How to design resilient applications with Hystrix

18 January 2016 | **Insight Zuhlke, Software Engineering** | **Andreas Landerer**
Reading time: 7 minutes

When designing and implementing distributed systems, resiliency often plays a minor role. Michael T. Nygard explains in his book “Release It” why we should write software that is cynical and expects bad things to happen. A resilient application keeps processing transactions even when a single component fails. A system that has not been designed with resiliency in mind is bound to fail and it will fail where it matters most – in production.

Highly distributed environments pose new challenges

Resiliency is becoming increasingly important. Topics like Microservices or the Internet of Things (IoT) pose new challenges. In these highly distributed environments we can’t rely on dependencies to be always available or responsive. We have to be aware of the fact that each and every integration point poses a risk. They will fail at some point and we have to be prepared for that failure. Far too often we put too much faith in external resources. Every call to a database or an external service is liable to break at some point. If we fail to protect ourselves from these breaches, we risk that they might render our whole system invalid. In this event we can’t provide business value anymore and are likely to violate given service level agreements.

Prevent cascading failures

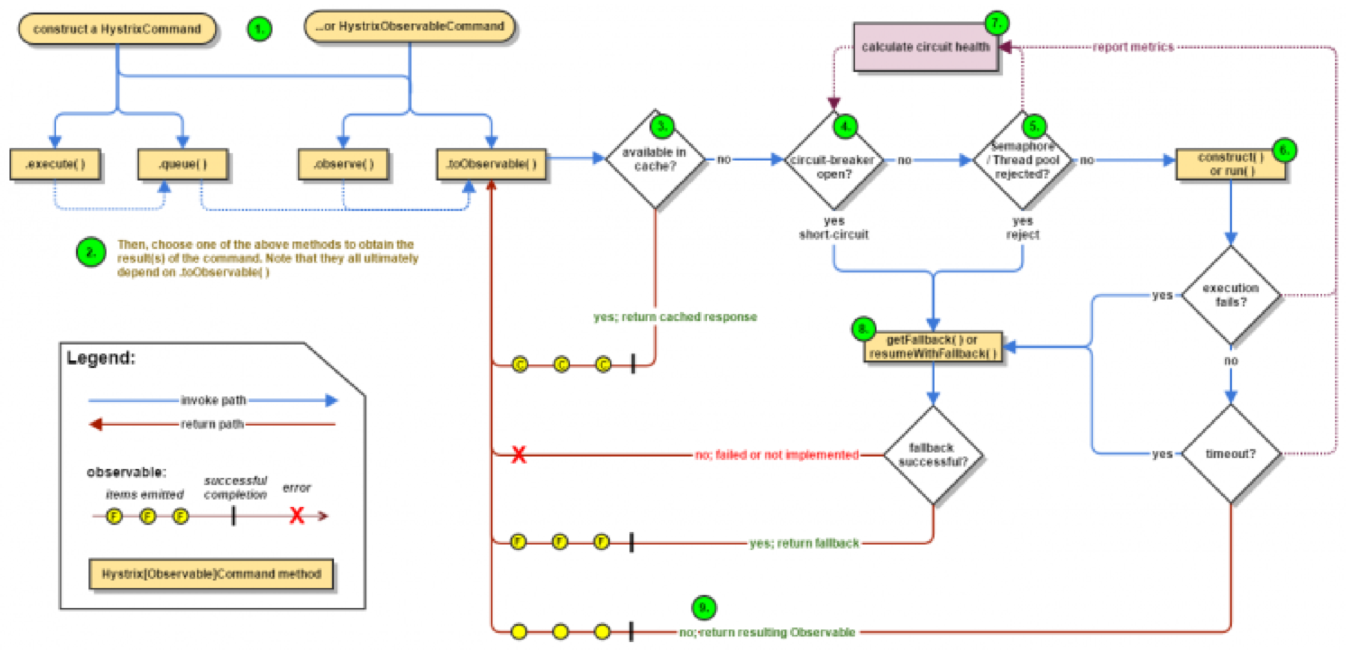
We have to accept that we cannot prevent failure. It is only a matter of time until parts of our systems will break. Therefore we have to design them to be able to cope with such occurrences. The very key to resiliency is that we prevent failures in one part of the system from propagating or multiplying themselves across layers or system boundaries. Isolating our system will prevent those cracks from rendering the whole system invalid. Generally speaking, there are two kinds of failures: an external resource might respond with an immediate error or it might respond slowly. Receiving an error straightaway is always preferable to receiving a slow response. In the latter case, calling an external resource without specifying an explicit time-out is liable to bind your threads. In a worst case scenario all your threads are blocked and the failure of one single resource has led to the failure of the overall system.

