

# Bachelorarbeit

Cyrille Ngassam Nkwenga

## Reactive Design Patterns Implementierung eines Circuit Breaker Patterns in C++

Cyrille Ngassam Nkwenga

Reactive Design Patterns  
Implementierung eines Circuit Breaker Patterns in  
C++

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung  
im Studiengang Bachelor of Science Technische Informatik  
am Department Informatik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Stephan Pareigis  
Zweitgutachter: Prof. Dr. Jan Sudeikat

Eingereicht am: 11.04.2019

**Cyrille Ngassam Nkwenga**

**Thema der Arbeit**

Reactive Design Patterns

Implementierung eines Circuit Breaker Patterns in C++

**Stichworte**

Antwortbereit, Widerstandsfähig, Elastisch, Nachrichtenorientiert

**Kurzzusammenfassung**

Ein Circuit Breaker Pattern ist ein Software Entwurfsmuster, das dafür verwendet wird, Fehler in einem System zu erkennen und dessen Ausbreitung zu verhindern. Ziel dieser Arbeit ist es, das Circuit Breaker Pattern in C ++ zu implementieren. In dieser Arbeit wird eine detaillierte Analyse des Circuit Breakers durchgeführt, indem die Architektur, das Klassendiagramm und das Sequenzdiagramm des Circuit Breakers definiert und erstellt werden. Komponente wie Thread Pool, Thread Safe Queue und Command Wrapper werden implementiert, um die Funktion des hier entwickelten Circuit Breakers sicherzustellen.

Das Boost Unit Test Framework zum Testen des Circuit Breaker verwendet.

Die Durchführung von Experimenten, wird uns dabei helfen das implementierte Circuit Breaker zu bewerten.

...

---

**Cyrille Ngassam Nkwenga**

**Title of Thesis**

Reactive Design Patterns  
Implementation of Circuit Breaker Pattern in C++

**Keywords**

Responsive, Resilient, Elastic, Message driven

**Abstract**

A circuit breaker is software design pattern which safely connects different parts of the system so that failures do not spread uncontrollably across them. It helps to detect failures and prevent the propagation of failures across the whole system.

The aim of this work is to implement the circuit breaker pattern in C++.

In this work, a detailed analysis of the circuit breaker is performed by defining and creating the architecture, class diagram and sequence diagram of the circuit breaker. Components such as Thread Pool, Thread Safe Queue and Command Wrapper are implemented to ensure the functionality of the circuit breaker developed here.

Boost Unit Test Framework is used to test the circuit breaker.

To evaluate the implemented circuit breaker experiments are run. They consist of running a simulation and observing the circuit breaker behavior. To consolidate the observation made on the simulation, a second test is run where real request are sent to website.

...

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Reactive system properties . . . . .	2
1.1.1 Responsive . . . . .	2
1.1.2 Elastic . . . . .	2
1.1.3 Message driven . . . . .	2
1.1.4 Resilient . . . . .	3
1.2 Motivation . . . . .	3
<b>2 Circuit Breaker</b>	<b>4</b>
2.1 Context . . . . .	4
2.2 Problem . . . . .	4
2.3 Solution . . . . .	6
2.4 Design Requirements . . . . .	8
2.4.1 Functional requirements . . . . .	8
2.4.2 Non-Functional Requirements . . . . .	9
2.5 Design Analysis . . . . .	9
2.5.1 Component decoupling . . . . .	9
2.5.2 Accepting any task type . . . . .	10
2.5.3 State transition . . . . .	11
2.5.4 Failure detection . . . . .	11
2.5.5 Monitoring the circuit breaker . . . . .	11
2.5.6 Non blocking Client . . . . .	12
2.6 Pattern structure . . . . .	14
2.6.1 Sequence Diagrams . . . . .	17

2.7	Implementation with C++ 17 . . . . .	21
2.7.1	Example of how the circuit breaker can be used . . . . .	21
2.7.2	Command Class . . . . .	23
2.7.3	Logger Class . . . . .	25
2.7.4	Circuit Breaker Class implementation . . . . .	26
2.7.5	Sending the request to the service . . . . .	27
2.7.6	sending request on closed state . . . . .	28
2.7.7	sending request on open state . . . . .	29
2.7.8	Sending request on half open state . . . . .	30
2.7.9	Transition and Monitoring . . . . .	31
2.7.10	Thread Pool . . . . .	35
2.7.11	Interrupting the Thread in the Thread Pool . . . . .	37
2.7.12	Thread Safe Queue Class . . . . .	38
2.8	Conclusion . . . . .	42
<b>3</b>	<b>Test Execution</b>	<b>43</b>
3.1	Test Environment . . . . .	43
3.2	Test Strategy . . . . .	44
3.2.1	Simulation . . . . .	44
3.2.2	Web service requests . . . . .	44
3.3	Test Execution . . . . .	45
3.3.1	How to build the simulation . . . . .	45
3.3.2	Compiling the simulation code . . . . .	47
<b>4</b>	<b>Experimental Results</b>	<b>53</b>
4.1	Observations . . . . .	53
4.2	Simulation . . . . .	54
4.2.1	Effect of the deadline . . . . .	54
4.2.2	Effect of the failure threshold . . . . .	59
4.2.3	Effect of the retry time . . . . .	62
4.3	Web service Test . . . . .	65
4.3.1	Effect of the deadline . . . . .	66
4.4	Conclusion . . . . .	68

*Contents*

---

<b>5 Summary</b>	<b>70</b>
<b>A Appendices</b>	<b>74</b>
<b>Glossar</b>	<b>75</b>
<b>Selbstständigkeitserklärung</b>	<b>77</b>

# List of Figures

1.1	Reactive System, from <i>Reactive Manifesto</i> . . . . .	1
2.1	A client send a request to the service . . . . .	5
2.2	Circuit Breaker FSM, from <b><i>Release It !</i></b> by <i>Michael T. Nygard</i> , 1st edition, p.116 [3] . . . . .	6
2.3	A client uses a circuit breaker to send requests to a service . . . . .	7
2.4	Architecture of the Circuit breaker pattern . . . . .	13
2.5	Pattern structure : Circuit Breaker . . . . .	14
2.6	Pattern structure : Thread Pool . . . . .	15
2.7	Pattern structure : Client . . . . .	15
2.8	Pattern structure : Service . . . . .	16
2.9	Pattern structure : Observers . . . . .	16
2.10	A Circuit Breaker Class diagram . . . . .	17
2.11	Sequence diagram : Circuit Breaker operating in closed state . . . . .	18
2.12	Sequence diagram : Circuit Breaker operating in open state . . . . .	19
2.13	Sequence diagram : Circuit Breaker operating in half open state . . . . .	20
3.1	Running CMake to generate build files . . . . .	48
3.2	Running CMake to generate build files . . . . .	49
3.3	Running the simulation . . . . .	49
3.4	Results of the simulation . . . . .	50
3.5	Log file contents . . . . .	51
4.1	Duration in function of the deadline . . . . .	55
4.2	Success ratio in function of the deadline . . . . .	56
4.3	Success ratio in function of the deadline . . . . .	57



*List of Figures*

---

4.4	Trip ratio in function of the deadline . . . . .	58
4.5	Duration in function of the failure threshold . . . . .	60
4.6	Success ratio in function of the deadline . . . . .	61
4.7	Trip ratio in function of the failure threshold . . . . .	62
4.8	Duration in function of the retry time . . . . .	63
4.9	Success ratio in function of the retry time . . . . .	64
4.10	Trip ratio in function of the retry time . . . . .	65
4.11	Web service : Duration in function of deadline . . . . .	66
4.12	Web service : Success ratio in function of deadline . . . . .	67
4.13	Web service : Success ratio in function of deadline . . . . .	68

# 1 Introduction

Reactive is a set of design principles, a way of thinking about system architecture and design in a distributed environment where implementation techniques, tooling and design patterns are components of a larger whole system. A reactive system is an architectural style where every component of that system comes together to offer a service while being individually able to react to its surrounding. Reactive Systems are based on four important principles<sup>1</sup>:

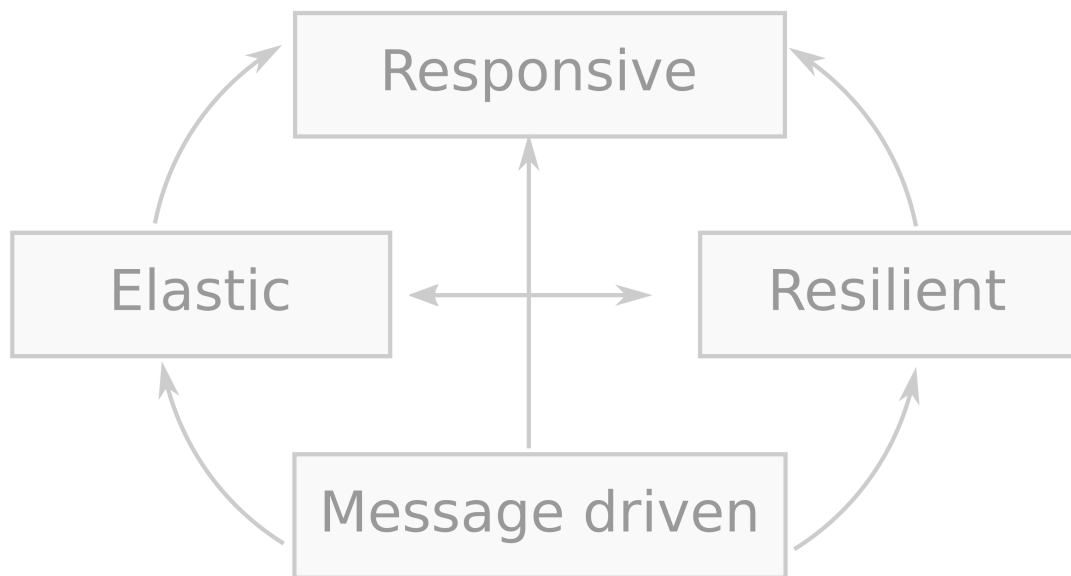


Figure 1.1: Reactive System, from *Reactive Manifesto*

---

<sup>1</sup>The Reactive Manifesto. *The Reactive Manifesto*. 2014. URL: <https://www.reactivemanifesto.org>.

## 1.1 Reactive system properties

As seen in the Fig 1.1, the foundation of reactive system are responsiveness, elasticity, resilience and message driven. Let see what these properties mean.

### 1.1.1 Responsive

The dictionary describes the word **responsive** as : “reacting quickly and positively. a flexible service that is responsive to changing social patterns synonyms: quick to react” A reactive system needs to react as quickly as possible. Responsiveness makes a system usable. It allows a system to detect problems in a timely manner. Responsiveness is more about how the system is reacting to users when it is used.

### 1.1.2 Elastic

The dictionary describes **elastic** as : “able to encompass much variety and change; **flexible and adaptable.**” The system remains responsive when the workload is varying. Under heavy load, reactive systems must adjust the resources used in order to stay responsive. Elasticity is more about how the system is reacting to changing load.

### 1.1.3 Message driven

In a reactive system, the communication among components is based on asynchronous message passing. Asynchronous message ensures loose coupling between the components of a reactive system. It ensures also error delegation, isolation and location transparency. Message driven is more about how the system is reacting to events.

### 1.1.4 Resilient

The dictionary defines **resilient** as : “(of a substance or object) able to recoil or spring **back into shape after bending**, stretching, or being compressed.” A resilient system should stay responsive after a failure. Resilience is achieved by replication, isolation and delegation and containment. Resilience is about how the system is reacting to failures that is happening in the system. The circuit breaker pattern belongs to this category.

## 1.2 Motivation

The circuit breaker pattern belongs to the fault tolerance and replication patterns, which are concerned about resiliency and help the developer to build a resilient system. The circuit breaker pattern allows the developer to monitor the Service’s health and prevent the failures of the Service to cascade across the whole system. It can prevent a client from using a defective Service until this one is fixed.

There are already some implementation of this pattern in programming languages like Java, Ruby, Python, Scala, C#. It was not possible to find a C++ implementation of this pattern<sup>2</sup>. A reason might be that at the time the circuit breaker pattern became known to some developers, the C++ standard (C++98 and C++03) didn’t provide tools to assist the developer in implementing this pattern. C++11 brought the basics tools needed to implement this pattern and C++17 brought a lot more facilities.

The circuit breaker could be used in the field of embedded systems where we see the arrival of technology such as Internet of Things. Today, we have more and more powerful embedded platforms such as Raspeberry PI and Beaglebone that can support heavy workloads.

The aim of this work is to provide a C++ implementation of the circuit breaker pattern.

---

<sup>2</sup><https://github.com/search?q=circuit+breaker>

## 2 Circuit Breaker

In this part a detailed analysis of the circuit breaker is presented. This analysis aims to define the circuit breaker architecture. These steps are necessary since no formal documentation describing the design and architecture of this pattern could be found.

### 2.1 Context

Decoupling two or more Components which need to communicate together.

### 2.2 Problem

In this work we will consider a system environment where two components are communicating together. The first component is using the service offered by the second component. The first component will be called **Client** second component will be called **Service**.

In this context, making a call to the Service can fail. There are many reasons for those failures to occur such as slow networks, remote resources being temporarily unavailable or just some timeouts. Those kind of failures can be fixed by themselves after a short period of time. In some situations faults might take longer to be fixed which can lead to a complete failure of the Service. When the Service is down, every request will fail. In this case, trying to use the Service could lead to a waste

of resources such as thread, memory. Every sent request should *fail immediately* so that the caller handles the failure and continues working.

When the Service is busy by processing a submitted request, it will not be able to process a new request. When processing the request, the service will need some time to complete the request. It might take longer as the usual latency(see image A in Figure 2.1). The client then has to wait until it gets the reply back from the service.

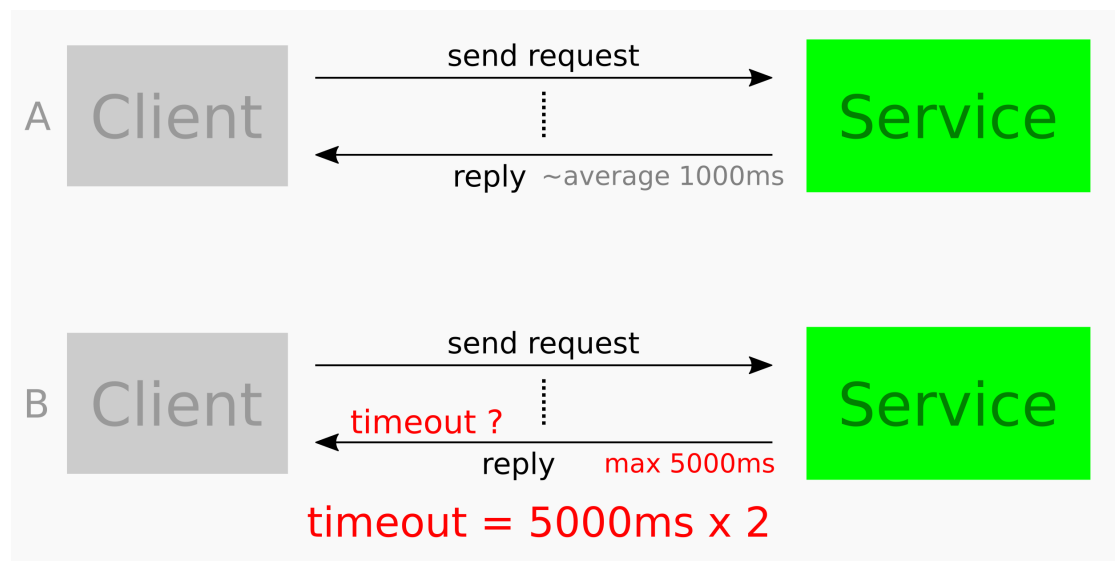


Figure 2.1: A client send a request to the service

How long should the client wait ? One approach would be to set a deadline during which a reply will be required and assume the service has failed in case the caller doesn't receive any answer in this deadline (see image B in Figure 2.1). The problem with this approach is that we now have the caller blocked until the deadline is reached just to know at the end that the Service has failed. What if there are many Service which depend on the caller and they are also blocked ? We are wasting resources such as *threads, database* for example. One solution , and the one that will be implemented in this work is the **Circuit Breaker Pattern**.

## 2.3 Solution

The circuit breaker pattern was first mentioned by Michael T. Nygard in his book **Release It !**[3]. Other author such as Roland Kuhn, Brian Hanafee and Jamie Allen have written description of the pattern in their book **Reactive Design Patterns**[10]. The job of the circuit breaker is to prevent a component from calling a Service that is likely to fail. It does it by wrapping requests to the service. When the service is unreachable the circuit breaker creates a failure response and sends it back to the client. The circuit breaker works in 3 different states as shown in Figure 2.2.

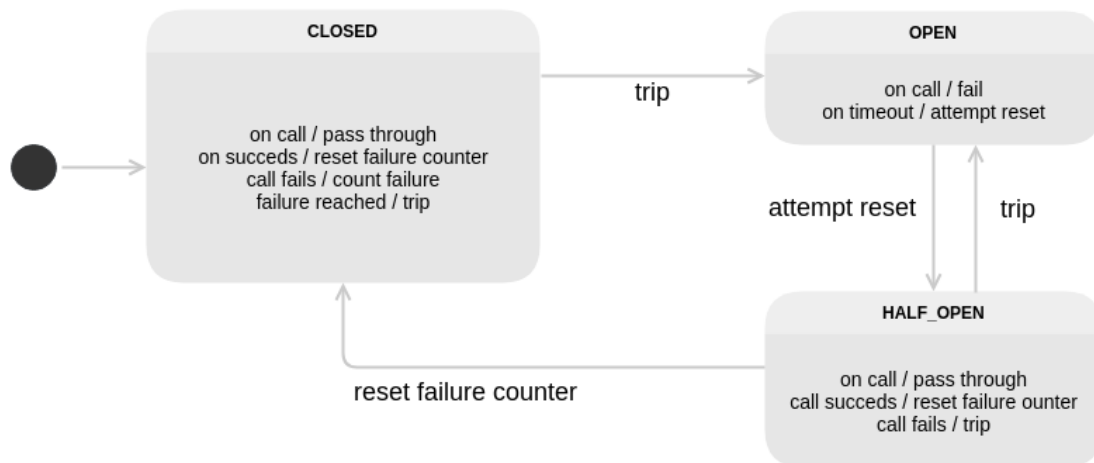


Figure 2.2: Circuit Breaker FSM, from **Release It !** by *Michael T. Nygard*, 1st edition, p.116 [3]

**Closed state** In closed state the circuit breaker lets the client's request go to the service. By a successful reply nothing special happens. By failure the circuit breaker updates the number of failures<sup>1</sup>. When this number of failure reaches a threshold the circuit breaker trips and changes its state to **Open** state.

