Kresil - Kotlin Resilience

Kotlin Multiplatform Library for Fault-Tolerance

Francisco José Barbosa Engenheiro

Supervisor: Pedro Félix

Final report written for Project and Seminary
BSc in Computer Science and Engineering

July 2024

Instituto Superior de Engenharia de Lisboa

Kresil - Kotlin Resilience

Kotlin Multiplatform Library for Fault-Tolerance

49428 Francisco José Barbosa Engenheiro

Supervisor: Pedro Félix, ISEL

Final report written for Project and Seminary
BSc in Computer Science and Engineering

July 2024

Abstract

This document addresses a need in the Kotlin Multiplatform ecosystem, by developing a fault-tolerance library. Kotlin Multiplatform is a technology that enables developers to share code across multiple platforms, promoting code reuse and maintainability.

The primary goal is to create a Kotlin Multiplatform library offering essential resilience mechanisms (e.g., Retry, Circuit Breaker, Rate Limiter). These mechanisms, alone or combined, prevent or mitigate the inevitable failures that occur in distributed systems, which impact the availability and reliability of services. Furthermore, the project aims to integrate these mechanisms into Ktor, a Kotlin Multiplatform framework for building asynchronous server and client services.

The project began with a review of existing solutions in various platforms and languages to identify common patterns and practices, as well as differences and limitations. With this knowledge, a model was designed to represent the common interface for all implemented resilience mechanisms by the library - the Mechanism Model. The project then explored each of the various resilience mechanisms, detailing their functionality, the chosen design and implementation details, configuration capabilities, default values associated with each policy, and their integration into the Ktor framework as plugins.

This project successfully developed and deployed a dedicated Kotlin Multiplatform library that offers resilience mechanisms, and includes a Ktor integration to provide an immediate installation and configuration of these mechanisms in asynchronous server and client architectures.

Keywords: Kotlin Multiplatform; library; fault-tolerance; resilience; distributed systems; Ktor framework

Resumo

Este documento aborda uma necessidade no ecossistema Kotlin Multiplataform, ao desenvolver uma biblioteca de tolerância a falhas. Kotlin Multiplatform é uma tecnologia que permite aos desenvolvidores partilhar código em várias plataformas, promovendo a reutilização e manutenção do mesmo.

O objetivo principal é criar uma biblioteca Kotlin Multiplataform que ofereça mecanismos de resiliência (e.g., Retry, Circuit Breaker, Rate Limiter). Estes mecanismos, sozinhos ou combinados, previnem ou mitigam as falhas inevitáveis que ocorrem em sistemas distribuídos, que afectam a disponibilidade e a fiabilidade dos serviços. Além disso, o projeto visa integrar estes mecanismos no Ktor, uma framework Kotlin Multiplatform para a construção de serviços assíncronos de servidor e cliente.

O projeto começou com uma revisão das soluções existentes em várias plataformas e linguagens para identificar padrões e práticas comuns, bem como diferenças e limitações. Com este conhecimento, foi concebido um modelo para representar a interface comum para todos os mecanismos de resiliência implementados pela biblioteca - o Mechanism Model. O projeto explorou vários mecanismos de resiliência, explicando a sua funcionalidade, detalhes de conceção e implementação escolhidos, capacidades de configuração, valores por defeito associados a cada política e a sua integração na framework Ktor como plugins.

Este projeto desenvolveu com sucesso e implantou com sucesso uma biblioteca Kotlin Multiplatform que oferece mecanismos de resiliência, e inclui uma integração Ktor para facilitar a instalação e configuração destes mecanismos em arquiteturas assíncronas de servidor e cliente.

Palavras-chave: Kotlin Multiplatform; biblioteca; tolerância a falhas; resiliência; sistemas distribuídos; framework Ktor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Our reliance on digital services has grown considerably, driving the need for these services to be highly reliable and always available. Whether it's financial transactions, healthcare systems, or social media platforms, users expect uninterrupted access and seamless experiences. This expectation places significant pressure on the underlying infrastructure to handle failures gracefully and maintain service continuity. Achieving this level of reliability requires mechanisms to manage and mitigate faults effectively.

Most of these services are built on distributed systems, which consist of independent networked computers that present themselves to users as a single coherent system [5]. Given the complexity of these systems, they are susceptible to failures caused by a variety of factors such as hardware malfunctions, software bugs, network issues, communication problems, or even human errors. As such, it is necessary to ensure that services within distributed systems are resilient, and more specifically, fault-tolerant.

A fault-tolerant service is a service that is able to maintain all or part of its functionality, or provide an alternative, when one or more of its associated components fail.

1.1.1 Resilience Mechanisms

Several resilience mechanisms have been developed to help build more robust and reliable systems. These mechanisms provide a set of tools and strategies to handle the inevitable occurrence of failures. Some of the most relevant mechanisms are described in Table 1.1.

These mechanisms can be used individually or combined to provide more complex resilience strategies. Additionally, they can be further categorized based on when they are activated:

- **Reactive Resilience**: Reacts to failures and mitigates their impact (e.g., the Retry mechanism is only triggered after a failure occurs);

- **Proactive Resilience**: Prevents failures from happening (e.g., the Rate Limiter mechanism is used to limit the rate of incoming requests, as a way to prevent the system from being overwhelmed and potentially fail - acts before a failure occurs).

Table 1.1: Resilience mechanisms examples. Retrieved from [4]

Name	Funcionality	Description
Retry	Repeats failed executions.	Many faults are transient and may self-correct after a short delay.
Circuit Breaker	Temporary blocks possible failures.	When a system is seriously struggling, failing fast is better than making clients wait.
Rate Limiter	Limits executions/period.	Limit the rate of incoming requests.
Time Limiter	Limits duration of execution.	Beyond a certain wait interval, a successful result is unlikely.
Bulkhead	Limits concurrent executions.	Resources are isolated into pools so that if one fails, the others will continue working.
Cache	Memorizes a successful result.	Some proportion of requests may be similar.
Fallback	Defines an alternative value to be returned (or action to be executed) on failure.	Things will still fail - plan what you will do when that happens.

1.1.2 Kotlin Multiplatform

Kotlin Multiplatform [6] is a technology that allows sharing code across multiple platforms, such as Android and iOS for mobile applications, and/or JVM, JavaScript and Native for multiplatform overall. This type of technology is particularly useful for building services that run on multiple platforms, without having to write potentially equivalent platform-specific code, this way maintaining a reusable codebase.

In 2019, Google announced Kotlin as the official language for Android development [7], and more recently, official support for Android development with Kotlin Multiplatform [8, 9]. This recognition strengthened Kotlin Multiplatform as a viable technology for building multiplatform applications.

As a recent technology that is still evolving, and only reached stable status late 2023 [10], it lacks some of the necessary tools (e.g., libraries, frameworks, etc.) that provide functionalities that are common in other ecosystems.

1.2 Problem

An analysis of the Kotlin Multiplatform ecosystem has revealed a significant gap: there is no specialized library that offers a wide variety of resilience mechanisms. To address this problem, it is important to develop a library that enables developers to integrate resilience mechanisms seamlessly into their projects, regardless of the platform they are targeting.

1.3 Related Work

1.3.1 Ktor

Ktor [11] is a Kotlin Multiplatform framework designed for building asynchronous servers and clients, such as web applications and microservices.

The framework already provides some of the aforementioned resilience mechanisms as plugins, that can be installed in the underlying pipepile to intercept specific phases of the request/response cycle and apply the desired behaviour (e.g., retrying a request on the client side [12], rate limiting the incoming requests on the server side [13]).

1.3.2 Other Solutions

Platform-Specific Libraries

There are several libraries that provide resilience mechanisms for different programming languages and platforms. Table 1.2 shows some examples of these libraries.

Table 1.2: Examples of libraries that provide resilience mechanisms.

Library	**Language**	**Plataform**
Netflix's Hystrix [14]	Java	JVM
Resilience4j [4]	Java/Kotlin	JVM
Polly [15]	C#	.NET

The Hystrix library served as an inspiration for Resilience4J, which is based on functional programming concepts. The primary distinction between the two is that, whereas Resilience4J relies on function composition to let the caller stack the specific decorators it needs by utilizing Java 8's features (e.g., functional interfaces, lambda expressions) [16], Hystrix embraces an object-oriented design where calls to external systems have to be wrapped in a *HystrixCommand* offering multiple functionalities.

The Resilience4j library served as the main source of inspiration for the project's development since it was considered a more modern way of implementing these mechanisms, follows a functional programming style, and is more in line with the characteristics of the Kotlin language. The Polly library was used as a secondary source to explore alternative approaches and design patterns that could be used in the project.

Arrow Library

The Arrow library, which presents itself as the functional companion to Kotlin's standard library, is a Kotlin Multiplatform library that focuses on functional programming and includes, among other modules, a resilience module. This module implements three of the most used design patterns around resilience [17]: retry and repeat computations using a *Schedule*, protect other services from being overloaded using a *CircuitBreaker*, and implement transactional behaviour in distributed systems in the form of a *Saga*.

However, the library does not include all considered essential resilience mechanisms, and the ones it does include are not as feature-rich as those offered by the aforementioned platform-specific libraries.

1.4 Project Goal

The goal of this project is to develop a Kotlin Multiplatform library that provides some of the aforementioned resilience mechanisms.

Additionally, the project aims to integrate these mechanisms in the Ktor framework, using its extensibility features to create plugins. This integration was considered because:

- Ktor is the only known Kotlin Multiplatform framework for server and client HTTP services development;
- Immediately provide a way to use the library in a specific context - HTTP client and server services.
- Validate the implementation and extensibility of the library.

The library should be easy to use and configure, taking advantage of Kotlin's language features to provide a concise and expressive API.

According to the time available for the project, the scope was limited to the implementation of the following resilience mechanisms and their respective plugins for the Ktor framework:

- Retry Mechanism and Plugin for Ktor Client;
- Circuit Breaker Mechanism and Plugin for Ktor Client;
- Rate Limiter Mechanism and Plugin for Ktor Server.

The project will also provide a battery of tests to ensure the correctness and reliability of the implemented mechanisms and plugins, as well as documentation to guide developers on how to use them.

1.5 Document Structure

The remaining of this document is structured as follows:

- **Chapter 2 - Kotlin Multiplatform**: Provides an overview of the Kotlin Multiplatform technology, its architecture, and how it can be used to share code across multiple platforms. Additionally, it describes the template used, the adopted project structure, and how to run tests in a multiplatform context;
- **Chapter 3 - Common Design and Implementation Strategy**: Describes the design and implementation aspects that are common to all the resilience mechanisms. Additionally, it describes the Ktor framework and how it was used in the project;
- **Chapter 4, 5, 6 - Retry, Circuit Breaker, Rate Limiter**: Each of these chapters describes the functionality, configuration, and implementation of the respective resilience mechanism in the library and as a plugin for the Ktor framework;
- **Chapter 7 - Conclusions**: This chapter provides a summary in conclusion format of the project, work methodology and future work.

Chapter 2

Kotlin Multiplatform

This chapter provides an overview of the Kotlin Multiplatform (KMP) technology, including its architecture, template structure, and other aspects relevant to the development of a multiplatform library. This technology allows developers to share code across multiple platforms, such as Android and iOS for mobile applications, and/or JVM, JavaScript and Native for multiplatform overall.

2.1 Architecture

A KMP project is divided into three main categories of code:

- **Common**: Code shared between all platforms (i.e., *CommonMain, CommonTest*);
- **Intermediate**: Code that can be shared on a subset of platforms (i.e., *AppleMain, AppleTest*);
- **Specific**: Code specific to a target platform (i.e., `<Platform>Main`, `<Platform>Test`).

Figure 2.1 illustrates an example of a KMP project architecture. Note that both the *Intermediate* and *Specific* categories are optional.

2.1.1 Template

It is possible to create a KMP project from scratch, but it is recommended to use a template to facilitate the project's setup and configuration. The official Kotlin Multiplatform template [18] provides a project structure that includes the necessary configurations for building, testing, and deploying a multiplatform library for most platforms.

2.1.2 Gradle Tasks

Gradle [19] is an open-source build automation tool for software construction which supports multiple programming languages, including Kotlin. Offers support for all phases of a build process including compilation, verification, dependency resolving, test execution, source code generation, packaging and publishing.

In a Gradle build using Kotlin DSL (domain-specific language), a project's configuration is primarily defined in two key files:

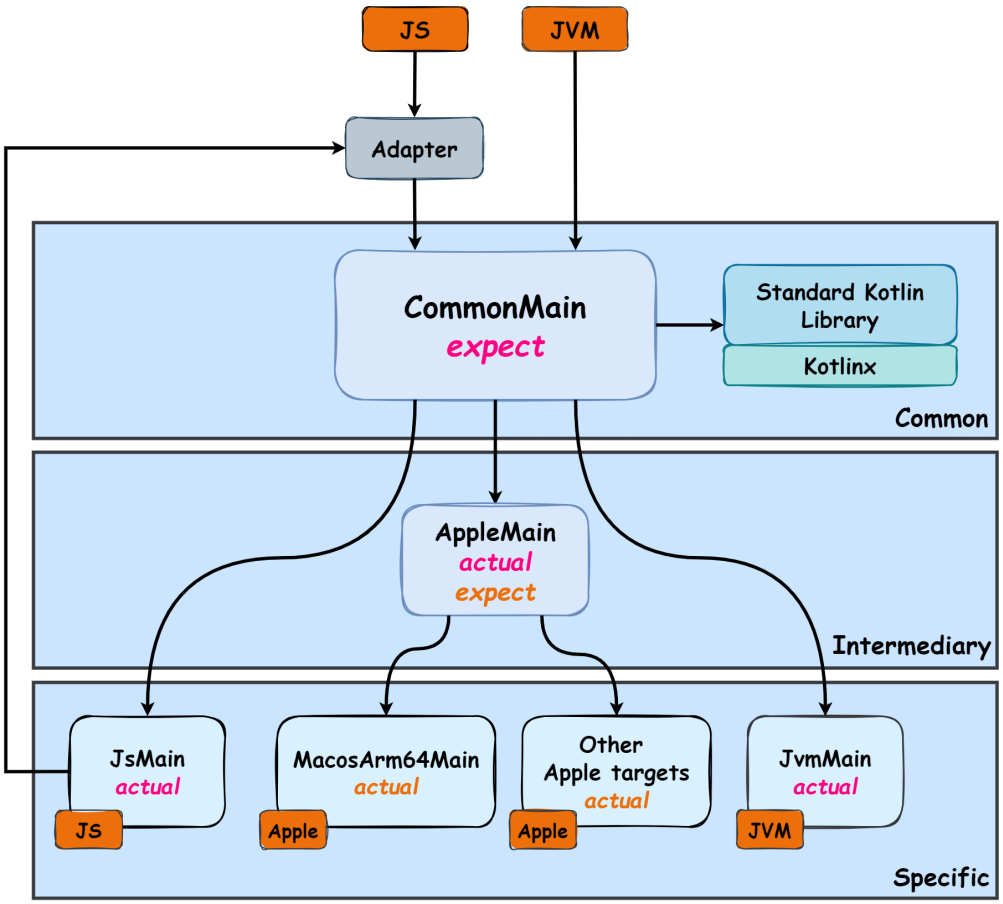

Figure 2.1: Example of a KMP project architecture.

- **build.gradle.kts** - Defines a project's build configuration. In KMP projects, this file is used to define a project's targets, dependencies in the respective source sets, and additional configurations if needed;
- **settings.gradle.kts** - Defines a project's structure and modules it contains.

