

# Resilience Engineering in a microservice landscape

Maurice Zeijen

# Let me introduce myself

**Maurice Zeijen**

Java developer for over 10 years

Lead Software Architect @ bol.com

**bol.com** 



# This is bol.com

bol.com is one of the most popular webshops in The Netherlands and Belgium since 1999.

**Visits per month:** > 28 million

**Products:** > 14 million

**Active Customers:** > 7 million

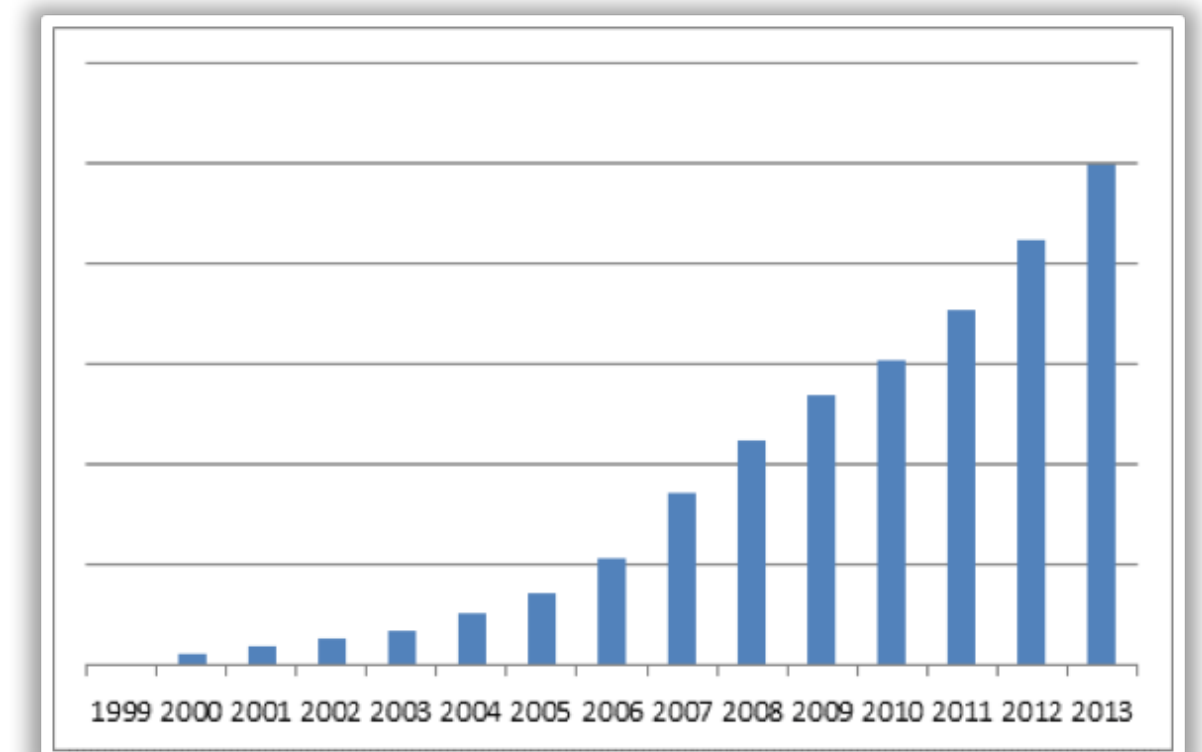
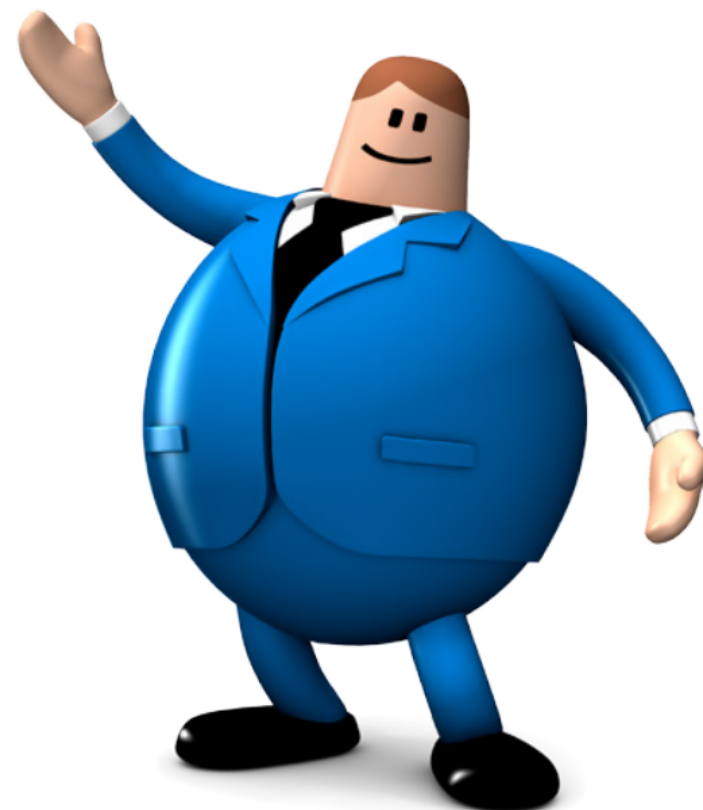
**Partners:** > 16.500

**Private sellers:** > 180.000

**Employees:** > 1.200

**IT engineers:** > 350

**bol.com** 



# What am I going to talk about?

- What is resilience?
- Why do we need resilience?
- Implementing resilience with Hystrix
- Resilience techniques
- Tips & tricks
- Operations
- Experiences @ bol.com
- Wrapup





## What is resilience?

//

***The ability of a substance or object to spring back into shape.***

***The capacity to recover quickly from difficulties.***

//

— Merriam Webster



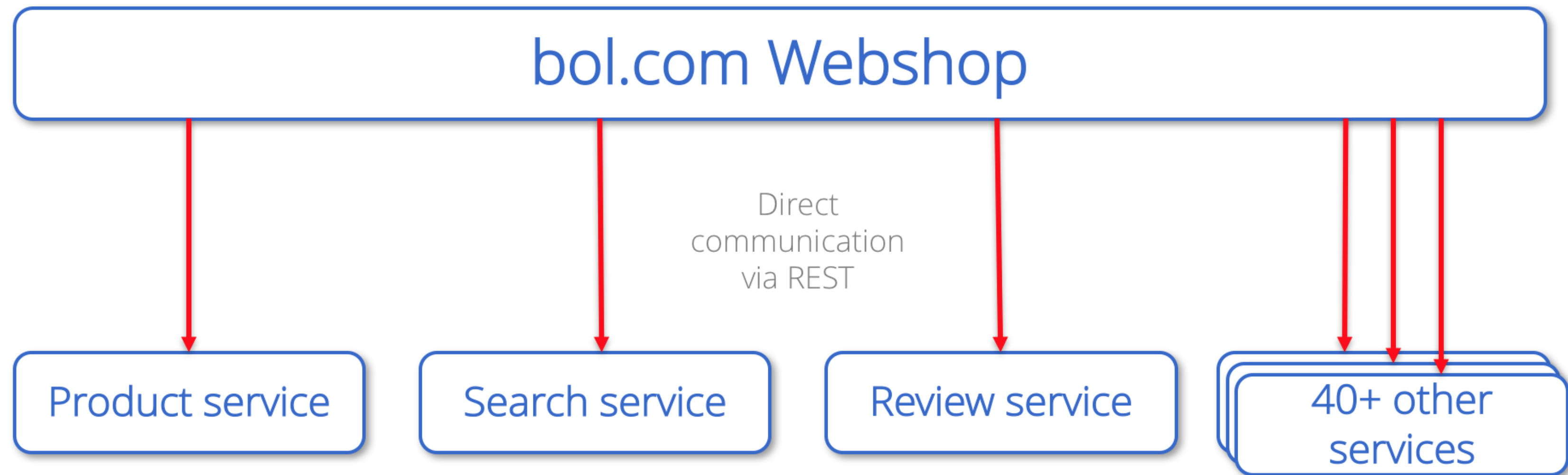
# Resilience in IT systems

The ability of a system to handle unexpected situations:

- without the user noticing it
- with a graceful degradation of service
- automatically recovering, as if it never happened



# Resilience in a (micro)service landscape





## Why do we need resilience?

Failures in today's complex, distributed, interconnected systems are not the exception.

**They are the normal case.**

# Complexity at bol.com

+50 User facing applications

+120 Backend services

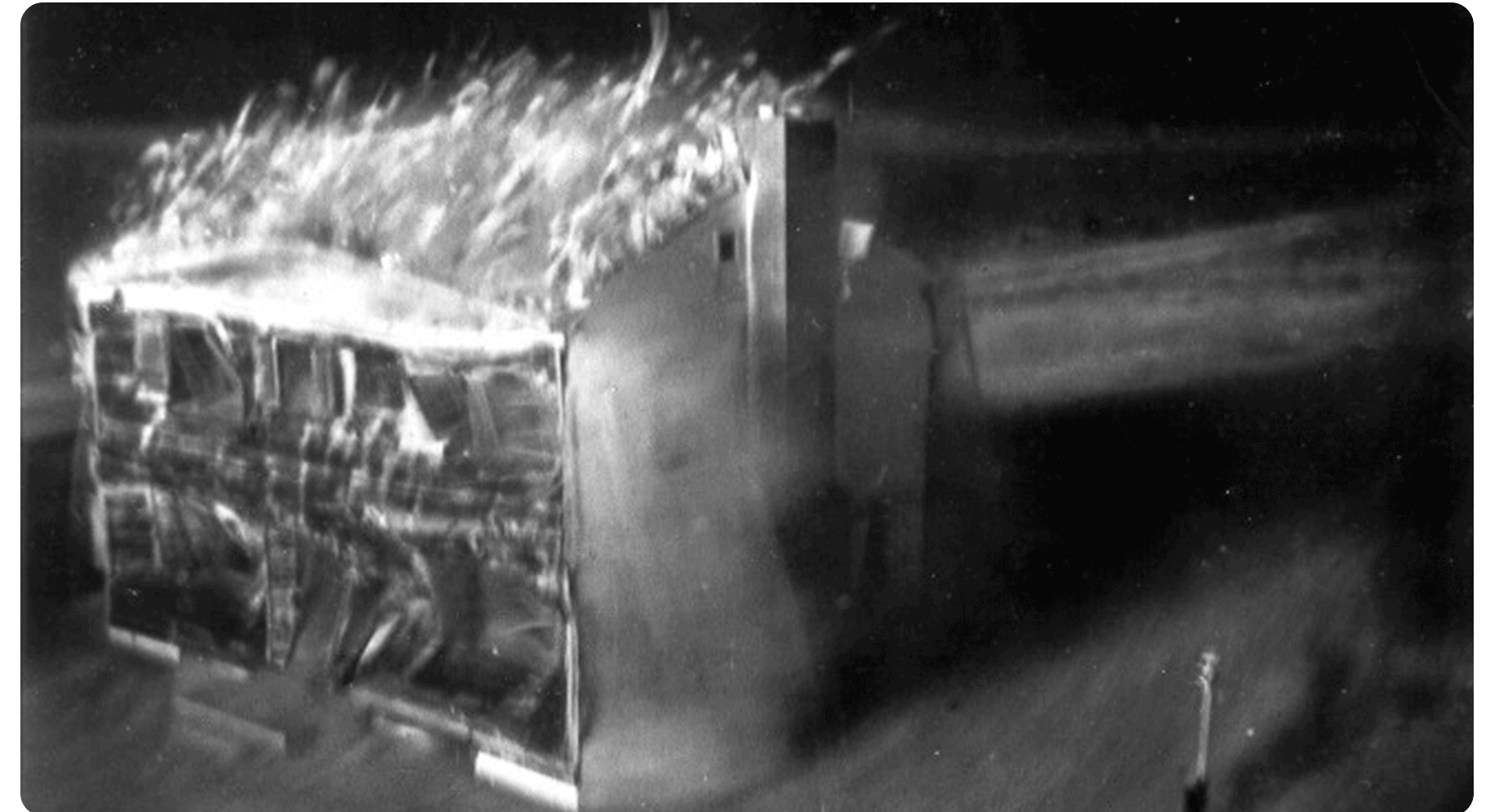
60 IT Teams

*All interconnected in some way or another...*



# What could possible go wrong?

- Services fail and go down.
- New versions of services can have bugs or are not backwards compatible.
- Network connections degrade or fail.
- Client libraries have bugs and misbehave.
- Services become slow.





# Latency: Destroyer of distributed systems



**Latency can cause cascading failures across multiple systems, even if they are only loosely connected.**





We need to be resilient!

Hurricane Ike, september 2008, Texas  
Only one house survived because it was build for hurricane conditions.



# Implementing resilience with Hystrix



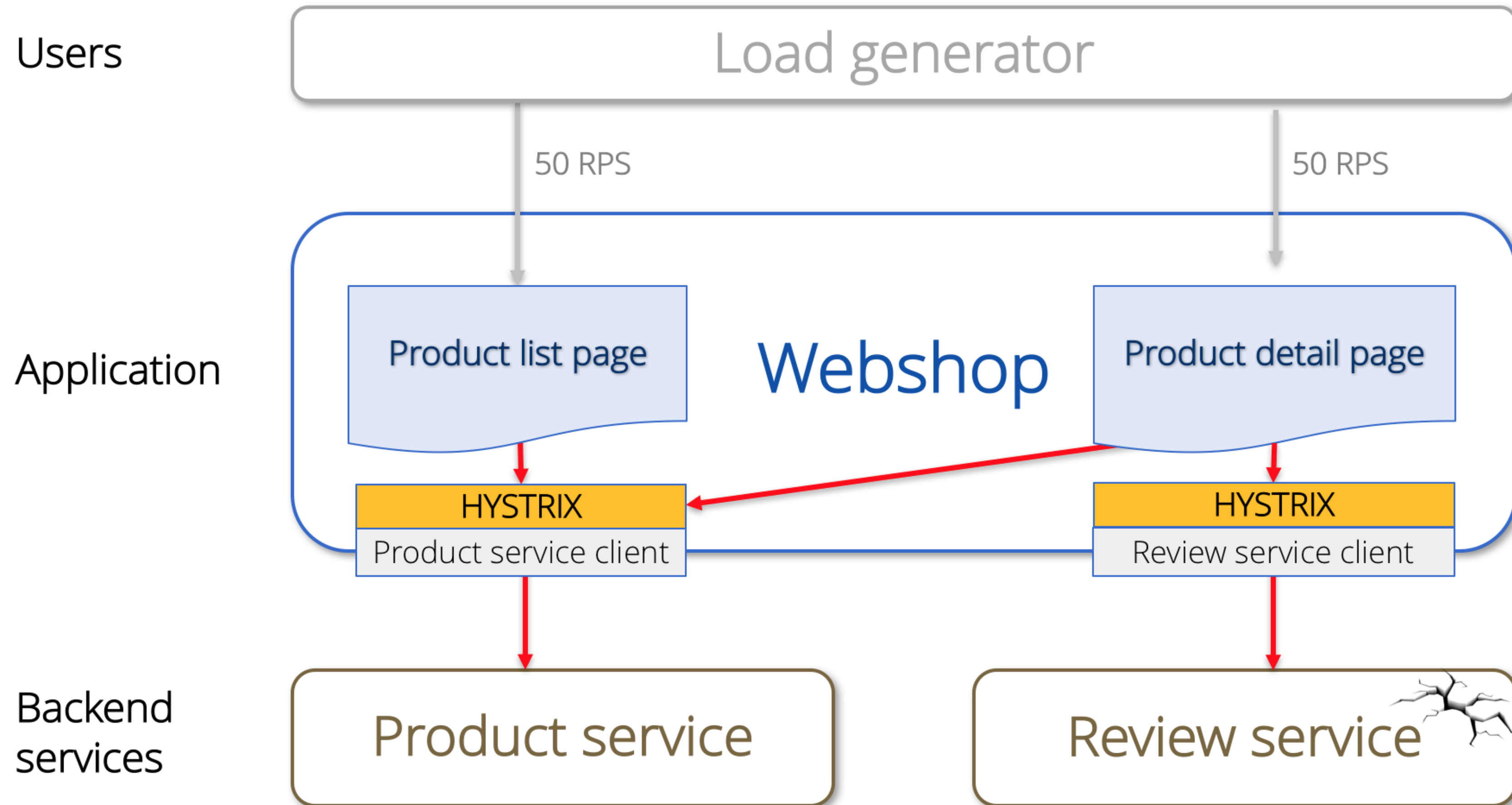
## Hystrix:

- is a open-source Java latency and fault tolerance library from Netflix.
- is designed to isolate points of access to remote systems and libraries.
- provides the means to handle failures gracefully.
- measures everything you want to know of it's execution.
- is relatively easy to use.

