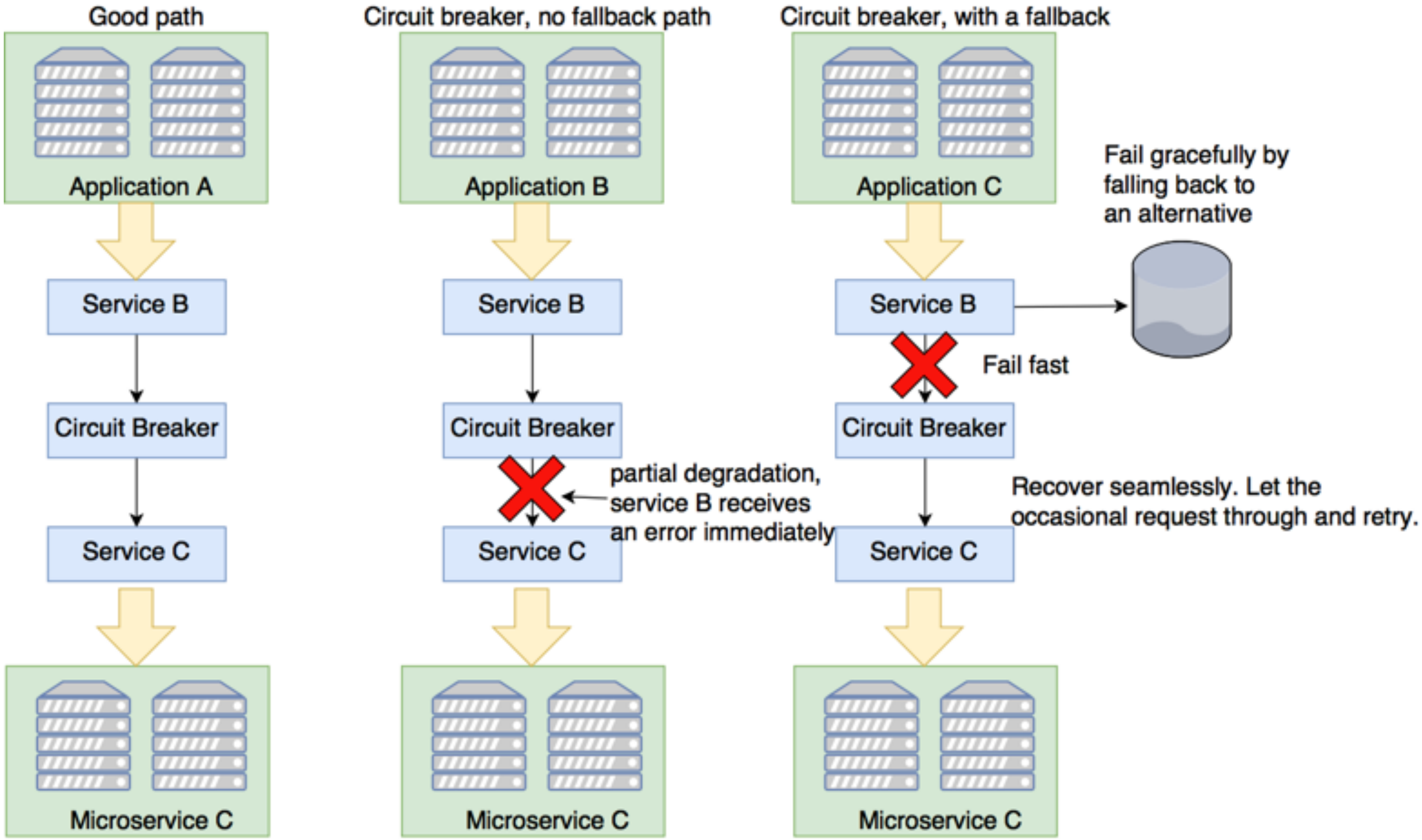
2. An Example Scenario



Services A and B are
dependent on Service C.
If Service C encounters
a performance issue,
everything goes down.

