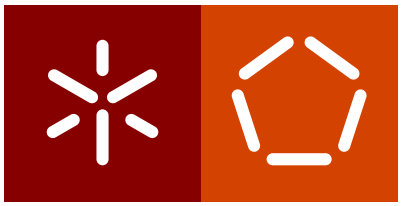
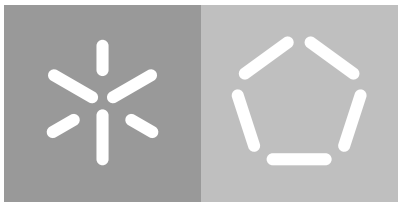


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marco António Rodrigues Oliveira Silva

Improving the Resilience of Microservices-based Applications

March 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marco António Rodrigues Oliveira Silva

Improving the Resilience of Microservices-based Applications

Master dissertation
Intregated Master's in Informatics Engineering

Dissertation supervised by
Prof. André Ferreira
Prof. Jácome Cunha

March 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

ACKNOWLEDGEMENTS

I would like to express my thanks to everyone who made all the work developed in this thesis possible. To my mentor Jácome Cunha, who has always been available to clarify any doubt, whether within his domain of knowledge, or in the mobilization of means within the institution to solve any problem. I would also like to thank you for all your support regarding the management and planning of all the work. To my other mentor, André Ferreira, who made it possible to get a case study that was adequate and vast enough for the study of the theme to be viable, only in this way it would be possible to achieve the objectives initially established for this dissertation. In addition, I thank you for all the contribution in this work with all your experience in the software industry. Also to my two mentors mentioned above, for all the knowledge transmitted in the most diverse aspects, whether or not human, for the confidence and effort in the proposal of such a rich and vogue theme today.

Finally, special thanks to all those who affected the work developed in this dissertation, whether or not directly. In a special way, I could not fail to mention my parents, for all the effort made so that the conclusion of this Masters was possible, for all the support in the least happy moments, encouragement and guidance in all the decisions taken during my academic career.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Atualmente, escalabilidade, manutenibilidade e disponibilidade são algumas das medidas mais utilizadas na avaliação qualitativa de software. Com uma presença cada vez maior de produtos de software no nosso dia a dia, há consequentemente a necessidade de torná-los melhores aos olhos do utilizador, surgindo novos desafios a serem explorados e superados na hora de projetar e desenvolver produtos de software.

Mais focado neste tema de dissertação de mestrado, a resiliência é de facto um ponto chave para o sucesso de um qualquer produto de software. Cada vez mais as pessoas se encontram diretamente ligadas a produtos de software no seu dia a dia, o que torna o bom funcionamento destes essencial. Assim sendo, o estudo de metodologias que permitam aumentar a resiliência e consequentemente a disponibilidade destes serviços ganha relevância.

O principal objetivo desta dissertação é desenvolver uma metodologia para aumentar a resiliência de soluções orientadas aos microsserviços. Assim, é fundamental primeiro entender quais soluções já desenvolvidas para esse fim. Após reunir um conjunto de técnicas para aumentar a resiliência, analisamos um caso de estudo procurando possíveis problemas de resiliência. Para além desta procura de vulnerabilidades, foram apresentadas propostas para a sua resolução, tendo em conta o conjunto de soluções já levantado. Por fim, e avançando para a construção da metodologia alvo da dissertação, procedeu-se à análise de todas as propostas apresentadas, bem como à caracterização das interações problemáticas. Desta forma, foi possível extrair a informação necessária do estudo para a construção da metodologia. Como resultado deste estudo, também foi possível identificar uma nova proposta para aumentar a resiliência diante das necessidades do estudo de caso e da recorrência em que esta se tornou útil.

Keywords– microservices, resilience, patterns, service degradation, distributed systems

ABSTRACT

Currently, scalability, maintainability, and availability are some of the most used measures in the qualitative evaluation of software among developers. With an increasing presence of software products in our daily lives, there is, the need to make these products better in the eyes of the user, therefore raising new challenges to be explored and overcome when designing and developing software products.

This work focuses on this master's thesis theme, resilience is in fact a key point for the success of any software product. More and more people are directly connected to software products in their daily lives, which makes their smooth functioning essential. Therefore, the study of methodologies that allow the increasing availability of these services undoubtedly gains relevance.

The major objective of this dissertation is to develop a methodology for increasing the resilience of microservices-based solutions. Thus, it was essential to first understand what solutions had already been developed for this purpose. After assembling a set of techniques for increasing resilience, we analyzed a case study and searched for possible resilience problems. Besides this search for vulnerabilities, proposals were made for their resolution, taking into account the set of solutions already raised. Finally, and moving towards the construction of the dissertation's target methodology, an analysis was performed of all the proposals made as well as the characterization of problematic interactions, making it possible to generalize the study and reach the objective of the dissertation. As a result of this study, it was also possible to identify a new proposal to increase resilience given the needs of the case study and the recurrence in which it has become useful.

Keywords- microservices, resilience, patterns, service degradation, distributed systems

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

Users are increasingly demanding more when it comes to the resilience of the software products they use in their daily lives. Consequently, developers need new solutions capable of meeting with this increase of the user's demands. Increasing the resilience of any solution affects the system's quality perceived, considering its greater availability. When solution components are unavailable, the service availability as a whole can only be maintained through service degradation.

Microservices success comes mainly from allowing the development of very complex solutions that with a monolithic paradigm were almost impossible to achieve. Moreover, its ability to scale horizontally is also determinant due to the higher computational load, owing to more complex tasks implemented in software and a higher number of users.

To better understand what a microservice-oriented solution actually comprises, start by defining the opposite, a monolithic solution. An application is monolothic if it contains the entire application code in a single codebase and shares the execution environments. Typically the process space is shared at runtime for the entire application. These types of solutions are simple to build, test and deploy when compared to a microservices architecture.

On the other hand, microservices provides an architectural style that promotes the solution division into simpler and smaller components. This division process must be carried out according to certain guidelines so that the resulting components are as independent and as stateless as possible. Ideally, this division should be oriented towards functional cohesion, making it easier to establish the responsibilities of each of the constituent components of the solution.

1.1 PROBLEM

The increase in performance of solutions that follow this architectural style is mainly due to its ability to scale. When a system is overloaded, not all of its components are homogeneously affected. In a microservice oriented solution, each component can in most cases be replicated as many times as necessary until the desired performance levels are reached. Deploy the overloaded component in a more powerful machine is also an option, satisfying the higher

computational power demand by adding more CPUs, memory and I/O capability to the machine. This is called *vertical scaling*.

One other key point for any software product today is its availability. By current standards, the fact that a product is not functional when necessary is a reason why it is no longer used or even replaced by other that offers the same or similar service. These service interruptions can be caused by several reasons, such as network or the constituent components failures. By that, we can see that availability and resilience are strongly related. More specifically, resilience is defined as the ability to deal with possible errors that occur in the solution components, preserving the product's functionality as much as possible.

On the other hand, resilience can also be seen as preparing a system for possible failures; more specifically, the ability to detect and prevent the occurrence and spread of failures. Also, it should be noted that such detection and treatment of system failures should disturb the system as little as possible, while witholding as much as possible from the user any problems that have occurred. Another of the great advantages of isolating functional components in a system is a greater ability to isolate possible errors. The failure of one of the functional components of the solution will not necessarily imply the non-operation of the remaining components, as long as they are not interdependent. Therefore, it is possible to degrade the service instead of its total failure, favoring availability.

Bearing in mind that microservices define the solution through small, simple, and oriented to functionality components, communication between them will always be needed to ensure their smooth operation. Even in an ideal scenario, without dependencies between components, the traffic routing process needs to be carried out through the infrastructure network. As a result, when any communication is made using the network, it becomes a potential source of failures.

As Rotem-Gal-Oz (2008) explains, when development is carried out in a distributed environment, preventive measures must be taken so that the system is prepared to deal with possible failures. As a consequence of the solution components distribution, the solution resilience becomes a problem. Therefore, this study focuses on ways to increase this resilience in microservices oriented solutions.

1.2 OBJECTIVES

This thesis aims to answer the following questions.

- **What are the available patterns to make a resilient microservices-based application?**

  The objective is to document and describe in detail the existent strategies to introduce resilience in a *microservices* oriented product.

- **How should each of the existent solutions be applied?**

The objective is to understand in which context each of the existing solutions is best suited. In addition, it is also essential to understand whether implementing the solution has any counterpart or any essential requirement for its correct implementation.

- **What is the benefit in terms of resilience?**

  Different solutions can be applied in different scenarios, resulting in different final results. It is necessary to comprehend whether the solution brings advantages effectively to the final solution as well as checking that the combinations of different solutions does not cancel out the effect.

1.3 Thesis Methodology

Naturally, before starting any study, it is necessary to understand the current scenario regarding the theme (Section 2). Bearing in mind that it is intended to build a methodology for applying solutions to increase resilience, the first step was the investigation and survey of the techniques already proposed for this purpose. With this study, it was possible to perceive the existence of a wide range of techniques and solutions that promise to increase the microservices oriented solutions resilience. In addition, we also perceive the existence of large amounts of information on the topic spread through blogs, non-scientific articles or forums, as this architectural style can be considered emergent and is mainly driven by industry. Since these means are not scientifically verified at all, we started an exhaustive analysis of these sources, thus enabling the collection of the most referenced set of solutions in the literature.

After meeting the set of proposals, an analysis of interactions between components of a case study was carried out, to find possible points of failure and propose solutions for their mitigation (Section 3). Still, in this case study, it should be noted that during its development process, no techniques were applied to increase the solution's resilience.

After completing the survey of possible error scenarios and proposing the respective solutions, each of these solution proposals was characterized using a set of key factors extracted from the analysis of each the solution components (Section 4). This set of factors characterizing interactions and the respective resilience solutions proposals compose the final objective of this study, the production of a methodology to increase the resilience of microservices based solutions.

2

STATE OF THE ART

In this chapter, we introduce in more detail the concepts necessary to understand this work (Section 2.1) and present the state of the art in creating resilient microservices oriented solutions (Sections 2.1.2 and 2.2).

2.1 CONCEPTS

2.1.1 *Microservices*

According to Newman (2015), as additional features are implemented for a product, the codebase also increases, which can cause problems. As this codebase increases in size, consequently, find the code that implements each feature will also become more complex. Although there are already methodologies that aim to make these large code bases modular and easy to consult, if they are not applied correctly, the good maintenance of these code bases can be called into question. As a result, the implementation of error correction or the changing of features becomes increasingly difficult due to the dispersion of the code.

In a microservices based architecture, the services division is oriented to features, thus avoiding the problems associated with large teams and large code bases (Newman, 2015). Furthermore, this focus on features counteracts the tendency of excessive growth of components in size, avoiding maintenance issues.

The benefits of an architecture based on microservices are numerous and varied, many of which are common to a distributed system. However, this architectural style tends to obtain better results than a distributed system, since it takes more detail into all the principles initially established in a distributed system.

One of the major benefits of this architectural style is the technological heterogeneity, allowing the choice of the most appropriate technology for implementing each of the services. In addition, the components that make up the solution are independent, they have a low complexity level and when possible are stateless. This allow new instances of a service to be more easily deployed if necessary, making the solution more scalable. Furthermore, this independence facilitates the deployment of the solution since only components that undergo

changes need to be deployed. In addition, this independence also facilitates the organization of developers in teams dedicated to each of the services, thus enabling the parallelization of the development of each of these components. Finally, this feature promotes the reuse of the components already developed, since different combinations of these give rise to different solutions.

2.1.2 *Resilience*

The microservices' architectural style is not entirely new, but nowadays, the number of applications that follow it is increasing. Since a microservice-based architecture is made up of innumerable small components, naturally the complexity and the probability of the occurrence of system failures increase accordingly. Moreover, nowadays more and more software products are part of the daily life of ordinary people in the most diverse areas (personal telephone, commercial establishments, etc.) (res, 2019). In this way, the guarantee of the correct functioning of these services proves to be an increasing concern regarding the availability of these services for the longest possible time.

In this way, an application's resilience is its ability to deal with errors that may occur, while still maintaining its functionality as much as possible. The higher the level of resilience, the greater the solution's ability to make the user unaware of the error (res, 2019).

Based on the articles (Clay) and (pag), we can observe that the *Amazon* services' unavailability for just one minute represents a potential loss of 234,000 USD, which shows, in fact, the importance of availability nowadays. In addition, it will not be difficult to imagine the consequences of a failure in products that support very delicate institutions, such as hospitals, police forces, emergency services, or even air or rail traffic control.

In order to quantify this ability to deal with the occurrence of failures, there are already some metrics used for this purpose. These are MTTR (Mean Time to Recovery), MTTI (Mean Time to Identify) (mtt) and the number of failures and errors detected in the system.

Considering the aforementioned definition of resilience, in order to make the user unaware of the error as much as possible, it is first necessary to be able to detect this same failure. In a monolithic solution, the monitoring process is already a challenging task, let alone in a solution consisting of several instances to be monitored. In addition, if the solution is divided into several components, the means for establishing communication between these components will also have to be monitored since this component can fail, as mentioned in the seven fallacies of distributed systems (Rotem-Gal-Oz, 2008). All these efforts to monitor the constituent components of the solution allow not only to have an overview of the health status of the infrastructure as a whole but also to detect possible problems in the constituent components of the solution.

2.1.3 *Grey Literature*

During the process of gathering information on the resilience theme in a microservices-oriented architecture, we realize that the information found on the topic is mostly present on blogs, forums, or websites. Thus, since all this information is present in these less conventional media, naturally it will not be subject to any scientific verification necessary for the consideration of a credible reference. (Kamei et al., 2019) reinforce the fact that more and more non-scientific articles appear that address content related to Software Engineering. Furthermore, the authors reinforce that more and more researchers are showing interest in the content of non-scientific articles with the major objective of filling this gap.

Furthermore, having this theme of microservices aroused interest among the development community only around 2014/2015, when (Newman, 2015) was published, it is understandable that the verification and documentation process of information dispersed by unofficial documents has not yet been completed in full.

2.2 PROPOSED SOLUTIONS FOR RESILIENCE IN MICROSERVICES

During the research of contents on the theme of this dissertation, a strong gap was observed between the scientific aspect and the practical component. In particular, the presence of large amounts of information on this topic was observed but mostly scientifically unverified. This happens since this architectural style is still quite recent but mainly because the main drivers for its evolution are directly linked to development and not so much to scientific research.

Mostly, the proposed solutions to the problem arise in a format of architectural patterns to be applied. But, once again, the study (Taibi et al., 2018b), reveals the non-systematization of solutions that aim at increasing the resilience of a software product, among the patterns identified by the systematic study. However, only one was referred to as effective to increase the resilience of applications.

Furthermore, despite the observed gap, this theme has been gaining relevance not only among the development community but also among the research community, which causes the appearance of scientific articles dedicated to this topic.

As mentioned above, the problem to which we intend to respond is not yet fully established concerning documentation and scientific verification. In this way, the proposals to be presented have mainly two sources: *i)* scientific documents or books and *ii) blogs, posts on proprietary websites, forums, informal articles, etc.*.

Table 1 presents the content covered for each of the articles, making it possible to have an overview of which solutions are most addressed in the literature.

		Source														
		Livora (2016)	Hameed Addeen (2019)	Taibi et al. (2018a)	Balalaie et al. (2018)	Buss	Benhallouk	Gökalp	Rosner and Potukar	Gerard	Rocela	Schutta	Márton	Finnigan (2018)	Indrasiri	Behara
Pattern	**Circuit Breaker**	✔	✔		✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	Retry		✔				✔	✔	✔	✔	✔		✔			✔
	Timeout	✔	✔						✔	✔	✔				✔	✔
	Bulkheads	✔								✔			✔	✔	✔	
	Fallback							✔	✔		✔			✔		
	Cache		✔							✔			✔	✔		
	Fail Fast	✔											✔			
	Rate Limiting									✔			✔			
	Logging										✔					
	Client-side Discovery (Service Registry)			✔												
	Health Endpoint Monitoring						✔									
	Queue-based Load Levelling						✔									
	Throttling						✔									
	External Configuration System						✔									
	Compensating Transaction						✔									

Table 1: Representation of the patterns addressed in the analyzed articles.

It is still possible to observe that all six patterns mentioned most frequently are found in at least in one scientifically verified document, reinforcing the results obtained from the study of the literature.

Regarding the set of standards to be considered in the study, in a first selection, only those that were mentioned in at least one of the scientifically verified articles found on the theme were considered. Finally, among the solutions selected in the first selection, the six most mentioned standards were selected in both verified and unverified literature.

In the following paragraphs we describe each one of these patterns.

2.2.1 *Circuit Breaker*

Besides the component failure itself, another major concern in a microservice-based system are the components that depend on the component that has failed. This architectural pattern is intended precisely to prevent the failure from spreading to the rest. This pattern is called *Circuit Breaker* (Montesi and Weber, 2016).

This propagation is avoided by forwarding the requests destined to a specific service primarily to a component dedicated to the monitoring and analysis of its traffic, making it possible to detect symptoms of possible anomalies in the component.

More specifically, this pattern can be seen as a state machine, thus explaining the different possible states to follow. When this monitoring component is in the closed state (see Figure 1), requests are passed to the service, and any errors that eventually occur are recorded. The open state consists of blocking any request made to the service, and an error message is immediately returned. When in this state, it must be checked periodically so that this case regains its operational state and changes to a half-open state. Finally, the state half-open consists of passing only part of the requests to the service. If successive responses are returned, it will change to closed but, in the event of any error, the status will change to open.

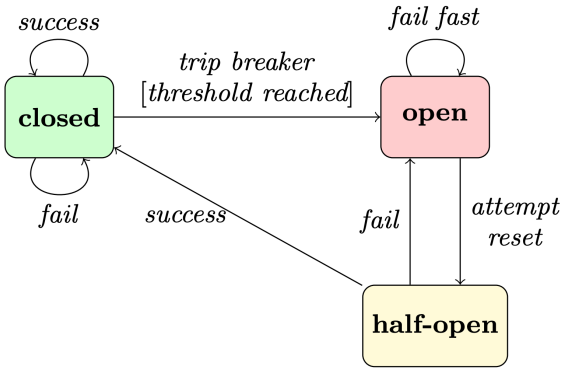


Figure 1: Circuit Breaker State Diagram. (Montesi and Weber, 2016)

2.2.2 *Retry*

As explained in Rosner and Potukar, this is a very simple pattern consisting of re-sending the request when it fails (represented in Figure 2). This resend can be configured according to the maximum number of attempts, as well as the time intervals between the various attempts. Thus, this pattern is suitable for resolving, for example, temporary network problems, occasional internal errors in the service, non-existent or too long response due to large amounts of traffic, etc.