In Gradle, tasks are the smallest unit of work that can be executed and are used to perform specific actions. The template uses the KMP plugin, which defines several Gradle tasks to assist in building, testing, and managing a project across multiple platforms configured as targets in the respective *build.gradle* file. Some of the most relevant tasks include:

- **build**: Compiles and assembles the project;
- **allTests**: Runs the test cases for all platforms. To run platform-specific tests, use the `<Platform>Test` task (e.g., *jvmTest*);
- **check**: Performs various checks on the project, including running tests and performing additional operations (e.g., linting, code analysis);
- **clean**: Deletes the build directory, allowing for a clean build (i.e., not using cached artifacts).

A Gradle project can be organized into multiple subprojects, each with its own build file, settings and tasks.

2.1.3 GitHub Actions

The template provides workflows for GitHub Actions [20], which are used to automate tasks such as building, testing, and deploying a project. The configurations for GitHub Actions are located in the *.github/workflows* folder, and include the following workflows:

- **gradle.yml** - Builds and tests the project using Gradle against some of the available platforms. Runs on push and pull request git events to the default branch;
- **deploy.yml** - Deploys the library artifacts to a repository in Maven Central [21], following a pre-defined authentication configuration.

2.1.4 Folder Structure

The template is organized into several folders, each serving a specific purpose. Below is a brief description of each folder, with an emphasis on the *src* folders:

- **.github/**: Contains configurations for GitHub Actions, which are used to automate tasks;
- **gradle/**: Contains configuration files and scripts related to the Gradle build system. This folder typically includes:
  - **wrapper/**: Contains the wrapper files and configurations, which standardizes a project on a given Gradle version for more reliable and robust builds [22];
  - **libs.versions.toml**: Defines the versions of the libraries and plugins used in a project, as dependencies, in a centralized manner (regularly known as version catalog [23]).
- **convention-plugins/**: Encapsulates and reuses common build logic across multiple Gradle projects or modules (e.g., for publishing, testing, etc.);
- **library/**: Contains the source code for the library and the build configuration file.
  - **src/**: Contains the source code for the library, divided into multiple submodules based on the target platforms:
    - **commonMain/**: Contains the common code shared across all platforms;
    - **jvmMain/**: Contains the source code specific to the JVM platform;
    - **jsMain/**: Contains the source code specific to the JavaScript platform;
    - **iosMain/**: Contains the source code specific to the iOS platform;
    - **androidMain/**: Contains the source code specific to the Android platform;
    - And all of these module counterparts for the test code (e.g., *commonTest, jvmTest, jsTest, etc*).
  - **build.gradle.kts**: Defines the targets, dependencies, and additional configurations for the library;
- **build.gradle.kts**: The main build configuration file for the project, where the subprojects common configurations are defined;

- **settings.gradle.kts**: Configures the Gradle build settings for the project, including the root project name and module inclusion.

For developing the KMP library, a different template structure was adopted, specifically replacing the *library* module from the original template with the following structure:

- **<kmp_library_name>**: name of a KMP library in root directory;
  - **apps**: defines the modules that will consume the KMP library (e.g.,js-app, `android-app`);
  - **lib or shared**: defines the library's code to be shared by the consuming modules;
    - **src**: defines the target submodules of the library including their test counterparts (i.e., `<Platform>Main`, `<Platform>Test`);
    - **build.gradle.kts**: defines the library's dependencies, targets, and additional configurations.

The same approach was used for organizing the Ktor Client and Server Plugin modules, which are also Kotlin Multiplatform libraries.

2.2 Platform-Dependent Code

As code sharing across platforms is the primary objective of KMP, it should be written as platform-independently as possible (i.e., aggregating as much code in the hierarchically higher categories). However, it is sometimes necessary to create specific code for a given platform, regularly referred to as *target*, in the following situations:

- Access to API's specific to the *target* is required (e.g., *Java's File API*);
- The libraries available in the common category (i.e., *Standard Kotlin Library*, libraries and extensions from *Kotlinx*) do not cover the desired functionalities, and third-party libraries either do not support them or are avoided to reduce dependencies;
- When specific types and functions defined in Kotlin need to be accessed in other languages (e.g., *Javascript*). As such, an adapter is required to communicate with the common category code, in Kotlin, from the native code of the *target*, which can be defined in the *Intermediate* or *Specific* categories.

In such cases, platform-specific implementations can be defined in the *Intermediate* or *Specific* categories. One approach is through the *expect/actual* [24] mechanism. This mechanism allows for defining abstracted code and its platform-specific implementation. However, there are other methods available for achieving platform-specific functionality, such as the described *adapter*.

2.3 Supported Targets

The project supports the following targets:

- **JVM**: Allows running the code on the *Java Virtual Machine*;
- **JavaScript**: Allows running the code in a browser or *Node.js* environment;
- **Android**: Allows running the code on Android devices;
- **Native**: Allows running the code on platforms that support *Kotlin/Native*, excluding macOS and iOS, because the lack of access to the necessary hardware for testing.

2.4 Kotlin Library Practices

The library strives to follow the best practices for developing a Kotlin library, including:

- Adherence to Kotlin Coding Conventions for Libraries to ensure consistency and readability;
- Compliance with Kotlin API Guidelines to create a user-friendly and maintainable API.

2.5 Multiplatform Considerations

When developing a multiplatform library, it was necessary to consider the following aspects, which may be harder to manage in a multiplatform context:

- **Concurrency**: Techniques and tools used to manage concurrency, ensuring thread safety and performance across platforms. It's important to note that KMP's concurrency model primarily revolves around coroutines, which may use threads depending on the platform's dispatchers. Understanding platform-specific concurrency models is crucial as the guarantees and limitations can vary significantly (e.g., Kotlin/Native does not rely on Java's Memory Model [25].
- **Time Representation and Measurement**: Strategies for representing and measuring time consistently across platforms. For instance, not all platforms guarantee the same precision or accuracy in time measurements, nor provide a monotonic time source;
- **Synchronous vs Asynchronous**: Depending on the platform, the code might need to accommodate either synchronous operations or asynchronous operations. For example, JavaScript being single-threaded limits operations to asynchronous execution for non-blocking behavior.
- **Delay without active Blocking**: Strategies for implementing delays without actively blocking the thread (e.g., can't use *Thread.sleep*);)
- **Mocking**: Strategies for mocking dependencies in tests to isolate and verify the behavior of individual components.
- **Logging**: Consistent implementation of logging for debugging and monitoring purposes;

Chapter 3

Common Design and Implementation Strategy

This chapter outlines the shared design principles of the implemented mechanisms and the development approach used to integrate them with the Ktor framework. Additionally, it describes some of the problems faced during the development process and the solutions adopted to address them.

A comprehensive review and analysis of existing mechanism implementations 1.3 were conducted to identify common design elements and relevant features that could be incorporated into the design of the mechanisms implemented in this project.

3.1 Mechanism Model

The common design elements identified in the existing solutions were used to define a model, as shown in Figure 3.1. This model, named the Mechanism Model, is composed of multiple components, with distinct responsibilities and relations between them, that work together to provide a base foundation for the implementation of the mechanisms.

- **Configuration**: Represents a set of policies that, in conjuction, define the mechanism's intended behaviour (e.g., maximum number of retries, maximum wait duration, etc.);
- **Asynchronous Context**: Represents the mechanism's execution context, responsible for state management and event emission in an asynchronous environment;
- **State**: Represents the internal state of the mechanism;
- **Implementation**: Is parameterized by the Configuration component and executes the mechanism's behaviour. Represents the core component of the mechanism;
- **Registry**: Acts as a centralized container for storing and managing available mechanism implementations and configurations. The registry allows access to mechanism implementations throughout the application and enables the reuse of configurations to create new mechanisms;
- **Events**: Both the Asynchronous Context and Registry components are responsible for emitting events. The Asynchronous Context component emits events related to the mechanism's execution context, such as internal state transition changes. The Registry component emits events related

to *CRUD* operations performed in the registry. These events can be used for various purposes, such as logging and monitoring;

- **Metrics**: The mechanism's implementation component is responsible for recording metrics related to the mechanism's execution (e.g., number of retries, number of recorded failures, etc.). These metrics can be used for monitoring and analysis purposes;
- **Decorator**: The decorator is an extension of the Implementation component. It is inspired on the Resilience4J library [4] decorators, and provides a convenient way to wrap code blocks with the mechanism's behaviour;
- **Ktor Plugin**: Responsible for the integration of the mechanism implementation with the Ktor pipeline. The Configuration component is also used to create a specific plugin configuration, which extends the mechanism's behavior and adds additional features in an HTTP context.

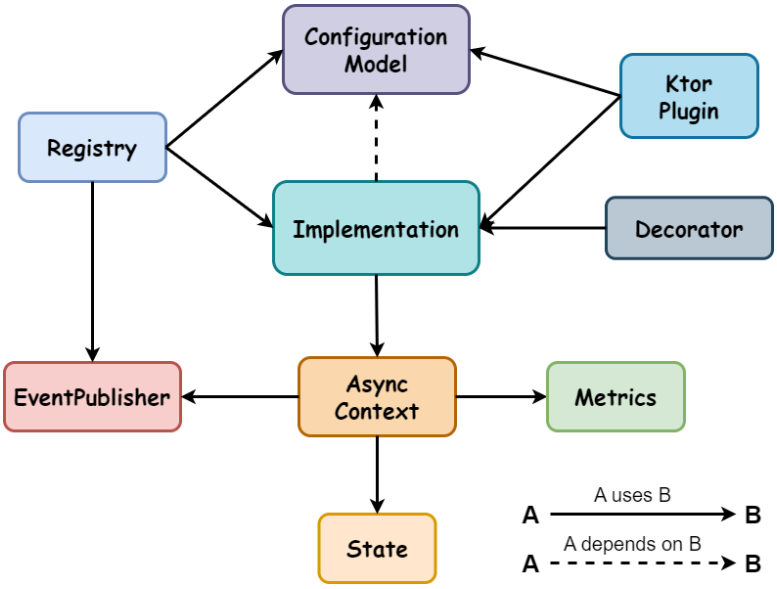

Figure 3.1: Mechanism Model

3.1.1 Configuration Design

From the beginning of the project, the Configuration component has been designed to use the builder creational pattern [26, 27], this way, separating the configuration definition (mutable) from the configuration usage (immutable). However, this initial implementation had a limitation: it was not possible to override a configuration (i.e., create a new configuration based on an existing one, and only modify specific properties while keeping the rest of the properties unchanged). To address this limitation, the Configuration component, particularly its builder, was redesigned to always be constructed with a base configuration. This redesign enables incremental configuration, where given a base configuration to the builder, on top of which new properties can be applied to generate a new immutable configuration.

The configuration process is as follows:

1. Begin with a default or initial (base) configuration;

2. Pass this configuration to a configuration builder;

3. On top of the builder, apply the desired configuration properties;

4. Generate a new configuration from the builder;

5. If further modifications are needed, pass the new configuration back to the builder, which will use it as the base configuration;

6. Repeat the process until the desired configuration is achieved.

Additionally, the builder also has the additional responsibility of validating the configuration properties before a configuration is generated.

3.1.2 Mechanism Execution Context

Independent of the execution environment, synchronous or asynchronous, the mechanism's execution context can only be one of the following:

- **Per Mechanism Instance**: A new execution context is created when the mechanism itself is instantiated (e.g., when the Circuit Breaker mechanism is used to decorate multiple operations, a single execution context is shared among all decorated operations);

- **Per Decoration**: When a decorator is applied to an operation, it creates a new execution context specific to that decoration, before invoking the underlying operation (e.g., the Circuit Breaker mechanism can be used to decorate a single operation, as decorating multiple operations would be a per mechanism instance context);

- **Per Method Invocation**: A new execution context is created each time the decorated method is invoked. This is the most granular form of execution context, providing isolation for each method call. If the underlying operation is *thread-safe*, then this form of execution context is also *thread-safe*, as only the calling thread executes the context (e.g., the Retry mechanism creates a new execution context for each underlying operation invocation, because each retry attempt cycle is independent of other cycles).

Note that the term *thread-safe* refers to a program that behaves correctly when multiple threads are executing it concurrently, ensuring that shared data is accessed and modified in a manner that preserves integrity and consistency. Additionally, the term *thread* used here is often used broadly to refer to the execution context, which may not always correspond to a traditional operating system thread (e.g., it could be a coroutine or a virtual thread). In the context of Kotlin Multiplatform, concurrency management relies on coroutines and the underlying threading model of each supported platform. For instance, on Kotlin/JVM, thread safety often aligns with the Java Memory Model [25], while on Kotlin/Native, it depends on the native threading mechanisms of the target platform.

3.1.3 Asynchronous Context

Since the mechanisms are designed to be cross-platform, the execution context must be flexible and support asynchronous operations, particularly for JavaScript, one of the supported targets (see Section 2.3), which is single-threaded and requires asynchronous (non-blocking) operations.

In Kotlin, asynchronous operations are typically handled using Kotlin Coroutines [28], which provide a way to write asynchronous code in a sequential manner, leveraging the concept of suspending functions implemented using contination passing style (CPS) [29].

Thread

In computer science, a thread [30] of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

In addition to being managed independently by an operating system's scheduler, threads are fundamental units of execution within a process. They enable concurrent execution of tasks, allowing a program to perform multiple operations simultaneously.

At a given time, only one thread can be active on a CPU core.

Kotlin Coroutines

A coroutine is an instance of a potentially suspendable computation, which can be suspended and resumed at a later time (uses a state machine to manage its execution). It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code [28]. However, a coroutine is not bound to any particular thread, as it may suspend its execution in one thread and resume in another one.

A suspended coroutine does not block the underlying thread, allowing it to execute other coroutines or tasks. In a thread, only one coroutine runs at a time.

A coroutine does not monopolize a thread during its execution. Instead, a single thread can run multiple coroutines by using time multiplexing, similar to how a single CPU can run multiple threads.

Another important feature of coroutines in Kotlin is that they enforce structured concurrency [28]. Structured concurrency is a programming paradigm that enforces a hierarchical structure on concurrent code, ensuring that all concurrent tasks are properly managed and cleaned up, and any errors are propagated correctly and not lost.

3.1.4 Event Publishing

As the execution context is asynchronous, the events must be also published asynchronously. To achieve this, the Kotlin Coroutines' asynchronous primitive Flow [31] was used to emit such events to potential listeners.

Flow

A Flow in Kotlin is an asynchronous data stream that can emit multiple values sequentially. Typically, it is cold, meaning it starts emitting values only when collected, ensuring that each collector receives

the full sequence of values from the start. In contrast, hot streams emit values regardless of active collectors, allowing multiple consumers to observe ongoing data emissions [32, 31].

Flows are distinct from Sequences [33] in Kotlin. While both can handle multiple values over time, Flows are designed for asynchronous operations and integrate with Kotlin's coroutine model, allowing value consumption to be suspended. In contrast, Sequences are synchronous and computed on demand (lazy), which blocks the calling thread during value computation.

Implementation

In a mechanism, events are emitted using a single hot Flow with no buffer, which means that, if there are no collectors (listeners) for the events, they are essentially lost (not recorded). As such, a listener must be registered before the event is emitted, in order to receive it.

Each mechanism implementation has two ways to register listeners for events:

- Receive all events emitted by the mechanism;
- Receive only a specific type of event emitted by the mechanism. To achieve this, a filter is applied to the Flow.

The implementation also provides a way to cancel all registered listeners by leveraging the coroutine's structured concurrency [28], which is useful for cleaning up resources when the mechanism is no longer needed. However, the cancellation of the registered listeners up to a given time does not affect later registrations.

3.2 Decoration

In functional programming, a high-order function [34] is a function that takes one or more functions as arguments and/or returns a function as its result.