**Open state** When the circuit breaker is in the Open state, every request sent to the service fails immediately, without letting the request go to the service. After

<sup>1</sup>this parameter will be called failure threshold in this work.

spending a defined time limit<sup>2</sup> the circuit breaker changes its state to Half-Open. It does it because the service might have become available in the meantime.

**Half-Open** In Half-Open state, the circuit breaker lets only one request go through the service and see if the request succeeds. On success, the circuit breaker changes its state to the Closed state. The circuit resets. On fail, the circuit breaker changes its state back to Open state. The circuit trips.

Figure 2.3 shows an example of a client sending a request through the circuit breaker. In this figure, by the first call, the circuit breaker is closed and everything works fine. Then, after successive calls, the circuit breaker trips and opens. In this case, the circuit fails immediately. This operation takes 10ms instead of 1000ms. After a timeout, the circuit breaker changes its state to Half-Open. Now, the client sends a request and the circuit breaker lets it go to the service. The request succeeds and the circuit breaker changes its state to Closed state.

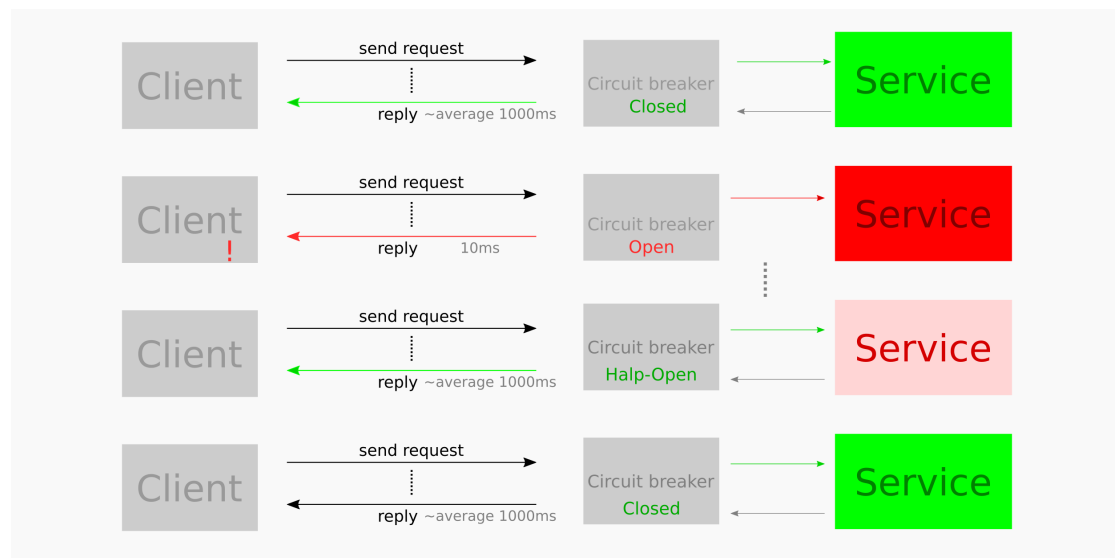


Figure 2.3: A client uses a circuit breaker to send requests to a service

---

<sup>2</sup>this parameter will be called retry time in this work.



The circuit breaker pattern is used on different platform like :

- **Apache Common**<sup>3</sup> has implemented this pattern for the Java language. Apache Common provides the following three components :
  1. EventCountCircuitbreaker ( concrete class )
  2. ThresholdCircuitbreaker ( concrete class )
  3. AbstractCircuitbreaker( A based class to used if you want to implement your own circuit breaker).
- **Spring Cloud** provides Netflix/Hystrix<sup>4</sup> though this code is no more in active development.

Hystrix is used for example by Netflix and Ebay and is the most known circuit breaker implementation in Java.

## 2.4 Design Requirements

The circuit breaker pattern allows us to decouple two Components which need to communicate together.

### 2.4.1 Functional requirements

The circuit breaker pattern is mostly driven by its non-functional requirements rather than by the functional requirements. We will concentrate ourselves on the non-functional requirements of this pattern.

---

<sup>3</sup><https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/concurrent/CircuitBreaker.html>

<sup>4</sup><https://github.com/Netflix/Hystrix/>

## 2.4.2 Non-Functional Requirements

To decouple two components which need to communicate together and prevents failure to cascade across the whole system, the circuit must fulfill the following non-function requirements:

- The Service invocation call must be decoupled from the service execution.
- The circuit must be able to work with any type of service.
- When the failure threshold is reached the circuit breaker must become open.
- When open, the circuit breaker should transition to the half open state after delay defined by the client.
- The circuit breaker must fail fast when it is in open state.
- The circuit breaker must provide a mechanism to detect failures.
- It must be possible to monitor the circuit breaker activity.
- The client doesn't have to block after submitting a request to circuit breaker.

## 2.5 Design Analysis

This section describes the steps that must be taken in order to implement the circuit breaker pattern. All the steps described here are the results of this thesis.

### 2.5.1 Component decoupling

Decoupling method invocation from method execution makes the system asynchronous. This will make the system responsive. The client will be able to continue its work and use the result of its request when it needs it. We need a component which allows us to execute the service in a dedicated thread. Since C++ 11, the STL provides a tools named `std::async()` to achieve it.

The problem with `std::async` is that the standard is not clear about how many threads are created on successive call of `std::async`. On my system, repeatedly calling `std::async` with launch policy argument set to `std::launch::async`, most of the time only one new thread is created and the successive call are run on this same thread, but there were some case in this experiment where successive call on `std::async` created many different thread. A possible solution is to implement the **Active Object pattern**<sup>5</sup>. The Active Object pattern allows us to decouple method invocation from execution. The problem with this pattern is that it uses only one thread, which doesn't scale well when the number of request grows. A better solution is to implement the **Thread Pool Pattern**. The Thread pool pattern allows us to decouple method invocation from execution. The circuit breaker will push the request in the thread pool's task queue and the new task will be run in a free thread. The consequence is that the circuit breaker handles concurrent requests very well. When the time needed to process the request is very short and there are not many of them, using only one thread will be needed. However if processing the request needs much more time and if there is a big number of requests, using only one thread will result in failures of most of all submitted requests. The problem is that requests are queued in a task queue and are processed sequentially in FIFO order. Newly submitted requests will have to wait for the old requests to be served first which make it impossible to hold on the deadline. Using more threads improve the overall circuit breaker performance, hence the use of the Thread Pool pattern.

### 2.5.2 Accepting any task type

The circuit breaker must be able to work with any task type. This allows the client to have different circuit breakers for different services. Being undependable of the task makes the circuit breaker code reusable, which is is huge benefit. C++ 17

---

<sup>5</sup>Douglas Schmidt et al. "Concurrency Patterns". In: *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. 1st ed. John Wiley & Sons, 2000, pp. 318–342. ISBN: 0471606952.

allows us to achieve that with the use of templates and type traits<sup>6</sup>. The benefit we gain from this is that the circuit breaker presents only one public interface that the client can use to submit its request. C++ 17 is required because since C++ 17 class type deduction is now possible. Before C++ 17 template type deduction was only possible in template functions.

### 2.5.3 State transition

A state machine is used to manage the circuit breaker states. Based on the current circuit breaker state the right action will be taken. The state machine works as described in section 2.3. This state machine allows the circuit breaker to fail immediately when the open state is active. This requirement will prevent the client from waiting when the circuit breaker is open. The state machine will also transition from the open state to the half open state the time spent in open state has reached a limit set by the user..

### 2.5.4 Failure detection

When the client has submitted a request, the circuit breaker should be able to notify the client that the request has failed no matter what the reason is. With C++ it is possible to store exception in a `std::future`. Since the circuit breaker returns a `std::future`, by failure the circuit will store an exception describing the error in the `std::future`.

### 2.5.5 Monitoring the circuit breaker

Monitoring the circuit breaker will make it possible to see how well the service is behaving. it makes it possible to see how often the service is failing and take action when needed. To enable monitoring the circuit breaker must provide an

---

<sup>6</sup>C++ 17 provides a header `<type_traits>` which contains many utilities template to use such as `std::invoke_result_t`, `std::decay_t`

interface to query the number of submitted requests, the number of failures, the number of successful requests, the number of time the circuit breaker has been in the open state which is the number of time the failure threshold has been reached. The circuit breaker has three parameters that influence its behavior and thus influences the monitoring attribute. These parameters are **deadline**, **failure threshold** and **retry time**.

### Deadline

This parameter describes how long the circuit breaker will wait for the result. The value of this parameter has a big impact on the circuit breaker behavior.

### Retry time

In the open state, the circuit will wait for some time before it transits from open to the half-open state. This value value defines how long the circuit breaker stays in the open state before checking the service status.

### Failure threshold

This value defines how often the service could fail before the circuit transits from the closed to the open state. There is no proper rule to find the best value. It depends on the context the circuit breaker will be used in. Financial institutes, production company will definitively not allow a big value. In some cases setting this value to 3 might be to high.

### 2.5.6 Non blocking Client

The client doesn't have to block after submitting a request. C++ 17 has `std::future` which allow us to achieve this. The circuit breaker returns a `std::future` when

submitting a request. With `std::future` the client can continue working and fetch the result stored in the `std::future` when needed. By non blocking client the circuit breaker allows the system to be asynchronous.

The analysis of the requirements leads to the system shown in Figure 2.4. This architecture can be resumed as a client - server architecture where the part containing the circuit breaker represents the server.

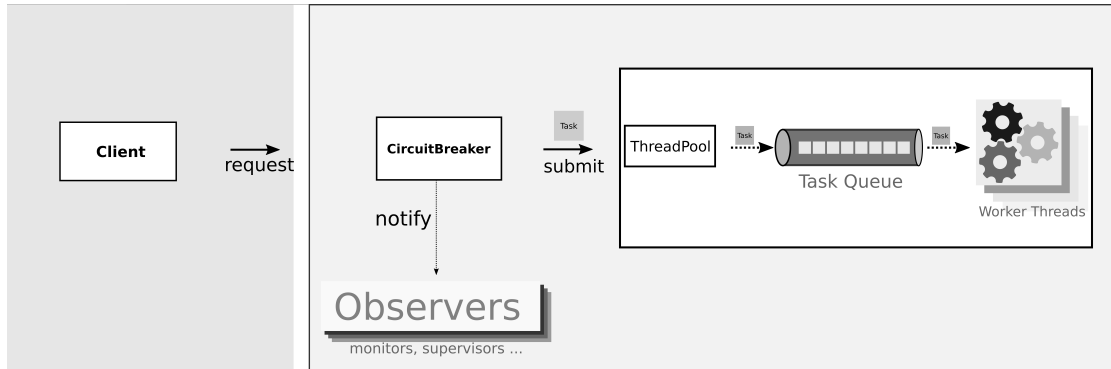


Figure 2.4: Architecture of the Circuit breaker pattern

## 2.6 Pattern structure

This section describes the structure of the circuit breaker pattern.

**Circuit Breaker :** The circuit breaker is the interface between the client and the service. To prevent the client from blocking infinitely, the circuit breaker places the request in a thread pool's task queue which will then be run by a worker thread. The circuit breaker then waits for the results for a time defined by the deadline. Passed this deadline the result is handled back to the client. The circuit breaker also provides information to monitor the service.

<b>Class:</b> CircuitBreaker	<b>Collaborators</b>  * Client  * Threadpool  * Service
<b>Responsibility :</b>  * prevent the client from using a defective component  * prevent the failures of a service to cascade across the whole system  * provide information to monitor the service it is protecting from.	

Figure 2.5: Pattern structure : Circuit Breaker

**Thread Pool :** The thread pool allows concurrency in the system. It avoids the thread creation and destruction overhead when new requests are submitted. In the context of the circuit breaker, it allows the circuit breaker to accept concurrent requests by decoupling method invocation from method execution. For every

accepted request by the circuit breaker, a task is pushed into the task queue and is processed by a worker thread.

<b>Class:</b> ThreadPool	<b>Collaborators</b> * Client * CircuitBreaker * Service
<b>Responsibility :</b>  * decouple method invocation from method execution  * manages concurrent requests  * maintains multiple threads waiting for tasks represented by services to run	

Figure 2.6: Pattern structure : Thread Pool

**Client :** To call the service the client sends requests to the circuit breaker, which depending on its state, forwards the request to the service by pushing the task into the thread pool. When the result or an exception was returned by the service, the client receives the response from the circuit breaker.

<b>Class:</b> Client	<b>Collaborators</b>  * CircuitBreaker
<b>Responsibility :</b>  * sends requests to the service	



Figure 2.7: Pattern structure : Client

**Service :** The service is the component that processes the request submitted by the client. On new request the service is executed by the thread pool. When something happened the service throws an exception to inform the client and the circuit breaker about the failures.

<b>Class:</b> Service	<b>Collaborators</b>
<b>Responsibility :</b>  * process the request sent by the client	* Client * CircuitBreaker * ThreadPool

Figure 2.8: Pattern structure : Service

**Observers** Observers are optional participants which get notified when an event occurred. The circuit breaker chooses the events it can send notifications and interested observers register themselves by the circuit breaker to get notified when these events occur. These observers could be monitors, supervisors other any other components.

<b>Class:</b> Observers	<b>Collaborators</b>
<b>Responsibility :</b>  * monitor the circuit breaker	* CircuitBreaker

Figure 2.9: Pattern structure : Observers

The analysis of the requirements listed above and the pattern structure described lead to the class diagram shown in Figure 2.10. This diagram shows the structure of the circuit breaker which has been implemented.

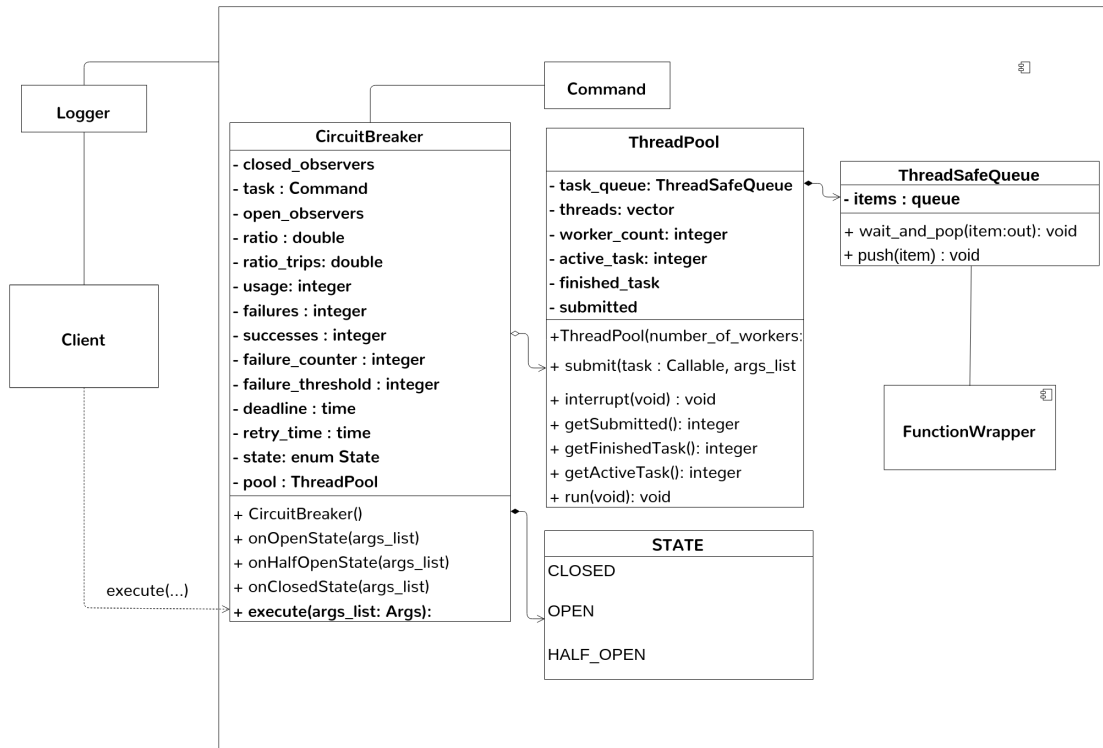


Figure 2.10: A Circuit Breaker Class diagram

### 2.6.1 Sequence Diagrams

This section describes the relevant scenarios when using the circuit breaker pattern. As mentioned previously in section 2.3, the circuit breaker has three states and only one of them is active at a time.

**Scenario I : Close state**

In the scenario described in Figure 2.11. The client sends a request to the circuit breaker. The circuit breaker accepts the request and submits it to the thread pool. The thread pool creates a new task and pushes this task in its task queue. The task will then be concurrently run in a worker thread. The thread pool updates its local attributes and return a future object back to the circuit breaker. The returned future is then checked by the circuit breaker to detect failure or success. In case of failure, the circuit breaker checks first if the failure threshold is reached. If the failure threshold is reached the circuit breaker changes its state to the open state. The client regains control and returns the result.

