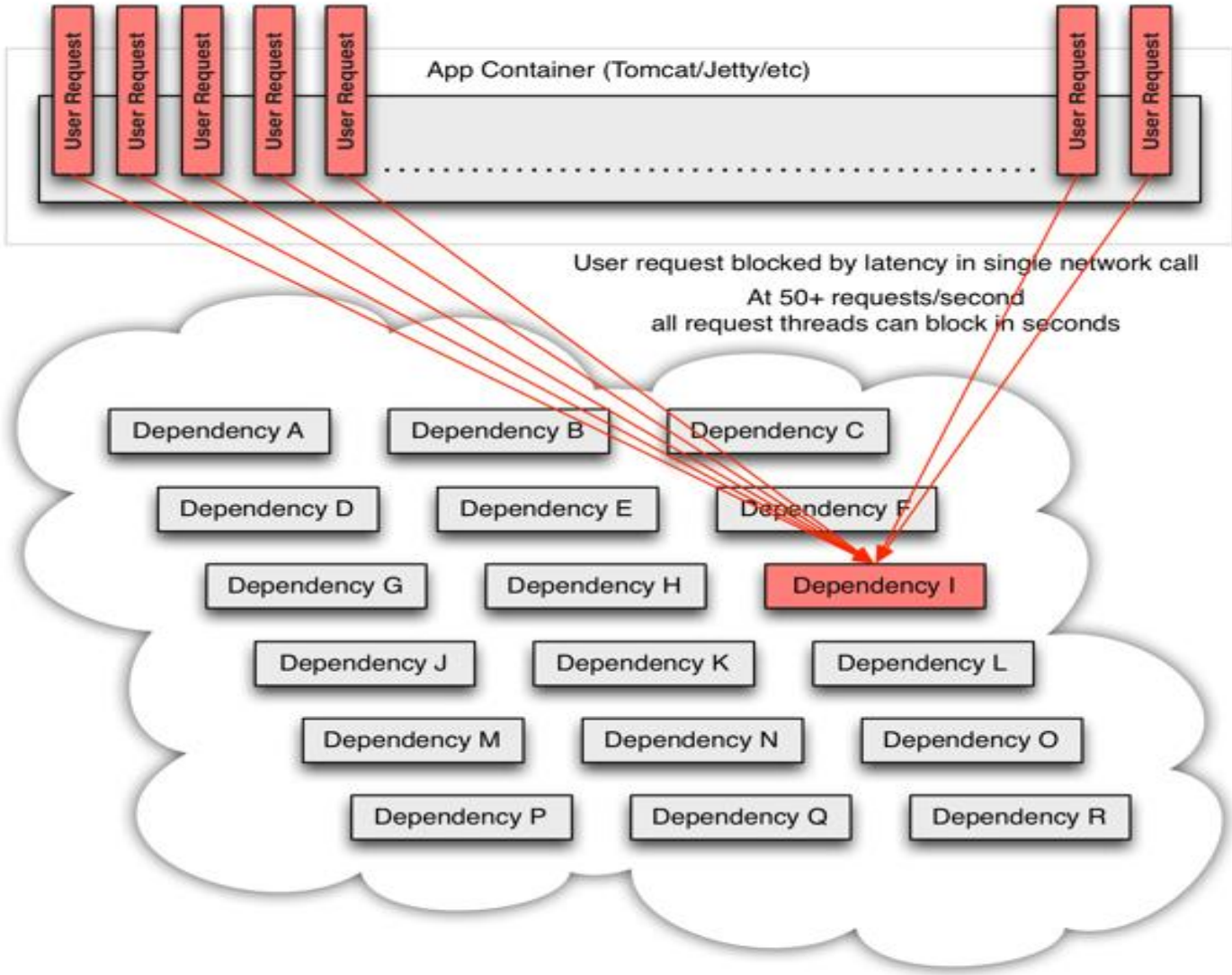
Latency and Fault Tolerance for Distributed Systems
(Circuit Breaker Pattern – Resilience4j)

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.