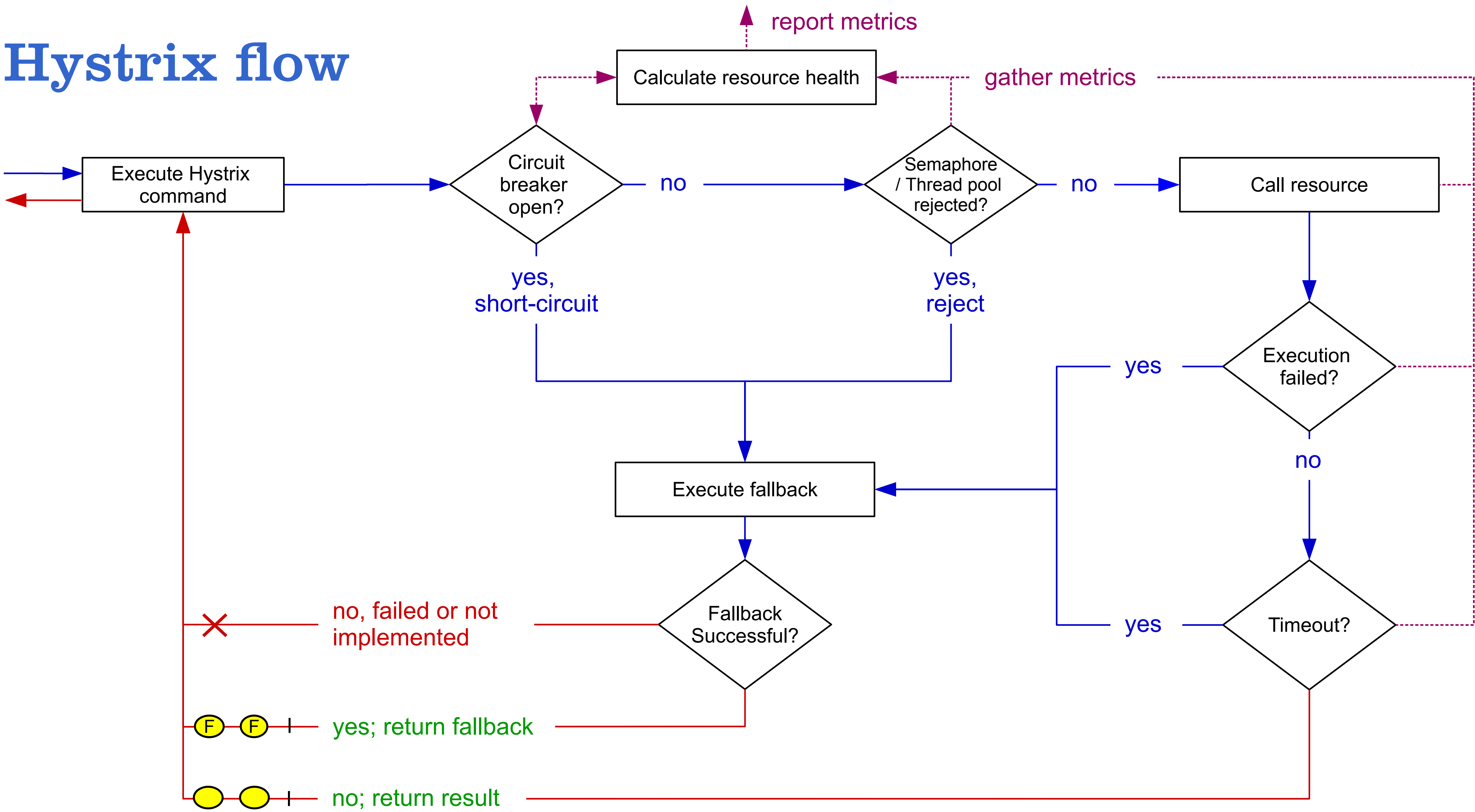
There are similar libraries for other non-jvm languages.



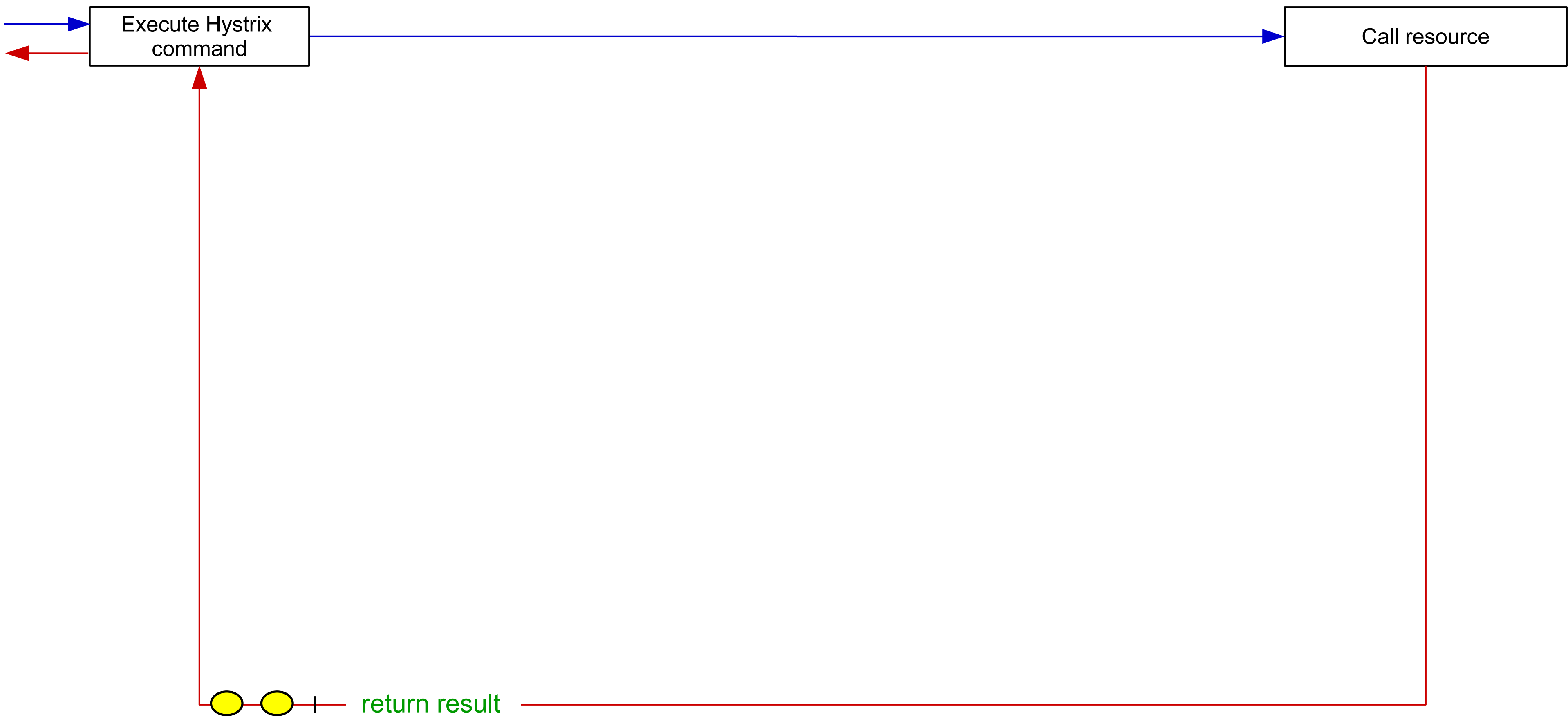
# Where to apply the resilience techniques with Hystrix



# Hystrix flow



# Hystrix command





# Implementing a Hystrix command

```
public class GetReviewsCommand extends HystrixCommand<List<Review>> {

    private final ReviewClient reviewClient;
    private final String productId;

    public GetReviewsCommand(ReviewClient reviewClient, String productId) {
        super(
            Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("review"))
                .andCommandKey(HystrixCommandKey.Factory.asKey("GetReviews"))
        );
        this.reviewClient = reviewClient;
        this.productId = productId;
    }

    @Override
    protected List<Review> run() throws Exception {
        return reviewClient.getReviews(productId);
    }
}
```

- You need to implement a command for every resource endpoint.
- Non-blocking clients are also supported by implementing the HystrixObservableCommand.

# Execution a command

```
// Synchronous, blocking
List<Review> reviews = new GetReviewsCommand(client, productId).execute();
```

```
// Asynchronous, blocking, with a Future
Future<List<Review>> reviews = new GetReviewsCommand(client, productId).queue();
/// Do some other things
List<Review> reviews = reviewsFuture.get();
```

```
// Asynchronous, blocking, with Observables
Observable<List<Review>> reviews = new GetReviewsCommand(client, productId).observe();
reviewsObservable.subscribe(reviews -> {
    // do something with the reviews
});
```

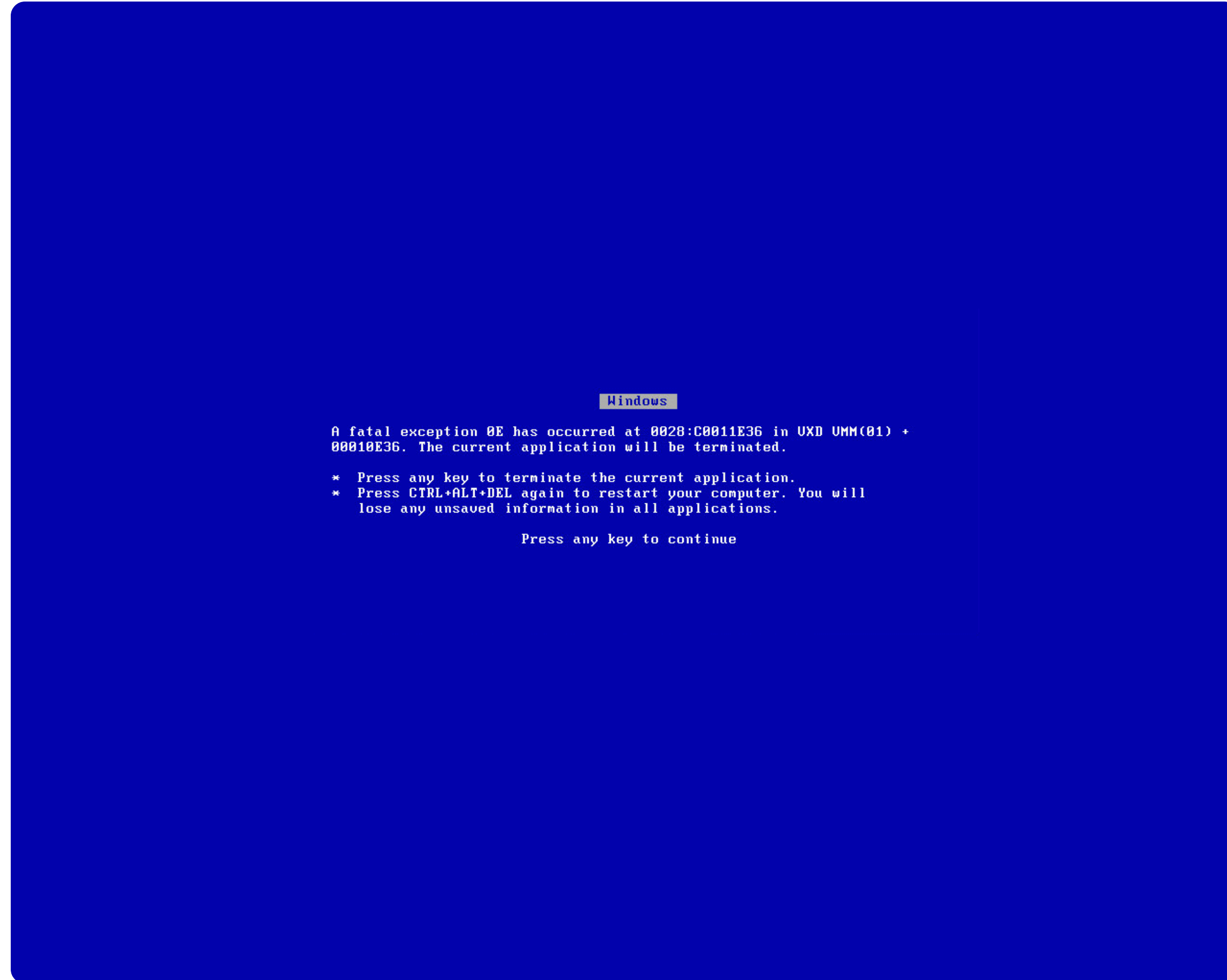
```
// Asynchronous, non-blocking stream with Observables
Observable<Review> reviews = new GetReviewStreamCommand(client, productId).observe();
reviewsObservable.subscribe(review -> {
    // do something with the each review
});
```

- Create a new command for every call.
- Only RxJava 1 is supported.
- CompletableFuture is not supported.
- Backpressure is not supported.

# Resilience techniques

- Fallback
- Timeouts
- Bulkheading & load shedding
- Health insights
- Circuit breaker

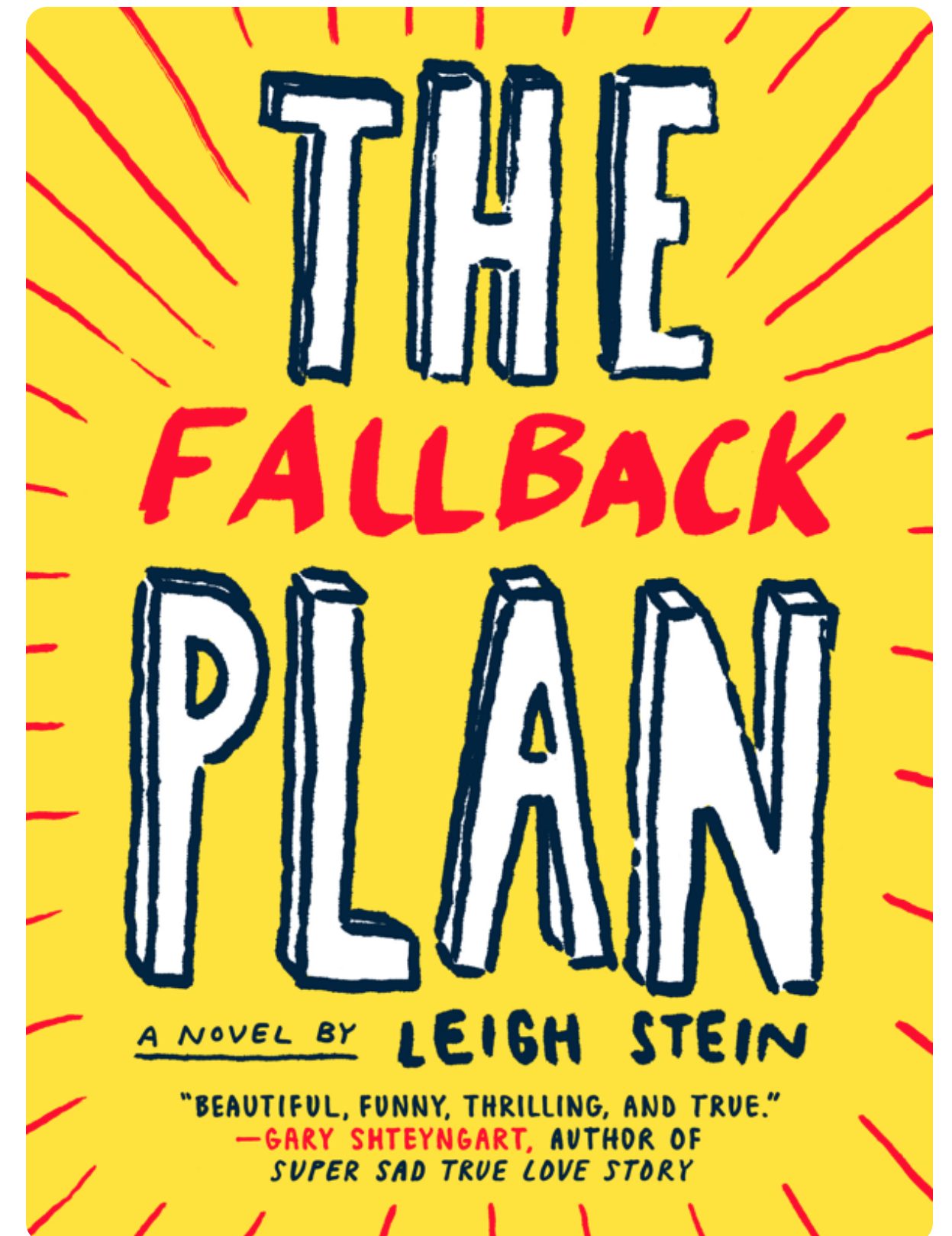
# Graceful degradation with the fallback



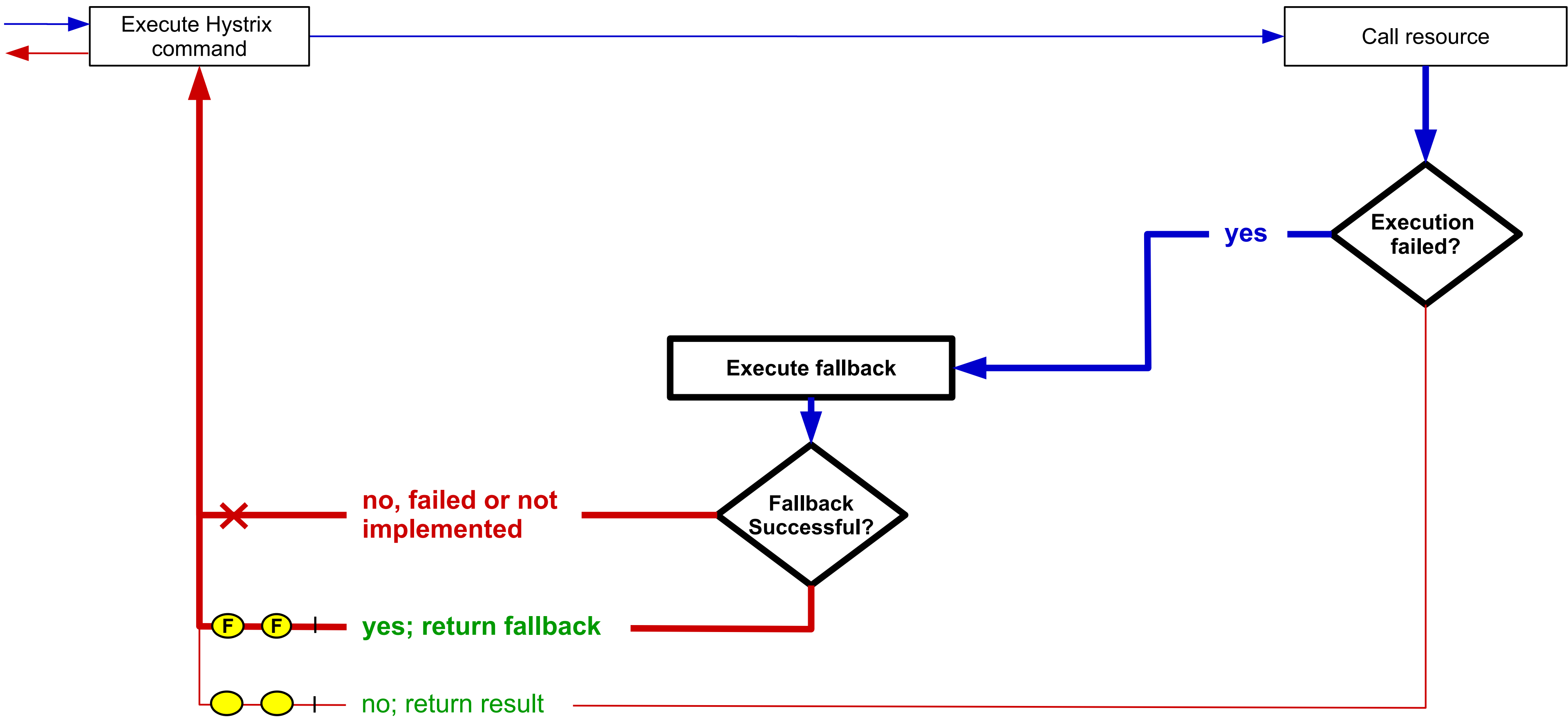
# Goal of the fallback

Giving your user the best possible experience when your system is having issues.

