Hystrix: Latency and Fault Tolerance for Distributed Systems

What Is Hystrix For?

Hystrix is designed to do the following:

- Give protection from and control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.

- Stop cascading failures in a complex distributed system.

- Fail fast and rapidly recover.

- Fallback and gracefully degrade when possible.

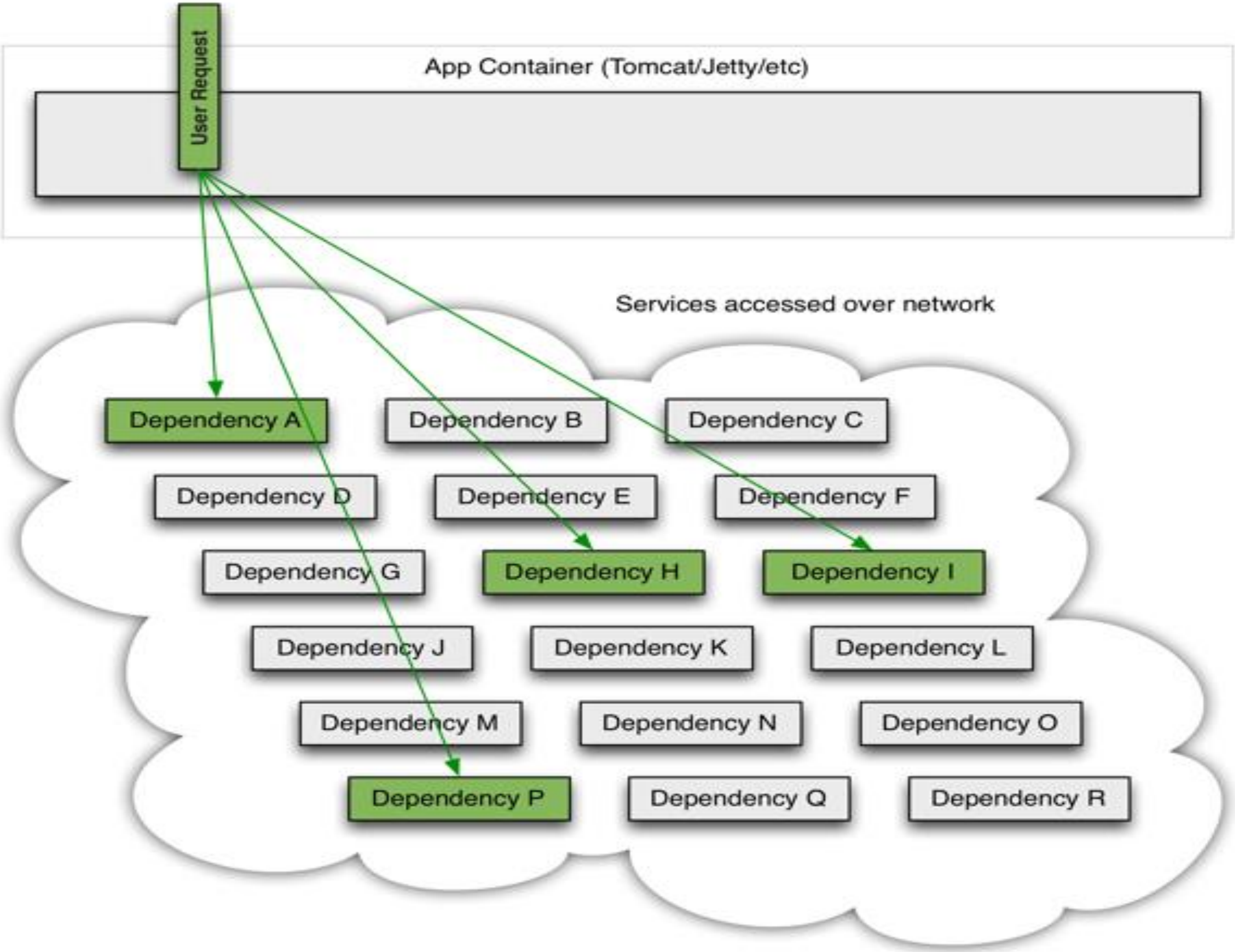
- Enable near real-time monitoring, alerting, and operational control.

What Problem Does Hystrix Solve?