- Service B never expects slowdowns with calls to service C. When Service C starts running slowly, not only does the thread pool for requests to Service C start backing up, the number of database connections in the service container's connection pools become exhausted because these connections are being held open because the calls out to Service C never complete

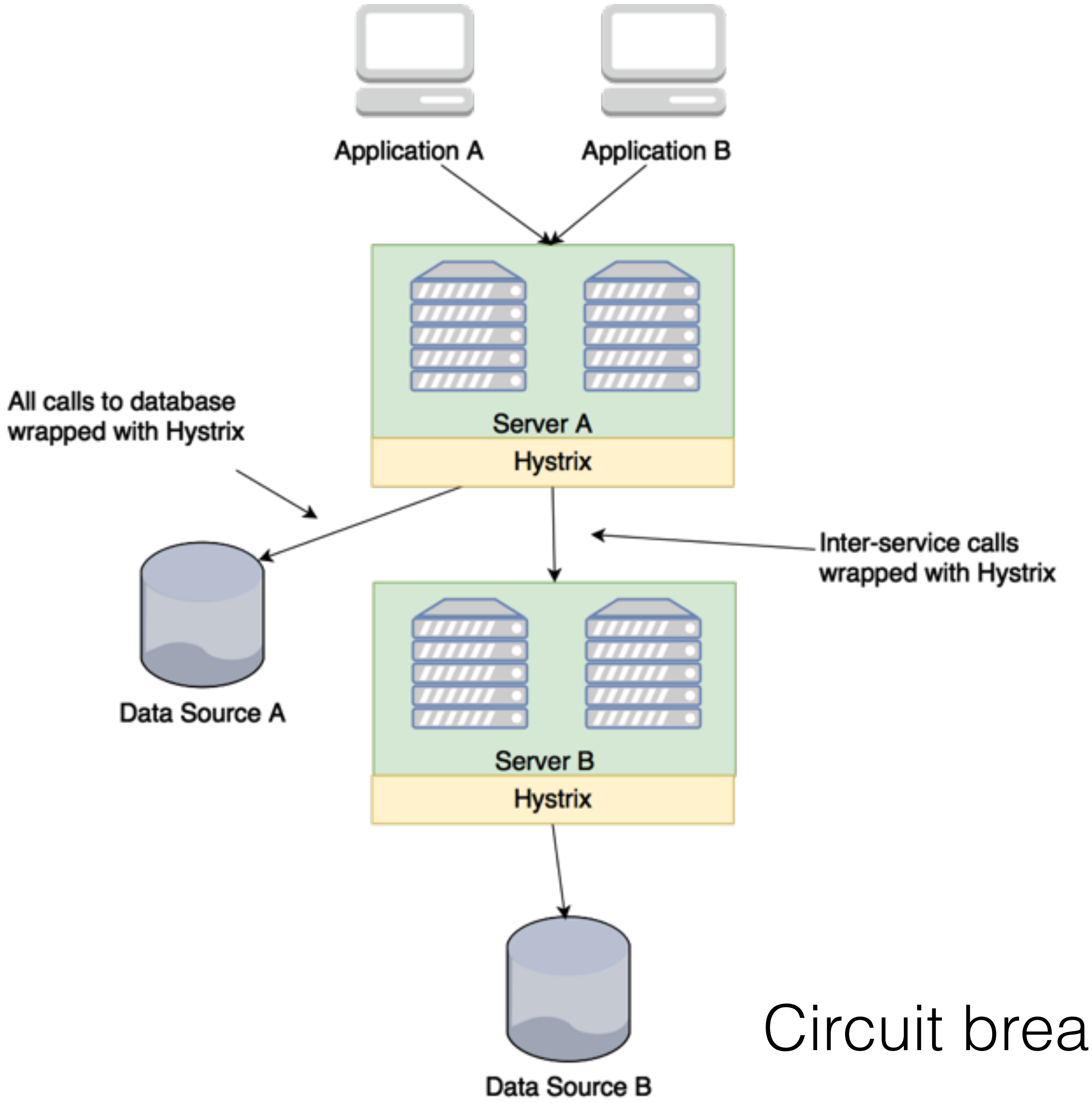
- Then, service A starts running out of resources because it's calling service B, which is running slow because of service C. Eventually, all three applications stop responding because they run out of resources while waiting for requests to complete.


Good path
Circuit breaker, no fallback path
Circuit breaker, with a fallback
Application A
Application B
Application C
Fail gracefully by
falling back to
an alternative
Service B
Service B
Service B
Fail fast
Circuit Breaker
Circuit Breaker
Circuit Breaker
partial degradation,
service B receives
an error immediately
Recover seamlessly. Let the
occasional request through and retry.
Service C
Service C
Service C
Microservice C
Microservice C
Microservice C

3. Introducing Hystrix

- Building implementation of circuit breaker, fallback and bulk patterns requires tremendous knowledge of threads and thread management

- Netflix Hystrix library provides a battle-tested library, which is used daily in Netflix’s microservice architecture


Application A
Application B
Server A
Hystrix
All calls to database wrapped with Hystrix
Inter-service calls wrapped with Hystrix
Data Source A
Server B
Hystrix
Data Source B