A decorator is a structural design pattern that attaches additional responsibilities to an object dynamically using aggregation/composition to provide a flexible alternative to subclassing for extending functionality [27].

In the context of these mechanisms, a decorator is a high-order function that wraps an operation (e.g., method, function) with the behaviour of the mechanism. The decorator needs to abide by the decoration principle, which states that a decorator should not modify the operation's signature (i.e., name, parameters and return value), only alter its behaviour. For callers of the operation, the decorator should be transparent (as if the operation was not decorated).

3.2.1 Operation Types

An operation represents a unit of work that can be decorated with the mechanism's behaviour (e.g., a function that makes an HTTP request).

In early development stages, the mechanisms were designed to support only one type of operation: a function that takes no arguments and returns a value. But this design was too restrictive, as it did not allow for operations that receive additional arguments. To address this limitation, the operation types were redefined to support different scenarios:

- **Supplier**: Accepts no arguments and produces a result, which is based on Java's Supplier functional interface [35];
- **Function**: Accepts one argument and produces a result, which is based on Java's Function functional interface [36];
- **BiFunction**: Accepts two arguments and produces a result, which is based on Java's BiFunction functional interface [37].

As the operations were intended for potential asynchronous contexts, the operation types were redefined to be suspendable, representing potentially suspendable operations.

Operation Context

In later stages of development, it was considered to give the aforementioned operation types context-awareness, meaning that the operations would have access to the mechanism's execution context in an immutable state (i.e., for read-only purposes).

As a result, and to maintain backwards compatibility with the existing operation types (which allowed for method references [38] to be decorated), a new family of operation types was created with a `Ctx` prefix (e.g., `CtxSupplier`). These new operation types receive the mechanism's execution context as an additional argument, allowing the operations to be able to define logic based on the mechanism's context.

3.3 Exception Handling

In order to not enforce a specific exception handling strategy on the caller (e.g., always rethrowing exceptions), the mechanisms were designed to allow the caller to specify how caught exceptions should be handled.

For each mechanism, the caller can provide an exception handler that is invoked when an exception occurs during the mechanism's execution. By default, the exception handler rethrows the exception, but the caller can provide a custom exception handler to handle the exception in a different way (e.g., logging the exception, mapping it to a different exception type, etc.).

3.4 Time Measurement

Time is a critical aspect of most mechanisms, as they often rely on time-based policies (e.g., wait duration, time interval, etc.) to determine their behavior.

To ensure the accuracy and reliability of these time-based policies, the mechanisms use a comparable time mark with a monotonic time source to represent a snapshot of a point in time. A monotonic time source continuously increases and is not affected by system clock changes (e.g., time adjustments, time zone changes, etc.) which can occur with real-time clocks.

With a given time mark, the mechanisms can accurately measure time by retrieving the elapsed duration between two time marks, and apply time-based policies accordingly. This strategy avoids the need to launch a thread or coroutine just to measure elapsed time, which could introduce unnecessary overhead and complexity.

3.5 Ktor Integration

Integration with Ktor presented an opportunity to apply the mechanisms in a real-world scenario and validate their design and implementation. But before integrating the mechanisms with Ktor, it is necessary to understand its architecture and how to extend its functionality using plugins.

3.5.1 Plugin

In Ktor, a plugin is a reusable component that can be installed in an application (for Ktor server) or an HTTP client (for Ktor client) to extend its functionality. They represent a way to encapsulate common tasks (e.g., logging, authentication, serialization, etc.) and make it reusable across different applications. Plugins are installed in the application's pipeline, where they can intercept and modify the request and response processing flow, as the Figure 3.2 illustrates for the server side.

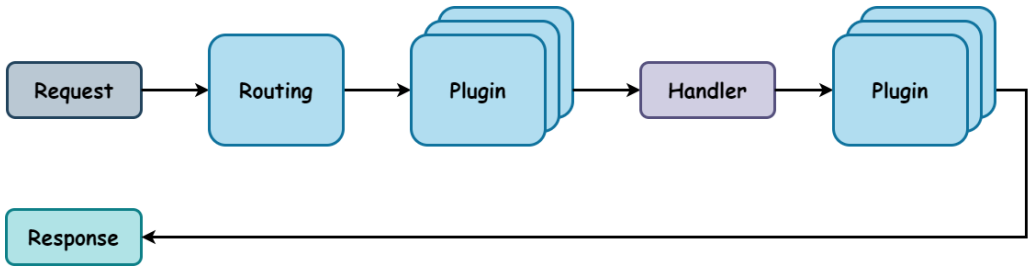

Figure 3.2: Ktor Server Architecture

On the server side, as a request comes in, it goes through a series of steps [39]:

1. It is routed to the correct handler via the routing mechanism (which is also a plugin);
2. Before being handed off to the handler, it goes through one or more Plugins;
3. The handler (application logic) handles the request;
4. Before the response is sent to the client, it goes through one or more Plugins.

3.5.2 Pipeline

A pipeline, represented in Figure 3.3, is a collection of zero or more interceptors, grouped in one or more ordered phases. Each interceptor can perform custom logic before and after processing a request.

A plugin is also an interceptor, but an interceptor is not a plugin. An interceptor is a function that can be added to intercept a specific pipeline phase and perform custom logic. A plugin however, is a collection of zero or more interceptors mixed with external configuration and other logic (e.g., add more pipeline phases) in a single reusable component.

Both client and server sides have pipelines, but they differ in number, phases, and purpose. Tables 3.1 and 3.2 show the phases of the server and client pipelines, respectively.

3.5.3 Custom Plugins

Ktor provides a custom plugin API that allows developers to create their own plugins in both client and server sides.

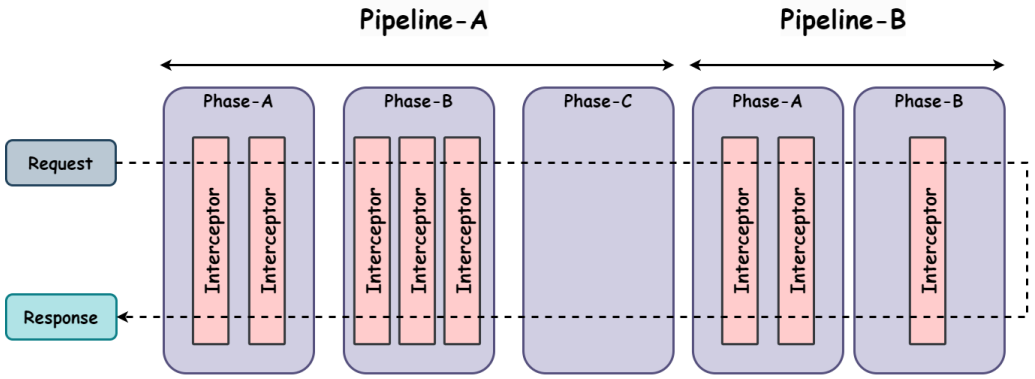

Figure 3.3: Ktor Pipeline Example

Table 3.1: Ktor Server Pipelines

Pipeline	**Description**	**Ordered Phases**
ApplicationSend	Responsible for sending responses	Before, Transform, Render, Content-Encoding, Transfer-Encoding, After, Engine
ApplicationReceive	Responsible for receiving requests	Before, Transform, After
ApplicationCall	Responsible for executing application calls	Setup, Monitoring, Plugins, Call, Fallback

Table 3.2: Ktor Client Pipelines

Pipeline	**Description**	**Ordered Phases**
HttpRequest	Processes all requests sent by the client	Before, State, Transform, Render, Send
HttpSend	Manages the actual transmission of the request and handling of the response	Before, State, Monitoring, Engine, Receive
HttpReceive	Used for receiving a response without processing	Before, State, After
HttpResponse	Used for processing responses	Receive, Parse, Transform, State, After

Since Ktor `2.0.0`, the custom plugin API has been simplified [40, 41] and no longer requires an understanding of internal Ktor concepts, such as pipelines, phases, etc. Instead, developers have access to different stages of handling requests and responses using general handlers (e.g., `onCallReceive`, `onCallRespond`), which intercept the related phases of the pipeline.

3.6 Development Roadmap

Originally, the project was planned to be developed by implementing all mechanisms for all targets simultaneously, testing them, and then integrating with Ktor.

However, due to the complexity of the mechanisms and the need to ensure correct implementation and testing in various contexts, the development strategy was revised.

For each mechanism, the following steps were taken:

1. Implement the Mechanism Model, including all its components and any additional components required by the mechanism;
2. Write tests for the implemented mechanism;
3. Ensure that the mechanism works correctly for all targets;
4. Implement the Ktor Plugin that uses the mechanism;
5. Write tests for the Ktor Plugin. Due to time constraints, unit tests and integration tests were not conducted; however, functional tests [42] were performed using a sample Ktor application;

Chapter 4

Retry

This chapter provides an in-depth look at the Retry reactive resilience mechanism, explaining the problems it aims to solve and how it addresses them through its functionality. It also covers available configurations and implementation details, providing descriptions and examples for both the library version and its integration as a plugin within the Ktor framework.

4.1 Introduction

The Retry mechanism is a *"resilience pattern that allows an application to handle transient failures when it tries to connect to a service or network resource. By transparently retrying a failed operation, the application can improve its stability and availability"* [43].

The Retry mechanism, as illustrated in Figure 4.1, is particularly useful when the application is interacting with services that are prone to temporary (transient) failures, such as network issues, temporary unavailability of services, or timeouts when services are overloaded. These issues are often brief and resolve themselves within a short period, meaning that retrying the operation after a short delay can often succeed (e.g., a call to a service that is temporarily overloaded might succeed if retried after a few seconds) [43].

Without a Retry mechanism, an application might treat such transient failures as critical, leading to unnecessary disruptions in service, increased latency, and a poor user experience.

When an application detects a failure, it can handle it using the following strategies [43]:

- **Cancel**: If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception (e.g., an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted);
- **Retry**: If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances (e.g., a network issue). In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated;
- **Retry After Delay**: If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time (wait duration) before retrying the request. The amount of time to wait before retrying depends on:

- the type of failure and the probability that it'll be corrected during this time;
- the delay strategy used.

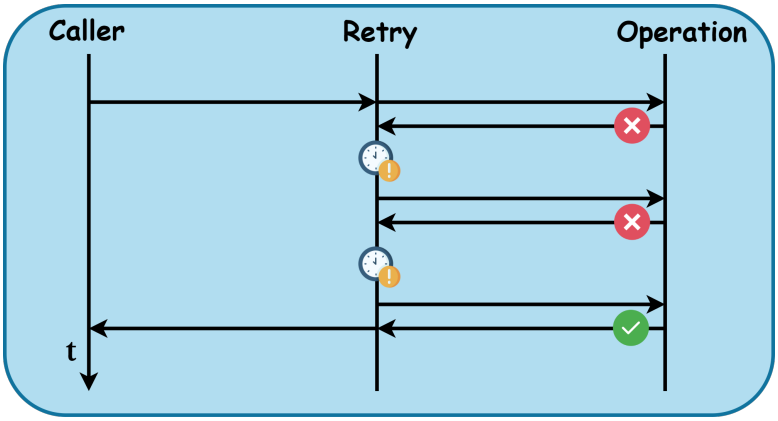

Figure 4.1: Retry Execution Example

4.1.1 Delay Strategies

The retry delay strategy defines the amount of time the application should wait before retrying the operation, and it can be one of the following:

- **No Delay**: This strategy does not introduce any delay between retries;
- **Constant Delay**: Introduces a fixed, constant delay between each retry attempt;
- **Linear Delay**: Increases the delay duration linearly with each retry attempt. The delay is calculated by multiplying the initial delay by the retry attempt number (e.g., `initialDelay=1s, attempts=4, result=[1, 2, 3, 4]`);
- **Exponential Delay**: This strategy exponentially increases the delay duration with each retry attempt, by using the *exponential backoff algorithm*. Essentially, the delay is calculated by multiplying the initial delay by a specified multiplier raised to the power of the retry attempt number (e.g., `initialDelay=1s, multiplier=2, attempts=4, result=[1, 2, 4, 8]`);
- **Custom Delay**: This strategy uses a custom function to calculate the delay duration based on the retry attempt number and possible context (e.g., last known error, if any), allowing for more complex delay strategies (e.g., a delay that increases linearly for the first 3 attempts, then exponentially for the next attempts).

In both linear and exponential delay strategies, a maximum delay can be set to prevent potentially excessive delays caused by the growth of the delay duration with each retry attempt.

Additionally, introducing a *jitter* [44] can help mitigate the *thundering herd problem* [45] by randomizing the calculated delay duration. This randomization prevents multiple clients from synchronizing their retries of the same operation, thereby spreading out the retries over a period of time.

4.2 Implementation Aspects

4.2.1 Custom Delay Provider

A custom delay provider can be used to implement a custom delay strategy that differs from the standard delay strategies.

A *delay strategy* determines the next wait time between retry attempts, while a *delay provider* executes the actual waiting period by pausing or blocking the process (dependends on the implementation) for the specified duration.

For example, for testing purposes, the custom delay provider might be implemented to not delay using real time, but instead, immediately continue the execution, and only keeping track of the time that would have been spent waiting (virtual time). This can be useful for speeding up tests that involve Retry mechanisms, as it allows for faster execution without actually waiting for the delay duration.

A custom delay provider also allows maintaining external state (e.g., additional configuration through other dependencies) that can be used to calculate the delay duration, which can't be achieved with the standard delay strategies.

4.2.2 Ignoring Exceptions

The idea of ignoring exceptions that shouldn't trigger a retry was initially considered, by adding an extra configuration property (as offered by the Resilience4j library [46]). However, it was decided that the *retry-on-error* should be the only policy for determining if an operation should be retried based on the caught exception. The same principle was applied to the *retry-on-result* policy, which determines if the operation should be retried based on the result of the operation.

4.2.3 Result Mapper

During the initial development phases, if an operation failed after multiple retries, the last caught exception was always rethrown or if successful, the result was returned as is. This approach is sufficient for most use cases, and the studied implementations of the Retry mechanism follow this pattern, but it was decided to provide a more flexible solution (e.g., the caller doesn't want a specific exception to be thrown, but instead wants to log it only). To address this, an exception handler was introduced to the configuration. This handler allowed for additional processing of the caught exception (e.g., decide if it should be rethrown or logged), but still did not provide a way to modify the result of the underlying operation.

To provide even more flexibility, a result mapper was introduced. This mapper enables customized handling of the result or exception once the Retry mechanism could not be used any more, allowing for more complex scenarios to be managed while keeping the already established functionality intact. Additionally, the result mapper can be configured independently for each decoration. If omitted, the exception handler specified in the Retry mechanism configuration is used to manage exceptions, while the original result is returned unmodified.

4.2.4 Retry Context

The retry context represents the asynchronous context mechanism's model component implementation (see Section 3.1.3), which is responsible for managing the state and controlling the flow of retryable operations. It maintains internal state information, such as the number of retry attempts and the last-caught exception, and has access to the configuration settings defined for the Retry mechanism. Additionally, it emits events to potential listeners.

In the later stages of development, the retry context was made available to the operation being retried (see Section **??**), allowing it to access the context and potentially alter its behaviour based on an immutable version of its state.

Event Emission

The retry context emits the following events when the operation:

- Has succeeded and the Retry mechanism is no longer needed;
- Is about to be retried;
- Has failed, and the Retry mechanism cannot be used any more. Two different event types can be emitted in this case based on if the last-caught exception was expected or not.

4.2.5 Execution Flow

A Retry mechanism can be used to decorate an operation. Each time the operation is executed, a new retry context is created (per-method invocation). The outcome of the operation—success or failure—is determined by the policies defined in the retry configuration.