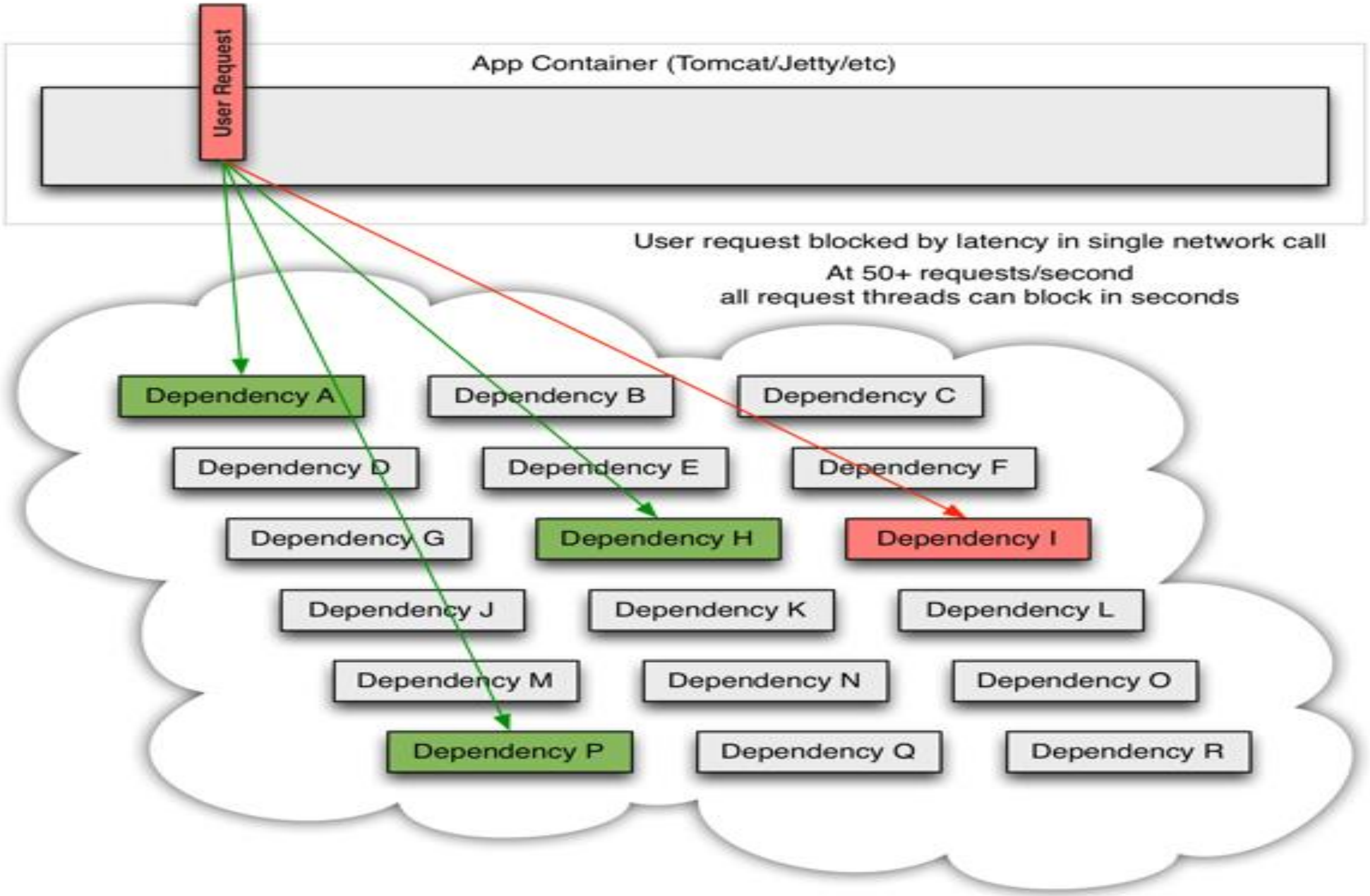
Applications in complex distributed architectures have dozens of dependencies, each of which will inevitably fail at some point.

If the host application is not isolated from these external failures, it risks being taken down with them.

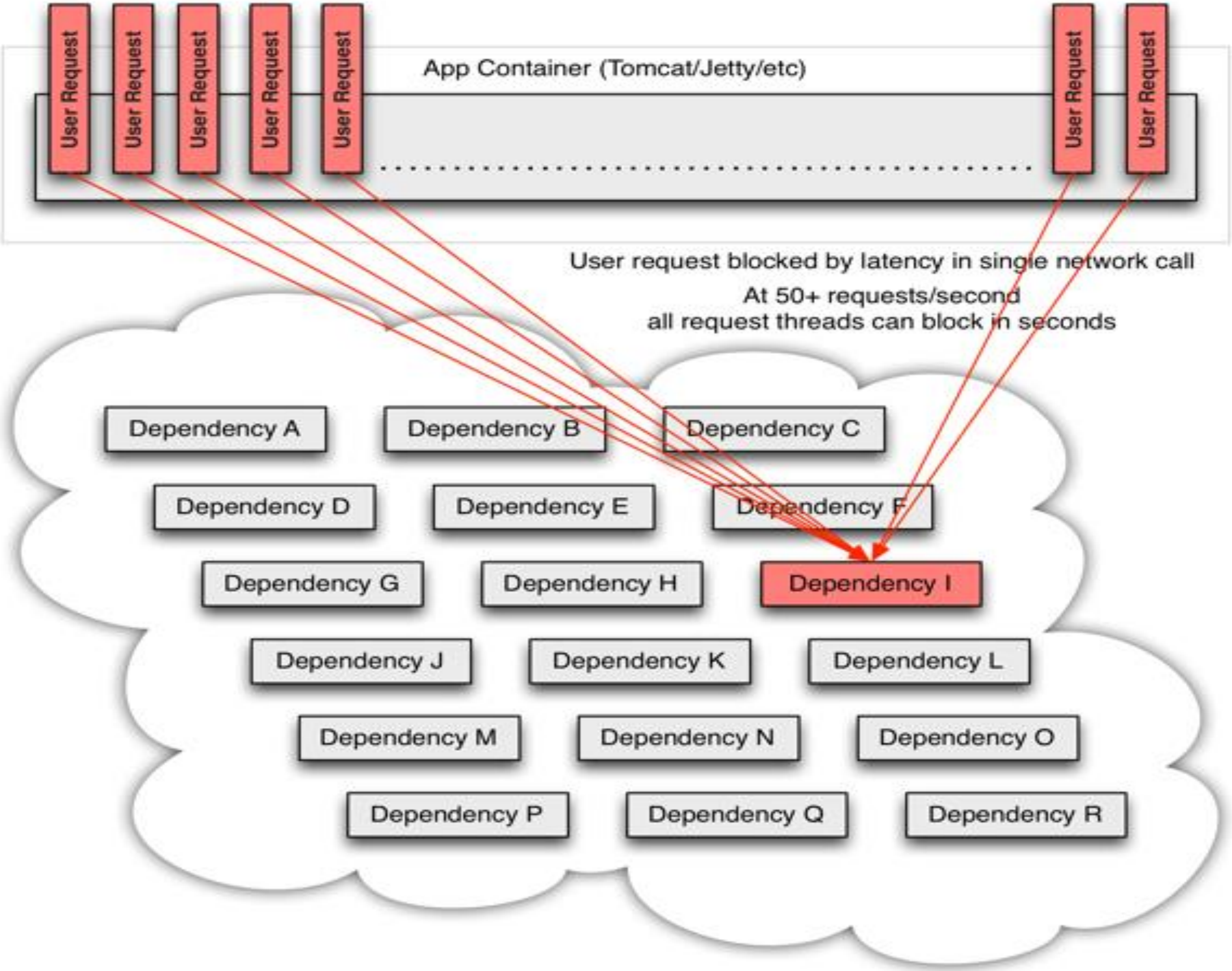
When everything is healthy the request flow can look like this:



When one of many backend systems becomes latent it can block the entire user request:



With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.



These issues are even worse when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.

All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

What Design Principles Underlie Hystrix?