- Hide the feature → Fail silent
- Use a default → Static fallback
- Use an alternative → Stubbed fallback or fallback via network
- If a fallback makes no sense → Fail fast



# Failures & fallback





# Stubbed fallback example

The product list title is created by a service based on the product category and selected filters.

Verkopen Zakelijk Cadeaubon Inloggen Bestelstatus Lijstjes Klantenservice

**bol.com**

Waar ben je naar op zoek? Computer Zoeken

Kies een categorie Word lid van Select Moederdag Outlet Aanbiedingen

Gratis verzending vanaf 20 euro, gratis retourneren, bezorging waar en wanneer je wilt met **Select** artikelen\*

Computer > **Laptops**

Type gebruik

Uitleg & kiezen

☐ Dagelijks gebruik (386)

☐ Zakelijk gebruik (331)

☒ **Films en series bekijken**

☐ Gaming (257)

☐ Foto- en videobewerking (125)

☐ Studeren (basisgebruik) (111)

☐ Studeren (grafisch krachtig) (3)

**Laptop voor Films en Series**

Wil je graag films en series kijken op je laptop? De laptops uit deze categorie beschikken over een Full HD-scherm, zodat films en series in hoge kwaliteit worden weergegeven. Bovendien is het ook prettig voor het bekijken van foto's of websites. Deze groep laptops is prima geschikt voor dagelijks gebruik.

315 resultaten

Sorteer op: Beoordeling

Gekozen filters: Films en series bekijken

Wis alle filters

5

# Stubbed fallback example

Service fails: fallback the title to just the product category name.

Verkopen Zakelijk Cadeaubon

Inloggen Bestelstatus Lijstjes Klantenservice

**bol.com**

Waar ben je naar op zoek?

Computer Zoeken

0

Kies een categorie

Word lid van Select Moederdag Outlet Aanbiedingen

Gratis verzending vanaf 20 euro, gratis retourneren, bezorging waar en wanneer je wilt met **Select** artikelen\*

Computer > Laptops

Type gebruik

Uitleg & kiezen

☐ Dagelijks gebruik (386)

☐ Zakelijk gebruik (331)

☒ Films en series bekijken

☐ Gaming (257)

☐ Foto- en videobewerking (125)

☐ Studeren (basisgebruik) (111)

☐ Studeren (grafisch krachtig) (3)


Laptops


315 resultaten


Sorteer op: Beoordeling

Gekozen filters: Films en series bekijken

Wis alle filters







# When a fallback probably makes no sense

Some cases when you probably don't want a fallback:

- **For write operations**

If a write fails, you probably want the failure to propagate back to the caller.

- **For batch or offline operations**

If your Hystrix command is starting a batch job or some other offline computation, it's usually more appropriate to propagate the error back to the caller.

- **Within backend services**

Often it is not possible choose an appropriate fallback within a backend service. Those commands should fail fast, return a decent error message and let the user-facing application provide the fallback, or they apply a fallback and add meta data to the response that the fallback was applied.

# Implementing the fallback within the HystrixCommand

```
public class GetReviewsCommand extends HystrixCommand<List<Review>> {  
  
    // Fields and constructor  
  
    @Override  
    protected List<Review> run() throws Exception {  
        return reviewClient.getReviews(productId);  
    }  
  
    @Override  
    protected List<Review> getFallback() {  
        return Collections.emptyList();  
    }  
  
}
```

# Other forms of a fallback or when not fallback

If no fallback is implemented or the fallback throws an exception then the Command itself will throw a `HystrixRuntimeException`.

```
try {  
    Observable<Review> reviews = new GetReviewsCommand(client, productId).execute();  
} catch (HystrixRuntimeException e) {  
    Exception actualException = e.getCause();  
  
    // Do something usefull with the real exception.  
}
```

