

Reliable Messaging using the CloudEvents Router

Bachelor Thesis

Department of Computer Science
OST – University of Applied Sciences of Eastern Switzerland
Campus Rapperswil-Jona

Fall Term 2020/2021

Authors	Linus Basig, Fabrizio Lazzaretti
Advisor	Prof. Dr. Olaf Zimmermann
Project Partner	CARU AG
External Co-Examiner	Dr. Gerald Reif
Internal Co-Examiner	Ivan Bütler
Printing Date	January 15, 2021

Abstract

CARU is an AgeTech startup offering a voice-controlled emergency-call device designed to help the elderly live an independent life for longer. The device will react to a cry for help and automatically activate an emergency call to up to 5 family members. CARU's software architecture is heavily event-driven, envisioning a unified event plane letting events flow between all connected systems. To support this vision, CARU's events are structured according to the CloudEvents specification of the Cloud Native Computing Foundation (CNCF). In our previous student research project, we designed, implemented, and open-sourced the CloudEvents Router¹ – a first step towards this vision.

To entrust mission-critical events to the CloudEvents Router, it has to guarantee that these messages arrive. The goal of this bachelor thesis is to extend the router to provide such a delivery guarantee. This guarantee should also apply when an event is routed across different protocols. We focused on the protocols that the CloudEvents specification defines a protocol binding for: AMQP, Kafka, MQTT, NATS, and WebHooks.

As a starting point, we conducted extensive literature research to identify and compare existing reliable messaging definitions and patterns.

Our research showed that the most feasible approach to provide reliable routing is to implement an end-to-end delivery guarantee of "At Least Once". This guarantee applies from the event source, via the router, to the event destination. We then analyzed which delivery guarantees are provided by the mentioned messaging protocols and how interoperable the supporting concepts are. Having adapted the architecture design of the existing CloudEvents Router to fulfill the new requirements, we implemented a proof-of-concept that routes messages between AMQP and MQTT with an "At Least Once" delivery guarantee.

We released our open-source CloudEvents Router implementation, now supporting the desired "At Least Once" delivery guarantee in addition to the already existing "Best Effort" policy. The new version also supports the AMQP protocol complementing the already existing support for MQTT. With this new guarantee, CARU has started evaluating the CloudEvents Router for productive use on their device.

¹<https://github.com/ce-rust/cerk>

Management Summary

Context and Problem

CARU AG is an AgeTech startup with the mission to help the elderly to live independently for longer. Its internet-connected device allows calling for help by voice command and offers data-driven functionalities powered by its many sensors. The different software components at CARU communicate mostly over events: Whenever something notable happens, an event is published. Other components react to these events and create new events themselves. Most events are structured according to the CloudEvents specification of the Cloud Native Computing Foundation (CNCF). By using the same event structure everywhere, CARU wants to create a unified event plane and simplify the communication between the different software systems. These include the ones on the device, in the cloud, and potentially even from third parties. Connecting the many systems involved is a complicated undertaking. The simplest proposal of always broadcasting each event to all systems quickly shows to be inefficient and causing security risks.

To address this shortcoming, the CloudEvents Router was designed and implemented in the preceding student research project. The CloudEvents Router is responsible for deciding which event should be shared with which system. This decision is based on the meta-data defined by the CloudEvents specification.

To entrust mission-critical events to the CloudEvents Router, it has to guarantee that these messages arrive. The goal of this bachelor thesis is to extend the CloudEvents Router to provide a delivery guarantee.

Methods and Results

As a starting point, we conducted extensive literature research to identify and compare reliable messaging definitions and patterns. We then analyzed how different messaging protocols, which are of interest to our industry partner CARU, provide delivery guarantees, and how interoperable their concepts are.

After adapting the existing CloudEvents Router's architecture design to fulfill the new requirements, we implemented a proof-of-concept that routes messages between AMQP and MQTT with an "At Least Once" delivery guarantee. We validated our implementation's correctness in two end-to-end test-scenarios, and both test cases demonstrated that the shortcomings of the previous implementation had been overcome.