When the underlying operation is executed, the retry context captures any exception or result that occurs. If the operation fails, the retry context consults the policies defined in the configuration to decide if a retry should be attempted. If so, the operation is repeated after a delay, calculated based on the specified delay strategy. But if the retry context determines that the operation should not be retried, the operation is considered to have failed.

If the operation eventually succeeds through the Retry mechanism, the retry is considered successful. In all cases where it cannot be used any more (e.g., the maximum number of attempts has been reached), the result mapper is called to map the final result before it is returned to the caller.

This description can be seen in the Retry mechanism execution flow diagram, as illustrated in Figure 4.2.

4.3 Configuration

The Retry mechanism can be configured using a dedicated configuration builder with the properties listed in Table 4.1.

The default configuration can be overriden by calling the respective builder methods. The values that compose the default configuration are the most common and recommended for most use cases, but they can be adjusted to better suit the application's needs.

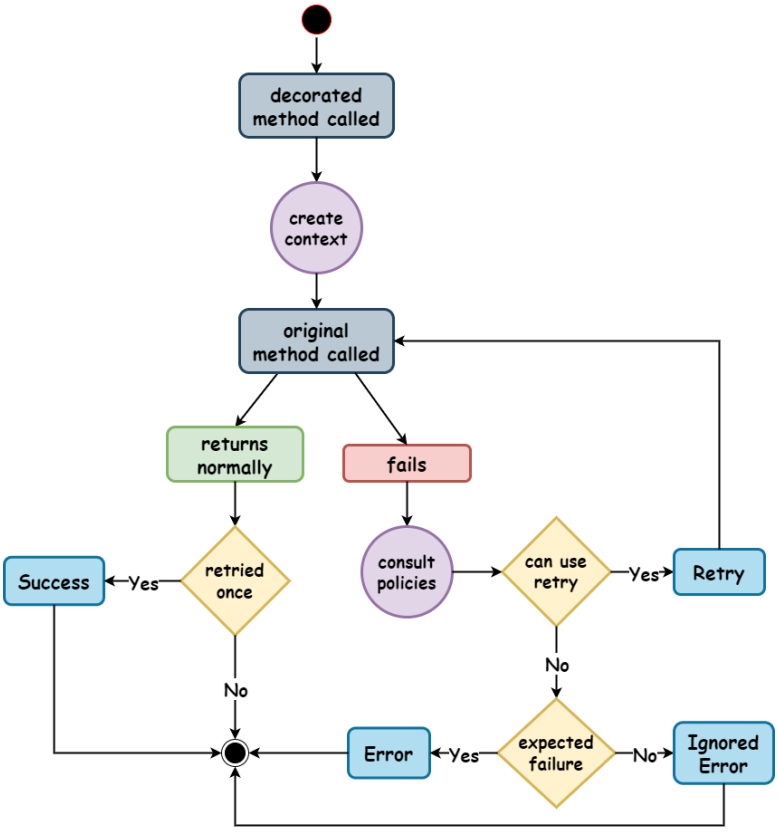

Figure 4.2: Retry Mechanism Execution Flow

Table 4.1: Configuration Properties for `RetryConfigBuilder`

Config Property	Default Value/Behaviour	Description
`maxAttempts`	`3`	The maximum number of attempts (including the initial call as the first attempt).
`retryPredicate`	`throwable -> true`	Predicate to determine if the operation should be retried based on the caught throwable.
`retryOnResultPredicate`	`result -> false`	Predicate to determine if the operation should be retried based on the result of the operation.
`delayStrategy`	`Exponential[ initialDelay=500ms, multiplier=2.0, maxDelay=1m]`	The strategy for calculating delay between retries.
`exceptionHandler`	`Rethrow throwable if any`	The handler for exceptions that occur during a retry operation.

4.4 Ktor Integration

The Retry mechanism was integrated with Ktor by implementing a custom plugin that can be added to the application's Ktor pipeline.

4.4.1 Plugin Implementation

The plugin was designed for Ktor clients to enable automatic retries for HTTP requests. It intercepts the sending phase of a request, applying configurable retry logic based on conditions such as response status or exceptions.

Before retrying a request, the plugin copies the original request and propagates its completion state to the copy, ensuring completion synchronization between the original and the retry request. This ensures that if the original request is cancelled or encounters an error, the retry attempt is promptly cancelled or handled accordingly, leveraging structured concurrency principles. It helps maintain consistency and reliability in handling retries, particularly in scenarios involving intermittent network issues or server errors.

The plugin allows for two levels of configuration:

- **Global Configuration**: The configuration is set during the plugin installation and applies to all requests made by the client;
- **Per-Request Customization**: The configuration can be overridden for a specific request, allowing for more fine-grained control over the retry logic (e.g., disabling the Retry mechanism for a specific request).

A listener to log all events emitted by the retry context is also enabled, which cancels its subscription when the request is completed.

4.4.2 Configuration

The Retry Ktor plugin can be configured by using a dedicated configuration builder with the properties listed in Table 4.2. For context, a predicate is a function that returns a boolean value based on the input parameters.

Additionally, methods relevant to an HTTP context were added to the configuration builder to simplify the configuration process:

- `retryOnServerErrors`: Retries the HTTP call if the response status code is in the range: 500-599. Based on `retryOnCallPredicate`;
- `retryOnServerErrorsIfIdempotent`: Similar to `retryOnServerErrors`, but only retries if the request method is idempotent [47] (e.g., `GET, PUT, DELETE`);
- `retryOnTimeout`: Retries the HTTP call if the exception thrown is a timeout exception (e.g., connection timeout exceeded, request timeout exceeded). Based on `retryOnExceptionPredicate`;

Table 4.2: Configuration Properties for `RetryPluginConfigBuilder`

Config Property	**Default Value/Behaviour**	**Description**
`maxAttempts`	`3`	The maximum number of attempts (including the initial call as the first attempt).
`retryOnExceptionPredicate`	`throwable -> true`	Predicate to determine if the operation should be retried based on the caught throwable.
`delayStrategy`	`Exponential[ initialDelay=500ms, multiplier=2.0, maxDelay=1m]`	The strategy for calculating delay between retries.
`retryOnCallPredicate`	`(request, response) -> retries if a 5xx response is received from a server`	Predicate to determine if an HTTP call should be retried based on the respective request and response.
`modifyRequestOnRetry`	`(requestBuilder, attempt) -> no-operation`	Callback to modify the request between retries (e.g., add a header with the current attempt number).

Chapter 5

Circuit Breaker

This chapter provides an in-depth look at the Circuit Breaker reactive resilience mechanism, explaining the problems it aims to solve and how it addresses them through its functionality. It also covers available configurations and implementation details, providing descriptions and examples for both the library version and its integration as a plugin within the Ktor framework.

5.1 Introduction

The Circuit Breaker mechanism is a *"resilience pattern that prevents an application from performing an operation that is likely to fail. By allowing it to continue without waiting for the fault to be fixed or wasting CPU cycles while it determines that the fault is long-lasting "* [48].

When an application calls a remote service, there's always a risk of failure (e.g., network issues, service unavailability, or timeouts). Some of these faults are temporary (transient) and resolved quickly, but others can be long-lasting and severe (e.g., a complete loss of connectivity, complete failure of a service).

Moreover, if a service is under heavy-load, a failure in one part of the system might lead to cascading failures. For example, if an operation is set to time out and fail after a certain period, multiple concurrent requests could be blocked until the timeout expires. These blocked requests consume critical resources like memory, threads, and database connections, potentially leading to system-wide failures [48]. In such cases, it's more efficient for the application to quickly acknowledge the failure and handle it appropriately, rather than repeatedly retrying the operation.

5.1.1 Relation to the Retry Mechanism

The Circuit Breaker mechanism is related to the retry mechanism (see Chapter 4), as both are used to handle faults that might occur when connecting to a remote service or resource. However, the two mechanisms are used in different situations. The retry mechanism is used to retry an operation in the expectation that it'll succeed, while the Circuit Breaker mechanism is used to prevent an application from performing an operation that is likely to fail [48].

An application can combine these two mechanisms by using the retry mechanism to invoke an operation through a Circuit Breaker. However, the retry logic should be sensitive to any failures returned by the Circuit Breaker and abandon retry attempts if it indicates that a failure is not transient [48].

5.1.2 State Machine

The Circuit Breaker mechanism works as an electrical circuit breaker [49], which is a safety device that automatically stops the flow of electric current in a circuit as a safety measure (e.g., to prevent a fire) when it detects a fault condition. In contrast to the electrical circuit breaker which needs a manual reset, the Circuit Breaker mechanism automatically resets itself if the fault condition is considered resolved.

The default implementation of the Circuit Breaker uses a state machine, as illustrated in Figure 5.1, with the following states:

- **Closed** - The circuit allows the operation to execute and records its execution result (i.e., success or failure). If a configurable failure threshold is reached, the circuit transitions to the open state;
- **Open** - The circuit does not allow the operation to be executed. Instead, a predefined failure result is returned. The circuit remains open for a configurable period of time, after which it transitions to the Half-open state. This timer is used to allow the system to recover from the fault condition;
- **Half-Open** - The circuit allows a limited number of requests to execute the operation. If the combined results of these requests and the recorded ones are below the failure threshold, it is assumed that the fault causing the failure has been resolved. Consequently, the Circuit Breaker switches to the `Closed` state. If not, the Circuit Breaker assumes that the fault is still present, so it reverts to the `Open` state (restarting the timeout timer).

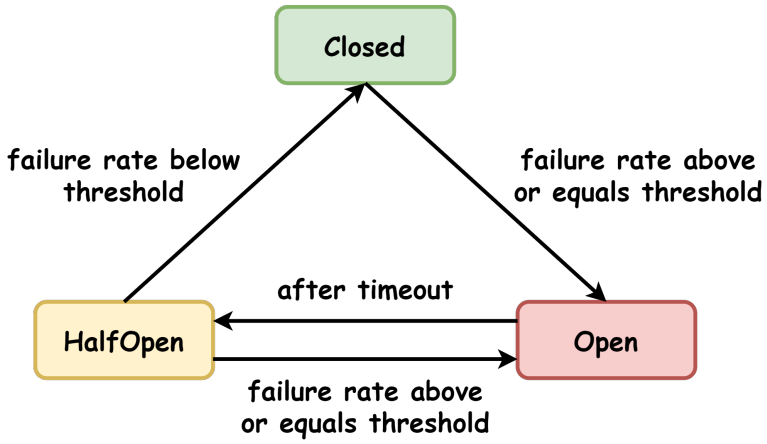

Figure 5.1: Circuit Breaker State Machine.

The `Half-Open` state is useful to prevent a recovering service from suddenly being flooded with requests. As a service recovers, it might be able to support a limited volume of requests until the recovery is complete, but while recovery is in progress, a flood of work can cause the service to time out or fail again [48].

Other implementations might have additional states (e.g., for maintenance, testing, metrics purposes), usually manually triggered, such as:

- **Forced Open** - The circuit is always open, preventing the operation from executing.

- **Forced Closed** - The circuit is always closed, allowing the operation to execute as if the Circuit Breaker was not present.

5.1.3 Operation Execution

An operation protected by a Circuit Breaker is executed through the Circuit Breaker. When the operation is invoked, the Circuit Breaker checks its state to determine if the operation should be executed. The same Circuit Breaker instance can be used to protect multiple operations, sharing the same state and configuration, as illustrated in Figure 5.2.

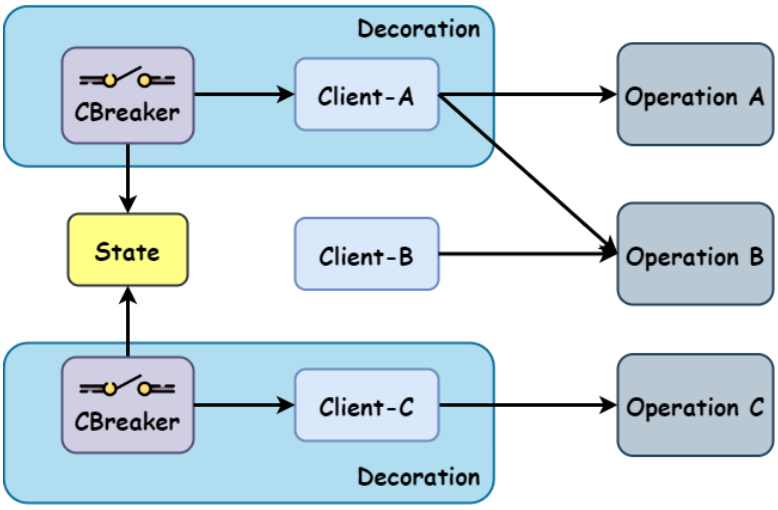

Figure 5.2: Circuit Breaker Decoration.

Regarding the underlying operation execution, it's important to note that the Circuit Breaker does not synchronize the received calls (i.e., it does not prevent multiple threads from invoking the operation concurrently when the circuit is in a state that allows the operation to be executed). To restrict the number of concurrent calls to the operation, the Bulkhead [50] mechanism should be used in combination with the Circuit Breaker.

5.1.4 Recording Execution Results

The Circuit Breaker mechanism records the results of the operation executions to determine when to open the circuit. This recording process uses a sliding window [51] which can be implemented in different ways:

- **Count-based** - The Circuit Breaker opens when the number of failures exceeds a predefined threshold (e.g., 50% of the last 10 operations failed, assuming a window size of 10).
- **Time-based** - The Circuit Breaker opens when the failure rate exceeds a predefined threshold over a specific period of time (e.g., 50% of the operations failed within the last 10 seconds, assuming a window size of 10 with one-second intervals).

The time-based sliding window is used to track the outcomes of recent operation executions over a specific time interval, making decisions based on recent performance rather than solely relying on historical data (which might not be relevant in some cases).

As newer results are recorded, older results are removed from the window, keeping a FIFO (First In, First Out) order. This approach ensures that the window size remains constant and that the Circuit Breaker can make decisions based on the most recent data.

5.2 Implementation Aspects

The Circuit Breaker mechanism was implemented with the following components:

- **State**: Manages the Circuit Breaker state transitions and mutual exclusion;
  - **Reducer** - Processes actions and updates the state accordingly;
  - **Sliding Window** - Records the operation's execution results;
- **Circuit Breaker** - Consults the state, executes operations and dispatches actions to the state reducer.

An example of the interaction between these components is illustrated in Figure 5.3, where the Circuit Breaker receives two orderly requests to execute an operation after working for some time:

1. Client-A requests the operation to be executed. As the operation is protected by the Circuit Breaker, the Circuit Breaker is consulted.
2. The Circuit Breaker checks the shared state, in mutual exclusion, and sees that it is `Closed`, allowing the operation to be executed under its vigilance.
3. The operation is executed outside mutual exclusion.
4. The operation succeeds, and the result is recorded by the Circuit Breaker. Even though the operation succeeded, the minimum throughput is reached, and as such, the failure rate is considered. Since the failure rate equals the threshold, the Circuit Breaker transitions to the `Open` state.
5. Client-B requests the operation to be executed through the Circuit Breaker.
6. The Circuit Breaker checks the shared state, in mutual exclusion, and sees that it is `Open`, thus preventing the operation from being executed.
7. Client-B receives a predefined failure result.

5.2.1 State Reducer

The Circuit Breaker was implemented using the state reducer pattern, in order to delegate the state transitions and mutual exclusion to a single component.

The reducer pattern is a design pattern that allows the state of a system to be managed by a single reducer function that takes the current state and an action as input and returns the new state. It is a common pattern in the Redux [52] state management library and the React [53] library, both widely used in the JavaScript ecosystem.