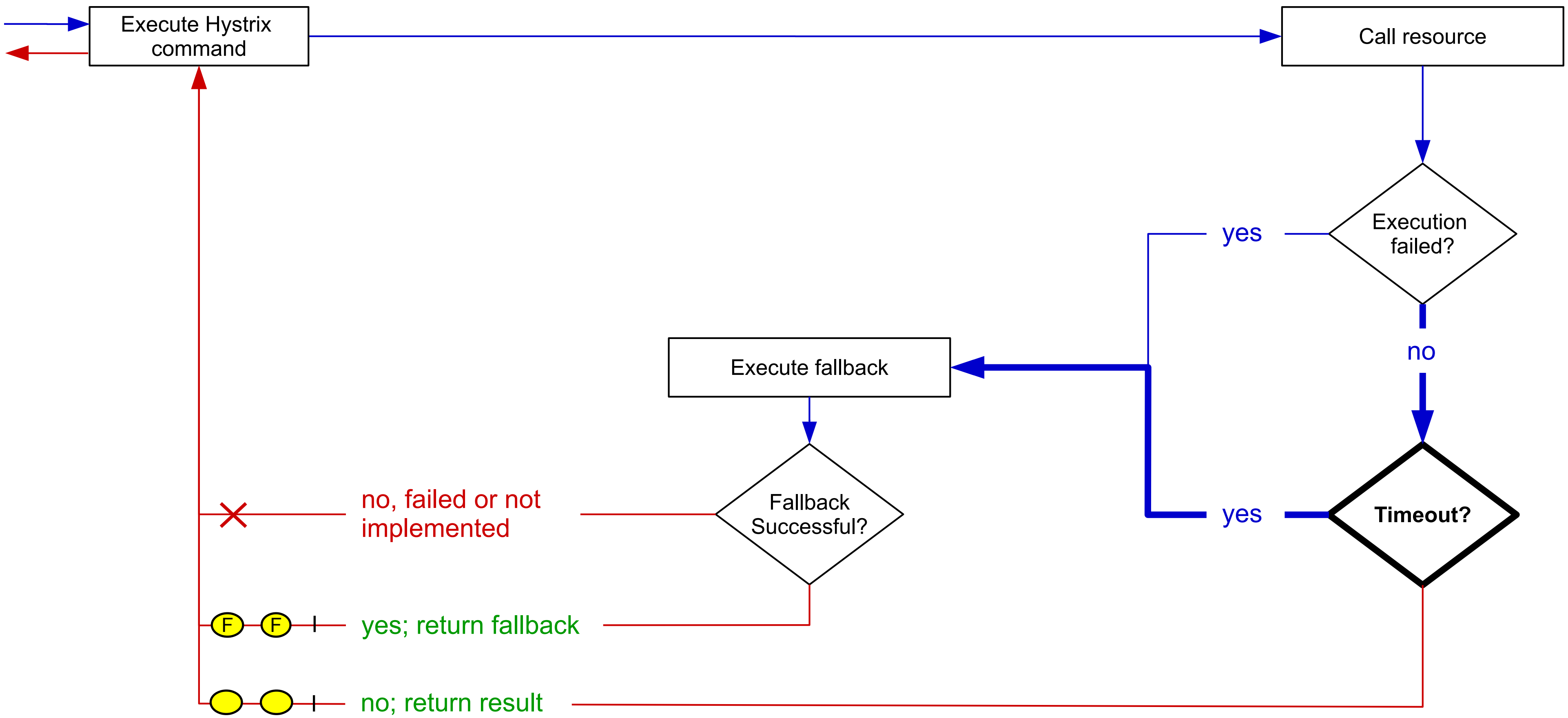


# Protecting against latency

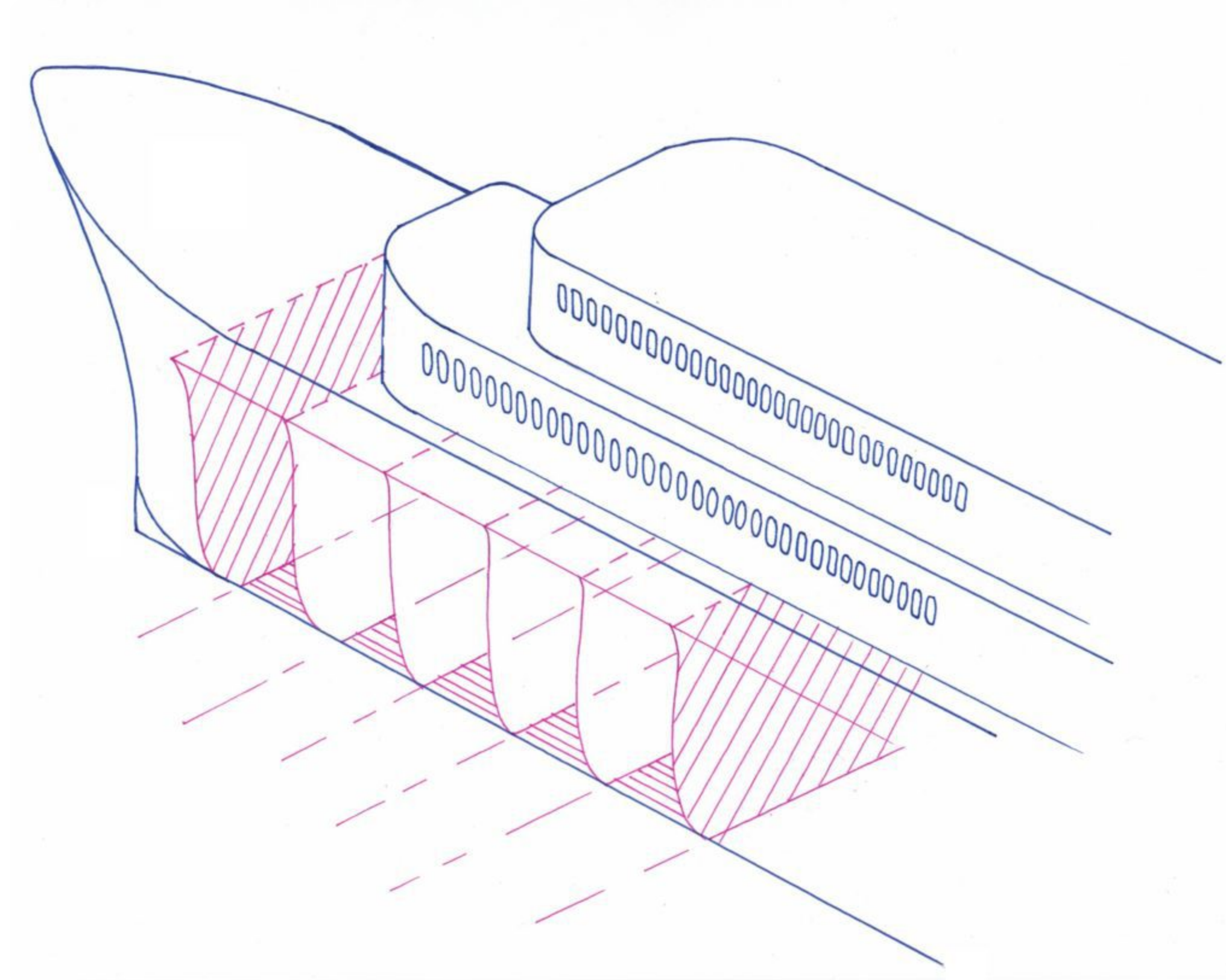




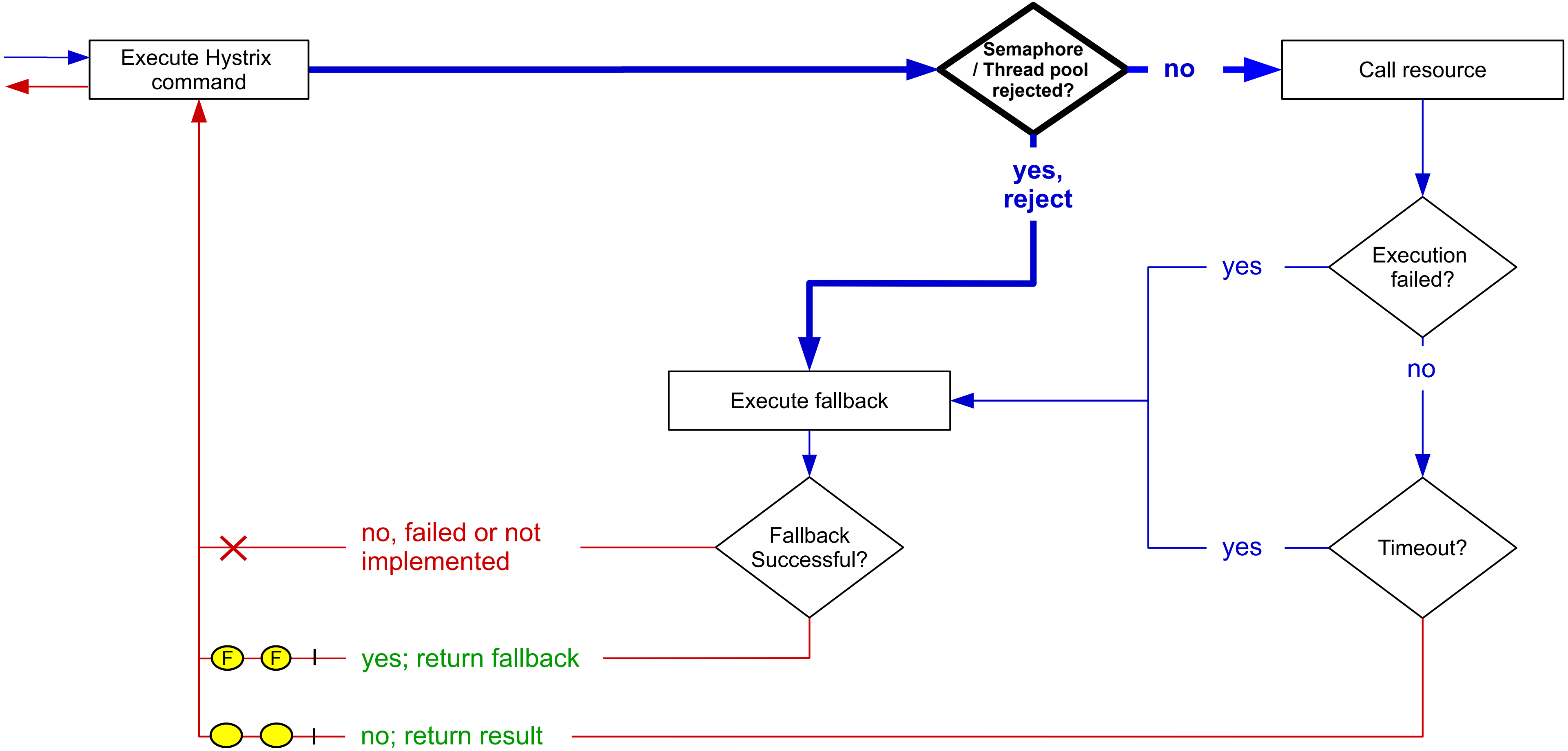
# Latency



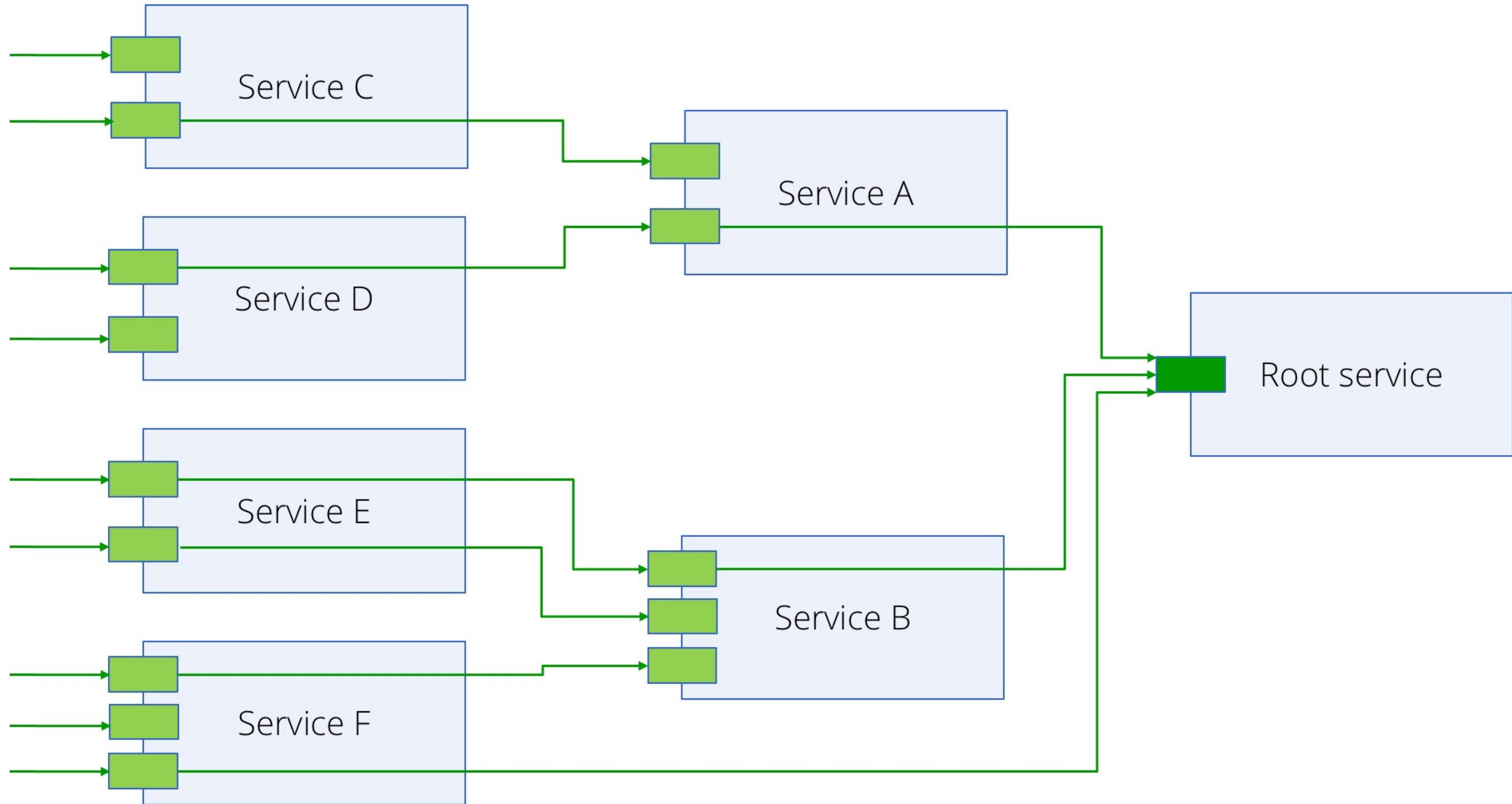
# Bulkheading & load shedding



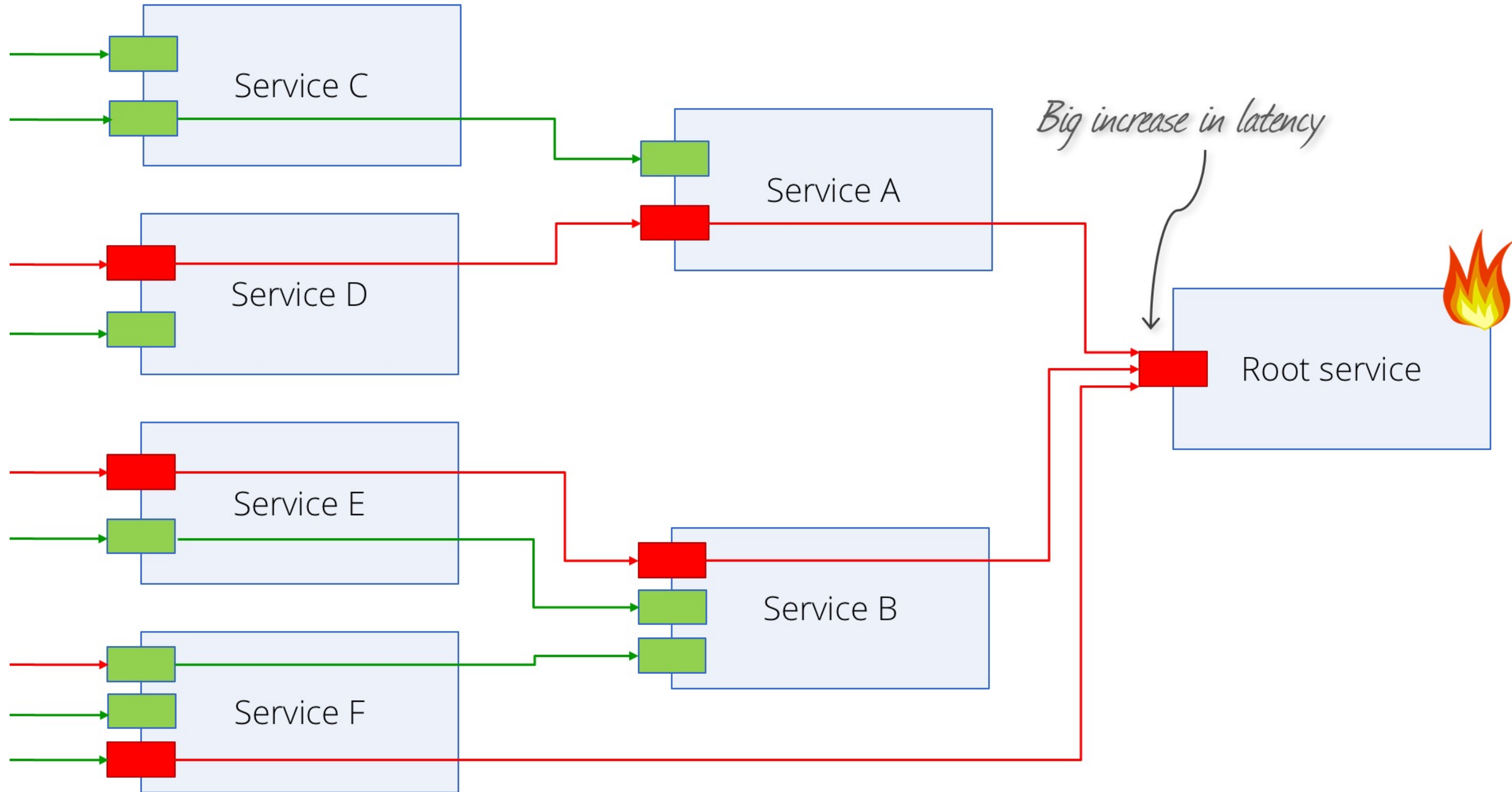
# Bulkheading & load shedding



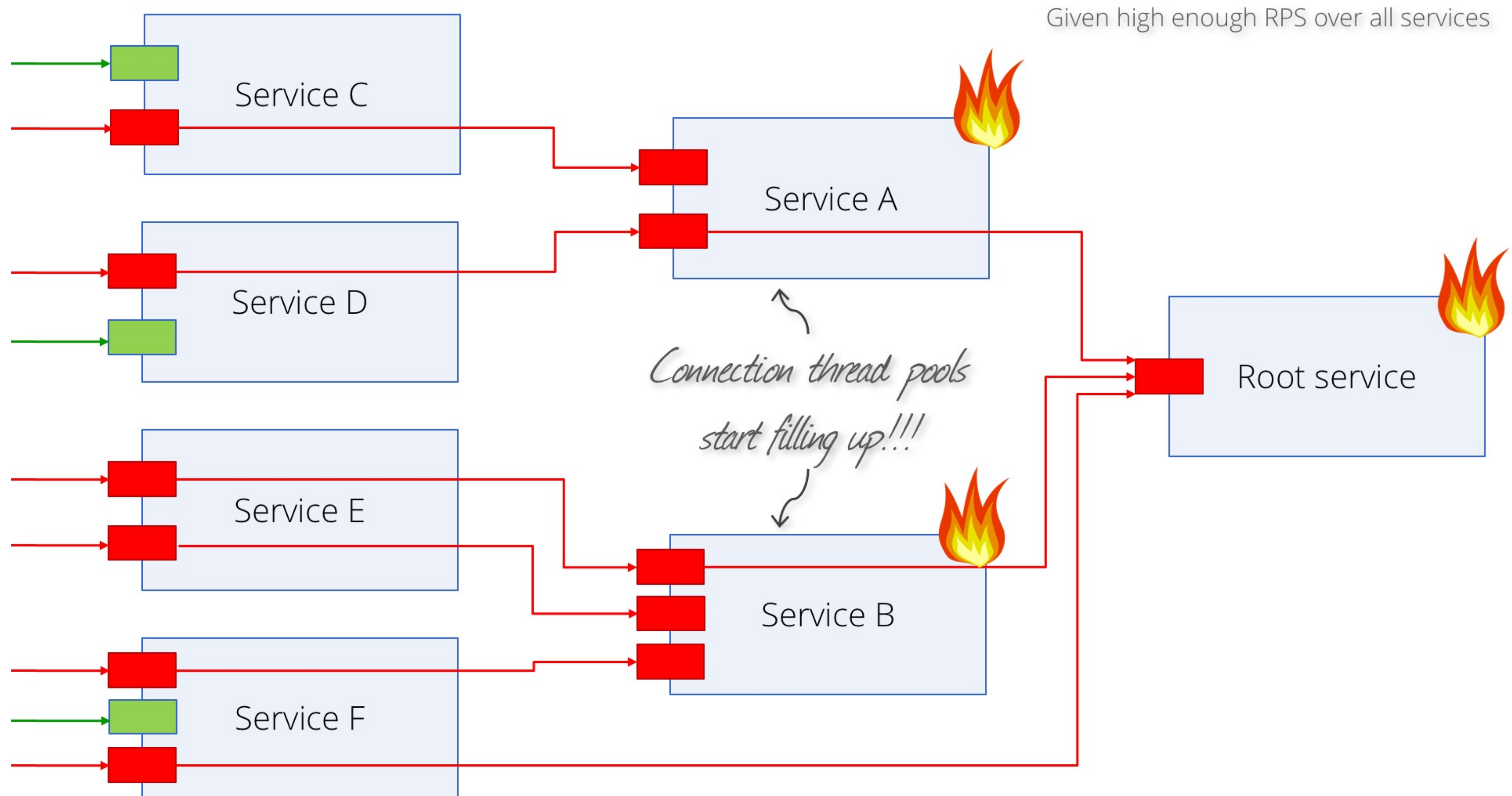
# Cascading latency example



# Cascade to direct dependencies

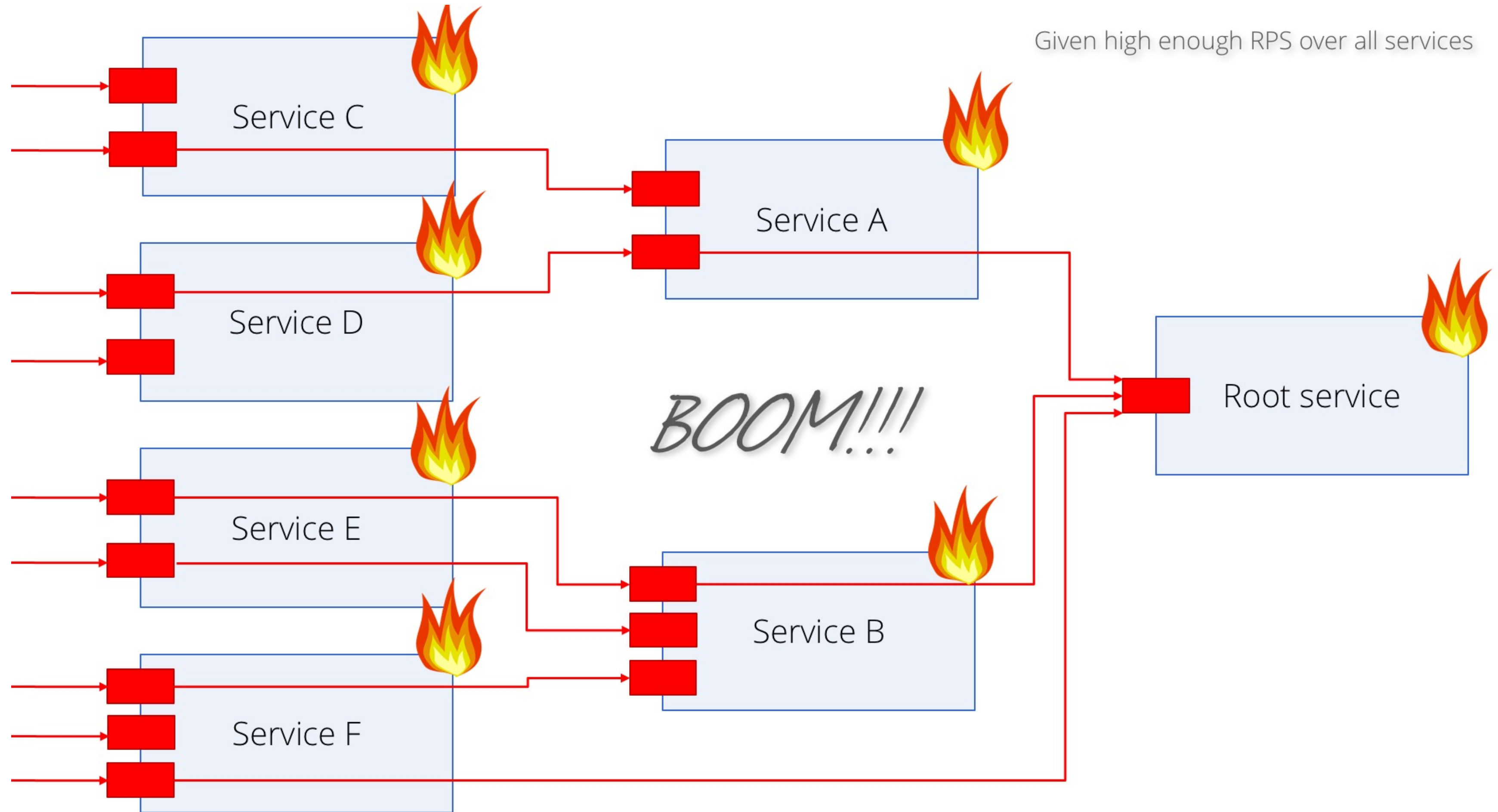


# Cascade to indirect dependencies

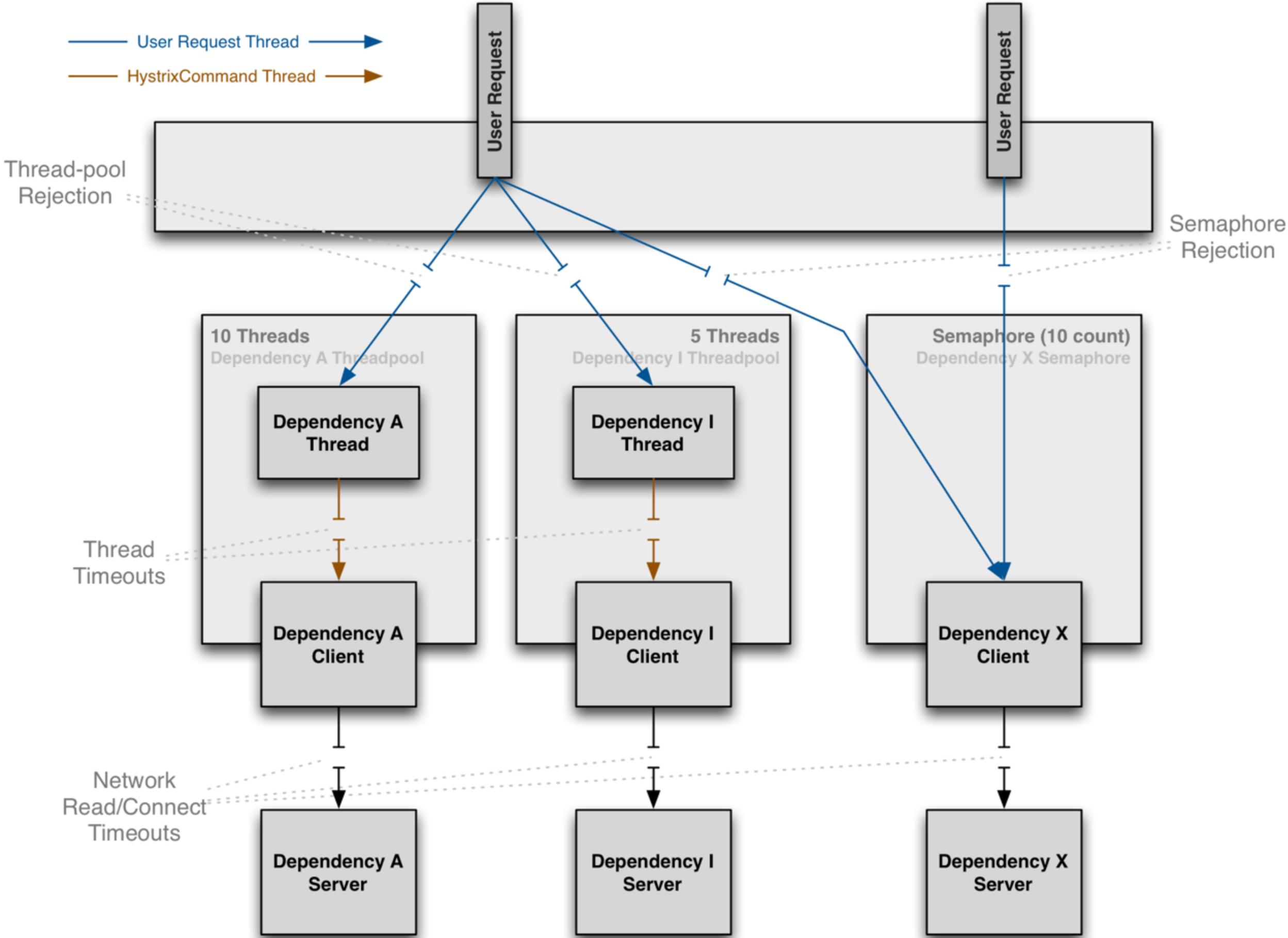




# Full cascade



# Bulkheading with threads or semaphores



# Threads

## Advantages:

- Calling thread may "walk away" if execution of the command times out.
- Hystrix can try to interrupt the Hystrix thread.