❑ Preventing any single dependency from using up all container (such as Tomcat) user threads.

❑ Using isolation circuit breaker patterns to limit the impact of any one dependency.

❑ Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting

❑ Optimizing for time-to-recovery by means of low latency propagation of configuration changes and support for dynamic property changes in most aspects of Hystrix, which allows you to make real-time operational modifications with low latency feedback loops.

Pattern: Circuit Breaker

Problem

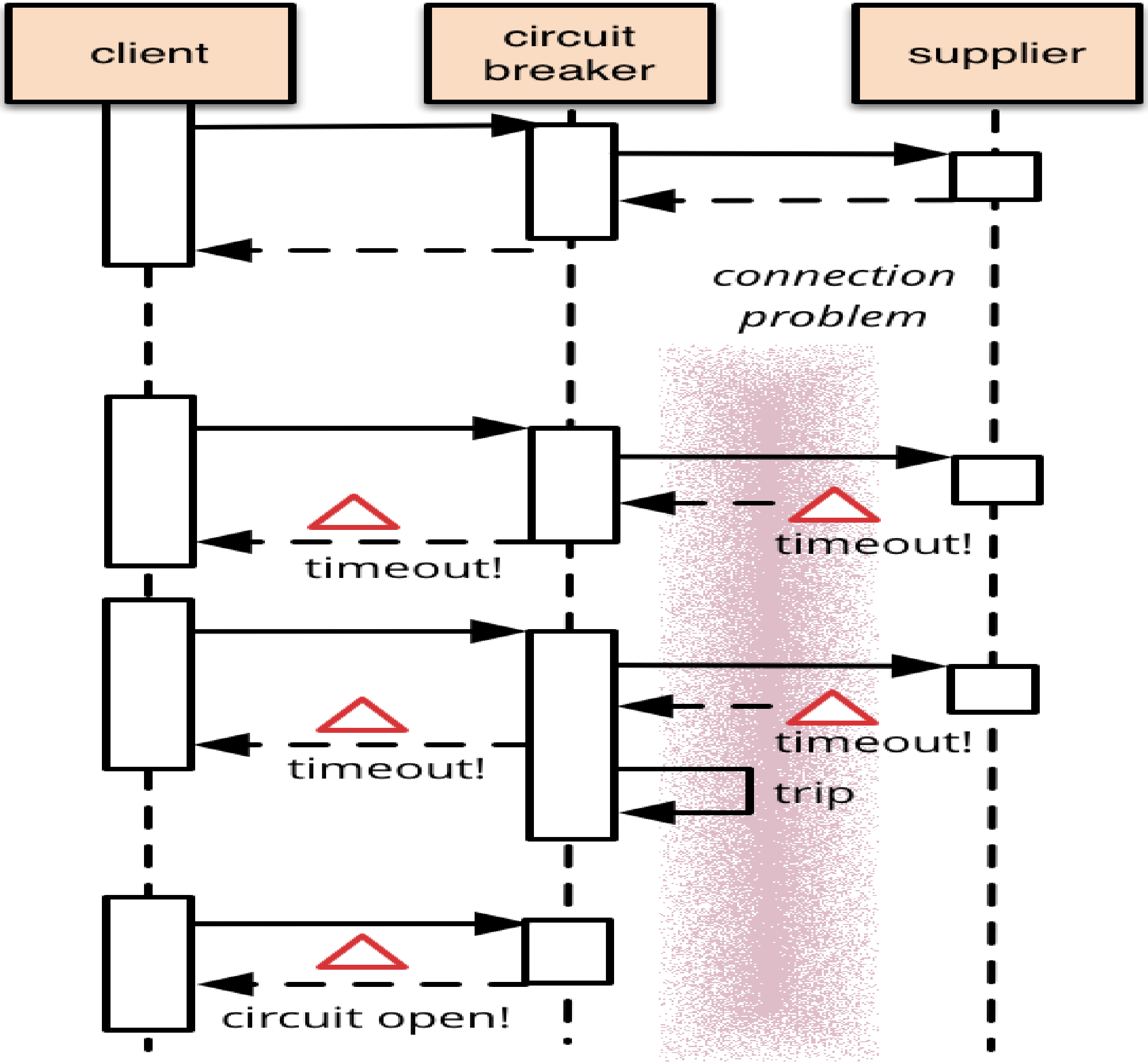
How to prevent a network or service failure from cascading to other services?