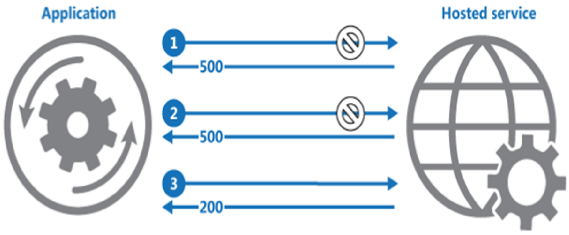


Figure 2: Retry Diagram. Rosner and Potukar

2.2.3 *Timeout*

This is an architectural pattern that allows you to interrupt a request to a service when an error is suspected. This suspicion usually translates into setting a maximum waiting time for an answer. It should also be noted that this waiting time must be adjusted, taking into account the context in which the pattern is applied. Indrasiri

2.2.4 *Bulkhead*

This is an architectural strategy very similar to that used in the construction of ships. A vessel consists of sections completely isolated from each other, thus preventing the propagation of any anomaly that occurs in that module to the adjacent ones.

In a microservice-oriented architecture, the concept is the same, thus allowing this pattern to limit the resources used by each of the components of the solution Wasson et al.. More specifically, the resources to be limited may be the number of competing requests for a component, thus preventing saturation of the network that interconnects it and consequently slowing down the rest of the infrastructure network as well. Also, the limitation of the resources used can be proved useful considering the overload of a particular component or module that shares physical resources will not affect the neighboring components or modules.

2.2.5 *Fallback*

This is a standard that implements the establishment of a pre-defined response to be sent in case an error occurs in the communication with the external service. This pre-defined message will replace the sending of the requested information, announcing the occurrence of some type of error that made impossible to get the desired information. In this way, the customer will always receive a timely response, even if pre-defined, thus avoiding longer waits that may occur because of an error in communication. Since this pre-defined message is informative of any communication anomaly, it facilitates service degradation. On the other hand, this mechanism performs a filtering of messages to be forwarded to the client. If the response returned by the service contains error messages or unexpected structures, the pre-defined message will also be sent to the customer. Here, the replacement of the message received from the external service promotes the isolation of the error from the external component, since it prevents the customer from processing messages for which he is not prepared.

Figure 3 illustrates a possible application of this *Fallback* Finnigan (2018) pattern, together with *Circuit Breaker* and *Bulkhead*.

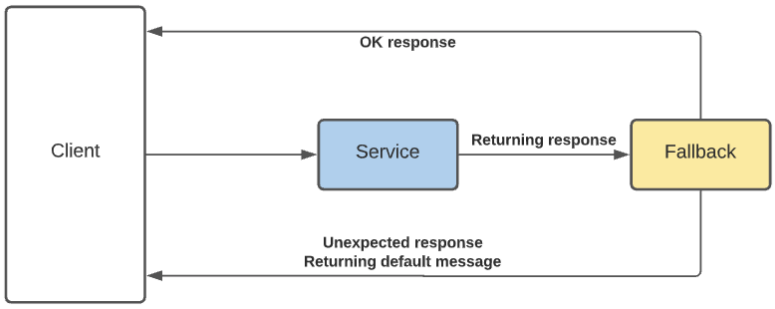


Figure 3: Diagram of representation of error handling using the *Fallback* pattern. Finnigan (2018)

2.2.6 *Cache*

Not being directly related to resilience but representing a determining factor for the occurrence of errors in the constituent components of a microservices-oriented solution will be the volume of traffic.

For this, as the service responds to requests, these responses can be saved so that in the future if the same information is requested again, it is possible to return the response without requiring the service. Thus, the load on the service is reduced, and therefore the probability of errors occurring Finnigan (2018).

A diagram representing the interconnection between the processing and cache components is shown in Figure 4.

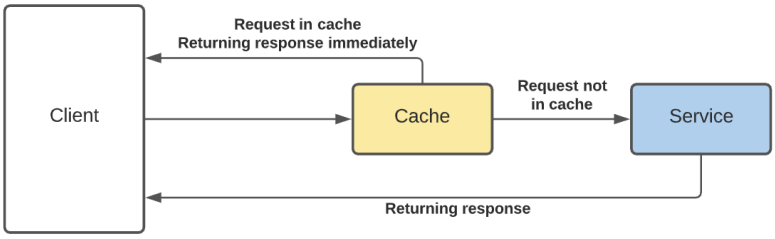


Figure 4: Representative diagram of the organization of the processing and *cache* components. Finnigan (2018)

3

PEOPLE TRANSPORTATION SYSTEM - A CASE STUDY

In order to enable the validation of the surveyed methodologies, it is essential to find a case study large enough, so that there are sufficient error scenarios that allow the application and corroboration of all the surveyed techniques.

This case study follows a microservices architectural pattern and its main objective is to monitor and forecast the arrival times of *TUB* buses *(Transportes Urbanos de Braga and Bosch Braga)* for each of the stops on the respective line. Users are also provided with the real-time location of buses in circulation by consulting a mobile application available for *Android* and *iOS*.

This solution is divided into two major groups, components of presentation (*in italics*) and backend (**in bold**).

- *Mobile App*
- *Tracker*
- *Backoffice*
- **API Gateway**
- **Routes**
- **Location history**
- **Real time location**
- **Arrival times**
- **Authentication**
- **Alerts**
- **Route Deviation**
- **Update Manager**

To make the resilience analysis more efficient and objective, only the components responsible for supporting the monitoring service (*Tracker, Routes, Location History and Real Time Location*) and forecasting arrival times (*Arrivals*) at the bus stops were target of analysis. In addition to the components directly related to the monitoring task, all the presentation components (*Mobile App and Backoffice*) were also analysed. This way, only the interactions directly related to functionalities available to the users were analysed. The diagram present in the Figure 5 represents all components targeted for resilience analysis, except for *API Gateway*, taking into account that this is only responsible for forwarding requests to the respective microservice.

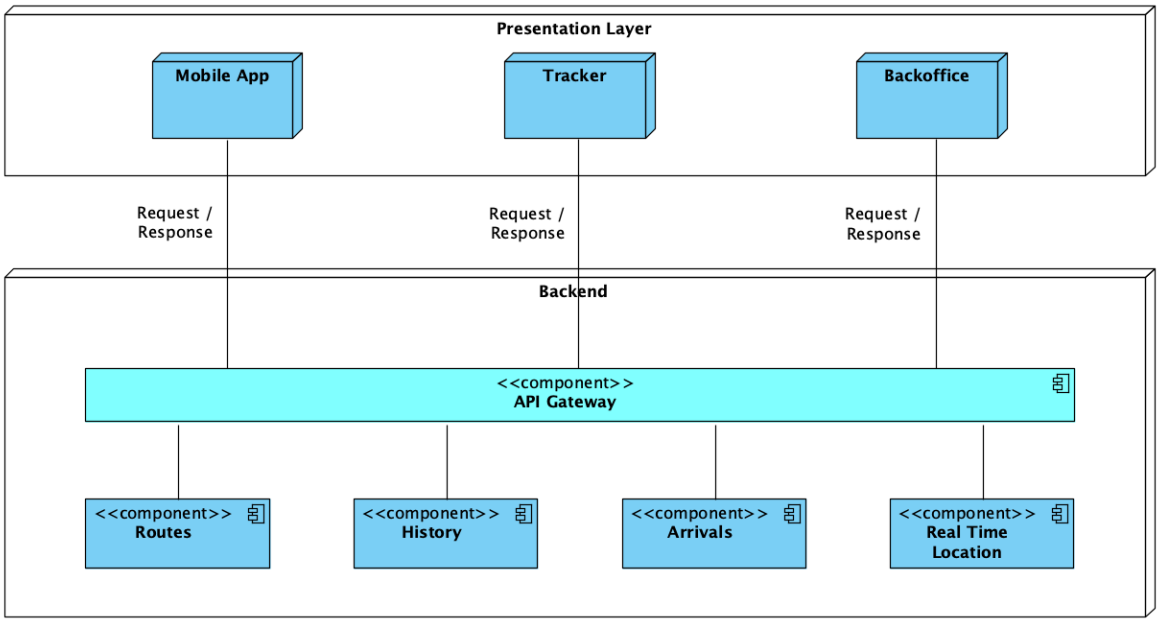


Figure 5: Solution architecture to be analyzed.

Naturally, the components of the *presentation* layer will be responsible for communication with the **backend** components. Therefore, for the solution resilience study, it is crucial to have a complete and detailed notion about the information sent by each of the presentation components as well as the communication flow of sending these requests. Finally, the content was also analyzed in terms of its importance for the correct functioning of the solution.

3.1 TRACKER

The Tracker component, is present in each of the monitored buses and is responsible for sending all the data relating to the buses in operation to the different **backend** components.

Within this Tracker component, there are several modules with different functionalities. The main module is called *geofencing* and it is where the event communication algorithm triggered by the movements of the buses is precisely defined.

As a result of the analysis made to this *geofencing* algorithm, we can find it in three different states of operation.

1. *getting_schedule*
2. *schedule_candidate*
3. *in_schedule*

When this device is started, the status [1 - *getting_schedule*], defined in the Figure 6, is immediately assumed, sending a request of the type GET to the service *Routes*, being requested the candidate schedules for the bus. When this same information is received, the operating state is immediately changed to the state [2 - *schedule_candidate*].

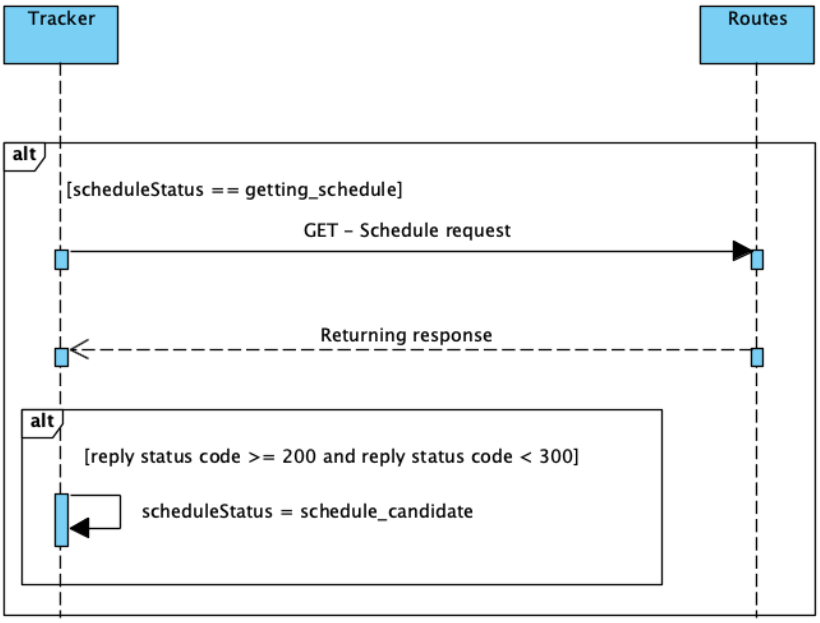


Figure 6: Getting schedule sequence diagram.

Once in the state [2 - *schedule_candidate*], the component will periodically check if the bus is within the radius of action of any stop that belongs to the route to be taken. When the bus position is within the radius of one of the programmed stops, a GET request to the *Routes (schedules)* service is sent, with data being requested regarding the route to be taken by the bus (line id, schedule id, direction and list of stops). Next, POST requests are sent to both the *Arrivals* and *History* service, thus informing the arrival of the bus at a stop on its

schedule through the isArrive: TRUE attribute. Finally, after detecting this first stop, the Tracker state is changed to the state [3 - *in_schedule*].

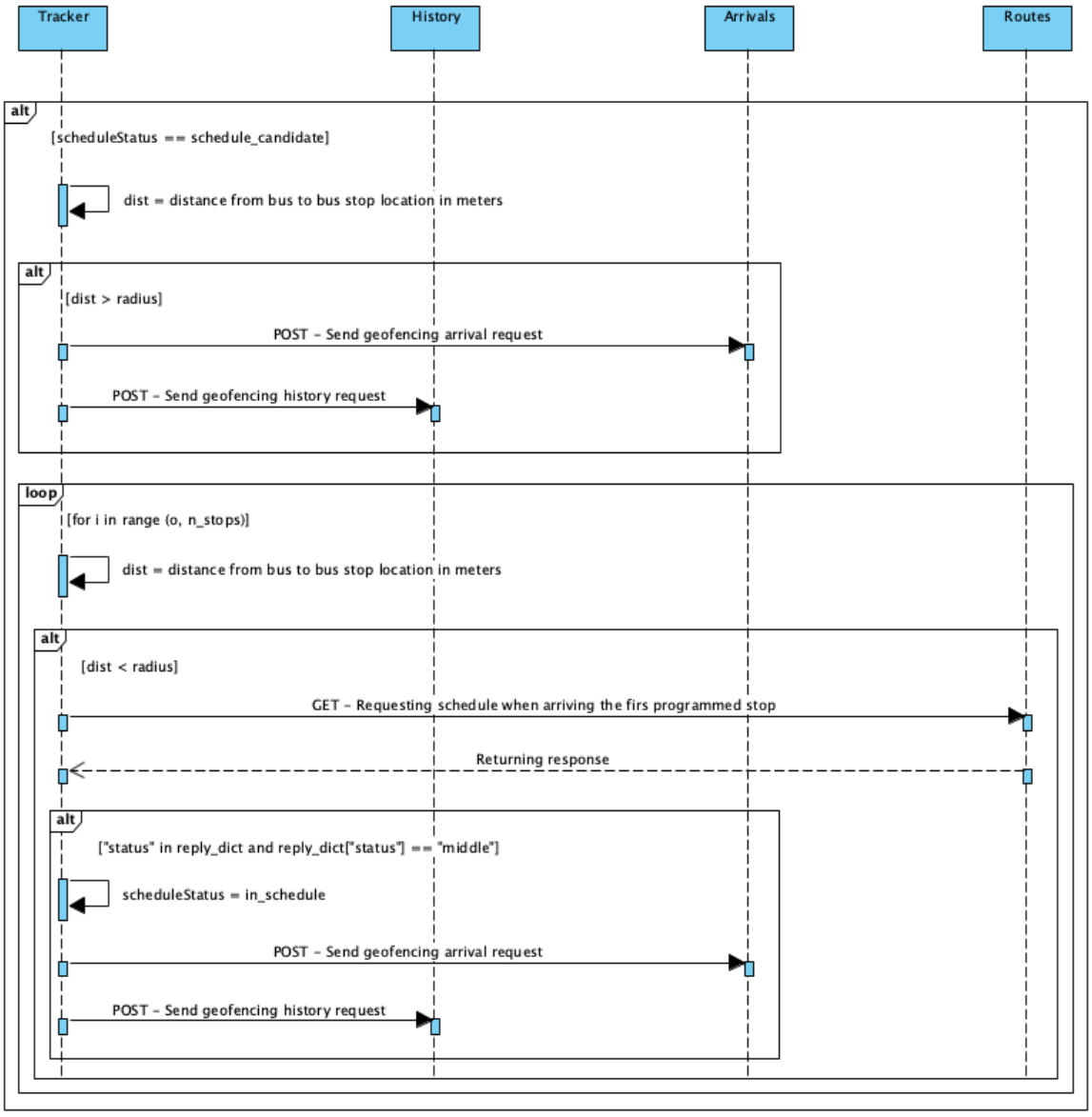


Figure 7: Schedule candidate sequence diagram.

This state [3 - *in_schedule*] is where the Tracker is expected to remain most of the time. The execution time in this state can be divided into two major moments, whether the bus is within the range of action of a stop or not. When within a bus stop range of action, it periodically checks to see if it is still close enough or no. At that moment, if it is not the last stop on its route, the departure will be communicated to the *Arrivals* and *History* services, sending the isArrive attribute marked as false.

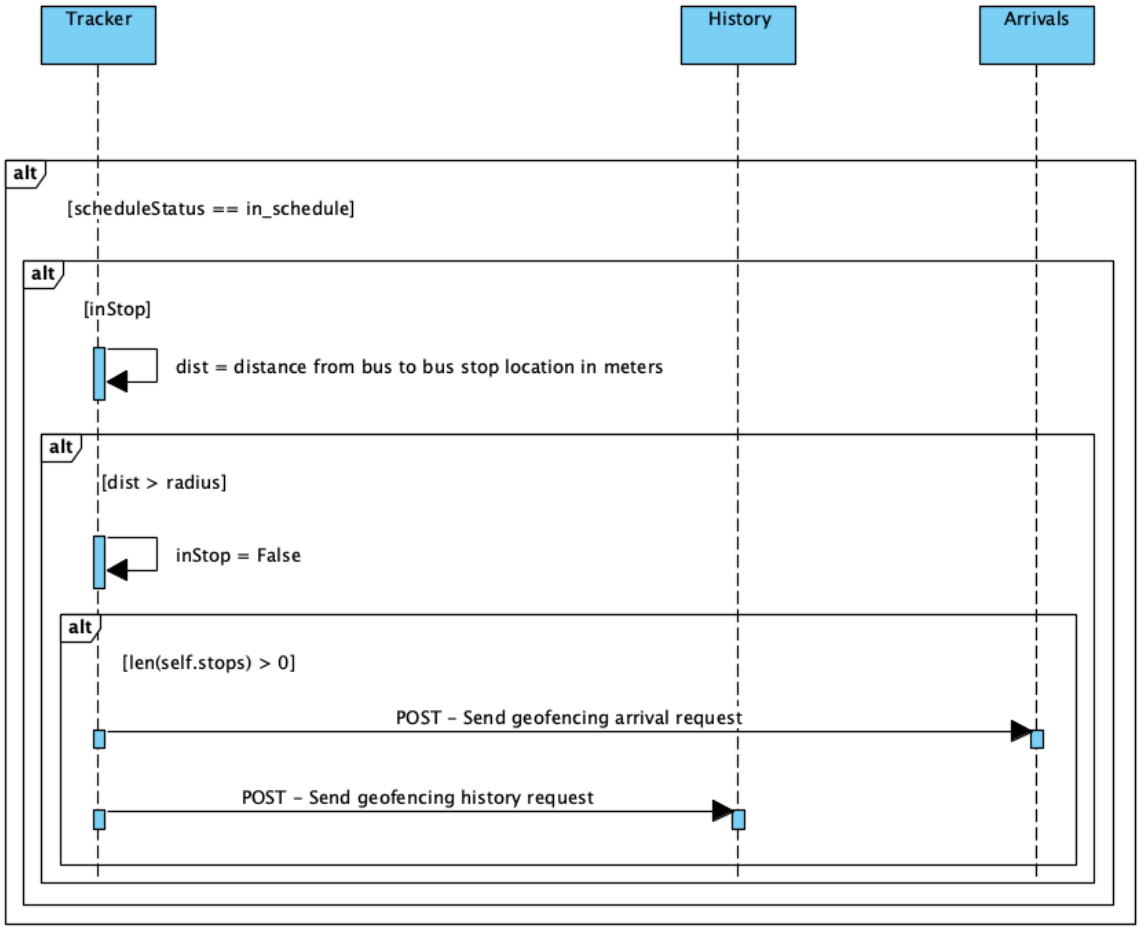


Figure 8: *Tracker* component checking if it is no longer in range of a stop.