## Disadvantages:

- Threads add a little bit of computational overhead and memory usage  
*We never had any issues with it*
- Thread pools are harder to tune because they can be used by multiple commands.

# Semaphores

## **Advantages:**

- Very low overhead
- Easier to tune because semaphores are not shared between commands

## **Disadvantages:**

- Only limits the number of concurrent call, so it doesn't fully isolate.
- Calling thread can not "walk away" if command execution times out.
- Hystrix can't interrupt the thread.

# When to use thread or semaphore?

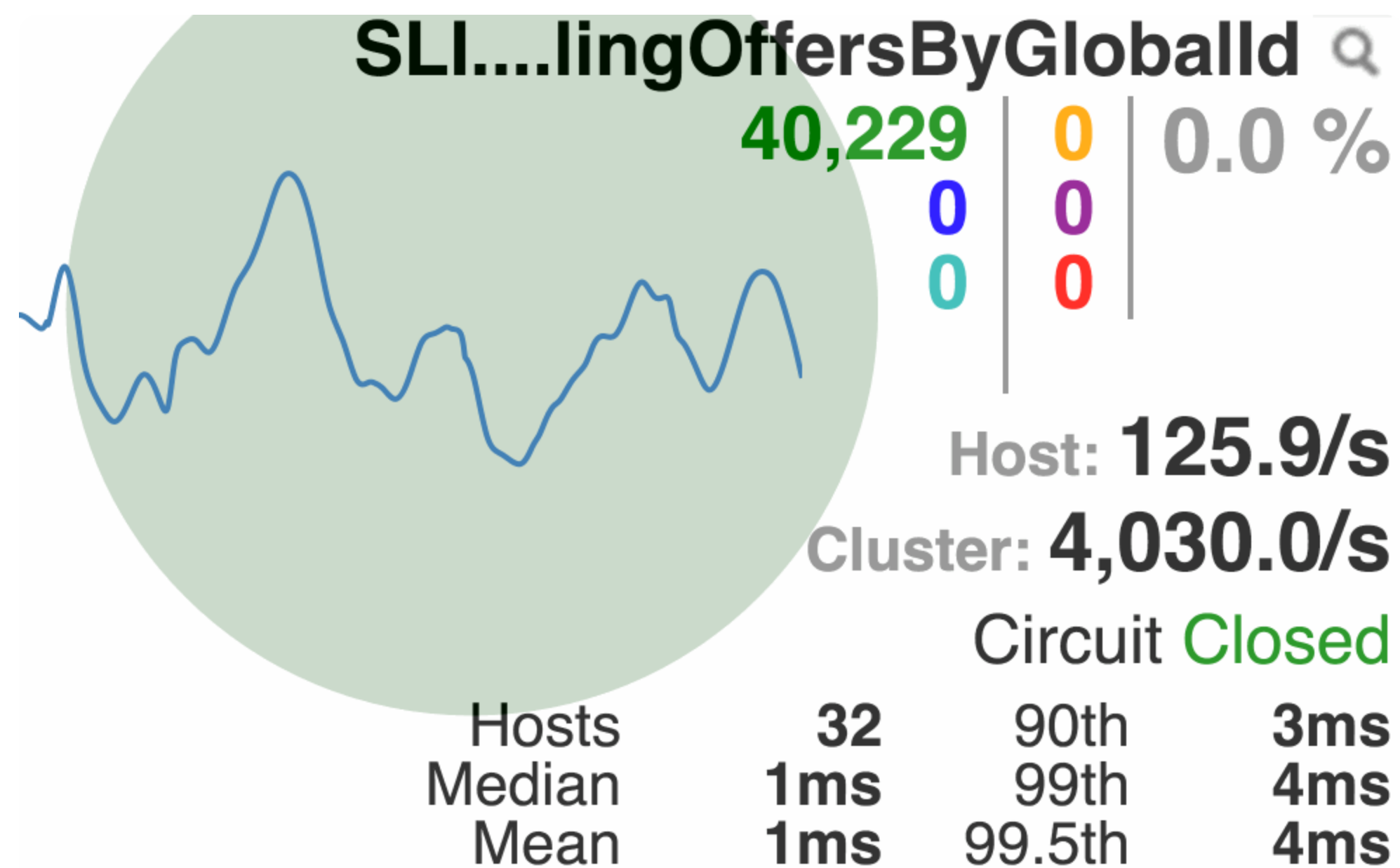
The default and the recommended setting is **thread isolation**.

Generally you should use **semaphore isolation** only:

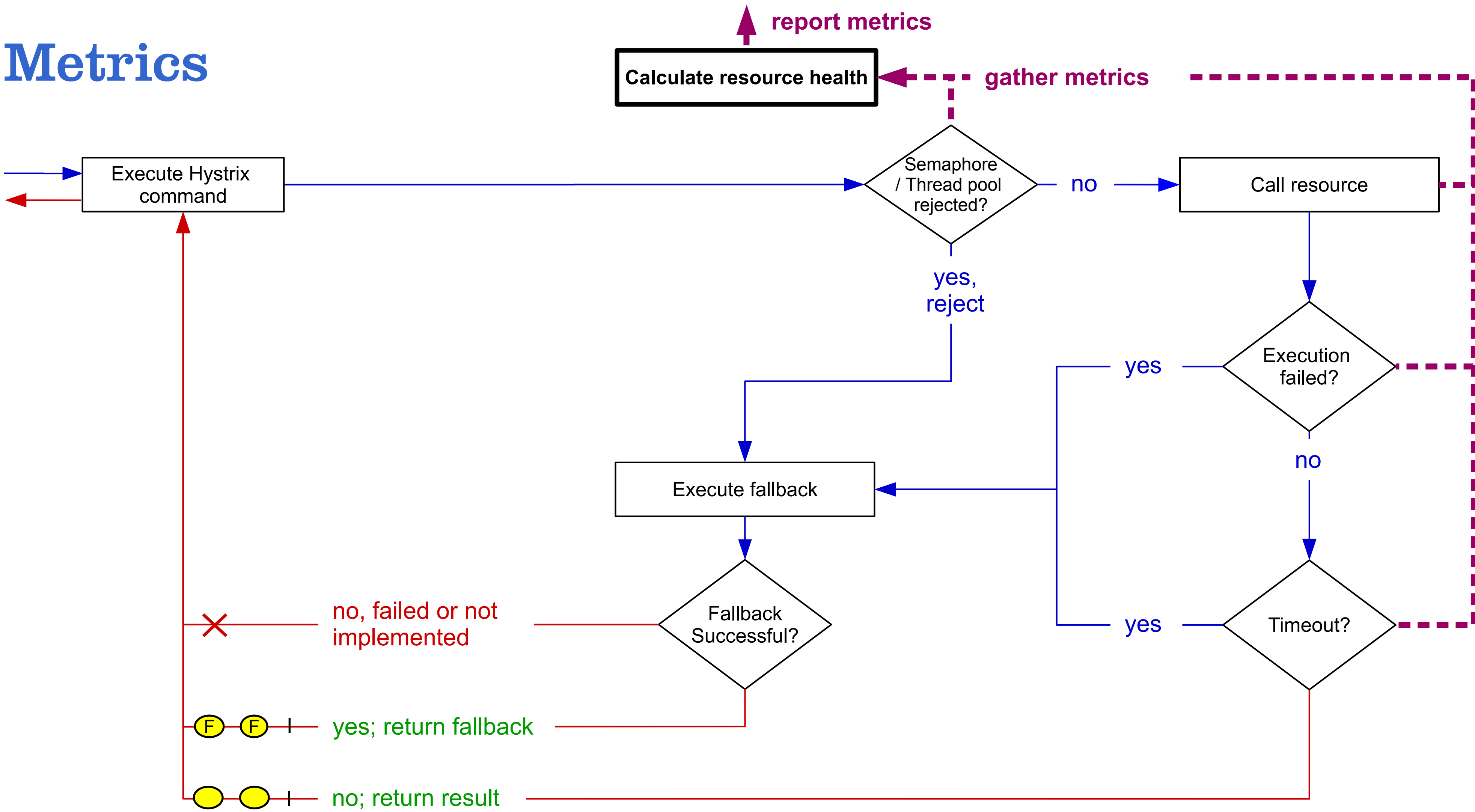
- when the thread overhead is too high for your use-case.
- when you are using a client with request based timeouts, which you know to be reliable.
- when you are using an asynchronous, non-block client with the **HystrixObservableCommand**.



# Insight into your downstream resources



# Metrics



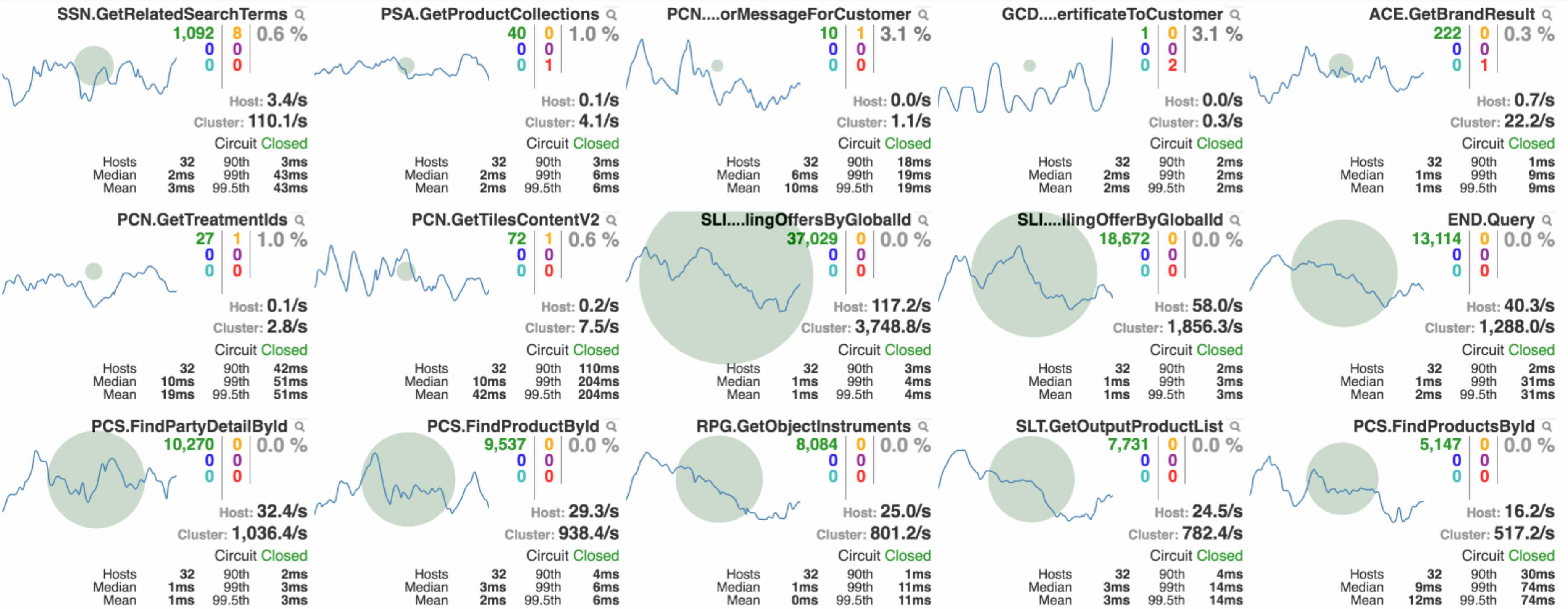
# Dashboard for real-time metrics

## Hystrix Stream: WSP

Filters:

**Circuit** Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %



# Long time metrics

## GETTILESCONTENTV2 - COMMAND METRICS

Error percentage

0.004%

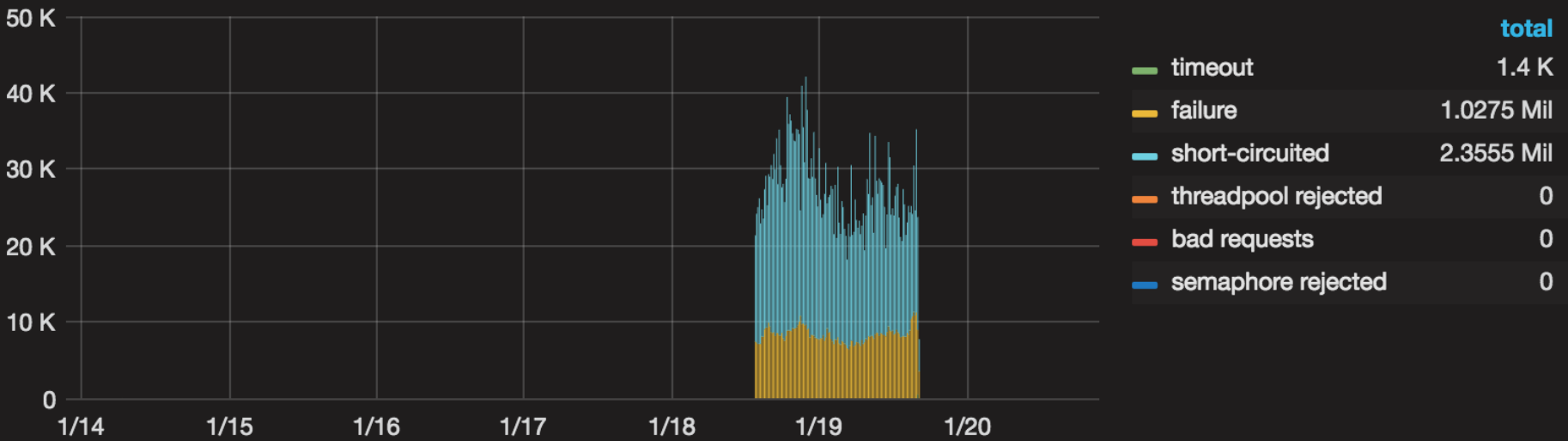
Circuit breaker  
state

OK

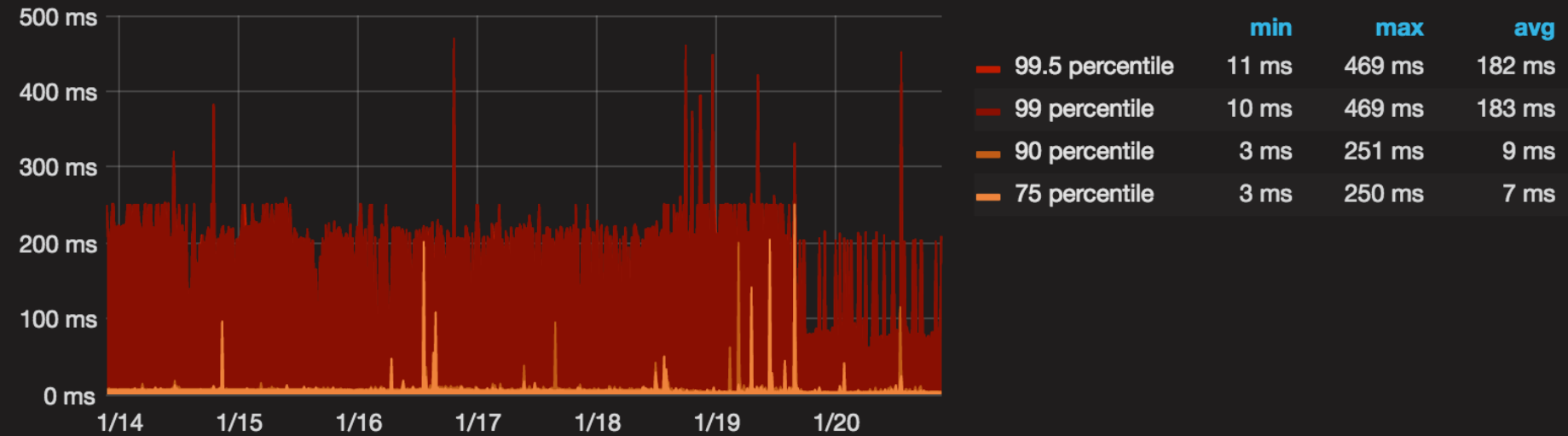
Successfull command executions



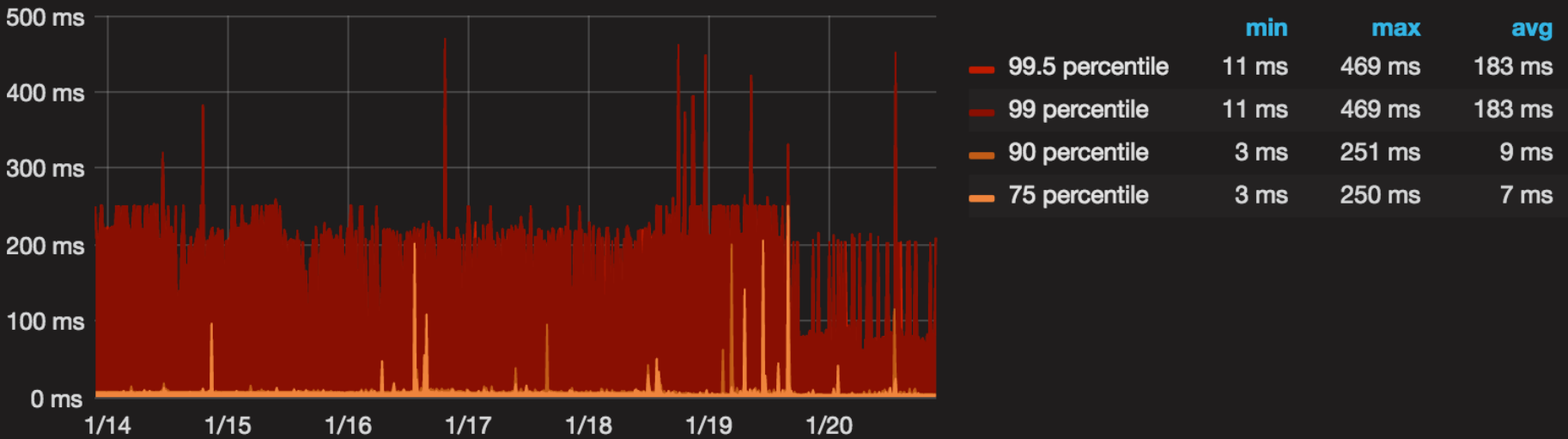
Failed command executions



External request execution latency

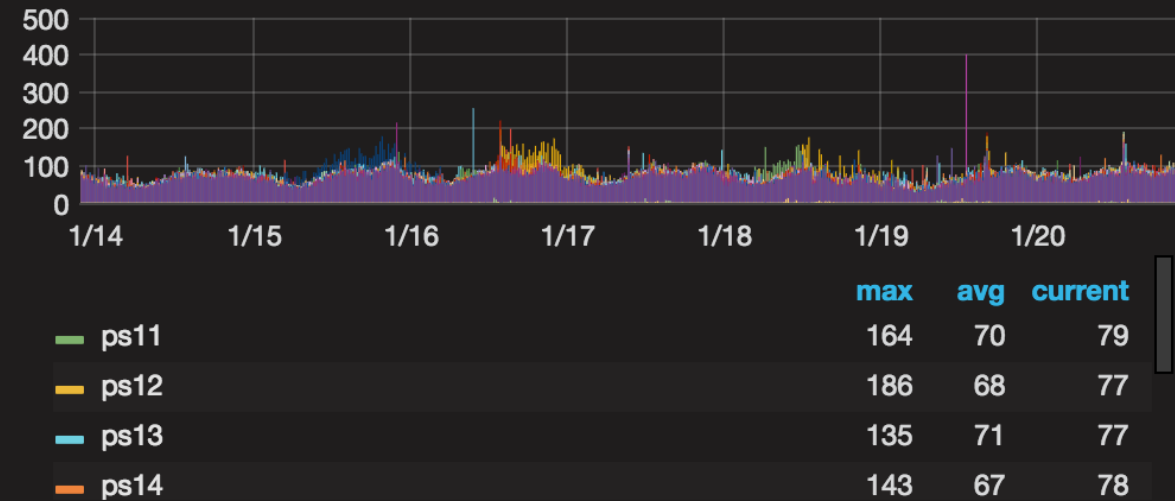


Command execution latency

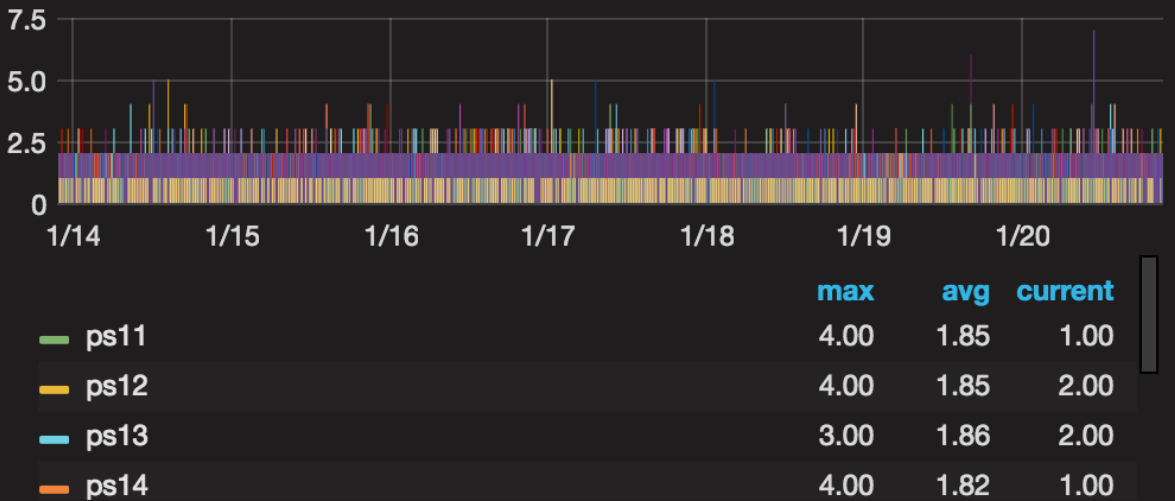


## DEFAULT - THREAD POOL

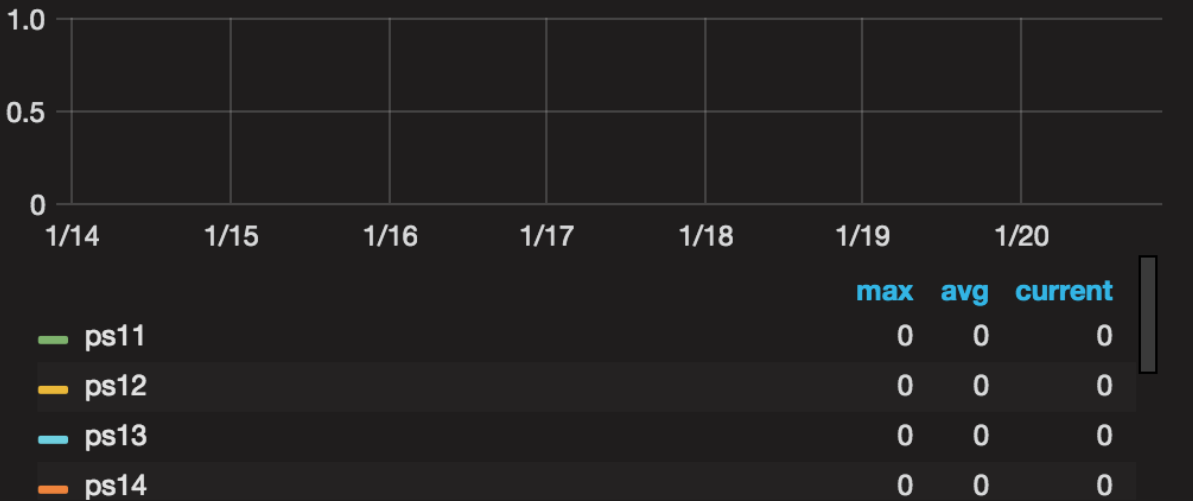
Rolling threads executed



Max active threads at same time

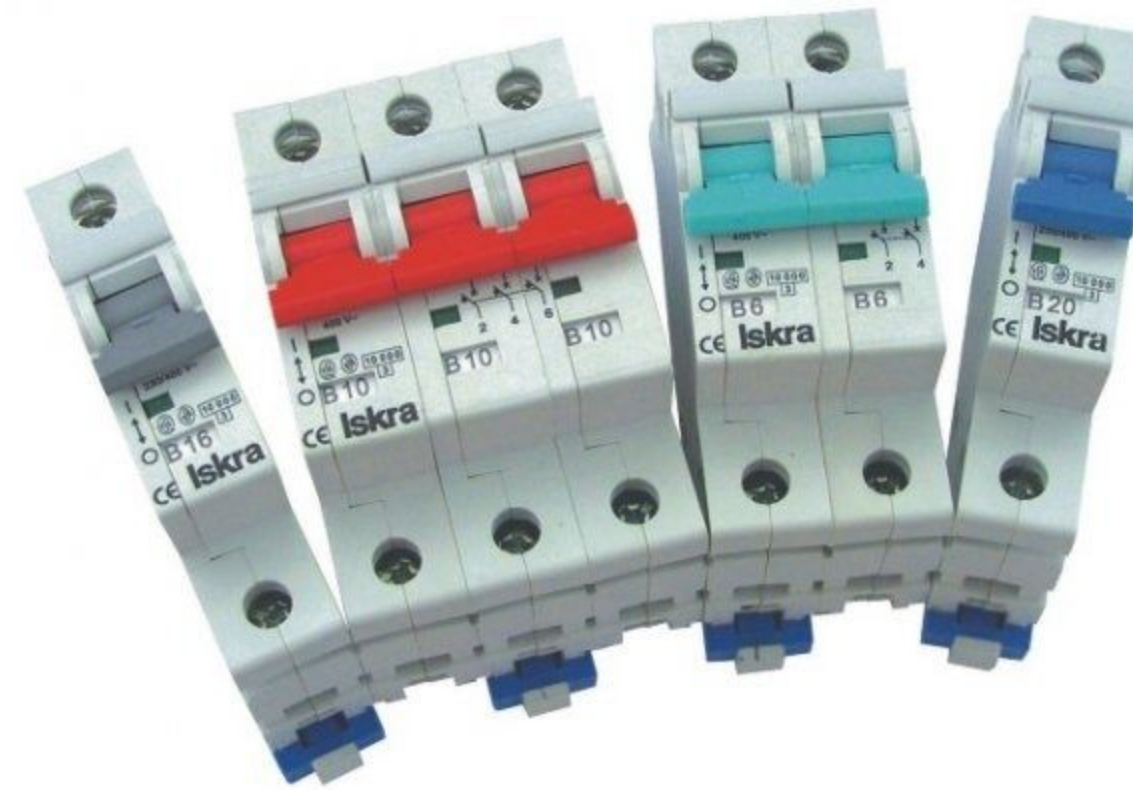


Rolling rejected commands



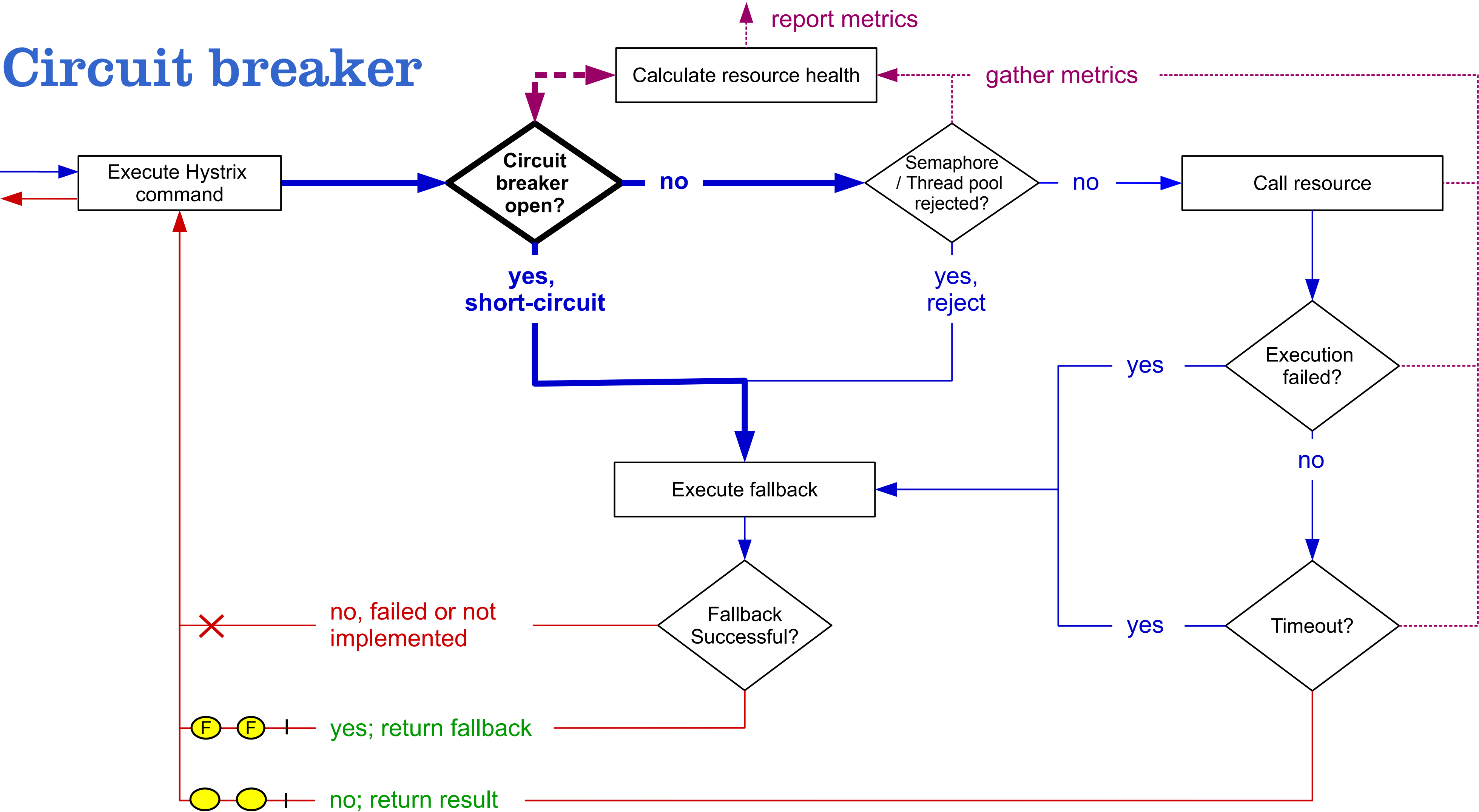


# Fail fast with the circuit breaker





# Circuit breaker



## Tips & Tricks



# Setup monitoring and alerting based on Hystrix metrics

**Using Hystrix, or any other circuit breaker solution, without monitoring and alerting is like being blind.**

All the circuits could be open and you wouldn't know...





# Alerting: critical and non-critical commands

Applications at bol.com make distinction in command criticality:

- **Critical**

Commands for services which can't have a sensible fallback and an outage will have a big impact.

- **Non-Critical**

Commands for services which have sensible fallbacks or failures don't have a big impact when an outage occurs.



If the error count is too high or the circuit breaker opens of a **critical** command then the responsible team and/or operations engineers get notified immediately.

