TU WIEN Informatics

Unified Framework for robust Microservice Communication

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Software Engineering & Internet Computing

eingereicht von

Alexander Allacher, BSc
Matrikelnummer 11810873

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 29. August 2023

Alexander Allacher	Jürgen Cito

TU WIEN Informatics

Unified Framework for robust Microservice Communication

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master programme Software Engineering & Internet Computing

by

Alexander Allacher, BSc
Registration Number 11810873

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, 29th August, 2023

Alexander Allacher	Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Alexander Allacher, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. August 2023

Alexander Allacher

Kurzfassung

Microservice-Architekturen sind zu einem verbreiteten Ansatz in der Softwareentwicklung geworden und bieten Skalierbarkeit, lose Kopplung und Wartbarkeit. Dieser Ansatz ermöglicht es uns, Systeme als einer Menge an kleinen, unabhängig bereitstellbaren Diensten zu erstellen, die über simple Mechanismen kommunizieren. Da Microservices von Natur aus verteilte Systeme sind, bringen sie, insbesondere bei der Kommunikation, Herausforderungen mit sich. Traditionelle synchrone Kommunikation, wie REST, kann zu zeitlicher Kopplung und kaskadierenden Fehlern führen. Asynchrone Kommunikation hingegen bringt eigene Herausforderungen mit sich, einschließlich des Bedarfs an Message Brokern. Außerdem ist ein Umdenken bei der Gestaltung der Interaktionen notwendig. Diese Arbeit führt eine domänenspezifische Sprache (DSL) ein, um die Kommunikation von Microservices zu definiert. Dabei wird darauf abgezielt, die Lücke zwischen diesen beiden Kommunikationsparadigmen, mit besonderem Augenmerk auf Robustheit, zu schließen.

Durch die Anwendung von ingenieurwissenschaftlichen Forschungsmethoden haben wir wesentliche Aspekte der Microservice-Kommunikation identifiziert und sie in ein Framework integriert. Ausgehend von der DSL generiert dieses Framework den Großteil des notwendigen Codes und reduziert somit das Risiko häufiger Fehler. Um die vorgeschlagene DSL und das Framework zu validieren, bewerten wir ihre Anwendbarkeit anhand verschiedener Open-Source-Projekte. Die Bewertung konzentriert sich auf Robustheit, Wiederverwendbarkeit von Schnittstellendefinitionen und Asynchronität. Weiters zeigt sie auf, wie die erkannten Fehler in diesen Projekten vermieden werden hätte können. Da der Großteil der analysierten Projekte nur zu Demonstrationszwecken dienen, könnten die Ergebnisse in realen Szenarien eine begrenzte Anwendbarkeit haben. Andererseits existieren diese Demonstrationsprojekte nicht ohne Grund. Sie wurden erstellt, damit Entwicklerinnen und Entwickler die Entwicklung von Microservice-Architekturen lernen können. Zusammenfassend zeigt die Arbeit zwar einen vielversprechenden Ansatz zur Harmonisierung von synchroner und asynchroner Kommunikation in Microservices, dennoch ist weitere Forschung, insbesondere in Bereichen der Kommunikation mit externen Diensten, erforderlich.

Abstract

Microservices architectures have become a prevalent approach in software development, offering scalability, loose coupling, and maintainability. This approach allows us to build systems systems as sets of small, independently deployable services that communicate through lightweight mechanisms. However, the distributed nature of microservices presents challenges, particularly in communication. Traditional synchronous communication, such as REST, can lead to temporal coupling and cascading failures. Asynchronous communication, on the other hand, presents its own set of challenges, including the need for message brokers and a different way of thinking about interactions. This thesis introduces a domain-specific language (DSL) that defines microservice communication, aiming to bridge the gap between these two communication paradigms while focusing on robustness.

By leveraging engineering research methodology, we identified essential aspects of microservice communication and integrated them into a unified framework. This framework, powered by the DSL, aims to generate most of the necessary code, reducing the risk of common pitfalls. To validate the proposed DSL and framework, we evaluate its applicability using various open-source projects. The evaluation focuses on robustness, reusability of interface definitions, and asynchronicity, and it demonstrates how oversights in these projects could have been avoided. As most of the analyzed projects are only for demonstration purposes, the results might have limited applicability in real-world scenarios, but on the other hand, these demonstration projects are there for a reason, so developers can learn how to create microservice architectures. In conclusion, while the thesis presents a promising approach to harmonize synchronous and asynchronous communication in microservices, further research is needed, especially in areas like communication with external services.

Contents

CHAPTER 1

Introduction

After Netflix presented their microservice architecture in 2014 and shared what they learned from a successful transition from a monolith to this new architecture, the idea of microservices spread quickly [WKR21]. It emerged as a new architectural style that is based on the idea of building a system as a set of small services, that are independently deployable and communicate with each other via light-weight mechanisms, like calls over the network [DGL+17, WKR21]. Although microservices are not a new idea, as they are based, on service-oriented architectures (SOA), they are different in many aspects. It focuses way more on creating highly maintainable and scalable systems with strong emphasis on high cohesion, by breaking the complexity down into small loosely coupled services [DGL+17]. Therefore, microservices are an option worth considering, as nowadays, new features need to get deployed as fast as possible, due to frequently changing business requirements. The classic monolith architecture is not suitable anymore, and therefore, many companies switched to microservices, as they allow deploying new features independently of each other [BZ16].

With all the benefits, there are also some major drawbacks. The most prominent one is that microservices are inherently distributed systems, and developing and maintaining such a system is way more complex than a monolith. We need to think about the way how microservices can communicate, especially how we define the contracts [DGL+17]. For many years, communication happened usually via REST over HTTP, but such synchronous communication introduces tight temporal coupling between the services, which can lead to cascading failures and other problems [LF14, New19]. The alternative is asynchronous communication, which gained attention in recent years. Nevertheless, it is not a silver bullet, as it introduces new challenges, like the need for a message broker [KS21].

1.1 Target Audience

Our paper shows how we can make use of known techniques for communication in microservices by creating a DSL that can be used to describe many common aspects relevant to robust communication. As we propose a way how we could unify synchronous and asynchronous communication, we hope that our work can be used by researchers to investigate further on this topic. Further, we create a sample implementation of our DSL and code generators. Hence, we believe that our work is interesting for practitioners, too.

1.2 Motivation

Bogner et al. [BFWZ19] conducted 17 semi-structured interviews with professionals that implemented real-world microservice architectures to gain in-depth knowledge of the hurdles they had to overcome and how the perceived software quality changed. Further, they investigated what technologies the companies were using. Although some made heavy use of events for communication, they stated that „. . . the de facto standard was RESTful HTTP.“ While most of the participants felt ok with mainly using synchronous communication, others categorized it as harmful. As we want to introduce an online shop system in our motivating example, it is worth mentioning that Bogner et al. [BFWZ19] had multiple interviews about projects where they replaced an of-the-shelf retail solution with a custom one built as a microservice architecture.

In the systematic mapping study by Di Francesco et al. [DLM19], they showed which topics are currently researched in the field of microservices. Although communication is one of the most important aspects of microservices, due to their distributed nature [Tem20], communication is not listed as a trending research topic. This is interesting, as Temoor et al. [Tem20] define microservices as „the smallest independent process that communicates via a messaging“ and according to Jamshidi et al. [JPM+18], there is a trend towards more asynchronous communication and supporting libraries, with the goal to make microservices more robust.

In more recent years, we can already find some papers that pay more attention to asynchronous communication like Kul and Sayar in 2021 [KS21], but they do not focus on how to combine both synchronous and asynchronous communication.

1.2.1 Example

To illustrate the different problems and challenges that can occur when implementing microservices, we will use an example of a very simplified online shop system. The system consists of two microservices: the **CatalogService** which is responsible for managing the products, categories, and available stock, and the **OrderService** which is responsible for managing the order process, including payment. For understanding the difficulties and problems we want to solve, we introduce the following example scenarios (ES):

1. **ES1:** A customer wants to buy a product, and therefore sends a request to the OrderService.
2. **ES2:** The company wants to introduce a new customer service.

When we implement the communication between the OrderService and the CatalogService with classic synchronous REST calls to deal with **ES1** after the OrderService receives the request, it needs to check if the product is available and what the price is. So it makes a request to the CatalogService, which checks the stock and returns the price. This increases the latency, as the OrderService needs to wait for the response of the CatalogService. Further, if the CatalogService is not available, the OrderService cannot process the request. In cases which really need synchronous communication we should at least add retries and circuit breakers to avoid cascading failures, but it is hard to ensure that every request from every service fulfills these requirements.

If direct synchronous communication is not necessary, we could use asynchronous communication. For example, an event-driven approach where the CatalogService publishes an event when the stock changes. Then the OrderService subscribes to this event and stores the information in its own database. So we can get rid of this temporal coupling, but now we need to ensure that the CatalogService reliably publishes all events and the OrderService is able to handle the events correctly. One can imagine that this is not trivial when we have many different microservices which send many different events, considering idempotency, retries, and so on.

In **ES2**, we want to introduce a new customer service that can handle user accounts, so they do not need to enter all their information every time they want to buy something. Further, the customer service should have all the orders a customer made, so they can easily see what they bought in the past. Before, we already made the decision to use event-driven, asynchronous communication, so we can easily add a new service that listens to the already published events. However, this only works for new events, so the customer service would not have access to all past orders. As a consequence, we need to find a way to get the old, potentially huge amounts of, data to the customer service.

1.3 Research Questions

To deal with the problems mentioned, this thesis mainly focuses on the following research questions:

1. **RQ1** How can we design a DSL that models robust synchronous and asynchronous microservice communication?
2. **RQ2** Which types of microservices can we generate code for?

With **RQ1**, we aim to analyze which mistakes can be made when implementing microservice communication. We want to find best practices and patterns to avoid these

mistakes and to make communication more robust, for synchronous and asynchronous transmission. Further on this topic, we want to investigate which scenarios can be solved with asynchronous communication and which ones need synchronous communication. Then we want to find a way to combine both communication types so that we can use the advantages of both while keeping the overhead low. This should be achieved by designing a DSL, which can be used to model the communication between microservices, considering the stated aspects.

RQ2 in consequence, is about creating code generators for the defined DSL, which can generate code for different types of microservices. By doing this, we want to show that the DSL is not only a theoretical concept, but can be used in practice. To further show the applicability of the DSL, we want to evaluate it by analyzing open-source projects that use microservices and show how we would model their communication with our DSL and what pitfalls we can avoid. We do not expect to be able to generate code for all types of microservices, and we certainly will not find a solution for every problem that can occur when implementing communication for microservices. Nevertheless, we want to show that we can avoid many problems by using our DSL but also make the limitations of our DSL clear.

1.4 Evaluation

To evaluate the capabilities of our DSL, we will analyze open-source projects that use microservices. How have they implemented their communication and what problems did they encounter? We especially focus on robustness, reusability of interface definitions, and asynchronicity, as these are the main goals of our DSL. Aderaldo et al. [AMPJ17] specified criteria to evaluate microservice architecture projects. In the evaluation, we will select applicable criteria and use them to find microservice architectures we can use to test our DSL. Additionally, Aderaldo et al. listed four open-source projects that use microservices that fulfill their criteria and which makes them a good example to start with. We will use the found projects to evaluate our DSL by showing how we would model their communication with our DSL and what pitfalls we can avoid.

1.5 Methodology

To answer our research questions, we want to apply engineering research [RbAB+21]. But first, we want to collect a list of common errors which happen when implementing microservices. From the result of the first step, we want to identify important aspects of microservice communication and define what is necessary to achieve them. By combining the found concepts and methods in one framework, we want to show that it can reduce the risk of getting them wrong since the resulting framework should generate most of the necessary code.

To achieve these goals, we first want to create a metamodel that can be used to describe the communication. This is then used to derive a DSL, which is applicable for our needs.

The DSL is intentionally kept simple, but should still meet our stated requirements.

As this DSL should be used to configure the proposed framework and generate code, we are going to create a sample implementation for the framework and the code generators too. To validate our results, we plan to analyze some existing microservice projects with the identified errors from the first step and show how they could have avoided these errors by using our framework.

1.6 Structure

The remainder of this thesis is structured as follows: Chapter 2 introduces the background of microservices and the problems we want to solve. Chapter 3 presents related work, showing how other researchers have tried to solve similar problems we want to solve and how our approach differs from theirs. The metamodel and DSL are presented in Chapter 4, which also includes a description of how we implemented the framework. Chapter 5 explains the design decisions we made and why we made them, so it also explains how our framework works. In Chapter 6, we evaluate our framework by going through some open-source projects in detail. Finally, Chapter 7 concludes this thesis, lists the limitations, and gives an outlook on future work.

CHAPTER 2

Background

For a better understanding of the following chapters, this chapter will define important concepts around microservices. It will also give an overview over the current state of the art for microservice architectures, their benefits and challenges.

2.1 Monothlic Architecture

Traditionally, software applications are developed as a single unit. As these can get quite large, projects are often divided into modules, but in the end, they still share the same resources on one machine. This is the way software has been developed for decades and therefore has good support in most programming languages and frameworks [DGL+17].

Especially in the beginning, monoliths are easy to develop because they are just one application. However, while the application grows, it becomes harder to maintain and extend, as a developer often has to understand the whole, or at least large parts of the application to make changes. It is easy to break the application without knowing how things are connected. Updates of dependencies, like external libraries, could also be a problem due to the lack of knowledge of how the application is using them. Sometimes such an update may only break the application only in certain edge cases [DGL+17].

As a monolith is one application, it is also deployed as one application. The clear benefit of this is that the deployment process is simple and straightforward. On the other side, every time something is changed, the whole application has to be redeployed. Large applications can take a long time to deploy, which can be problematic when the application is used in production [DGL+17].

Later we will see that microservices are scaled by running multiple instances of the same service. For monoliths, this is often hard and very expensive, as the whole application has to be run multiple times, although only parts of it are under heavy load. Additionally, when building a monolithic application, synchronization for ensuring consistency is easy,

as all parts of the application are using the same resources. When running multiple instances of the same application, this is not the case anymore. This results in the need that a monolith has to be designed in a way so it can be run multiple times without breaking consistency [DGL+17].

2.2 Microservices

Inspired by service-oriented architecture (SOA), microservices are a software architecture pattern that structures an application as a collection of loosely coupled and cohesive services. Each service is a small, independent process, and they communicate with each other with messages. A microservice architecture is an inherently distributed system with all its benefits and challenges. Without forbidding or defining a certain programming paradigm, it provides a set of guidelines and principles that help to build independent services which should fulfill one purpose. Due to their limited scope of functionality, microservices are easier to understand and maintain than monoliths. Testing and examining the functionality of a service is straightforward because it is independent of other services [DGL+17].

In terms of deployment, microservices are prone to be shipped as containers (e.g. Docker containers), which makes it easy to deploy them on different machines. As a result, scaling is simple. Only the services that are under heavy load have to be scaled. By allowing to run multiple instances of the same service, microservices can not only be scaled horizontally, but also allow to run different versions of the same service at the same time. When done right, this allows updating services without downtime because the old version can be shut down after the new version is up and running. In particular, they can even run side by side for some time to allow other services to adapt to the new version [DGL+17].

2.3 Synchronous Communication

Communication between services is a key aspect of microservice architectures. The medium over which services communicate is the network [ALFT21]. One popular example of a protocol that is used for communication over the network is HTTP. This fact alone does not make the communication synchronous, but rather the fact that a sender waits for the server's response before continuing with its execution. It should be noted that this is the case even if the code is asynchronous (non-blocking) [MJBC22].

So what is the reason that synchronous communication is the de-facto standard for communication between services [BFWZ19]? An explanation could be due to the request-response pattern synchronous communication being real-time by nature. This means that the sender can be sure that the receiver has received the message and processed it. Moreover, this kind of communication is easy to reason about and can be used for time-sensitive tasks [MJBC22].