When the bus is outside the stop range of action, it periodically check to see if it is close enough to one of the stops planned on its route. If this happens, the *Arrivals* and *History* services are notified through POST requests. It should be noted that this bus stop arrivals events requests sent have the isArrive attribute set to true. In addition, if the bus reaches the end of its route, the Tracker state is reset to the state [1], *getting_scheedule*. Finally, regardless of whether the bus is within the range of one of its stops or not, in each iteration, a POST request is sent to the *History* service to record the bus position at all times.

Besides the execution of this *geofencing* algorithm, there is also a constant registration of the current location of each bus for the non-persistent location service specified in Figure 10 sequence diagram.

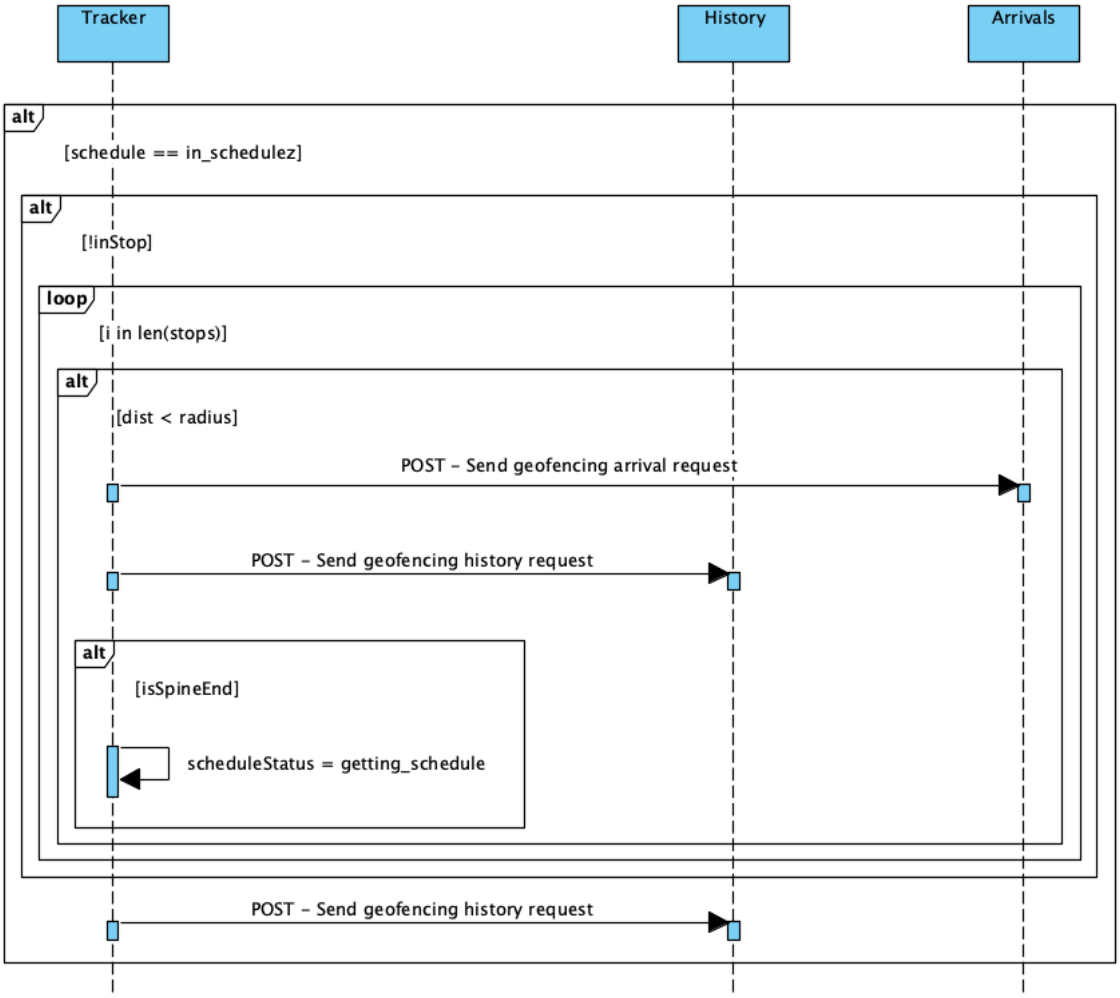


Figure 9: *Tracker* component in search of a stop within his range.

3.1.1 *Requests analysis*

Having already made the detailed analysis of the entire communication workflow for the *geofencing* implementation algorithm, it is necessary to reflect on how and if it is possible to effectively increase the resilience of the solution by applying the previously studied patterns. For this, we made a more detailed analysis regarding not only the relevance of the information that is exchanged between the client and backend layers, but also the context in which the request is sent as well as possible problems caused by the lack of the requested information.

As a result of this analysis, we found two different types of requests along the *geofencing* algorithm. The first group identified comprises GET requests triggered at the time of requesting a timetable for a bus either before it enters any of the stops or at the moment of entering the first stop of the route to be carried out. The information requested in these

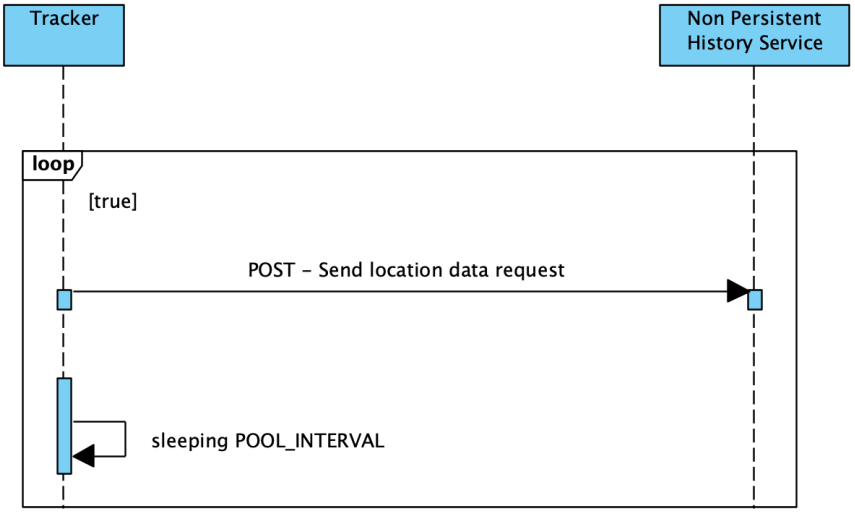


Figure 10: Publication of bus locations in real time.

requests is essential for the correct execution of the algorithm since the transitions of the 3 states previously mentioned are based on the information from these requests. Therefore, this information is vital for the functional correctness of the service. Possible faults in this communication have a high impact on service correctness.

Still, in this *geofencing* algorithm implementation, another requests group was identified, but now of the type POST. Furthermore, the sending of these requests is done following an asynchronous communication, using a pool of requests responsible for sending them when possible. In this case study, effectively an *acknowledgment* message is currently implemented, but this information is only saved in a logger, having no effect in the error handling. Thus, to increase the resilience of the solution in this scenario, it is necessary to prepare the **backend** components so that they can receive the maximum of information sent by the various *Tracker* components successfully. Information losses can be cause by the *backend* service unavailability or any fault in the communications process, as the events are not saved internally by the *Tracker*.

Figures 29 and 29 present in the appendix show the complete sequence diagrams for the *Tracker* interaction before and after patterns application.

3.1.2 *Error Analysis*

Firstly, it is necessary to understand which points of failure are present in the solution in order that the pattern's application can be effective.

For this, we made an analysis of the possible error scenarios and points of fault for each of the components that are involved in the execution of the *geofencing* algorithm.

In an initial phase, we identify which components were involved in the execution of the *geofencing* algorithm. Thus, the *Tracker, Arrivals, History* and finally *Routes* components were identified. After this work, we proceeded to the analysis of each of the execution contexts and, consequently, the identification of the error scenarios associated with each of them.

Facing a solution oriented to microservices, network problems can originate faults, considering that after all it is still a distributed system ((Dis, 2019)). Furthermore, taking into account that the *Tracker* component is distributed among the various buses operating in the city, this need for total mobility by this component implies the establishment of a connection to the support servers of the bus monitoring service through broadband solutions. This makes the connection even more unstable when compared to a connection on a local network.

Regarding the other components, they are all located in a common infrastructure where communication is expected to be established with a much higher level of reliability. We should also note that, since the solution does not require communication between these components to support the monitoring service, the probability of occurrence of network-related failures decreases considering that there is no need for communication between them.

Thus, the following error scenarios were identified:

- Lack of connectivity between the *Tracker* and the **backend**
- Errors processing messages of the component receiving the requests (backend)

Then, we will analyse the error scenarios previously identified and we will also propose solutions, considering factors such as the need for persistent information in circulation, its validity over time or even the performance/availability commitment of the service.

3.1.3 *Improving the resilience of components interacting with Tracker*

Arrivals component

The *Arrivals* component computes the arrival time estimates for each bus at the following stops. To enable the estimates calculations, it is also necessary to record all the bus stop entry and exit events. Only after this, it is possible to analyse the events data and produce the desired estimates. Naturally, it is intended that the accuracy of these estimates produced by the component are as high as possible, which implies the collection and recording of the largest amount of information possible.

In a scenario which the *Tracker* is unable to communicate with the *Arrivals* service support component, the only option to buffer the collected bus stops and exit events is use of the device's internal memory for storage. This way, even if communication is not possible, the

information will not be lost, which will allow its use in the calculation of future estimates. Regarding the recovery of information saved locally on the device, there are two different recovery strategies. The first consists of an attempt to forward information over the network when connectivity is restored. However, it will be necessary to consider the possible overload of the network when recovering connectivity and the problems that can be caused by this overload. In addition, if any problem occurs with the acknowledgment response that confirms the information reception, there is the possibility of resending an event that is already registered, so it is necessary to proceed with its disposal. The second option is the *Tracker* notify the monitoring support team that could not communicate and has information that need to be recovered. This strategy has the advantage of not overloading the network with the information previously collected, but it requires manual treatment, more specifically, the recovery and introduction of the events information saved by the *Tracker* locally. As in the previous mechanism, it is necessary to confirm that none of the events stored locally are not already registered to preserve data consistency.

Since in the interaction between these two components the information flow is mostly in the direction *Tracker - Arrivals*, the impossibility of communication or malfunction of the intervening components produces practically the same result, the non-registration of the bus stops entry and exit events. Thus, the whole analysis made for the first error scenario also applies to this one, with only a small behavioral difference. In a malfunction scenario of the component responsible for the events registration, the *Tracker* component is not protected against unexpected messages returned by the **backend** component. These messages outside the established standards can cause problems in the component's operation. In order to avoid this, this messages must be intercepted and replaced by a default one previously configured that the system can certainly interpret without errors. In this way, it is possible to avoid the error propagation from the service provider component to the client component. This functionality is supported by the *Fallback* pattern.

In this way, the proposed solution for this interaction results from integrating the following solutions.

- **Retry** - resubmission of requests against occasional failures in either the backend services or the network
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - as a preventive measure against traffic spikes than can lead to crashes or inconsistent component operation
- **Fallback** - to prepare the component against processing messages with unexpected structures or error messages.

Finally, given the nature of the *Bulkhead* solution, it requires an analysis not at the level of the interaction between components, but at the level of functionalities distribution among the different solution components. This analysis is present in 3.4.

The diagram present in the figure 11 represents the new behavior of the *Tracker - Arrivals* interaction.

History

The *History* component is another essential component in the execution of the *geofencing* algorithm. Being responsible for the persistent registration not only of the entrance and exit events of the buses being monitored but also of the respective locations at each moment, this component needs to handle high volumes of data. Furthermore, taking into account that the information stored by this component may be useful for studies on the performance of buses, the more information that is recorded, the clearer and more accurate the vision of the operation of the buses will be.

Considering the execution scenario, the similarities are clear with the interaction previously analyzed. The data to be recorded is practically the same, except that this component is additionally responsible for recording the locations at each moment of the buses. This additional registration of the locations only reinforces the need to implement the *Circuit Breaker* pattern as a measure to preserve the component's good functioning, given the enormous amount of information to be registered. Given the relevance of the information to be recorded, the same backup mechanism composed of the *Cache, Fallback, Timeout* and *Retry* (optional) patterns, proposed in the interaction with the *Arrivals* component should also be applied in this scenario.

Real Time Location

Bearing in mind that one of the solution's features will be the availability of locations at all times for bus users, for performance reasons, one of the components was developed specially for this same feature. In this context, the component register the current buses locations and make them available later to users. Since another component is already responsible for the persistence of these data, the locations are only kept in memory for a short period of time. This is due to the fact that locations are constantly being updated and only the most recent ones should be ready to be returned to the users.

Taking into account that the information processed by this component is considerably volatile, it is acceptable to sacrifice the information delivery guarantee in favour of performance. Regarding the *Circuit Breaker* pattern application, it is undoubtedly helpful when considering that promotes the proper functioning of the component by deviating requests if the component shows signs of malfunction. The application of a *Fallback* mechanism will undoubtedly be an asset, taking into account that it allows the configuration of predefined

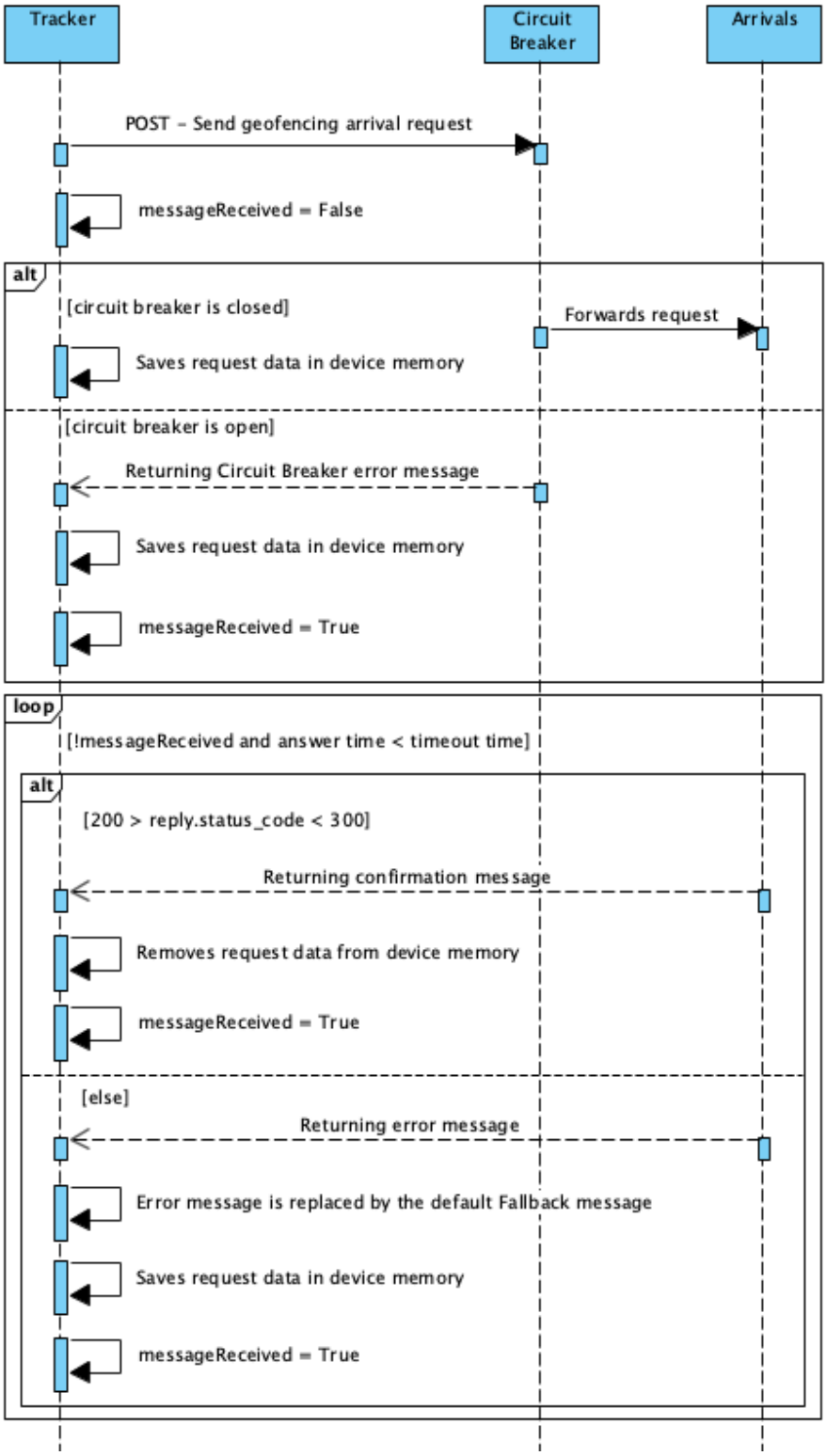


Figure 11: Sequence diagram for the *Arrivals* service and *Tracker* component.

messages in the event of an error occurring. In this way, not only is the task of logging facilitated, but it also prevents the occurrence of errors caused by the failure to send information. Regarding the *Timeout* pattern, this does not apply in this context, since the information is considerably volatile there is no point in implementing an acknowledgement mechanism. Also, due to the fact that in this interaction there is only information registration and not information request, the *Cache* pattern does not apply. Looking now at the rate of updates in the location information produced by the *Tracker* component, the application of the *Retry* pattern makes little sense. Implementing a mechanism of this kind becomes unnecessary when the rate of the information update to be sent is very high, meaning that by the time the information was resent, there would probably already exists more updated information for registration.

In this way, the proposed solution for this interaction results from integrating the following solutions.

- **Circuit Breaker** - as a preventive measure against the high volumes of data processed by this component
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.

Observe the sequence diagram in Figure 12 that represents the new interaction behavior.