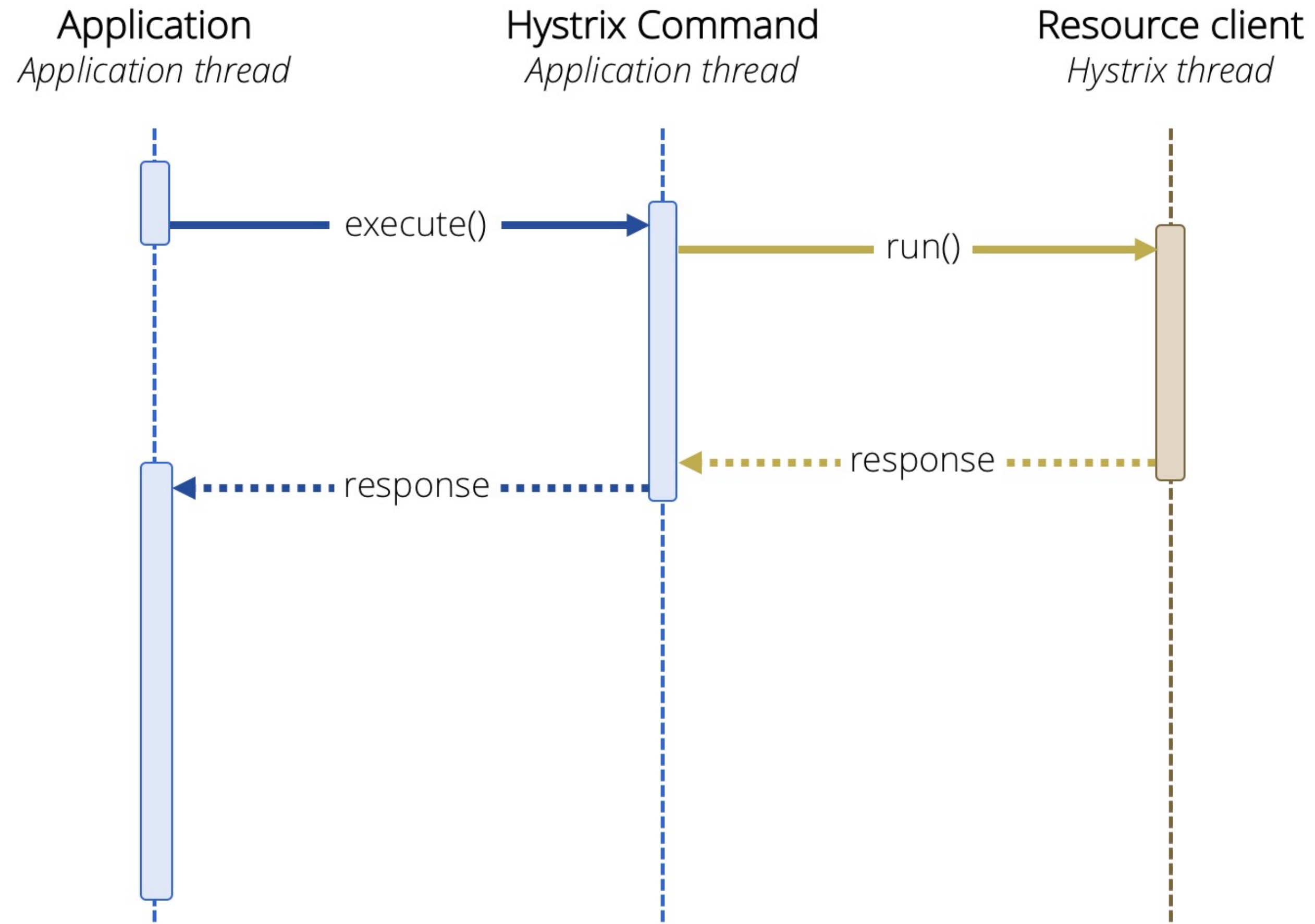
For **non-critical** commands only a warning is issued on the monitoring screens.

# Configure the (HTTP) client

**Don't forget the following settings on your (HTTP) client:**

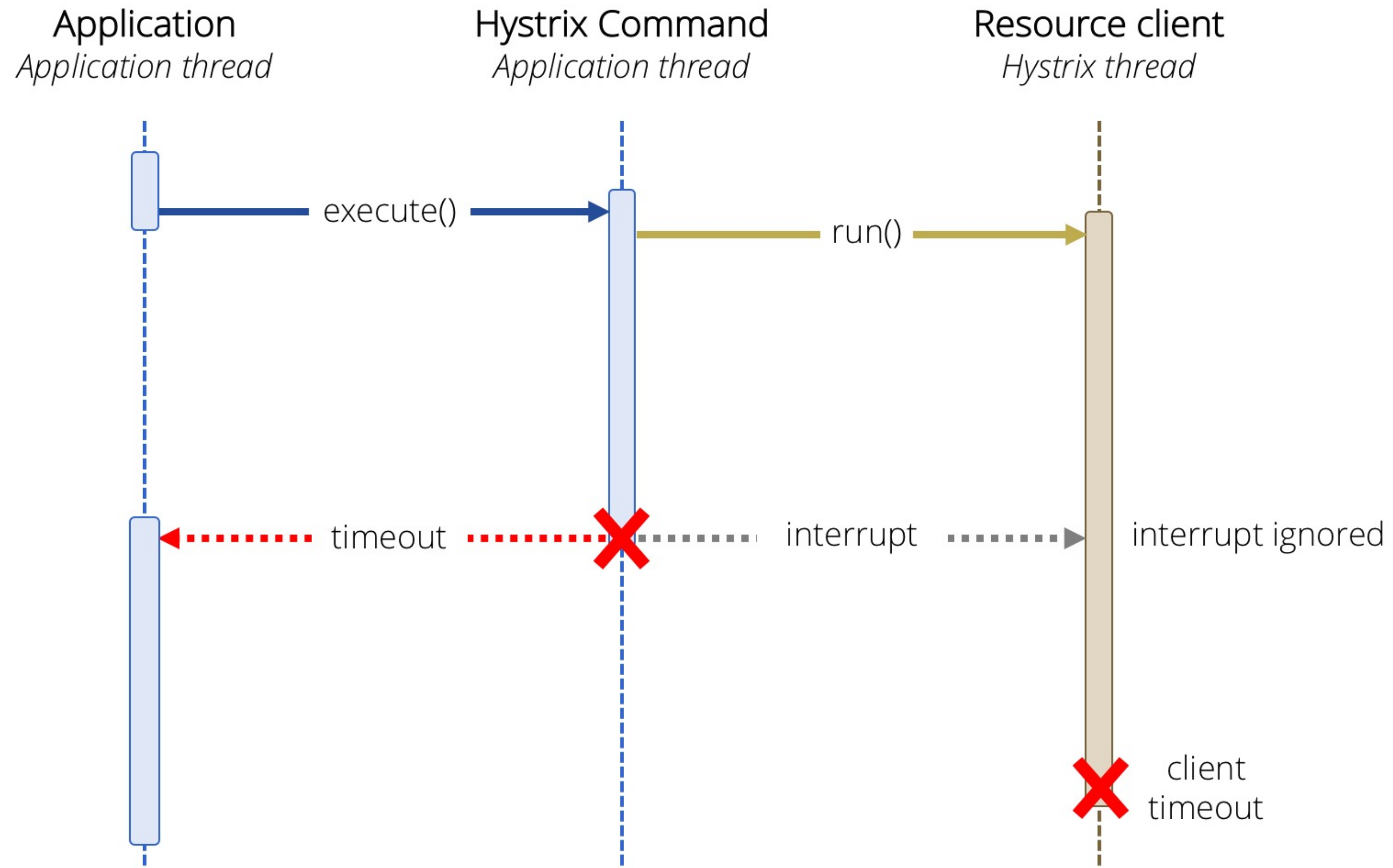
- Client timeout should be tuned according to the command timeout.
- Connection pools should be sized with regards to the Hystrix max. concurrent request settings.

# Normal request / response with thread pool isolation



# Too long client timeout with thread pool isolation

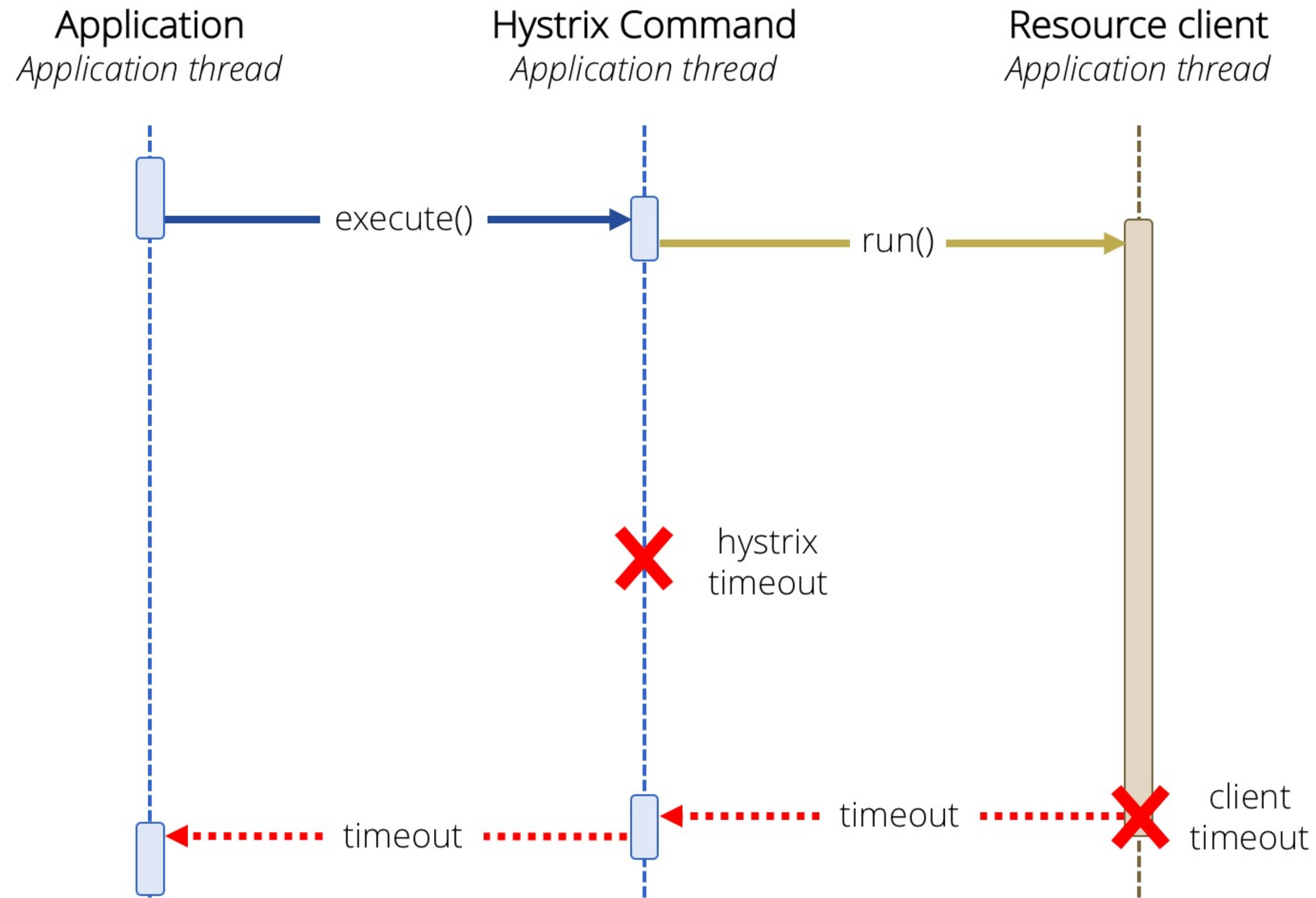
With thread pools, long client timeout settings causes unnecessary load shedding.



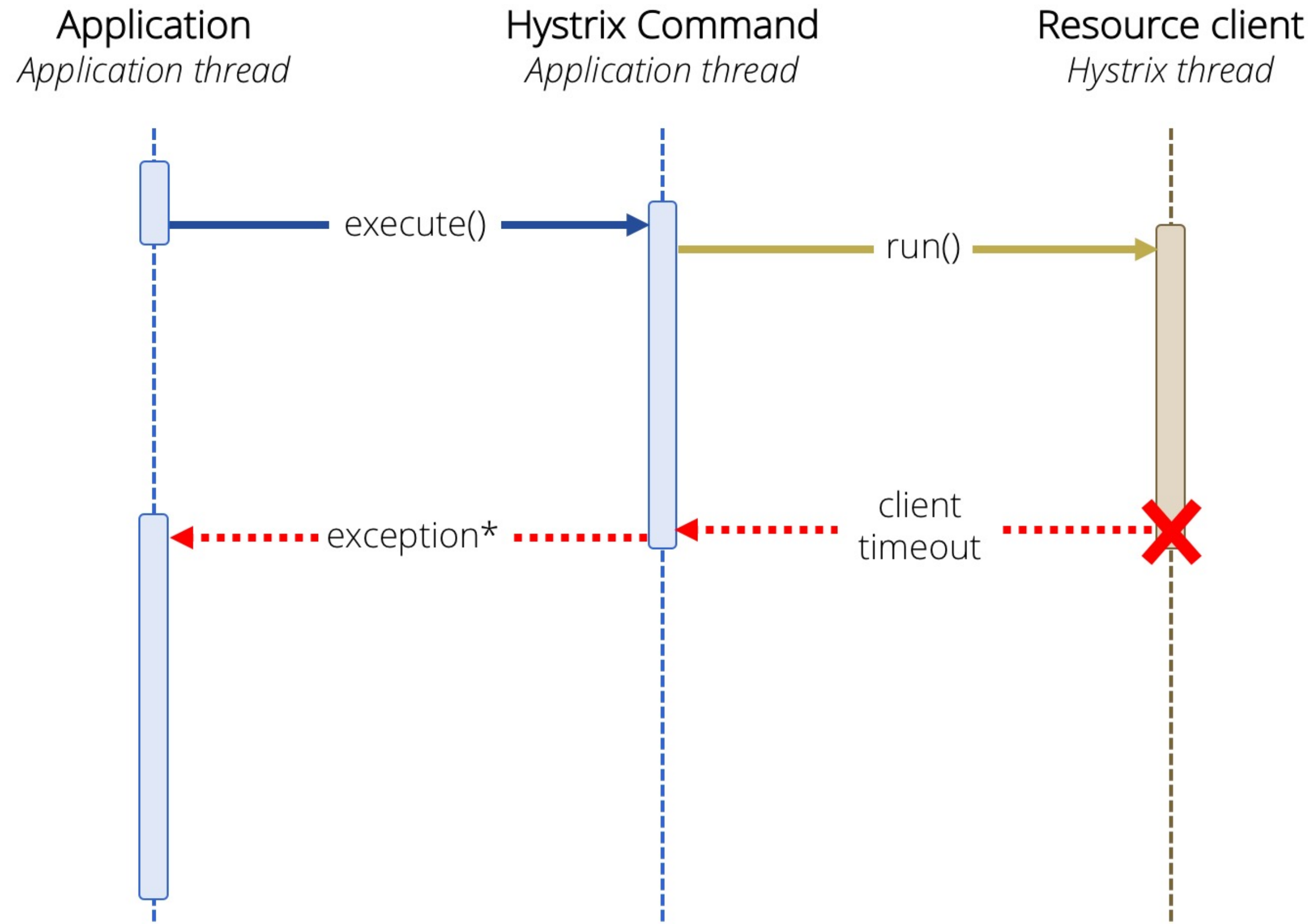


# Too long client timeout with semaphore isolation

With semaphores, long client timeout settings causes unnecessary load shedding and additional **latency**.

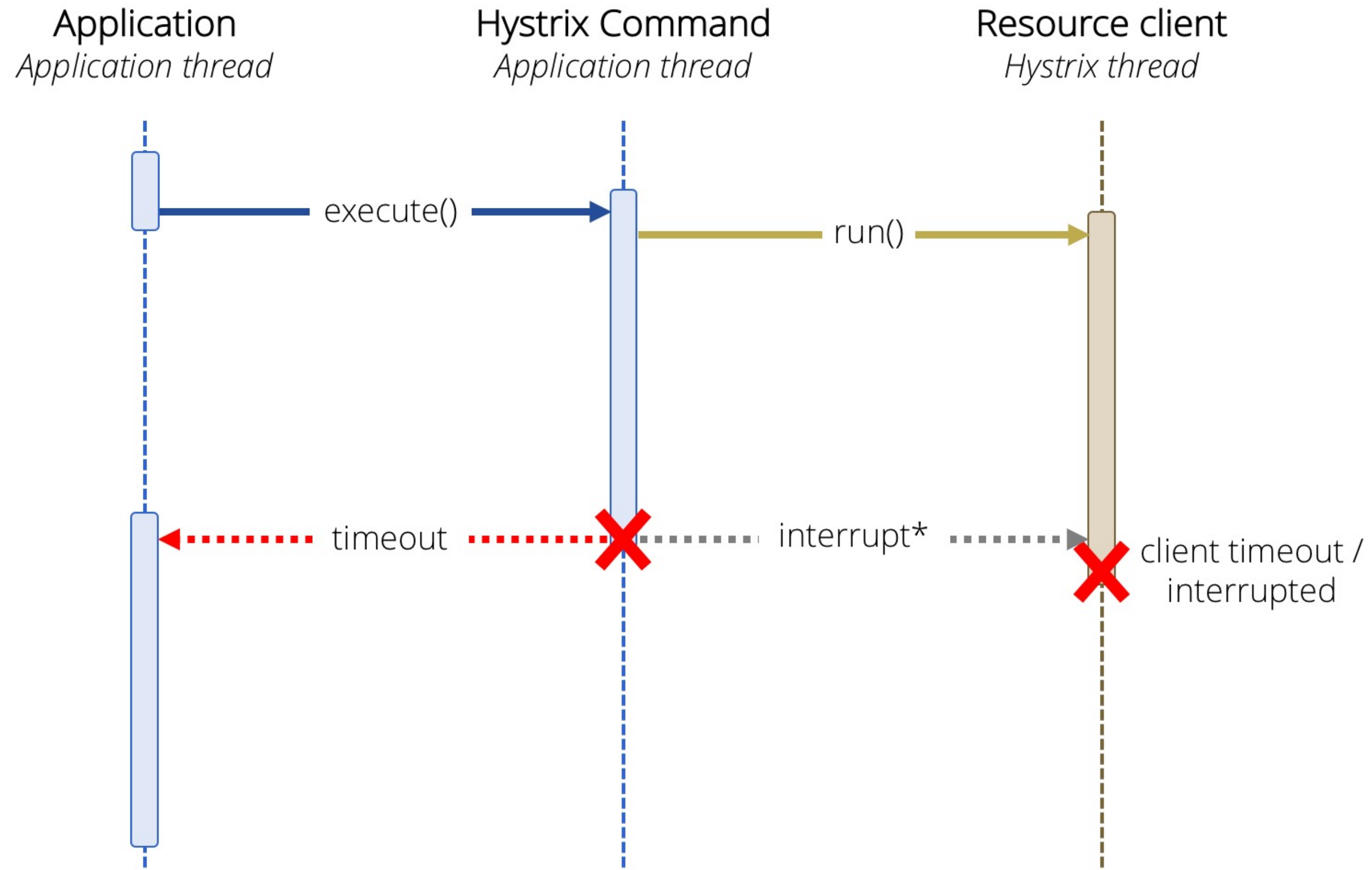


# Solution: Make client timeout before Hystrix



\* Hystrix counts any exception from the client as an exception, even if it is a timeout.