Employ circuit breakers

In order to avoid this Scenario, Nygard describes the circuit breaker pattern. This pattern facilitates isolation of our system and avoids propagation of failures across layers or system boundaries. In the normal “closed” state, the circuit breaker executes operations as usual and your calls will be forwarded to the external resources. When the call fails, the circuit breaker makes a note of the failure. Once the number or frequency of failures exceeds a certain threshold, the circuit breaker trips and “opens” the circuit. From that point on calls fail immediately, without any attempt to execute the real operation. After a while, the circuit breaker switches into “half-open” state, where the next call is allowed to execute the real operation. Depending on the outcome of the call, the circuit breaker either switches to the closed or to the open state again.

Hystrix

Hystrix is an open source framework developed by the company Netflix, facilitating latency and fault tolerance for distributed systems. At its core it provides an implementation of the circuit breaker pattern that you can easily utilize in your applications. The following image gives you an overview on how Hystrix makes use of the circuit breaker pattern. In the subsequent paragraphs, I will elaborate on how you can incorporate it into your application.


construct a HystrixCommand
1.
...or HystrixObservableCommand
.execute( )
.queue( )
.observe( )
.toObservable( )
2.
Then, choose one of the above methods to obtain the result(s) of the command. Note that they all ultimately depend on .toObservable( )
3.
available in cache?
no
yes; return cached response
4.
circuit-breaker open?
yes short-circuit
5.
Semaphore / Thread pool rejected?
yes reject
6.
construct( ) or run( )
execution fails?
yes
no
timeout?
7.
calculate circuit health
report metrics
8.
getFallback( ) or resumeWithFallback( )
fallback successful?
no; failed or not implemented
yes; return fallback
9.
no; return resulting Observable
Legend:
invoke path
return path
observable:
items emitted
successful completion
error
Hystrix[Observable]Command method

The circuit breaker pattern in Hystrix [1]

Utilize Hystrix commands

In order to isolate your system from external resources, you can encapsulate the calls to such resources inside Hystrix commands. These commands are offered in three flavours: synchronous, asynchronous and reactive. The following snippet illustrates how easy it is to implement a Hystrix command.

```
public final class GetMarketDataFeedsCommand extends
                HystrixCommand<List<MarketDataFeed>> {

        private final ExternalMarketDataFeedsService service;

        public GetMarketDataFeedsCommand(
                        ExternalMarketDataFeedsService service) {
                super(HystrixCommandGroupKey.Factory.
                                asKey("MarketDataFeedsGroup"));
                this.service = service;
        }
        @Override
        protected List<MarketDataFeed> run() throws Exception {
                return service.getMarketDataFeeds();
        }
}
```

Each command is derived from HystrixCommand – or its
observable counterpart HystrixObservableCommand – and implements the single mandatory ***run***-method. With this method you call any external resources. If the call is not successful or the external resource responds slowly, and the default time-out of 1000 ms is exceeded, the method will fail fast and throw a HystrixRuntimeException preventing your threads from being blocked indefinitely. The time-out is freely selectable in the case that the default does not meet your requirements. In addition to implementing the ***run***-method you have to call the constructor of the parent class providing a command group key. Hystrix provides thread-pools on a command group level. This is important if your system is unbalanced and the number of requests exceed the number of available threads. Once all available threads are occupied, Hystrix will circumvent the call and fail fast.