As illustrated in Figure 5.4, the reducer pattern has the following characteristics:

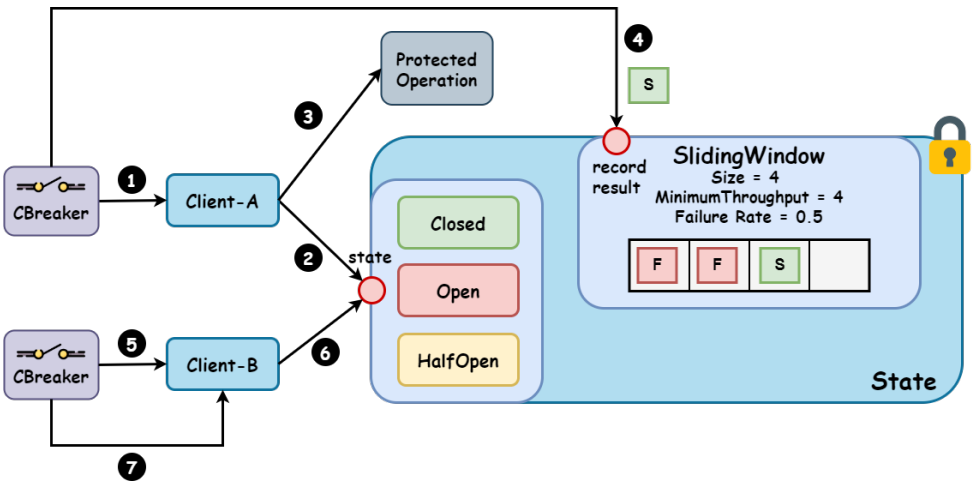

Figure 5.3: Circuit Breaker Execution Example.

- **State** - The state of the system is stored in a single immutable object and can be consulted as a snapshot of the system at any point in time;
- **Action** - An event that potentially triggers a state transition;
- **Dispatch** - A function that sends an action to the reducer to trigger a state transition. In more complex scenarios, it can also dispatch multiple actions based on a single received action, and handle side effects;
- **Reducer** - A function that takes the current state and an action as input and returns the new state and a list of effects (which could be empty) to be executed outside the reducer. This function must be a pure function, meaning it must not have side effects and return the same output for the same input (deterministic).
- **Effect** - Represents a potential side effect that is returned by the reducer as an event and handled by the dispatcher (e.g., logging, fetching external data).

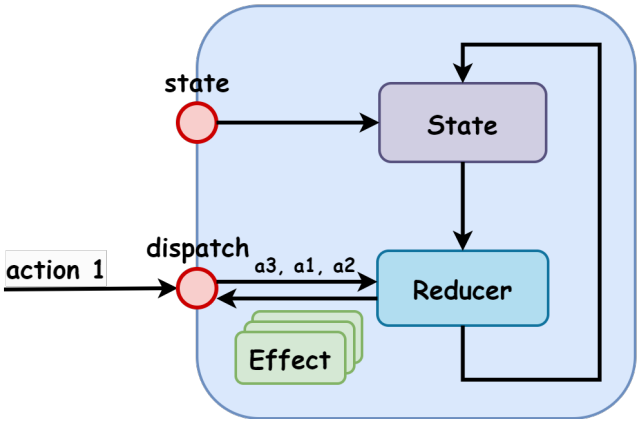

Figure 5.4: State Reducer Pattern.

5.2.2 Sliding Window

The sliding window, which is part of the Circuit Breaker's state, is used to record the operation's execution results and calculate the failure rate. It is implemented as a circular/ring buffer.

A circular buffer is a fixed-size buffer with a pointer to indicate the next empty position, incrementing with each new entry. When full, new data overwrites the oldest data, as the pointer wraps around to the beginning of the buffer, as illustrated in Figure 5.5. This eliminates the need to shift elements for new entries and allows for constant time complexity for adding elements [54].

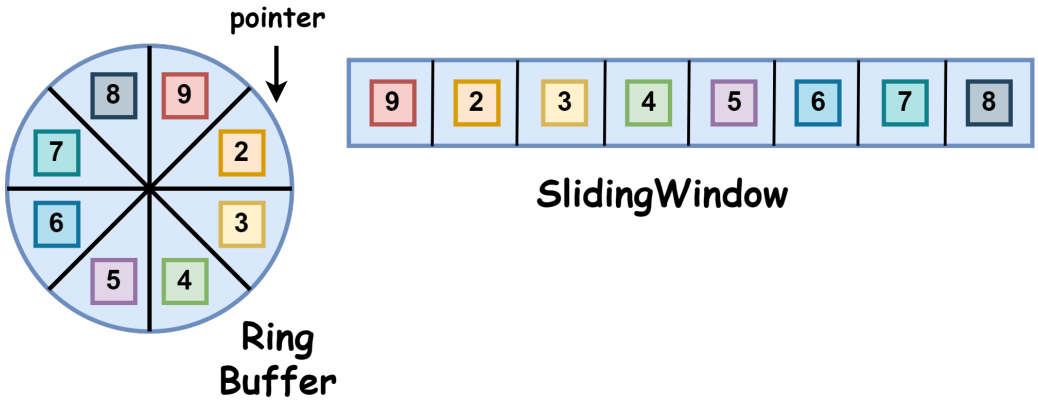

Figure 5.5: Circular Buffer.

5.2.3 Delay Strategy

The delay strategy is used to determine the duration the Circuit Breaker will remain in the `Open` state before transitioning to the `HalfOpen` state.

This mechanism presents the same delay strategy options as the Retry mechanism (See Section 4.1.1 and Section 4.2.1).

5.2.4 Delay Handling

The first approach to handle the delay in the `Open` state was to launch a coroutine that transitions, after a delay, the Circuit Breaker to the `HalfOpen` state. But this approach had two problems: handling cancellation and the reducer presenting side effects.

Instead, a comparable time marker was used to store the exact time when the Circuit Breaker transitioned to the `Open` state. This marker is consulted when callers send external events to the reducer component, behaving as an internal event. Even on state observations, the reducer component receives an event with the current time to determine if a transition is needed. If the delay has passed, the Circuit Breaker transitions, automatically, to the `HalfOpen` state, before attending any external events and potentially ignoring them.

The same approach was used for the `HalfOpen` state, where the Circuit Breaker transitions back to the `Open` state if the maximum wait duration, specified in the configuration, has passed.

5.2.5 Manual State Transition and Reset

In later stages of the implementation, the Circuit Breaker functionality was extended to override the current state with a new state, by allowing the caller to perform a manual state transition. This feature was considered useful for testing, debugging, and maintenance purposes (e.g., maintain the Circuit Breaker in the `Closed` state to only record the operation's execution results). It's worth noting that a manual state transition maintains the same behaviour as an automatic state transition, and won't perform any additional actions. For example, a transition from the `HalfOpen` state to the `Open` state will advance the delay strategy attempt counter by one.

A reset, however, places the Circuit Breaker in the `Closed` state, clearing the sliding window and resetting the failure rate. Effectively resetting the Circuit Breaker to its initial state.

5.2.6 Retry After

In some use cases, when a call is rejected by the Circuit Breaker, it might be useful to provide the caller with information on when the operation can be retried. In this scenario, the Circuit Breaker can return a recommended minimum duration for the caller to wait before retrying the operation. This duration is equivalent to the time remaining for the Circuit Breaker to transition to the `HalfOpen` state from the `Open` state. However, a call can be rejected by the Circuit Breaker for other reasons, such as the permitted number of calls in the `HalfOpen` state being exceeded. In this case, the Circuit Breaker can provide a retry-after period equivalent to the total duration of the `Open` state to give the system enough time to stabilize before the next retry attempt.

With this information, an (outer) Retry mechanism can be configured in the pipeline before the (inner) Circuit Breaker mechanism. When the Circuit Breaker rejects a call, the Retry mechanism uses the provided retry-after period to delay the next retry attempt. Note that this behaviour can only be possible if the Retry mechanism is sensitive to the Circuit Breaker's rejection response.

5.2.7 Event Emission

The Circuit Breaker emits events to notify listeners when:

- A call is rejected;
- The underlying operation is executed. Two different event types can be emitted in this case based on the operation's success or failure;
- The Circuit Breaker state changes manually or automatically;
- The Circuit Breaker is reset.

5.3 Configuration

The Circuit Breaker mechanism can be configured using a dedicated configuration builder with the properties listed in Table 5.1.

Table 5.1: Configuration Properties for `CircuitBreakerConfigBuilder`

Config Property	Default Value/Behaviour	Description
`failureRateThreshold`	`0.5`	The rate in percentage of calls recorded as failures that will trigger the Circuit Breaker to transition to the `Open` state if equaled or exceeded.
`permittedNumberOfCalls InHalfOpenState`	`10`	The number of calls allowed in the `HalfOpen` state.
`maxWaitDurationInHalf OpenState`	`0`	The maximum duration the Circuit Breaker will wait in the `HalfOpen` state before transitioning back to the `Open` state. If set to `0`, it waits indefinitely until all permitted calls are executed.
`slidingWindow`	`SlidingWindow[ size=100, minimumThroughput=100, type=COUNT_BASED]`	Configures the sliding window used to record calls and calculate the failure rate. Minimum throughput is the minimum number of calls that must be recorded before the failure rate is calculated.
`delayStrategyInOpenState`	`Constant[ delay=1m]`	The strategy used to determine the duration the Circuit Breaker will remain in the `Open` state before transitioning to the `HalfOpen` state.
`recordExceptionPredicate`	`throwable -> true`	Predicate to determine if an exception thrown by the underlying operation should be recorded as a failure, and as such, increase the failure rate.
`recordResultPredicate`	`result -> false`	Predicate to determine if the result of the underlying operation should be recorded as a failure, and as such, increase the failure rate.
`exceptionHandler`	`Rethrow throwable if any`	The handler for exceptions that occur during a call through the Circuit Breaker.

The default configuration can be overriden by calling the respective builder methods. The values that compose the default configuration are the most common and recommended for most use cases, but they can be adjusted to better suit the application's needs.

5.4 Ktor Integration

The Circuit Breaker mechanism was integrated with Ktor by implementing a custom plugin that can be added to the application's Ktor pipeline, using the implemented Circuit Breaker mechanism.

5.4.1 Plugin Implementation

The plugin was designed for Ktor clients to wire the Circuit Breaker mechanism to the HTTP client pipeline. The Circuit Breaker plugin will act before the request is sent to the server, consulting the Circuit Breaker state to determine if the request should be executed and after the response is received, recording the result of the operation. Depending on the result, the Circuit Breaker records the operation as a success or failure.

A listener to log all events emitted by the circuit breaker context is also enabled, which cancels its subscription when the request is completed.

5.4.2 Configuration

The Circuit Breaker Ktor plugin can be configured by using a dedicated configuration builder with the properties listed in Table 5.2.

Table 5.2: Configuration Properties for `CircuitBreakerPluginConfigBuilder`

Config Property	Default Value/Behaviour	Description
`failureRateThreshold`	`0.5`	The rate in percentage of calls recorded as failures that will trigger the Circuit Breaker to transition to the `Open` state if equaled or exceeded.
`permittedNumberOfCalls InHalfOpenState`	`10`	The number of calls allowed in the `HalfOpen` state.
`maxWaitDurationInHalf OpenState`	`0`	The maximum duration the Circuit Breaker will wait in the `HalfOpen` state before transitioning back to the `Open` state. If set to `0`, it waits indefinitely until all permitted calls are executed.
`slidingWindow`	`SlidingWindow[ size=100, minimumThroughput=100, type=COUNT_BASED]`	Configures the sliding window used to record calls and calculate the failure rate. Minimum throughput is the minimum number of calls that must be recorded before the failure rate is calculated.
`delayStrategyInOpenState`	`Exponential[ initialDelay=30s, multiplier=2.0, maxDelay=10m]`	The strategy used to determine the duration the Circuit Breaker will remain in the `Open` state before transitioning to the `HalfOpen` state.
`recordResponseAsFailure Predicate`	`(response) -> records a failure if a 5xx response is received from a server`	Predicate to determine if a response should be recorded as a failure.

Additionally, methods relevant to an HTTP context were added to the configuration builder to simplify the configuration process:

- `recordFailureOnServerErrors`: Records server responses with status codes in the range 500-599 as failures. Based on `recordResponseAsFailurePredicate`;

Another feature thought to be useful was the ability to modify the response when a call is rejected by the Circuit Breaker (e.g., to include a Retry-After header and status code 503). However, this feature was left as a future improvement due to time constraints.

Chapter 6

Rate Limiter

This chapter provides an in-depth look at the Rate Limiter proactive resilience mechanism, explaining the problems it aims to solve and how it addresses them through its functionality. It also covers available configurations and implementation details, providing descriptions and examples for both the library version and its integration as a plugin within the Ktor framework.

6.1 Introduction

The Rate Limiter is a proactive resilience mechanism that aims to control the rate at which requests are made to a service or resource. To understand this concept more thoroughly, it is important to break down the terms that compose it:

- **Rate**: Refers to the frequency at which something occurs that could be bound to a time unit;
- **Limiter**: Refers to a mechanism that imposes a restriction or cap on the extent or quantity of something.

By combining these concepts, a Rate Limiter controls the frequency of requests by setting a maximum limit on the number of requests that can be made within a given time period, as shown in Figure 6.1. Additionally, rate limiting can also be applied to the number of concurrent requests that can be made simultaneously. This helps to prevent overloading downstream services, ensuring stability and availability of the system [55].

Advantages of using a Rate Limiter include [56, 3]:

- **Prevent malicious activities**: Rate limiting can help protect against brute force, denial-of-service, and distributed denial-of-service (DDoS) attacks, web scraping, and other malicious activities [57];
- **Manage Resource Utilization**: By controlling the rate of requests, rate limiting helps to manage resource utilization, prevent resource starvation, and ensure that resources are available for all users following predefined SLAs (Service Level Agreements);
- **Prevent Abuse**: Stops users from monopolizing resources and prevents excessive or unnecessary requests, ensuring fair resource distribution and optimal network or server performance;

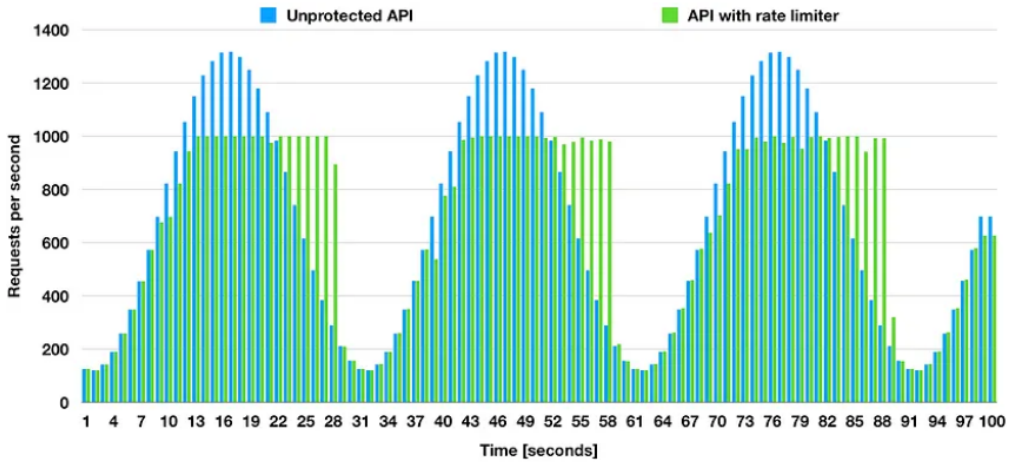

Figure 6.1: API with and without Rate Limiter. Retrieved from [1]

.