On the other hand, synchronous communication has some drawbacks. When creating request chains, where one service calls another service, which calls another service, and so on, it results in a dependency chain. So if any of these services are unavailable or at peak load, the whole chain is affected [New19]. One contradiction to mention is that synchronous communication is often used due to its fast responses, but by creating such request chains, we introduce a lot of latency, which in turn makes the response slow.

Two commonly used protocols for synchronous communication are REST and RPC.

2.3.1 REST

Representational State Transfer (REST) is a popular protocol for communication over the network. It is based on the HTTP protocol and uses its methods to perform actions on resources. The most common methods are GET, POST, PUT and DELETE [HSYK18]. The communication is stateless, which means that the server does not store any information about the client. This is especially important when having multiple instances of the same service. Also, the error handling is simple, as a failed request can be retried. A downside is by always transferring the whole state of a resource, REST is not very efficient when it comes to bandwidth [Fie00]. Often the data is encoded in JSON which itself is also not very efficient as it is only text. Yet, on the other hand, it is an advantage because it makes it very compatible across many different technologies.

As REST is for communication between a client and a server (the client sends a request to the server), it is necessary that the implementation of both sides align. The client has to know the structure of the request and the response, and the server has to know how to handle it. To not have to implement this logic twice, it is common to use a tool like OpenAPI. It is a standardized specification language for REST APIs that is independent of the programming language and framework. YAML or JSON are used to describe the API, and from this description, the client and server code can be generated. This is called API-first approach, but there is also a code-first approach, where the API is generated from the code [Fou23].

2.3.2 RPC

Remote Procedure Call (RPC) is a concept that allows a program to execute a function on a remote machine. The goal is to make the remote function call look like a local function call while transmitting control and data over the network in an efficient manner. For strong typing, it is necessary that there are stubs on the client side and skeletons on the server side at compile time. When a remote function is called, the client stub transparently forwards the call to the server and therefore abstracts the network communication [SG01]. Nevertheless, one has to be very careful when using RPC, as these calls are over the network and, therefore, can fail, are slow, and so on. When moving from a monolith to a microservice architecture, and only replacing the calls to local functions with remote calls, we introduce way too much communication over the network, which will lead to

performance issues. In conclusion, rethinking the communication is necessary to use remote procedure calls in moderation [LF14].

One popular implementation of RPC is gRPC, a high-performance and platform-agnostic framework for RPC developed by Google. The interfaces are defined using Protocol Buffers, which serialize the data in a binary format, so it is more efficient than JSON. These definitions are used to generate client and server code [Goo23a].

2.4 Asynchronous Communication

In the often-cited article from Lewis and Fowler published in 2014, they state that synchronous calls between services are problematic, as they can lead to cascading failures [LF14]. This is nothing new, as already in 2001, long before microservices were a thing, Saif and Greaves introduced the idea that synchronous communication is not a good fit for distributed systems, and we should rather use asynchronous messages instead [SG01]. To decouple services from each other in terms of time and synchronization, asynchronous communication is used. In opposition to synchronous communication, the sender does not communicate with the receiver directly, but rather sends a message to a message broker, which then delivers the message to the receiver when the receiver is ready to process it. This way, the sender does not have to wait for the response of the receiver and can continue with its execution. Another benefit over direct, synchronous communication is the possibility of many-to-many communication, as the sender does not have to know who is listening to the message. This makes it a good fit for a distributed system like microservices [KS21].

With the publish/subscribe paradigm, the notion of topics is introduced. The topics are used to categorize messages so that the sender can send a message to a topic, and all subscribers of this topic will receive it. Thereby, the sender does not have to know who is listening to the topic, and the subscribers do not have to know who is sending the message. So it can be that a topic does not have any subscribers, which means that messages to this topic do not get consumed. Besides this topic-based approach, there is also a content-based approach, where the subscribers can define filters so that they only receive messages that match the filter. The linking piece in this approach is the message broker. Depending on the filtering approach, it distributes the messages to the subscribers [KS21]. Due to the asynchronous nature, it is not guaranteed that the system is always consistent, and therefore it is eventually consistent [KS21].

2.5 Event-Driven Architecture

Event-Driven Architecture (EDA) is a software architecture pattern that promotes the production, detection, consumption, and reaction to events. Thereby, these events get distinguished into three categories: commands, events, and queries. When a service wants another one to do something that changes the state of the system, it sends a command. The receiver of the command is responsible for handling it and maybe sends a response

back. Events, on the other hand, are a notification that something has happened. So they are published by a service that has changed the state of the system, independent of who is listening. This means events do not request any action, but it could be that a service reacts to a given event, yet, there is no response to it. Queries are used to request information from a service, which then responds with a result. Unlike the other two, queries have no side effects [Sto18].

All this is done in order to create loosely coupled services that are independent of each other. To achieve this, it is important to share as little as possible so no other service can couple to it. Through the use of messaging, data gets distributed, so a subscriber does not need to perform an action on the source but rather can perform it on the stored data [Sto18].

2.6 Event Sourcing

When all changes to the state of an application are published as an event, the events describe the history of the application. When these are now immutably stored in the order they occurred, it results in an event log. Event sourcing is a pattern that uses this event log as the source of truth [Sto18]. Due to the fact that processing all the events can be quite expensive, it is common to store the current state of the application as a snapshot [Fow17]. It is always possible to replay the events to get the current state of the application, which leads to a corruption-resistant system [Sto18]. Event sourcing offers many benefits which are similar to those of a version control system, and it is often used in combination with CQRS. On the negative side, it is more complex to implement and understand, and there is the problem with replaying events that depend on external services. Another non-trivial aspect is that the schema of the data can change over time, which has to be taken into account when replaying events [Fow17].

2.7 Choreography over Orchestration

Besides the type of communication, there are also different ways how services can communicate with each other. These communication patterns describe how the services are connected and how they interact with each other. We can see this as a spectrum, where on one side, we have orchestration and on the other side, choreography. Orchestration is a centralized approach, where a handful of services are responsible to coordinate the communication and, by extension, whole business processes. In the days of SOA, this approach was often used, as it was easy to implement and understand. Unfortunately, orchestration increases the coupling between services and responsibilities get taken away from some services and transferred to orchestrators, which leads to a loss of autonomy.

As microservices are all about decentralization and autonomy, choreography is used to embrace these principles and allow for collaboration. In choreography, services are communicating with each other directly, without a central instance that orchestrates the communication. Publish/Subscribe mechanisms are a good fit for this. As already

stated, they allow services to subscribe and publish events without knowing who is listening [DGL+17, KS21].

2.8 Idempotency

Idempotency is a property of a function, which defines that the result of the function does not change when the function is called multiple times with the same input. So an action is idempotent if it can be performed multiple times, but it has the same effect as if it would be performed only once. In practice, it is a bit more complicated. For example, if we have a function that writes something to the log, then the log message will be in the log multiple times. The same counts for the usage of resources. But for our purposes, we can say that it is enough if the result of the function does not change, even if the function has other side effects.

Idempotency is important in distributed systems, as we cannot guarantee that a message is only delivered once. Multiple deliveries of messages can happen due to network, or other failures, which lead to a retry. When using message brokers, it is not always possible to guarantee that a message is only delivered once, and there can also be retries due to failures. What makes it even more complicated to enforce idempotency is that in a microservice architecture, we can have multiple instances of the same service, which means that the same message can be delivered to multiple instances of the same service. This can, for example, happen as a result of load balancing [Hel12].

One way to solve this problem is to give each message a unique identifier, which is stored somewhere, for example, in a database. When a message is received, the identifier is checked against the database, and if it is already there, the message is discarded. However, implementing this manually is a lot of work, and can lead to subtle bugs [RV13].

2.9 Ciruit Breaker

The Circuit Breaker pattern is a popular microservices pattern that plays a crucial role in enhancing the robustness and resilience of distributed services. When using reties, it is always a good idea to use circuit breakers, as a failing service can be easily overloaded with retries and therefore hindered from recovering. A further goal of circuit breakers is to prevent failure propagation to dependent services. This is accomplished by refusing incoming requests when a specific condition, like a system overload, is fulfilled, thereby safeguarding system latency at the cost of reduced availability. Another advantage of circuit breakers is that they can lead to faster failures and, therefore, faster recovery, as resources get freed up [SSKT22].

Even when auto-scaling is used, it is possible that the system is overloaded, maybe due to a sudden spike in traffic. This is owed to the fact that, typically the auto-scaling takes some time to react to the increased load. Hence a circuit breaker allows for a faster

reaction. It must be kept in mind that circuit breakers are a trade-off between availability and latency, as they can lead to reduced availability [SSKT22].

2.9.1 Client-Side Circuit Breaker

The client-side circuit breaker is implemented in the client, and therefore the client is responsible for handling the failures. This is done by intercepting all the requests to a given service [SSKT22]. Consequently, the client-side circuit breaker is only for self-limiting and helps more to prevent the client from getting overloaded.

2.9.2 Server-Side Circuit Breaker

The server-side circuit breaker is implemented in the service. After a server receives a request, it first goes through the circuit breaker, which checks if the request can be processed. If the circuit breaker is closed, the request is processed, and the response gets sent back to the client [SSKT22]. Due to these facts, the server-side circuit breaker still adds additional load to the server, even when it is open.

2.9.3 Proxy-Based Circuit Breaker

The third approach is the proxy-based circuit breaker. These circuit breakers are standalone services that are placed between the client and the service. Therefore, all communication between the client and the service goes through the proxy, which adds an additional hop and as a result can increase latency [SSKT22].

2.10 Tolerant Reader

The tolerant reader principle, as explained by Martin Fowler, is a crucial concept in the world of microservices, which aims at reducing the coupling between different parts of a system. While some degree of coupling is inevitable due to the need for services to communicate via interfaces, many teams exacerbate this coupling unnecessarily. In this context, Postel's Law [Pos81] can be applied, which states that a system should be as tolerant as possible when it comes to the input it receives, but as strict as possible when it comes to the output it produces. This means that a system should be able to handle input that is not exactly what it expects, by only using the parts of the input it needs, and ignoring the rest. In particular, this principle is relevant when services need to evolve over time. Because service definitions will inevitably need to change, it is important to design services in a way that minimizes disruption. Additionally, it is wise to have all communication with a service in one place which translates the external interface to the internal interface. This way, if the external interface changes, only the communication layer needs to be changed, and the internal interface can remain the same [Fow11].

CHAPTER 3

Related Work

In the ever-evolving field of microservices, numerous approaches have been proposed to tackle the challenges of communication. This chapter presents some of the most relevant approaches and compares them to ours.

3.1 Jolie

Jolie is a service-oriented programming language that is based on the service-oriented programming paradigm. Most traditional programming languages are designed for computations on a single machine and need different frameworks or libraries to create abstractions for communication and coordination between services. Jolie wants to solve this problem by providing native abstractions for managing these aspects and composing services. This way they want to change the way how developers think about services and make the development easier. To achieve these goals, they are heavily focusing on API design and contracts. As a result, it is possible to develop everything as a monolithic service, and if needed, some parts can be extracted and deployed as separate microservices on another machine. In terms of communication, they support synchronous as well as asynchronous, and it is protocol agnostic, with many mainstream protocols already included [GGM23]. A Jolie program consists of two parts, the deployment description and the behavior description. The deployment description is used to describe the interfaces. Further, this means what operations the service can handle, as well as the request and response types. Next, it is possible to define the protocol and port on which the service is listening. The behavior description is used to describe, as the name suggests, the actual behavior/functionalities of the service. This part consists of computations, but also communication with other services, which, as stated before, can be request-response aka. synchronous or one-way aka. asynchronous [SMMR16].

Although Jolie is a great tool, Casale et al. [CAvdH+20] criticized that the need for a custom interpreter could be a problem in the future due to vendor lock-in. Additionally,

there is no possibility to define dependencies for a function. This is a problem when functions are extracted to run as separate services.

While Jolie wants to rethink the way we formalize microservice definition and communication by introducing a new programming language, we opt for a more pragmatic approach by reusing existing technologies. First of all, it reduces the need for developers to learn a new language, which is not always easy, and often they do not want to. Second, it allows using existing libraries and frameworks. As this is not the case for Jolie, it is unlikely to introduce Jolie in an already existing project. We want to provide a solution where it is possible to describe the communication between services in a language-agnostic way, which then can be used to generate the code. In addition, we want to combine synchronous and asynchronous communication, so it is possible to let the developer or framework decide which communication mode should be used in a given situation.

3.2 Ballerina

Similar to Jolie, Ballerina is a programming language that is designed for writing network-distributed applications. In fact, it can be used to model applications that heavily rely on communication over the network. Yet it provides ways to orchestrate the system in a reliable way. The language has many great features like null-safety or explicit error handling. Another feature that is absent in most mainstream languages is that it can represent its code as a sequence diagram, to easier understand how the services interact. Although there exists an own virtual machine where Ballerina code can run, due to performance issues, it mostly runs on the JVM (Java Virtual Machine). This means that it is possible to extend it with Java code, but then the visualization with the sequence diagram falls apart [WSO23, WEPL18].

Considering these facts, the argument against Jolie is also true for Ballerina. It is a completely new programming language, which means that developers have to learn it. We also want to point out that by having the code of multiple services in one project, it contradicts some key principles of microservices. At which point is a system still a microservice architecture, and when is it a distributed monolith?

3.3 CORBA

A somewhat older approach is CORBA, the Common Object Request Broker Architecture, with its first version introduced in 1991 [Gro23]. One big goal of CORBA was to make communication possible between programs developed in different languages long before microservices were a thing, therefore, concepts like loose coupling were not considered [Hen06]. Communication is mainly done synchronously, but it is also possible to send async messages [A+98]. Although CORBA has been around for a long time, it is not particularly popular. This has to do with its complexity and problems with firewalls that established web services do not have [Mes12]. Henning [Hen06] stated that some of the initial object services specifications were not only complicated but also lacked any

practical use. Further, commercial use was quite expensive, and the learning curve was steep [Hen06].

Compared to our approach, CORBA was not designed with microservices in mind, and it does not support an event-driven architecture. Our goal is to make it way simpler to choose and switch between communication modes and make developing decoupled microservices as easy as possible, which is not the case with CORBA.

3.4 Microservice DSL (MDSL)

MDSL is a domain-specific language for describing microservices, which is an abstraction over other definition languages like OpenAPI, WSDL, or Protobuf. It supports defining data types which are the data transfer objects (DTOs), and operations which consist of an expected input and a provided output type. Additionally, it is possible to define the endpoint, the location and the communication protocol with protocol-specific details in a construct called provider. An operation can get bound to a provider, which means that the operation is available on the endpoint of the provider. The clients need to be declared separately, and they contain the information on which endpoint from which provider they want to use [Zim18].

One drawback of MDSL is that it is not possible to define asynchronous communication. Therefore, Liberali introduced an extension for MDSL called AsyncMDSL [Lib20], which is at the time of writing in a technical preview state. This extension allows defining asynchronous communication in a similar way as MDSL allows for synchronous communication and therefore is suitable for defining message-queue-based communication. The main difference is that there is no response defined, as the client does not get one back from the server. Moreover, it allows for defining a channel with additional properties for its intended usage like point-to-point or publish-subscribe and other useful properties like expiration time or quality of service. Message brokers like RabbitMQ or Kafka can be used as providers, which expose channels defined before. In the end, AsyncMDSL is used to generate an AsyncAPI definition, which then can be used to generate the code.

At first glance, MDSL and AsyncMDSL seem to be quite similar to our approach. But there are some key differences. First of all, there is still a strict separation between synchronous and asynchronous communication, which we want to avoid by allowing a unification of both communication modes. Second, clients and servers are defined separately, which is a bit confusing, as, in the end, an OpenAPI or AsyncAPI definition is generated, which does not have this separation anyway. This leads to unnecessary complexity and makes it harder to understand. We not only want to avoid this but rather add some syntactic sugar to make it easier to define common patterns like CRUD operations. As stated before, robustness is a key aspect of microservices, and therefore we want to add the option for defining aspects relevant to robustness like retries or timeouts.

