Pattern: Circuit Breaker

Context

You have applied the Microservice architecture (../microservices.html). Services sometimes collaborate when handling requests. When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable. Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

Problem

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

Forces

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.

Example

`RegistrationServiceProxy` from the Microservices Example application (https://github.com/cer/microservices-examples) is an example of a component, which is written in Scala, that uses a circuit breaker to handle failures when invoking a remote service.

```
@Component
class RegistrationServiceProxy @Autowired()(restTemplate: RestTemplate) extends RegistrationService {

  @Value("${user_registration_url}")
  var userRegistrationUrl: String = _

  @HystrixCommand(commandProperties=Array(new HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="800")))
  override def registerUser(emailAddress: String, password: String): Either[RegistrationError, String] = {
    try {
      val response = restTemplate.postForEntity(userRegistrationUrl,
        RegistrationBackendRequest(emailAddress, password),
        classOf[RegistrationBackendResponse])
      response.getStatusCode match {
        case HttpStatus.OK =>
          Right(response.getBody.id)
      }
    } catch {
      case e: HttpClientErrorException if e.getStatusCode == HttpStatus.CONFLICT =>
        Left(DuplicateRegistrationError)
    }
  }
}
```

The `@HystrixCommand` arranges for calls to `registerUser()` to be executed using a circuit breaker.

The circuit breaker functionality is enabled using the `@EnableCircuitBreaker` annotation on the `UserRegistrationConfiguration` class.

```
@EnableCircuitBreaker
class UserRegistrationConfiguration {
```

Resulting Context

This pattern has the following benefits:

- Services handle the failure of the services that they invoke

This pattern has the following issues:

- It is challenging to choose timeout values without creating false positives or introducing excessive latency.

Related patterns

- The Microservice Chassis (../microservice-chassis.html) might implement this pattern
- An API Gateway (../apigateway.html) will use this pattern to invoke services
- A Server-side discovery (../server-side-discovery.html) router might use this pattern to invoke services

See also

- Netflix Hystrix (https://github.com/Netflix/Hystrix) is an example of a library that implements this pattern