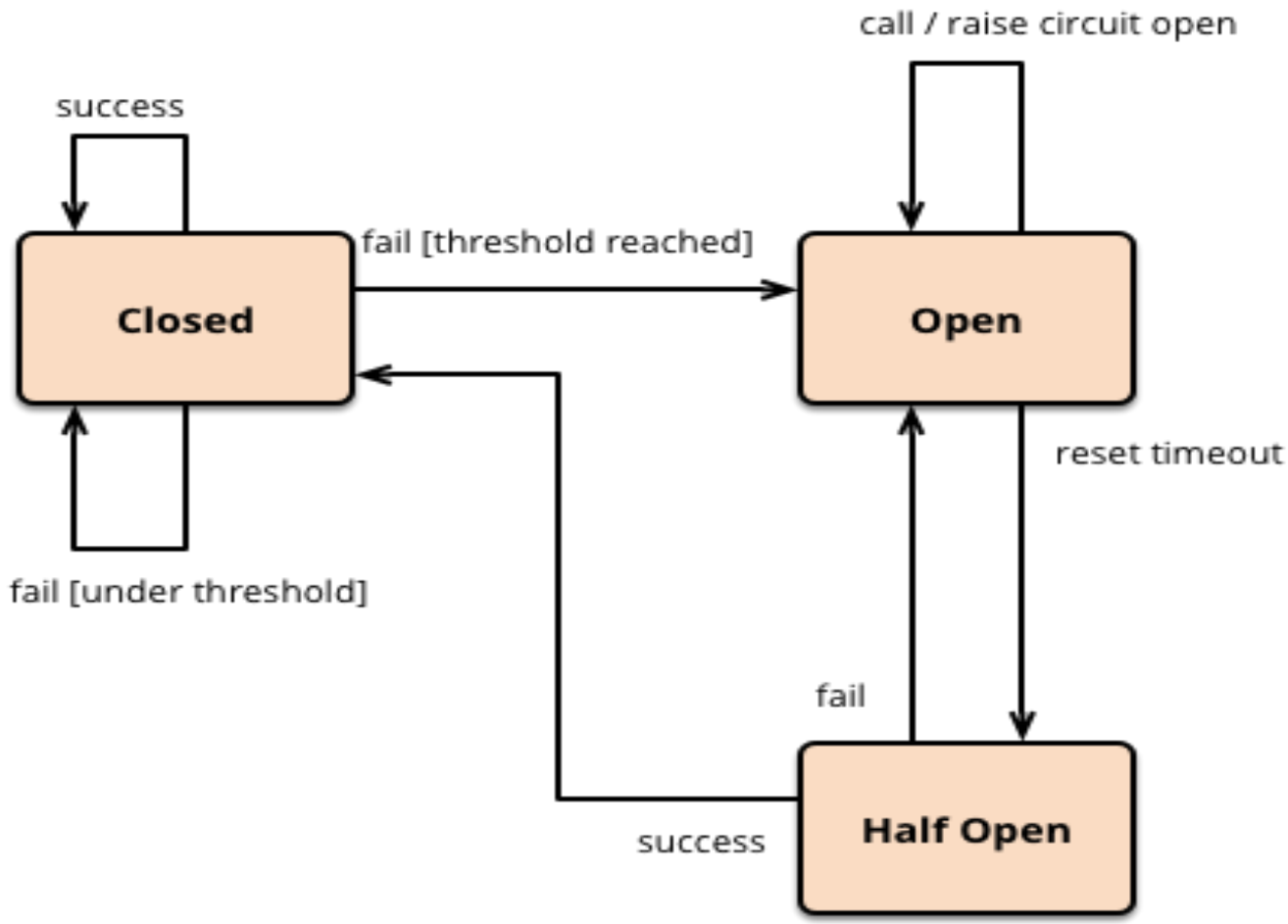
CircuitBreaker

When a service invokes another service, there is always a possibility that it may be down or having high latency.

This may lead to exhaustion of the threads as they might be waiting for other requests to complete.