Solution

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

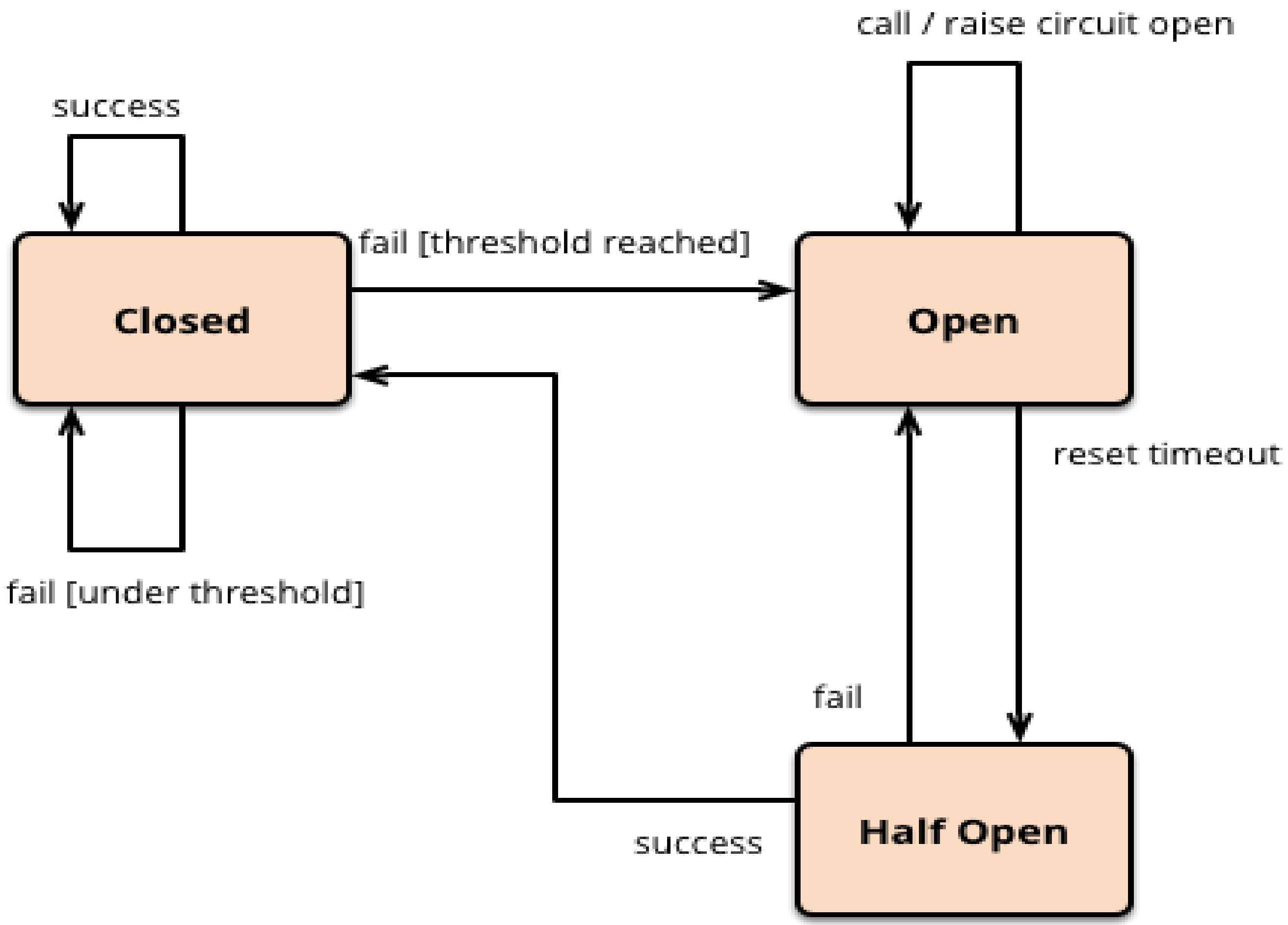

client
circuit breaker
supplier
connection problem
timeout!
timeout!
timeout!
timeout!
trip
circuit open!

Hystrix can wrap methods with circuit breaker that uses the following logic :

- ❑ By default the circuit is closed and the original method is executed.
- ❑ The original method throws an exception, the fallback is executed.
- ❑ Error rate hits the threshold, the circuit opens.
- ❑ When the circuit is open the original method is not executed anymore, only the fallback.
- ❑ After a predefined amount of time the circuit is closed, and the flow starts from the beginning.

The basic idea behind the circuit breaker is very simple.

We wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all.


call / raise circuit open
success
fail [threshold reached]
Closed
Open
fail [under threshold]
reset timeout
fail
success
Half Open

A service failure in the lower level of services can cause cascading failure all the way up to the user.

When calls to a particular service is greater than circuitBreaker.requestVolumeThreshold (default: 20 requests) and failure percentage is greater than circuitBreaker.errorThresholdPercentage (default: >50%) in a rolling window defined by metrics.rollingStats.timeInMilliseconds (default: 10 seconds), the circuit opens and the call is not made.

In cases of error and an open circuit a fallback can be provided by the developer.

How to Include Hystrix

To include Hystrix in your project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-netflix-hystrix

```
@SpringBootApplication
@EnableCircuitBreaker
public class Application {
   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }   }

@Component
public class StoreIntegration {

   @HystrixCommand(fallbackMethod = "defaultStores")
   public Object getStores(Map<String, Object> parameters) {
      //do stuff that might fail
   }

   public Object defaultStores(Map<String, Object> parameters) {
      return /* something useful */;
   } }
```

```
@HystrixCommand(commandProperties = {
        @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value =
"10000"),
        @HystrixProperty(name = "metrics.rollingStats.timeInMilliseconds", value =
"10000"),
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value =
"5"),
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value =
"100")
  },
        fallbackMethod = "fallbackCall")
```

Execution

The following Properties control how HystrixCommand.run() executes.

execution.isolation.strategy

This property indicates which isolation strategy HystrixCommand.run() executes with, one of the following two choices:

THREAD — it executes on a separate thread and concurrent requests are limited by the number of threads in the thread-pool

SEMAPHORE — it executes on the calling thread and concurrent requests are limited by the semaphore count

Thread or Semaphore

The default, and the recommended setting, is to run HystrixCommands using thread isolation (THREAD) and HystrixObservableCommands using semaphore isolation (SEMAPHORE).

Commands executed in threads have an extra layer of protection against latencies beyond what network timeouts can offer.

Generally the only time we should use semaphore isolation for HystrixCommands is when the call is so high volume (hundreds per second, per instance) that the overhead of separate threads is too high; this typically only applies to non-network calls.

Isolation strategy

Default Value	THREAD
Possible Values	THREAD, SEMAPHORE
Default Property	hystrix.command.default.execution.isolation.strategy

fallback.enabled

This property determines whether a call to HystrixCommand.getFallback() will be attempted when failure or rejection occurs.

Default Value	true
Default Property	hystrix.command.default.fallback.enabled

circuitBreaker.requestVolumeThreshold

This property sets the minimum number of requests in a rolling window that will trip the circuit.

For example, if the value is 20, then if only 19 requests are received in the rolling window (say a window of 10 seconds) the circuit will not trip open even if all 19 failed.

Default Value	20
Default Property	hystrix.command.default.circuitBreaker.requestVolumeThreshold

circuitBreaker.sleepWindowInMilliseconds

This property sets the amount of time, after tripping the circuit, to reject requests before allowing attempts again to determine if the circuit should again be closed.

Default Value	5000
Default Property	hystrix.command.default.circuitBreaker. sleepWindowInMilliseconds

circuitBreaker.errorThresholdPercentage

This property sets the error percentage at or above which the circuit should trip open and start short-circuiting requests to fallback logic.

Default Value	50
Default Property	hystrix.command.default.circuitBreaker. errorThresholdPercentage

Batching network requests with the Hystrix Collapser

Request collapser

One of the disadvantages of microservice architecture is the performance penalty caused by high number of cross service HTTP calls.