Graceful degradation of service

There are situations where failing fast is not a viable option and you would prefer to take some alternative action, like hitting a local cache, or queuing up requests for retry later on. In that case you prefer to fail silently and gracefully degrade the service offered by your system. Therefore, the Hystrix commands provide an optional ***fallback***-method, which you can use in order to implement your alternative solution.

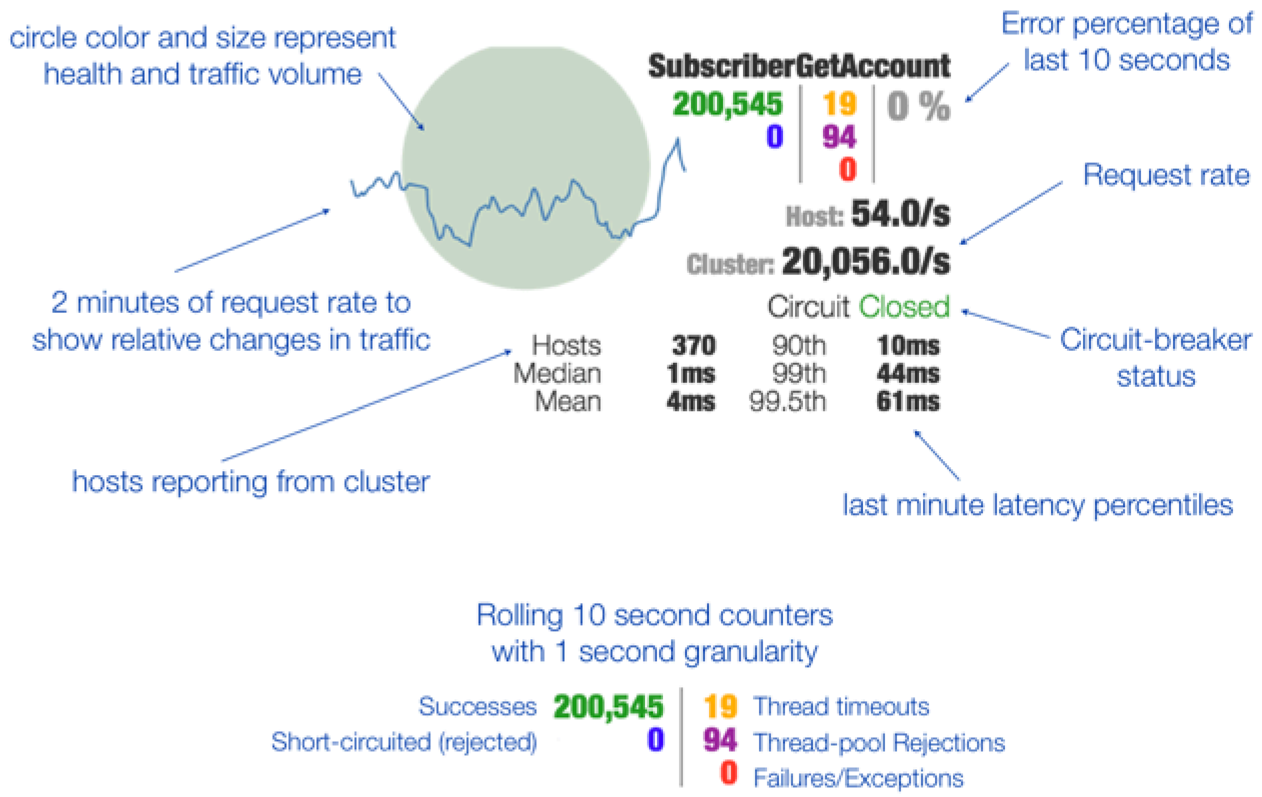
```
@Override
protected List<MarketDataFeed> getFallback() {
        return localCache.getMarketDataFeeds();
}
```

Tripping the circuit breaker

Hystrix keeps statistics regarding the number and frequency of failed commands. If these statistics exceed the corresponding thresholds, the circuit breaker opens and subsequent calls will fail immediately, giving the external resource some time to recover. After a default timeframe of 5000 ms has elapsed the framework will allow a single call to be forwarded. If this call succeeds, the circuit breaker will close again, otherwise it remains open.