- **Improve User Experience**: Rate limiting reduces delays and enhances the responsiveness of the service for legitimate users, improving the overall user experience;
- **Reduce Costs**: Helps avoid extra costs associated with overloading resources, as limiting request rates reduces the demand on resources and prevents the need for additional capacity.

One common implementation of rate limiting is through the use of an API Gateway, which acts as an intermediary between clients and backend services. It serves as a single entry point for API calls, managing and routing them to the appropriate services while also providing various functionalities to enhance security, performance, and usability. Key features of an API Gateway's traffic management include the implementation of rate limiting and throttling policies. These policies control the rate of incoming requests and set rules and limits to regulate traffic, preventing the overloading of backend services [58].

6.1.1 Relation to the Throttling Mechanism

Rate limiting and throttling are closely related concepts often used interchangeably, since both mechanisms control the rate of incoming requests to protect services from being overwhelmed, but they have distinct differences. Rate limiting refers to the process of restricting the number of requests that can be made to a service within a given time period. Throttling, on the other hand, specifically adjusts the flow rate based on current system load. Is typically used to ensure that high-priority requests are served first by delaying less critical ones.

To better illustrate the difference between these two concepts, consider the following analogy: rate limiting is like traffic lights that control the number of cars that can pass through an intersection within a certain time frame, whereas throttling is like adjusting the speed of cars based on traffic conditions to prevent congestion.

6.1.2 Semaphore

A semaphore is a synchronization primitive that restricts the number of simultaneous accesses to a shared resource up to a specified limit. Key characteristics and operations of a semaphore include [59,

60]:

- **Permits**: The semaphore keeps track of available permits. Each permit represents a single unit of resource access that can be granted. Resource access may require multiple permits, depending on the use case;
- **Acquire**: This operation decreases the number of available permits by a given number. If no permits are available, the acquiring entity is blocked (in the case of threads) or suspended (in the case of coroutines) until a permit is released;
- **Release**: This operation increases the number of available permits by a given number. Since releasing permits possibly created conditions for other entities (waiting to acquire permits) to proceed, then the semaphore notifies them.

There are two main types of semaphores:

- **Binary Semaphore**: This type has only two states: available (1 permit) and unavailable (0 permits). In literature, it is also misconceptionally known as a mutex (mutual exclusion lock), but differs in ownership rights. An acquired mutex can only be released by the entity that acquired it, while a semaphore can be signalled by any other entity.
- **Counting Semaphore**: This type can have a count greater than one, allowing multiple entities to acquire permits up to a specified limit. It is used for managing access to a resource pool.

In the context of rate limiting, a semaphore can be used to control the rate of incoming requests by granting or denying access based on the availability of permits.

6.1.3 Rate Limiting Algorithms

There are several algorithms that can be used to implement rate limiting, each with benefits and drawbacks to consider. Therefore, the choice depends on the specific requirements of the system and the challenges it faces (i.e., implementation on a single server or distributed system, handling bursty traffic, etc.) [2, 61]:

Token Bucket

This algorithm is used in packet-switched and telecommunications networks to check data transmissions against defined limits on bandwidth and burstiness (a measure of unevenness or variations in traffic flow). The Token Bucket algorithm is based on the concept of a bucket that holds tokens, where each token represents a unit of data or a permit to transmit data. The key features of the Token Bucket algorithm include:

- Has a fixed capacity (maximum number of tokens it can hold) and a refill rate (tokens added per unit of time);
- Tokens are added to the bucket at a constant rate, up to the maximum capacity;
- When a packet arrives, it must acquire a token from the bucket to proceed;

- If no tokens are available, the packet is considered non-conformant and may be dropped or delayed.

This algorithm provides a variable request rate and is suitable for handling bursts of requests, as long as the bucket has enough tokens to accommodate them. It can be implemented by a semaphore that releases permits in an automated process, without the need for manual release.

Leaky Bucket

The Leaky Bucket algorithm is another rate limiting algorithm used in network traffic management, and is conceptually similar to the Token Bucket algorithm. This algorithm's advantage is that it smooths out bursts of requests and processes them at an approximately constant rate (analogous to water leaking out of a bucket at a constant rate).

The bucket has a fixed capacity and a leak rate that determines how quickly the bucket empties. When a request arrives, it is placed in the bucket. When the rate of incoming requests exceeds the leak rate, the bucket fills up and overflows, causing the requests to be rejected or marked as non-conformant.

Fixed Window Counter

The Fixed Window Counter algorithm essentially counts the number of requests made within discrete, fixed time intervals.

The key features of the Fixed Window Counter algorithm include:

- Requests are counted within a fixed time window (e.g., 1 second, 1 minute);
- When a request is made, the counter is incremented;
- If the counter exceeds the rate limit, the request is rejected;
- The counter is reset at the beginning of each time window.

The Fixed Window Counter algorithm is easy to implement but can lead to bursty traffic patterns, as requests are not evenly distributed within the time window. For example, if the rate limit is 5 requests per minute and a user makes 5 requests at the end of the time window, they can circumvent the limit by making 5 more requests at the beginning of the next time window, effectively doubling the allowed requests as shown in Figure 6.2.

Additionally, the Fixed Window Counter algorithm can lead to a stampeding effect, where previously rejected requests are retried simultaneously when the time window resets. This effect can cause spikes in traffic and overload the system, especially when dealing with a large number of clients. Note that this algorithm is behaviorally similar to the Token Bucket algorithm, but the latter allows for a gradual accumulation of tokens over time never exceeding the maximum capacity (e.g., 2 tokens per second, up to a maximum of 10 tokens), whereas the former resets the counter at the beginning of each time window.

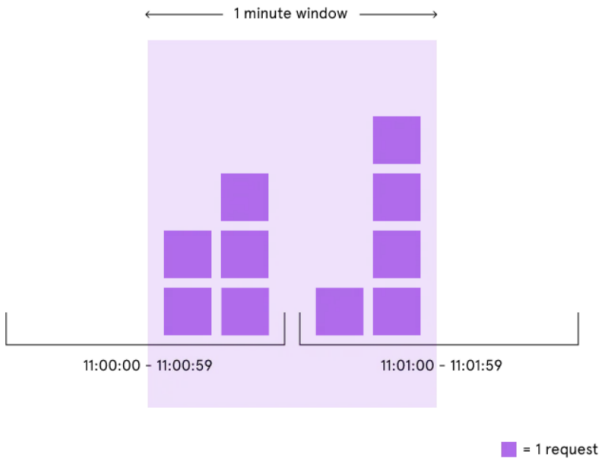

Figure 6.2: Fixed Window Counter Problem. Retrieved from [2]

.

Sliding Window Log

The Sliding Window Log algorithm maintains a continuous record of requests over a rolling time period.

Key features of the Sliding Window Log algorithm include:

- Tracks requests within a continuously moving window (e.g., last 60 seconds);
- Maintains a log of timestamps for each request in the sliding window;
- Checks the log to count the number of requests in the current window whenever a new request is made; As time progresses, the sliding window shifts, adding new requests to the log and removing old ones that fall outside the current window.
- Rejects the request if the sum of requests in the current window exceeds the rate limit.

Unlike the Fixed Window Counter algorithm, the Sliding Window Log algorithm provides a more accurate representation of request rates over time, and it does not suffer from the boundary burstiness problem. However, it can be memory-intensive, as it stores a timestamp for each request, and computationally expensive, as each request requires summing up prior requests, which may not scale well in high-throughput scenarios and distributed architectures.

Sliding Window Counter

The Sliding Window Counter algorithm essentially combines the Fixed Window Counter and Sliding Window Log algorithms to provide a more balanced approach to rate limiting.

Key features of the Sliding Window Counter algorithm include:

- Divides the time window into smaller fixed segments (e.g., 1-second segments within a 60-second window);
- Tracks the count of requests for each segment;

- Calculates the total number of requests in the current window by summing the counts of the relevant segments;
- Rejects the request if the total number of requests within the sliding window exceeds the rate limit;
- Continuously updates the sliding window by shifting the segments as time progresses, ensuring an accurate count for the current window.

This hybrid approach smooths out traffic bursts by weighting the previous window's request rate based on the current timestamp, similar to the sliding log method (e.g., if the current window is 25% through, the previous window's count is weighted by 75%). The Sliding Window Counter algorithm provides the flexibility to scale rate limiting with good performance, avoiding the starvation problem of the Leaky Bucket and the burstiness at the boundary of fixed window implementations [3].

6.1.4 Rate Limit Exceeded

Rate limiting algorithms can be configured to deploy different strategies for handling requests when the rate limit is exceeded (rate-limited requests). The most common approaches include:

- **Reject**: Immediately deny the request and return an error response message (e.g., throwing an exception, returning an HTTP status code such as 429 - Too Many Requests), indicating that the rate limit has been reached;
- **Wait**: Place the request in a queue to be processed later when the rate limit allows, ensuring that the request is not lost and will eventually be handled;
- **Both**: Combine the previous two approaches by placing the request in a queue with a timeout. If the request cannot be processed within the timeout period, it is rejected.

6.1.5 Types of Rate Limiting

Besides the algorithm-based classification, rate limiting can be implemented in various ways depending on the criteria used to limit the rate of requests. A few common types of rate limiting include [62]:

- **Total Requests**: Limits the total number of requests to a service within a given time period;
- **Key-Based**: Limits requests based on specific keys, such as user ID, IP address, or API key, allowing different limits for different users, groups or clients;
- **User-Based**: Applies rate limits per user, ensuring that individual users cannot exceed their allocated request quota as agreed in the SLA.

Available Solutions

Following some of the state-of-the-art libraries that provide rate limiting solutions, The Resilience4j library [4] offers two distinct implementations: an atomic Rate Limiter and a semaphore-based Rate Limiter. The atomic Rate Limiter uses atomic operations to control the rate of requests, where the

caller threads themselves are responsible for managing the rate limit state. In contrast, the semaphore-based Rate Limiter employs a lock-based semaphore and a scheduler to refill it, passively blocking requests when the rate limit is exceeded.

The Polly library [15] provides a thin layer over the .NET `System.Threading.RateLimiting` [63] package, which includes, besides the aforementioned rate limiting algorithms, the following additional rate limiting implementations:

- `ConcurrencyLimiter`: Limits the number of concurrent requests that can be made to a service;
- `PartitionedRateLimiter`: Allows rate limiting based on different keys or partitions;
- `ChainedPartitionedRateLimiter`: Chains multiple Rate Limiters together to provide a cascading rate limiting strategy (e.g., first check a global Rate Limiter, then a more specific partitioned Rate Limiter). If one Rate Limiter fails, all previously acquired permits are released, ensuring atomicity in rate limiting decisions.

Additionally, the package allows configuration of custom policies for handling rate limit exceedances, including queue processing strategies (e.g., newest first, oldest first) and auto-replenishment of permits. Just as Resilience4j, the Polly library offers a lock-free version of the Token Bucket algorithm for improved performance in high-throughput scenarios.

At the time of writing, none of the mentioned libraries provide a distributed rate limiting solution, which is a common requirement in modern architectures where services are deployed across multiple nodes.

6.1.6 Distribution

In a distributed system, rate limiting can be challenging due to the need to maintain consistency across multiple instances of the rate-limited service. One simple way to enforce the limit is to set up sticky sessions in the load balancer so that each consumer gets sent to exactly one node. However, this approach has several disadvantages, including a lack of fault tolerance and scaling problems when nodes get overloaded [3].

A better option is to use a shared data store (e.g., a database) to store the rate limit state, allowing all instances of the service to access and update the state in a consistent manner. This approach has the disadvantage of introducing additional latency due to network communication with the data store and race conditions when multiple instances try to update the state simultaneously.

A naive approach to distributed rate limiting is to use a 'get-then-set' approach, where the rate limit state is read from the data store, incremented, and then written back. The issue with this model is that during the read-increment-store cycle, additional requests may arrive, leading each to attempt storing an incremented counter-value, potentially resulting in an invalid (lower) counter-value. This can enable consumers to send a high rate of requests, bypassing rate limiting controls [3].

A few solutions arise to address these challenges:

- **Locks**: Prevent concurrent access to the rate limit state by using locks to synchronize access. However, locks can introduce performance bottlenecks in high-concurrency scenarios;

- **Atomic Operations**: Use atomic operations provided by the shared data store to increment and check counters efficiently. Using more of the 'set-then-get' approach can reduce the number of round trips to the data store.

To minimize latency introduced by using a shared data store, rate limit checks can be performed locally (in-memory) with an eventually consistent model as shown in Figure 6.3. Each node periodically syncs with the shared store, pushing counter-increments and retrieving updated values (to update the local cache). *"The periodic rate at which nodes converge should be configurable. Shorter sync intervals will result in less divergence of data points when spreading traffic across multiple nodes in the cluster (e.g., when sitting behind a round robin balancer). Whereas longer sync intervals put less read/write pressure on the datastore and less overhead on each node to fetch new synced values"* [3].

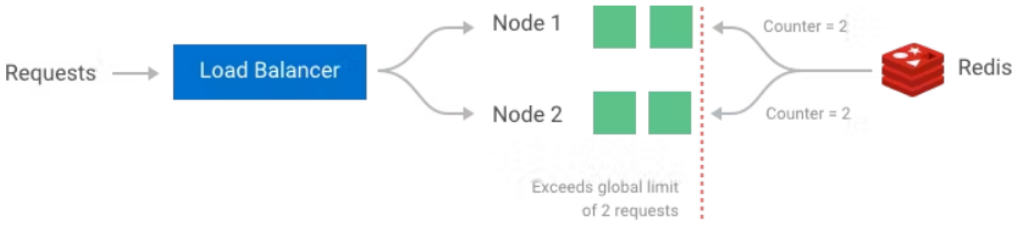

Figure 6.3: Distributed Rate Limiting with a Shared Data Store. Retrieved from [3]

.

6.2 Implementation Aspects

6.2.1 Semaphore-Based Rate Limiter

The semaphore-based Rate Limiter is a simple yet effective way to control the rate of requests to a service. This semaphore controls both the rate frequency and rate concurrency of requests by granting or denying access based on the availability of permits.

The implementation uses a counting semaphore based on a specified algorithm (e.g., Token Bucket, Fixed Window Counter) to apply rate limiting behaviour, namely, the replenishment of permits as described in the respective sections.

Synchronization Style

The adopted synchronization style for the semaphore-based Rate Limiter was the Kernel or Delegation of execution in favor of the Monitor style.

In the Monitor style of synchronization, the thread that creates or sees favorable conditions for other threads to advance to the next state, will signal those threads. It is the responsibility of those other threads to complete their own request of sorts after they exit the condition where they were waiting upon.

In the Kernel or Delegation of execution synchronization style, the thread that creates or sees favorable conditions for other threads to advance to the next state is responsible for completing the requests of those other threads. In successful cases, the threads in the dormant state that were signaled do not have to do anything besides confirming that their request was completed and return

immediately from the synchronizer. This style of synchronization is usually associated with one or more requests that a thread wants to see completed. These same threads will delegate the completion of the requests to another thread, while keeping a local reference to it. This in turn will enable the synchronizer to resume its functions without waiting for the requests to be completed.

The mentioned local reference is what enables the thread to not lose track of the synchronizer state that it was waiting upon. Whereas, in the monitor-style synchronization, such state could be potentially lost since the thread, when awakened, has to recheck it which could have changed in the meantime.

Algorithm Template

Initially, the semaphore-based Rate Limiter only supported the Fixed Window Counter algorithm indirectly. However, the API has been redesigned to be extensible, based on the Template Method pattern [27] allowing for the integration of the various rate limiting algorithms. This new design provides callers with the flexibility to choose the algorithm that best suits their use case and future additions of new algorithms without modifying the existing code.