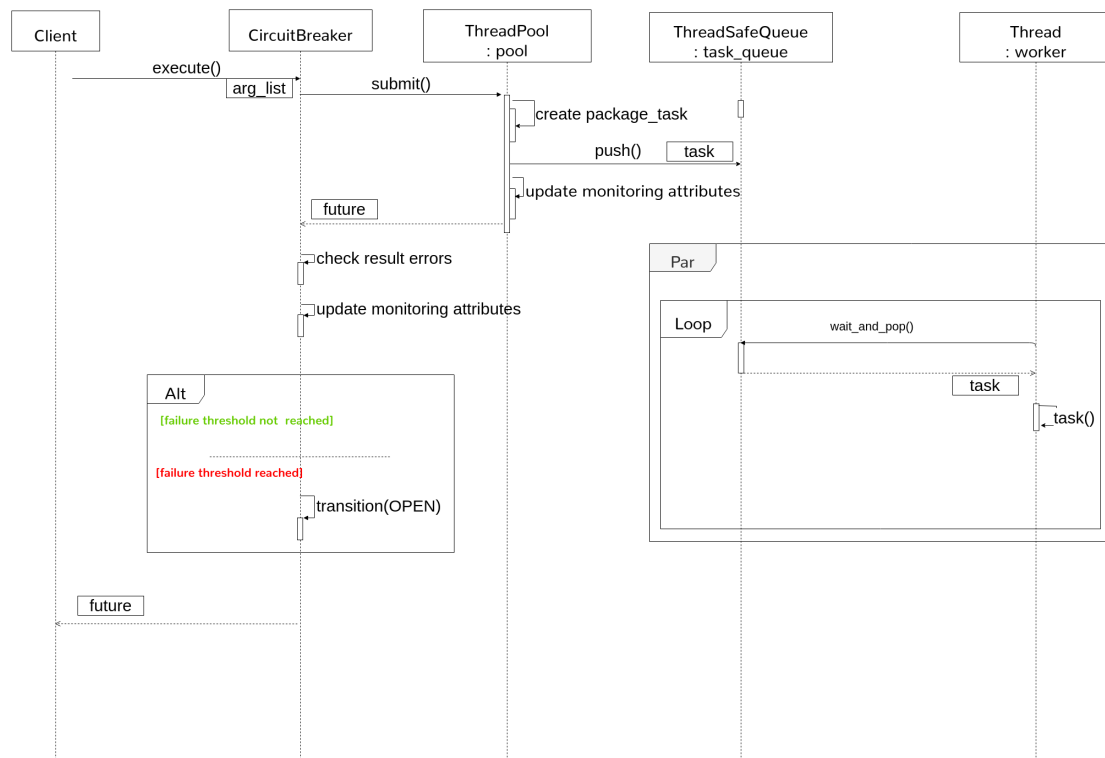


Figure 2.11: Sequence diagram : Circuit Breaker operating in closed state

### Scenario II : Open State

In this scenario in Figure 2.12 the client sends a request to the circuit breaker. The circuit breaker checks if it is already the time to transit to the half open state. Since the circuit breaker is open, the request is not submitted to the thread pool. Worker threads continue working in parallel if there are task in the task queue. The circuit breaker updates it local attributes and on timeout transits from the open to the half open state. The client regains controls and continue his task.

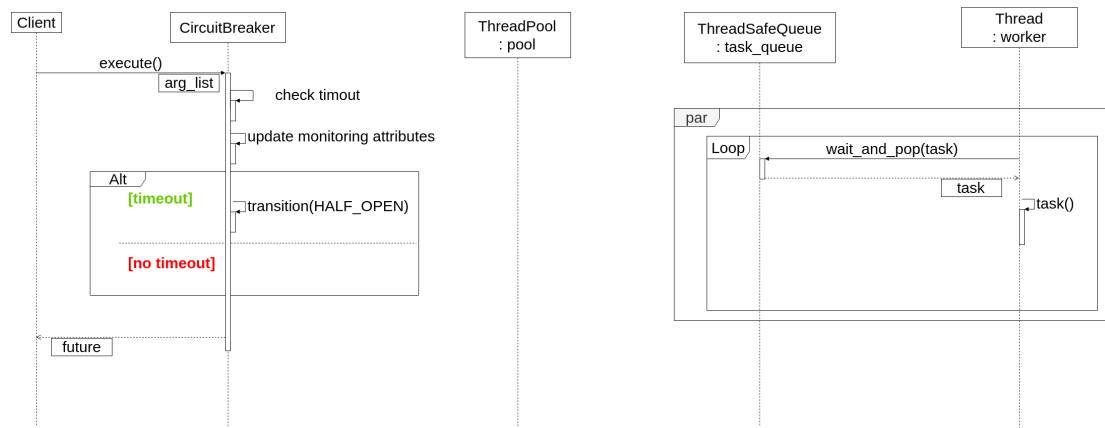


Figure 2.12: Sequence diagram : Circuit Breaker operating in open state

### Scenario III : Half Open State

In the scenario described in Figure 2.13 The client sends a request to the circuit breaker. The circuit breaker accepts the request and submits it to the thread pool. The thread pool creates a new task and pushes it in its task queue. The task will then be run in parallel by a worker thread. The thread pool updates its local attributes and return a future object back to the circuit breaker. The returned future is then checked by the circuit breaker to detect failure or success. In case of failure the circuit breaker transits from the half open to the open state. In case of success the circuit breaker transits from the half open to the closed state.

## 2 Circuit Breaker

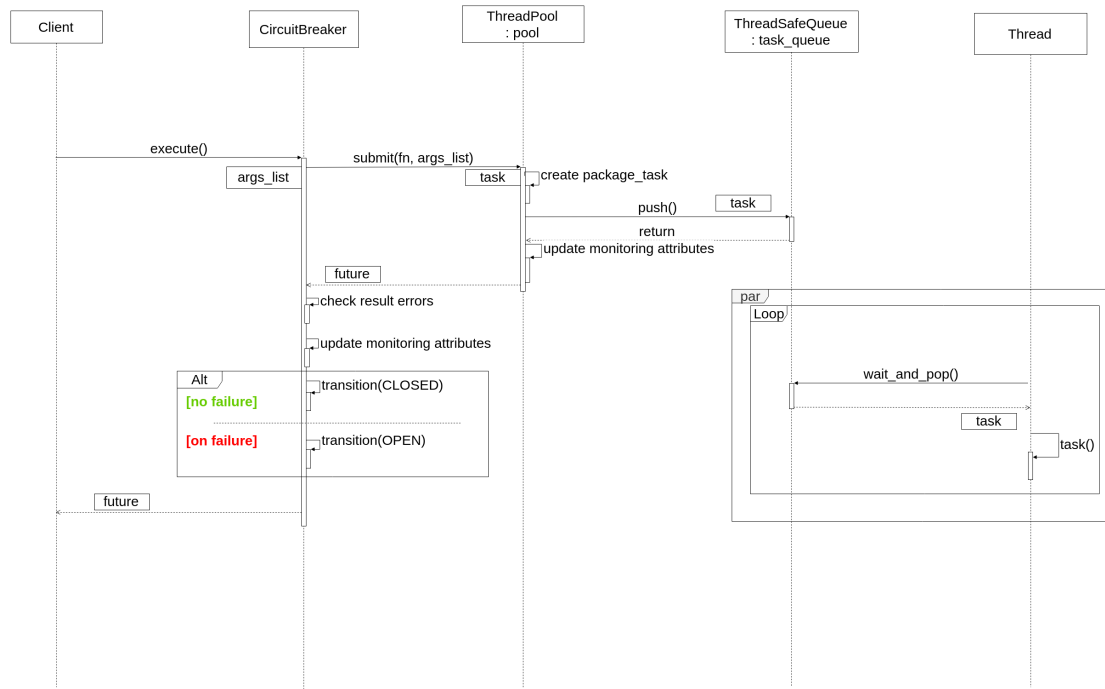


Figure 2.13: Sequence diagram : Circuit Breaker operating in half open state

## 2.7 Implementation with C++ 17

In the following find how the circuit breaker pattern could be implemented in the C++ programming language. A Logger module is implemented to allow logging capability.

### 2.7.1 Example of how the circuit breaker can be used

Listing 2.1 shows how the circuit breaker class can be used to send a request to the service. In this example, a Command object is first created and is given a service that will be used to execute the request. The Command class is a custom wrapper developed to replace the `std::function` utility that is provided the C++ standard library. The Command class is developed to encapsulate a callable object (functor, function) and make the callable movable. The C++'s `std::function` is not a good candidate because it is not movable and this implementation required the task object to be movable. The Command is to be used in place where a `std::function` could be used. A thread pool instance is then created. The thread pool is created with a number of threads set to `WORKERS` which is a macro set to 4 by default. The C++ standard doesn't provide a thread pool for threads management. For this reason a thread pool is implemented for this thesis. The circuit breaker instance is created and the command instance is passed to its constructor. The **DEADLINE**, **RETRY\_TIME** and **FAILURE\_THRESHOLD** are macros defined by default to **50**, **70** and **3**. The thread pool is passed to the circuit breaker and it is now ready to operate.

Listing 2.1: Circuit Breaker usage demonstration

```
1 #include "threadpool.h"
2 #include "circuitbreaker.h"
3 #include <log.h> // < -- Logger library
4 #include <iostream>
5 #include <memory> // <-- Smart pointers
6
7 #define WORKERS 4
```

```
8 #define FAILURE_THRESHOLD 3
9 #define DEADLINE 50
10 #define RETRY_TIME 70
11 using namespace std;
12
13 using duration_ms_t = std::chrono::milliseconds;
14 int job(int a, int b){
15     std::this_thread::sleep_for(duration_ms_t(70));
16     if(b > DEADLINE)
17         throw std::logic_error("b is greater than the deadline");
18     return a + b;
19 }
20
21 int main(int argc, char const *argv[])
22 {
23     Command service {job}; // Command is a movable Functor class
24                             // that wraps any callable object.
25     std::shared_ptr<ThreadPool> pool = std::make_shared<
26         ThreadPool>(WORKERS);
27     CircuitBreaker cb(service, duration_ms_t(DEADLINE),
28         duration_ms_t(RETRY_TIME),FAILURE_THRESHOLD);
29     cb.setPool(pool);
30     std::future<int> fut_res;
31     int res;
32     /*
33     * calling the same service through the circuit breaker
34     */
35     try {
36         fut_res = cb.execute(40, 2);
37         res = fut_res.get();
38         LOG("Circuit Usage Request Result : ", res);
39     }
40     // The service sleep for 70ms which is longer than the
41     // deadline which is 50ms
42     // so the request a TimeoutError is thrown by the circuit
43     // breaker.
44     catch(std::exception &e){
45         LOG_ERROR("Circuit Breaker : service failed with the
46             error :", e.what());
```

```
41     }
42     ...
43     // example a calling the service directly without the cicuit
         breaker
44     fut_res = pool->submit(job,40, 2); // We have to wait until
         the result is ready.
45     try{
46         res = fut_res.get();
47         LOG("Circuit Usage Request Result : ", res);
48     }
49     catch(std::exception &e){
50         LOG_ERROR("Service : service failed with the error :", e
                    .what());
51     }
52 }
```

CircuitBreaker::execute() returns a std::future object and this object might store an exception if an error occurred by the execution of the request. We must enclose the call to std::future::get() in a try-catch block. The result is logged using a logger developed for this thesis.

For comparison, on the same Listing 2.1 the same request is directly sent to service without using the circuit breaker.

### 2.7.2 Command Class

The implementation of the Command class is shown in Listing 2.2. As already described above, the Command class is wrapper around a callable object. This class is needed to make a callable object movable. The Command stores the callable and delegates the execution to the callable instance when the Command is called.



Listing 2.2: Command class: Class

```
1  template<typename R, typename...Args>
2  class Command{
3  public:
4      using result_type = std::decay_t<R>;
5      using function_type = R (*)(Args...);
6
7      Command(function_type op) : fn{op}{
8
9      }
10
11     Command(const Command &other){
12         this->fn = other.fn;
13     }
14
15     Command(Command&& other){
16         this->fn = other.fn;
17     }
18
19     Command &operator=(const Command& other){
20         this->fn = other.fn;
21     }
22
23     Command &operator=(Command&& other){
24         this->fn = other.fn;
25     }
26
27     result_type operator()(Args&&... args){
28         return std::invoke(std::forward<function_type>(fn), std
29             ::forward<Args>(args)...);
30     }
31 private:
32     function_type fn;
33
34 };
```

### 2.7.3 Logger Class

The C++ standard library doesn't provide a logger tools. To avoid external dependency to library like Boost<sup>7</sup>, a Logger tools is implemented to be able to create log data. The Logger provides three levels of severity which are DEBUG, ERROR and WARNING. DEBUG is the default severity level. The Logger can write the log data to the standard output or to a file. By default the log data is written to the standard output as seen in the Listing 2.3

Listing 2.3: Logger : log.h

```
1  #ifndef LOG_H
2  #define LOG_H
3  #include "logger.h"
4
5      #ifdef LOG_TO_FILE
6
7          static Loggin::Logger<LogFilePolicy> log_instance ("logger.
8              log");
9
10         #define LOG log_instance.print<Severity_Type::DEBUG>
11         #define LOG_WARN log_instance.print<Severity_Type::WARNING>
12         #define LOG_ERROR log_instance.print<Severity_Type::ERROR>
13
14         #else
15         static Loggin::Logger<LogConsolPolicy> log_instance ("
16             Console Logger");
17         #define LOG log_instance.print<Severity_Type::DEBUG>
18         #define LOG_WARN log_instance.print<Severity_Type::WARNING>
19         #define LOG_ERROR log_instance.print<Severity_Type::ERROR>
20
21         #endif
22 #endif // LOG_H
```

The Logger class is thread safe and can be used concurrently. It is divided into two classes which are a Logger class that implements the core functionality and

---

<sup>7</sup>Boost is a set of libraries for the C++ programming language <https://www.boost.org/>

a LogPolicy Class that writes the logs to a defined location. A LogFilePolicy is provided to write logs data into a file. A LogConsolePolicy is provided to write logs data into the console. Further details of the implementations of those classes can be found in the appendix.

### 2.7.4 Circuit Breaker Class implementation

One requirement derived from the requirements analysis is that the circuit breaker must accept any task type. To achieve this requirement C++ templates are intensively used.

Listing 2.4: Circuit Breaker : Class template definition

```
1  template<typename R,typename duration_t, typename... Args>
2  class CircuitBreaker
3  {
4
5  private:
6      Command<R, Args...> task;
7      /**
8       * @brief time_to_retry time to wait before transitioning
9       *         from OPEN state to HALF_OPEN state
10     */
11     duration_t time_to_retry;
12     /**
13     * @brief deadline time to wait for the service's reply
14     */
15     duration_t deadline;
16     ... // other attributes
17 public:
18     explicit CircuitBreaker(Command<R, Args...> task, duration_t
19         deadline, duration_t time_to_retry, int
20         failure_threshold):
21         task{task} ,time_to_retry{time_to_retry}, deadline{
22             deadline}, failure_threshold{failure_threshold}
23     {
24         ... // attribute initialization
```

```
22     state = State::CLOSED;
23     ... // other initializations ...
24 }
25 };
```

The implementation makes usage of C++ template type deduction to be able to work with any task signature. In Listing 2.4 the template parameter `R` captures the returned type of the task. This parameter will be automatically deduced by the compiler. The `duration_t` template parameter captures the type used to store the deadline attribute which is a `std::chrono::duration`. The last template parameter is a parameter pack, which represents a type list. The parameter packs represents the list of type arguments accepted by the task object. Type deductions are used by the circuit breaker class members to forward the task parameters and to return the result back to the caller.

### 2.7.5 Sending the request to the service

Listing 2.5 shows the state machine is used to determine the action to run. Sending a request to the service is made through the `execute(...)` member method. This method accepts the parameters required by the task object. Depending on the current circuit breaker state, the parameters are passed the to the method corresponding to the active state. A `std::future` object is returned back to the caller. The return type is automatically deduced with the help of the template system.

Listing 2.5: Circuit Breaker : Sending the request to the service

```
1     std::future<R> execute(Args&&... args){
2         ++usage;
3         std::future<R> res;
4         if(state == State::CLOSED){
5             res = onClosedState(std::forward<Args>(args)...);
6         }
7         else if(state == State::OPEN){
8             res = onOpenState(std::forward<Args>(args)...);
9         }
10        else if(state == State::HALF_OPEN){
```

```
11         res = onHalfOpenState(std::forward<Args>(args)...);
12     }
13     updateRatio();
14     return res;
15 }
```

When `execute()` is called, the request is routed depending on the current state.

### 2.7.6 sending request on closed state

Listing 2.6 shows the actions taken by the circuit breaker when it receives a request. The implementation uses the `std::future` and `std::promise` to handle the result of the task. The task is submitted in the thread pool with the arguments needed by the task. The thread pool returns a `std::future` which will contain the result when the task is executed. The circuit breaker then waits for the result to be ready in the time limit set by the deadline by calling `async_result.wait_for(deadline)`. The method `wait_for()` returns a `std::future_status` which is an enum type. This status result is used to detect whether the result is ready or if there is a timeout meaning that the result was not ready in the deadline defined. Depending on the status the result is stored in the future object through the `std::promise` variable. In case of timeout a `TimeoutError` exception is stored on the future handle to notify the caller about the failure. When the task has thrown an exception, this exception is stored in the `std::future` and the result is handled back to the caller. On failure monitoring attributes are updated. When the failure threshold is reached, the circuit breaker trips and makes a transition from the closed state to the open state.