Future Work

The CloudEvents Router lays a solid foundation for CARU to bring their vision of a unified event plane into reality. To advance the vision, we propose three topics to consider for future work:

- Extend the CloudEvents Router to integrate with monitoring solutions from the CNCF
- Adoption of the now stable async API in Rust to lower the resource requirements even more
- Implementation of the new CloudSubscriptions Discovery API and CloudEvents Subscription API specification
- Streamlining of the configuration management

These are interesting research topics that would further improve the CloudEvents Router.

Acknowledgments

First of all, we would like to thank our advisor, Prof. Dr. Olaf Zimmermann, for his exceeding support and guidance.

We are also thankful for Reto Aebersold and Thomas Helbling from CARU for sponsoring this thesis and their invaluable input.

Lastly, we are incredibly grateful for our patient friends, colleagues, and partners. We want to especially thank Urs Basig, Livio Bieri, Marco Crisafulli, Hannah Li Hägi, Maja Lazzaretti, Gloria Shi, and Paul Tingle for their feedback and proofreading.

Contents

1. Introduction	11
1.1. CARU's Vision	11
1.2. Creation of the CloudEvents Router	13
1.3. Creation of the Official CloudEvents Rust SDK	15
1.4. The Problem	16
1.5. The Solution Strategy	17
2. Reliable Messaging	19
2.1. What is Reliable Messaging?	19
2.2. Reliable Messaging – Patterns and Strategies for Message Routing	19
2.3. Protocol Interoperability in a Stateless Message Router	20
2.4. Focus on Stateless Routing	20
2.5. Decision on Delivery Guarantee Selection	21
3. Architecture Design	23
3.1. Design Changes for Reliable Messaging	23
3.1.1. Routing with "Best Effort"	23
3.1.2. Routing with an "At Least Once" Delivery Guarantee	24
3.2. Error Scenarios	24
4. Implementation	27
4.1. Overview	27
4.2. Noteworthy Modifications	28
4.2.1. Adaption of the MQTT Port	28
4.2.2. Improve the Getting Started Experience	31
5. Validation	35
5.1. Single Message Routing Integration Tests	35
5.1.1. Test Method	35
5.1.2. Test Result	37
5.2. Load Test	37
5.2.1. Test Method	37
5.2.2. Test Result	39
5.3. Performance Test	39
5.4. Developer Experience Showcase	39
5.4.1. Test Method	40
5.4.2. Test Result	41
5.5. Summary	42
6. Conclusion and Outlook	43
6.1. Outcome	43
6.2. Findings	44
6.3. Outlook	44
List of Figures	45
List of Tables	47

List of Listings	49
Glossary	51
Acronyms	59
A. Problem Statement	67
B. Reliable Messaging – Patterns and Strategies for Message Routing	71
C. Protocol Interoperability in a Stateless Message Router	89
D. Architecture Documentation (arc42)	101
E. Load Test Statistics	187
F. Analyzing the Load Test	191
G. Developer Experience Showcase: Implementation Steps	199
H. Used Tools	205
I. Code Appendixes Listings	209

1. Introduction

The content of this bachelor thesis builds on the work done during our student research project "CloudEvents Router".[1] This chapter introduces our industry partner, summarizes the work done previously, and introduces the core problem of this thesis.

The industry partner for the student project thesis, and now this bachelor thesis, is CARU. The company is an AgeTech startup with the mission to help the elderly live independently for longer. Its Internet of Things (IoT) device, the CARU Device (formerly known as CARU SmartSensor), allows calling for help by voice command or touch. Figure 1.1 shows the CARU Device placed in the home of a senior citizen. In addition to this alarm-call-feature, the device allows its users to communicate with their relatives via voice messages. The relatives can listen and reply to these voice messages with the "CARU Family Chat" smartphone app. The device is also equipped with a multitude of sensors that allow a variety of data-driven functionalities. To access these services, CARU offers a web-based application for relatives and professional caretakers.

1.1. CARU's Vision

The software architecture used by CARU is heavily event-driven. This means that the different components of their software landscape publish events when something notable happens. In turn, other components act on these events and publish new events themselves.