This pattern functions in a similar fashion to an electrical Circuit Breaker:

- When a number of consecutive failures cross the defined threshold, the Circuit Breaker trips.
- For the duration of the timeout period, all requests invoking 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.


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

```java
@GetMapping("/")
@CircuitBreaker(name ="SimpleService", fallbackMethod = "getDefaultInfo" )
public String getInfo()
{
    String msg = restTemplate.getForObject("http://SimpleService/",String.class);
    return "Unshakable "+msg;
}
```

```yaml
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowType: COUNT_BASED
      slidingWindowSize: 100
      permittedNumberOfCallsInHalfOpenState: 10
      waitDurationInOpenState: 10
      failureRateThreshold: 60
      registerHealthIndicator: true
```

RateLimiter

Rate Limiting pattern ensures that a service accepts only a defined maximum number of requests during a window.

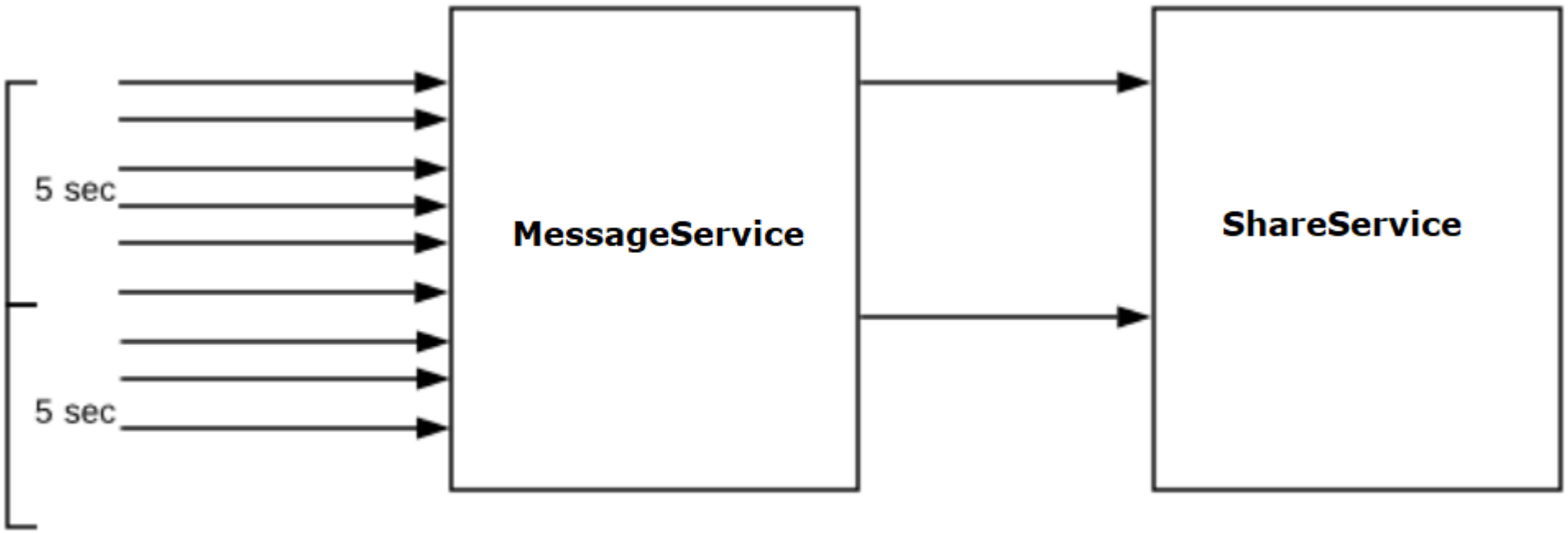
This ensures that underline resources are used as per their limits and don't exhaust.

Retry

Retry pattern enables an application to handle transient failures while calling to external services.

It ensures retrying operations on external resources a set number of times. If it doesn't succeed after all the retry attempts, it should fail and response should be handled gracefully by the application.