Listing 2.6: Circuit Breaker : Sending the request to the service on closed state

```
1     std::future<R> onClosedState(Args&&... args){
2         std::promise<R> result_not_ready;
3
4         std::future<R> result = result_not_ready.get_future();
5         std::future<R> async_result = pool->submit(task, std::
            forward<Args>(args)...);
```

```
6         std::future_status status = async_result.wait_for(
           deadline);
7         if( status == std::future_status::ready){
8             try {
9                 result_not_ready.set_value(async_result.get());
10                reset();
11            } catch (const ServiceError &e) {
12                result_not_ready.set_exception(std::
13                    make_exception_ptr(e));
14                failure_count();
15                if(getFailure_counter() >= getFailure_threshold
16                    ()){
17                    ++failure_threshold_reached;
18                    trip();
19                }
20            }
21        } else if(status == std::future_status::timeout){
22            result_not_ready.set_exception(std::
23                make_exception_ptr(TimeoutError()));
24            failure_count();
25            if(getFailure_counter() >= getFailure_threshold()){
26                ++failure_threshold_reached;
27                trip();
28            }
29        }
30        return result;
31    }
```

### 2.7.7 sending request on open state

On open state the circuit breaker measures the elapsed time since the time it had changed to open. The elapsed time is compared to the retry time. When the time passed in the open state is greater than the retry time the circuit breaker makes a transition from the open state to the half open state. In any case, the circuit breaker creates a failures responses and handles it back to the caller by storing a

custom `ServiceError` exception in a `std::future` object and returns immediately (fails fast). The stored exception is used to inform the caller about the failure.

Listing 2.7: Circuit Breaker : Sending the request on open state

```
1  auto tmp = std::chrono::system_clock::now();
2      std::promise<R> error;
3      updateFailures();
4      auto elapsed_time_duration =std::chrono::duration_cast<
5          duration_t>(tmp - getFailure_time());
6      if( elapsed_time_duration >= getTime_to_retry()){
7          transition(State::HALF_OPEN);
8      }
9      error.set_exception(std::make_exception_ptr(ServiceError
10         ("SYSTEM NOW")));
11     return error.get_future();
```

### 2.7.8 Sending request on half open state

In the half open state, the circuit breaker sends the request to service and sees if the request succeed. If the service failed the circuit breaker makes a transition from the half open to the open state. In case of success the circuit updates it monitoring attributes and makes a transition from the half open state to the closed state.

Listing 2.8: Circuit Breaker : Sending the request on half-open state

```
1  std::future<R> onHalfOpenState(Args&&... args){
2      std::promise<R> result_not_ready;
3      std::future<R> result = result_not_ready.get_future();
4      std::future<R> async_result = pool->submit(task, std::
5          forward<Args>(args)...);
6      std::future_status status = async_result.wait_for(
7          deadline);
8      if( status == std::future_status::ready){
9          try {
10             result_not_ready.set_value(async_result.get());
11             reset();
12         } catch (const ServiceError &e) {
```

```
11         result_not_ready.set_exception(std::
           make_exception_ptr(e));
12         updateFailures();
13         trip();
14     }
15 }
16 else if(status == std::future_status::timeout){
17     result_not_ready.set_exception(std::
           make_exception_ptr(TimeoutError()));
18     updateFailures();
19     trip();
20 }
21 return result;
22 }
```

## 2.7.9 Transition and Monitoring

On state transition, the circuit breaker first checks to see if the new state is different from the current one. If the new state is different from the old one it changes its state to the new state.. After changing its state the circuit breaker notifies the registered observers about the change. Monitors or supervisor can register callable object which will be executed when the circuit breaker changes whether to the open or the closed state. This method is executed on the caller thread. The observers must be short so that they don't monopolize the processor.

Listing 2.9: Circuit Breaker : Sending the request on half-open state

```
1 void transition(State state){
2     if(this->state != state){
3         this->state = state;
4         if(state == State::OPEN){
5             std::for_each(open_observers.begin(),
                           open_observers.end(), [](FunctionWrapper &
                           observer){
6                 observer();
7             });
8         }else if(state == State::CLOSED){
```



```
9         std::for_each(closed_observers.begin(),
10                       closed_observers.end(), [](FunctionWrapper &
11                                                   observer){
12             observer();
13         });
14     }
```

One interesting aspect in the code above is the presence of observers. Those observers are declared in the circuit breaker class as seen in the Listing 2.10 below. The observers are actually callbacks functions. To be more resilient, a supervisor (not implemented in this work) is needed to take action in case the circuit breaker becomes open. An observer could be a function that informs the supervisor about the situation so that he could start a recovery by replicating the service component. These observers can be used to supervise the service component depending on the parameters of interest.

Listing 2.10: Circuit Breaker : Supervisors

```
1     template<typename R,typename duration_t, typename... Args>
2     class CircuitBreaker
3     {
4
5     private:
6         ...
7         /**
8          * @brief closed_observers this is the list of observer (
9            callback) to be call when the circuit
10           * becomes closed
11          */
12         std::vector<FunctionWrapper> closed_observers;
13         /**
14          * @brief open_observers this is the list of observer (
15            callback) to be call when the circuit
16           * becomes open.
17          */
18         std::vector<FunctionWrapper> open_observers;
19         /**
20          * @brief listeners this is the list of observer (callback)
21            to be call when the circuit
22           * changes it state.
23          */
24         std::vector<FunctionWrapper> listeners;
25         ...
26     };
```

The implementation provides some attribute that could be used to monitor the service component. Listing 2.11 shows the member attributes that could be used to monitor the service component performance. The success ratio attribute can be used to evaluate how the service performs since it creation. A low success ratio could be an indication that the service component doesn't work and should be replaced.

Listing 2.11: Circuit Breaker : Monitoring variable

```
1     template<typename R,typename duration_t, typename... Args>
2     class CircuitBreaker
3     {
4
5     private:
6         ...
7         /**
8          * @brief success_ratio represents the success ratio until
9          * now.
10          * It holds a percentage value
11          */
12         double success_ratio;
13         /**
14          * @brief ratio_trip this variable represent ratio at which
15          * the circuit trips.
16          * It holds a percentage value
17          */
18         double ratio_trip;
19         /**
20          * @brief failure_threshold_reached this variable represents
21          * the number of time
22          * the circuit breaker has tripped.
23          */
24         int failure_threshold_reached;
25         /**
26          * @brief usage the number of call made through this circuit
27          * breaker instance.
28          * this helps when you want to monitor the circuit breaker
29          */
30         int usage;
31         /**
32          * @brief failures total failures count since the programme
33          * started
34          */
35         int failures;
```

```
34     * @brief successes number of successfully calls
35     */
36     int successes;
37     ...
38 };
```

There are many **Replications Patterns**<sup>8</sup> which could be applied here to replicate the service. Those patterns are :

- Active - Passive Replication Pattern
- Active - Active Replication Pattern
- Multiple-Master Replication Pattern

Maybe the reason the service is down is because it is being overloaded. One listener could inform the supervisor about the situation and so this one could add more resources (threads, thread pool ...) to handle the incoming flow of requests, which will make the system **elastic**. One other observation is that a **Rate Limiter**<sup>9</sup>(not implemented in this work) can be used too to prevent a too fast client for overwhelming the service. This will help the system to stay **responsive** when the system is overloaded.

### 2.7.10 Thread Pool

The thread pool pattern is implemented as the solution to the requirement which stated that the Service invocation call must be decoupled from the service execution. To process requests and stay responsive, the circuit breaker uses a thread pool. The thread pool accepts any type of task. C++ templates are used to be able to accept any task type and makes the thread pool reusable since it is task independent. To submit tasks the thread pool has a method **submit()** which takes

---

<sup>8</sup>Roland Kuhn, Brian Hanafée, and Jamie Allen. “Replication Patterns”. In: *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. Chap. 13, p. 184. ISBN: 9781617291807.

<sup>9</sup>Roland Kuhn, Brian Hanafée, and Jamie Allen. “Fault tolerance and recovery patterns”. In: *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. Chap. 12, p. 179. ISBN: 9781617291807.

a callable object and the list of accepted parameters for this callable. The signature of the task is deduced from the task instance. In Listing 2.12 the Callable template parameter represents the callable object type and ...Args is a template parameter pack that represents the list of argument , in the order, accepted by Callable. `std::invoke_result_t` (from C++ 17) is used to deduce the return type of the a Callable instance when called with the arguments of the type contains in the parameter pack ...Args. These information are used to build a `std::package_task` that will be run in a Worker Thread. The task is then moved in the task queue. The move semantic is used to avoid to make a copy which could be expensive in the case where the Callable is an heavy functor object. `std::forward` are used for perfect forwarding. Perfect forwarding allow the Callable instance to receive the correct type properties. That means, if the Callable is taking a reference as parameter then a reference will be forwarded to the task. If it need a const then a const will be forwarded to the task instance. This improves the overall performance as it will be seen in the tests. `std::future` are used so that the caller doesn't need to block after a task has been submitted.

Listing 2.12: ThreadPool Class implementation

```
1  class ThreadPool{
2  private:
3      /**
4       * @brief done flag indicating that a thread should
5         terminate and quit.
6       */
7      std::atomic_bool done;
8
9      /**
10     * @brief workers_count current number of available threads
11       used by this thread pool.
12     */
13     size_t workers_count;
14
15     /**
16     * @brief task_queue contains the tasks to be run.
17     */
18     ThreadSafeQueue<FunctionWrapper> task_queue;
```

```
17     * @brief threads list of actual threads in use.
18     */
19     std::vector<std::thread> threads;
20     ...
21
22     public:
23     /**
24     * @brief submit place a new task in the task queue. This
25     * new task will
26     * be processed by a free thread.
27     * When there is no free thread, the task stays in the task
28     * queue until a thread
29     * becomes available to run it.
30     * this threadpool can process any any task with any
31     * signature.
32     */
33     template<typename Callable, typename... Args,
34             typename = std::enable_if_t<std::
35             is_move_constructible_v<Callable>>>
36     std::future<std::invoke_result_t<std::decay_t<Callable>,
37     std::decay_t<Args>...>>
38     submit(Callable &&op, Args&&... args){
39     using result_type =std::invoke_result_t<std::decay_t<
40     Callable>, std::decay_t<Args>...>;
41     std::packaged_task<result_type()> task(std::bind(std::
42     forward<Callable>(op), std::forward<Args>(args)...))
43     ;
44     std::future<result_type> result(task.get_future());
45     task_queue.push(std::move(task));
46     submitted.fetch_add(1, std::memory_order_relaxed);
47     return result;
48     }
49     ...
```

### 2.7.11 Interrupting the Thread in the Thread Pool

It is possible to interrupt the worker threads. Interrupting a thread will not cancel the already submitted task. The worker threads will first run all the tasks which

was submitted until the interrupt request is sent. Listing 2.13 shows how we could interrupt the threads present in the thread pool. It puts a task in the task queue which will set the flag `done` to `true`. The current thread pool implementation doesn't provide a mean to interrupt threads individually.

Listing 2.13: Thread Pool : Thread cancelling

```
1 class ThreadPool{
2 private:
3     ...
4
5 public:
6     /    /**
7         * @brief interrupt interrupts all currently created threads
8         */
9     void interrupt(){
10         std::for_each(threads.begin(), threads.end(), [&](std::
11             thread &t){
12                 if(t.joinable()){
13                     submit([&]{
14                         done = true;
15                     });
16                 }
17             });
18     ...
```

### 2.7.12 Thread Safe Queue Class

The Thread Pool class is using a queue to store the submitted tasks. The C++ standard doesn't provide a thread safe queue. A thread safe queue is implemented to allow clients to concurrently add tasks into the thread pool. Listing 2.14 shows the implementation of the thread safe queue used in this thesis. The new C++ move semantic is used to insert or remove elements into the queue.

Listing 2.14: ThreadSafeQueue Class implementation

```
1  template <typename T>
2      class ThreadSafeQueue
3      {
4      public:
5          ThreadSafeQueue ()
6          {
7
8          }
9
10         ThreadSafeQueue( ThreadSafeQueue const& other)
11         {
12             std::lock_guard<std::mutex>locker(other.itemMutex);
13             items = other.items;
14         }
15
16         ThreadSafeQueue(ThreadSafeQueue&& other)
17         {
18             std::lock_guard<std::mutex>locker(other.itemMutex);
19             items = std::move(other.items);
20             other.clear();
21         }
22
23         ThreadSafeQueue& operator =(ThreadSafeQueue&& other)
24         {
25             std::lock(itemMutex, other.itemMutex);
26             std::lock_guard<std::mutex> this_lock(itemMutex, std
                ::adopt_lock);
27             std::lock_guard<std::mutex> other_lock(other.
                itemMutex, std::adopt_lock);
28             items = std::move(other.items);
29             other.clear();
30             return *this;
31         }
32     }
33
34     ThreadSafeQueue& operator =(ThreadSafeQueue& other)
35     {
36         std::lock(itemMutex, other.itemMutex);
```



```
37         std::lock_guard<std::mutex> this_lock(itemMutex, std
           ::adopt_lock);
38         std::lock_guard<std::mutex> other_lock(other.
           itemMutex, std::adopt_lock);
39         items = std::move(other.items);
40         return *this;
41
42     }
43
44     /**
45      * @brief push pushes item into the queue
46      * @param item the element to be pushed into the queue
47      */
48     void push(T item)
49     {
50         std::lock_guard<std::mutex> locker(itemMutex);
51         items.push(std::move(item));
52         itemCond.notify_one();
53     }
54
55     /**
56      * @brief wait_and_pop takes the last insert item from
           the queue.
57      *
58      * @param item contains the removed element
59      */
60     void wait_and_pop(T& item)
61     {
62         std::unique_lock<std::mutex> locker(itemMutex);
63         itemCond.wait(locker, [this]{return !items.empty();})
           ;
64         item = std::move(items.front());
65         items.pop();
66     }
67
68     /**
69      * @brief wait_and_pop takes the last insert item from
           the queue.
70      * @return handle to the removed element
```

```
71     */
72     std::shared_ptr<T> wait_and_pop()
73     {
74         std::unique_lock<std::mutex> locker(itemMutex);
75         itemCond.wait(locker, [this]{return !items.empty();})
76         ;
77         std::shared_ptr<T> result (std::make_shared<T> (std
78             ::move(items.front())));
79         items.pop();
80         return result;
81     }
82
83     /**
84     * @brief try_pop non blocking removing method. this
85     * method try to remove
86     * the last inserted element from the queue.
87     * @param item contains the removed inserted element.
88     * @return true on success. False is returned when the
89     * queue is empty.
90     */
91     bool try_pop(T& item)
92     {
93         std::lock_guard<std::mutex> locker(itemMutex);
94         if(items.empty())
95             return false;
96         item = std::move(items.front());
97         items.pop();
98         return true;
99     }
100
101     /**
102     * @brief try_pop non blocking removing method. this
103     * method try to remove
104     * the last inserted element from the queue.
105     * @return handle to the removed element when the queue
106     * is not empty.
107     * returns an invalid pointer when the queue is empty.
108     */
```

```
104     std::shared_ptr<T> try_pop()
105     {
106         std::lock_guard<std::mutex> locker(itemMutex);
107         if(items.empty())
108             return std::shared_ptr<T>();
109         std::shared_ptr<T> result (std::make_shared<T> (std
            ::move(items.front())));
110         items.pop();
111         return result;
112     }
113     ...
114     ...
115
116     private:
117         std::queue<T> items;
118         mutable std::mutex itemMutex;
119         std::condition_variable itemCond;
120     };
```

## 2.8 Conclusion

The C++ 17 standard is used to implement the circuit breaker pattern. The circuit breaker implemented is task-independent so that it can be reused or extended. This is made possible with the use of C++ Templates and the new C++ 17 Class template type deduction. Basic monitoring facilities is implemented. Supervisors can be registered to be called when the circuit breaker state changes. The current implementation makes it easy to apply the Replication Patterns by installing a supervisors to be notified when the circuit is open. The Thread Pool pattern is implemented to make it possible to decouple method invocation from method execution.

# 3 Test Execution

## 3.1 Test Environment

The test will be run using a system with the following specifications :

Operating System : Ubuntu 18.04.1 LTS

Processor : Intel core i7-8550U 1.8GHz-4GHz 8th Generation

RAM : 12 GB

CPU Architecture : 64-bit Architecture

Desktop Environment : GNOME 3.28.2

Compiler : GCC 7.3.0 (Ubuntu 7.3.0-27ubuntu1 18.04)

Programming Language : C++ 17 Standard

Library dependency : libcurl4-openssl-dev 7.58, Boost 1.58++

Build System : CMake 3.13

**GCC 7.3.0** is default installed in Ubuntu 18.04 and offers complete C++ 17 support. **libcurl** is needed to send request to remote web-services.

## 3.2 Test Strategy

Two strategies are used to test the circuit breaker pattern implementation.

### 3.2.1 Simulation

In this strategy the service is put into sleep with randomly chosen delay on every request. The client sends consecutively a varying number of request in the series of 1 - 5 - 10 - 100 - 500 - 1000 - 10000 - 1000000. The same series of request is also sent without using the circuit breaker. A pause of 500ms is made between each series of requests.

### 3.2.2 Web service requests

In this strategy instead of a simulation, requests are sent to an independent web site. The website URL is [www.example.com](http://www.example.com) which offers a public API. For this test only the series of request from 1 to 100 requests will be used since the website response time degrades itself the more requests are sent. The deadline used for this test is defined to a value which corresponds to the mean response time after a group of 30 requests at different time interval. The mean value of this observation leads to 476ms which is set as the deadline to use in the circuit breaker.

The aims of these tests is to observe the latency of the requests depending on the value chosen for the three parameters deadline, failure threshold and time to retry. This will help us to see the impact of those parameters on the circuit breaker pattern.

## 3.3 Test Execution

This chapter describes the steps to follow in order to build and execute the simulation.

### 3.3.1 How to build the simulation

The build tools to compile the simulation is based around CMake. The simulation is making use of constant macros which must be defined at compile time. The constant macros help the simulation code to make a decision on the type of the simulation and the parameters to pass to the circuit breaker instance.

#### Parameter passed to the circuit breaker instance

The circuit breaker needs three parameters to be set so that it can start protecting the service component : deadline, failures threshold and retry time. To set the deadline, the simulation is using a percent of the maximal processing time needed by the service component. This time is passed as macro at compile time.

**PROCESSING\_DURATION=X** This value is used in the experiments to select different values for the deadline. The deadline is set as a percent of this value. X is an integer value.