CARU wants to create a unified event plane where events can flow between all connected systems. These include the systems on the device, in the cloud, and potentially even from third parties. As the fundament to achieve a unified event plane, CARU embraces CloudEvents as their standardized event format.

CloudEvents is a specification proposed by the Serverless Working Group of the Cloud Native Computing Foundation (CNCF). The goal of the specification is to significantly simplify event declaration and delivery across services, platforms, and vendors. To achieve this goal, the specification defines the event structure, different serialization formats, and multiple protocol bindings. The specification is attracting lots of attention and contributions from major cloud providers and Software as a Service (SaaS) companies. Listing 1 shows an example of a CloudEvent in the JavaScript Object Notation (JSON) serialization format.[3, 4]

```
1 {  
2   "type": "caru.sensor.voice",  
3   "specversion": "1.0",  
4   "source": "crn:eu:::device:wwugxd5t",  
5   "id": "e30dc55b-b872-40dc-b53e-82cddfea454c",  
6   "contenttype": "text/plain",  
7   "data": "hilfe"  
8 }
```

Listing 1: A CloudEvent in the JSON Serialization Format

Always broadcasting each event to all systems would be inefficient and cause security risks. The goal of the preceding student research project was to find a solution that routes events based on the content



Figure 1.1.: The CARU Device placed in the home of a senior citizen[2]

of the fields defined in the CloudEvents specification. An additional requirement was that the solution must be lightweight enough to run on the CARU Device, which has strict resource limitations. As a start, we conducted a market analysis of existing off-the-shelf solutions. We selected 10 messaging products¹ and conducted a high-level analysis of them. After the high-level analysis, we looked closer into two of these 10 products (Apache Camel and Node-RED). Both looked very promising and offered the possibility to extend them to support CloudEvents. Unfortunately, their resource requirements were incompatible with the available resources on the CARU Device. We decided to design and prototype our own specialized CloudEvents Router. Together with CARU, we selected Rust as the programming language.

Figure 1.2 shows CARU’s envisioned system context with the unified event plane implemented. Whenever events are exchanged, they are formatted according to the CloudEvents specification. The CloudEvents Routers are placed in each system and connect over an appropriate messaging protocol to their neighboring systems. The services of a system are then also connected to their system’s CloudEvents Router over an appropriate messaging protocol. At the moment, the most used messaging protocol is the Message Queuing Telemetry Transport (MQTT) protocol. However, with new services and partners, there could be a heterogeneous mix of messaging protocols in the future.

¹ The 10 messaging products that were analyzed are:

- CloudEvent Router and Gateway (github.com/rh-event-flow/cloudevents-gateway)
- Knative Eventing v0.9 (github.com/knative/eventing/tree/v0.9.0)
- Pacifica Dispatcher v0.2.3 (github.com/pacifica/pacifica-dispatcher/tree/v0.2.3)
- Serverless Event Gateway v0.9.1 (github.com/serverless/event-gateway/tree/0.9.1)
- Amazon Simple Notification Service (aws.amazon.com/sns)
- Apache Camel v2.24.2 (camel.apache.org/components/2.x)
- Crossbar.io v19.10.1 (github.com/crossbario/crossbar/tree/v19.10.1)
- D-Bus v1.12 (gitlab.freedesktop.org/dbus/dbus/tree/dbus-1.12)
- Node-RED v1.0.1 (github.com/node-red/node-red/tree/1.0.1)
- RabbitMQ v3.8 (github.com/rabbitmq/rabbitmq-server/tree/v3.8.0)