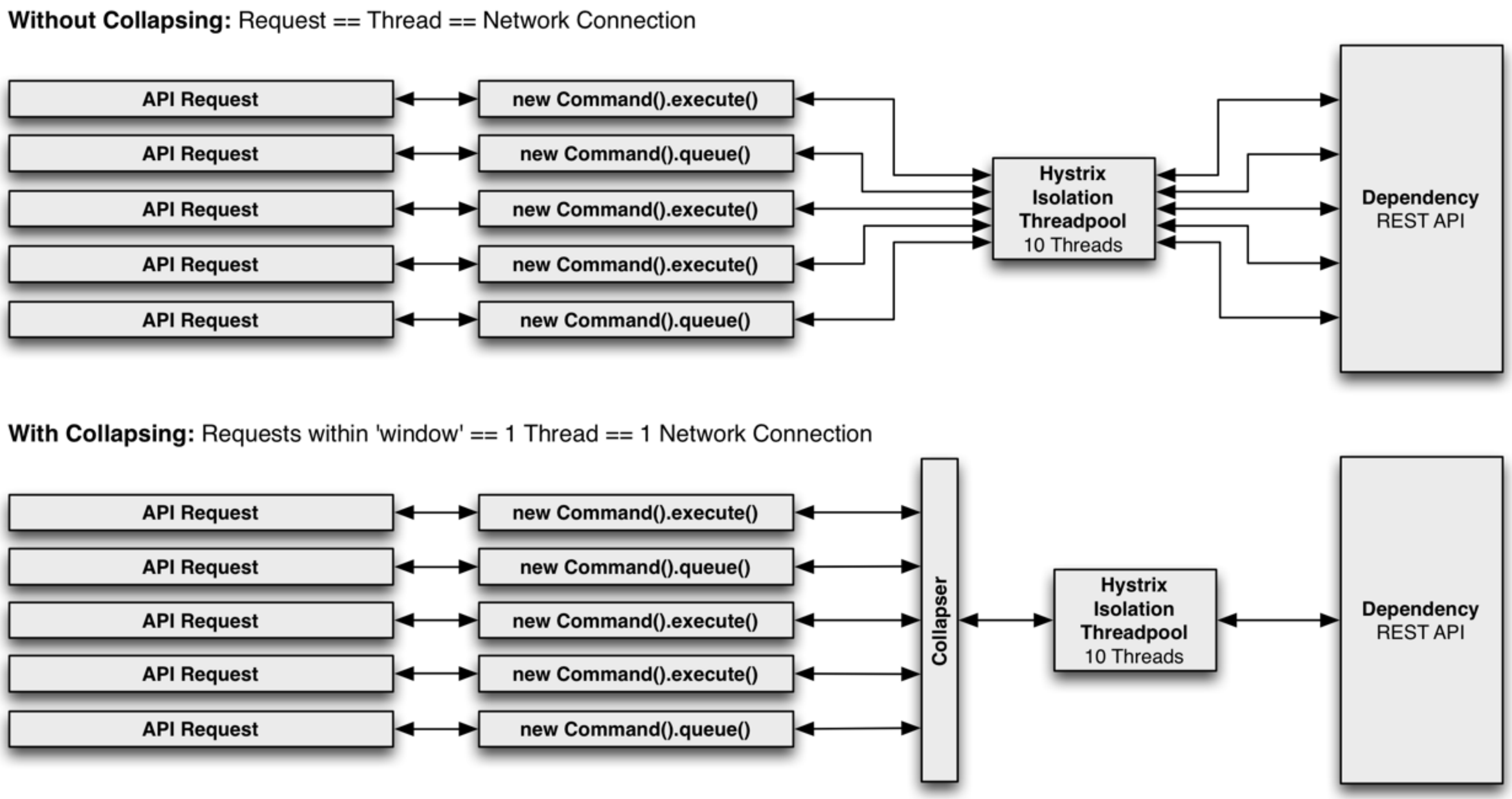
Advantage of batching the calls are :

- ❑ The communication overhead of multiple calls are reduced into a single call
- ❑ The backend may be able to more efficiently serve a single request with more parameters then multiple requests
- ❑ If there are multiple requests with the same parameter then only one of the requests have to be executed.

In the case of a normal Hystrix command, every executed command is given it's own thread context to execute its network transaction.

In the case of the collapser, commands are consumed within a configurable window of time and run together in one batch command.

This allows for multiple API requests to occur over a single network transaction.



The implementation is quite straightforward again.

First we define a method

❑ returning a Future

❑ Annotated with @HystrixController

The method body will never be actually executed, it can be left empty.

```
@HystrixCollapser(scope =
com.netflix.hystrix.HystrixCollapser.Scope.GLOBAL, batchMethod =
"getCustomerByIds")
public Future<MessageWrapper> getCustomerById(Integer id) {
   throw new RuntimeException("This method body should not be
executed");
}
```

Then we define the collapser method

❑ Taking a list of the same type as the original method
❑ Returning a list of the same type as the original method
❑ Annotated with HystrixCommand

```
@HystrixCommand
public List<MessageWrapper> getCustomerByIds(List<Integer> ids) {
```

This approach can be used not only to to collapse multiple HTTP calls, but for SQL queries or anything else.

Hystrix Metrics Stream

To enable the Hystrix metrics stream include a dependency on spring-boot-starter-actuator.

This will expose the /hystrix.stream as a management endpoint.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Circuit Breaker: Hystrix Dashboard

Hystrix Dashboard

One of the main benefits of Hystrix is the set of metrics it gathers about each HystrixCommand. The Hystrix Dashboard displays the health of each circuit breaker in an efficient manner.

How to Include Hystrix Dashboard

To include the Hystrix Dashboard in the project use the starter with group org.springframework.cloud and artifact id spring-cloud-starter-hystrix-netflix-dashboard.

To run the Hystrix Dashboard annotate the Spring Boot main class with @EnableHystrixDashboard. We can visit /hystrix and point the dashboard to an individual instances /hystrix.stream endpoint in a Hystrix client application.

