Hystrix

Raúl Estrada

Octubre 2020

1. Overview

A typical distributed system consists of many services collaborating together.

These services are prone to failure or delayed responses. If a service fails it may impact on other services affecting performance and possibly making other parts of application inaccessible or in the worst case bring down the whole application.

Of course, there are solutions available that help make applications resilient and fault tolerant – one such framework is Hystrix.

The Hystrix framework library helps to control the interaction between services by providing fault tolerance and latency tolerance. It improves overall resilience of the system by isolating the failing services and stopping the cascading effect of failures.

In this series of posts we will begin by looking at how Hystrix comes to the rescue when a service or system fails and what Hystrix can accomplish in these circumstances.

2. Simple Example

The way Hystrix provides fault and latency tolerance is to isolate and wrap calls to remote services.

In this simple example we wrap a call in the *run()* method of the *HystrixCommand:*

```java
class CommandHelloWorld extends HystrixCommand<String> {

    private String name;

    CommandHelloWorld(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        return "Hello " + name + "!";
    }
}
```

and we execute the call as follows:

```java
@Test
public void givenInputBobAndDefaultSettings_whenCommandExecuted_thenReturnHelloBob(){
    assertThat(new CommandHelloWorld("Bob").execute(), equalTo("Hello Bob!"));
}
```

3. Maven Setup

To use Hystrix in a Maven projects, we need to have *hystrix-core* and *rxjava-core* dependency from Netflix in the project *pom.xml*:

```
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-core</artifactId>
    <version>1.5.4</version>
</dependency>
```

The latest version can always be found here.

```
<dependency>
    <groupId>com.netflix.rxjava</groupId>
    <artifactId>rxjava-core</artifactId>
    <version>0.20.7</version>
</dependency>
```

The latest version of this library can always be found here.

4. Setting up Remote Service

Let's start by simulating a real world example.

In the example below, the class *RemoteServiceTestSimulator* represents a service on a remote server. It has a method which responds with a message after the given period of time. We can imagine that this wait is a simulation of a time consuming process at the remote system resulting in a delayed response to the calling service:

```java
class RemoteServiceTestSimulator {

    private long wait;

    RemoteServiceTestSimulator(long wait) throws InterruptedException {
        this.wait = wait;
    }

    String execute() throws InterruptedException {
        Thread.sleep(wait);
        return "Success";
    }
}
```

And here is our sample client that calls the *RemoteServiceTestSimulator*.

The call to the service is isolated and wrapped in the *run()* method of a *HystrixCommand*. Its this wrapping that provides the resilience we touched upon above:

```
class RemoteServiceTestCommand extends HystrixCommand<String> {

    private RemoteServiceTestSimulator remoteService;

    RemoteServiceTestCommand(Setter config, RemoteServiceTestSimulator remoteService) {
        super(config);
        this.remoteService = remoteService;
    }

    @Override
    protected String run() throws Exception {
        return remoteService.execute();
    }
}
```

The call is executed by calling the *execute()* method on an instance of the *RemoteServiceTestCommand* object.

The following test demonstrates how this is done:

```java
@Test
public void givenSvcTimeoutOf100AndDefaultSettings_whenRemoteSvcExecuted_thenReturnSuccess()
  throws InterruptedException {

    HystrixCommand.Setter config = HystrixCommand
      .Setter
      .withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceGroup2"));

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(100)).execute(),
      equalTo("Success"));
}
```

So far we have seen how to wrap remote service calls in the *HystrixCommand* object. In the section below let's look at how to deal with a situation when the remote service starts to deteriorate.

5. Working With Remote Service and Defensive Programming

5.1. Defensive Programming With Timeout

It is general programming practice to set timeouts for calls to remote services.

Let's begin by looking at how to set timeout on *HystrixCommand* and how it helps by short circuiting:

```java
@Test
public void givenSvcTimeoutOf5000AndExecTimeoutOf10000_whenRemoteSvcExecuted_thenReturnSuccess()
  throws InterruptedException {

    HystrixCommand.Setter config = HystrixCommand
      .Setter
      .withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceGroupTest4"));

    HystrixCommandProperties.Setter commandProperties = HystrixCommandProperties.Setter();
    commandProperties.withExecutionTimeoutInMilliseconds(10_000);
    config.andCommandPropertiesDefaults(commandProperties);

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(500)).execute(),
      equalTo("Success"));
}
```

In the above test, we are delaying the service's response by setting the timeout to 500 ms. We are also setting the execution timeout on *HystrixCommand* to be 10,000 ms, thus allowing sufficient time for the remote service to respond.

Now let's see what happens when the execution timeout is less than the service timeout call:

```java
@Test(expected = HystrixRuntimeException.class)
public void givenSvcTimeoutOf15000AndExecTimeoutOf5000_whenRemoteSvcExecuted_thenExpectHre()
  throws InterruptedException {

    HystrixCommand.Setter config = HystrixCommand
      .Setter
      .withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceGroupTest5"));

    HystrixCommandProperties.Setter commandProperties = HystrixCommandProperties.Setter();
    commandProperties.withExecutionTimeoutInMilliseconds(5_000);
    config.andCommandPropertiesDefaults(commandProperties);

    new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(15_000)).execute();
}
```

Notice how we've lowered the bar and set the execution timeout to 5,000 ms.