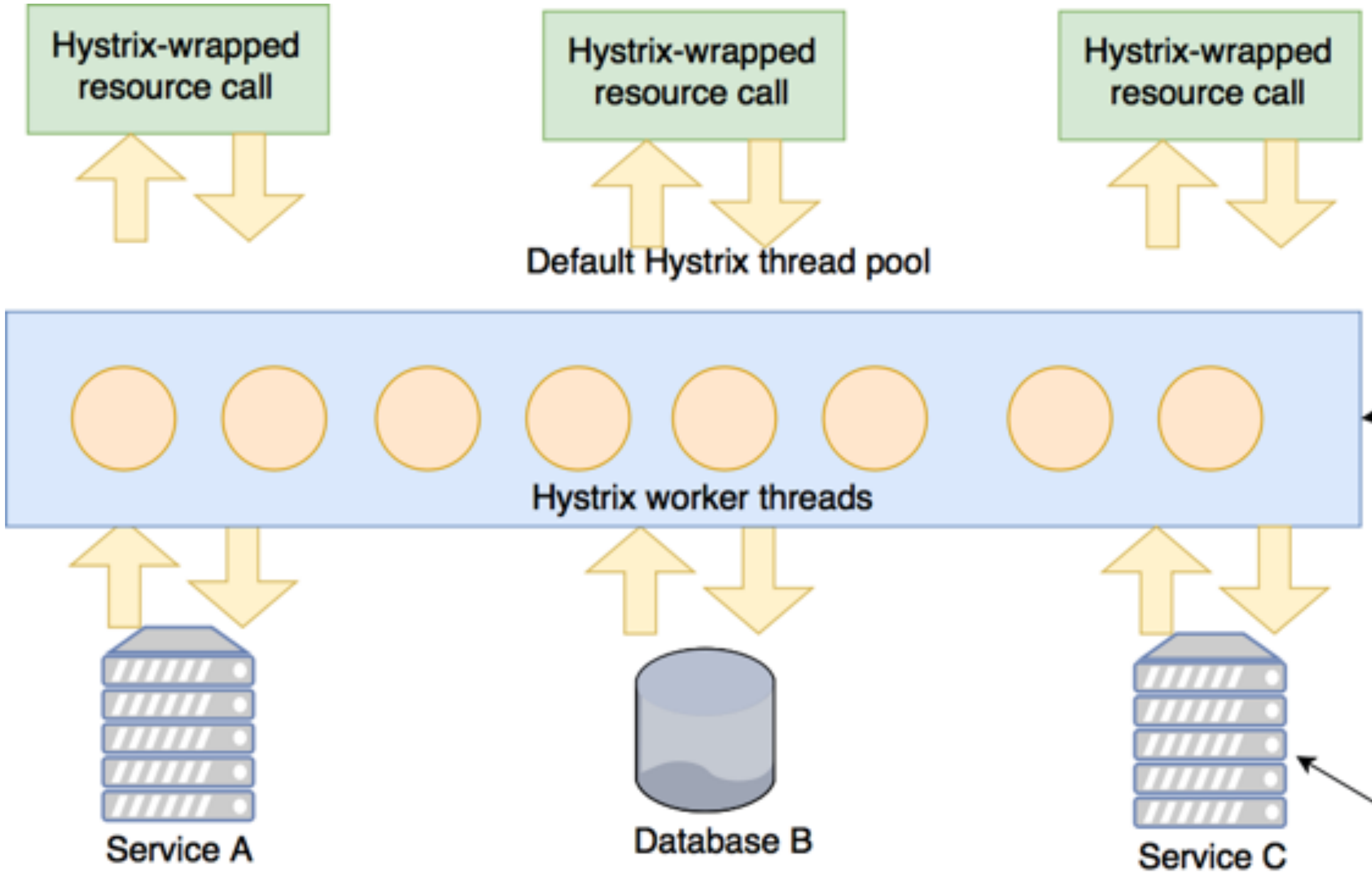
Circuit breaker pattern

- Bulkhead pattern with Hystrix

    - In a microservice-based application, you’ll often need to call multiple microservices to complete a particular task. Without using a bulkhead pattern, the default behavior for these calls is that the calls are executed using the same threads that are used for handling requests for entire Java container