Routes component

Finally, the *Routes* component is consulted to get routes information, which are essential for the *geofencing* algorithm evolution. Given the importance of this information, it is expected to propose the *Retry* pattern application. In practice, the *geofencing* algorithm typology itself already implements it. This is due to the fact that it is based on a state system and an infinite cycle. If the information is not correctly obtained, the status does not change, so in each iteration a new information request attempt is made. Regarding the *Circuit Breaker* pattern, it once again presents advantages, considering that its major function is to preserve the component's accurate functioning and the importance of obtaining this information for the algorithm evolution. *Cache* mechanisms can also be useful since they can ease the computational load of a component by reusing previously requested information. *Timeout* mechanisms can also be useful in this information request, since they avoid any excessive waiting for a service response. More specifically, this parameter must be adjusted according to the execution rhythm of the *geofencing* algorithm interactions. This way, resources consumption in waiting for outdated request can be avoided. Finally, once again, a *Fallback* mechanism proves to be useful in most cases, bearing in mind that it filters possible responses for which the system is not prepared, avoiding new errors. In this way, the error tends to stay contained to the original component.

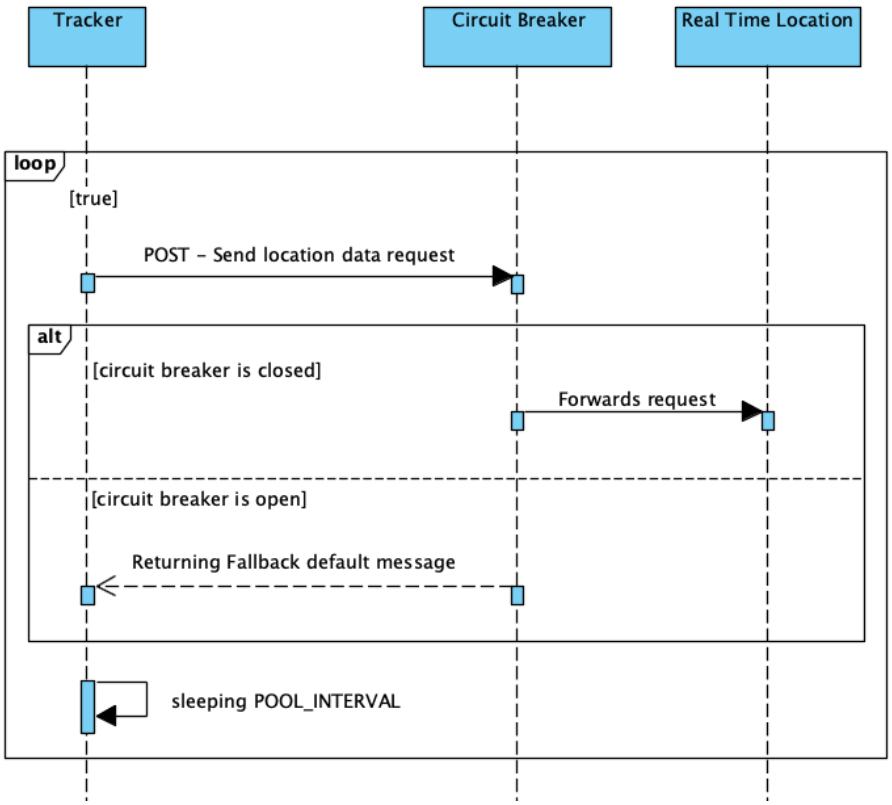


Figure 12: Sequence diagram for the *Real Time* service and *Tracker* component.

So, the proposed solution for this interaction results from integrating the following solutions.

- **Circuit Breaker** - to preserve the component's good functioning due to the information relevance served for the *geofencing* algorithm
- **Cache** - to reuse previously requested information, relieving the component computational load
- **Retry** (already implemented by default) - to try to obtain the necessary information as soon as possible for the algorithm evolution.
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.

The new behavior of the interaction is represented by the sequence diagram in Figure 13.

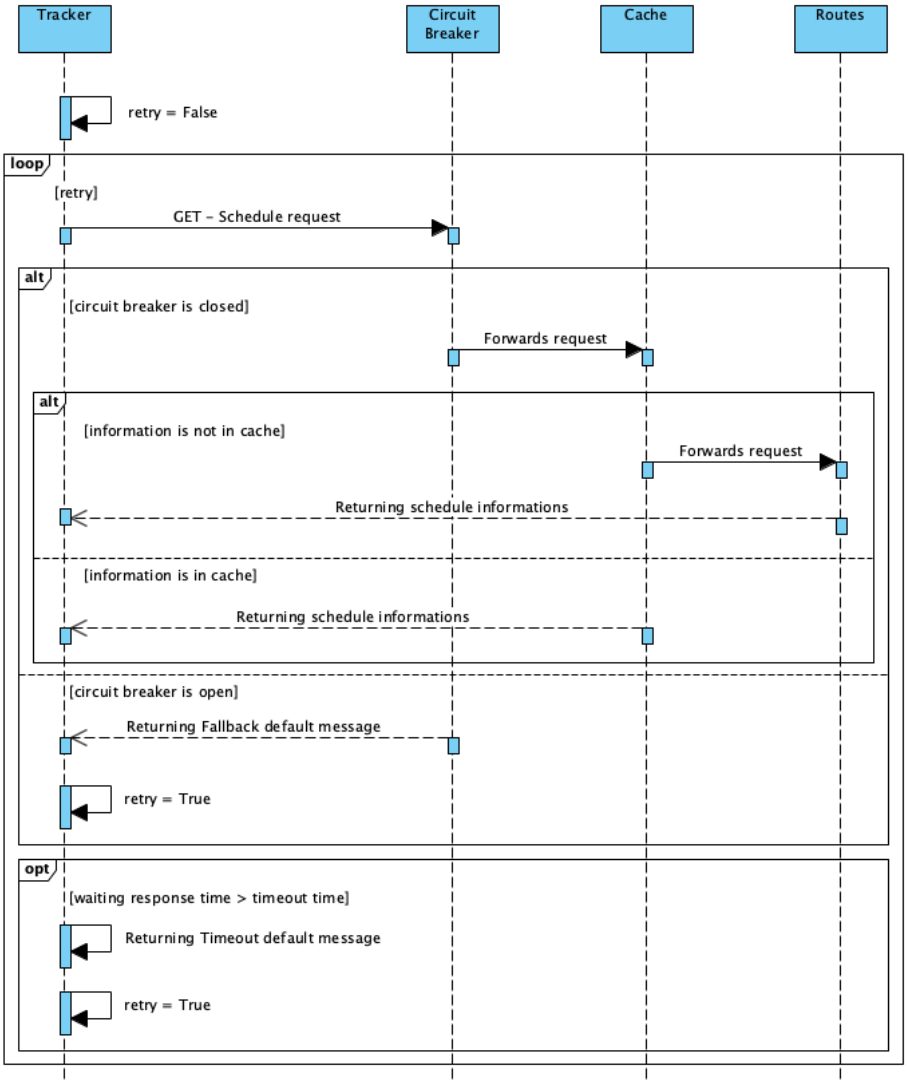


Figure 13: Sequence diagram for the *Routes* service and *Tracker* component.

3.2 BACKOFFICE (WEB APPLICATION)

As a complement to the *Tracker* component described in 3.1, a web application is used to make all the necessary configurations and have access to the data collected during the buses routes. Among the data collected is the in operation buses stop entry and exit events history as well as past buses location data and the stops arrival time estimates.

This locations monitoring task can be done according to two levels of detail, one based on a spine of stops [1] and one based on a map. Besides this monitoring task, it is also possible to consult the passage history of the buses being monitored within a certain time interval. Thus, in general, this web application is divided into three major areas, buses stop entry

1 representation of the bus route as shown in Figure 14

and exit events history (Figure 16), live buses location (Figure 15) and stops arrival time estimates (Figure 14). In addition to these three large areas of the platform, there is also an area at the top of the page common to these where some real-time information about the status of the buses being monitored is presented (highlighted in red).

This common area has as main objective to always show some basic information about the buses operation. Buses can be marked as *On time*, *Delayed*, *In advance* and finally *Empty*. In order to always keep this information up to date in the monitoring areas, a recurrent *GET* request is made to the *Arrivals* service which requests the information about the buses (Figure 17, (**I**)), more specifically which of them are meeting the schedule (state *In time*), delayed (state *Delayed*) or if arrived or left earlier than were supposed to (state *Advanced*).

In the first of the three main monitoring areas, a spine based representation of the buses locations is presented. This is also the default area when the users authenticates in the monitoring platform. The buses location is represented by the intervals between stops on this spine so, for a real-time representation of the location, it will only be necessary to have access to the events of arrival and departure from each of the buses stops. When a new line is added for monitoring, new requests start to be sent for buses that are in operation in this newly added line. The information is available in the *Arrivals* component, being requested through *GET* requests. As a complement, it is still possible to consult the complete history in and out events practised by the bus in the current route. To be able to show this data, two additional requests are sent for the services *Routes* and *Arrivals*. The first one requests the information regarding the route that the bus is taking (designation of stops, order of stops, etc.) and finally the second one requests the registration of the current round trip.

Figure 14: Representation of buses live monitoring as well as the stops arrival time estimates.

Another real-time monitoring zone is also available, but now based on a map. Now, it will be necessary to regularly obtain the positions of the buses so it is possible to keep the representation of the locations on the map updated. These locations are obtained through *GET* requests made in a loop to the *Real Time Location* service.

Finally, there is an area for consulting the history of timetables practised by buses. After filling in all the fields required for the search, the requested route information from the

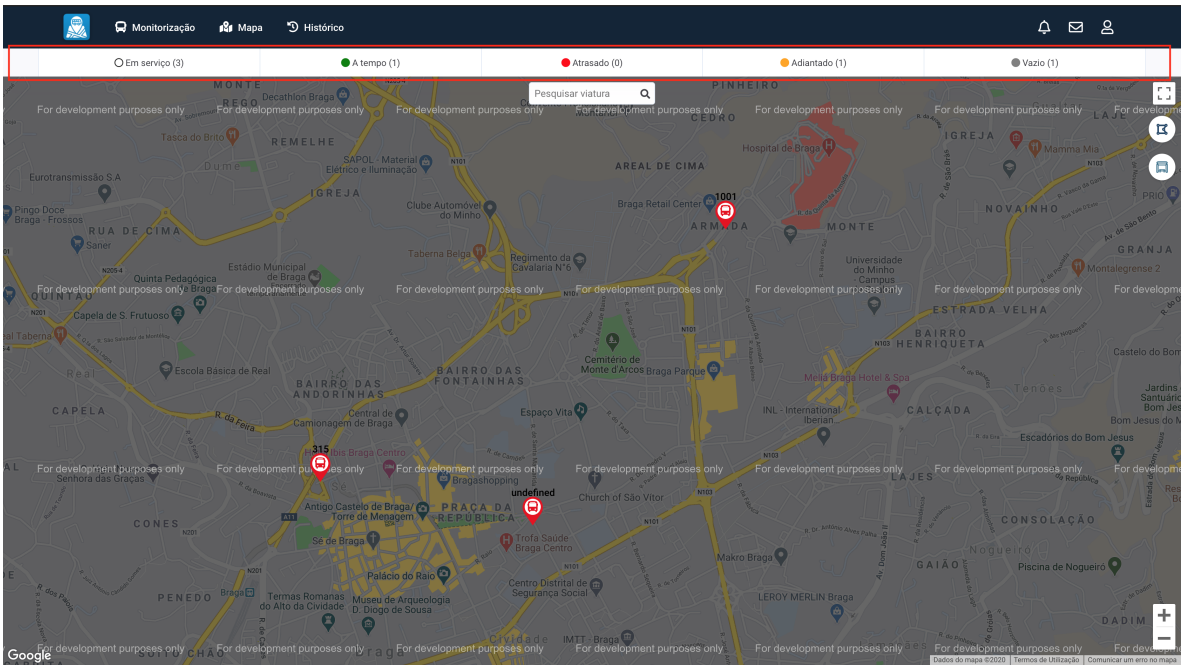
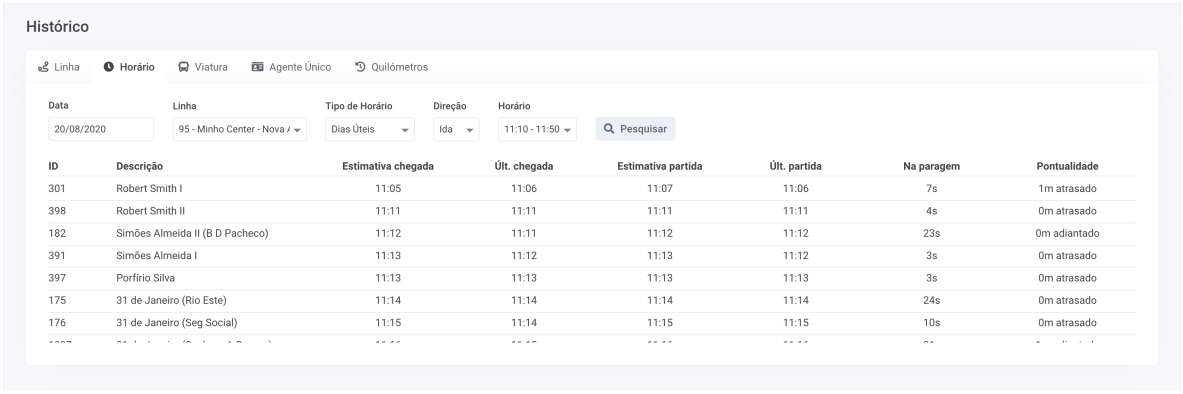


Figure 15: Real-time buses location representation.

Routes service and the times recorded in the selected time interval from the *Arrivals* service are requested. It should also be noted that all these requests are of the type *GET*.



ID	Descrição	Estimativa chegada	Últ. chegada	Estimativa partida	Últ. partida	Na paragem	Pontualidade
301	Robert Smith I	11:05	11:06	11:07	11:06	7s	1m atrasado
398	Robert Smith II	11:11	11:11	11:11	11:11	4s	0m atrasado
182	Simões Almeida II (B D Pacheco)	11:12	11:11	11:12	11:12	23s	0m adiantado
391	Simões Almeida I	11:13	11:12	11:13	11:12	3s	0m atrasado
397	Porfírio Silva	11:13	11:13	11:13	11:13	3s	0m atrasado
175	31 de Janeiro (Rio Este)	11:14	11:14	11:14	11:14	24s	0m atrasado
176	31 de Janeiro (Seg Social)	11:15	11:14	11:15	11:15	10s	0m atrasado

Figure 16: Representation of the history of bus stops.

3.2.1 *Requests analysis*

After the detailed analysis of the operation of this monitoring web application described in Figure 17, so that it is possible to choose the most efficient solution for increasing resilience in this area, it is necessary not only to consider the typology of requests triggered by this component but also the number of possible simultaneous users or not.

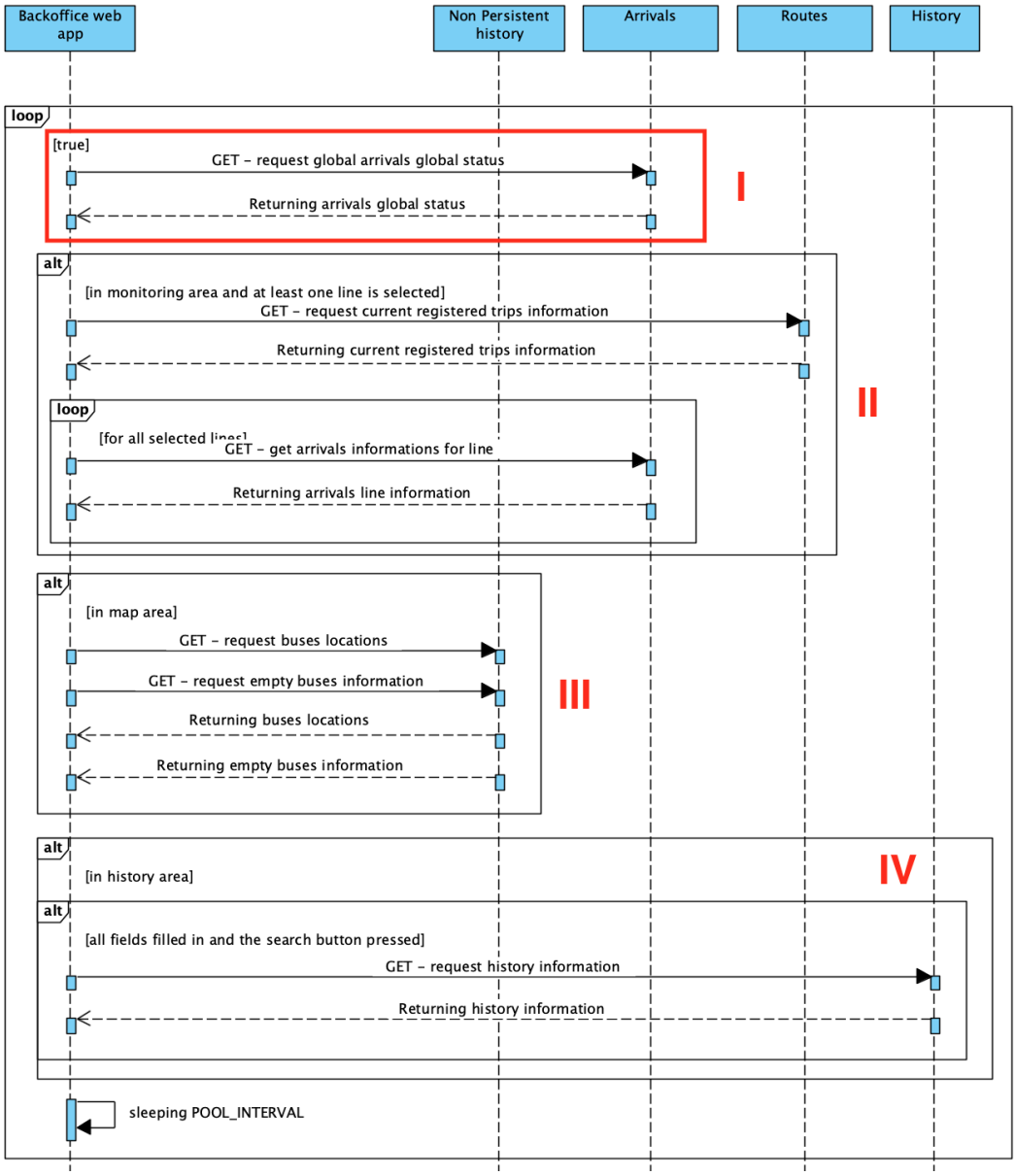


Figure 17: Backoffice sequence diagram.

Bearing in mind that the main areas of this web application have as their exclusive function the consultation of information regarding the practised schedules, current location and estimated arrival times of the buses under monitoring, the requests that support this component will be mostly of the *GET* type. In addition, the requests' typology also indicates that the information flow will be entirely from the **backend** components to the *frontend* component, which will be an important factor in choosing the solutions to be applied to increase resilience of the solution.