3.5 MicroBuilder

MicroBuilder can be used to define and generate code for REST-API-based microservices. One part of MicroBuilder is the MicroDSL, a domain-specific language defined with the Eclipse Modeling Framework. The metamodel is defined in Ecore, and they provided a concrete syntax with Xtext. All the concepts are centered around RESTful communication between microservices [TDKA+17].

The second part of MicroBuilder is the code generator, which generates code for the microservices based on the MicroDSL definition. They call it MicroGenerator and the included code generators are created with Xtend. As the generated code is based on Spring Boot, the programming language in use is Java. Moreover, they make use of multiple tools for service discovery, load balancing, and resilient communication. The resulting code contains interfaces for a REST API, with insert, update, and delete operations. In their paper, they also included a table that compares how many lines of code they were able to save by using MicroBuilder [TDKA+17].

In comparison to our approach, MicroBuilder is fully focused on REST APIs. It is not possible to define asynchronous communication. MircoDSL also allows defining connection details like URLs and ports, which are not part of our approach, as we want to keep the communication details out of the definition. There are some similarities like MicroBuilder provides CRUD operations out-of-the-box. We will do this too, as we also want to reduce the amount of code the developer has to write. Continuing with the similarities, they provide some concepts for robustness like circuit breakers. However, they do not allow for configuring these aspects in their DSL.

CHAPTER 4

RobComDSL

In order to solve the stated problems, we propose a DSL for describing the services and the communication interfaces. We call this DSL RobComDSL. First, we present an abstract metamodel as a diagram to give an overview of how the DSL is structured and what it can describe. Then we show a concrete implementation of the DSL based on JSON, so existing tools and parsers can be utilized. Finally, we describe how the DSL is used to generate the source code for the services.

4.1 Metamodel

The metamodel depicted in Figure 4.1 shows the main concepts of the DSL. A service definition describes which data a particular service owns and which actions can be performed on it. The data is described by the model, which comprises a list of fields that in turn can be primitive types or child models. As a consequence, the data model is a tree structure similar to a document in a document database. This comes with all the advantages and drawbacks known from this kind of data storage. In particular, it is not possible to define relations between models apart from the tree-structured child-parent relations. More on this in Section 5.6. As can be seen in the figure, a service definition contains a second way to describe data. This definition is for data transfer objects (DTOs) which are only used to transfer data between services and do not get stored in the database. Basically, this is the main difference between models and DTOs. The actions, which are also a part of the model, have such DTOs as input and output parameters. CRUD (create, read, update, delete) operations are implicitly defined for every model. Every other operation needs to be defined as an action. In addition, the model contains a configuration comprising the description of which criteria in terms of robust communication have to be fulfilled when communicating with the service.

The configuration shown at the top of Figure 4.1 contains the own service definition, besides that it also contains a list of other service definitions of services it depends on.

These are the services, which a service wants to communicate with. As it may not be interested in all the models of a related service, it is possible to specify which data a service should receive from the other service. Moreover, a selection of fields can be supplied in order to only store a subset of the data. Lastly, the configuration contains a list of key-value pairs which can be used to configure how the source code of the service should be generated.

4.2 DSL Implementation

The DSL is implemented in JSON because it is a well-known and easy-to-use format. Yet it is easy to parse, there are many libraries for different programming languages available, and the tooling support is mature. For defining the structure of the JSON document, we use a simplified BNF (Backus-Naur form) notation. Besides other JSON specific details, it omits, for better readability, the definition of whitespaces or the definition of number representations.

The definition is split into multiple files. This is done due to the reason that the configuration is specific to a certain service, while the service definition needs to be shared between services. Every service has its own service definition which is used to generate the controllers and the models. In addition, every service can have multiple other service definitions, of services it depends on, and therefore are used to generate the clients. As a result, every service definition is stored in a separate file, so it is easy to share it by copying or linking the whole file and not having to copy parts of a file. These service definition files follow this structure:

⟨*ServiceDefinition*⟩ ::= \{
⟨*ServiceNameProp*⟩,
⟨*ModelsProp*⟩,
⟨*DTOsProp*⟩
}

⟨*ServiceNameProp*⟩ ::= `"serviceName":` ⟨*String*⟩

⟨*ModelsProp*⟩ ::= `"models": {` ⟨*Models*⟩ `}`

⟨*Models*⟩ ::= ⟨*Model*⟩
| ⟨*Model*⟩`,` ⟨*Models*⟩

⟨*Model*⟩ ::= ⟨*String*⟩`: {`
⟨*PluralNameProp*⟩,
⟨*FieldsProp*⟩
(`,` ⟨*ActionsProp*⟩)?
(`,` ⟨*ConfigProp*⟩)?
}

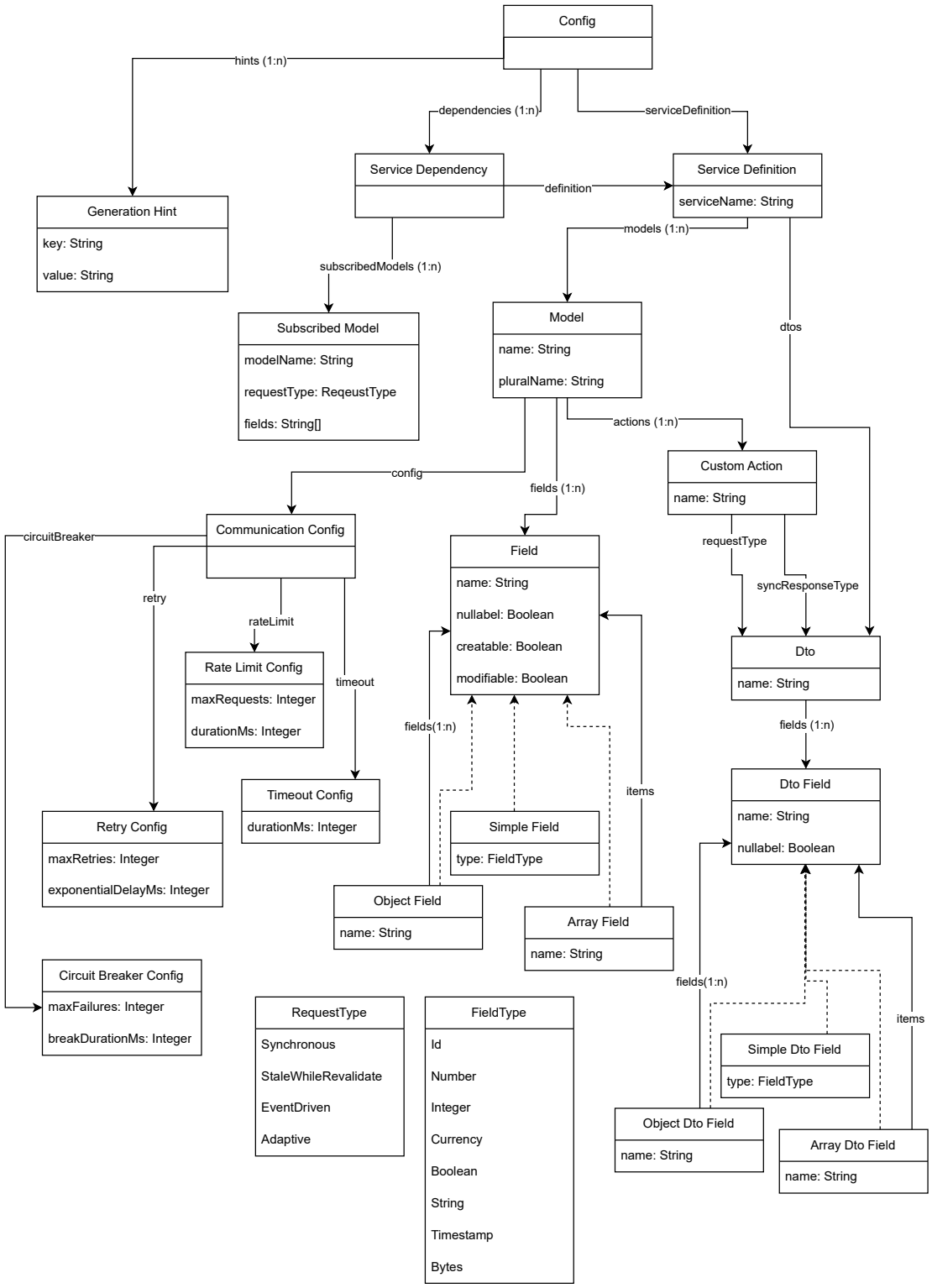

Figure 4.1: Diagram of RobCom metamodel

⟨*PluralNameProp*⟩ ::= `"pluralName":` ⟨*String*⟩

⟨*FieldsProp*⟩ ::= `"fields": {` ⟨*Fields*⟩ `}`

⟨*Fields*⟩ ::= ⟨*FieldProp*⟩
| ⟨*FieldProp*⟩`,` ⟨*Fields*⟩

⟨*FieldProp*⟩ ::= ⟨*String*⟩`:` ⟨*Field*⟩

⟨*Field*⟩ ::= ⟨*SimpleField*⟩
| ⟨*ObjectField*⟩
| ⟨*ArrayField*⟩

⟨*SimpleField*⟩ ::= `{`
⟨*TypeProp*⟩
(`,` ⟨*NullableProp*⟩)?
(`,` ⟨*CreatableProp*⟩)?
(`,` ⟨*ModifiableProp*⟩)?
`}`

⟨*ObjectField*⟩ ::= `{`
`"type": "object",`
⟨*NameProp*⟩`,`
⟨*FieldsProp*⟩
(`,` ⟨*NullableProp*⟩)?
(`,` ⟨*CreatableProp*⟩)?
(`,` ⟨*ModifiableProp*⟩)?
`}`

⟨*ArrayField*⟩ ::= `{`
`"type": "array",`
⟨*ItemsProp*⟩
(`,` ⟨*NullableProp*⟩)?
(`,` ⟨*CreatableProp*⟩)?
(`,` ⟨*ModifiableProp*⟩)?
`}`

⟨*TypeProp*⟩ ::= `"type":` ⟨*String*⟩

⟨*NullableProp*⟩ ::= `"nullable":` ⟨*Boolean*⟩

⟨*CreatableProp*⟩ ::= `"modifiable":` ⟨*Boolean*⟩

⟨*ModifiableProp*⟩ ::= `"modifiable":` ⟨*Boolean*⟩

⟨*ItemsProp*⟩ ::= `"items":` ⟨*Field*⟩

⟨*NameProp*⟩ ::= `"name":` ⟨*String*⟩

⟨*ActionsProp*⟩ ::= `"actions": {` ⟨*Actions*⟩ `}`

⟨*Actions*⟩ ::= ⟨*Action*⟩
| ⟨*Action*⟩`,` ⟨*Actions*⟩

⟨*Action*⟩ ::= ⟨*String*⟩`: {`
⟨*RequestTypeProp*⟩`,`
⟨*SyncResponseTypeField*⟩
`}`

⟨*RequestTypeProp*⟩ ::= `"requestType":` ⟨*String*⟩

⟨*SyncResponseTypeField*⟩ ::= `"syncResponseType":` ⟨*String*⟩

⟨*ConfigField*⟩ ::= `"config": {`
(⟨*RetryField*⟩)?
(`,` ⟨*CircuitBreakerField*⟩)?
(`,` ⟨*RateLimitField*⟩)?
(`,` ⟨*TimeoutField*⟩)?
`}`

⟨*RetryField*⟩ ::= `"retry": {`
`"max":` ⟨*Integer*⟩`,`
`"exponentialDelayMs":` ⟨*Integer*⟩
`}`

⟨*CircuitBreakerField*⟩ ::= `"circuitBreaker": {`
`"maxFailures":` ⟨*Integer*⟩`,`
`"breakDurationMs":` ⟨*Integer*⟩
`}`

⟨*RateLimitField*⟩ ::= `"rateLimit": {`
`"max":` ⟨*Integer*⟩`,`
`"durationMs":` ⟨*Integer*⟩
`}`

⟨*TimeoutField*⟩ ::= `"timeout":` ⟨*Integer*⟩

⟨*DTOsField*⟩ ::= `"dtos": {` ⟨*DTOs*⟩ `}`

⟨*DTOs*⟩ ::= ⟨*DTO*⟩
| ⟨*DTO*⟩`,` ⟨*DTOs*⟩

⟨*DTO*⟩ ::= ⟨*String*⟩`:` { ⟨*DtoFieldsProp*⟩ }

⟨*DtoFieldsProp*⟩ ::= `"fields":` { ⟨*DtoFields*⟩ }

⟨*DtoFields*⟩ ::= ⟨*DtoFieldProp*⟩
| ⟨*DtoFieldProp*⟩`,` ⟨*DtoFields*⟩

⟨*DtoFieldProp*⟩ ::= ⟨*String*⟩`:` ⟨*DtoField*⟩

⟨*DtoField*⟩ ::= ⟨*SimpleDtoField*⟩
| ⟨*ObjectDtoField*⟩
| ⟨*ArrayDtoField*⟩

⟨*SimpleDtoField*⟩ ::= {
⟨*TypeProp*⟩
(`,` ⟨*NullableProp*⟩)?
}

⟨*ObjectDtoField*⟩ ::= {
`"type": "object",`
⟨*NameProp*⟩`,`
⟨*DtoFieldsProp*⟩
(`,` ⟨*NullableProp*⟩)?
}

⟨*ArrayDtoField*⟩ ::= {
`"type": "array",`
⟨*DtoArrayItemsProp*⟩
(`,` ⟨*NullableProp*⟩)?
}

⟨*DtoArrayItemsProp*⟩ ::= `"items":` ⟨*DtoField*⟩

To define which service definition is the own one, and which are the dependencies, the configuration file is used. This is done by defining the file name, or list of file names respectively, of the service definitions. It also contains the definition, which models of a dependent service are used and which fields of the model should be stored. This determines how the clients are generated. Simply deleting the models or fields from the copied service definition might break the generated code and is therefore not recommended. For example, if protobufs are used, the index of the fields depends on the order in the service definition, hence removing a field would change the index of all following fields, and consequently communication between the services will not work. It would also complicate the versioning of the service definitions, as someone needs to remove all unused models and fields, every time the source service definition changes and gets copied to the dependent service.

Furthermore, it contains „generation hints“ which are used to configure the generation of the source code. As this is designed to hold implementation-specific information, it is not further specified how this should be used. Summing everything up, the configuration file has the following structure:

⟨*RobComConfig*⟩ ::= `{`
⟨*ServiceDefinitionProp*⟩
(`,` ⟨*DependenciesProp*⟩)?
(`,` ⟨*GenerationHintsProp*⟩)?
`}`

⟨*ServiceDefinitionProp*⟩ ::= `"serviceDefinition":` ⟨*String*⟩

⟨*DependenciesProp*⟩ ::= `"dependencies": [` ⟨*Dependencies*⟩ `]`

⟨*Dependencies*⟩ ::= ⟨*Dependency*⟩
| ⟨*Dependency*⟩`,` ⟨*Dependencies*⟩

⟨*Dependency*⟩ ::= `{`
⟨*DependencyServiceDefinitionProp*⟩`,`
⟨*SubscribedModelsProp*⟩
`}`

⟨*DependencyServiceDefinitionProp*⟩ ::= `"serviceDefinition":` ⟨*String*⟩

⟨*SubscribedModelsProp*⟩ ::= `"subscribedModels": [` ⟨*SubscribedModels*⟩ `]`

⟨*SubscribedModels*⟩ ::= ⟨*SubscribedModel*⟩
| ⟨*SubscribedModel*⟩`,` ⟨*SubscribedModels*⟩

⟨*SubscribedModel*⟩ ::= `{`
⟨*ModelProp*⟩`,`
⟨*RequestTypeProp*⟩
(`,` ⟨*FieldsProp*⟩)?
`}`

⟨*ModelProp*⟩ ::= `"model":` ⟨*String*⟩

⟨*RequestTypeProp*⟩ ::= `"requestType":` ⟨*RequestType*⟩

⟨*RequestType*⟩ ::= `"Synchronous"`
| `"StaleWhileRevalidate"`
| `"EventDriven"`
| `"Adaptive"`

⟨*FieldsProp*⟩ ::= `"fields": [` ⟨*Fields*⟩ `]`

⟨*Fields*⟩ ::= ⟨*Field*⟩
| ⟨*Field*⟩`,` ⟨*Fields*⟩

⟨*Field*⟩ ::= ⟨*String*⟩

⟨*GenerationHintsProp*⟩ ::= `"generationHints": {` ⟨*KeyValuePais*⟩ `}`

⟨*KeyValuePais*⟩ ::= ⟨*KeyValuePair*⟩
| ⟨*KeyValuePair*⟩`,` ⟨*KeyValuePais*⟩

⟨*KeyValuePair*⟩ ::= ⟨*String*⟩`:` ⟨*String*⟩

To give an example of what this would look like in practice, we show the service definition in Figure 4.2 for the OrderService introduced in Section 1.2.1. Further, the configuration for the OrderService is shown in Figure 4.3. The OrderService has one model called „order“ which has the fields „orderNumber“, „orderDate“, „customerId“ and „items“, which is an array of objects. In this example, the number is automatically generated and therefore cannot be set by the client and only the customer id is allowed to be empty, so guests can order too. In the configuration, it is defined that the OrderService depends on the ProductService and that it is interested in the „product“ model.