# Solution: Make client timeout just after Hystrix



\* Interrupts are often not reliable. Clients don't always respond to it, depending on timing...

# Trick: Map common timeouts to Hystrix timeouts

Create execution hook plugin:

```
public class MapTimeoutExecutionHook extends HystrixCommandExecutionHook {  
  
    @Override  
    public <T> Exception onExecutionError(HystrixInvokable<T> command, Exception exception) {  
  
        if(isTimeoutException(exception)) {  
  
            HystrixTimeoutException timeoutException = new HystrixTimeoutException();  
            timeoutException.initCause(exception);  
  
            return timeoutException;  
        }  
  
        return e;  
    }  
}
```

Register plugin:

```
HystrixPlugins.getInstance().registerCommandExecutionHook(new MapTimeoutExecutionHook());
```



# Trick: Set the command timeout as the request timeout

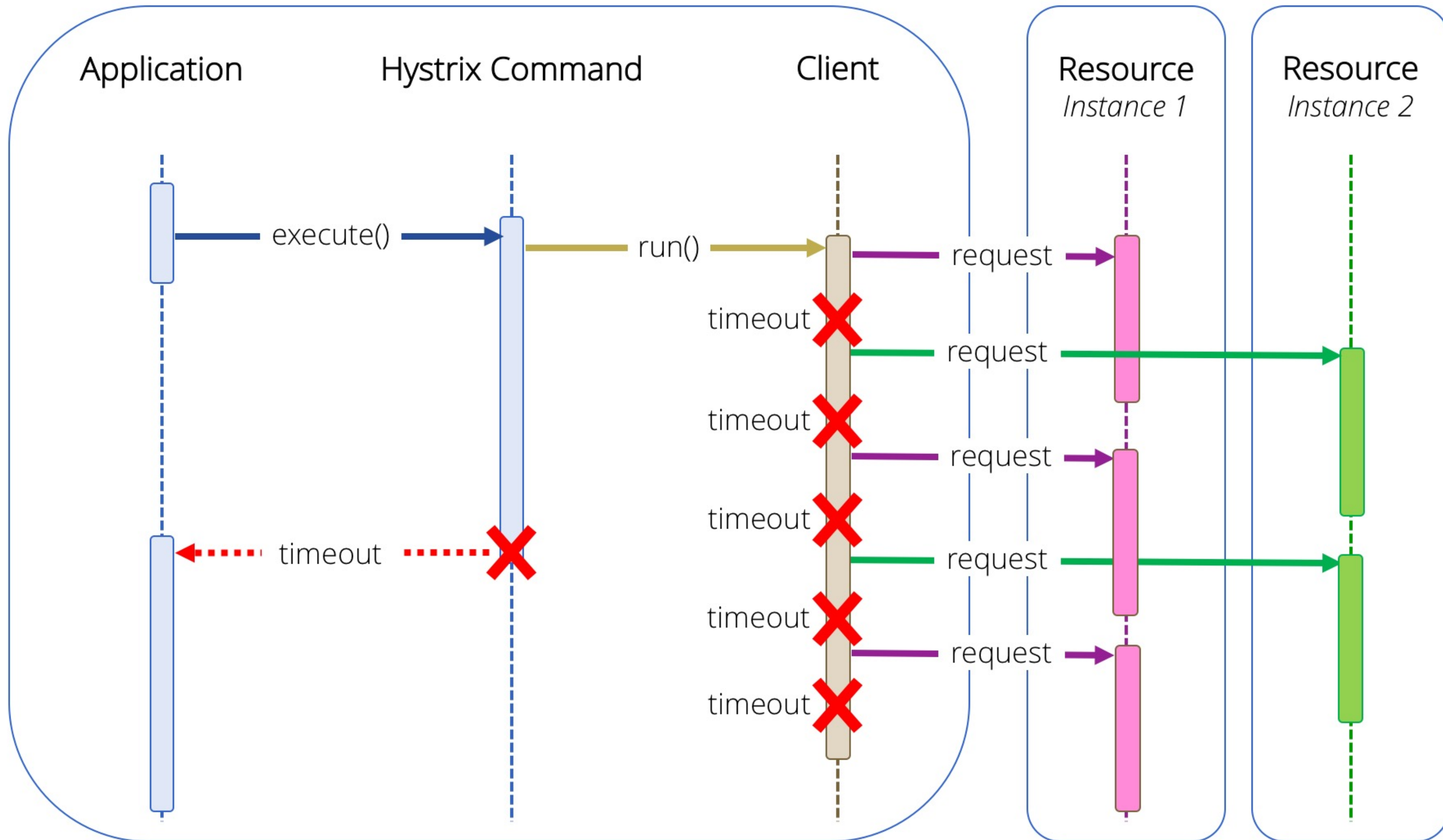
Extract timeout from command and use as read timeout:

```
public class GetReviewsCommand extends HystrixCommand<List<Review>> {  
  
    // fields and constructor  
  
    @Override  
    protected List<Review> run() {  
  
        int readTimeout = getProperties().executionTimeoutInMilliseconds().get();  
  
        return client.getReviews(productId, readTimeout);  
    }  
}
```

*Only when not using retries!*

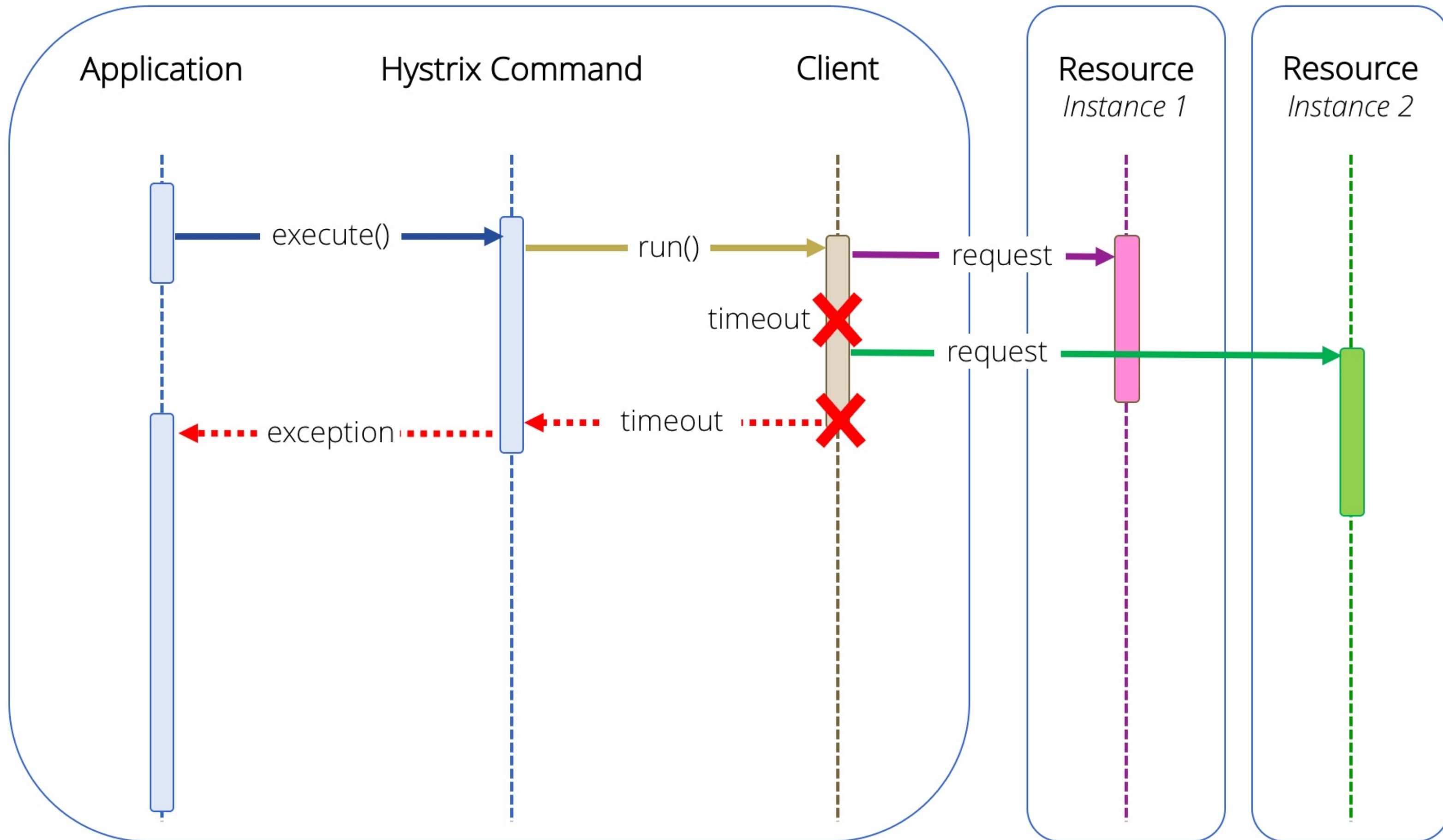
# Don't have too many retries!

Too many retries also causes unnecessary load shedding, additional latency and can flood the resource.



# Advice: have at most one retry

After one retry it should be good or just let it fail.



# Define a naming scheme

Commands, thread pools and command groups have key names.

## **A key:**

- should be descriptive, indicating which resource and what part of it is being used.
- needs to be unique within an application.

## **A uniform naming style:**

- Prevents naming conflicts.
- Promotes descriptive names.
- Makes it easier to use for monitoring and metrics purposes.



# Naming scheme example:

Pattern applied at bol.com:

**{service-id}.{command-name}[{service-version}]**

Examples:

- PCS.GetProductById
- PCS.GetProductByIdV2

# Tuning: the numbers tell the tale

- Tune on production and based on real traffic patterns.
- Base the timeout settings on the behaviour of the resource, not on the time you are willing to wait for the resource\*.
- Only re-tune if the behavior or performance characteristics of the command have changed, based on alerts and monitoring.
- Don't forget to tune the timeouts of the client!



*\* If the healthy latency is too long then add an extra timeout layer around the command.*

# Tuning: command timeout and concurrency formulas

Base on the metrics of a healthy resource under peak pressure:

## Command settings

- **Timeout:** 99,5<sup>th</sup> percentile command latency
- **Max Concurrent executions:**  $\{\text{request/second}\} \times \{99,5\text{th \% latency in seconds}\} + \{\text{breathing room}\}$
- **Thread pool size:**\*  $\{\text{combined request/second}\} \times \{\text{max 99,5th \% latency in seconds}\} + \{\text{breathing room}\}$

## Settings for a new resource:

- **Timeout:** Use performance test data or set to 1 second or higher.
- **Max Concurrent executions / thread pool size:**\* Use performance test data or set to 10 or more.

Be generous and tune again as soon as you have the production metrics!

*\* Commands with big differences in latency characteristics should have separate thread pools*

# Tuning: (HTTP) client tuning formulas timeout formulas

## Connect timeout

- Within same network: **100ms or lower**
- To other network: **measure**

## Read/socket timeout

- When not using retries: **equal to command timeout.**
- When using retries: ***{99.5th resource latency} - {median resource latency}***



# Operations





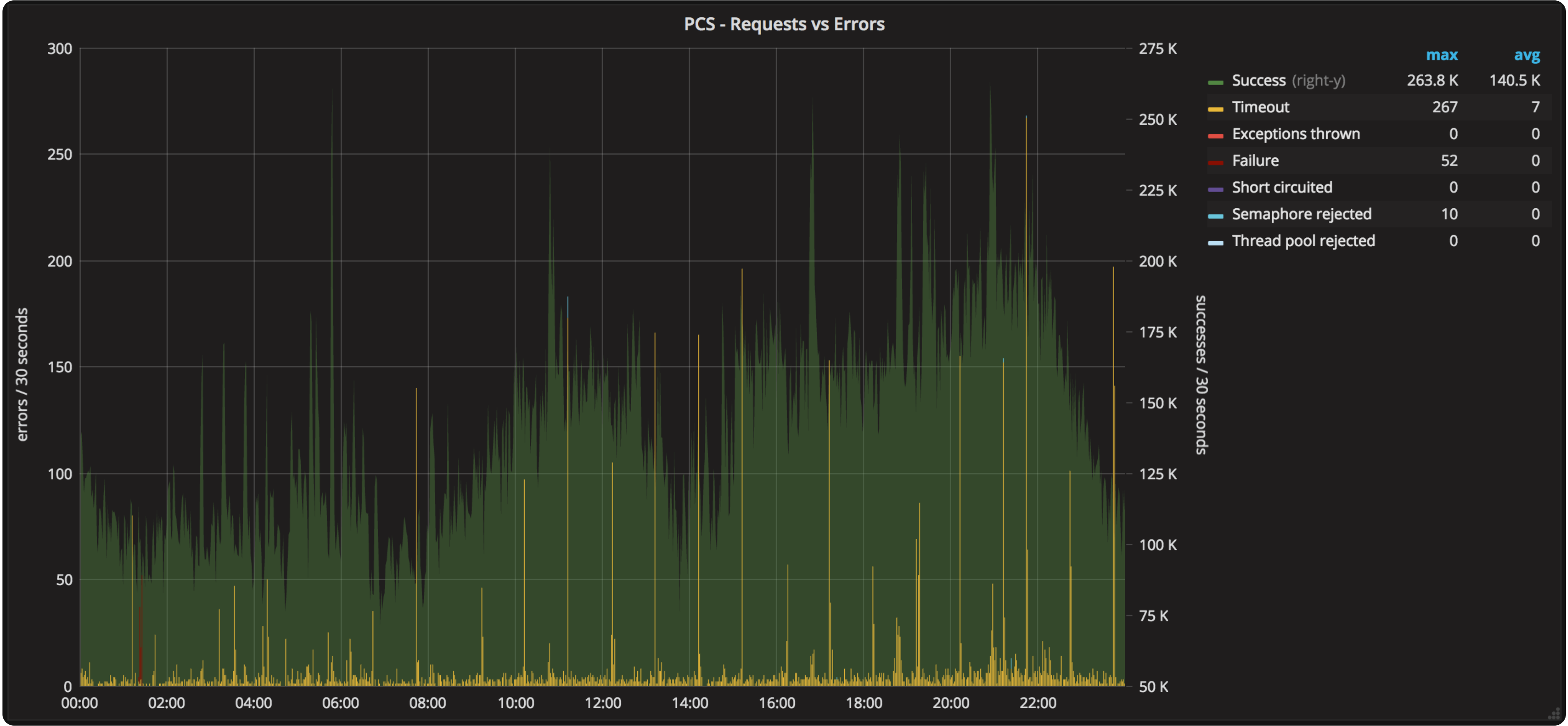
# Expect Jitter and Failure

Hystrix measures and reports metrics with very small granularity which reveals “jitter” — seen as bursts of timeouts, thread-pool rejections, slow downs, ...

Some of the causes:

- Garbage collection
- New machines starting up and “warming up”
- Different payload sizes for different request arguments
- Bursty call patterns
- Cache misses

# Example of jitters



# When Hystrix is reporting failures

**If you notice timeouts, load shedding or circuit breakers being open, don't overreact by immediately reconfiguring the commands.**

Do not give a command more resources (increasing thread pool, timeouts, queues) to try to give it some temporary breathing room. It may make things worse!

Find out what is causing Hystrix to shed load, short-circuit, timeout and reject before doing any configuration changes.

It may be that Hystrix is just doing it's job...



# Experiences @ bol.com



- Resilience prevented many small outages and a couple of big ones.
- We sleep **a lot** better now!
- Hystrix is a great library:
  - It does its job well.
  - It integrates with our tools.
  - It is easy to learn and implement.

# Experiences @ bol.com

- Tuning is often overlooked and not trivial!
  - Incorrect timeouts.
  - Incorrect thread-pool sizes.
  - Client not correctly configured.
- Implementing Hystrix is more an **organisational challenge** than a technical challenge.







## Wrap up

- Failures are not the exception, they are normal. We need to be resilient.
- Techniques to increase resilience:
  - Implement fallback's where possible.
  - Manage timeouts carefully.
  - Use bulk heading and load shedding to prevent resource hijacking.
  - Add circuit breakers to skip futile calls to unhealthy systems.
- Adding resilience doesn't need to be hard with great libraries like Hystrix.

**Thank you for your attention**

**Any questions?**

Slides:	<a href="http://bit.ly/goto-resilience">http://bit.ly/goto-resilience</a>
Hystrix project:	<a href="https://github.com/Netflix/Hystrix">https://github.com/Netflix/Hystrix</a>
bol.com tech blog:	<a href="https://techlab.bol.com/">https://techlab.bol.com/</a>
Jobs @ bol.com:	<a href="https://banen.bol.com/">https://banen.bol.com/</a>

SLIDES