Contrary to what happened in the component analyzed in 3.1, where *GET* requests required essential information for the continuity of the operation of the *geofencing* algorithm and the information returned was calculated at runtime, now we found a different scenario.

This being a component associated with the consultation of bus monitoring data, there is now greater freedom to use a service degradation philosophy since the monitoring activity will not be affected in any way by the applied solution.

In this way, we only have a large group of requests of the type *GET* on analysis and of a lower priority level than those that were previously analyzed taking into account that the response to these is not decisive for the proper functioning of the monitoring component and it only affects the visualization of the information already collected.

3.2.2 *Error analysis*

Analogous to the *geofencing* component, in order to be able to increase the solution resilience level, it is essential to understand where failures can effectively occur so it is possible to implement mechanisms to solve them if they occur.

Regarding the solution typology to be oriented towards microservices, there is no difference in error scenarios from the 3.1 analysis, given the need for communication over the network which is not at all totally reliable.

In this way, the error scenarios identified above for the interaction between *Tracker* and the components supporting the service are the same:

- Lack of connectivity by the *Backoffice* component
- Errors processing messages of the components receiving requests (backend)

As it was possible to verify earlier, this monitoring component is supported by the backend *Arrivals*, *Routes* and *Real Time Location* services, so we will make the analysis of the two error scenarios for each of these support components.

3.2.3 *Improving the resilience of components interacting with Backoffice*

Arrivals

Regarding the *Arrivals* component, the interaction with it is based on consulting the buses general status information or specifically from one of the buses in operation (bus status and buses stop arrivals estimates).

When requesting general information about the buses' status (Figure 17, highlighted in red), considering that the information is constantly updated, any attempt to cache information would be a waste of resources. The only proposal for this interaction is the

application of the *Circuit Breaker* pattern. As this interaction consists of requesting the buses' global status at every second, it is necessary to prepare the component against traffic spikes.

When requesting the stops arrival times estimates (Figure 17, (**II**)), we propose the *Cache* pattern application. Although the information only remains valid until there is a new estimate calculation, any saved resource is important considering the computational load exercised in this component by the *Tracker*. Even following the large computational load to which this component is subjected, it would not make sense to apply a *Retry* solution, considering that the web application itself already deals with refreshing the information to be presented. Such a solution would only unnecessarily overload the service support component.

Since one of the players in this interaction is a presentation component, the performance factor is no longer the single highest priority. We also need to consider the platform usability. For this, the application of the *Fallback* pattern implements a protection mechanism against possible responses for which the system is not prepared. In this way, errors are avoided due to unexpected responses, and it is also possible to present an informational message to the user when it is necessary to resort to service degradation. We also propose the application of the *Timeout* pattern in situations where information is updated periodically. Setting the maximum waiting time to the information update time interval avoids waiting for requests for information that will not be used. Finally, it is also proposed to implement the *Circuit Breaker* pattern, taking into account not only the importance of the services provided by this component for the system as a whole but also as a preventive measure of overloads, considering that more up-to-date information is necessary to constantly send requests in search of new updates.

Therefore, the final solution consists of the following proposals.

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - to preserve the components good functioning due to the information relevance served for the *geofencing* algorithm

The new interaction behavior resulting from resilience patterns application is represented in Figures 18 and 19.

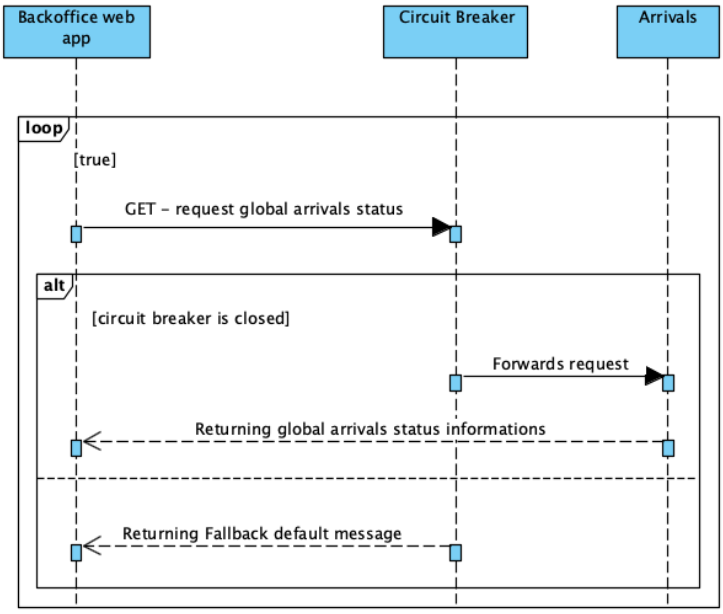


Figure 18: Diagram showing the new behavior in obtaining the general arrival status of buses.

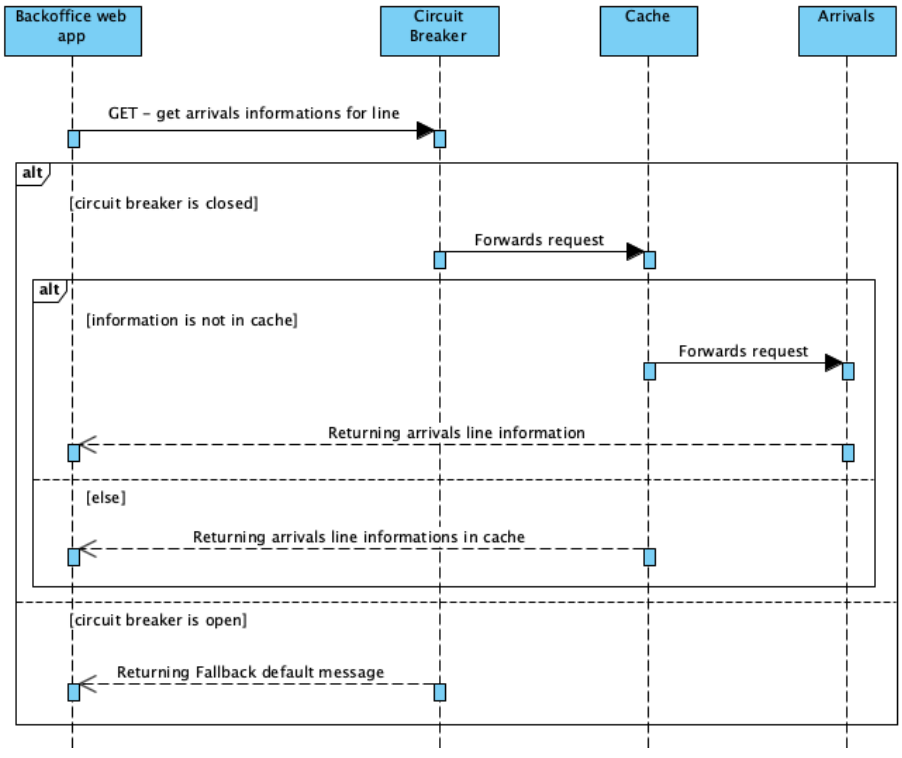


Figure 19: Representative diagram of the new behavior in obtaining information about a given line.

Routes

Moving now to the analysis of the *Routes* component (Figure 17, (**II**)), it is associated with the availability of all information related to the routes to be taken by the buses. Thus, it follows that this information will not be constantly updated, which represents a potential point of improvement. This low rate of information updates thus enhances the advantages of applying the *Cache* pattern, taking advantage of the information previously requested to be served again without having to reconsult the component. Additionally, in the event of component unavailability, if the requested information has already been requested recently, the failure will not affect the operation of the solution in any way.

Considering that this component is requested to get information, the *Timeout* pattern should be applied so that it is not possible to wait long periods of time for answers that maybe are already outdated. This way, it is possible to invalidate the request after a certain period, releasing resources earlier. On the other hand, this pattern also helps contain the fault to the original component by replacing any unexpected messages by predefined ones.

In the event of component failure, the solution should be protected against unexpected error messages. So, we propose the application of the *Fallback* pattern, in favour of containing the errors spread.

Considering that in this interaction the client is an information-presentation-component, it will not make sense to implement a *Retry* mechanism, taking into account that the request for information is not decisive for the smooth functioning of the solution but also because it could generate amounts of traffic unnecessarily.

Finally, considering that there will not be numerous users with access to this monitoring platform, and it is usually only necessary to request the route information for presentation on the page once, the complexity added by the *Circuit Breaker* pattern is not justified to the solution since this interaction does not prove to be heavy.

Therefore, the final solution consists of the following proposals.

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - to preserve the components good functioning due to the information relevance served for the *geofencing* algorithm

The diagram in the Figure 20 represents the new component behaviour.

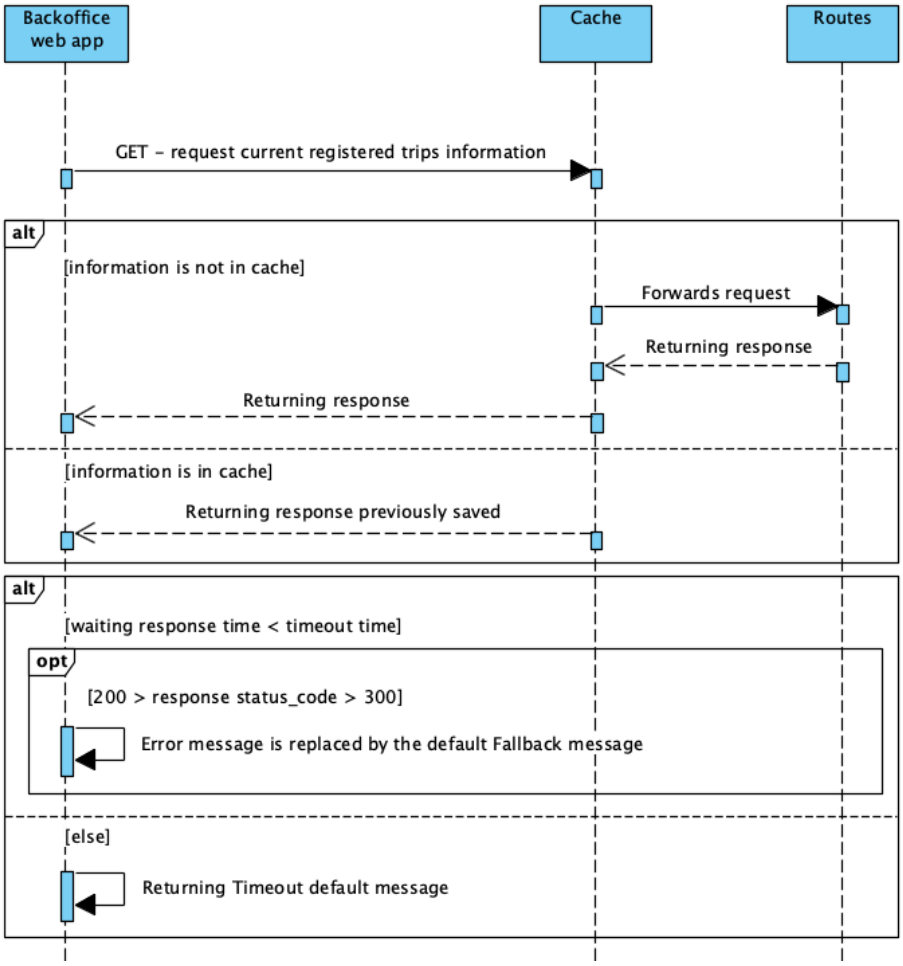


Figure 20: Diagram representing the new behavior in obtaining routes information.

History

This *History* component is similar in everything to the previously analyzed component. It also provides data on past bus entry and exit events. Thus, for the same reasons, it is also proposed to apply the *Cache*, *Fallback* and *Timeout* patterns.

As most of the information becomes static when registered (Figure 17, (**IV**)), the *Circuit Breaker* pattern is not proposed, considering that increases the interaction complexity level, when only a very small number of users will request information through the *Backoffice* web app.

It should be noted that, unlike the previous component, the information provided will always increase over time, which will require greater storage capacity of the cache mechanism to obtain satisfactory results. Another difference with the previous component is the fact that the information provided is never modified. This is due to the fact that it is information

related to events in the past which should not be changed under any circumstances. This non-change of data favors the effectiveness of the *Cache* solution.

This way, the final solution consists of the following proposals.

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.

Taking into account the similarity with the scenario described in 3.2.3, the diagram present in Figure 20 representing the new behavior is also referred to.

Real Time Location

Finally, the web application also interacts with the *Real Time Location* component (Figure 17, (**III**)). Once again, the type of information handled by this component is decisive for determining a strategy for increasing the resilience of the solution or not. Considering the volatility of the information that this component handles, it is not proposed to apply any solution to this context. This web application will only request the current locations of buses that are in operation. Considering that these locations are constantly updated, any temporary data storage effort would be in vain, thus invalidating the application of the *Cache* solution.

Once again, the combination of the *Fallback* and *Timeout* solutions promises better error handling capabilities.The *Fallback* pattern implements the protection against unexpected messages while the *Timeout* pattern excessive waiting times in case of no response is returned.

Any *Retry* mechanism will also be of little use in this interaction considering that the request for information is already made in a relatively short period of time, thus not justifying not only the complexity added to the solution but also the additional traffic generated. Once again, taking into account that only a few users will have access to this monitoring platform, the application of a *Circuit Breaker* solution for this interaction is also not justified.

Therefore, the final solution consists of the following proposals.

- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.

The diagram in Figure 21 is representative of the new behavior when obtaining buses locations.

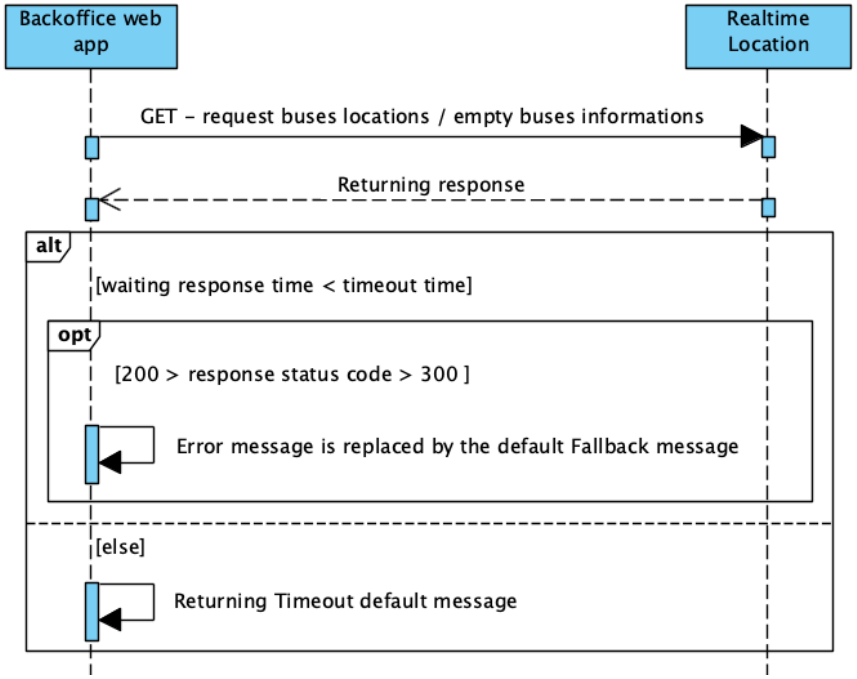


Figure 21: Representation of the new behavior in obtaining information regarding the location of buses.

3.3 MOBILE APP

Finally, the last major component of this solution is a mobile application. Through this, users have access to the defined timetables, the location of the buses that are in circulation and also an estimate of the time of arrival at the stops based on previous tickets.

For the presentation of all this information to the user, three large navigation areas are used. The application landing page, shown in Figure 22, presents information such as the lines and stops marked as favorite by the user and the estimate times for buses arrivals for the favorites lines and stops. It also allows access to more detailed information when selecting one of the displayed lines. This information consists of the schedules visualization defined by the company, a representation of all scheduled stops in spine format and finally an area where the stops locations are placed in a map as well as the real-time location of the buses that are currently in operation.

The second navigation area, shown in Figure 23, consists of the presentation of all the lines that the company has in operation. When selecting a line presented in this list, the

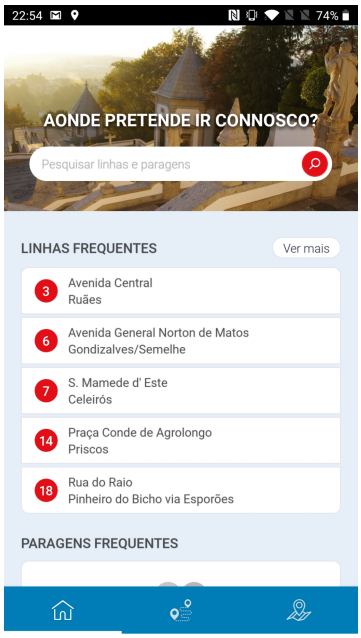


Figure 22: Mobile App landing page.

same information is shown when selecting a line in the landing page represented in Figure 22.

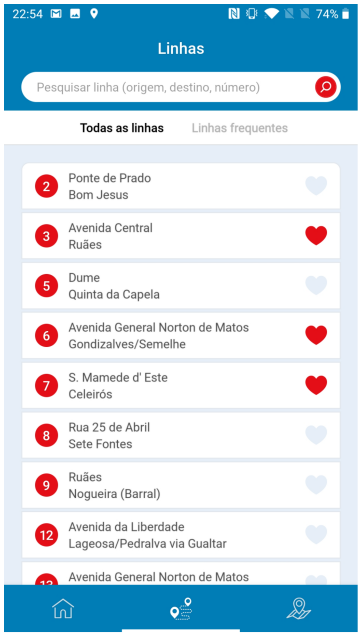


Figure 23: Mobile App lines list.

Finally, the third navigation zone, shown in Figure 24, consists of a map. Through this, the user has the option to search for places and get information about them, consult the stops and buses that are around him or the place searched within a radius of one kilometer.

The user can also get information about bus stops, such as the buses that are approaching it and the arriving remaining times expected.

Figure 24: Mobile App map.

Considering that except for the bus stops history and the global status of the buses in operation, the remaining information consumed by this component is the same as the one consumed by the *Backoffice* component. In this way, refer to the sequence diagram in Figure 17 to specify the component's information flow.

3.3.1 *Requests analysis*

Now having a better understanding of the mobile application scope, it is possible to study in detail its interactions with the support components. Like the component analyzed in 3.2, this mobile application has as main purpose making the information about buses in circulation available to users. Consequently, the requests generated will mostly be of the *GET* type, with the information flow predominating towards the supporting components for the mobile application.

This being a component of exclusive information consultation, some similarities are expected regarding the solutions proposed for the component previously analyzed. However, there are some determinant details in this component that may result in changes in the proposal for the final solution.