4.3 Code-Generation / Mapping

To make use of the DSL, we need a framework that parses the DSL and provides the necessary functionality. We decided to heavily rely on code generation, as it allows us to implement the framework for basically every programming language. The only alternative would be to use some kind of metaprogramming, which is not available in every language.

4.3.1 Models

With RobComDSL we can describe tree-structured models of a service, consequently, we need to recursively generate the model classes from the DSL. In order to do so, a new file is generated for every model. To keep it simple, the class name is the name of the model. Next, some default fields are added to the classes, which are an id, the created and updated timestamp, and the cumulative hash (Section 5.4.2). The generated classes also contain the fields of the models as attributes and getters and setters for them. The equals method is also generated, which compares the id of the model.

4.3.2 Change Detection Proxies

As the RobCom framework automatically publishes changes when a model is updated, it needs to know which fields have changed. To do so, proxies get generated which wrap the

```
{
  "serviceName": "OrderService",
  "models": {
    "order": {
      "pluralName": "Orders",
      "fields": {
        "orderNumber": {
            "type": "string", "modifiable": false, "creatable": false
        },
        "orderDate": { "type": "timestamp", "nullable": false },
        "customerId": { "type": "id" },
        "items": {
            "type": "array",
            "nullable": false,
            "items": {
                "type": "object",
                "name": "orderItem",
                "fields": {
                    "productId": { "type": "id", "nullable": false },
                    "quantity": { "type": "number", "nullable": false },
                    "price": { "type": "number", "nullable": false }
                }
            }
        }
        // ...
      }
    }
  }
}
```

Figure 4.2: Example service definition for the OrderService

```
{
    "serviceDefinition": "OrderService.json",
    "dependencies": [
        {
            "serviceDefinition": "ProductService.json",
            "subscribedModels": [
                {
                "model": "product",
                "requestType": "EventDriven",
                "fields": ["name", "price"]
                }
            ]
        }
        // ...
    ],
    "generationHints": {
        // ...
    }
}
```

Figure 4.3: Example configuration for the OrderService

model classes and track the changes. These proxies have the same attributes as the model classes. The getters simply return the value of the field, while the setters set the value and mark the field as changed. For child objects, the proxies also wrap the child object, so their changes can be tracked recursively. The most complex part of the proxies are fields that hold arrays of child objects. A replacement of an array can be detected easily, by the setter methods, and the modification of a child object can be detected by the proxy of the child object. Nevertheless, the addition or removal of a child object is more difficult to detect. As a consequence, the returned array needs to be wrapped in an array change detection proxy, which contains separate arrays for added and removed objects. All these change detection proxies have methods to check if changes have occurred and to get the changes. A simplified example of a change detection proxy is shown in Figure 4.4.

4.3.3 DTOs

To transfer data between services, DTOs are used. There are separate DTOs for creating, updating, deleting and reading a model and for a full response, a change response or a delete response. In the DSL, it is possible to define which fields are needed for which DTOs, by setting the creatable, or updatable attribute of a field. These fields are excluded from the respective DTOs. The default fields are again added to the DTOs like it is done for the models. It should be noted that the same DTO can and should be used for synchronous and asynchronous communication. As we decided to use protobufs

```
public class OrderChangeDetectionProxy: Order,
    IEntityChangeDetectionProxy<Order>
{
    private bool _orderDateChanged;
    // other changed flags for fields ...
    public Order Entity { get; init; }
    public bool HasChanged {
        get { return _orderDateChanged || ... }
    }

    public override DateTime OrderDate {
        get {
            return Entity.DateTime;
        }
        set {
            if(Object.Equals(Entity.OrderDate, value)) return;
            _orderDateChanged = true;
            Entity.OrderDate = value;
        }
    }
    // other properties ...

    public string GetCumulativeSha256Hash()
    {
        // see Algorithm 5.1
    }
}
```

Figure 4.4: Example Change Detection Proxy for Order Model

(Section 5.5) for the communication, the RobCom framework does not generate the DTOs directly but generates protobufs which are then used to generate the DTOs.

It is also possible to define DTOs yourself in the DSL, which then can be used for custom actions. These are also converted to protobufs and then to DTOs. In contrast to the other DTOs, these DTOs only contain the fields defined in the DSL and get no additional fields added.

4.3.4 Repositories

As the RobCom framework should be technology-agnostic, it is not possible to generate the whole repository. Rather, it generates an interface that needs to be implemented by the developer. This interface contains methods to create, read, update and delete a model. We give these interfaces the prefix „RepositoryInternal“ to indicate that they

should not be used directly. Instead, repository classes are generated which get the implementation of the internal repository injected and provide a public interface. By doing so, it is possible to wrap the objects returned by the internal repository in change detection proxies. For create and delete methods, the repository classes additionally publish a respective event to the message broker. As the read methods wrap the returned objects in change detection proxies, the update method can simply check if the object has changed. If this is the case, it saves the changes to the database by calling the internal repository. Importantly, it can create a change event from the change detection proxy and publish it to the message broker. This guarantees that all changes are automatically published, so it allows for event-driven communication between the services.

4.3.5 Clients

In the DSL it is possible to define dependent services and which models of these services are used. For every specified model, a client is generated which provides methods to create, read, update and delete the model as well as methods for all custom actions. For generating the read methods, it is also important to consider which request type is defined for the given model. What the different request types mean is explained in Section 5.1. Besides the method with the configured request type, the client also always provides a method that allows synchronous reading of the model, as this is always possible and can be used as a fallback. For other methods, including the custom ones, the client contains a synchronous and an asynchronous method. The synchronous method directly calls the service and returns the result. For create this is a full response, for update a change response, for delete a delete response, and for custom actions, it returns the response defined in the DSL. The asynchronous method, on the other hand, does not return the result directly but rather publishes the changes to the message broker. This allows for event-driven communication between the services.

To make the clients more robust, there is a communication config in the DSL that defines four robustness criteria. By default, the clients come with automatic retries and circuit breakers. The developer can additionally configure rate limiting and timeouts. Before sending a request, the client creates an idempotency key, so the receiver can detect duplicate requests.

4.3.6 RobCom Context

The RobCom context is the central class of the RobCom framework. It contains information about the request like if it is synchronous or asynchronous. It also provides repositories and clients for the services defined in the DSL. As a result, it can handle transaction management and trigger the publishing of events. Overall, it is essential to manage the communication between the services. A simplified example of a RobCom context is shown in Figure 4.5. It has public properties for the repositories and clients, so they can be accessed from the service layer.

```
public class RobcomContext: IAsyncDisposable
{

    private OrderContext _dbContext;
    private OutboxPublisherBackgroundService _publisherBgService;
    public RequestType RequestType { get; }
    public RobcomContextInternal Internal { get; }
    public IOrderRepository OrderRepository { get; }
    public ProductClient ProductClient { get; }

    public RobcomContext(CheckoutContext dbContext,
        OutboxPublisherBackgroundService publisherBgService,
        CancellationToken CancellationToken,
        ProductGrpcControllerClient productGrpcClient)
    {
        // ...
    }

    public void PublishEvents(List<OutboxEvent> events)
    {
        // ...
    }

    public async Task SaveChanges()
    {
        List<OutboxEvent> events = new List<OutboxEvent>();
        events.AddRange(
            await OrderRepository.SaveChangesAsync());
        await Internal.SaveChanges();
        PublishEvents(events);
        await _dbContext.SaveChangesAsync(CancellationToken);
        _publisherBgService.StartProcessingOutstandingEvents();
    }

    public ValueTask DisposeAsync()
    {
        // ...
    }
}
```

Figure 4.5: Example RobCom Context

4.3.7 Controllers

The RobCom framework generates controllers for every model. These controllers are the endpoints for synchronous communication and implement methods for creating, reading, updating and deleting a model, as well as for all custom actions. Before forwarding the request to the service layer, they first initialize the RobCom context. Next, it verifies if the request was already processed, by checking the idempotency key. If this is not the case, it hands the request to the service layer and then returns the result. Before sending the response, the idempotency key is saved in the database and the transaction is committed.

4.3.8 Listeners

For asynchronous communication, the RobCom framework generates listeners for every model. These are the endpoints for asynchronous communication and, like the synchronous counterparts, implement methods for creating, reading, updating and deleting a model, as well as for all custom actions. They also initialize the RobCom context and check the idempotency key. Then the requests get forwarded to the service layer, which is exactly the same as for synchronous communication. This is part of the reason why the RobCom framework provides a unified framework for synchronous and asynchronous communication. The main difference is that the listeners do not return a response, as it is not necessary for asynchronous communication. This is due to the fact that without regard to the request type, all changes are published to the message broker. So the response would contain the same information as the published event, which is redundant.

Another difference is that the listeners are also generated for dependent services and models. These, however, are only needed if the request type of the model is event-driven or adaptive. The listeners simply receive the events and update the database accordingly, so the data is ready when it gets requested. Afterward, these listeners also call the service layer, where the developer can add methods to react to the events.

4.3.9 Services

The service layer implements out-of-the-box methods for creating, reading, updating and deleting a model. This implementation is very basic and only saves the data to the database or reads it from it. To extend the functionality, the developer can override these methods and implement the desired functionality. For custom actions, it is always necessary to implement the method on your own.

Every method gets the request, no matter if it is synchronous or asynchronous, and the RobCom context as parameters. All calls to the database are done through the RobCom context, so it can handle transaction management and change tracking, which is essential for automatic event publishing. Moreover, all communication with other services is done through the RobCom context, as it knows the configured request type and can therefore decide if the request should be synchronous or asynchronous.

4.4 Implementation

We decided to implement the RobCom framework in C#, as we wanted to use the Pitstop project (Section 6.5.1) and the E Shop On Containers project (Section 6.5.2) to test and evaluate our DSL and framework. Why we have chosen these projects is explained in Section 6.3. Further, C# is a modern and widely used programming language and provides some libraries which we can use to build our framework upon.

Although C# has support for code generation in the form of source generators, which are executed at compile time and can also incrementally update the generated code [S+23], we decided to create a CLI tool that does the job. This is due to the fact that source generators can only generate source code, but we also need to generate protobuf files. Additionally, the option to generate source code with a CLI tool is available for every programming language, while source generators or similar constructs do not exist for every language.

For communication, we decided to use gRPC (Section 5.5) and rabbitMQ as they are widely used and have good support in C#. Due to these reasons, we used the libraries Grpc.AspNetCore, Grpc.Net.Client and RabbitMQ.Client which are all available on NuGet. In order to make the communication more robust, we used the library Polly (Microsoft.Extensions.Http.Polly) which provides automatic retry, circuit breaker and rate limiting. For timeouts, we could directly use the timeout of gRPC.

As both Pitstop and E Shop On Containers use Entity Framework Core as ORM, we decided to use it as well. Because the RobCom framework is technology-agnostic, it only provides interfaces for repositories and the developer needs to implement them. This means that the developer has to create a database context and then implement all the functions from the repository interfaces. As these implementations need the database context as a parameter in the constructor and RobCom generates code for initializing the repositories, it is required to know the type of the database context at generation time. This necessitated that the developer specifies the type of the database context and the according factory class in the DSL as generation hints.

Apart from this, the RobCom framework is implemented how it is described in Section 4.3. To make RobCom available as a CLI tool, we needed to set the options „PackAsTool“ to true, „ToolCommandName“ to „robcom“ and „PackageOutputPath“ to the path where the tool should be saved. Every time the content of the definition files changes, the developer needs to run the tool to generate the code.

CHAPTER 5

Design Decisions

This chapter describes how the RobCom framework works and enumerates the key aspects of it. Further, it explains why we made certain design decisions and what the benefits and drawbacks of these decisions are.

5.1 Querying Data

Querying data is a very important part of every application, and it is also a part where the use of RobCom can provide many benefits. They all come from the fact that the RobCom framework generates clients for all services defined as dependencies, which can be used to query data from subscribed models. An important goal of RobCom is to provide a unified way for communicating with other services, regardless of synchronous or asynchronous communication. Consequently, it provides multiple options for querying data. One thing to point out is that the business logic (the service layer) for the server, as well as for the client, can stay the same, regardless of the communication style, because the RobCom framework abstracts away the communication layer to provide a unified interface. The four different options for querying data are the following.

5.1.1 Synchronous Requests

The first option is to use the generated client to send a synchronous request to the server. Even if this is not the selected communication style in the RobCom configuration, it is always possible to use it, as it provides the following benefits:

- Most direct way to query data
- Data is as fresh as possible
- Easy mental-model

Although it is the most straightforward way to query data, it is also the least robust one, due to the temporal coupling it introduces. It only works if the server is available and has the capacity to handle the request. Next, the client has to wait for the response from the server, which can take some time, depending on the network latency and the load of the server. So the drawbacks are as follows:

- Temporal coupling
- Server can be overwhelmed
- Single request can trigger multiple subsequent requests
- Often used but rarely changed data leads to unnecessary communication

To combat some of the drawbacks, the RobCom framework provides the following options: By default, retries are enabled, so if a request fails, it is retried a certain number of times. The waiting interval between the retries is calculated by an exponential function $f(x, r) = x^r$, where x is the base and r is the retry count. The base as well as the maximum number of retries can be configured in the RobCom configuration. It must be noted that retries will increase the latency and the load on the server, so they should be used with care.

This leads to the next option, which is a circuit breaker (Section 2.9). It is also enabled by default and the number of failures before opening and the break duration can be configured in the RobCom configuration. The circuit breaker configuration could be more advanced, but we wanted it to be simple and easy to use.

An option for a timeout is present in RobCom too, but it is disabled by default. When configuring it, someone should take the retries into account, because if the timeout is too short, the request will fail before the retries are even started.

Last but not least, the RobCom framework provides a way to define rate limits for requests. These limits are meant to be enforced by the client. This decision is based on the fact that RobCom is intended to be used in a microservice environment, where multiple instances of the same service are running in parallel. Therefore, enforcing the rate limits on the server is not that easy and would require communication between the instances, which would add a lot of complexity to the framework.

5.1.2 Stale-While-Revalidate

The second option which can be configured in the RobCom config is to use the generated client to send a request to the server, but then use a stale-while-revalidate strategy to update the data. This is depicted by the state machine in Figure 5.1. Similar to other common implementations of this strategy, the data is missing in the beginning, so the client sends a synchronous request to the server. While the data is fresh, the client can use it without any further communication with the server.