RateLimiter



```java
@GetMapping("/")
 @RateLimiter(name="simpleService", fallbackMethod = "getDefaultInfo")
public String getInfo()
{
    String msg = restTemplate.getForObject("http://SimpleService/",String.class);
    return "3Tier architecture... "+msg;
}
```

```yaml
resilience4j.ratelimiter:
  instances:
    stockService:
      limitForPeriod: 5
      limitRefreshPeriod: 500ns
      timeoutDuration: 5s
```

Bulkhead

Bulkhead ensures the failure in one part of the system doesn't cause the whole system down. It controls the number of concurrent calls a component can take. This way, the number of resources waiting for the response from that component is limited.

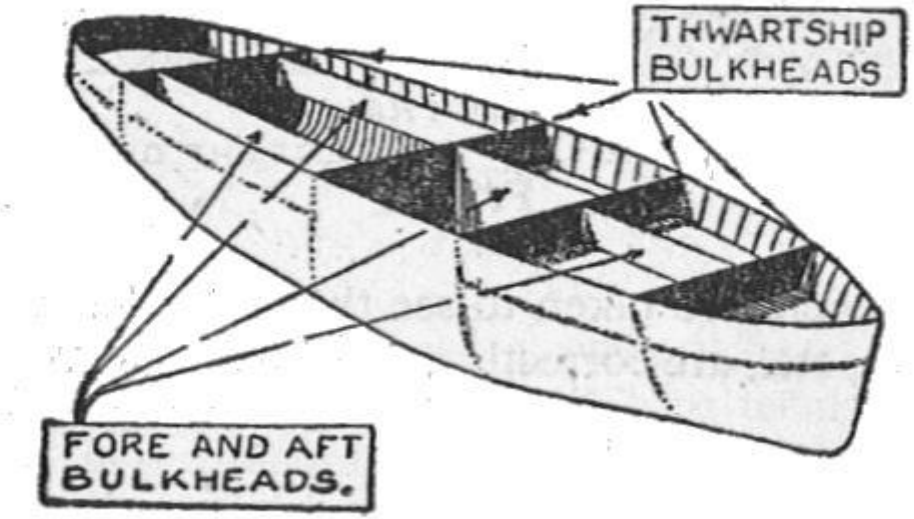
There are two types of bulkhead implementation:

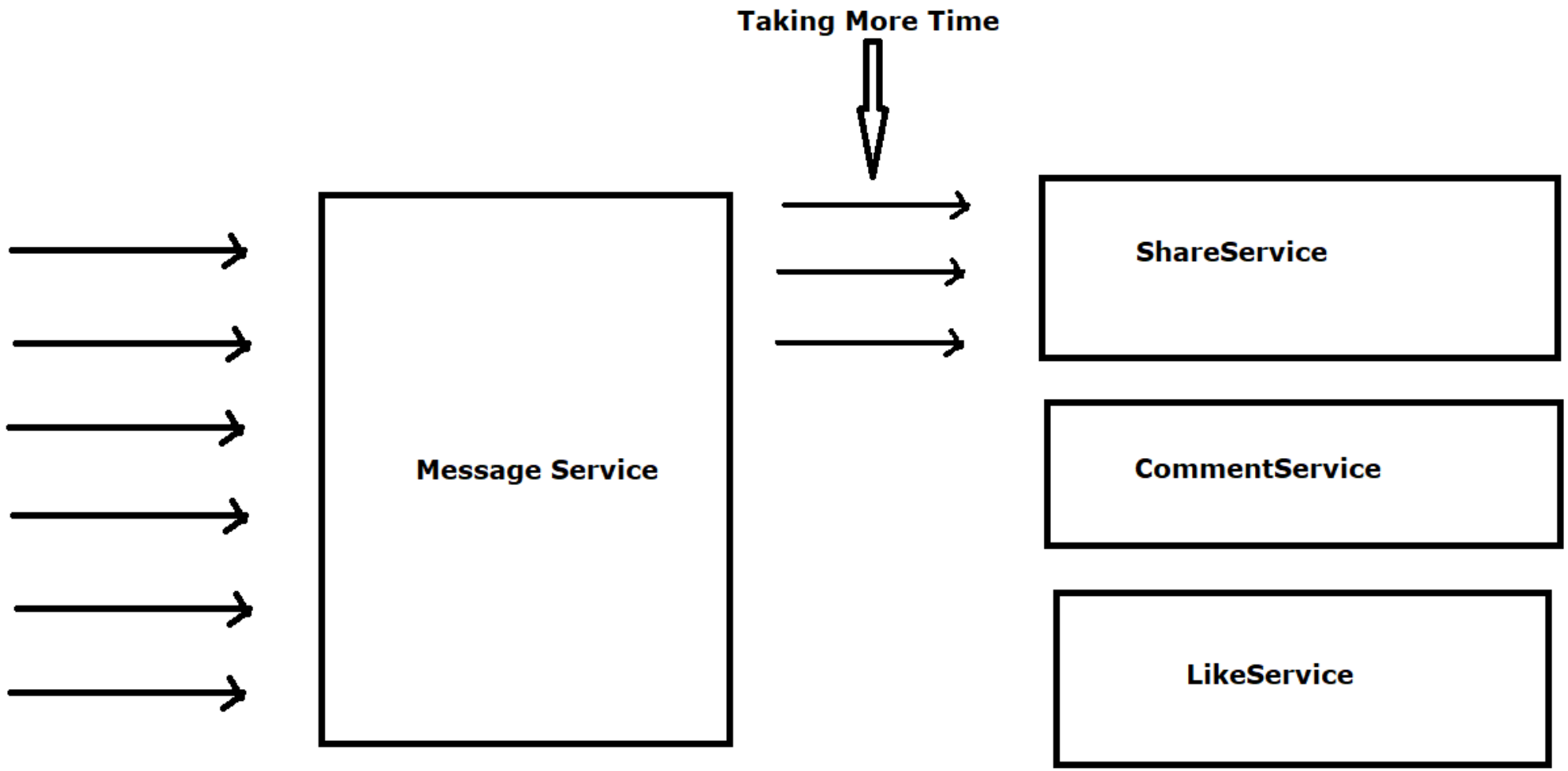
The semaphore isolation approach limits the number of concurrent requests to the service. It rejects requests immediately once the limit is hit. The thread pool isolation approach uses a thread pool to separate the service from the caller and contain it to a subset of system resources.

The thread pool approach also provides a waiting queue, rejecting requests only when both the pool and queue are full. Thread pool management adds some overhead, which slightly reduces performance compared to using a semaphore, but allows hanging threads to time out.

A ship is split into small multiple compartments using Bulkheads. Bulkheads are used to seal parts of the ship to prevent entire ship from sinking in case of flood.

Similarly failures should be expected when we design software. The application should be split into multiple components and resources should be isolated in such a way that failure of one component is not affecting the other.




Taking More Time
Message Service
ShareService
CommentService
LikeService

```java
@GetMapping("/")
@Bulkhead(name = "simpleService", fallbackMethod = "getDefaultInfo", type = Bulkhead.Type.SEMAPHORE)
public String getInfo()
{
    String msg = restTemplate.getForObject("http://SimpleService/",String.class);
    return "3Tier architecture... "+msg;
}
```

```yaml
server:
  port: 8081
  tomcat:
    threads:
      max: 15

resilience4j.bulkhead:
  instances:
    simpleService:
      maxConcurrentCalls: 10
      maxWaitDuration: 10ms
```

CircuitBreaker

Config property	Default Value	Description
failureRateThreshold	50	Configures the failure rate threshold in percentage. When the failure rate is equal or greater than the threshold the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallRateThreshold	100	Configures a threshold in percentage. The CircuitBreaker considers a call as slow when the call duration is greater than `slowCallDurationThreshold` When the percentage of slow calls is equal or greater the threshold, the CircuitBreaker transitions to open and starts short-circuiting calls.
slowCallDurationThreshold	60000 [ms]	Configures the duration threshold above which calls are considered as slow and increase the rate of slow calls.
permittedNumberOfCalls InHalfOpenState	10	Configures the number of permitted calls when the CircuitBreaker is half open.