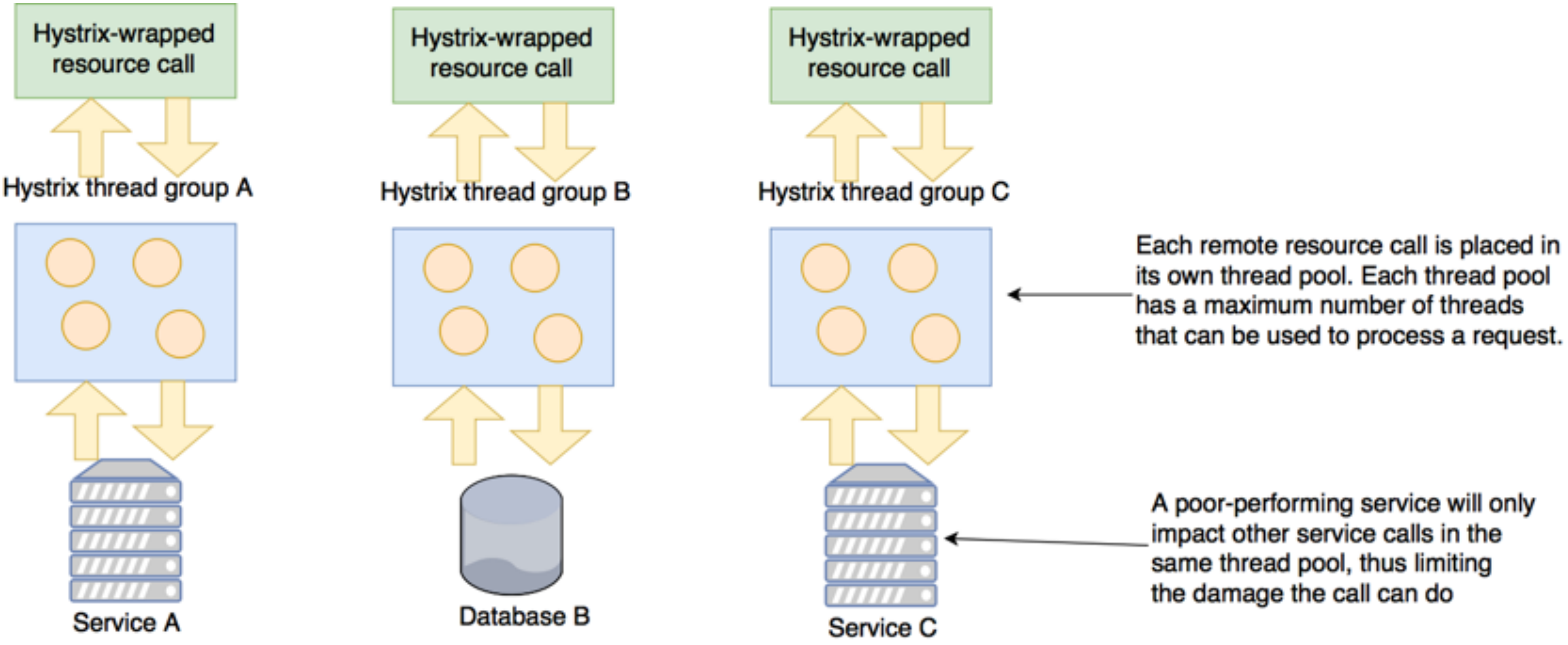
    - Hystrix uses a thread pool to delegate all requests for remote services.

    - By default, all Hystrix commands will share the same thread pool to process requests. This thread pool will have 10 threads in it to process remote service calls and those remote service calls could be anything, including REST call, database call and etc.


Hystrix-wrapped resource call
Hystrix-wrapped resource call
Hystrix-wrapped resource call
Default Hystrix thread pool
Hystrix worker threads
Service A
Database B
Service C

All remote resource calls are in a single shared thread pool.