One of the factors that distinguishes this component from the one previously analyzed is the potential high number of users. If in the solution proposed previously, the implementa-

tion of cache mechanisms was proposed in order to avoid unnecessary accesses, this scenario further enhances its advantages considering that the probability of having requests for the same information is substantially higher. Despite the great promises of this type of solution, it is necessary to pay attention to some scenarios in which the results may not be as expected. See the case of the navigation area for displaying the map. Here, only information relating to buses and stops that are within a radius of one kilometer from the user's current location or the location searched for is displayed. This invalidates almost any result that may have been saved previously taking into account that data locations are usually quite dynamic.

3.3.2 *Error analysis*

Bearing in mind that this component of the solution is a mobile application with access to the storage resources of the device in which it is installed, there is the possibility of using these to improve the application's resilience.

In this way, the error scenarios initially identified during the analysis of the interaction of the *Tracker* component and backend services that support it are maintained. The following error scenarios then exist:

- Lack of connectivity of the *Mobile App* or failure to locate DNS services, errors in the packet forwarding by the ISP
- Errors processing messages of the components receiving requests (backend)

In this way, we will analyze the error scenarios taking into account the interactions between the *Mobile App* component and the support components of the *Routes, Real Time Location* service and finally *Arrivals*.

3.3.3 *Improving the resilience of components interacting with Mobile App*

Routes

Once again, given that the *Routes* component has the main function of providing information regarding the routes to be followed by buses, we propose the application of the *Cache* pattern since this information is not updated frequently. We can apply this caching mechanism not only at the *Routes* service level but also in the mobile application itself. The first case make it possible to reuse previously requested information for future requests, relieving the component load and speeding up the response time. As for the second case, once again the low update rate of this information makes it possible to reuse the most recent information that was obtained on a device, possibly omitting service failures and improving the user

experience. This reuse can not only be done in the event of a service failure but also adds a level of offline functionality to the application.

Another solution applicable in this scenario will be the *Fallback* pattern. This pattern implementation avoids the processing of unexpected or not well structured messages, allowing the fault to be contained in the original service.

As a complement to this last pattern, the *Timeout* pattern appears. The establishment of a maximum waiting time for the response of a given request prevents the occupation of resources with potentially invalidated requests, considering them as failed. We also propose this pattern as a mechanism for improving error handling in the mobile application.

Now, looking at the universe of users of this mobile application, there will be many users. Thus, it makes sense to protect the component against potential traffic spikes. For this protection, the application of the *Circuit Breaker* pattern is suggested.

Finally, and considering the potential high number of users, the application of the *Retry* pattern is not recommended. This is due to the fact there is not application scenario in which this mechanism clearly improves the operation of the application, but also by the fact that this is a solution that increases the number of requests by itself.

Therefore, the final solution consists of the following proposals.

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - to preserve the components good functioning due to the information relevance served for the *geofencing* algorithm

The sequence diagram shown in Figure 25 represents the new mobile application interaction behaviour when interacting with the *Routes* component.

Real Time Location

Considering that the information present in this service is constantly being updated, it makes sense to apply the *Timeout* pattern with the timeout parameter established in accordance with the data update rate. This way, it prevents long waits for information that probably are already out of date.

Still due to the constant updating of the data, implementing a *Retry* mechanism is not worth it given that new requests are constantly being triggered to always present the most updated location to the user.

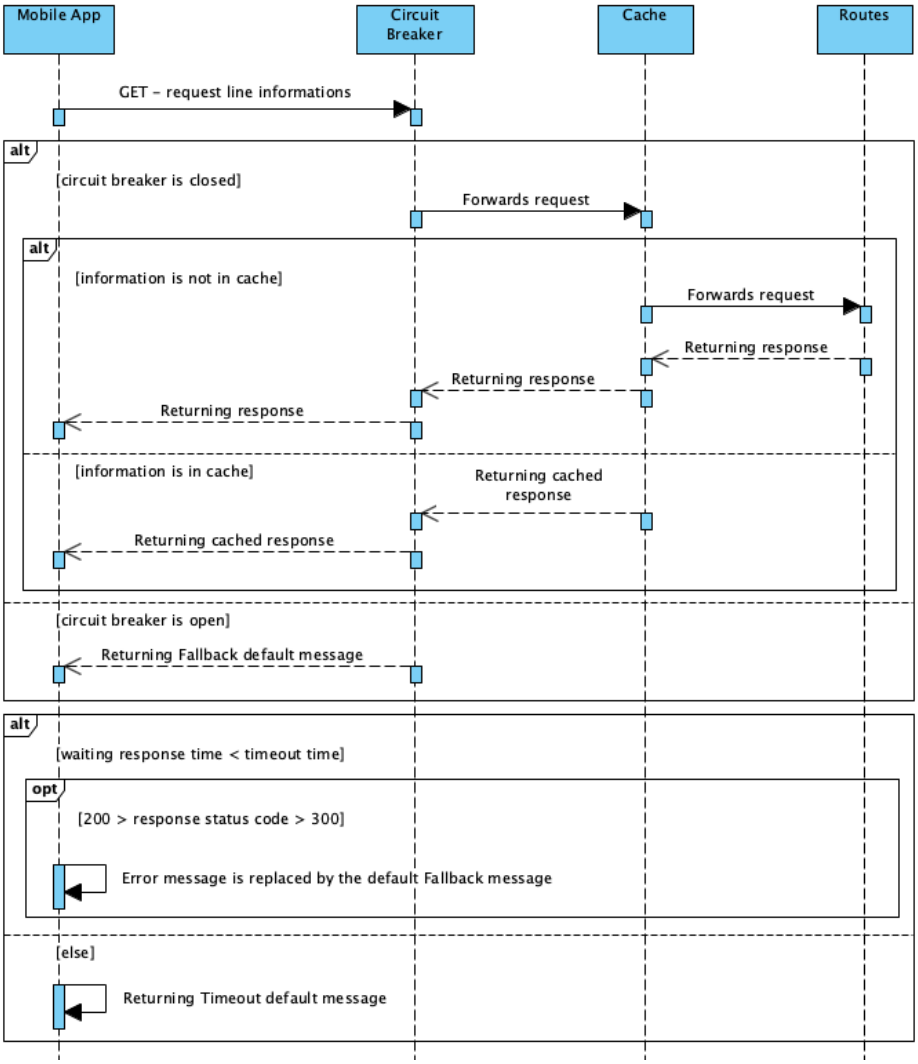


Figure 25: Representation of the new behavior in obtaining route information by the mobile application.

The *Fallback* pattern application brings once again a new dimension regarding the ability to handle errors that may occur, so we propose its application.

Still within the error handling, one other way to improve the user experience is to store the last location got, making it possible to at least present some result, even if out of date to the user. This functionality is implemented by the *Cache* pattern.

Finally, given the high number of possible users, we propose to implement the *Circuit Breaker* pattern, considering that its principal purpose is the preservation of the components' proper functioning.

Therefore, the final solution consists of the following proposals:

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - to preserve the components good functioning due to the information relevance served for the *geofencing* algorithm

The diagram present in the Figure 26 represents the new behavior in obtaining location information from the mobile application.

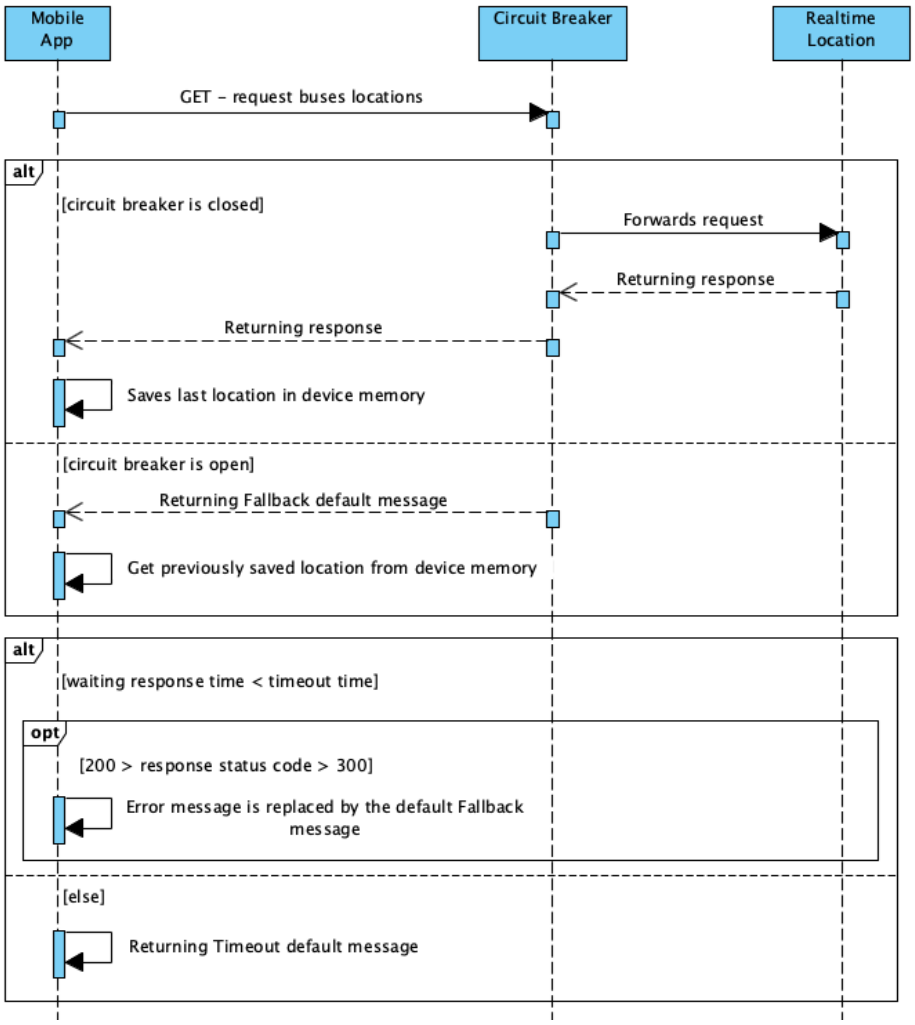


Figure 26: Representation of the new behavior in obtaining route information by the mobile application.

Arrivals

Finally, this mobile application only requests the stops of a route that is running and the estimates for the next arrivals of the buses to the stops, which is data that is constantly updated. When the mobile app does not have connectivity, the *Cache* pattern application could be proposed, but this would only have an effect in the time period between bus stops, considering that at each stop or departure event, the data is updated.

For requesting the next stop time estimates, the update rate is the same, considering that for each new incoming or outgoing event the forecasts are updated. Therefore, we propose the application of the *Cache* pattern, although aware that the scenario is not ideal to take advantage of 100% of the solution.

Furthermore, the implementation of an error message in case of service failure will be decisive considering that in this way the error handling in the presentation component becomes much more direct (*Fallback*).

As a complement to the error handling mechanism previously mentioned, we propose the implementation of the *Timeout* pattern, adjusted at most to the refreshing time of the information, thus avoiding waiting for responses to potentially outdated requests.

Bearing in mind that in any of the areas of this mobile application it is always essential to present the most recent information possible, consequently requests are constantly triggered for the most diverse information presented. Thus, *Retry* mechanisms will not be useful considering that the information is already being naturally refreshed, whether or not there is a failure, and also because this mechanism increases the traffic generated for the normal functioning of the application. Moreover, given the large number of possible users, the consequences are even worse.

Precisely due to this high number of possible users, we propose the application of the *Circuit Breaker* pattern, which combats possible traffic spikes and aims to preserve the components' proper functioning.

Therefore, the final solution consists of the following proposals:

- **Cache** - to reuse previously requested information, relieving the component computational load
- **Fallback** - as a preventive measure against possible unexpected messages, helping to contain errors.
- **Timeout** - establishment of the maximum waiting time for the acknowledgement message.
- **Circuit Breaker** - to preserve the components good functioning due to the information relevance served for the *geofencing* algorithm

Given the similarity of this mobile application to the backoffice component, despite the reasons why the resilience solutions proposed are different, the final proposal remains, referring to the diagram in Figure 19 for the representation of the new interaction behaviour.

3.4 ARCHITECTURAL ANALYSIS

As previously identified, one of the solutions raised as a proposal for increasing resilience requires analysis at a different level from the other resilience solutions. More specifically, the *Bulkhead* strategy requires an analysis at the solution architectural level, considering that it promotes the allocation of resource usage limits to certain features provided by the component. Excessive consumption of these computational resources can happen for the overload of a certain functionality provided by the microservice, and also because of the occurrence of some type of error that causes the consumption of resources excessively. It is essential to have a clear idea of what features of what features are available for each of the constituent solution components is essential. Also it should be noted that the two major tasks to be performed by this component are the registration of all bus entry and exit events at the respective stops and the calculation of estimated arrival times to the following stops. Additionally, given the strong relationship between the data involved in these two major tasks, they are supported by the same solution component (*Arrivals*), which may reveal a problem due to the computational load of each of them. Although at the moment, the calculation of arrival time estimates is made using an average of the arrival times of the last thirty days, this is still a demanding task given the amount of information to process. This proposal also appears as a preventive measure for a future change of this calculation method to one that presents a higher degree of precision and will also probably be heavier to the hardware. This way, any interferences due to scarcity of resources in any of the services are prevented, whether this is caused at a time when the calculation of arrival time estimates is in progress, or by peak traffic in the registration of buses entering and leaving events in the stops. Finally, the amount of resources allocated to each of the features will depend very much on the resources made available as well as on the desired prioritization regarding the features in question.

This technique is usually implemented using pools of requests dedicated to each of the functionalities provided by the component. In this way, it is possible to limit the requests number that are accepted simultaineously. If a lower level control is needed (closer to the hardware), the resources of service orchestration frameworks, such as Kubernetes (kub), can be used.

To conclude, considering not only the importance of the good functioning of the *Arrivals* component for the general functioning of the solution but also the computational load of the

features supported by this component, we propose the application of the *Bulkhead* pattern in the *Arrivals* component.

3.5 Implementation decisions

Since microservice-oriented systems are very similar to distributed systems, which means that the solution consists of several components implemented in different locations, it is necessary to understand where to implement the resilience solutions so that they achieve the highest level of effectiveness.

We then proceeded to research this topic on scientifically verified platforms. Once again, due to the fact that this topic is not yet properly established scientifically, we found no relevant information on the issue. In this way, we carried out a study of potential advantages and disadvantages, either from a *client-side* or *server-side* approach.

During the more detailed research on these two strategies, we concluded that with the application of a *server-side* resilience solution, the point of failure would only pass from the component providing the service to the resilience solution itself, considering that it would now be responsible for component traffic. However optimized this component may be, it still represents a single point of failure, which is not at all beneficial to any software solution.

Regarding a *client-side* application strategy, this will no longer just move the point of failure point considering that in this way, the techniques for increasing the resilience of the solution are applied at the level of the component originating the orders. In addition, the information necessary for the operation of the resilience solutions is stored directly in the client, thus not being dependent on any other means of communication. While carrying out this study of advantages and disadvantages, we found a library developed by *Netflix* dedicated to increasing the resilience of distributed systems called Netflix (2018). This library has the major function of involving the call of any external service to the component in an *hystrix* object. Furthermore, the entire implementation of the resilience enhancement mechanism is implemented within this same *hystrix* object. This promotes code modularity and makes the implementation of the resilience solution not imply any change in the business layer. However, considering the resilience solutions that were raised in the study described in the section 2.2, all of these agree with this methodology except for two, *Bulkhead* and *Cache*. For the first, considering that it is not a solution that acts at the level of interaction between components but at the architectural level, it does not integrate into any of these strategies. Regarding the second resilience solution, it will not follow the same strategy as the other solutions. The primary function of *Cache* mechanisms is to reuse previously requested information. Through this information reuse, accesses to the support component of the service are saved while also reducing the response time where the information has already been previously calculated. If this mechanism follows a *client-side* strategy, it will be

extremely difficult to manage all these small data sets, from their update to their disposal. In addition, we do not expect that a single user will make a general and recurring use of all the functionalities of the service. This means that at the time of requesting information, the information stored would practically never be up to date. Thus, the application of the *Cache* solution proposal should follow a *server-side* strategy, therefore taking advantage of the interactions of multiple users and also facilitating the actions of discarding and updating the information stored by this resilience solution.

In this way, we propose the implementation of the resilience solutions according to the strategies *client-side* and *server-side* as seen in the following installation diagram.

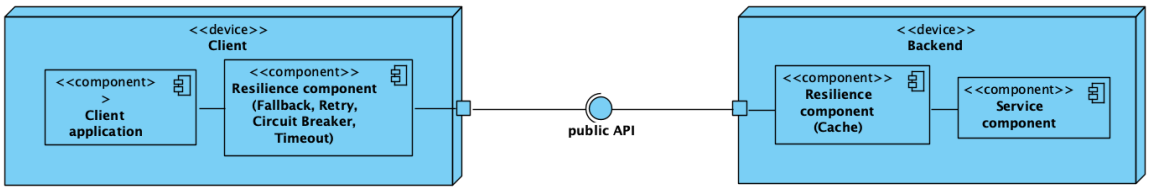


Figure 27: Diagram of installation of resilience solutions.

4

METHODOLOGY

Since the final objective of this study is to create a methodology applicable to any microservices oriented solution, it is essential to understand why and when each of the resilience patterns should be applied.

In order to better understand the execution scenarios in which we propose each of the resilience solutions, it is necessary to consider all analyzes of each of the case study interactions. In this way, it is possible to understand the interaction characteristics that motivate each of the proposed resilience solutions.

From the analysis of all the case study interactions, it is possible to extract the key factors from Table 2 to be integrated in the methodology specification.

Key point	Description
Presentation Component	If one of the intervenient components is in the interactions is of the presentation type.
Request type	If the interaction consists of getting or posting information to the backend services.
Information validity	How often the information involved in the interaction is updated.
Importance of safeguarding information in circulation	How important is the information to the system. Defines whether the priority is that no information is lost or performance.
Number of interactions	The number of communications in a certain period of time.
Computational load versus potential number of clients	The computational load considering the task complexity and the number of potential clients.

Table 2: Characterizers used in the analysis of the case study interactions.

4.1 RESULTS ANALYSIS

Table 3 represents all the case study interactions defined according to the Table 2 characterizers.