slidingWindowType	COUNT_BASED	Configures the type of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed. Sliding window can either be count-based or time-based. If the sliding window is COUNT_BASED, the last `slidingWindowSize` calls are recorded and aggregated. If the sliding window is TIME_BASED, the calls of the last `slidingWindowSize` seconds recorded and aggregated.
slidingWindowSize	100	Configures the size of the sliding window which is used to record the outcome of calls when the CircuitBreaker is closed.

minimumNumberOfCalls	10	Configures the minimum number of calls which are required (per sliding window period) before the CircuitBreaker can calculate the error rate. For example, if minimumNumberOfCalls is 10, then at least 10 calls must be recorded, before the failure rate can be calculated. If only 9 calls have been recorded the CircuitBreaker will not transition to open even if all 9 calls have failed.
waitDurationInOpenState	60000 [ms]	The time that the CircuitBreaker should wait before transitioning from open to half-open.

Create and configure a Bulkhead

You can provide a custom global BulkheadConfig. In order to create a custom global BulkheadConfig, you can use the BulkheadConfig builder. You can use the builder to configure the following properties.

Config property	Default value	Description
maxConcurrentCalls	25	Max amount of parallel executions allowed by the bulkhead
maxWaitDuration	0	Max amount of time a thread should be blocked for when attempting to enter a saturated bulkhead.

RateLimiter

Config property	Default value	Description
timeoutDuration	5 [s]	The default wait time a thread waits for a permission
limitRefreshPeriod	500 [ns]	The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
limitForPeriod	50	The number of permissions available during one limit refresh period

Retry

Config property	Default value	Description
maxAttempts	3	The maximum number of retry attempts
waitDuration	500 [ms]	A fixed wait duration between retry attempts
intervalFunction	numOfAttempts -> waitDuration	A function to modify the waiting interval after a failure. By default the wait duration remains constant.
retryOnResultPredicate	result -> false	Configures a Predicate which evaluates if a result should be retried. The Predicate must return true, if the result should be retried, otherwise it must return false.
retryOnExceptionPredicate	throwable -> true	Configures a Predicate which evaluates if an exception should be retried. The Predicate must return true, if the exception should be retried, otherwise it must return false.
retryExceptions	empty	Configures a list of error classes that are recorded as a failure and thus are retried.
ignoreExceptions	empty	Configures a list of error classes that are ignored and thus are not retried.

```
RetryConfig config = RetryConfig.custom()
  .maxAttempts(2)
  .waitDuration(Duration.ofMillis(1000))
  .retryOnResult(response -> response.getStatus() == 500)
  .retryOnException(e -> e instanceof WebServiceException)
  .retryExceptions(IOException.class, TimeoutException.class)
  .ignoreExceptions(BusinessException.class,
                    OtherBusinessException.class)
  .build();
```

resilience4j has the ability to add multiple fault tolerance features into one call. It is more configurable and amount of code needs to be written is less with right amount of abstractions.

```
@CircuitBreaker(name = BACKEND, fallbackMethod = "fallback")
@RateLimiter(name = BACKEND)
@Bulkhead(name = BACKEND)
@Retry(name = BACKEND, fallbackMethod = "fallback")
@TimeLimiter(name = BACKEND)
Public String postCallProxyES(...) {
     ... . ..
}
```

The default Resilience4j Aspects order is the following:
Retry (CircuitBreaker (RateLimiter (TimeLimiter (Bulkhead (Function)))))

Bulkhead	Rate Limiter
Limit number of concurrent calls at a time.	Limit number total calls in given period of time
Ex: Allow 5 concurrent calls at a time	Ex: Allow 5 calls every 2 second.
In above example, first 5 calls will start processing in parallel while any further calls will keep waiting. As soon as one of those 5 in-process calls is finished, next waiting calls will be immediately eligible to be executed.	In above example, 2 second window starts & first 5 calls will start processing (may be parallel or not parallel). Those 5 calls might finish before 2 seconds, but next waiting calls will NOT be immediately eligible to be executed. After 2 seconds window is over, next 2 seconds window will start & then only next waiting calls will be eligible to be executed.