Turbine (Aggregates Hystrix data)

Looking at an individual instances Hystrix data is not very useful in terms of the overall health of the system.

Turbine is an application that aggregates all of the relevant /hystrix.stream endpoints into a combined /turbine.stream for use in the Hystrix Dashboard.

Individual instances are located via Eureka.

Running Turbine is as simple as annotating your main class with the @EnableTurbine annotation (e.g. using spring-cloud-starter-netflix-turbine to set up the classpath).

The configuration key turbine.appConfig is a list of eureka serviceIds that turbine will use to lookup instances.

The turbine stream is then used in the Hystrix dashboard using a url that looks like:

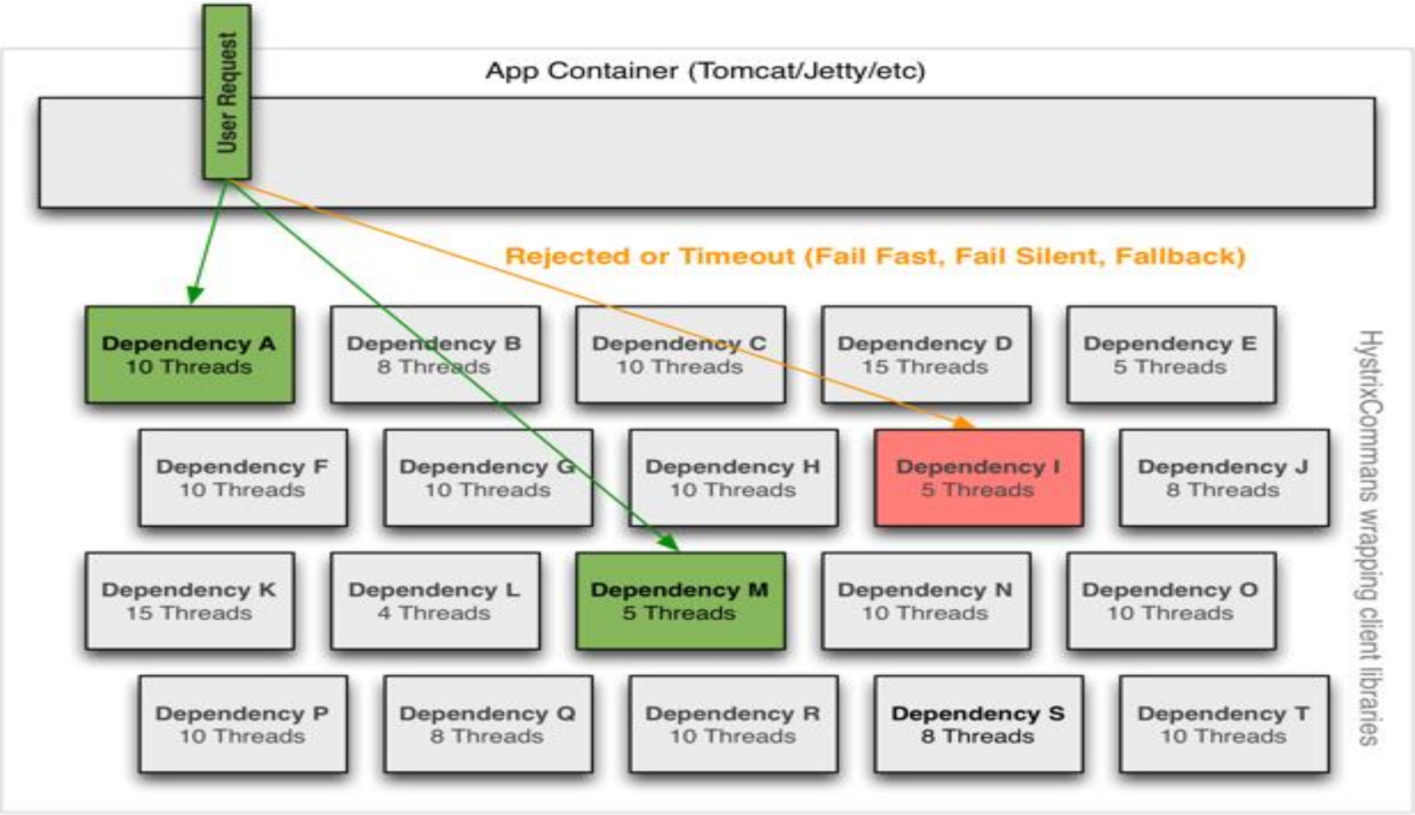
http://my.turbine.sever:8080/turbine.stream?cluster=CLUSTERNAME

```
turbine:
  aggregator:
    clusterConfig: CUSTOMERS
  appConfig: customers
```

example to work if there is an app registered with Eureka called "customers":

Isolation

Hystrix employs the bulkhead pattern to isolate dependencies from each other and to limit concurrent access to any one of them.