Drain Operation

Besides the standard acquire and release operations, the semaphore-based Rate Limiter also supports a drain operation. This operation drains all available permits left in the current time window. It is configured based on the result received from the underlying rate-limited service. This is useful for scenarios where the service needs to indicate that it is overwhelmed or needs a break, this way pausing / delaying incoming requests effectively, allowing the service to recover.

6.2.2 Available Types and Distribution

The semaphore-based Rate Limiter controls the rate of requests to a service using a counting semaphore to manage permits. This approach ensures that the rate and concurrency of requests are limited according to the availability of permits, which are managed based on the specified rate limiting algorithm, as described in the previous sections.

In addition to the basic rate limiting mechanism, it was designed, through composition, a Keyed Rate Limiter, as shown in Figure 6.4. The Keyed Rate Limiter provides a fine-grained approach to rate limiting by allowing different rate limits to be set for different keys (e.g., user ID, IP address, API key). This way, the rate of requests can be controlled separately for each key, as the state is maintained independently for each key. This approach is useful for scenarios where different clients or users have different rate limits based on their usage patterns or SLAs, and the abuse of one client does not affect the rate limit of another. Additionally, the same rate limiter configuration can be used to limit the rate of requests based on different keys, leveraging the composition aspect even further.

Both the basic and Keyed Rate Limiters are designed to support distributed environments. In these settings, the state can be stored in a shared data store, which must be implemented manually. By default, the Rate Limiter uses an in-memory data store to manage the state. However, the API is designed to allow replacement with a shared data store (e.g., Redis, Cassandra) to ensure consistency across multiple instances of the rate-limited service.

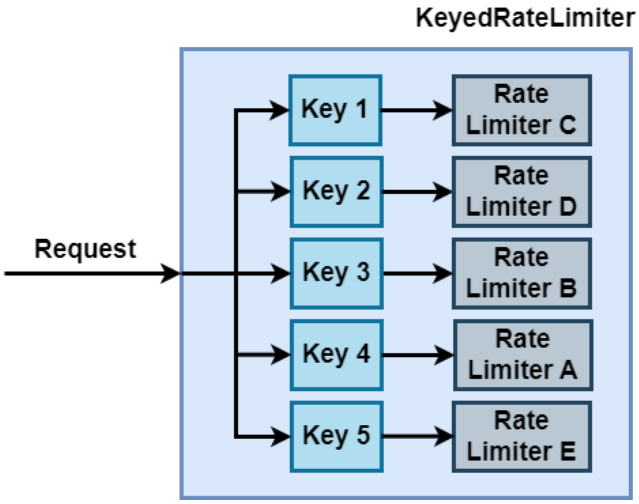

Figure 6.4: Keyed Rate Limiter

Rate Limiter State

The Rate Limiter state consists of the following attributes:

- **Available Permits**: The number of permits available for granting access to requests. This value is decremented when a request acquires permits and incremented when permits are released;
- **Last Replenishment Period**: The timestamp of the last time the permits were replenished;
- **Local Queue**: An optional queue of rate-limited requests that are waiting for permits to be released.

6.2.3 Distributed Rate Limiting

A distributed Rate Limiter is a rate limiting mechanism that was replicated across multiple nodes in a distributed system. Because of this, the shared data store is used to store the rate limit state needed to be thread-safe, as multiple nodes can access and update the state concurrently.

Upon further analysis, several challenges were identified when implementing distributed rate limiting:

- **Release more than one permit**: More than one permit can be acquired per request, so considering the number of permits released, might have created conditions for other requests to advance (not just one if using one permit only release);
- **Enqueued request can give up at any time**: When an enqueued request gives up, the Rate Limiter must remove it from the distributed queue (as well as the local queue) to ensure consistency. However, there's a chance that the distributed queue doesn't have the request since it's in transit and as such, the request cannot give up as it was already completed;
- **Node alert**: How to identify and communicate between different rate-limited service instances;
- **FIFO order garantee**: Between the two or more rate-limited service instances, there's no guarantee that the request that arrived at a given Rate Limiter is the first request that was

enqueued in the distributed queue. As its in the distributed queue that the FIFO order is enforced, not in the local queue.

To address these challenges, an example of a distributed rate limiting architecture was designed as shown in Figure 6.6, and described in the following steps:

1. A caller requests 5 permits to be released on the Rate Limiter of a given node;
2. Since releasing permits might have created conditions for other requests to advance, the Rate Limiter consults the distributed queue to see if there are any requests waiting;
3. The Rate Limiter sees that the oldest request in the distributed queue is waiting for 4 permits but was processed by another node. As such, it removes the request from the distributed queue;
4. In order to complete that node request, the Rate Limiter publishes a message in a pub/sub system [64] to alert it that the request can be removed from its local queue (knowing that each request, in the distributed queue, stores the identifier of the node that processed it);
5. The other node, which had previously subscribed to the pub/sub system, receives the message and removes the (corresponding) request from its local queue, resuming processing;
6. Since the Rate Limiter of the first node only partially completed the possible requests that could advance, it rechecks the distributed queue to see if there are any requests waiting. The Rate Limiter sees that the oldest request in the distributed queue is waiting for 1 permit but was processed by the node where it resides;
7. The Rate Limiter removes the request from the distributed queue and completes the request;
8. Since the request was processed by the same node, it doesn't need to alert itself, and as such, the (corresponding) request is only removed from its local queue, resuming processing.

6.2.4 Disposable Resource

In certain implementations, the Rate Limiter may need to manage resources that require proper cleanup when they are no longer needed. For example, if the Rate Limiter maintains open connections to a database or a shared data store for storing the state, it is crucial to release these resources gracefully to prevent resource leaks and ensure system stability.

To address this, the Rate Limiter implements a disposable pattern, providing a mechanism to close or dispose of any open resources [65]. This mechanism should be called when the Rate Limiter is no longer needed, such as during application shutdown or when the Rate Limiter is being replaced or reconfigured.

6.2.5 Retry after Rate Limited

In some use cases, when a request is rate-limited, it is useful to inform the client about when they should retry the request [66]. For instance, in the Fixed Window Counter algorithm, the optimal

time for the client to retry is at the beginning of the next time window (i.e., the retry duration is algorithm-specific).

With this information, an (outer) Retry mechanism can be configured in the pipeline before the (inner) Rate Limiter to retry the request after the recommended duration. Note that this behaviour can only be possible if the Retry mechanism is sensitive to the Rate Limiter's rejection response.

Additionally, to calculate the retry-after duration, the Rate Limiter should not assume a queue is configured. As such, the calculated duration solely depends on the rate limiting algorithm replenishment period and the total number of permits. The duration represents the minimum time the caller should wait before retrying the request, but does not guarantee that the request will be processed.

A few examples of the retry-after duration calculation in some the aforementioned algorithms are shown in Figure 6.5.

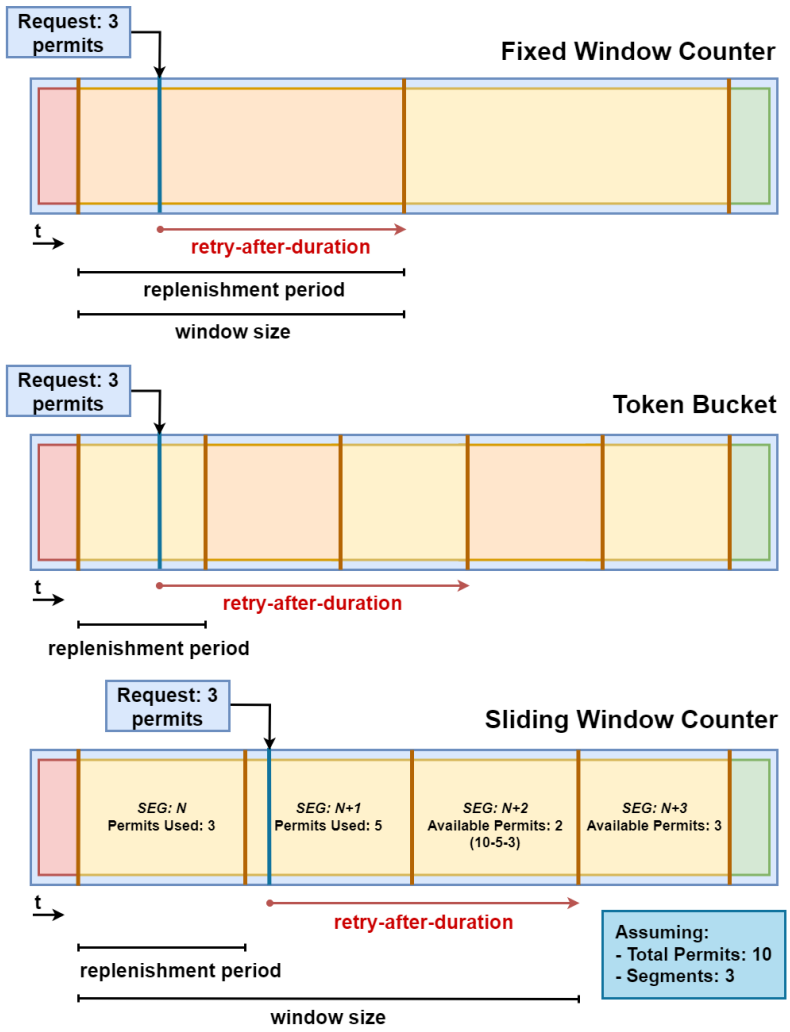

Figure 6.5: Retry After Rate-Limited Examples

6.2.6 Dynamic Configuration

Modern Rate Limiters often need to adapt to changing conditions and requirements dynamically. Dynamic configuration allows modifying the Rate Limiter's behavior at runtime.

A challenge arises when changing the Rate Limiter's configuration dynamically: such changes can

only take effect in the next replenishment period. If applied immediately, the retry-after policy, for example, would be inconsistent with the new configuration, causing previously rate-limited requests to retry based on outdated information.

To address this challenge, the Rate Limiter provides mechanisms to update the configuration dynamically, including the replenishment period and the total number of permits.

Additionally, it's worth noting that in distributed environments that use an eventually consistent model for rate limiting, it can be challenging to ensure that configuration changes are propagated consistently across all nodes.

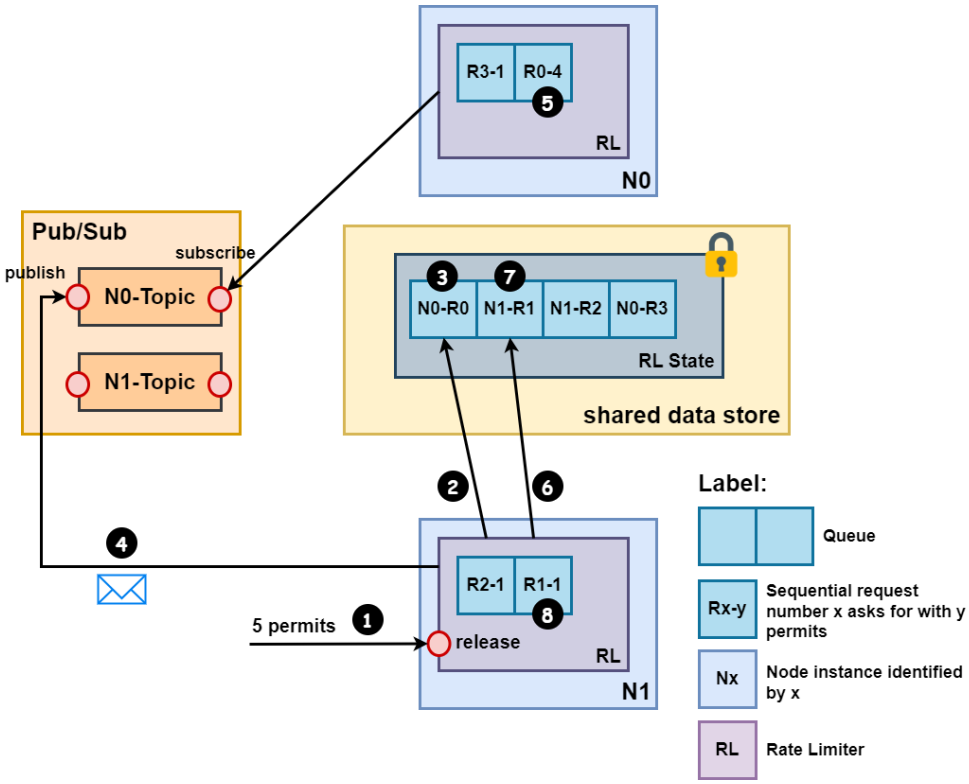

Figure 6.6: Distributed Rate Limiting Architecture Example

6.2.7 Event Emission

The Rate Limiter emits events to notify listeners when a request to a rate-limited service:

- Has succeeded to be processed;
- Has been rate-limited. Two different event types can be emitted in this case based on if the request was immediately rejected or queued.

6.3 Configuration

The Rate Limiter mechanism can be configured using a dedicated configuration builder with the properties listed in Table 6.1.

The default configuration can be overriden by calling the respective builder methods. The values that compose the default configuration are the most common and recommended for most use cases, but they can be adjusted to better suit the application's needs.

Table 6.1: Configuration Properties for `RateLimiterConfigBuilder`

Config Property	Default Value/Behaviour	Description
`algorithm`	`FixedWindowCounter[ totalPermits=1000, replenishmentPeriod=1m, queueLength=0]`	The rate limiting algorithm and its configuration.
`baseTimeoutDuration`	`10s`	The default duration a request will be placed in the queue if rate-limited. After this duration, the request will be rejected.
`onRejected`	`Rethrow throwable if any`	The exception handler that will be called when a request is rejected by the Rate Limiter.

6.4 Ktor Integration

The Rate Limiter plugin integrates with Ktor by implementing a custom plugin that manages rate limiting for HTTP requests in the application's pipeline.

6.4.1 Plugin Implementation

The plugin uses the Keyed Rate Limiter to control the rate of incoming requests based on a configurable key (e.g., client IP, user agent). Initially, the key was enforced to be a string, but it was later changed to be a type that represents any key that can be used to identify a request (e.g., a map).

The plugin intercepts incoming HTTP requests at a configurable phase in the application pipeline. By default, it intercepts requests in the first phase of the process to ensure that rate limiting is applied as early as possible in order to avoid unnecessary processing. However, requests can be rate-limited at any phase in the pipeline by configuring the intercept phase, to handle more complex scenarios (e.g., need to access more request information which is only available in later phases of the pipeline to determine the rate limit key).

When a request is rate-limited, the plugin provides a callback to handle the rejected request to allow for custom behavior, such as placing a *Retry-After* header in the response to inform the client when they can retry the request.

In developing the plugin, there were thoughts of including blacklisting or whitelisting certain requests from rate limiting, even the concept of shadow banning. Ultimately, it was decided, due to time constraints, to exclude these features and solely provide a way to exclude requests from rate limiting based on request properties (e.g., path). To determine if a request is excluded from rate limiting or acquired permits, request attributes were used to transfer state information between the different phases of the pipeline.

Since the plugin uses a *Semaphore-based Rate Limiter* which itself implements a *counting* semaphore, it was deemed necessary to give the caller the ability to calculate the weight of a call for rate limiting. This way, the caller can determine how many permits a request should acquire based on request properties (e.g., bytes transferred).

On a successful request that has passed rate limiting, the plugin provides a callback to handle the

response, allowing for custom behavior to be executed (e.g., adding a custom header to the response to indicate that the request was rate-limited, how many requests are left until the rate limit is reached, etc.).