	Presentation component	Information flow direction	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computational load versus potential number of clients	Solution Proposal
Tracker - Routes	No	Get	High	Not important	Low	Low /Considerable number of users	Retry, Timeout, Cache, Fallback, Circuit Breaker
Tracker - Arrivals	No	Post	High	Important	High	High /Considerable number of users	Retry (opcional), Timeout, Circuit Breaker, Fallback, Backup
Tracker - RTL [1]	No	Post	Instant	No Important	Very high	High /Considerable number of users	Circuit Breaker, Fallback
Tracker - History	No	Post	High	Important	High	High / Considerable number of users	Retry (opcional), Timeout, Circuit Breaker, Fallback, Backup
BO [2] - Routes	Yes	Get	High	Not important	Low	Low / few users	Cache, Timeout, Fallback
BO - Arrivals	Yes	Get	Intermediate	Not important	High	High / few users	Cache, Fallback, Timeout, Circuit Breaker
BO - RTL	Yes	Get	Instant	Not important	Very high	Low / few users	Timeout, Fallback
BO - History	Yes	Get	High	Not important	High	Low / few users	Cache, Timeout, Fallback
APP [3] - Routes	Yes	Get	High	Not important	Low	Low / many users	Cache, Timeout, Fallback, Circuit Breaker
APP - Arrivals	Yes	Get	Intermediate	Not important	High	High / many users	Cache, Fallback, Timeout, Circuit Breaker
APP - RTL	Yes	Get	Instant	Not important	Very high	Low / many users	Timeout, Fallback, Cache, Circuit Breaker

Table 3: Solution proposals for the interactions present in the case study.

Based on this new characterization method for the interactions identified in the case study, it is possible to identify that the interaction pair *Tracker - Arrivals*, *Tracker - History* and *Backoffice - Routes*, *Backoffice - History* have similar characteristics. As a result, one of each of these interactions pairs will be excluded, as it does not add any additional information to the study.

Regarding the involvement or not of a presentation component in the interaction, preparing the interaction components to handle the error can be divided into two major groups - those that involve presentation components and those that do not. This division happens because a presentation component does not have a well-defined line of execution of actions, but rather depends on user inputs. The type of request sent is also a determining factor when preparing a component to deal with faults. Bearing in mind that when dealing with both publishing and getting operations, naturally the resilience techniques will differ.

Another of the decisive factors for choosing the best solution to increase resilience is the information validity over time. Through this factor, it is possible to understand if the reuse of previously requested information is a viable option or not to deliver correct functionality. We should also note that only operations that update existing information affect the information's validity. New information registration operations typically do not affect the validity of the information.

1 Real Time Location
2 Backoffice
3 Mobile Application

The importance of information in circulation is another factor to consider, as it requires the implementation of mechanisms to safeguard the information in case of error. The proper functioning of these mechanisms directly affects the effectiveness of the solution as a whole, since they deal with information essential for the proper functioning of the solution.

Finally, the relationship between the computational load required by the actions and the number of requests that need to be processed is also decisive for the application or not of protection measures against system overloads. In addition, the potential number of users will also be a factor to consider considering that the higher the number of potential users, the greater the requests processing capacity required.

Through the characterizing factors identified in table X and the analysis of each of the interactions in the case study, it is now possible to associate each type of interaction found with a set of proposed solutions for the case study.

4.1.1 *Analysis of the case study generalizability*

The interactions identified in the case study were also analyzed to verify if they represent most of the interaction types present in software products. This generalization of capabilities is essential to produce a real and effective methodology.

The first point analyzed was the requests type sent. The case study presents both information registration operations (typically identified as *POST or UPDATE* requests) and information request operations (*GET* requests).

In addition, we found interactions with different types of validity (high, intermediate and instantaneous) regarding information in circulation. This allows better understanding of when and why the resilience patterns should be applied.

In any software product considered distributed, there is still a need for constant communication between the components, so information is constantly in circulation. Considering the possible failures that may occur during communication, it is necessary to determine the consequences of their loss. In this case study, we can find interactions where the information in very important for the solution operation as well as interactions where the information is constantly being updated. This constant updates make this information of low importance. This diversity of interactions reveals the wide scope of the case study in terms of information validity.

Finally, the computational load of the interactions in the components was also analyzed, considering the potential number of users. Once again, the case study revealed interactions of both high and low complexity for sets of users in both large and limited numbers.

4.2 METHODOLOGY

As a result of the analysis of the interactions present in the case study and respective resilience solutions proposals, next we present the conclusions obtained about each of the resilience solutions. We characterize each of these according to its intention, problem that it solves and which solution it implements.

- **Circuit Breaker**

  Problem: When a component is up and running but overloaded, there is a greater probability of errors occurring.

  Solution: This pattern implements a mechanism to evaluate the component's functioning. If successive errors occur, traffic will be diverted in an attempt to recover the component's normal functioning.

  Compromises: The application of this pattern requires an additional complexity of the mechanism for evaluating the functionality of the component. This mechanism involves the analysis of all incoming and outgoing traffic of the component to be monitored.

- **Retry**

  Problem: When some information of high importance cannot be obtained. The application of this pattern tries to solve occasional failures of the external service.

  Solution: This pattern implements the successive sending of requests in an attempt to obtain the desired information.

  Compromises: The application of this pattern implies a slight increase in the number of requests sent. In a situation where the failure of the external service is not momentary, these additional attempts to obtain information can contribute even more to the total failure of this external service, if the time intervals between successive attempts are poorly configured.

- **Timeout**

  Problem: Failure to obtain a timely response from an external service.

  Solution: This pattern implements the definition of maximum waiting time for a request. If this time is exceeded, the request is considered unsuccessful.

  Compromises: The defined waiting time must be adjusted taking into account the complexity of the task to be performed. If this time is too short, most requests will be considered as failed and when it is too long, the promised effects of the pattern will be compromised.

- **Bulkhead**

  Problem: When a component of the solution houses one or more different complex functionalities.

  Solution: This solution consists of implementing resource usage limits for each of the features.

  Compromises: This division of resources is complex to implement and manage.

- **Fallback**

  Problem: Errors in any of the components of a solution are inevitable. Thus, it is necessary to contain this error and prevent it from affecting the operation of the other components.

  Solution: So that the error does not spread, this pattern implements a mechanism to replace any unexpected message with a predefined one, thus avoiding errors originating in the processing of unexpected messages.

  Compromises: The implementation of this pattern requires all traffic to pass through this check.

- **Cache**

  Problem: When the component that provides some information is not in operation at the moment.

  Solution: This pattern implements the reuse of previously obtained information that is still valid. In this way, it is possible to omit a service failure as well as to relieve the component load.

  Compromises: The entire management of this cache mechanism is highly complex. The invalidation of the entries in memory or the registration of new entries at the right times is crucial for the smooth functioning of the solution. Furthermore, the implementation of this pattern still requires extra hardware for caching records.

- **Backup**

  Problem: The information present in the requests is of high importance and must be preserved to the maximum.

  Solution: This pattern implements a mechanism for safeguarding request information based on receiving feedback from the recipient. Until the confirmation of receipt is received, the information is saved in memory.

  Compromises: It is necessary to consider the additional requests generated for the implementation of the feedback mechanism as well as the availability of the memory

to be used. This whole mechanism and chain of events is complex, so it should only be applied to important interactions.

Then, the interactions extracted from the case study are presented, as well as a detailed justification analysis of the application of each of the resilience solutions.

Presentation component	Request type	Information validity	Importance of safeguarding information in criculation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
No	Get	High	Not important	Low	Low / considerable number of users	Retry, Timeout, Cache, Fallback, Circuit Breaker

Table 4: First type interaction identified in the case study.

Considering the type of interaction characterized by Table 4, we begin by proposing the application of the *Retry* pattern. Since none of the components involved is a presentation component, in the event of any failure occurring, the recovery needs to be made programmatically. This *Retry* mechanism implements a strategy of new attempts to access the service. We also note that this mechanism inevitably increases the number of requests originated by the component, and may even be harmful in the event of a malfunction of the external service due to overload. Precisely as a way of combating possible external component overloads given the considerable number of users and characterizing its possible unavailability, the *Circuit Breaker* pattern appears. Along with this mechanism, the *Fallback* and *Timeout* solutions are also integrated as being very useful for collecting information on determining the failure of an request. Finally, considering a high information validity, the *Cache* pattern appears, in order to make the most requested information available. This relieves the computational load on the component while accelerating the response time. Consequently, not only does the load relief on the component further reduce the likelihood of failure, but also in a scenario of component unavailability, the information found in this mechanism may continue to be served normally. Finally, maintaining this *Cache* mechanism requires some attention and effort to make the most of it.

Presentation component	Request type	Information validity	Importance of safeguarding information in criculation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
No	Post	High	Important	High	High / considerable number of users	Retry (optional), Timeout, Fallback, Circuit Breaker, Backup

Table 5: Second type interaction identified in the case study.

Still within the interactions of the *Tracker* component, apparently the proposed resilience solutions have not undergone major changes in relation to the previously described interac-

tion but in fact, considering the characterizing factors of the Table 5, it is clear that these are actually quite different.

As in the previous interaction, presentation components are not involved in the communication, which will cause any failure to require programmatic treatment. Furthermore, it should be noted that the information is not obtained but registered, which means that one of the intervening components produces some information that needs to be received and registered by the other components. Looking further at the fact that we consider the information in circulation to be important, it will be essential to implement a mechanism to safeguard the information in the event of any failure in the information's delivery. For this, implementing an additional mechanism for sending feedback at the time of receiving the information will be crucial. In the event of not receiving this confirmation, the information must be stored with the component originating the data and not getting lost. This information storage can only be done if the information is considered valid and useful for a long period. This safeguard mechanism is guaranteed by the patterns *Timeout*, *Cache* (on the client side) and *Fallback*, which establish the maximum waiting time for considering the sending of information as failed, the safeguard of the information to be sent and finally the implementation of the mechanism for sending feedback and configuration of pre-defined error messages.

Given that this is an action of recording information and with a considerable number of clients, we expect a high load on the receiving component. To this end, the proposal for the application of the *Circuit Breaker* pattern appears, whose major function is to monitor and improve the quality of the service, protecting it against possible overloads. Still considering the high amount of traffic, we propose the application of the *Retry* mechanism as an option to recover the possible safeguarded information in the local memory of the information-emitting component. We should always do the implementation or not of this mechanism taking into account that it will naturally originate at the moment of recovery the additional sending of requests besides the normal functioning of the interaction.

Presentation component	Request type	Information validity	Importance of safeguarding information in criculation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
No	Post	Instant	Not important	Very high	High / considerable number of users	Fallback, Circuit Breaker

Table 6: Third type interaction identified in the case study.

Continuing with interactions that do not involve presentation components (Table 6), the following differs from those already presented regarding the validity of the information. Here, its validity is practically instantaneous, which makes any attempt to recover lost information totally in vain. Thus, given the number of users and the high number of

interactions between the components, we propose the application of the *Circuit Breaker* pattern, aiming at the component's protection against possible peak loads. The *Fallback* resilience solution appears with the purpose of preventing the propagation of possible errors that may occur through the pre-configuration of error messages. For interactions of this type, we do not recommend the application of mechanisms to guarantee delivery or to safeguard information, since the information is short-lived and the major picture should be the continuous operation of the component with the best possible performance.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	High	Not important	Low	Low / few users	Cache, Fallback, Timeout

Table 7: Fourth type interaction identified in the case study.

Faced with interactions that now involve direct user interaction (Table 7), there is no longer a need to programmatically resolve all possible errors in communication. In this way, the application of *Retry* mechanisms is no longer strictly necessary for solving problems, even though these can be useful to improve the user experience. Thus, to the *Cache, Timeout* and *Fallback* resilience solutions we propose the addition of the *Circuit Breaker* pattern as a preventive measure against possible overloads of the service support component, though this leads to a higher level of complexity for the interaction. For this case, we found that the application of this mechanism was not helpful given the small number of users and the low computational load associated with it. Regarding the remaining proposals, the application of the *Cache* solution becomes obvious considering the long validity of the information. It will also be necessary to consider the cost of maintaining this *Cache* mechanism, which varies with the need or not to invalidate and update the information present in this memory. Regarding the *Timeout* and *Fallback* solutions, they respectively intend to invalidate a request for information in a timely manner and to protect against the occurrence of errors for unexpected responses through the pre-definition of messages from previous errors previously.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	Intermediate	Not important	High	High / few users	Cache, Fallback, Timeout, Circuit Breaker

Table 8: Fifth type interaction identified in the case study.

Moving on to the next type of interaction (Table 8), apparently there are few differences in relation to the immediately previous interaction, but the same cannot be said for the proposed solution. Keeping with the fact that this is for obtaining information, we are present now in an interaction where the validity of the information is not as long as before, but it is also not short enough for the immediate disposal of a *Cache* solution. We propose the application of a *Cache* mechanism, always considering the maintenance effort to maintain the resilience solution and the probable number of accesses while the information is not updated. The solutions *Fallback* and *Timeout* appear in this scenario with exactly the same purpose as the previous interaction - the invalidation of an request in which some error may have occurred in due time. Finally, even though the interaction presents few users, we consider the computational load of the service provision high, which is why we propose the application of the *Circuit Breaker* solution. Now in this interaction, protection against possible spikes in requests to the component is even more important because of the computational load of each request.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	Instant	Not important	Very high	Low / few users	Timeout, Fallback

Table 9: Sixth type interaction identified in the case study.

Contrary to the previous scenario, we now find an interaction (Table 9) that has a low computational load but with a much higher rate of requests. Typically, these types of interactions arise in situations where it is necessary to keep the information as up to date as possible. Therefore, any guarantee of safeguarding information in transit is discarded. Another of the discarded techniques will be the reuse of information, considering that it has very short validity. Finally, we do not recommend the application of a *Circuit Breaker* mechanism since the complexity of the interaction is low, the number of users is also reduced, and the focus of this is the flow of information as quick as possible. Any mechanism that adds complexity to the communication is not indicated. Thus, there remains the application of the solutions *Timeout* and *Fallback*, with the purpose respectively of discarding requests for information that are too long and requesting information that is probably already obsolete as well as the treatment in the simplest way possible of errors that may occur.

This is an interaction that in all resembles the interaction Table 7, only distinguishing itself by the number of potential users (Table 10). Naturally, this high number of users will create an additional load on the service support component. Thus, besides the proposals previously made, we propose also to apply the *Circuit Breaker* pattern, aiming the component protection against possible peak loads.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	High	Not important	Low	Low / many users	Cache, Timeout, Fallback, Circuit Breaker

Table 10: Seventh type interaction identified in the case study.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	Intermediate	Not important	High	High / many users	Cache, Timeout, Fallback, Circuit Breaker

Table 11: Eighth type interaction identified in the case study.

Again, only the potential number of users differentiates this interaction from the one previously presented (Table 11). Regarding the proposal for resilience solutions to be applied, this also remains for this case, considering that in the previous scenario it was the computacional load of an individual request which motivated its application. Now, this pattern makes even more sense considering that a high number of users will request this service already considered heavy.

It should also be noted that we included this interaction in the methodology description to represent the versatility of this resilience solution because, as can be seen, it promises improvements both in a scenario with a low and high number of users. If the computational load of each request is considered high, it will make more sense given that this mechanism has properties that intend to preserve the component's good functioning and with the least possible errors.

Presentation component	Request type	Information validity	Importance of safeguarding information in circulation	Number of interactions	Computacional load versus potencial nº of users	**Solution proposals**
Yes	Get	Instant	Not important	Very high	Low / many users	Cache, Timeout, Fallback, Circuit Breaker

Table 12: Ninth type interaction identified in the case study.

This last interaction (Table 12) is also similar to the interaction represented in Table 9 and again differing only by the highest number of users. Therefore, similarly to the situations previously described, we propose the application of the *Circuit Breaker* solution. We also propose the application of the *Cache* mechanism in situations where it is not possible to get the most up-to-date information, thus making it possible to present the last information

obtained successfully that is saved in the device memory. This mechanism will improve the user experience and, in the first phase, it may even hide a possible failure from the user. We should also note that the information from the moment when it was obtained must always accompany the presentation of the cached data, so it does not mislead the user.

4.2.1 *Bulkhead pattern analysis*

Unlike the other resilience solutions raised, the *Bulkhead* pattern requires an analysis at the architectural level, taking into account that it consists of establishing resource usage limits for each of the functionality support components.

In microservices-oriented solutions, it would be natural to propose the division of a problematic component into different components. This division already implements by itself the non-interference in the functioning of two features that are hosted in the same component, but we can not make decisions always in such a linear way. If two functionalities work on the same data set, additional efforts would be necessary to ensure the information consistency and synchronization between the two new components. The information replication by the new components would clearly bring an enormous additional effort regarding the data consistency and without guarantees of perfect operation. Also, the isolation of this information in an independent component can create a stress point when accessing it. Furthermore, the management of the components of a microservices based solution is challenging, so the creation of additional components only adds complexity, which is not desired. So, the application of this *Bulkhead* principle allows to prevent more extreme interference between functionalities that are hosted by the same component, since it promotes the implementation of limits for the use of hardware resources in each of the functionalities.

Therefore, we propose the application of this pattern only in situations where it is not possible to easily divide the components that support the features considered problematic. This impossibility of division may be due to the high level of data coupling or interdependencies between the components that support the functionalities.

4.3 THE BACKUP PATTERN

During the analysis of the case study, we found an interaction type in a recurring way, which none of the resilience proposals raised previously had solved completely. For the need of safeguarding high priority information when in circulation, we developed a pattern dedicated to this specific scenario, considering the importance of this kind of interaction for the solution funcionality. Since, through the case study analysis, we verified that any of the

resilience patterns gathered previously had proved to be promising in mitigating this error, we propose this newly developed strategy as a resilience pattern.

Taking into consideration that one of the major problems for the resilience of a microservices-oriented solution is the fact that this is a distributed system and consequently network dependent, for the establishment of the necessary communication between components in any solution of this genre. Therefore, it is necessary to apply a mechanism that addresses the network unreliability. Our solution safeguards the information in circulation on the network, taking advantage of the internal memory of the component originating the request. For any request in which it is necessary to safeguard information in transit, we must register this information in the internal memory at the time of shipment. In addition, we must also prepare the receiving component to send an acknowledgement message confirming receipt of the request. When this acknowledgement is received by the component originating the request, it may discard the information saved at the time of sending. If this message is not received, using the *Timeout* technique, the wait for this confirmation message should only be for a limited period. If the time is exceeded, we must persist with the information in the equipment until recovery is possible.

The retrieval of this information may also be based on two major strategies, one performed automatically and the another manually. For the first one, actions such as checking the existence of connectivity with the target component or even with the Internet will be essential for determining whether or not automatic recovery can take place. For the second strategy, an alert can be raised for any monitoring system normally present in this kind of microservices-oriented solution. However, the retrieving and information migration processes become dependent on human intervention. This strategy has the advantage of not generating additional traffic to solve the problem, while the first one will not depend on human intervention and eventually will be solved by itself. Depending on the application situation, this additional traffic generated can become dangerous for the functioning of the information receiving component, and it can originate traffic spikes. Still regarding data recovery, another precaution we need to take is the development of a data entry mechanism that guarantees the non-repetition of records, thus ensuring data integrity after the data recovery process.