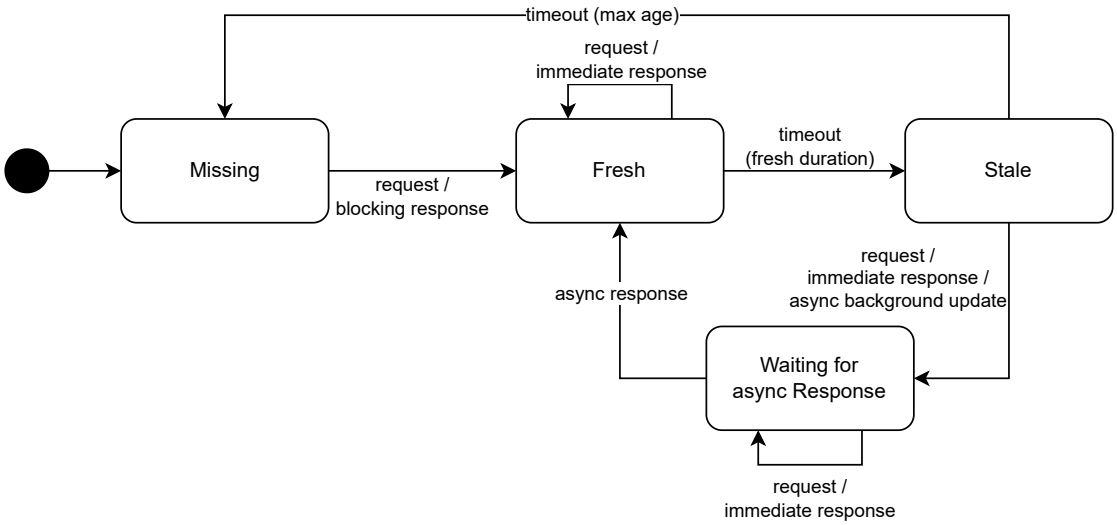

Figure 5.1: State machine for stale-while-revalidate

The difference to other implementations comes into play when the data is stale. Normally, the update request would be sent in the background, but this still would be synchronous. As RobCom unifies synchronous and asynchronous communication and uses the same DTOs and the same business logic for both, it is possible to send the update request asynchronously via a message broker. As the update is not time critical, we can afford to wait for a response message and then replace the data in the cache. Only when the data is older than the max-age, a synchronous request is sent again to the server. This strategy comes with the following benefits:

- Communication is async most of the time
- Lower risk of overwhelming the server
- Data access is fast
- No initial data transfer

Nevertheless, some requests still have to be sent synchronously, which can lead to the following drawbacks:

- Temporal coupling
- Many synchronous requests when cache times are configured suboptimal
- Often data is not fresh

5.1.3 Event-Driven

Instead of requesting data from the server, it is also possible that the server proactively publishes the data to a message broker and then the client can subscribe to the events. This is commonly known as an event-driven system and is a popular way for asynchronous communication between microservices. The RobCom framework provides a way to automatically publish events for changes to the data, but more on this will be discussed in Section 5.4. Further, the RobCom framework offers an automatic way to subscribe to these events and store the data in the database. In the configuration part of the RobCom definition, it is possible to specify the dependent services and which of their models should be subscribed to. Additionally, it is possible to declare which subset of fields of the model should be stored in the database. Due to this, it is easy to harness the benefits of event-driven systems, which are the following:

- Data is quite fresh
- Access to data is fast and no communication with the server is needed
- No temporal coupling
- Seldom changed but often accessed data does not lead to unnecessary communication
- Complex queries can be done on their own database

This concept is quite powerful and robust, but it also comes with drawbacks. Some of them are taken care of by the RobCom framework, but some are inherent to the concept:

- Implementation is more complex
- Manually publishing events is error-prone
- Data duplication
- Only works for new data

The first two drawbacks can be solved by the RobCom framework, as already mentioned it provides a way to automatically publish events and to subscribe to them. So the complexity is moved from the developer to the framework. Data duplication on the other hand is inherent to the concept and is the reason why this strategy is robust. The only measures against this are to strip down the data to the bare minimum of required fields, which, as already described, can be accomplished with RobCom. Nevertheless, for services that only need a small but potentially unknown subset of the data, it might be better to use another communication strategy. This is the reason why RobCom provides multiple options for querying data and wants to unify synchronous and asynchronous communication.

One problem with event-driven systems is that a new service needs to get all the data from the past as described in **ES2**, which can be a lot of data. There are multiple ways to solve this problem, like, for example, instruct a service to publish all its data to a message broker, so the newly subscribed service can store it in its database. This can potentially lead to a huge spike in the load of the server, so it should be done with care, or spread over a longer period of time.

5.1.4 Adaptive Data Querying

In order to combine the benefits of the strategies described above, the RobCom framework provides a way to adaptively query data. This is also a possible solution to the problem of how new event-driven services can get data from the past. The idea is to basically have an event-driven system, but when data is missing, the client can fall back to a synchronous request, which then gets stored. In conclusion, this is somewhat a combination of the stale-while-revalidate and the subscribing to events strategy. The following benefits can be achieved with this combination:

- Data is quite fresh
- Access to data is often fast
- seldom synchronous communication with the server is needed
- New services can get data from the past

With this approach, it is also possible to delete data from the own database, when it is unlikely that it will be accessed again. The decision on which data should be deleted needs to be made by the developer. As microservices should be small in terms of lines of code, it is reasonable to assume that a developer can deduce if there is some business logic in place that likely needs the data again. When this is not the case, the developer can create a cleanup function that deletes old/unnecessary data based on the deduced criteria. So the amount of duplicated data can be reduced while maintaining most benefits of an event-driven system. In the rare case that data needs to be accessed again, it will automatically trigger a synchronous request, and then the result will be stored in the database again. Despite all the benefits, this approach also comes with some drawbacks:

- Still some temporal coupling
- Still data duplication
- Developers need to decide which data should be deleted
- No complex queries on the database possible

Especially the last drawback could be a big one. As the data in the database might be incomplete, it is not possible to do complex queries on it and get the same results as if the query would be done on the server. In the end, it is a trade-off between data duplication and the ability to do complex queries on the data. Therefore, the developer needs to decide which strategy is the best for the use case at hand.

5.2 Mutating Data

Besides querying data, mutating data is also a very important part of every application. By mutating data, we mean creating, updating and deleting data, as all of them change the state of the application. These basic operations are created for every model out of the box, but it is also possible to define custom actions (Section 5.3), which can be used to mutate data. Similar to methods for querying data, the RobCom framework tries to provide a unified way for mutating data, regardless of the communication style. Further, the goal is to simplify the implementation by moving as much complexity as possible to the framework. As mutations are, as the name suggests, changing the state of the application, it is good practice to ensure that every mutation is only applied once. This is also known as idempotency and is a very important property of a robust system, hence RobCom has some mechanisms in place to ensure this property. Every request executed by the RobCom framework has a unique id, which is used to identify the request. When a request is sent to a service, the RobCom framework checks if the request-id is already known and if this is the case, the request is not executed again. Due to all these reasons, the same business logic can be used for synchronous and asynchronous communication.

5.2.1 Synchronous Mutations

The first option for mutating data is to use the generated client to send a synchronous request to the server. This is the most straightforward way to mutate data, and has some key advantages:

- Most direct way to mutate data
- Easy mental-model
- Instant feedback
- Knowledge that the request was successful
- Fixed order of execution

Especially the last two points are very important for many use cases. Many business processes are implemented in a way that some steps should only be executed if the previous steps were successful. With the example of an online shop, we introduced in the beginning (Section 1.2.1), we can create the following illustrative scenario: A customer

wants to buy a digital product that immediately gets downloaded after the payment was successful. In this case, there is a fixed order of execution. First, the payment has to be successful, then the product can be downloaded.

Besides the advantages, synchronous mutations also have some drawbacks which are quite similar to the ones of synchronous requests (Section 5.1.1), as they are inherent to the synchronous communication style:

- Temporal coupling
- Server can be overwhelmed
- Single request can trigger multiple subsequent requests

To deal with some of the drawbacks, the RobCom framework has some mechanisms in place, which are mostly described in Section 5.1.1. So rate limiting, timeouts and circuit breakers are equivalent to the ones for synchronous requests. This is also true for retries, but here, the fact that RobCom ensures idempotency comes into play. Without idempotency, retries could cause mutations to get applied multiple times, which could lead to incorrect data.

5.2.2 Message-Based Mutations

Alternatively to synchronous mutations, the RobCom framework also supports message-based mutations. This means that a client publishes the desired mutation to a message broker. The server is listening to the message broker, and when it receives a mutation, it applies it to its state. As always, the same services are used here, as are used for synchronous mutations. Nevertheless, there is one key difference, between the two communication styles. When the synchronous controller receives a mutation request, it hands it to the service, which returns a response to the controller, which then sends it back to the client. However, the asynchronous controller receives a mutation request, it also hands it to the service, which returns a response, but the asynchronous controller ignores the result and does not send any response back to the client. This is due to the fact that any changes to the data are published as events anyway, so the client can subscribe to them and get notified about the changes. How this is done is described in Section 5.4. Taking all this into account, the following benefits can be achieved with event-driven mutations:

- No temporal coupling
- Service can handle the request when it has capacity

As both types of mutations have vastly different properties and therefore, are suited for different use cases, both can be used in parallel. The decision on which one to use can be made by the developer, on a per-request basis. The downsides of event-driven mutations are basically the opposite of the benefits of synchronous mutations:

- No instant feedback
- No knowledge that the request was successful
- No fixed order of execution
- Eventual consistency
- No trivial way to give feedback to the user

There is not much that can be done about these points, as they are inherent to the asynchronous communication style. One option to tackle the last point would be to use optimistic updates. This means the data gets updated immediately on the client, and then the mutation request is sent to the server. If the request fails, the data on the client gets reverted to the previous state. This way the user gets instant feedback, but it is not guaranteed that the mutation will be applied. Therefore, this approach is only suited for use cases where it is not that critical that the mutation is applied, hence we decided against implementing it in the RobCom framework.

5.3 Custom Actions

Although the RobCom framework provides an out-of-the-box way to create, read, update and delete data, there are many use cases where this is not enough. To tackle this problem, the RobCom framework provides a way to define custom actions, which can be used to implement more complex business logic. These actions can be defined in the RobComDSL as part of a model, and then the framework generates the according source code for the synchronous controllers and asynchronous listeners for the server.

For all subscribers, the RobCom framework generates the according source code for the client. So the communication layer is abstracted away and unified. As a result, the developer can focus on the business logic, which is implemented in the service layer. As already mentioned, the RobCom framework creates DTOs for CRUD operations out of the box, but for custom actions, the developer has to define the DTOs separately, to allow for maximum flexibility. Every custom action has a DTO as an input parameter which stays the same for both communication styles.

On the other hand, the output DTO is only defined for synchronous communication, as the asynchronous communication style does not return a response itself. This decision is based on the fact that changes are published as events anyway (Section 5.4), so the client can subscribe to them and get notified about the changes. All other aspects in terms of robustness are handled by the RobCom framework the same way as for CRUD operations.

5.4 Automatic Publishing / Subscribing of Events

RobCom is intended to allow for event-driven communication between microservices while moving as much complexity as possible to the framework. Due to the fact that event-driven systems are all about publishing and subscribing to events, the RobCom framework provides a way to automatically, and robustly, execute these tasks, while trying to keep the communication between the services at a minimum.

5.4.1 Publishing Events

The RobCom framework adds a layer between the service layer and the persistence layer. So it is capable of intercepting all mutations to the data. When a service has finished its calculations and wants to persist the changes, it hands the data to the RobCom framework. The framework then checks which fields have changed, which is described in detail in Section 5.7. This is done for multiple reasons:

- Reduce writes to the database
- Reduce the number of events published
- Reduce the size of the events

If the change tracking indicates that a model did not change, the RobCom framework does not need to persist the data nor does it need to publish an event. By exactly knowing which fields have changed, as a consequence, it is possible to publish events that only contain the changed fields. The receiver needs to apply the changes to its model, but this can lead to incorrect data if some events in between were missed, or if the events were not received in the correct order. To tackle this problem, the RobCom framework adds a hash field to every model, so the receiver can check if its model has the same state as the sender. Keep in mind that all this is automatically done by the RobCom framework, so the developer does not need to worry about it.

As mentioned before, a dependent service can decide to only store a subset of the fields of a model. This, however, renders the hash field useless, as the receiver does not know the state of the fields it has not stored.

5.4.2 Cumulative Hash

The problem described before can be solved by using an algorithm that we call cumulative hash. It is designed to compute a hash value which can be used to check if two models have the same state, after applying a mutation, even if one only stores a subset of the fields.

The Algorithm 5.1 has one input which is the root object of the changed model. It then iterates over all child nodes (the fields) of the node and checks if they have changed. Depending on the type of the node, it then adds the name of the node and the value

to a set of values. If the child node is an object or an array, the algorithm is called recursively. In the case of an object, it does not suffice to only check if the field has changed, rather it is also necessary to check if the object itself has changes. This check is also done recursively. For arrays, there are two scenarios that need to be considered. If the array itself has changed, the whole array needs to be replaced, so the CumulativeHash algorithm is called recursively for every element of the array. The other case, where the array stays the same, but some elements have changed, got added or removed, allows for a smarter approach. The update event can contain three lists, one for added elements, one for modified elements and one for deleted elements. To reflect this in the hash value, the algorithm creates three sets and then adds the hash values of the elements to the corresponding set.

When an object is created, all fields are marked as changed and therefore the same algorithm can be used to create the initial hash value. The previous hash value is the first value in the vector of values, which is empty for created objects. As depicted, the algorithm depends on a hash function $hash(\vec{v})$ which takes a vector of values as input and returns a hash value.

5.4.3 Subscribing to Events

By including the cumulative hash, a receiver can check if its model has the correct state. Failing this, it can send an asynchronous request to the sender to get the correct state. As it is not possible to know which fields are incorrect, the sender has to send the whole model. This, however, should be an exception, therefore, it should not happen often and not introduce too much overhead.

As the receiver might only store a subset of the fields, a special algorithm is needed to merge the received event with the existing model and calculate the resulting cumulative hash which is needed for comparison. Such an algorithm is depicted in Algorithm 5.2. It takes a node and a change event as input and returns the updated node and the cumulative hash value. For primitive fields, the algorithm simply updates the value and adds the name and the value to the vector of values. For objects, it is straightforward too, as the algorithm is called recursively. For arrays, there are two scenarios to consider: If the array itself has changed, it is simply replaced by the new array. In the other case, where elements have been added, modified or deleted, the algorithm iterates over the lists and applies the changes to the array. For modified elements, the CumulativeApply algorithm is called recursively.

In the current version, only the hash value of the root node is compared, as only root nodes are published as events, and their children as a part of them. A way to optimize this would be to also compare the hash values of all child nodes. This would allow for knowing which child node has missing or incorrect data, and then it would be possible to only update this one. As the RobCom framework does not allow for requesting child nodes, this is not the case. It could be changed, but this would add a lot of complexity

Algorithm 5.1: CumulativeHash

Input: A tree node N
Output: cumulative hash value Σ

```
1  s⃗ ← {N.CumulativeHash};
2  forall child nodes C of N do
3      if isPrimitive(C) ∧ hasChanged(C) then
4          s⃗ ← s⃗ ∪ C.Name ∪ C.Value;
5      end
6      if isObject(C) ∧ (hasChanged(C) ∨ hasChanged(C.Value)) then
7          s⃗ ← s⃗ ∪ C.Name ∪ CumulativeHash(C.Value);
8      end
9      if isArray(C) then
10         if hasChanged(C) then
11             s⃗ ← s⃗ ∪ C.Name;
12             forall elements E of C do
13                 s⃗ ← s⃗ ∪ CumulativeHash(E);
14             end
15         end
16         else if hasAnyChanged(C.Value) then
17             a⃗ ← ∅;
18             m⃗ ← ∅;
19             d⃗ ← ∅;
20             forall elements E of C.Value do
21                 if isAdded(E) then
22                     a⃗ ← a⃗ ∪ CumulativeHash(E);
23                 end
24                 if isModified(E) then
25                     m⃗ ← m⃗ ∪ CumulativeHash(E);
26                 end
27                 if isDeleted(E) then
28                     d⃗ ← d⃗ ∪ CumulativeHash(E);
29                 end
30             end
31             s⃗ ← s⃗ ∪ C.Name ∪ a⃗ ∪ m⃗ ∪ d⃗;
32         end
33     end
34 end
35 return hash(s⃗);
```

to the framework, and therefore we decided against it. In addition, it could lead to more communication when multiple child nodes are missing or are incorrect.