Monitor your system health

Another feature that Hystrix offers is a dashboard recording your system's health in real time. Each Hystrix command is represented in this dashboard by a circle and additional statistics. The colour of the circle changes from green to yellow, then to orange, and finally to red as the health of the circuit decreases. The size indicates the traffic volume – the more traffic the larger the circle grows. Additional statistics are provided regarding request rate, error rate, latency and much more (see image below). This provides you with a great real-time description of your system – and it comes for free.



circle color and size represent health and traffic volume
SubscriberGetAccount
200,545
0
19
94
0
0 %
Error percentage of last 10 seconds
Request rate
Host: 54.0/s
Cluster: 20,056.0/s
2 minutes of request rate to show relative changes in traffic
Circuit Closed
Circuit-breaker status
Hosts 370
Median 1ms
Mean 4ms
90th 10ms
99th 44ms
99.5th 61ms
hosts reporting from cluster
last minute latency percentiles
Rolling 10 second counters with 1 second granularity
Successes 200,545
Short-circuited (rejected) 0
19 Thread timeouts
94 Thread-pool Rejections
0 Failures/Exceptions

Monitoring your circuit breakers with the Hystrix dashboard [2]

Adapting Hystrix to your needs

Hystrix offers you the capabilities to adapt the behaviour of the circuit breakers to your needs. It is, however, recommended to start with the default configuration and monitor the behaviour of the system in production. As it turns out, the default behaviour is viable for most applications. Nonetheless, I would like to demonstrate to you how easy it is to change some basic configuration parameters. Let’s say you want to change the size of your thread-pools to allow for 40 threads to be executed concurrently:

```
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                                 .asKey("MarketDataFeedsGroup"))
                                 .andThreadPoolPropertiesDefaults(
HystrixThreadPoolProperties.Setter()
                                                 .withCoreSize(40)));
```

Or you could change the time-out settings of the circuit. The default time-out is set to 1000 ms. If you have an application with a high request rate, you might want to avoid having your threads blocked for such a large amount of time. If you want to wait for no longer than 200 ms, you could apply the following configuration:

```
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                                 .asKey("MarketDataFeedsGroup"))
                                 .andCommandPropertiesDefaults(
HystrixCommandProperties.Setter()
.withExecutionTimeoutInMilliseconds(200)));
```

It is also possible to configure the Hystrix dashboard. It may be desirable to extend the size of the statistical rolling window defining for how long the statistics are kept persistent. The default configuration only takes the last ten seconds into account, when calculating error rates, for example. If you would like to review the last 60 seconds, you simply have to add the following configuration to your command:

```
super(Setter.withGroupKey(HystrixCommandGroupKey.Factory
                                .asKey("MarketDataFeedsGroup"))
                                .andCommandPropertiesDefaults(
HystrixCommandProperties.Setter()
.withMetricsRollingStatisticalWindowInMilliseconds(
60000)));
```

Further reading

I hope that this blog post has given you some useful insights into the Hystrix framework. If you want to learn more, do not hesitate to get in touch with me. In addition to that, you should check out the following sites:

- Hystrix on GitHub
- Hystrix wiki
- Comprehensive list of configuration parameters
- Hystrix dashboard wiki
- Zühlke Github Site

Resources

[1] https://raw.githubusercontent.com/wiki/Netflix/Hystrix/images/hystrix-command-flow-chart.png
[2] https://github.com/Netflix/Hystrix/wiki/images/dashboard-annoted-circuit-640.png