Additionally, on application shutdown, the plugin ensures that any opened resources are properly disposed of to prevent resource leaks, as each rate limiter enforces a disposable pattern.

The plugin allows for two levels of configuration:

- **Global Configuration**: The default configuration for the Rate Limiter mechanism, which applies to all requests by default;
- **Route-Specific Configuration**: Custom configuration for specific routes or endpoints, allowing for fine-grained control over rate limiting behavior.

6.4.2 Configuration

The Rate Limiter Ktor plugin can be configured using a dedicated configuration builder with the properties listed in Table 6.2.

Table 6.2: Configuration Properties for `RateLimiterPluginConfigBuilder`

Config Property	Default Value/Behaviour	Description
`rateLimiterConfig`	`FixedWindowCounter[ totalPermits=1000, replenishmentPeriod=1m, queueLength=0]`	The configuration for the Rate Limiter mechanism.
`keyResolver`	`call -> concatenates the remote host and user agent from the call's request`	A function to resolve the key for rate limiting from the application call.
`onRejectedCall`	`(call, retryAfterDuration) -> adds a response 'Retry-After' header with the retry duration in seconds and status code 429 (Too Many Requests).`	A callback to handle requests that are rejected due to rate limiting.
`onSuccessCall`	adds a response 'X-Rate-Limited' header with a value of 'false'.	A callback to handle successful requests that have passed rate limiting.
`excludePredicate`	`call -> false`	A predicate to determine if a request should be excluded from rate limiting.
`interceptPhase`	first phase	The phase in the application pipeline to intercept for rate limiting.
`callWeight`	`call -> 1`	A function to calculate the weight of a call for rate limiting.

Chapter 7

Conclusions

The development of the project this document describes was prompted by the identified gap in the Kotlin Multiplatform ecosystem: the absence of a library for fault-tolerance.

The aim was to develop a Kotlin Multiplatform library that provides essential resilience mechanisms, which alone or combined, prevent or mitigate the inevitable failures that occur in distributed systems. The Ktor integration was a later addition to the project, as of the time of writing, is the only Kotlin Multiplatform framework for developing asynchronous server and client services. This addition was a strategic decision to validate the library's implementation and provide immediate usage in a specific and widely-used context - HTTP client and server services.

The project began with a review and analysis of the existing solutions on resilience mechanism implementations, in several platforms and languages, in order to identify the common patterns and practices, as well as the differences and limitations. This research was crucial to understand the current state of the art of resilience mechanisms, and to help identify gaps that the library could address, improve upon, or equally provide. With that knowledge, a model was designed to represent the common interface for all implemented resilience mechanisms by the library - the Mechanism Model. Additionally, the project explored the Ktor framework to understand its architecture and extensibility points.

Various resilience mechanisms were strategically explored in the project, selected to cover both client-side and server-side applications, with an incremental difficulty level from simpler mechanisms like Retry to more complex solutions such as Circuit Breaker and distributed rate limiting. Each mechanism was detailed in terms of their functionality, design and implementation, configuration capabilities, default values associated with each policy, and their integration into the Ktor framework as plugins.

In addition to the code implementation, the project strived to maintain a high level of test coverage and on-site documentation with usage examples and considerations. In the later stages of the project, demos using the plugins were developed to showcase the mechanisms in action in real-world scenarios and from a multiplatform perspective, including running them in a browser using Kotlin/JS.

In summary, this project successfully developed and deployed a dedicated Kotlin Multiplatform library that offers resilience mechanisms, and includes a Ktor integration to leverage these mechanisms in server and client asynchronous architectures. Although the project did not delve deeply into Kotlin-specific details or even programming-related topics, unless absolutely necessary, it aimed to follow the

best practices for idiomatic Kotlin and library design principles.

7.1 Software Engineering Practices

Continuous efforts were made throughout the project to adhere to the best practices for software development and project management. Utilizing GitHub Projects proved invaluable as it provided a comprehensive view of planned tasks, tasks in progress, tasks in review, and tasks completed, enabling asynchronous collaboration and real-time updates. Each issue created was accompanied by detailed considerations, ensuring thoughtful evaluation and organization of development tasks into pull requests and different branches. The default branch remained untouched until deemed merge-ready after a review process. This approach facilitated the tracking of progress and to ensure that the project was on track to meet the established goals. Learning and applying Git commit semantics ensured clear and readable intentions throughout the version control process.

Additionally, proficiency in GitHub Actions enabled the development of workflows that streamlined CI/CD processes. As an example, a dedicated workflow was established for this document to ensure the remote version had the latest updates to be consulted at any time without the need to build it locally.

7.2 Future Work

The library provides a starting point for building reliable and robust distributed systems in the Kotlin Multiplatform ecosystem, but there is still room for improvement and expansion.

The following are some possible future work that can be done to enhance the library in no particular order:

- **Additional Resilience Mechanisms**: Implement additional resilience mechanisms to cover more complex scenarios and requirements, such as Bulkhead, Cache, Timeout, and Fallback. This will complete the set of considered essential mechanisms for building resilient systems. Additionally, implement the respective Ktor plugins for these mechanisms;

- **Registry**: Implement a registry system to access and manage the resilience mechanisms and their configurations at runtime and in a centralized manner;

- **Metrics**: Implement a metrics system to collect information of the resilience mechanisms execution (e.g., the number of retries, the number of failures, etc.) for monitoring and analysis purposes. This could provide, among other benefits, insights on how to adjust the configuration of the mechanisms to improve the system's resilience and responsiveness;

- **Testing Improvements**: Enhance the testing suites with more comprehensive, complex and concurrent test cases to ensure the reliability and robustness of the library, as well as to cover more edge cases and scenarios. Use a dedicated Kotlin Multiplatform testing framework such as Kotest [67] to improve the maintainability and readability of the future developed tests. Tests that involve benchmarking and performance evaluation can also be added to measure the

overhead of the mechanisms, and provide insights on how to optimize them. Extend these tests to cover the Ktor plugins as well;

- **Dedicated Pipeline**: Implement a pipeline, where multiple mechanisms can be attached in a specific order to handle different scenarios (e.g., combine an outer retry with an inner rate limiter, to retry after a request is rejected due to rate limiting);
- **Adapter for JavaScript**: Implement an adapter to call the library from JavaScript code. As the current implementation of the library is only usable in Kotlin code, an adapter is needed to make it accessible from JavaScript;
- **Chaos Engineering**: Besides unit, integration and possibly functional tests, implement a Chaos Engineering module to test the resilience of the systems using the library in a controlled environment.
- **Documentation and Examples**: Improve the documentation and provide more examples to help developers understand and use the library effectively. A website or a dedicated page should be created to host these resources. A documentation engine such as Dokka [68] could be used to generate an additional documentation API website.
- **CI/CD**: Optimize the CI/CD pipeline by integrating tools like Dependabot [69] to receive notifications and updates on the dependencies used in the project and other tools to enhance the development workflow;
- **Selective Dependency Import**: Offer the option to import one or more mechanisms as dependencies, rather than requiring all of them. This will allow developers to include only the necessary parts of the library, enhancing modularity and reducing the application's footprint. The same approach should be applied to the respective Ktor plugins.

Bibliography

[1] Boris Storozhuk. Rate limiter internals in resilience4j. [Online; accessed 3-July-2024].

[2] Nikrad Mahdi. An alternative approach to rate limiting, 2017. [Online; accessed 28-June-2024].

[3] Kong. How to design a scalable rate limiting algorithm. [Online; accessed 30-June-2024].

[4] Resilience4j contributors. Resilience4j: User guide. `https://resilience4j.readme.io/docs/getting-started`, 2024. [Online; accessed 6-March-2024].

[5] FreeCodeCamp contributors. A thorough introduction to distributed systems. `https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c`, 2024. [Online; accessed 5-March-2024].

[6] JetBrains contributors. Kotlin multiplatform. `https://kotlinlang.org/docs/multiplatform.html`, 2024. [Online; accessed 7-March-2024].

[7] Android Developers. Kotlin-first android development. `https://developer.android.com/kotlin/first?hl=en`, 2024. [Online; accessed 7-March-2024].

[8] Google Developers. Android support for kotlin multiplatform to share business logic across mobile, web, server, and desktop. `https://android-developers.googleblog.com/2024/05/android-support-for-kotlin-multiplatform-to-share-business-logic-across-mobile-web-server.html`, 2024. [Online; accessed 22-May-2024].

[9] Touchlab. Google i/o 2024: Kotlin multiplatform at google scale! `https://touchlab.co/KMP-at-google`, 2024. [Online; accessed 22-May-2024].

[10] JetBrains. Kotlin Multiplatform is Stable, 2024. [Online; accessed 28-May-2024].

[11] JetBrains contributors. Ktor: Web applications. `https://ktor.io`, 2024. [Online; accessed 7-March-2024].

[12] Ktor Contributors. Ktor: Client request retry plugin. `https://ktor.io/docs/client-request-retry.html`, 2024. [Online; accessed 22-May-2024].

[13] Ktor Contributors. Ktor: Server rate limit plugin. `https://ktor.io/docs/server-rate-limit.html`, 2024. [Online; accessed 22-May-2024].

[14] Netflix contributors. Hystrix: Latency and fault tolerance for distributed systems. https://github.com/Netflix/Hystrix, 2024. [Online; accessed 6-March-2024].

[15] App-vNext contributors. Polly: Resilience strategies. https://github.com/App-vNext/Polly#resilience-strategies, 2024. [Online; accessed 6-March-2024].

[16] Exoscale. Migrate from hystrix to resilience4j. https://www.exoscale.com/syslog/migrate-from-hystrix-to-resilience4j/, 2024. [Online; accessed 22-May-2024].

[17] Arrow Contributors. Introduction to resilience. https://arrow-kt.io/learn/resilience/intro/, 2024. [Online; accessed 22-May-2024].

[18] JetBrains contributors. Kotlin multiplatform: Github template. https://github.com/Kotlin/multiplatform-library-template, 2024. [Online; accessed 22-May-2024].

[19] Gradle contributors. Gradle. https://gradle.org/, 2024. [Online; accessed 22-May-2024].

[20] GitHub Contributors. Github actions. https://github.com/features/actions, 2024. [Online; accessed 22-May-2024].

[21] Aaron Linskens. The history of maven central and sonatype: A journey from past to present, November 14 2023. [Online; accessed 28-May-2024].

[22] Gradle Contributors. Gradle wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html, 2024. [Online; accessed 22-May-2024].

[23] Gradle. Gradle user manual: Java platform plugin, 2024. [Online; accessed 28-May-2024].

[24] JetBrains contributors. Kotlin multiplatform: Expect/actual. https://kotlinlang.org/docs/multiplatform-expect-actual.html, 2024. [Online; accessed 12-March-2024].

[25] Oracle. Java language specification: Memory model. https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4, 2015. [Online; accessed 10-July-2024].

[26] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 3rd edition, 2017. [Online; accessed 23-May-2024].

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. [Online; accessed 28-May-2024].

[28] JetBrains contributors. Kotlin coroutines. https://kotlinlang.org/docs/coroutines-basics.html, 2024. [Online; accessed 24-May-2024].

[29] Saverio Perugini. *Programming Languages: Concepts and Implementation*. Jones Bartlett Learning, 2021. [Online; accessed 26-May-2024].

[30] Oracle. Thread. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html, 2024. [Online; accessed 26-May-2024].

[31] JetBrains contributors. Kotlin flow. https://kotlinlang.org/docs/flow.html#flows, 2024. [Online; accessed 24-May-2024].

[32] Android. Stateflow and sharedflow. https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=en. [Online; accessed 28-May-2024].

[33] JetBrains contributors. Kotlin sequences. https://kotlinlang.org/docs/sequences.html, 2024. [Online; accessed 24-May-2024].

[34] David Mertz. *Functional Programming in Python*. O'Reilly Media, Inc., 2015. [Online; accessed 26-May-2024].

[35] Oracle. Supplier. https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html, 2024. [Online; accessed 26-May-2024].

[36] Oracle. Function. https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html, 2024. [Online; accessed 26-May-2024].

[37] Oracle. Bifunction. https://docs.oracle.com/javase/8/docs/api/java/util/function/BiFunction.html, 2024. [Online; accessed 26-May-2024].

[38] Oracle. Method references. https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html, 2024. [Online; accessed 26-May-2024].

[39] Ktor Contributors. Adding functionality with server plugins. https://ktor.io/docs/server-plugins.html#add_functionality, 2024. [Online; accessed 23-May-2024].

[40] Ktor Contributors. Ktor server custom plugins. https://ktor.io/docs/server-custom-plugins.html, 2024. [Online; accessed 24-May-2024].

[41] Ktor Contributors. Ktor client custom plugins. https://ktor.io/docs/client-custom-plugins.html, 2024. [Online; accessed 24-May-2024].

[42] Software Testing Help. The difference between unit, integration, and functional testing. https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/, 2024. [Online; accessed 24-May-2024].

[43] Microsoft. Retry pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/retry, 2023. [Online; accessed 24-May-2024].

[44] Amazon Web Services. Exponential backoff and jitter. [Online; accessed 10-July-2024].

[45] Eric S. Raymond. Thundering herd problem. http://www.catb.org/esr/jargon/html/T/thundering-herd-problem.html, 2024. [Online; accessed 10-July-2024].

[46] Resilience4j contributors. Resilience4j: Retry. https://resilience4j.readme.io/docs/retry, 2024. [Online; accessed 26-May-2024].

[47] MDN Web Docs. Idempotent - MDN Web Docs, 2024. [Online; accessed 26-May-2024].

[48] Microsoft. Circuit Breaker pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker, 2023. [Online; accessed 30-May-2024].

[49] ABB Electrification. Circuit breaker basics. [Online; accessed 30-May-2024].

[50] Microsoft. Bulkhead pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead, 2023. [Online; accessed 30-May-2024].

[51] GeeksforGeeks. Window sliding technique, 2024. [Online; accessed 30-May-2024].

[52] Redux Team. Redux Fundamentals, Part 3: State, Actions, and Reducers. [Online; accessed 01-June-2024].

[53] React Team. useReducer - React. [Online; accessed 01-June-2024].

[54] Baeldung. Circular buffer in computer science. [Online; accessed 07-June-2024].

[55] Microsoft. Rate Limiting pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/rate-limiting, 2023. [Online; accessed 23-June-2024].

[56] Solo.io. Ultimate guide to rate limiting. [Online; accessed 28-June-2024].

[57] Cloudflare. What is rate limiting? — rate limiting and bots. [Online; accessed 23-June-2024].

[58] Red Hat. What does an api gateway do? [Online; accessed 28-June-2024].

[59] Oracle. Class semaphore. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html. [Online; accessed 28-June-2024].

[60] Oracle. Multithreaded programming guide. https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032s/index.html, 2010. [Online; accessed 28-June-2024].

[61] Nordic APIs. Different algorithms to implement rate limiting in apis, 2023. [Online; accessed 28-June-2024].

[62] Redis. Rate limiting. https://redis.io/topics/ratelimit, 2024. [Online; accessed 13-July-2024].

[63] Microsoft. Announcing rate limiting for .net. [Online; accessed 30-June-2024].

[64] Microsoft. Publisher-Subscriber pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber. [Online; accessed 7-July-2024].

[65] Microsoft. Implementing a Dispose Method. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose, 2023. [Online; accessed 5-July-2024].

[66] MDN Web Docs. HTTP Status 429: Too Many Requests. [Online; accessed 4-July-2024].

[67] Kotest. Kotest. [Online; accessed 4-July-2024].

[68] JetBrains. Get started with dokka. [Online; accessed 07-June-2024].

[69] GitHub. Dependabot Quickstart Guide. [Online; accessed 4-July-2024].