5.5 gRPC

In Section 2.3, we compared different synchronous communication protocols. Based on this, we decided to use gRPC as the communication protocol for RobCom, due to its higher efficiency. So the RobCom definition is used to generate the according to protobuf files, which then are used to generate the source code for the gRPC client and server stubs as well as the messages (DTOs). One advantage of this is that in RobCom we only have to define the model and which of its fields should be accessible or modifiable by other services, and then all the different messages for CRUD operations are generated automatically. As a matter of fact, this approach not only reduces the amount of configuration code (like RobComDSL or protobuf) that has to be written, but it also makes it easier to keep fields for different operations in sync, because they are generated from the same source. For example, to add a description field to a product in RobComDSL we only have to add it to the model and then regenerate the protobuf files. When writing the protobuf files by hand, we would have to add the field to the messages for create, read and update operations, which is more error-prone.

5.5.1 gRPC for Async Communication

By going with gRPC, we can also take advantage of the more efficient protobuf serialization, compared to JSON or XML, for asynchronous communication. As one of the big goals of this thesis is to provide a unified framework for synchronous and asynchronous communication, this step makes total sense. So instead of encoding the messages in JSON and publishing them to a message broker, we can use the DTOs generated from protobuf to convert the data to a binary format and then publish the bytes directly to the message broker. This not only makes communication more efficient but also reduces the amount of code that has to be written, because the same logic, mappers and so on, can be used for synchronous and asynchronous communication. A service does not need to know if a request was sent to it via gRPC or a message broker, because the DTOs are the same. Only the communication layer has to be implemented for both communication styles, which is taken care of by the RobCom framework. As soon as the communication layer receives a message via either channel, it can pass it to the service and then send the response back to the caller via the same channel.

5.5.2 Tolerant Reader Problem

We already mentioned the tolerant reader principle in Section 2.10, but in short, it means that a service should be able to read messages, even if the message contains fields that the service does not know. Yet, in the way RobComDSL is designed, and due to the use of protobuf, this property is not guaranteed. This is owed to the fact that protobuf encoded data does not contain any information about its structure, hence the smaller

Algorithm 5.2: CumulativeApply

Input: A tree node N and a change event Δ
Output: A updated node N' and the cumulative hash value Σ of the node

$\vec{s} \leftarrow \{N.CumulativeHash\}$;
$N' \leftarrow$ copy of N;
forall *child nodes* C_Δ *of* Δ **do**
  $C \leftarrow \emptyset$;
  if $C_\Delta.Name \in C.Names$ **then**
    $C \leftarrow N.GetChild(C_\Delta.Name)$;
  end
  $C_{old} \leftarrow copyofC$ **if** $isPrimitive(C)$ **then**
    $C.Value \leftarrow C_\Delta.Value$;
    $\vec{s} \leftarrow \vec{s} \cup C.Name \cup C_\Delta.Value$;
  end
  if $isObject(C)$ **then**
    $(C.Value, \Sigma) \leftarrow CumulativeApply(C.Value, C_\Delta.Value)$;
    $\vec{s} \leftarrow \vec{s} \cup C.Name \cup \Sigma$;
  end
  if $isArray(C_\Delta)$ **then**
    $\vec{s} \leftarrow \vec{s} \cup C_\Delta.Name$;
    if $isReplace(C_\Delta)$ **then**
      $C.Value \leftarrow C_\Delta.Value$;
      forall *elements* E_Δ *of* $C_\Delta.Value$ **do**
        $\vec{s} \leftarrow \vec{s} \cup CumulativeHash(E_\Delta)$;
      end
    end
    else
      forall *added elements* E_Δ *of* $C_\Delta.Value$ **do**
        $C.Value \leftarrow C.Value \cup E_\Delta$;
        $\vec{s} \leftarrow \vec{s} \cup CumulativeHash(E_\Delta)$;
      end
      forall *modified elements* E_Δ *of* $C_\Delta.Value$ **do**
        $m \leftarrow C.Value.GetItem(E_\Delta)$;
        $(m', \Sigma) \leftarrow CumulativeApply(m, E_\Delta)$;
        $C.Value \leftarrow C.Value \setminus m \cup m'$;
        $\vec{s} \leftarrow \vec{s} \cup \Sigma$;
      end
      forall *deleted elements* E_Δ *of* $C_\Delta.Value$ **do**
        $C.Value \leftarrow C.Value \setminus E_\Delta$;
        $\vec{s} \leftarrow \vec{s} \cup CumulativeHash(E_\Delta)$;
      end
    end
  end
  $N' \leftarrow N' \setminus C_{old} \cup C$;
end
return $(N', hash(\vec{s}))$;

size, so the receiver can only make sense of the data if it knows the protobuf definition of the message. Every field in a protobuf file has a unique id, which is used to identify the field in the encoded message. As long as these ids stay the same, it is possible to add new fields to a message, without breaking the compatibility with readers with older versions of the message [Goo23b]. It is even possible to reserve ids for future fields so that they can be added later. When using RobComDSL the protobuf files are generated from the RobCom definition. The fields in a message get ascending ids, based on the order they are defined in the RobCom definition. Therefore, adding or changing a field in the RobCom definition can break the compatibility with older versions of the message and dependent services need to be updated.

There are multiple ways to solve this problem. The first one would be to only add new fields to the end of the RobCom definition so that the ids of the existing fields stay the same. As the RobCom framework adds further fields to messages, like ids, it is important that these fields are only added at the beginning of the message.

Another solution would be to add an index field to fields in the RobCom definition, which then gets mapped to the id field in the protobuf definition. This solution would be more robust, but as RobCom is intended to be protocol agnostic, it would be a violation of this principle because this index field is only used for protobuf, but not JSON or XML. Of course, using a different serialization format, like JSON or XML, which contains information about the structure of the data, would also get rid of this problem.

A more involved approach would be to use the reflection feature of gRPC. It allows a client to query the server for its protobuf definition. Based on this information, the client can adjust itself and only read the fields that it knows. This would be a very robust solution, but in return would add a lot of complexity to the framework. Additionally, the client would need to know when to query the server for its definition, as between the point in time when the server changes its definition and the client queries it, the client would not be able to read the messages from the server. This can be solved by publishing an event when the definition changes, or to automatically query the definition when the deserialization method of the client fails. Also, adding a version number to every message would do the trick.

5.6 Document-Based Data-Structure

The models in RobCom can only be defined in a tree structure. This means that for every data item, there is only one root element, which can contain other elements, which then can contain other elements and so on. Consequently, only one-to-one and one-to-many relations can be defined in a parent-child manner. Only the root elements can be accessed directly, all other elements can only be accessed via their parent. This is also common for document databases, like MongoDB. In contrast to document-based databases, relational databases can have relations between tables that can be used to join data, and they can be automatically enforced by the database due to the use of foreign keys. The rationale

behind the decision to only allow tree-structured data is based on the following arguments and the arguments in Section 5.7:

5.6.1 Relational Data in Document-Based Databases

As RobCom is intended to be technology-agnostic, it should be possible to use it with many different types of databases. In particular, it should support the use of document-based databases, like MongoDB, or relational databases, like PostgreSQL. The main selling points of relational databases are their ACID (atomicity, consistency, isolation, durability) properties, which guarantee, among other things, that the data is always in a consistent state. This involves that foreign keys are checked when inserting or updating data so that only valid relations can be created. In a document-based database, this is not the case because there are no relations between documents. Of course, it is possible to store the id of a document in another document, but the database does not know about this relation and therefore cannot check it. All the checks have to be done by the application logic.

If RobCom were designed in a way so that the data is of relational nature, it would be hard to use it with document-based databases, because the RobCom framework would have to implement the checks that the database would normally do. This, however, is just possible to a certain degree, as it would necessitate that there is some distributed locking mechanism in place, so the checks can be done consistently, even when multiple instances of the same service are running. Besides, the fact that this would add a lot of complexity to the framework, it would also blow the scope of this framework.

5.6.2 Document-Based Data in Relational Databases

Ideally, microservices built with RobCom should use document-based databases because they are, for obvious reasons, a better fit for the document-based data structure. Nonetheless, it should also be possible to be used with other types of databases, like relational databases, as maybe someone needs to use a relational database for some reason. In this case, the data has to be mapped to a relational data structure. Starting with the root document and recursively walking through every sub-document in the tree structure, a table for every document has to be created. The fields of the document then become the columns of the table, and every child document gets a foreign key to its parent document. This way one-to-one and one-to-many can get produced, which is exactly what is needed to simulate the document-based data structure. In document-based databases, the relations between parents and children do not need to be checked as they are fixed by the data structure. In relational databases, however, this is not the case, but with the use of foreign keys, the database can check the relations and so both types of databases will behave the same in this regard.

Querying the data is also possible, but it is more complicated and has more overhead than document-based databases. In order to get all the data of a document, all the tables that are involved in the tree structure have to be joined. This could be done by using

the repository pattern, where only every root document has its own repository. In the query methods of the repositories, the joins need to be defined, so that the data can be queried as a whole. The same applies to inserting, updating or deleting data.

5.7 Tracking Changes

A major goal of RobCom is to provide an out-of-the-box solution for publishing changes to the data, which is discussed in Section 5.4. To facilitate this feature, the RobCom framework needs to know which data has changed, so it can publish the changes. For fields with primitive types, this is trivial, as there can be a simple flag in place for every field, which indicates if the field has changed. This simple solution falls apart when relations between models are introduced.

5.7.1 Relations without restrictions

We tried to solve this problem for all relations in general, without any further restrictions, similar to the way how data is organized in a relational database. The idea was to keep track of any relations that are added or removed to or from a model. Besides the fact that this is not that easy to accomplish, many questions arise:

1. How should the event look like?
2. For which of the two models should an event be published when a relation is added or removed?
3. What happens for relations between a model of the own service and a model of a dependent service?

We thought that the event should contain the id or the list of ids of the related models. Although this would not be too hard to implement, it could lead to poor performance. For example, a service wants to read an order and all its items. It first needs to query the order from the server, which needs to load the order and all the items from the database to compute the list of ids. Then the service needs to query all the items from the server, which needs to load the items from the database again. As a consequence, this is not a good answer to the first question. Implementing a way to tell the RobCom framework which relations should be included in the event could be a better solution, but definitely a more complex one.

The second question is a bit easier to answer. A common way to solve this problem is to publish separate events for changes in relations. This would also cope with the situation where a dependent service only subscribes to one model of the relation, but not the other. However, creating an order with three items would lead to seven events. One for the order, three for the items and three for the relations between them. This is not only a lot of overhead, but it also makes it hard to implement as the events could arrive out of sync.

Due to the different ways of querying data, the third question involves many aspects. Storing the connection between two models as a relation in a relational database would only work if the query type is event-driven. In the other three modes, the data is not or not completely stored in the database, so the database would not be able to enforce the relation.

In conclusion, it is not feasible to allow for arbitrary relations between models, as it would add a lot of complexity to the framework. Hence, we decided to only allow for tree shaped relations, which are described in the next section.

5.7.2 Tree shaped Relations

To ease the problem, we added some restrictions on how relations can be modeled with RobComDSL. By only allowing parent-child relations, and therefore getting a tree-shaped data structure, many aspects discussed in the previous section can be solved. First of all, events are only published for the root elements. This not only avoids the n + 1 problem but also reduces the number of events that need to be published. Further, changes can be determined by recursively walking through the tree structure and checking if any of the fields have changed. This, of course, requires that every node knows which of its own fields have changed, but this can be accomplished by using the same mechanism as for the root elements, by having a flag for every field. In the change event, only fields that have changed are included, so the event is as small as possible. For lists, it is necessary to track if elements were added or removed. Then the change event can contain separate lists for added, removed and changed elements.

Any other relations between models can be handled by storing the ids of the related models, but the responsibility to keep the data consistent lies on the developer. This is a valid trade-off, as document-based databases work the same way.

CHAPTER 6

Evaluation

When using the engineering research methodology, it is common to first define the problem, then find a solution and finally evaluate it. Points one and two were already discussed in detail in previous chapters. Hence, in this chapter, we analyze multiple open-source projects which implement microservice architectures, in order to evaluate our approach. The findings of this analysis will be part of the answer for **RQ1**, where we stated that we want to analyze the mistakes made in the communication between the services. The rest of **RQ1** was already discussed in detail in the Chapters 4 and 5. For **RQ2**, we analyze the projects with regard to compatibility with our framework. We try to find the limitations of our framework and what types of microservices can be implemented with it. The results of this analysis are described in Section 6.1.

6.1 Overview

An overview of the evaluated projects can be seen in Table 6.1, where x means that it is used for all communication, o means that it is used for some communication and blank means that it is not used. The evaluation also showed us for what types of microservices our framework is suitable (**RQ2**): We can say, our framework is a good fit for microservice architectures that use synchronous communication, as it makes it easy to add timeouts, retries and circuit breakers. Additionally, it introduces asynchronous communication, and still uses the same interfaces and business logic for both types of communication. Due to our design decisions, it works with document-based data structures but is not suitable for microservices that rely on relations between the data. For new projects, this is not a big problem as the system can be designed with this in mind, but for existing projects, it is a problem, which limits the applicability of our framework. In the current state, we also cannot generate code for event sourcing, so it is not suitable for projects using this concept. We had one example of this, the Pitstop project and we could still use our framework, but we had to remove the event sourcing.

As we mainly focus on inter-service communication, we did not look into how the communication between the frontend and the services is done. So our framework cannot generate code for the frontend as this typically has no direct access to the message broker and rather uses synchronous communication. Nevertheless, it is possible to use the generated protobuf files for the frontend and use them to generate the communication code.

Some services used external services which were not part of the project, like a mail service or a payment service. As a matter of fact, RobCom does not provide any support for this kind of communication. In summary, our framework is best suited when it is used for new projects, with document-based data structures and designed in a way so most communication can happen asynchronously but some communication still needs to be synchronous. Further, it necessitates that both sides use the framework, so it can control all the communication between the services.

Another finding we want to mention concerns the cumulative hash Algorithm 5.1. As one of our goals was to reduce the communication overhead by only sending the changed data, we needed some way to detect if the data is equal on both sides (sender and receiver). In particular, the data would differ if events got lost or processed in the wrong order. This can simply be solved by using a hash value as a checksum. However, as the receiver is allowed to only store a subset of the fields, a hash would not work. Hence, we introduced the cumulative hash algorithm. Although the algorithm works for our use case, the evaluation showed that there is a less complex solution to this problem with different trade-offs. By adding a version number to each data item that gets incremented for every published change, we can check for missed, or out-of-order processed events. This change would save on processing power, but we lose the ability to compare checksums to detect other changes in data.

6.2 Robustness and Efficiency Criteria

Based on our research, we always favor asynchronous communication over synchronous communication, as it is more robust and efficient due to decoupling. We also look into how much effort it is to publish and subscribe to events, and if the outbox pattern is used, to allow for transactional publishing of events. Another important aspect is that the interfaces are defined in some technology-agnostic way so that they can be generated for different languages. This not only reduces the amount of code that must be written and maintained but also makes communication more robust, as it suppresses errors like wrong field names or missing fields.

When using synchronous communication, we argue that timeouts and circuit breakers are a must-have, as they can prevent services from getting overwhelmed, and by doing so, can prevent cascading failures. We can put rate limiting in the same category, but this requires the developer to know in advance how many requests the service can handle.

Next, we look for constructs that allow for idempotent communication, as this can prevent

Projects	Asynchronous	Event-Sourcing	Outbox	Synchronous	Retry	Circuit-Breaker	Timeout	Rate-Limit	Idempotency	Interface Generation
Acme Air				x						
Spring Cloud Example Project				x		x				x
Sock Shop	o			x		o	o			
Staffjoy				x			o			x
Pitstop	x	o		x	o					
E Shop On Containers	o		x	x	o	o	o		o	o
Project with RobCom	x		x	x	x	x	x	x	x	x

Table 6.1: Robustness attributes of the evaluated projects

requests from being executed multiple times. This aspect is especially important when using retries because they can lead to duplicate requests. Also, caching can be used to improve the efficiency of the communication.