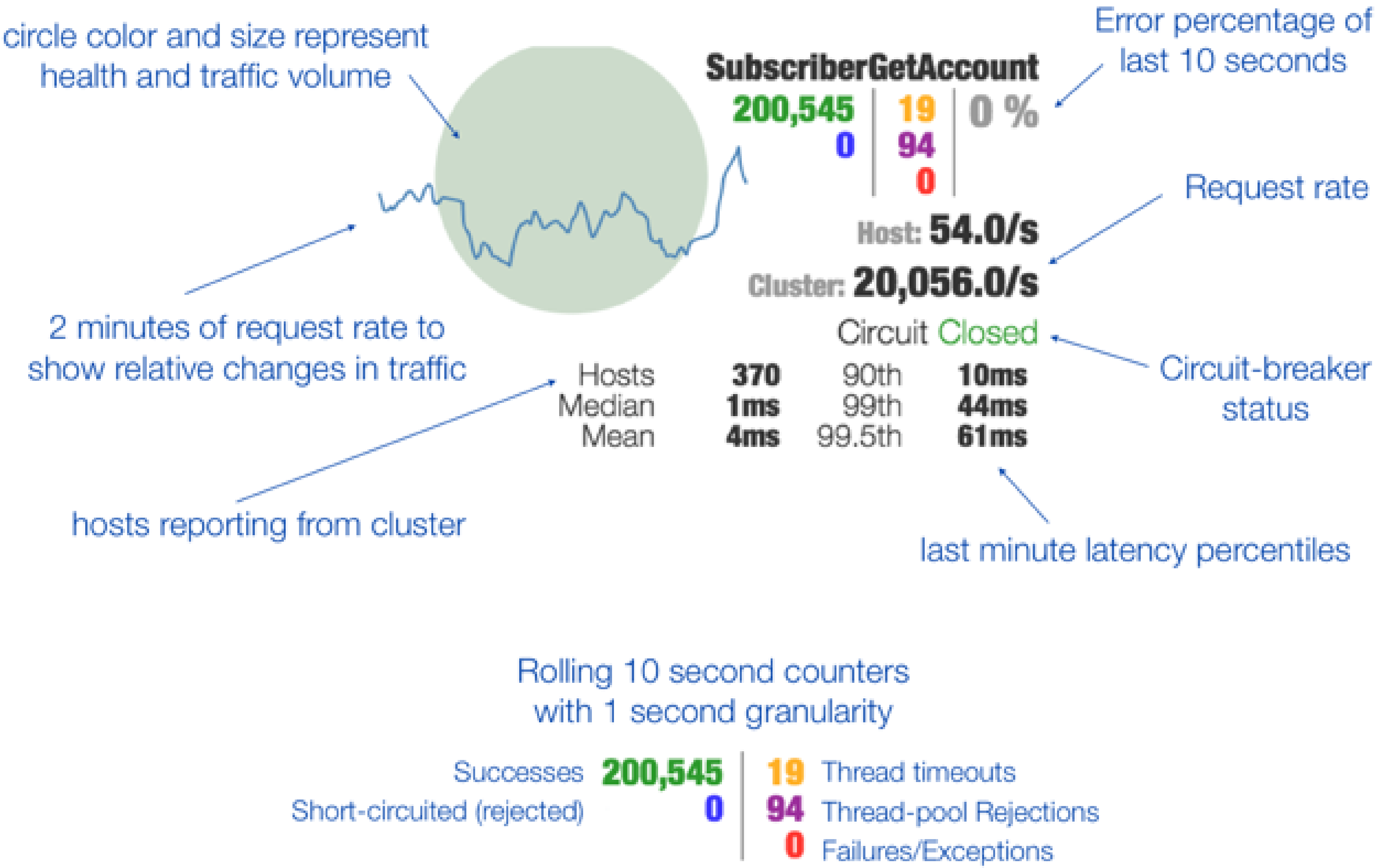


ThreadPool Properties

Most of the time the default value of 10 threads will be fine (often it could be made smaller).
To determine if it needs to be larger, a basic formula for calculating the size is:

requests per second at peak when healthy × 99th percentile latency in seconds + some breathing room