**FAILURES\_THRESHOLD=X** This is the value to set the failures threshold the experiments. X is an integer value.

**RETRY\_TIME=X** This value is used to set the retry time to use for the experiments. X is an integer value.

**TIME\_UNIT\_MS=FLAG** This flag is used to select the time unit : milliseconds or microseconds. The possible values of FLAG are ON and OFF. Setting this flag to ON will set the time unit to milliseconds. Setting it to OFF will set the time unit to microseconds. This flag is set to OFF by default.

#### Parameter passed to the Thread Pool instance

The circuit breaker uses a thread pool to run submitted requests. You can control the number of worker threads used in the thread pool by setting the `MTHREADING` macro.

**MTHREADING=X** This value indicates the number of threads to use. X is an integer.

#### Parameters that select the type of the simulation

You have the possibility to run a simulation or to send requests to a real website with the macro `USE_REMOTE_SERVICE`. If you choose to use remote service, you can choose the URL to locate the remote web service by setting the `URL` macro.

**USE\_REMOTE\_SERVICE=FLAG** This flag is used to select the type of the simulation. The possible values are `ON` and `OFF`. Setting this flag to `ON` will use send the requests to real web site. You can select URL where the request will be sent by setting the `URL` macro. Setting the macro to `OFF` will send the requests to the simulation service. When `USE_REMOTE_SERVICE` is set to `OFF` the `URL` macro is not taken into account.

**URL=URL\_X** The `URL` macro is used to select the URL of the website to use when `USE_REMOTE_SERVICE=ON`. There are 3 different website to choose from where `URL_2` is the most stable and recommended. The possible values for `URL_X` are `URL_1`, `URL_2` and `URL_3`. By default `URL` is set to `URL_2`.

The websites with `URL_1` and `URL_3` offer an API where you can define the delay before the website sends a response to your request. You can configure the response the website should send back. There are the `URL_X` used the requests. You can change them to point to another URL.

- `URL_1` : <http://www.mocky.io/v2/5c405ffe0f00007408e7b3f9/?mocky-delay=500ms>

- URL\_2 : https://example.com
- URL\_3 : https://reqres.in/api/users

#### Displaying Debug output

When the circuit breaker instance is destructed, his destructor method can log the summary usage of this circuit breaker instance. You can control this behavior by setting the `DEBUG_ON` macro.

**DEBUG\_ON=FLAG** This macro controls whether or not the circuit breaker instance writes a summary usage through the logger. The possible values are `ON` and `OFF`. Setting this flag to `ON` will write the summary usage of the circuit breaker through the logger. Setting this flag to `OFF` will deactivate the logging of the summary usage. By default this flag is set to `ON`.

### 3.3.2 Compiling the simulation code

Building the code source is made using CMake. We need to pass in the parameters we have talked about on the previous section. When the experiments is run, the results are save in a file whose name is defined by most of these flags / macro. The results are saved in a directory named `log`. This directory is automatically created in the build directory where CMake is executed. An example of how the build files can be generated for an experiments that run a simulation is shown below:

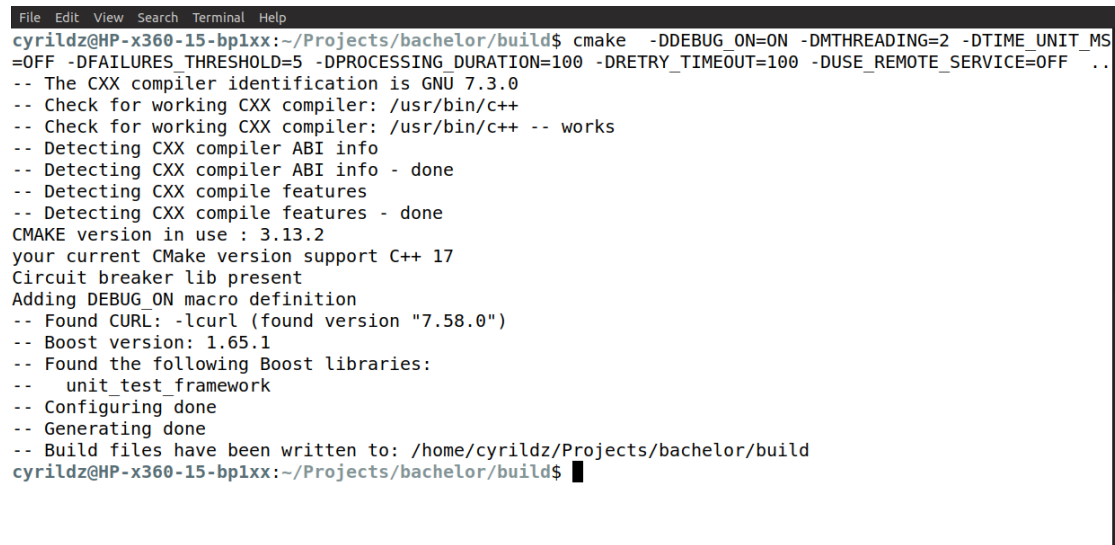
Listing 3.1: CMake command to generate build files for a simulation

```
1 #From Project root directory :
2 mkdir build
3 cd build
4 cmake -DDEBUG_ON=ON -DMTHREADING=2 -DTIME_UNIT_MS=OFF -
      DFAILURES_THRESHOLD=5 -DPROCESSING_DURATION=100 -
      DRETRY_TIMEOUT=100 -DUSE_REMOTE_SERVICE=OFF ..
5 make Experiments
6 ./experiments/Experiments
7 # results are written into experiments/log directory
```



## Generating Makefile

Running the **cmake** command from the code listed in Listing 3.1 will generate a Makefile that can be used by the make program to build the executable. The output of this command is shown in Figure 3.1.



```
File Edit View Search Terminal Help
cyrildz@HP-x360-15-bp1xx:~/Projects/bachelor/build$ cmake -DDEBUG_ON=ON -DMTHREADING=2 -DTIME_UNIT_MS
=OFF -DFAILURES_THRESHOLD=5 -DPROCESSING_DURATION=100 -DRETRY_TIMEOUT=100 -DUSE_REMOTE_SERVICE=OFF ..
-- The CXX compiler identification is GNU 7.3.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
CMAKE version in use : 3.13.2
your current CMake version support C++ 17
Circuit breaker lib present
Adding DEBUG_ON macro definition
-- Found CURL: -lcurl (found version "7.58.0")
-- Boost version: 1.65.1
-- Found the following Boost libraries:
--   unit_test_framework
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cyrildz/Projects/bachelor/build
cyrildz@HP-x360-15-bp1xx:~/Projects/bachelor/build$
```

Figure 3.1: Running CMake to generate build files

## Compiling the code

The **make** program is used to build the code source by using a Makefile. A Makefile contains the instructions needed to build the simulation code. Running the **cmake** command listed in Listing 3.1 has produced a Makefile in the build directory. The generated Makefile will be used by **make**. Running the **make** command from the Listing will compile the simulation code, as shown in Figure 3.2.

### 3 Test Execution

```
File Edit View Search Terminal Help
cyrildz@HP-x360-15-bp1xx:~/Projects/bachelor/build$ make Experiments
CMAKE version in use : 3.13.2
your current CMake version support C++ 17
Circuit breaker lib present
Adding DEBUG_ON macro definition
-- Boost version: 1.65.1
-- Found the following Boost libraries:
--   unit_test framework
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cyrildz/Projects/bachelor/build
Scanning dependencies of target logger
[ 16%] Building CXX object CMakeFiles/logger.dir/include/logger/logpolicy.cpp.o
[ 33%] Linking CXX static library liblogger.a
[ 33%] Built target logger
Scanning dependencies of target Experiments
[ 50%] Building CXX object experiments/CMakeFiles/Experiments.dir/service.cpp.o
[ 66%] Building CXX object experiments/CMakeFiles/Experiments.dir/utls.cpp.o
[ 83%] Building CXX object experiments/CMakeFiles/Experiments.dir/experiments.cpp.o
[100%] Linking CXX executable Experiments
[100%] Built target Experiments
cyrildz@HP-x360-15-bp1xx:~/Projects/bachelor/build$
```

Figure 3.2: Running CMake to generate build files

### Running the Simulation

The last command from the Listing 3.1 is used to run the simulation.

```
1 ./experiments/Experiments
```

Figure 3.3 shows an example of the output produced when the simulation is running.

```
File Edit View Search Terminal Help
cyrildz@HP-x360-15-bp1xx:~/Projects/bachelor/build$ ./experiments/Experiments
virtual void LogConsolePolicy::open_stream(const string&) opening Console Logger
virtual void LogConsolePolicy::open_stream(const string&) opening Console Logger
virtual void LogConsolePolicy::open_stream(const string&) opening Console Logger
0000000 < Tue Apr 9 16:13:31 2019 - 0010049 > <-DEBUG> : Reactor: Circuit Breaker
0000000 < Tue Apr 9 16:13:31 2019 - 0010076 > <-DEBUG> : TestRunner constructed ...
0000001 < Tue Apr 9 16:13:31 2019 - 0010386 > <-DEBUG> : Running Test started ...
-----
0000002 < Tue Apr 9 16:13:31 2019 - 0010410 > <-DEBUG> : Circuit Breaker test started with request = 10
0000003 < Tue Apr 9 16:13:31 2019 - 0011024 > <-DEBUG> : usage summary : success = 0; error = 10; usage : 10; deadline(us): 50 Success RATIO(%) : 0
number of trip : 2 Trip Ratio(%) : 100
0000004 < Tue Apr 9 16:13:31 2019 - 0011046 > <-DEBUG> : CB THREAD POOL usage summary : submitted tasks : 5 finished tasks : 5 still active : 0
0000005 < Tue Apr 9 16:13:31 2019 - 0011184 > <-DEBUG> : Circuit Breaker test started with request = 50
0000006 < Tue Apr 9 16:13:31 2019 - 0012382 > <-DEBUG> : usage summary : success = 4; error = 46; usage : 50; deadline(us): 50 Success RATIO(%) : 8
number of trip : 7 Trip Ratio(%) : 70
0000007 < Tue Apr 9 16:13:31 2019 - 0012407 > <-DEBUG> : CB THREAD POOL usage summary : submitted tasks : 21 finished tasks : 21 still active : 0
0000008 < Tue Apr 9 16:13:31 2019 - 0012434 > <-DEBUG> : Circuit Breaker test started with request = 100
0000009 < Tue Apr 9 16:13:31 2019 - 0014634 > <-DEBUG> : usage summary : success = 2; error = 98; usage : 100; deadline(us): 50 Success RATIO(%) : 2
number of trip : 17 Trip Ratio(%) : 85
0000010 < Tue Apr 9 16:13:31 2019 - 0014675 > <-DEBUG> : CB THREAD POOL usage summary : submitted tasks : 40 finished tasks : 40 still active : 0
0000011 < Tue Apr 9 16:13:31 2019 - 0014773 > <-DEBUG> : Circuit Breaker test started with request = 500
0000012 < Tue Apr 9 16:13:31 2019 - 0030448 > <-DEBUG> : usage summary : success = 15; error = 485; usage : 500; deadline(us): 50 Success RATIO(%) : 3
number of trip : 73 Trip Ratio(%) : 73
0000013 < Tue Apr 9 16:13:31 2019 - 0039523 > <-DEBUG> : CB THREAD POOL usage summary : submitted tasks : 141 finished tasks : 141 still active : 0
-----
0000014 < Tue Apr 9 16:13:31 2019 - 0039618 > <-DEBUG> : Service test started with request = 10
0000015 < Tue Apr 9 16:13:31 2019 - 0032553 > <-DEBUG> : Service test started with request = 50
0000016 < Tue Apr 9 16:13:31 2019 - 0040489 > <-DEBUG> : Service test started with request = 100
0000017 < Tue Apr 9 16:13:31 2019 - 0054396 > <-DEBUG> : Service test started with request = 500
0000018 < Tue Apr 9 16:13:31 2019 - 0100603 > <-DEBUG> : Running Test finished
-----
0000019 < Tue Apr 9 16:13:31 2019 - 0100695 > <-DEBUG> : Circuit Breaker test started with request = 10
0000020 < Tue Apr 9 16:13:31 2019 - 0101670 > <-DEBUG> : usage summary : success = 1; error = 9; usage : 10; deadline(us): 75 Success RATIO(%) : 10
Number of trip : 1 Trip Ratio(%) : 50
```

Figure 3.3: Running the simulation

If you run this code, the values shown here may vary from yours since these depend on the host machine's characteristics.

After execution there will be two log files created in the log directory. One file contains the results of the simulation run with the circuit breaker and the other file contains the results of the simulation run without the circuit breaker. Figure 3.4 shows an example of the output produced when the simulation has come to end. File's information such as name and location are displayed at the end of the simulation.

```

File Edit View Search Terminal Help
0000054 < Tue Apr 9 16:13:31 2019 - 0282324 > -<DEBUG> : Circuit Breaker test started with request = 500
0000055 < Tue Apr 9 16:13:31 2019 - 0371422 > -<DEBUG> : usage summary : success = 484; error = 16; usage : 500; deadline(us): 150 Success RATIO(%):96
.8 Number of trip :0 Trip Ratio(%):0
0000056 < Tue Apr 9 16:13:31 2019 - 0371488 > -<DEBUG> : CB THREAD POOL usage summary : submitted tasks : 2639 finished tasks : 2639 still active : 0
-----
0000057 < Tue Apr 9 16:13:31 2019 - 0371575 > -<DEBUG> : Running Test finished
-----
0000058 < Tue Apr 9 16:13:31 2019 - 0371646 > -<DEBUG> : Circuit Breaker test started with request = 10
0000059 < Tue Apr 9 16:13:31 2019 - 0373594 > -<DEBUG> : usage summary : success = 10; error = 0; usage : 10; deadline(us): 200 Success RATIO(%):10
0 Number of trip :0 Trip Ratio(%):0
0000060 < Tue Apr 9 16:13:31 2019 - 0373646 > -<DEBUG> : CB THREAD POOL usage summary : submitted tasks : 2649 finished tasks : 2649 still active : 0
0000061 < Tue Apr 9 16:13:31 2019 - 0373721 > -<DEBUG> : Circuit Breaker test started with request = 50
0000062 < Tue Apr 9 16:13:31 2019 - 0383104 > -<DEBUG> : usage summary : success = 50; error = 0; usage : 50; deadline(us): 200 Success RATIO(%):10
0 Number of trip :0 Trip Ratio(%):0
0000063 < Tue Apr 9 16:13:31 2019 - 0383163 > -<DEBUG> : CB THREAD POOL usage summary : submitted tasks : 2699 finished tasks : 2699 still active : 0
0000064 < Tue Apr 9 16:13:31 2019 - 0383240 > -<DEBUG> : Circuit Breaker test started with request = 100
0000065 < Tue Apr 9 16:13:31 2019 - 0401698 > -<DEBUG> : usage summary : success = 100; error = 0; usage : 100; deadline(us): 200 Success RATIO(%):10
0 Number of trip :0 Trip Ratio(%):0
0000066 < Tue Apr 9 16:13:31 2019 - 0401802 > -<DEBUG> : CB THREAD POOL usage summary : submitted tasks : 2799 finished tasks : 2799 still active : 0
0000067 < Tue Apr 9 16:13:31 2019 - 0401916 > -<DEBUG> : Circuit Breaker test started with request = 500
0000068 < Tue Apr 9 16:13:31 2019 - 0493288 > -<DEBUG> : usage summary : success = 500; error = 0; usage : 500; deadline(us): 200 Success RATIO(%):10
0 Number of trip :0 Trip Ratio(%):0
0000069 < Tue Apr 9 16:13:31 2019 - 0493356 > -<DEBUG> : CB THREAD POOL usage summary : submitted tasks : 3299 finished tasks : 3299 still active : 0
-----
0000070 < Tue Apr 9 16:13:31 2019 - 0493446 > -<DEBUG> : Running Test finished
-----
0000071 < Tue Apr 9 16:13:31 2019 - 0493649 > -<DEBUG> : Saving test results into /home/cyrlidz/Projects/bachelor/build/experiments/log/Circuitbreaker_5_100_100_us_sim_T2
.txt
0000072 < Tue Apr 9 16:13:31 2019 - 0493961 > -<DEBUG> : Saving result finished ...
0000073 < Tue Apr 9 16:13:31 2019 - 0494065 > -<DEBUG> : Saving Service test results into /home/cyrlidz/Projects/bachelor/build/experiments/log/SERVICE_5_100_100_us_sim_T
2.txt
0000074 < Tue Apr 9 16:13:31 2019 - 0494180 > -<DEBUG> : Saving result finished ...
0000075 < Tue Apr 9 16:13:31 2019 - 0494217 > -<DEBUG> : TestRunner destructed ...
cyrlidz@HP-x360-15-bp1xx:~/Projects/bachelor/build$

```

Figure 3.4: Results of the simulation

The log file's name produced is based on the parameters used to run the simulation. In Figure 3.4, the file name `Circuitbreaker_5_100_100_us_sim_T2.txt` can be divided into 7 parts. Each part of this name has a meaning. Going from left to right, here is how the parts are interpreted as **<prefix>\_<failure threshold>\_<retry time>\_<maximal processing time>\_<time unit>\_<simulation type>\_<number of threads used>.txt**

**prefix** The prefix indicated whether the experiment has used the circuit breaker or not. The other possible value are `Circuitbreaker` and `Service`.