6.3 Selection Criteria for Projects

In order to find a suitable set of projects, we first examined the existing research in the field of microservices and looked at the microservice projects they used. Additionally, we searched for open-source projects which are commonly used as examples of microservice architectures.

Inspired by the work of Aderaldo et al. [AMPJ17], we defined the following criteria for the selection of projects, which we will use to evaluate our framework, by replacing the communication with our framework. Other projects, although not fulfilling all criteria, will be used to evaluate the framework, too, but only conceptually. This means that we will only analyze them and show what would change if we would replace the communication with our framework. As our requirements for the projects are different from those defined by Aderaldo et al., we selected the criteria which fit our needs and then extended them with our own. The result is shown in Table 6.2. Especially, the point of only one language used is important, as we only plan to implement the framework for a single language. Based on the found projects, we will select one language which we will use for the implementation of the framework and only projects written in this language will be used for in-depth evaluation.

Criteria	Description
Explicit Topological View	The project has an explicit topological view of its services.
Easy Access from a VCS	There should be a publicly accessible repository hosted on a version control system like GitHub or Bitbucket.
Container Orchestration	The project should support container orchestration (preferably Docker), to easily set up the project locally.
Community Usage and Interest	The project should be easy to use and attract the interest of its target research community.
Language	The project is written in only one language, to keep overhead low.
Version and Compatibility	The project is compatible with the current version of the language and uses state-of-the-art libraries and frameworks.
Number of Microservices	The project has a low number of microservices (<10).

Table 6.2: Criteria for the selection of microservice projects

6.4 Conceptual Evaluation

We analyzed the following projects and found that they did not fulfill all the criteria defined in Table 6.2. Nevertheless, we still want to use them for a conceptual evaluation, as they are good examples of microservice architectures.

6.4.1 Acme Air

This project is an online store for a fictional airline company, which is published on GitHub [TS15]. It claims to be scalable to serve billions of requests per day. Unfortunately, it has neither an explicit topological view nor does it contain any Docker files. The last commit was in 2015. By analyzing the code, we saw that the communication is only done via REST, and we could not find any constructs in place that would make the communication more robust. Confusingly, Adelardo et al. [AMPJ17] stated that it has at least circuit breakers in place, and it has reusable Docker images. Yet we could not find any signs of this.

The project is written in Java and Node.js, hence it is not suitable for further evaluation, as stated before. Due to the lack of an explicit topological view, or any other detailed documentation, it is hard to comprehend the project and completely understand the communication between the services. Nevertheless, when creating an implementation for RobCom for Java and Node.js, we think that it would drastically improve the robustness of the communication between the services, as already said, there are currently no

constructs in place to make the communication more robust. Due to the fact that the project already uses MongoDB for data storage, it would be easy to adapt it to the RobCom framework, because it only allows document-based data structures.

6.4.2 Spring Cloud Example Project

As an example of a microservice architecture, this project is a simple application for recommending movies, published on GitHub [Bas19]. All the services can be deployed with Docker, and they use Eureka as an API gateway and for service discovery. Due to the fact that the purpose of this project is to show how a microservice architecture can be developed with the Spring framework, every service is written in Java. When ignoring the API gateway, service discovery, and other services which do not have anything to do with the business logic, 5 microservices remain, according to their topological view. This seems to be a good fit for our purpose of evaluating our framework, as the quantity of services is acceptable and all of them are written in a single language. There is kind of an explicit topological view, but it only shows the connection to the different databases and not the communication between the services, which is the main focus of our evaluation. The last commit, except for an update of the license and readme, was in 2016.

The analysis of the code showed, however, another picture. We did neither find the depicted rating service nor the analysis service, but we found a UI that is not mentioned in the topological view or the readme. The movies, recommendation and users microservices offer only a REST API, which in turn only gets used by the UI. There is no other communication between the services. On the positive side, this project uses Hystrix [C+18], which is a library developed by Netflix, for circuit breaking and is not under active development anymore. Other than that, there are no constructs in place to make communication more robust, and they do not use any form of asynchronous communication.

By using the FeignClient library and the way they created their REST controllers, they do not need to write any code for the communication between the services, similar to our framework. This not only reduces the amount of code that must be written and maintained but also makes the communication more robust, because as long as the code library has no bugs, there cannot be any errors in the communication code, like wrong interfaces. In any other aspect, the communication is not robust and would benefit from our framework. However, as there is no communication between the services, there is no point in evaluating our framework with this project.

6.4.3 Sock Shop

This project is an online store for selling socks. The project was published on GitHub [C+21] under the name microservices-demo. There exists a topological view of the services, however, it looks like only the frontend communicates with the services. Only for the order, shipping and queue-master service, the communication is shown. But in reality, there is more communication between the services, as we found out by analyzing the

code. Some services were updated quite recently, but it looks like these are only bug fixes or maintenance work. Further positive aspects are that the project uses Docker for all services. There is some basic documentation and there exists a talk about the project. It also has over 3.5k stars on GitHub and is used in some recent research papers [RL19, XSI+22]. In conclusion, there is a community interest in the project. Unfortunately, it is not suitable for an in-depth evaluation, as it is written in multiple languages including Java, Go and Node.js, but it is a good fit for a conceptual evaluation.

First of all, we noticed that many services have OpenAPI definitions, which is a good thing, as it allows generating some basic communication code, which makes the communication more robust, as errors like wrong field names or missing fields cannot occur. Yet, they did not use the OpenAPI definitions to generate code, rather the sole purpose of them is to document the API. Most of the communication is done via REST and therefore synchronous. The catalog service and the payment service use a server-side circuit breaker, and only the requests made by the order service have a timeout. This really drives home the point that a framework like RobCom is a wise choice, as it makes it easy to add features for robust communication, like timeouts, circuit breakers, retries and rate limits, of which some are enabled out of the box. Otherwise, it can end in a situation like this, where some teams think about some aspects of robustness, and others do not.

There is also one example of asynchronous communication in this project. The shipping service uses a message queue to communicate with the queue-master service. So the processing of the orders is decoupled from the rest of the system.

Next, we want to take a look at the ordering process, as it is the most complex part of the system, with the most communication between the services. When the user wants to order something, the frontend sends a synchronous REST request to the customer service to get the customer's data. Next, it sends two requests in parallel to the customer service, one for the address and one for the credit card. After that, it sends a request to the order service which only contains an URL for the customer, address, credit card and the items from the shopping cart. As the order service gets only the URLs, it needs to send three requests to the customer service (customer, address, credit card) and one to the shopping cart service (items). These requests are sent in parallel and have timeouts. Subsequently, the order service sends a request to the payment service, and then to the shipping service. Only after all these requests are completed successfully, the user gets a response. This does not only take some time, but the chance of one out of the 10 requests failing is quite high, especially as there are no retries or fallbacks in place. For example, the payment service has a circuit breaker, so if it is not available, the order service will always return an error, and every time the user tries again, the customer service gets six requests. This increases the likelihood of the customer service failing, too.

We argue that this is a good example of a project which would benefit from our framework. Not only will it add some robustness to synchronous communication, but it will also make it easier to add asynchronous communication. Of course, it is not straightforward to replace the order process with asynchronous communication, as the frontend tries to give instant feedback to the user, but it is possible. We suggest that when the user

presses the order button, the frontend only needs to send the customer id to the order service. By using RobCom, the order service can subscribe to the customer data and has everything already on hand, and immediately sends the response back to the user. All six requests to the customer service can be removed. As the shopping cart data might change frequently, it is not desirable to subscribe to the change events. So this request can stay as it is, but with the difference that it is done after sending the response to the user. The order service then can send an asynchronous message to the payment service, by using a custom action and listening to payment events. When the payment is done, it can send an asynchronous message to the shipping service, which then can do further processing.

On one hand, this allows for a faster response to the user and asynchronous processing of the order, which is more robust, as it will still work if some services are temporarily not available or overloaded. But on the other hand, it does not give the user any feedback if the order was successful or not. We think that this is not a problem, as many errors that can occur during the order process are not the fault of the user, and therefore the user cannot do anything about it. So it is better when the company gets notified about the error, and they can fix it. And in other cases, where it is the user's fault, the user could get an email or something similar, which informs him about the error and how to fix it. This could be, for example, a link where the user can update his credit card information.

6.4.4 Staffjoy

This project is a shift scheduling software and compared to the other projects, it is a real product. Once they closed a seed round of 1.2 million dollars [Val17], but since then it has been shut down and published as open source [TTa23]. So it gives us a good insight into how a real product uses microservices. It has a topological view, which shows the communication between the services, and it is written in Go. Further, it can be executed with Vagrant or Kubernetes, and it has acceptable documentation.

The analysis of the code showed that the communication of the services is done via gRPC and the communication between the frontend and the services is done via REST. As those two protocols are not compatible, their documentation states that they use a gateway to translate between the two protocols. This is somewhat odd, as this diminishes the benefits of using gRPC, as a JSON payload gets generated and then converted to protobuf. We understand that it is not that easy to send a gRPC request from a browser, but then they could have at least used protobuf for the REST communication to get rid of the overhead of JSON and the need to convert between the two formats.

As they use gRPC, they also use protobufs so their interfaces are generated. The REST interfaces are then generated from the protobufs. This makes communication more robust, as errors like wrong field names or missing fields cannot occur. For some communication, we found that they set timeouts, but not for all. No other constructs for robust communication are in the code. By using RobCom, they could easily add timeouts, retries and circuit breakers to all communication. As RobCom also adds asynchronous

communication, they could also use this for the bot service, so it can work independently of the other services. Although it is only written in one language and fulfills our criteria, we will not use it for an in-depth evaluation, as it is written in Go, and we decided to use C# for our framework as can be seen in Section 6.5.1 and Section 6.5.2.

6.5 In-Depth Evaluation / Case study

In this section, we will analyze the following projects in detail, as they fulfill all our criteria and are written in C#, which is the language we chose for our framework. We will analyze the communication between the services and show how it can be improved by using our framework.

6.5.1 Pitstop

This project is software for managing car repair shops and was published on GitHub [vW23]. It has good documentation, which describes the architecture and shows a topological view of the services. It is noticeable that there are regular updates to the project, and it has been used in some research papers [IKB22, PRT21] before. According to the documentation, all communication between the services is asynchronous, and they use RabbitMQ as a message broker. Only the communication between the frontend and the services is synchronous. Further, all services have Docker files, and there are 5 of them with business logic and a frontend. This project is up-to-date and only written in C# and Dotnet 7, which makes it a good fit for our evaluation, as it fulfills all our criteria.

The analysis of the code showed that the communication between the services is indeed asynchronous. In the documentation, they state that they use retries when accessing external resources, like RabbitMQ. A counter-example can be found when looking at the code where the CustomerManagementAPI publishes an event. Anyway, using an outbox would be the better idea, as it would not only make it possible to decouple the sending of the event from the publishing and therefore would not block the thread, but it would also allow for transactional publishing of events. In this example, the changes first get saved to the database, and then the event gets published, so it is not guaranteed that the event gets published if something between saving the changes and publishing the event fails. When establishing a connection to the database or RabbitMQ, they use retries and timeouts. This is a thing our framework does not do. One aspect where RobCom can improve the code is the automatic publishing of events. Also, the subscribing and storing of events comes out-of-the-box with RobCom, so it does not need to be implemented manually, as it is done in this project.

The communication between the frontend and the services is synchronous and done via REST. To make it more robust, they use the Polly library to add retries. However, as they have nothing in place to enforce idempotency, it can happen that requests get executed multiple times.

Replacing the Communication with RobCom

As this project fulfills all our criteria, we will use it for an in-depth evaluation of our framework. First, we created the minimal service definitions with our DSL for all services. An example can be seen in Figure 6.1. Next, the definitions can be copied to all subscribing services and can get referenced in the RobCom config. In Figure 6.2, the RobCom config for the WorkshopManagementAPI is shown. It contains the dependencies to the CustomerManagementAPI and the VehicleManagementAPI, and the required generation hints. Like before, the communication between these services is event-driven which can also be seen in the config. After this, we could get rid of many classes, as they, or adequate replacements, are generated by RobCom. This includes the models, DTOs, mappers, events and event handlers which just store the received events in the database. We did not remove the controllers for synchronous communication, as we want to keep the synchronous communication for the frontend. This is done due to the fact that we only want to focus on inter-service communication. The usage of RobCom necessitates changes to the repositories, as they need to implement the InternalRepository interfaces generated by the framework. Some services used Entity Framework Core and others Dapper. To keep it simple, we changed all services to use Entity Framework Core. Last, the startup-config needs to be changed, so that RobCom gets configured correctly. For this, RobCom provides a handy extension method to add all necessary services and set the connection to RabbitMQ as well as to the other service. This includes adding a factory for the database context, which we defined in the generation hints.

Results

After replacing the communication with RobCom, we can see that we were able to get rid of many classes necessary for communication, as they are generated by RobCom. This especially includes all the code for publishing and subscribing to events. When someone adds some business logic that changes the model, it is impossible to forget to publish the event, and therefore we consider this a more robust solution. Additionally, the events are now stored in the outbox before they get published. This makes the publishing of events transactional, so it is guaranteed that the event gets published if the changes to the database were successful. As mentioned before, they stated that they use retries when accessing external resources, yet we could not find any evidence of this. But if they had done so, it would block the execution hence the number of retries would be limited. By introducing an outbox, the publishing of events can be done in the background and therefore can afford to wait until RabbitMQ is available again.

On the other side, we got aware of some oversights in our framework. For example, the notification service has no internal model and only subscribes to events. We did not think about this case, but it is easily fixed by adding a dummy model. In the future, we could change the framework so that it is possible to define no internal model. This, however, leads to the next inconvenience. To subscribe to events, the request type of the corresponding model must be set to event-driven or adaptive. This automatically stores all the events in the database, so the data is available when the service needs it. By

```
{
  "serviceName": "WorkshopManagementAPI",
  "models": {
    "workshopPlanning": {
      "pluralName": "workshopPlannings",
      "fields": {
        "workshopPlanningId": { "type": "string" },
        "jobs": {
          "type": "array",
          "items": {
            "type": "object",
            "name": "maintenanceJob",
            "fields": {
              "plannedStartTime": { "type": "timestamp" },
              "plannedEndTime": { "type": "timestamp" },
              "vehicleId": { "type": "id" },
              "customerId": { "type": "id" },
              "description": { "type": "string" },
              "actualStartTime": {
                "type": "timestamp",
                "nullable": true
              },
              "actualEndTime": {
                "type": "timestamp",
                "nullable": true
              },
              "notes": { "type": "string" }
            }
          }
        }
      }
    }
  }
}
```

Figure 6.1: Minimal service definition for the WorkshopManagementAPI

```
{
  "serviceDefinition": "workshop-management.robcom.json",
  "dependencies": [
    {
      "serviceDefinition": "customer-management.robcom.json",
      "subscribedModels": [
        { "model": "customer", "requestType": "event-drive"}
      ]
    },
    {
      "serviceDefinition": "vehicle-management.robcom.json",
      "subscribedModels": [
        {"model": "vehicle", "requestType": "event-drive"}
      ]
    }
  ],
  "generationHints": {
    "dbFactoryContextType": "IDbContextFactory<WorkshopManagementContext>",
    "dbContextType": "WorkshopManagementContext"
  }
}
```

Figure 6.2: RobCom config for WorkshopManagementAPI

subscribing to events, it is also possible to hook into this process and so it is possible to add some business logic, which reacts to the events. In the Pitstop project, this was the wanted behavior, most of the time. Nevertheless, there is an exception. The TimeService, which is there only for demonstration purposes, publishes events to indicate that a day has passed. The services do not need to store these events, only react to them. We were able to recreate this behavior by creating custom actions for every service, which we then called asynchronously. Doing so, unfortunately, changes the communication from events to actions. This means that before all services knew about the time service and subscribed to its events, but now the time service needs to know about all services and call their actions.