Hystrix Dashboard to monitor a single server



circle color and size represent health and traffic volume

2 minutes of request rate to show relative changes in traffic

hosts reporting from cluster

Error percentage of last 10 seconds

Request rate

Circuit-breaker status

last minute latency percentiles

SubscriberGetAccount

200,545 | **19** | **0 %**
0 | **94**
0

Host: 54.0/s
Cluster: 20,056.0/s

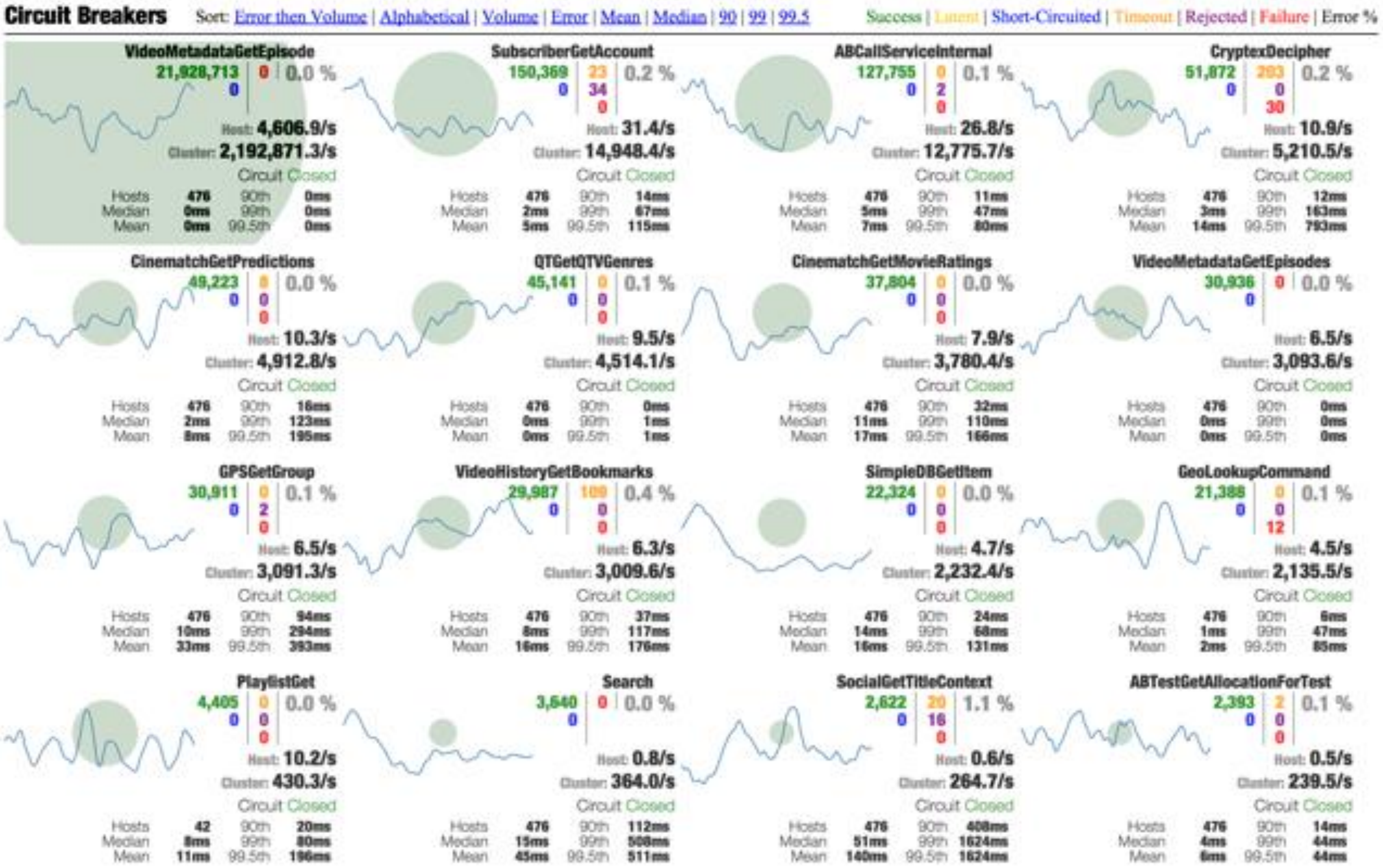
Circuit Closed

Hosts	**370**	90th	**10ms**
Median	**1ms**	99th	**44ms**
Mean	**4ms**	99.5th	**61ms**

Rolling 10 second counters with 1 second granularity

Successes **200,545** | **19** Thread timeouts
Short-circuited (rejected) **0** | **94** Thread-pool Rejections
0 Failures/Exceptions

Hystrix dashboard monitoring 476 servers aggregated using Turbine:



Circuit Breakers   Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5   Success | Latent | Short-Circuited | Timeout | Rejected | Failure | Error %

```
VideoMetadataGetEpisode
21,928,713 | 0 | 0.0 %
         0 |
Host: 4,606.9/s
Cluster: 2,192,871.3/s
Circuit Closed
Hosts   476  90th    0ms
Median  0ms  99th    0ms
Mean    0ms  99.5th  0ms

SubscriberGetAccount
150,369 | 23 | 0.2 %
      0 | 34 |
        |  0 |
Host: 31.4/s
Cluster: 14,948.4/s
Circuit Closed
Hosts   476  90th    14ms
Median  2ms  99th    67ms
Mean    5ms  99.5th  115ms

ABCallServiceInternal
127,755 | 0 | 0.1 %
      0 | 2 |
        | 0 |
Host: 26.8/s
Cluster: 12,775.7/s
Circuit Closed
Hosts   476  90th    11ms
Median  5ms  99th    47ms
Mean    7ms  99.5th  80ms

CryptexDecipher
51,872 | 293 | 0.2 %
     0 |   0 |
       |  30 |
Host: 10.9/s
Cluster: 5,210.5/s
Circuit Closed
Hosts   476   90th    12ms
Median  3ms   99th    163ms
Mean    14ms  99.5th  793ms

CinematchGetPredictions
49,223 | 0 | 0.0 %
     0 | 0 |
       | 0 |
Host: 10.3/s
Cluster: 4,912.8/s
Circuit Closed
Hosts   476  90th    16ms
Median  2ms  99th    123ms
Mean    8ms  99.5th  195ms

QTGetQTVGenres
45,141 | 0 | 0.1 %
     0 | 0 |
       | 0 |
Host: 9.5/s
Cluster: 4,514.1/s
Circuit Closed
Hosts   476  90th    0ms
Median  0ms  99th    1ms
Mean    0ms  99.5th  1ms

CinematchGetMovieRatings
37,804 | 0 | 0.0 %
     0 | 0 |
       | 0 |
Host: 7.9/s
Cluster: 3,780.4/s
Circuit Closed
Hosts   476   90th    32ms
Median  11ms  99th    110ms
Mean    17ms  99.5th  166ms

VideoMetadataGetEpisodes
30,936 | 0 | 0.0 %
     0 |
Host: 6.5/s
Cluster: 3,093.6/s
Circuit Closed
Hosts   476  90th    0ms
Median  0ms  99th    0ms
Mean    0ms  99.5th  0ms

GPSGetGroup
30,911 | 0 | 0.1 %
     0 | 2 |
       | 0 |
Host: 6.5/s
Cluster: 3,091.3/s
Circuit Closed
Hosts   476   90th    94ms
Median  10ms  99th    294ms
Mean    33ms  99.5th  393ms

VideoHistoryGetBookmarks
29,987 | 109 | 0.4 %
     0 |   0 |
       |   0 |
Host: 6.3/s
Cluster: 3,009.6/s
Circuit Closed
Hosts   476   90th    37ms
Median  8ms   99th    117ms
Mean    16ms  99.5th  176ms

SimpleDBGetItem
22,324 | 0 | 0.0 %
     0 | 0 |
       | 0 |
Host: 4.7/s
Cluster: 2,232.4/s
Circuit Closed
Hosts   476   90th    24ms
Median  14ms  99th    68ms
Mean    16ms  99.5th  131ms

GeoLookupCommand
21,388 |  0 | 0.1 %
     0 |  0 |
       | 12 |
Host: 4.5/s
Cluster: 2,135.5/s
Circuit Closed
Hosts   476  90th    6ms
Median  1ms  99th    47ms
Mean    2ms  99.5th  85ms

PlaylistGet
4,405 | 0 | 0.0 %
    0 | 0 |
      | 0 |
Host: 10.2/s
Cluster: 430.3/s
Circuit Closed
Hosts   42    90th    20ms
Median  8ms   99th    80ms
Mean    11ms  99.5th  196ms

Search
3,640 | 0 | 0.0 %
    0 |
Host: 0.8/s
Cluster: 364.0/s
Circuit Closed
Hosts   476   90th    112ms
Median  15ms  99th    508ms
Mean    45ms  99.5th  511ms

SocialGetTitleContext
2,622 | 20 | 1.1 %
    0 | 16 |
      |  0 |
Host: 0.6/s
Cluster: 264.7/s
Circuit Closed
Hosts   476    90th    408ms
Median  51ms   99th    1624ms
Mean    140ms  99.5th  1624ms

ABTestGetAllocationForTest
2,393 | 2 | 0.1 %
    0 | 0 |
      | 0 |
Host: 0.5/s
Cluster: 239.5/s
Circuit Closed
Hosts   476  90th    14ms
Median  4ms  99th    44ms
Mean    6ms  99.5th  44ms
```