**failure threshold** This indicates the value used for the failure threshold parameter.

### 3 Test Execution

**retry time** This indicates the value used for the retry time parameter.

**maximal processing time** This part indicates the value used for the maximal processing time.

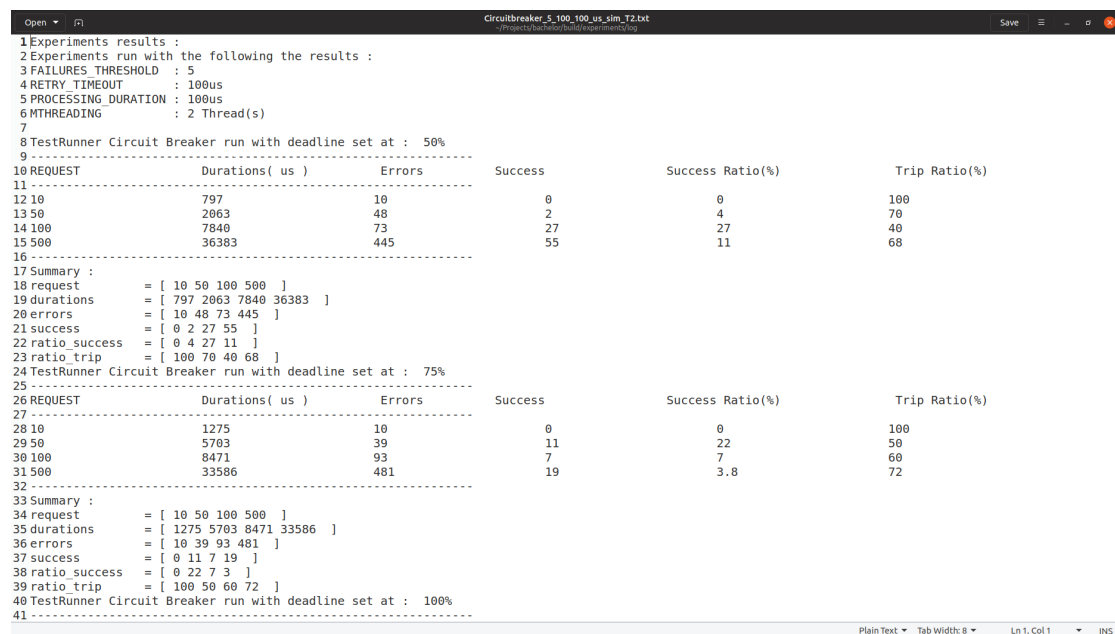
**time unit** This part indicates whether the time unit used was microseconds ( $\mu\text{s}$ ) or milliseconds (ms).

**simulation type** This part indicates the type of the simulation that was executed. There are two possible values : **sim** indicates that the results was obtained after running the simulation and **web** indicates that the results are obtained after sending requests to real web site.

**number of threads used** This part indicates the number of threads that was used in the thread pool. Possible value are **TX** where **X** is an integer.

Running the same code many time with unchanged parameters will append the results in the existing log file that was generated in the first run.

Figure 3.5 shows an example of the contents of the produced log file.



```
1 Experiments results :
2 Experiments run with the following the results :
3 FAILURES_THRESHOLD : 5
4 RETRY_TIMEOUT : 100us
5 PROCESSING_DURATION : 100us
6 MTHREADING : 2 Thread(s)
7
8 TestRunner Circuit Breaker run with deadline set at : 50%
9 -----
10 REQUEST Durations( us ) Errors Success Success Ratio(%) Trip Ratio(%)
11 -----
12 10 797 10 0 0 100
13 50 2063 48 2 4 70
14 100 7840 73 27 27 40
15 500 36383 445 55 11 68
16 -----
17 Summary :
18 request = [ 10 50 100 500 ]
19 durations = [ 797 2063 7840 36383 ]
20 errors = [ 10 48 73 445 ]
21 success = [ 0 2 27 55 ]
22 ratio_success = [ 0 4 27 11 ]
23 ratio_trip = [ 100 70 40 68 ]
24 TestRunner Circuit Breaker run with deadline set at : 75%
25 -----
26 REQUEST Durations( us ) Errors Success Success Ratio(%) Trip Ratio(%)
27 -----
28 10 1275 10 0 0 100
29 50 5703 39 11 22 50
30 100 8471 93 7 7 60
31 500 33586 481 19 3.8 72
32 -----
33 Summary :
34 request = [ 10 50 100 500 ]
35 durations = [ 1275 5703 8471 33586 ]
36 errors = [ 10 39 93 481 ]
37 success = [ 0 11 7 19 ]
38 ratio_success = [ 0 22 7 3 ]
39 ratio_trip = [ 100 50 60 72 ]
40 TestRunner Circuit Breaker run with deadline set at : 100%
41 -----
```

Figure 3.5: Log file contents

The results contained in these two files are plotted into diagrams. The diagrams are then compared to analyze the circuit breaker behavior during the test. Different results are produced by varying these parameters to observe their impact on the circuit breaker.

# 4 Experimental Results

This part of the document presents the results of the tests run on the circuit breaker. Observations are made on how the circuit breaker reacts to the different requests sent.

## 4.1 Observations

The results here show the impact of the the three main settings under which the circuit breaker should work once it is created. Those settings are :

- deadline (set to a percent of the `PROCESSING_DURATION`)
- retry time
- the failure threshold

For the simulation the idea was to test the circuit breaker with a large set of different requests number : [1, 10, 100, 1000, 10000, 100000, 1000000]. The obtained results has shown that there weren't any difference in the way the circuit breaker responded, so the requests was reduced to : [10, 50, 100, 10000] because it took too long to run 1000000 requests when the time unit is in milliseconds.

## 4.2 Simulation

In the simulation we observe how the circuit breaker reacts depending on the values of the deadline, the failure threshold and the retry time attributes. The simulations are run successively with :

- failure threshold = 5, retry time = 100ms, **deadline** varying to [10%, 50%, 100%, 150%, 200%]
- deadline = 100ms, retry time = 100ms, **failure threshold** to [3,5,10,30]
- failure threshold = 5, deadline = 100ms, **retry time** varying to [10ms, 50ms, 100ms]

### 4.2.1 Effect of the deadline

#### Effect on duration

In this simulation we see that the duration to process all the submitted requests grows with the deadline. In the scenario where the deadline is too high, the circuit breaker doesn't take longer compare to when the service is directly called, see Fig 4.1 where the deadline is above 100%. We see that a higher deadline value (above 100%) has no negative impact on the duration when the service is working properly. The circuit breaker doesn't wait longer as expected. A lower duration is not always an indication that the service is replying timely. Here the Figure shows that the deadline has a big impact when the service is not responding timely, which is the expected behavior of the circuit breaker. Another interesting point to observe is that the request sent through the circuit breaker doesn't take longer than the request sent without using the circuit breaker. This is because the implementation is using the C++ move semantic to forward the task and parameters around which keeps the call overhead very low.

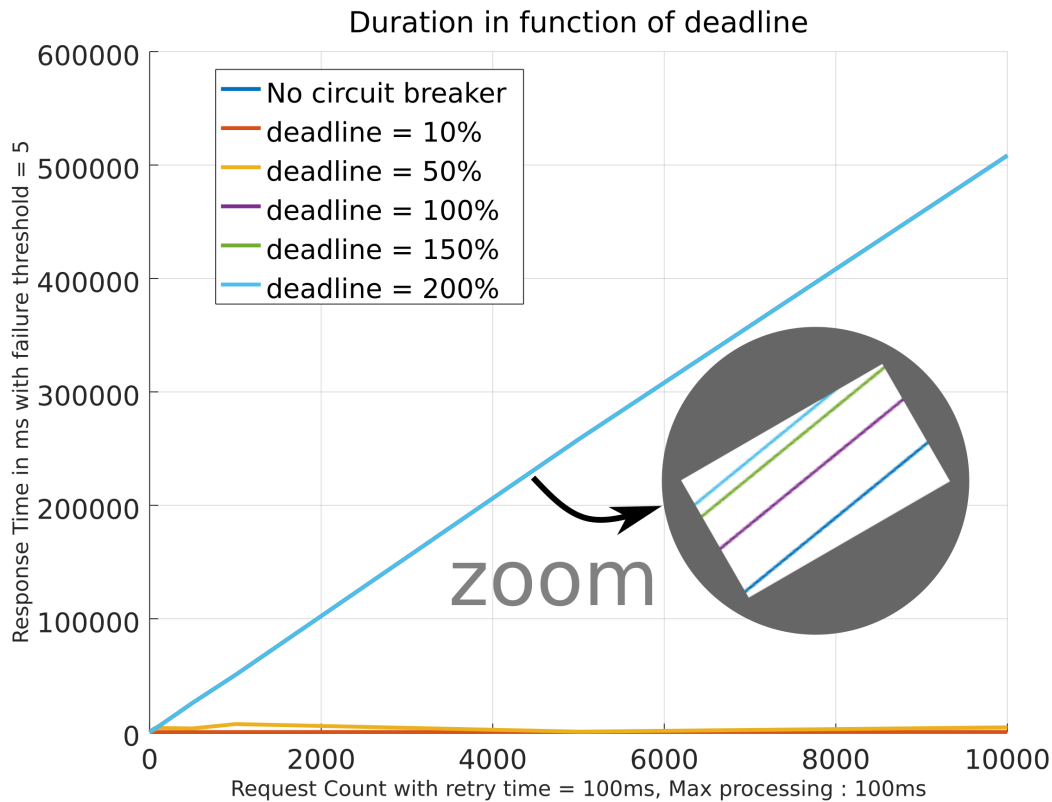


Figure 4.1: Duration in function of the deadline

### Effect on success ratio

Observing the success ratio in Fig 4.2 shows that setting the deadline to 100% of the maximum processing time required by the service is largely enough to lead to best results. The success ratio is defined as the number of successful request divided by the number of requests sent. Fig 4.2 also shows the success ratio is strongly impacted by the deadline's value. This is due to the successive submitted request which are still in the task queue waiting for a worker thread to run them. Since the service need more time to process them, the deadline will probably timeout before they are run by a worker thread. Increasing the number of worker threads in thread pool will improve this tendency. But since the service is not able to hold on the deadline, increasing the number of worker threads will be a waste of resources.



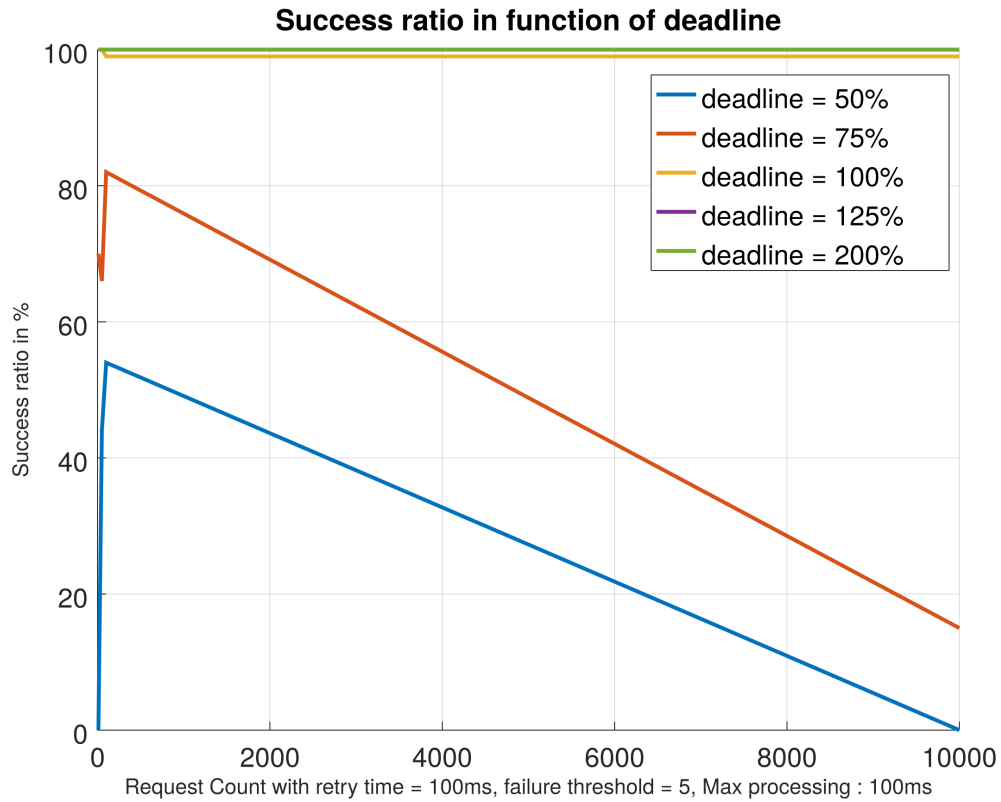


Figure 4.2: Success ratio in function of the deadline

The same observation is done with the time resolution set in microseconds in Fig 4.3 . We see that the Success ratio behavior is different. It doesn't decrease with the number of submitted requests. This is due to the fact that service is processing the request very fast (in microseconds) in comparison to Fig 4.2 where the time resolution is in milliseconds

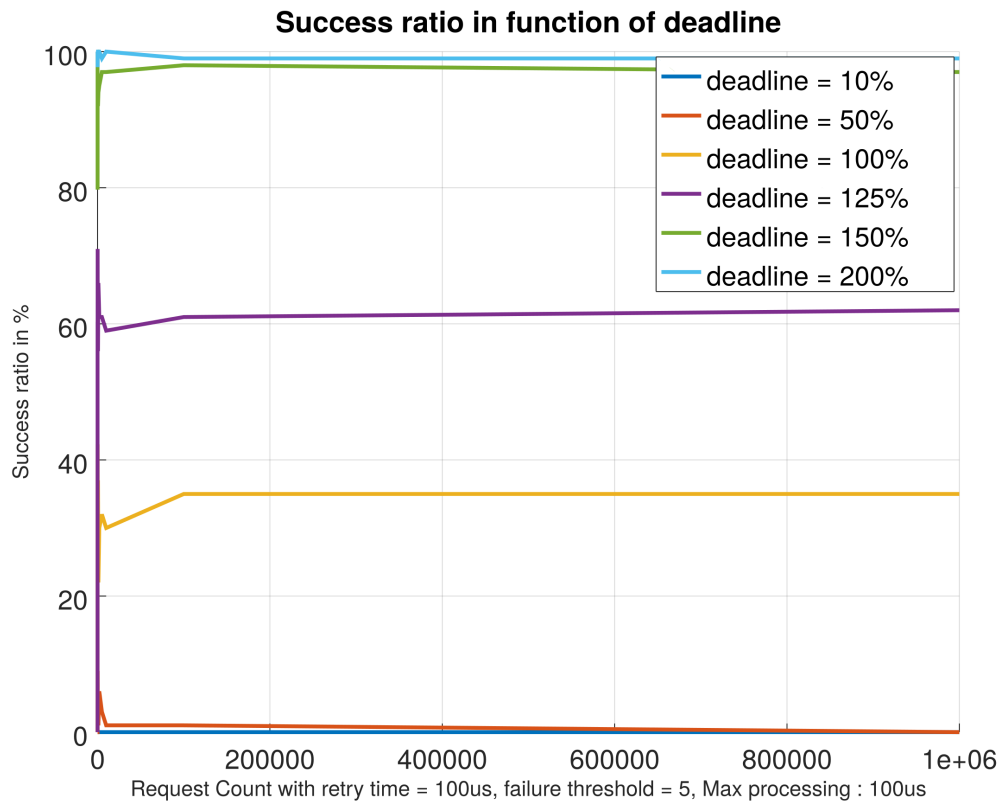


Figure 4.3: Success ratio in function of the deadline

### Effect on trip ratio

Observing the trip ratio in Fig 4.4 shows that a lower deadline has a big impact on the trip ratio. The trip ratio is the number of time the failure threshold is reached divided by the maximum number of trip.

$$max_{trip} = \frac{n_{request}}{failures\ threshold} \quad (4.1)$$

where :

- $n_{request}$  = number of request
- $failures\ threshold$  = number of failures in a row
- $max_{trip}$  = number of possible trips

$$trip\ ratio = \frac{n_{trip}}{max_{trip}} \quad (4.2)$$

where :

$n_{trip}$  = number of trip

$max_{trip}$  = number of possible trips

This information shows how often the circuit breaker becomes open. When there is a supervisor monitoring the circuit breaker it means it will be called most of the time the more the service is used.

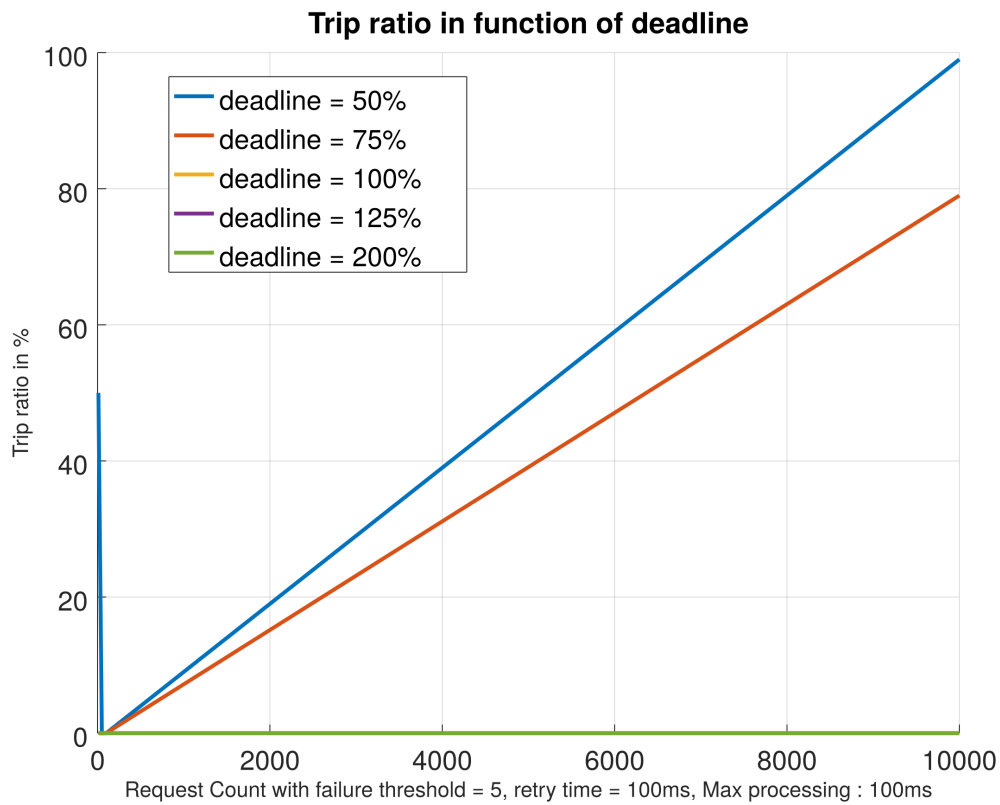


Figure 4.4: Trip ratio in function of the deadline

### 4.2.2 Effect of the failure threshold

We now observe the effect of failure threshold on the circuit. During experiments, it was observed that the failure threshold has no impact on the circuit breaker when the deadline is set at 100%. For this reason the observation here is done with the deadline  $< 100\%$ . For this test the deadline is set to 75% of the maximum processing time required by the service.