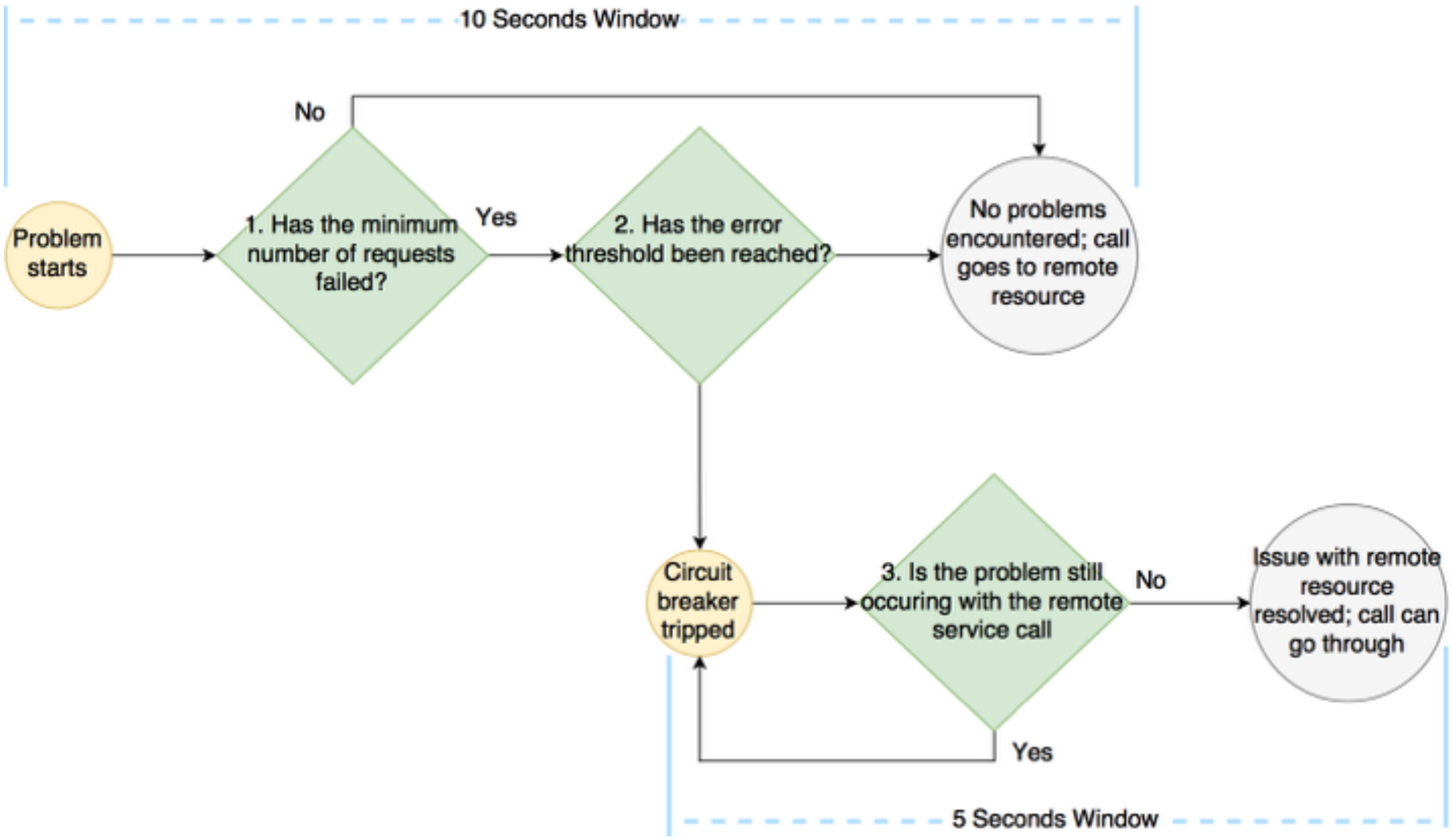
A single slow-performing service can saturate the Hystrix thread pool and cause resource exhaustion in the Java container hosting the service.



Each remote resource call is placed in its own thread pool. Each thread pool has a maximum number of threads that can be used to process a request.

A poor-performing service will only impact other service calls in the same thread pool, thus limiting the damage the call can do

Bulkhead pattern


10 Seconds Window
No
Problem starts
1. Has the minimum number of requests failed?
Yes
2. Has the error threshold been reached?
No problems encountered; call goes to remote resource
Circuit breaker tripped
3. Is the problem still occuring with the remote service call
No
Issue with remote resource resolved; call can go through
Yes
5 Seconds Window

Fine tuning Hystrix

- Thread context and Hystrix
  - When an Hystrix command is executed, it can be run with two different isolation strategies: THREAD and SEMAPHORE. By default, THREAD isolation is used
  - In THREAD mode, pools are isolated. This means Hystrix can interrupt the execution of a thread without worrying bout interrupting any other activity associated with the parent thread doing the original invocation
  - With SEMAPHORE-based isolation, Hystrix manages the distributed call without starting a new thread and will interrupt the parent thread if the call times out.

- Hystrix custom concurrency strategy
  - Hystrix allows you to define a custom concurrency strategy that will wrap your Hystrix calls and allows you to inject any additional parent thread context into the threads managed by the Hystrix command
  - To implement a custom HystrixConcurrencyStrategy you need to carry out three actions:
    1. Define your custom Hystrix Concurrency Strategy class
    2. Define a Java callable class to inject the UserContext into the Hystrix
Command
    3. Configure Spring Cloud to use your custom Hystrix Concurrency Strategy