We are expecting the service to respond within 5,000 ms, whereas we have set the service to respond after 15,000 ms. If you notice when you execute the test, the test will exit after 5,000 ms instead of waiting for 15,000 ms and will throw a *HystrixRuntimeException.*

This demonstrates how Hystrix does not wait longer than the configured timeout for a response. This helps make the system protected by Hystrix more responsive.

In the below sections we will look into setting thread pool size which prevents threads being exhausted and we will discuss its benefit.

5.2. Defensive Programming With Limited Thread Pool

Setting timeouts for service call does not solve all the issues associated with remote services.

When a remote service starts to respond slowly, a typical application will continue to call that remote service.

The application doesn't know if the remote service is healthy or not and new threads are spawned every time a request comes in. This will cause threads on an already struggling server to be used.

We don't want this to happen as we need these threads for other remote calls or processes running on our server and we also want to avoid CPU utilization spiking up.

Let's see how to set the thread pool size in *HystrixCommand*:

```java
@Test
public void givenSvcTimeoutOf500AndExecTimeoutOf10000AndThreadPool_whenRemoteSvcExecuted
  _thenReturnSuccess() throws InterruptedException {

    HystrixCommand.Setter config = HystrixCommand
      .Setter
      .withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceGroupThreadPool"));

    HystrixCommandProperties.Setter commandProperties = HystrixCommandProperties.Setter();
    commandProperties.withExecutionTimeoutInMilliseconds(10_000);
    config.andCommandPropertiesDefaults(commandProperties);
    config.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()
      .withMaxQueueSize(10)
      .withCoreSize(3)
      .withQueueSizeRejectionThreshold(10));

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(500)).execute(),
      equalTo("Success"));
}
```

In the above test, we are setting the maximum queue size, the core queue size and the queue rejection size. *Hystrix* will start rejecting the requests when the maximum number of threads have reached 10 and the task queue has reached a size of 10.

The core size is the number of threads that always stay alive in the thread pool.

5.3. Defensive Programming With Short Circuit Breaker Pattern

However, there is still an improvement that we can make to remote service calls.

Let's consider the case that the remote service has started failing.

We don't want to keep firing off requests at it and waste resources. We would ideally want to stop making requests for a certain amount of time in order to give the service time to recover before then resuming requests. This is what is called the *Short Circuit Breaker* pattern.

Let's see how Hystrix implements this pattern:

```java
@Test
public void givenCircuitBreakerSetup_whenRemoteSvcCmdExecuted_thenReturnSuccess()
  throws InterruptedException {

    HystrixCommand.Setter config = HystrixCommand
      .Setter
      .withGroupKey(HystrixCommandGroupKey.Factory.asKey("RemoteServiceGroupCircuitBreaker"));

    HystrixCommandProperties.Setter properties = HystrixCommandProperties.Setter();
    properties.withExecutionTimeoutInMilliseconds(1000);
    properties.withCircuitBreakerSleepWindowInMilliseconds(4000);
    properties.withExecutionIsolationStrategy
     (HystrixCommandProperties.ExecutionIsolationStrategy.THREAD);
    properties.withCircuitBreakerEnabled(true);
    properties.withCircuitBreakerRequestVolumeThreshold(1);

    config.andCommandPropertiesDefaults(properties);
    config.andThreadPoolPropertiesDefaults(HystrixThreadPoolProperties.Setter()
      .withMaxQueueSize(1)
      .withCoreSize(1)
      .withQueueSizeRejectionThreshold(1));

    assertThat(this.invokeRemoteService(config, 10_000), equalTo(null));
    assertThat(this.invokeRemoteService(config, 10_000), equalTo(null));
    assertThat(this.invokeRemoteService(config, 10_000), equalTo(null));

    Thread.sleep(5000);

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(500)).execute(),
      equalTo("Success"));

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(500)).execute(),
      equalTo("Success"));

    assertThat(new RemoteServiceTestCommand(config, new RemoteServiceTestSimulator(500)).execute(),
      equalTo("Success"));
}
```

```java
public String invokeRemoteService(HystrixCommand.Setter config, int timeout)
  throws InterruptedException {

    String response = null;

    try {
        response = new RemoteServiceTestCommand(config,
          new RemoteServiceTestSimulator(timeout)).execute();
    } catch (HystrixRuntimeException ex) {
        System.out.println("ex = " + ex);
    }

    return response;
}
```

In the above test we have set different circuit breaker properties. The most important ones are:

- The *CircuitBreakerSleepWindow* which is set to 4,000 ms. This configures the circuit breaker window and defines the time interval after which the request to the remote service will be resumed
- The *CircuitBreakerRequestVolumeThreshold* which is set to 1 and defines the minimum number of requests needed before the failure rate will be considered

With the above settings in place, our *HystrixCommand* will now trip open after two failed request. The third request will not even hit the remote service even though we have set the service delay to be 500 ms, *Hystrix* will short circuit and our method will return *null* as the response.

We will subsequently add a *Thread.sleep(5000)* in order to cross the limit of the sleep window that we have set. This will cause *Hystrix* to close the circuit and the subsequent requests will flow through successfully.

6. Conclusion

In summary Hystrix is designed to:

1. Provide protection and control over failures and latency from services typically accessed over the network
2. Stop cascading of failures resulting from some of the services being down
3. Fail fast and rapidly recover
4. Degrade gracefully where possible
5. Real time monitoring and alerting of command center on failures

In the next post we will see how to combine the benefits of Hystrix with the Spring framework.