#### **Effect on duration**

Fig 4.5 shows that the time needed to process the submitted requests grows with bigger failure threshold values. A lower failure threshold means the circuit breaker will be in open state most of the time when the service is constantly failing and so it will always fail fast, this is why the duration is shorter when the failure threshold is lower, as expected. Which failure threshold value should be chosen? There is no fixed answer to that question because it depend mostly on the context in which the circuit breaker will be used. In some factory like a chemical factory a failure threshold higher than 2 is not conceivable. In some others context like requesting data from a website, using a failure threshold with bigger value will not have any big impact. As we see, the failure threshold is import when the service is constantly failing.

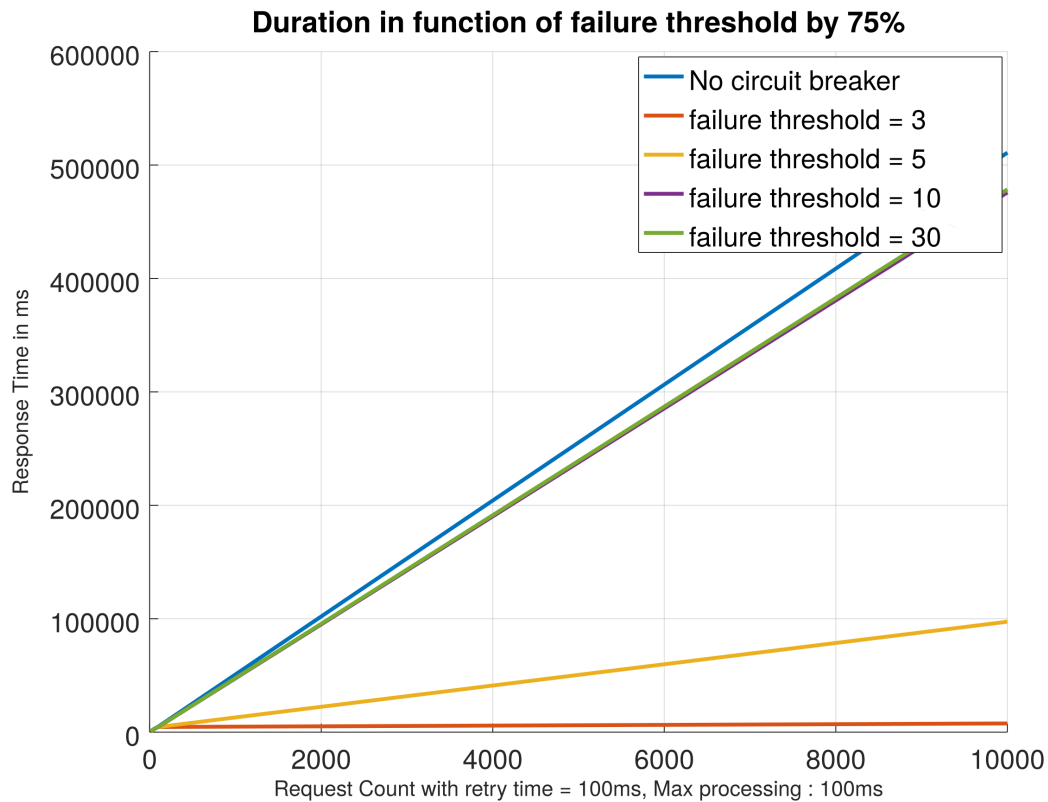


Figure 4.5: Duration in function of the failure threshold

### Effect on success ratio

Another effect to observe is the impact of the failure threshold on the success ratio. Fig 4.6 shows that the higher the failure threshold the higher the success ratio. This is explained by the fact that when the failure threshold value is big the circuit breaker allows more requests to fail in a row without transitioning from closed to open state. For example in a sequence of 10 consecutive requests, the two first requests might fail and the height others might succeed. With a failure threshold in this example set to 2 it will make the circuit breaker trip and the 8 others requests will simply fail fast. With a failure threshold set to 10, the circuit breaker will not trip after the two first requests and will be able to process the height others requests.

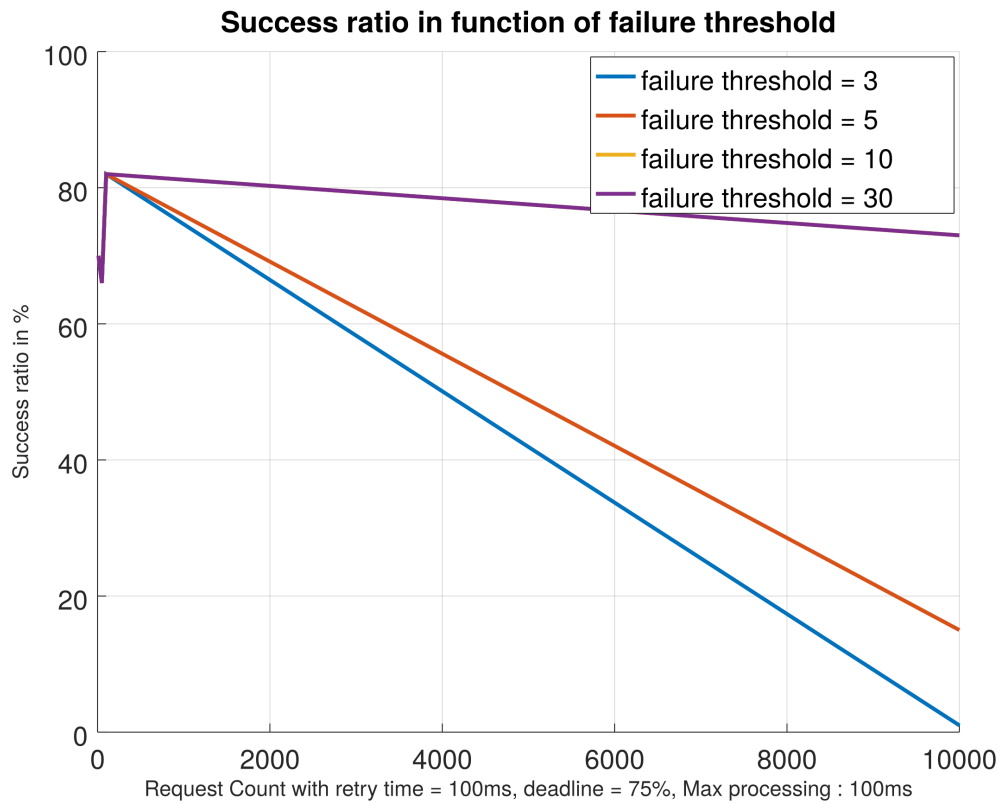


Figure 4.6: Success ratio in function of the deadline

### Effect on trip ratio

Here we see how often the circuit breaker becomes open. Fig 4.7 confirms the observation made on the effect of the failure threshold on the success ratio above. We observe in Fig 4.7 that the failure threshold has a big impact on how often the circuit breaker is in the open state when the service is not working. The lower the failure threshold the higher the trip ratio. For example when the failure threshold is set to 30, the circuit breaker never becomes open. This is explained by the fact that there must be 30 consecutive failures before the circuit breaker transitions from closed to the open state.

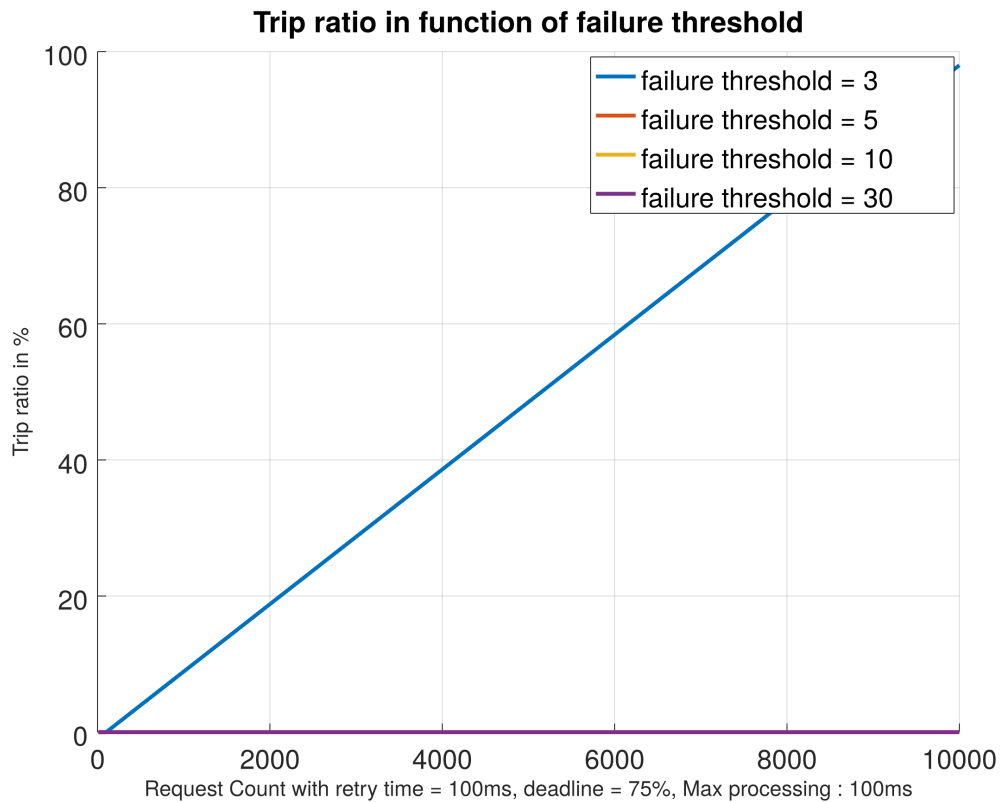


Figure 4.7: Trip ratio in function of the failure threshold

### 4.2.3 Effect of the retry time

In this part we observe the effect of the retry time value on the circuit breaker. For this observation the failure threshold is fixed at 5, the deadline is fixed at 75% and the retry time is varying.

#### Effect on duration

Fig 4.8 shows that the bigger the retry time the lower the duration time needed for processing the all the sent requests. This is explained by the fact that when the circuit breaker is open, it stays longer in that state until the retry time elapsed.

Since the circuit breaker is open it fails fast for every request sent in that period of time as expected.

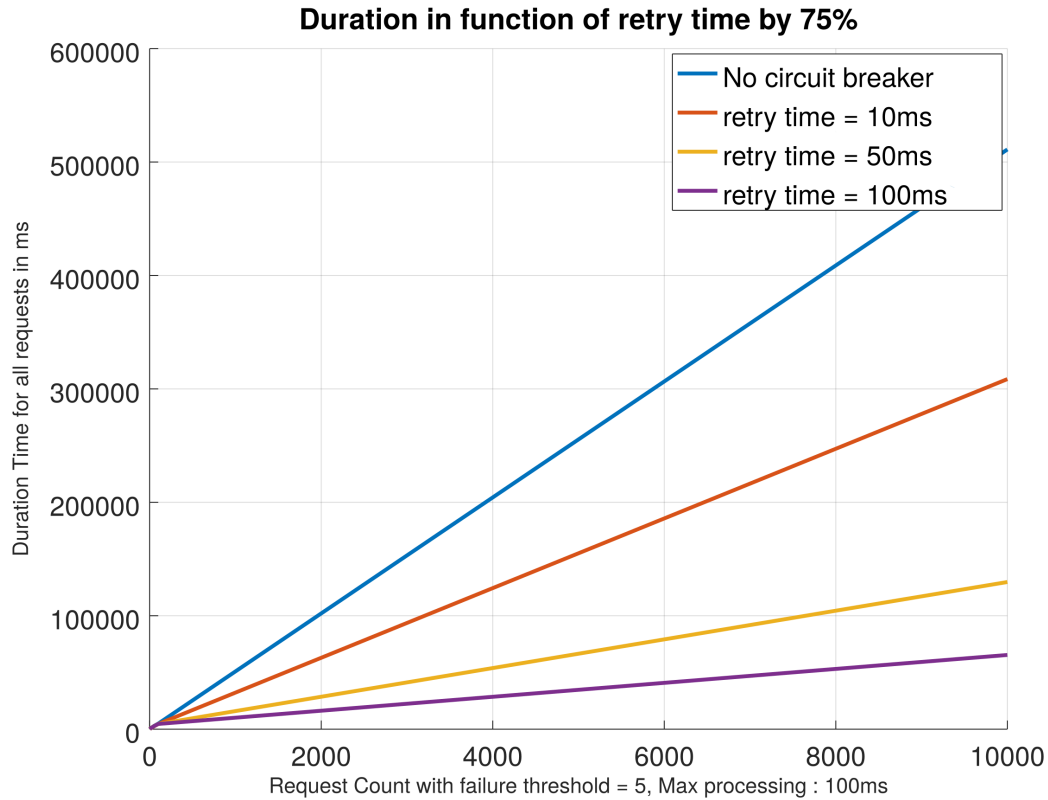


Figure 4.8: Duration in function of the retry time

### Effect on success ratio

Fig 4.9 shows that the retry time has an impact on the successfully completion of a submitted request. The higher the retry time the lower the success ratio. This observation align itself to the circuit breaker's behavior. A higher retry time means the circuit breaker stays longer in the open state when this state is active. Every request sent while the circuit breaker is open will be counted as a failure, which keeps the success ratio lower. A supervisor, when installed, could dynamically change this attribute depending on well the service is working. For example when



the service keeps failing the supervisor could increase or decrease the retry time depending the context.

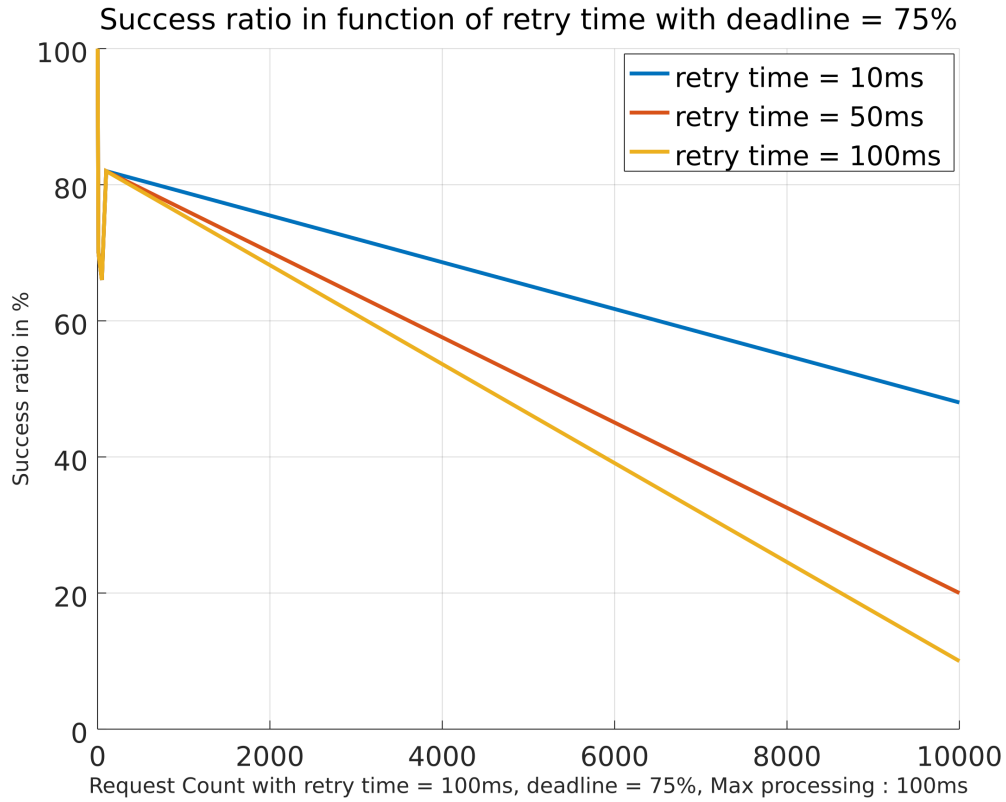


Figure 4.9: Success ratio in function of the retry time

### Effect on trip ratio

In Fig 4.10 we see that the trip ratio is growing with the value of the retry time. This behavior is explained by the way the trip ratio is computed.