For the WorkshopManagementAPI, the Pitstop project used event sourcing. Due to the fact that our framework does not support event sourcing, we had to remove this feature. In this case, this was not a problem, as the event sourcing was not used for business logic. We understand that event sourcing is a powerful tool, and we can understand that someone wants to use it in a microservice architecture, but it increases the complexity of the system, and therefore we decided to not support it.

Another thing we noticed is that our framework only allows subscribing to top-level models. We thought that this would not be a big problem when designing a project

with this limitation in mind. Although, now we are not so sure about this anymore. As the workshop model contains a list of maintenance job models, we decided to define the maintenance job model as a child model of the workshop model. In the Pitstop project, the invoice service and the notification service only need the maintenance job model and not the whole workshop planning model. Of course, we could simply change the definition, so that the maintenance job model is also a top-level model and the workshop planning model only contains a list of ids of the maintenance jobs. Yet we should not throw this finding under the rug, as it could be a problem in bigger projects where new services get added down the road.

6.5.2 E Shop On Containers

This is a sample application for a web shop, which was published by Microsoft on GitHub [DlTTA+23]. As it is intended to be used as a reference for building microservice architectures, it has extensive documentation explaining many aspects of the project including how the communication between the services is done, and depicting it in a topological view. The git history shows that this project is under active development, and gets always updated to the newest version of Dotnet. The project has over 24k stars on GitHub and is used in some recent research papers [RSL+21, NNCS20]. Hence, there is a community interest in the project.

In the documentation, they state that most of the communication between the services is asynchronous. Only the aggregators, which are backends for frontends (BFFs) use gRPC for requesting data from the services. When looking into the code, we found that the communication between the BFFs and the services has no constructs in place to make it more robust. Yet we think this is acceptable, as the BFFs only act as an advanced API gateway that combines data from multiple services. In addition, the frontends have retries and circuit breakers in place.

When analyzing the asynchronous communication, we found that they do not publish changes for every model. Rather, they publish highly specific events for a given action. By doing so, they can reduce the number of events, but it also makes it harder for new services to subscribe to the events. Often this design choice will lead to an adaption of the sender when a new service wants to subscribe to the events. An example of this is the catalog service, which publishes an event when the price of a product changes, but not when the name changes. In conclusion, all the events get published manually, which has the potential to lead to errors, as a developer can forget to publish an event. On the positive side, they use an outbox to store the events before they get published, so the publishing of events is transactional. When the publishing fails, some services retry the publishing, but not all. In particular, the catalog service does not retry it, although it uses an outbox. The order service, on the other hand, retries the publishing, but only when a new event needs to be published.

In contrast to RobCom, these services use different handlers for synchronous and asynchronous communication. All requests get a request id, which is used to check for

idempotency. Unfortunately, not all services check for duplicates, so it can happen that requests get executed multiple times. The order service checks for duplicates, but the basket service does not, which is acceptable, as the basket service does not handle any events where duplicates could be a problem. The catalog service does also not use the request id to check for idempotency, which is problematic as it can lead to removing bought products multiple times from the stock.

To reduce the communication that the order service needs to do on checkout, all the product data get sent to it, so it does not need to request it from the catalog service. When using the MVC Web app, the requests get sent to the BFF, which makes a synchronous request to the catalog service, and then sends the data to the order service. Arguably, this design choice is confusing at best and a security hazard at worst. The explanation for this is twofold: First, it provides less benefit if the BFF makes synchronous requests to the catalog service and not the order service, solely for the purpose that the services do not use any synchronous communication. It only moves the communication to a place where it is not expected. In this case, it is especially confusing, as only the product details get requested in sync on checkout, but not the stock. To check the stock availability, the order service needs to make an asynchronous request to the catalog service. Second, it is a security hazard, as the requester needs to check the product data before sending it to the order service. This is especially important when the product data gets sent from the frontend, as these requests can be manipulated. Unfortunately, this project has fallen into this trap. For the second frontend, the SPA, the data does not get checked by the BFF and gets sent directly to the services behind. By modifying the requests sent by the SPA, with the browser's dev tools, we were able to change the price of the products and buy them for a lower price. This shows that communication in a distributed system is not only about performance and robustness but also about security. Simple design choices to reduce communication can lead to vulnerabilities.

This project fulfills all our criteria, and therefore we could use it for an in-depth evaluation of our framework. Due to the results of the code analysis, we think that this project would benefit from our framework. Nevertheless, it is not that easy to replace the communication with RobCom. This is partially owed to the fact that events are highly specific and to the flawed communication constructs described before. Therefore, we decided to only replace the communication between the catalog service and the basket service, to show how the communication would change, which then can be applied to the other services, too. The catalog service now not only publishes an event when the price of a product changes, but also when any other field changes. This allows other services to act upon the changes. In particular, the basket service can now not only update the product price but also the name or the picture. Further, it is able to remove products from baskets when they get removed from the catalog. Also, other services like the order service can now subscribe to the products, so it has the data on hand when a checkout happens. This eliminates the security risk described before.

CHAPTER 7

Conclusion

In this thesis, created a DSL that helps unify synchronous and asynchronous communication. While doing so, we also focused on making communication more robust and reliable. We created a sample DSL and implemented a proof of concept to show that it is possible to create such a DSL. Through the evaluation, we wanted to validate our concept and find out which types of microservices and communication can benefit from our approach.

7.1 Key findings

We found that most of the analyzed projects use synchronous communication, which is not necessarily the best choice. Especially as concepts to make communication more robust are often not used. When they are used, it is often inconsistent, which could be the result of multiple developers working on a project with no clear guidelines in place. Just a few projects used DSLs like OpenAPI or gRPC to define communication, which is a good practice. As a consequence, our framework can provide many benefits, as it is possible to define communication in a language-agnostic way, which then can be used to generate the code. Surprisingly, we also stumbled across an example where flawed design decisions for communication led to security issues, which could have been avoided by using our framework.

By abstracting the communication and the necessary logic, it is possible to unify synchronous and asynchronous communication and use the same business logic for both. This allows developers to focus on the business logic and not on the communication, while still being able to decide which communication mode should be used in a given situation.

As expected, we encountered some shortcomings in our approach. Due to the fact that we focused on the communication between microservices, we did not consider the communication with the frontend. To keep the DSL and the framework simple, we can

only model document-based data and we do not support event-sourcing. In conclusion, our approach only works when both sides of the communication are using our framework, hence it required more manuel intervention when cummunication with external services is needed.

7.2 Limitations

As this thesis used open-source projects to evaluate the necessity and feasibility of the presented approach, and most of these projects are only used for demonstration purposes, it is not possible to predict how well it would work in a real-world scenario. However, we argue that these demonstration projects are there for a reason, so developers can learn how to create microservice architectures. Therefore, it is likely that in a real-world scenario, similar problems will occur. As we focused on small projects with only a few microservices, it is not clear how well it would work in a large-scale project with hundreds of microservices.

7.3 Future Research

Our concept only considered the communication between microservices, but not the communication with the frontend, hence it would be interesting to see how the communication from the frontend to the backend can be improved. We think that the concept of unified synchronous and asynchronous communication can have great potential and should be further investigated, how it can be enhanced. In our research, we also stumbled upon the concept of debouncing, which is commonly used for UI code to reduce the number of requests sent to the backend. Such a concept could be used for inter-microservice communication too, so not every change is published on its own, but instead, multiple changes are bundled together and sent as one message.

7.4 Final Reflection

While the results presented promising and potential benefits for microservices communication, it is not clear if those are worth the effort due to the pluralism of the microservice landscape. Yet, we think it gives a deeper understanding of the significance of making communication robust, reliable, and unified. This thesis has shed light on the current state of microservices communication as this is a very important topic because microservices are a distributed system and therefore heavily rely on communication.

List of Figures

List of Tables

List of Algorithms

Bibliography

[A+98] Rob Appelbaum et al. Asynchronous communications. https://www4.cs.fau.de/~geier/corba-faq/asynch-comm.html, 1998. Accessed: 6.6.2023.

[ALFT21] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137:106600, 2021.

[AMPJ17] Carlos M. Aderaldo, Nabor C. Mendonça, Claus Pahl, and Pooyan Jamshidi. Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13, May 2017.

[Bas19] Kenny Bastani. Spring Cloud Example Project. https://github.com/kbastani/spring-cloud-microservice-example, 2019. Accessed: 1.8.2023.

[BFWZ19] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Microservices in industry: Insights into technologies, characteristics, and software quality. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 187–195, March 2019.

[BZ16] Justus Bogner and Alfred Zimmermann. Towards integrating microservices with adaptable enterprise architecture. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 1–6, Sep. 2016.

[C+18] Ben Christensen et al. Hystrix. https://github.com/Netflix/Hystrix/, 2018. Accessed: 1.8.2023.

[C+21] Ian Crosby et al. Sock Shop : A Microservice Demo Application. https://github.com/microservices-demo/microservices-demo, 2021. Accessed: 1.8.2023.

[CAvdH+20] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, and L. Zhu. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, Aug 2020.

[DGL+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham, 2017.

[DLM19] Paolo Di Francesco, Patricia Lago, and Ivano Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, 2019.

[DlTTA+23] Cesar De la Torre, Eduard Tomàs, Christian Arenas, et al. .NET Microservices Sample Reference Application. `https://github.com/dotnet-architecture/eShopOnContainers`, 2023. Accessed: 15.8.2023.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000.

[Fou23] The Linux Foundation. OpenAPI. `https://www.openapis.org/`, 2023. Accessed: 6.6.2023.

[Fow11] Martin Fowler. TolerantReader. `https://martinfowler.com/bliki/TolerantReader.html`, May 2011. Accessed: 8.6.2023.

[Fow17] Martin Fowler. What do you mean by "Event-Driven"? `https://martinfowler.com/articles/201701-event-driven.html`, February 2017. Accessed: 8.6.2023.

[GGM23] Saverio Giallorenzo, Claudio Guidi, and Fabrizio Montesi. Jolie. `https://www.jolie-lang.org/`, 2023. Accessed: 6.6.2023.

[Goo23a] Google. gRPC: A high performance, open source universal RPC framework. `https://grpc.io/`, 2023. Accessed: 6.6.2023.

[Goo23b] Google. Protocol Buffers Documentation. `https://protobuf.dev/programming-guides/proto3/`, 2023. Accessed: 6.6.2023.

[Gro23] Object Management Group. Corba. `https://www.corba.org/`, 2023. Accessed: 6.6.2023.

[Hel12] Pat Helland. Idempotence is not a medical condition: An essential property for reliable systems. *Queue*, 10(4):30–46, apr 2012.

[Hen06] Michi Henning. The rise and fall of corba: There's a lot we can learn from corba's mistakes. *Queue*, 4(5):28–34, jun 2006.

[HSYK18] Xian Jun Hong, Hyun Sik Yang, and Young Han Kim. Performance analysis of restful api and rabbitmq for microservice web application. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 257–259, Oct 2018.

[IKB22] Andres Osamu Rodriguez Ishida, Kostas Kontogiannis, and Chris Brealey. Extracting micro service dependencies using log analysis. In *2022 IEEE 29th Annual Software Technology Conference (STC)*, pages 82–92, Oct 2022.

[JPM+18] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, May 2018.

[KS21] Seda Kul and Ahmet Sayar. A survey of publish/subscribe middleware systems for microservice communication. In *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 781–785, Oct 2021.

[LF14] James Lewis and Martin Fowler. Microservices a definition of this new architectural term. `https://martinfowler.com/articles/microservices.html`, March 2014.

[Lib20] Giacomo De Liberali. Asyncmdsl: a domain-specific language for modeling message-based systems. 7 2020.

[Mes12] Audrius Meskauskas. CORBA VS Webservices. `https://stackoverflow.com/a/13450269`, 2012. Accessed: 6.6.2023.

[MJBC22] James Montemagno, Tarun Jain, Clark Brent, and David Coulter. Communication in a microservice architecture. `https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture`, April 2022. Accessed: 6.6.2023.

[New19] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019.

[NNCS20] Espen Tønnessen Nordli, Phu H. Nguyen, Franck Chauvel, and Hui Song. Event-based customization of multi-tenant saas using microservices. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 171–180, Cham, 2020. Springer International Publishing.

[Pos81] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.

[PRT21] Sebastiano Panichella, Mohammad Imranur Rahman, and Davide Taibi. Structural coupling for microservices. *CoRR*, abs/2103.04674, 2021.

[RbAB+21] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. Empirical standards for software engineering research, 2021.

[RL19] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210, June 2019.

[RSL+21] Ali Rezaei Nasab, Mojtaba Shahin, Peng Liang, Mohammad Ehsan Basiri, Seyed Ali Hoseyni Raviz, Hourieh Khalajzadeh, Muhammad Waseem, and Amineh Naseri. Automated identification of security discussions in microservices systems: Industrial surveys and experiments. *Journal of Systems and Software*, 181:111046, 2021.

[RV13] Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 249–262, New York, NY, USA, 2013. Association for Computing Machinery.

[S+23] Chris Sienkiewicz et al. Incremental Generators. `https://github.com/dotnet/roslyn/blob/main/docs/features/incremental-generators.md`, 2023. Accessed 6.8.2023.

[SG01] U. Saif and D.J. Greaves. Communication primitives for ubiquitous systems or rpc considered harmful. In *Proceedings 21st International Conference on Distributed Computing Systems Workshops*, pages 240–245, April 2001.

[SMMR16] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices: Genericity in jolie. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, mar 2016.

[SSKT22] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the 2022 ACM/SPEC on International Conference on*

Performance Engineering, ICPE '22, page 17–27, New York, NY, USA, 2022. Association for Computing Machinery.

[Sto18] Ben Stopford. *Designing event-driven systems*. O'Reilly Media, Incorporated, 2018.

[TDKA+17] Branko Terzić, Vladimir Dimitrieski, Slavica Kordić (Aleksić), Gordana Milosavljevic, and Ivan Luković. Microbuilder: A model-driven tool for the specification of rest microservice architectures. 03 2017.

[Tem20] Muzaffar Temoor. *Architecture for Microservice Based System. A Report*. Dec 2020.

[TS15] Doug Tollefson and Andrew Spyker. Acme Air Sample and Benchmark. `https://github.com/acmeair/acmeair`, 2015. Accessed: 1.8.2023.

[TTa23] Philip I. Thomas, Sam Turner, and andhess. StaffjoyV2. `https://github.com/LandRover/StaffjoyV2`, 2023. Accessed: 5.8.2023.

[Val17] Angelica Valentine. Staffjoy Announces V2 and $1.2M Seed Round. `https://blog.staffjoy.com/staffjoy-announces-v2-and-1-2m-seed-round-8abb025a150d`, 2017. Accessed: 5.8.2023.

[vW23] Edwin van Wijk. Pitstop - Garage Management System. `https://github.com/EdwinVW/pitstop`, 2023. Accessed: 3.8.2023.

[WEPL18] Sanjiva Weerawarana, Chathura Ekanayake, Srinath Perera, and Frank Leymann. Bringing middleware to everyday programmers with ballerina. In Mathias Weske, Marco Montali, Ingo Weber, and Jan vom Brocke, editors, *Business Process Management*, pages 12–27, Cham, 2018. Springer International Publishing.

[WKR21] Yingying Wang, Harshavardhan Kadiyala, and Julia Rubin. Promises and challenges of microservices: an exploratory study. *Empirical Software Engineering*, 26(4):63, 2021.

[WSO23] WSO2. Ballerina. `https://ballerina.io/`, 2023. Accessed: 6.6.2023.

[XSI+22] Minxian Xu, Chenghao Song, Shashikant Ilager, Sukhpal Singh Gill, Juanjuan Zhao, Kejiang Ye, and Chengzhong Xu. Coscal: Multifaceted scaling of microservices with reinforcement learning. *IEEE Transactions on Network and Service Management*, 19(4):3995–4009, Dec 2022.

[Zim18] Olaf Zimmermann. Microservice DSL (MDSL). `https://microservice-api-patterns.github.io/MDSL-Specification/`, 2018. Accessed: 6.6.2023.