4.4 THREATS TO VALIDITY

Throughout this study, some obstacles were found that can be identified as vulnerabilities to the veracity of the results produced. This being a study integrated in a master's dissertation, it is natural that my individual experience is not very large and is based mainly on theoretical concepts. Even so, attendance in the web engineering course comes to fill this lack of experience and knowledge initially mentioned, since this course addresses topics related

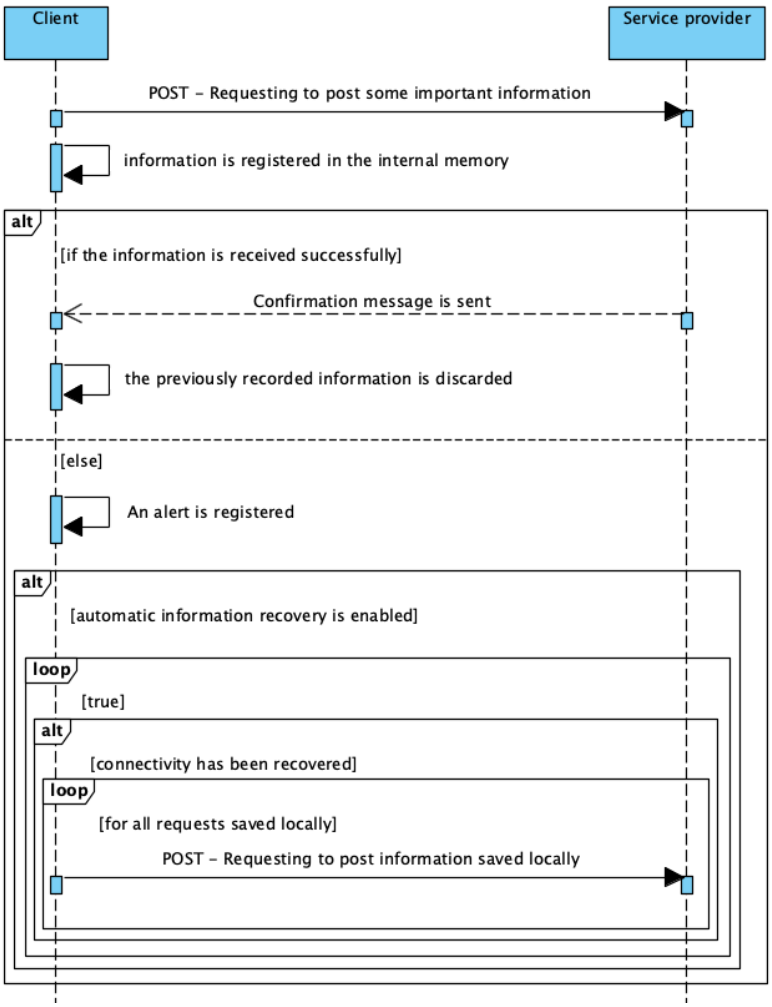


Figure 28: The *Backup* pattern.

to the correct design of web applications. As a way to counterbalance this gap, concepts and decisions have always been validated by this thesis supervisors, who have extensive experience in the area. In addition, the supervisor André Ferreira was responsible for this solution design, which implies a thorough knowledge of its operation. Furthermore, supervisors Jácome Cunha and André Ferreira, in addition to the large experience in the area, are currently teaching the specialization course of a Minho University master's degree, which addresses precisely the principles of architecture and design of solutions oriented to microservices.

Another essential point for the validation of the case study would be the implementation of strategies to increase resilience and practical validation but, given the complexity of the study and shortage of time, it was not possible. Still, we proceeded to validate each of the solutions theoretically, based on *UML* diagrams to specify all the relevant details implementation and new behavior to be implemented in the interactions between components.

All the proposals made in this study are based on a set of solutions previously selected in an initial state-of-the-art study phase. Thus, this selection process may not include all the solutions for certain interactions. As a form of mitigation, and taking into account the establishment of the theme among the scientific community, which leads to the practically null existence of scientifically verified information, we carried out an analysis of grey literature on the Internet on the subject. As a result of this study, the most mentioned set of resilience solutions in the analyzed sources was brought together.

Now, considering the case study itself and its creators, the fact that it was produced in an academic environment does not diminish it in relation to other software products produced in the business context. This is due to the fact that the entire process of idealization and construction of the set of product requirements was first done by a company already well established in the production of software, **Bosch** in Braga, but also because the entire production process was monitored and guided by experienced professionals in the field. Still, as a proof of the quality of this case study, it is in operation in the city of Braga and offering very relevant results regarding the operation of buses in the city.

Finally, the construction of a methodology based on the analysis of only one case study can be considered insufficient but, after a more detailed analysis of it, we noticed that it presents a wide variety of interaction types, being thus considered representative of the majority of interactions present in any software product.

5

CONCLUSION

In this dissertation, we made a study of the techniques for increasing the resilience of a microservices-oriented solution. Initially, we searched in scientific databases for information on the subject, where we could not get relevant information. This lack of information is due to the fact that the development of this theme is strongly driven by the software industry, which means that many of the techniques used are not yet present in scientifically verified documents. Thus, we concluded that the scientific literature is not yet mature enough regarding resilience in microservices-oriented architectures. Unlike scientifically verified sources, we found large amounts of information related to the topic in unverified media such as blogs, forums, own posts, company websites, among others. In general, all the information found on the subject on the Internet that we found is what is characterized as grey literature (Kamei et al., 2019).

Given that the elaboration of a dissertation always requires some type of verification on the information to refer to, it was necessary to carry out a study of this same gray literature, with the final objective being the assembly of the set of techniques most pointed out as effective for increasing resilience in architectures oriented to microservices.

Having already assembled the set of techniques for increasing resilience in architecture oriented to microservices resulting from the study carried out on the gray literature, we analyzed a case study that is currently implemented in urban buses in the city of Braga. This study focused on the possible error scenarios and points of failure in the solution that is currently in operation. Still in the study carried out, given the complexity of the solution to be analyzed, we considered only the components directly related to the functional part.

After identifying all possible error scenarios, we made the proposal for applying the resilience solutions previously raised in the study.

We should also note that we identified an error scenario in which the solutions raised above did not offer a solid and consistent solution. In addition, this identified point of failure represented the loss of highly relevant information for the solution, which is essential for a significant increase in the level of resilience. Given the importance of this new resilience-enhancing technique for the resilience of the case study, we added it to the set of solutions initially proposed.

Having carried out all the analysis of the case study, it was possible to generalize about the interactions analyzed and which solution proposals we presented. As a result of the analysis of the process of choosing resilience solutions, it was possible to derive a set of factors that characterize the interactions that identify the context in which the interaction is carried out in various categories.

To conclude, having already raised the interactions characterized according to the decision factors used previously, and the proposals presented for each one of them, we carried with the description of each of the interactions and the purpose of application of each of the proposals.

In this way, this entire process of analysis and proposal of solutions on these generic interactions makes up the target methodology presented as an objective to achieve with this dissertation.

This complete study also raises some questions and future work on this theme:

- the realization of these proposed solutions in a practical way both in several case studies and their application in production contexts;
- mostly, we designed the resilience solutions in such a way that the interference in the solutions business layer is minimal, which opens a space for a possible development of mechanisms for automatic application of these proposals;
- the relevance of languages that follow the Aspect-Oriented Programming paradigm (Lafferty and Cahill, 2003) in the automation processes of applying these solutions;
- the study of the performance-level resilience commitments that these solution proposals impose.

BIBLIOGRAPHY

Configure default memory requests and limits for a namespace. URL https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/.

What are MTTR and MTTI? — Sumo Logic. URL https://www.sumologic.com/mttr-mtti/.

How Does Page Load Time Affect Your Site Revenue? - MachMetrics Speed Blog. URL https://www.machmetrics.com/speed-blog/how-does-page-load-time-affect-your-site-revenue/.

Fallacies of distributed computing, 2019. URL https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing#The_fallacies.

Resilience in distributed systems. Technical report, Infosys, 2019. URL infosys.com.

Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian Tamburri, and Theodore Lynn. Microservices migration patterns. *Software: Practice and Experience*, 48, 07 2018. doi: 10.1002/spe.2608.

Samir Behara. Making your Microservices Resilient and Fault Tolerant - dotnetvibes. URL https://samirbehara.com/2018/08/06/making-your-microservices-resilient-and-fault-tolerant/.

Amine Benhallouk. Improve availability and resilience of your Microservices using these seven cloud design patterns. URL https://bit.ly/2RKXY95.

Silvio Buss. Resilience pattern for Java microservices. The Circuit Breaker. - DEV Community. URL https://dev.to/silviobuss/resilience-pattern-for-java-microservices-the-circuit-breaker-b2g.

Kelly Clay. Amazon.com Goes Down, Loses $66,240 Per Minute. URL https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#78b7fcc8495c.

K. Finnigan. *Enterprise Java Microservices*. Manning Publications, 2018. ISBN 9781617294242. URL https://books.google.pt/books?id=KaSNswEACAAJ.

Mario Gerard. Building Resilient Microservices - Technical Program Management. URL https://www.mariogerard.com/building-resilient-microservices/.

Gökhan Gökalp. Resiliency Patterns in Microservice Architecture — Gökhan Gökalp. URL https://www.gokhan-gokalp.com/en/resiliency-patterns-in-microservice-architecture/.

Hajar Hameed Addeen. A Dynamic Fault Tolerance Model for Microservices Architecture. Master's thesis, South Dakota State University, 2019.

Kasun Indrasiri. Microservices in Practice: From Architecture to Deployment - DZone Microservices. URL https://dzone.com/articles/microservices-in-practice-1.

Fernando K. Kamei, Sergio Soares, and Gustavo Pinto. The use of grey literature review as evidence for software engineering. In *Anais Estendidos do X Congresso Brasileiro de Software: Teoria e Prática*, pages 56–63, Porto Alegre, RS, Brasil, 2019. SBC. doi: 10.5753/cbsoft_estendido.2019.7656. URL https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/7656.

Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. *SIGPLAN Not.*, 38(11):1–12, October 2003. ISSN 0362-1340. doi: 10.1145/949343.949307. URL https://doi.org/10.1145/949343.949307.

Tomas Livora. Fault Tolerance in Microservices. Master's thesis, Universitas Masarykiana, 2016. URL https://is.muni.cz/th/ubkja/masters-thesis.pdf.

Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. Technical report, 2016.

Péter Márton. Designing a Microservices Architecture for Failure — @RisingStack. URL https://blog.risingstack.com/designing-microservices-architecture-for-failure/.

Netflix. Hystrix: Latency and Fault Tolerance for Distributed Systems, November 2018. URL https://github.com/Netflix/Hystrix.

S. Newman. *Building Microservices*. O'Reilly Media, 2015. ISBN 9781491950357. URL https://books.google.pt/books?id=1uUDoQEACAAJ.

John Rocela. Let's talk about Resilience - By. URL https://hackernoon.com/lets-talk-about-resilience-97051e14761f.

Frank Rosner and Alexander Potukar. Resilience design patterns: retry, fallback, timeout, circuit breaker - codecentric AG Blog. URL https://blog.codecentric.de/en/2019/06/resilience-design-patterns-retry-fallback-timeout-circuit-breaker/.

Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. *Doctor Dobbs Journal*, 01 2008.

Nathaniel Schutta. Should that be a Microservice? Part 5: Failure Isolation. URL https://content.pivotal.io/blog/should-that-be-a-microservice-part-5-failure-isolation.

Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. 03 2018a. doi: 10.5220/0006798302210232.

Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. 03 2018b. doi: 10.5220/0006798302210232.

Mike Wasson, Alex Buck, Sam Ferree, David Stanford, Adam Boeglin, and Marc Wilson. Bulkhead pattern - Cloud Design Patterns — Microsoft Docs. URL https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead.

A

SUPPORT MATERIAL

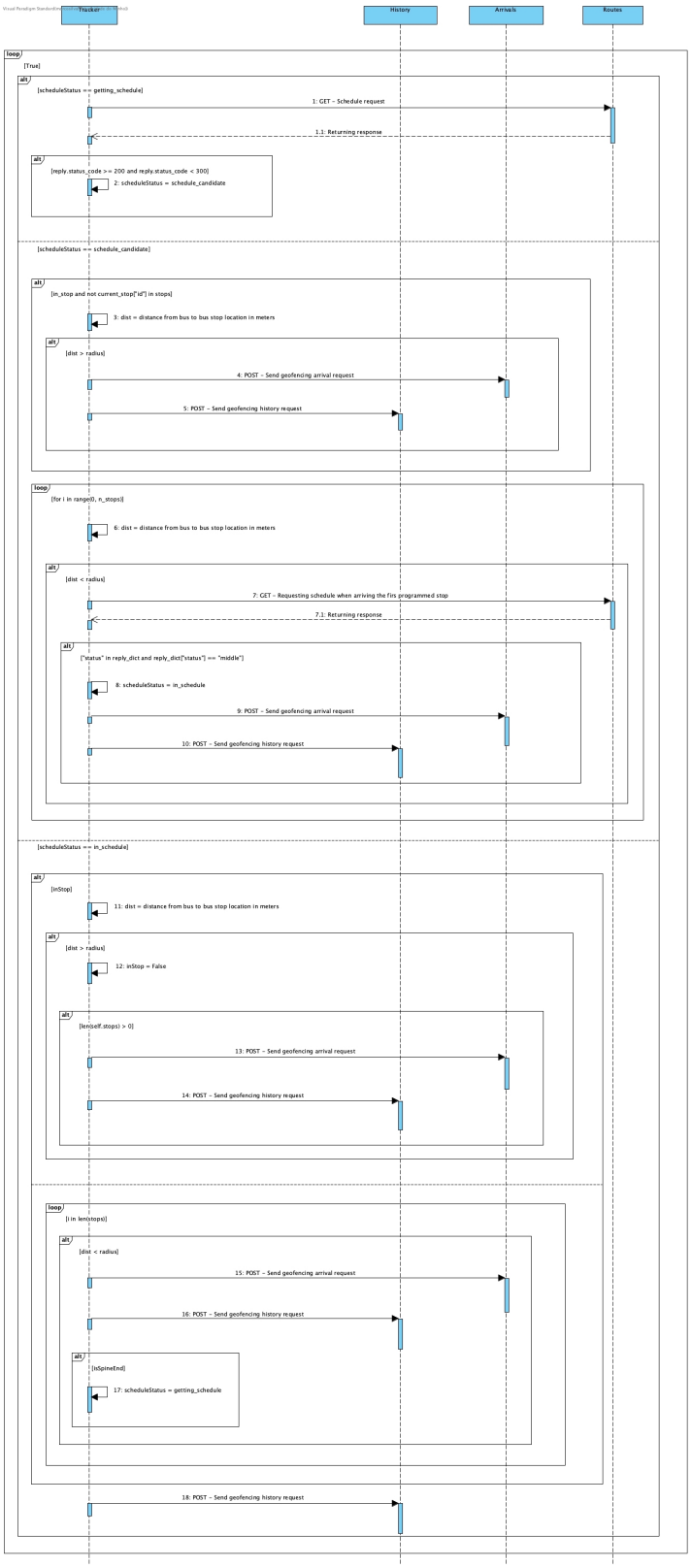


Figure 29: *Tracker* sequence diagram before patterns application.

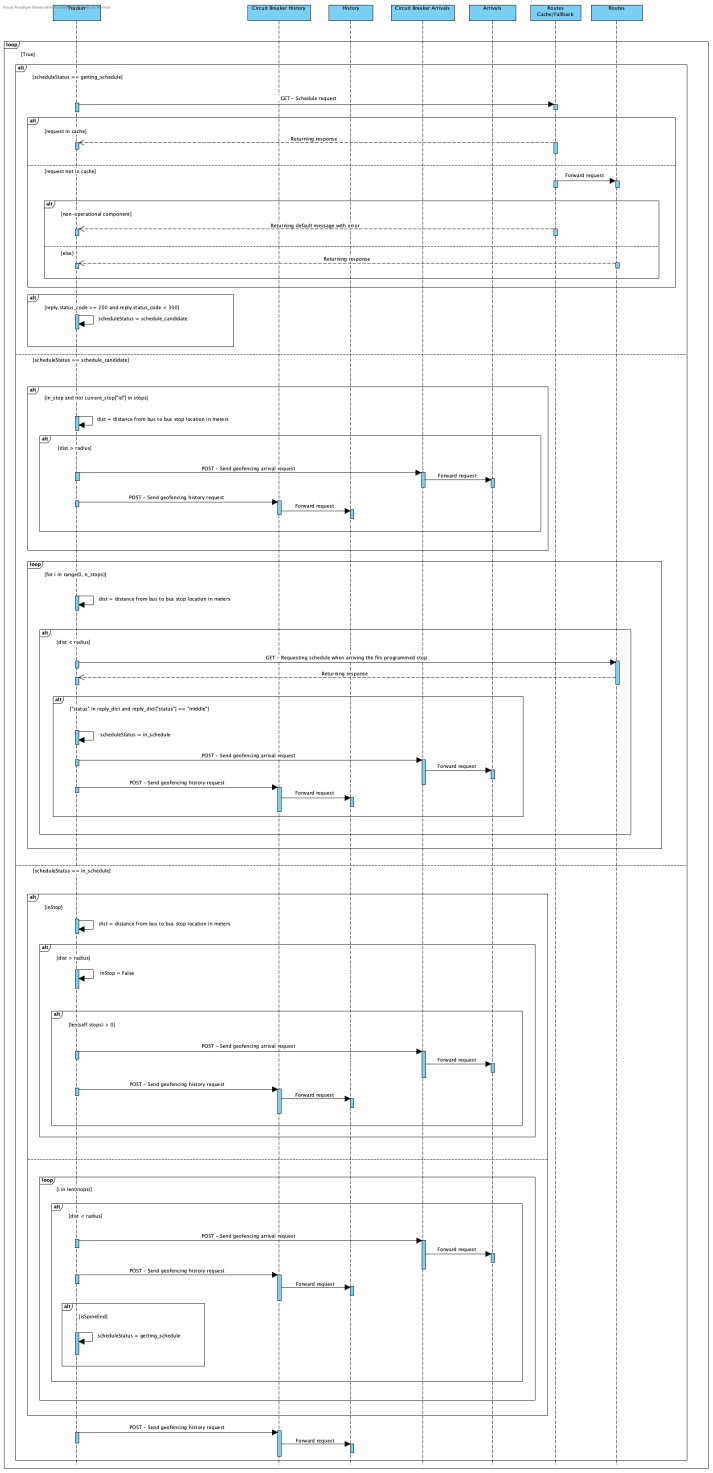


Figure 30: *Tracker* sequence diagram after patterns application.