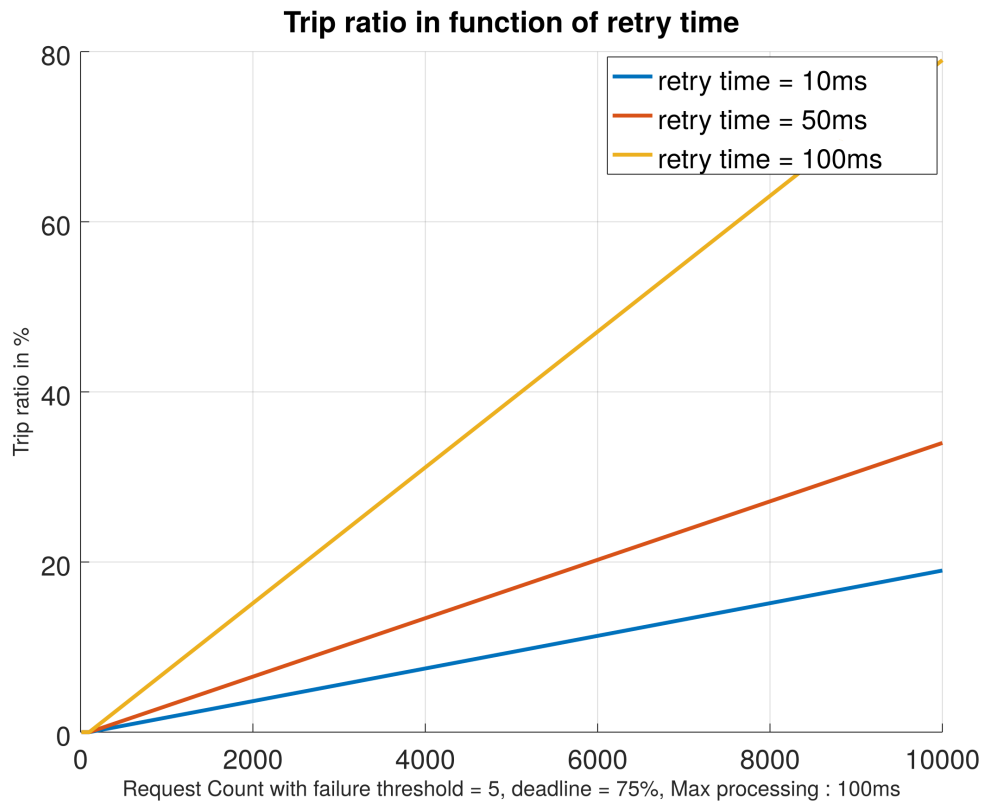


Figure 4.10: Trip ratio in function of the retry time

The trip ratio must be observed together with the success ratio to have a better view of the circuit breaker's performance. This is because when the circuit breaker is open, the failures are still counted, and with that the number of trip is still computed. This is why the trip ratio is growing with the retry time.

### 4.3 Web service Test

In this part we observe the circuit breaker's behavior when used with a real website request. The URL used for this test is <https://www.example.com>. The failure threshold is set to 5, the retry time is set to 476ms and the deadline is varying from 25% to 200% of the maximum response time. To obtain an approximation of the maximum response time, a series of 30 requests to the website was probed

and the response delay was measured. The obtained result is 476ms. The network used for this test is the HAW.1x wireless network.

The observations made for this test is not different from the simulation test. To avoid repetitions only the deadline observation is analyzed here.

### 4.3.1 Effect of the deadline

#### Effect on duration

To no surprise, the result shown on Fig 4.11 aligns itself with the results observed in section 4.2.1. A lower deadline leads to a lower duration. On the other hand setting the deadline much higher , 200% of 476ms doesn't lead to a duration longer than when the service is called without using the circuit breaker.

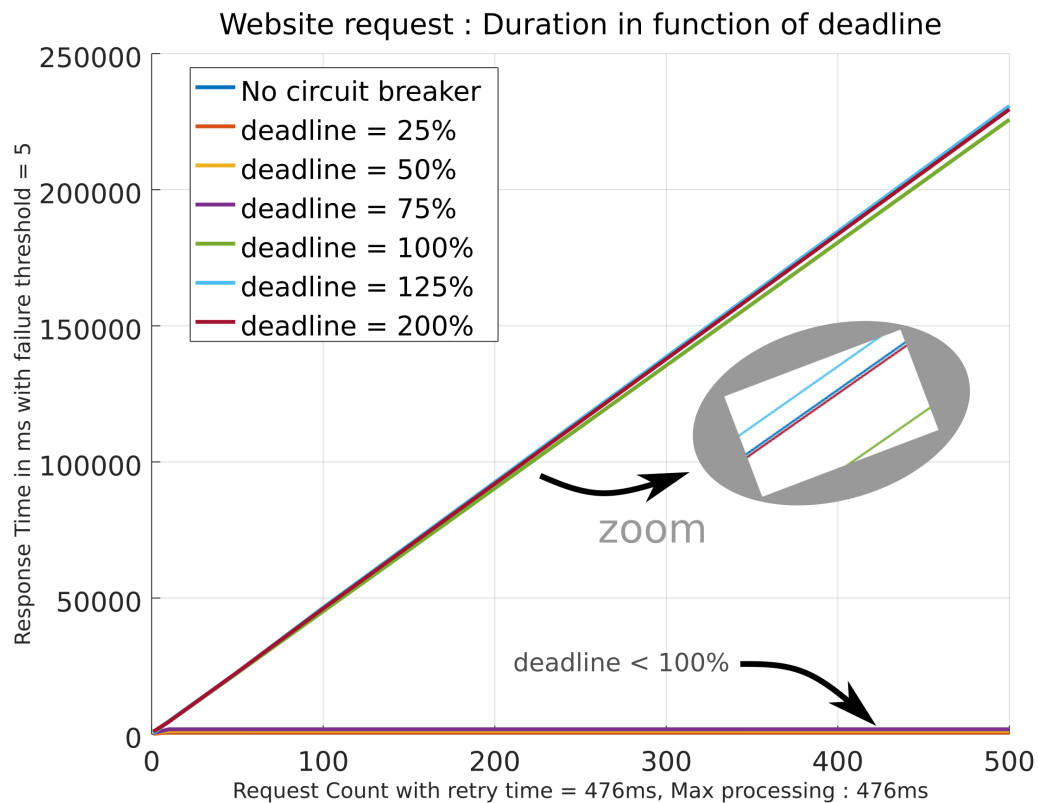


Figure 4.11: Web service : Duration in function of deadline

### Effect on success Ratio

The result we observe in Fig 4.12 is also not different from the tendencies we have observed previously. Setting the deadline below 100% of 476ms lead to no successful requests. This is due to the fact that at the time the test was run, the site always needs around 476ms to reply. To no surprise, setting the deadline above 100% of 476ms lead to more successes.

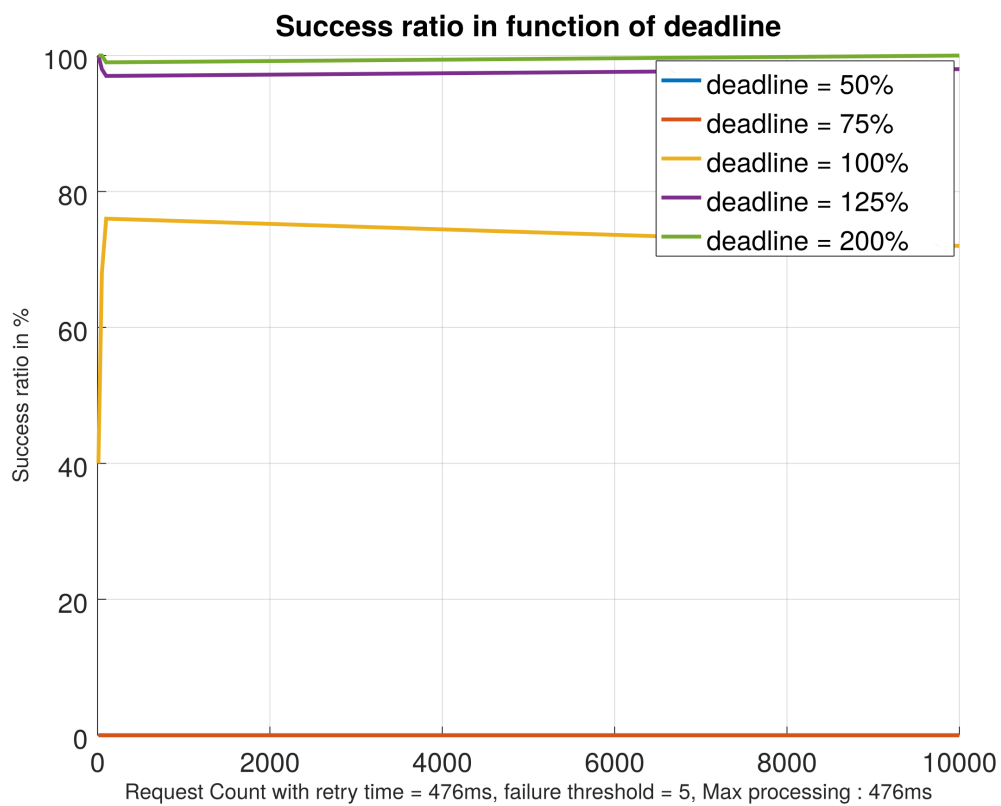


Figure 4.12: Web service : Success ratio in function of deadline

### Effect on trip Ratio

Since the website has a response time of around 476ms, for a deadline set far below 476ms every request is failing. On the other hand, with a deadline equal or greater than 476ms every request succeeds, so the circuit breaker can not trip. This result align itself to the observation made in the simulation test.

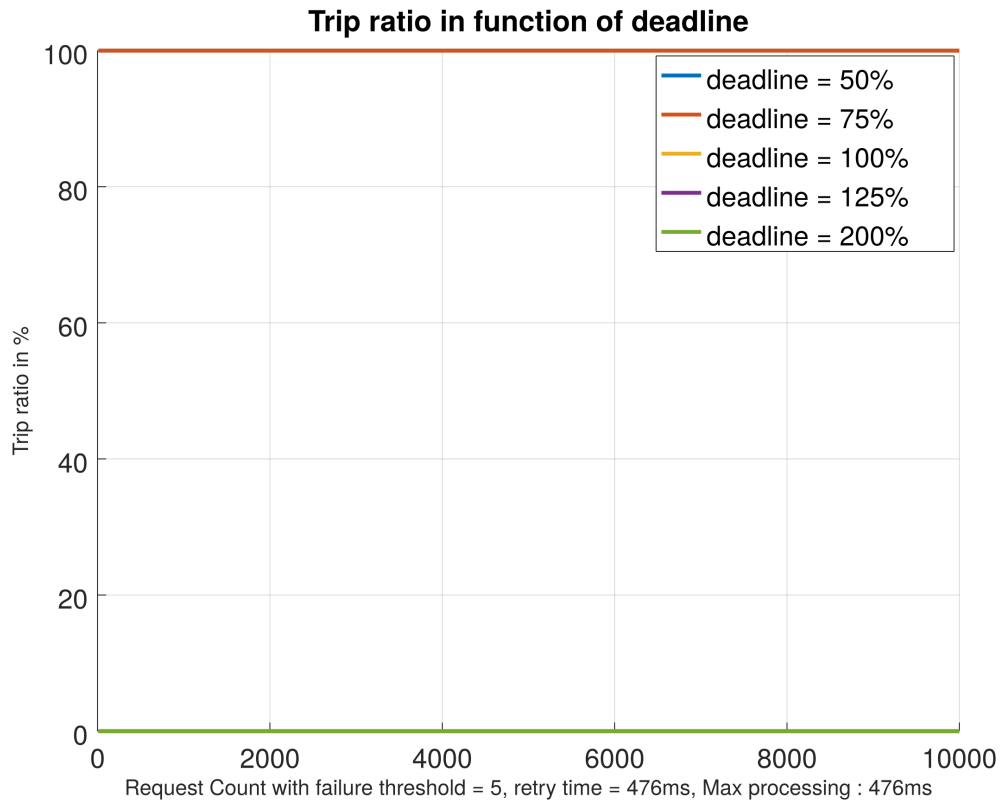


Figure 4.13: Web service : Success ratio in function of deadline

## 4.4 Conclusion

When the time resolution is in microseconds, choosing a deadline set at 100% of the maximum processing time required by the service doesn't lead to results near 90% success whereas it does when working in milliseconds. Because of thread

context switching and the operating system activities it is not possible for the service to hold on this strong deadline when the time unit is in microseconds. The deadline must be set much higher. The tests shows that 150% is good setup for the deadline for both time units. A big value for the deadline doesn't negatively impact the time needed to wait for the results of all submitted requests. The failure threshold and retry time have impact only when the service is not responding in the deadline. The test shows that the call overhead caused by the request going through the circuit breaker has no impact on the performance. In this circuit breaker implementation, calling the service directly took the same time as calling the service through the circuit breaker, so no penalties for using this circuit breaker implementation. The current implementation of this circuit breaker doesn't prevent the client from sending requests. When all the worker threads are busy processing the requests, the new arriving requests will not be processed until a worker is free. In this case the submitted tasks that are waiting in the queue will fail if they have a tight deadline. A solution to this problem will be to add back pressure capability to the Thread Pool.

## 5 Summary

The Circuit Breaker pattern, popularized by Michael Nygard in his book, *Release It!*, can prevent an application from repeatedly trying to execute an operation that's likely to fail. Using the circuit breaker allows us to detect failures in a system. It also allows us to prevent the failures to spread over the system and helps to build a resilient system.

Failure threshold, retry time and deadline are use In this work is was observed that the failures threshold, retry time and deadline have no influences as long as the service protected by the circuit breaker has no failures. When the service fails, the retry time helps the circuit breaker to periodically check the service state. In this case the circuit breaker reduces the processing by failing fast. The failure threshold limits the number of allowed consecutive failures whereas the deadline limits the waiting time. The choice of the failure threshold and the retry time is not aleatory and depends on the context in which the circuit breaker will be used.

The experiments have shown that the call overhead added has no negative impact on the performance. This implementation is using the new C++ move semantic. The move semantic avoid the copy of the request around. Instead only the addresses of the data are copied. This mechanism helps to save time. This has helped to keep the call overhead very low as the experiments have shown.

The Circuit breaker should not be used to collect requests when the service is down and process it later. Doing so could create a feedback loop, leading the system to oscillate between unavailable and overloaded.

One advantage of using the circuit breaker is that it makes the system easy to monitor. Nonetheless, to make fully use of the advantages of the circuit breaker it

is necessary to think about how the system should react in case of failures and how the information are forwarded. Message Broker such as RabbitMQ and ActiveMQ can be used to forward messages and help the system stay responsive.

This implementation can be used to implement a full C++ fault tolerant library for reactive system programming since it build as C++ library. The implemented circuit breaker can be used as such with any service without changing the code. This is made possible by using the C++ template system.



# Bibliography

- [1] Conal Elliott. “Declarative Event-Oriented Programming”. In: *Principles and Practice of Declarative Programming*. 2000. URL: <http://conal.net/papers/ppdp00/>.
- [2] Douglas Schmidt et al. “Concurrency Patterns”. In: *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. 1st ed. John Wiley & Sons, 2000, pp. 318–342. ISBN: 0471606952.
- [3] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. 1st ed. Pragmatic Bookshelf, 2007. ISBN: 9780978739218.
- [4] Jan Sommerville. *Software Engineering*. 9th ed. Addison-Wesley, 2011. ISBN: 978-0-13-703515-1.
- [5] Anthony Williams. *C++ Concurrency in Action : Practical Multithreading*. 1st ed. Manning Publications Co, 2012. ISBN: 9781933988771.
- [6] The Reactive Manifesto. *The Reactive Manifesto*. 2014. URL: <https://www.reactivemanifesto.org>.
- [7] Mark Richards. *Software Architecture Patterns : Understanding Common Architecture Patterns and When to Use Them*. 1st ed. Oreily, 2015. ISBN: 978-1-491-92424-2.
- [8] Jonas Bonér. *Reactive Microservices Architecture : Design Principles for Distributed Systems*. 2nd ed. Oreily, 2016. ISBN: 978-1-491-95779-0.
- [9] Roland Kuhn, Brian Hanafee, and Jamie Allen. “Fault tolerance and recovery patterns”. In: *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. Chap. 12, p. 179. ISBN: 9781617291807.
- [10] Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. ISBN: 9781617291807.

- [11] Roland Kuhn, Brian Hanafee, and Jamie Allen. “Replication Patterns”. In: *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. Chap. 13, p. 184. ISBN: 9781617291807.
- [12] Roland Kuhn, Brian Hanafee, and Jamie Allen. “Supervison”. In: *Reactive Design Patterns*. 1st ed. Manning Publications Co, 2017. Chap. 2, p. 30. ISBN: 9781617291807.
- [13] David Vandevorde, Nicolai M. Josuttis, and Douglas Gregor. 2nd ed. Addison-Wesley, 2018. ISBN: 978-0-321-71412-1.
- [14] David Vandevorde, Nicolai M. Josuttis, and Douglas Gregor. “Implementing traits”. In: *C++ Templates : The Complete Guide*. 2nd ed. Addison-Wesley, 2018. Chap. 19. ISBN: 978-0-321-71412-1.

# A Appendices

These documents are available in the CD.

- Doxygen Code documentation compressed in a zip file.
- Raw data of all tests performed(.zip)
- The code source of the implemented circuit breaker as an archive file. (.tar.gz)

# Glossar

**ActiveMQ** Apache ActiveMQ is a message broker written in Java with JMS, REST and WebSocket interfaces, however it supports protocols like AMQP, MQTT, OpenWire and STOMP that can be used by applications in different languages.

**CMake** CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.

**FSM** FSM(Finite State Machine), is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs.

**functor** In C++ a function object is a class that acts like a function. A functor is a class that implements the operator(). That means if f is an instance of a functor we can use f as function by calling f().

**make** GNU Make is a tool which controls the generation of executable and other non-source files of a program from the program's source files. Make gets its knowledge of how to build your program from a file called the makefile, which lists each of the non-source files and how to compute it from other files. When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program.

**RabbitMQ** RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, Message Queuing Telemetry Transport, and other protocols. Protocols supported by RabbitMQ are AMQP 0-9-1, AMQP 1.0, STOMP 1.0 through 1.2 and MQTT 3.1.1.

**std::async** The template function `async` in C++ runs a function `f` asynchronously (potentially in a separate thread which may be part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call.

**std::future** The class template `std::future` in C++ provides a mechanism to access the result of asynchronous operations.

**STL** The STL, Standard Template Library, is a software library for the standard C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called algorithms, containers, functions, and iterators.

**thread pool** A thread pool is a group of worker thread that run task on behalf of the application.

**URL** Uniform Resource Locator : URL is a string of characters that identifies a particular resource.

**Worker Thread** A worker thread is a thread which runs to execute a background task.

## **Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit**

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

*Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI*

## **Erklärung zur selbstständigen Bearbeitung der Arbeit**

Hiermit versichere ich,

Name: \_\_\_\_\_

Vorname: \_\_\_\_\_

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

**Reactive Design Patterns**

**Implementierung eines Circuit Breaker Patterns in C++**

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

---

Ort

Datum

Unterschrift im Original