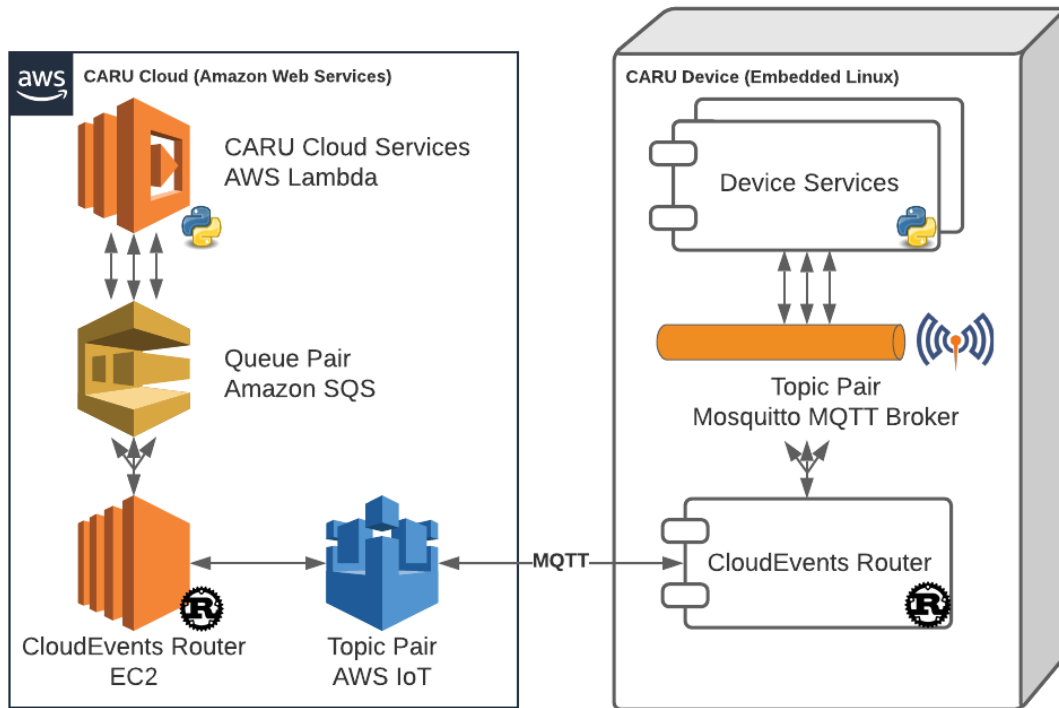


Figure 1.2.: CARU's Envisioned System Context

(This image is explained in detail in Appendix D "Architecture Documentation (arc42)")

1.2. Creation of the CloudEvents Router

For our implementation of the CloudEvents Router (code name CERK), we chose the Microkernel architecture. This means the router is composed of a Microkernel, that defines the interfaces to the plugins, and plugins that implement the bulk of the functionality. Besides the plugin interfaces, the kernel only consists of a small amount of glue code that facilitates the data-flow between the plugins. Because the plugins are defined through their interfaces, there can be different implementations of a specific plugin type. The user can then choose the implementation for each plugin type based on the specifics of his or her use case. The Microkernel pattern is commonly used as the architectural foundation of modern operating systems. A prominent example is Linux, which consists of a relatively small kernel and many plugins that implement the bulk of the functionality, e.g., hardware drivers, file systems, and different graphical desktop interfaces.[5]

After the student research project, the kernel defined four plugin types. Figure 1.3 shows these plugin types. When the router is started, the kernel initializes the **ConfigLoader**, the **Router**, and all the defined **Ports** with the help of the **Scheduler**. After the **ConfigLoader** is initialized, it loads the configuration of the **Ports** and the **Router** and sends the corresponding configuration to them via the kernel. As soon as the **Port** plugins receive their configuration, they use the received configuration to connect to their messaging partners over some messaging protocol and start to receive CloudEvents. When a **Port** receives a CloudEvent, it forwards the CloudEvent to the kernel, and the kernel then sends it to the **Router**. The **Router** applies the routing rules encoded in its configuration to the CloudEvent and returns the routing decision back to the kernel. A CloudEvent can be routed to any **Port** on the router or to none of them. Based on this routing decision, the kernel forwards the CloudEvent to the destination **Ports**. Each destination **Port** hands over the CloudEvent to its messaging partner, and the routing process for a single CloudEvent is complete. All four kernel module types are still in today's router, but some interactions between the modules have slightly changed during this thesis. The most important changes in the architecture of the router are discussed in

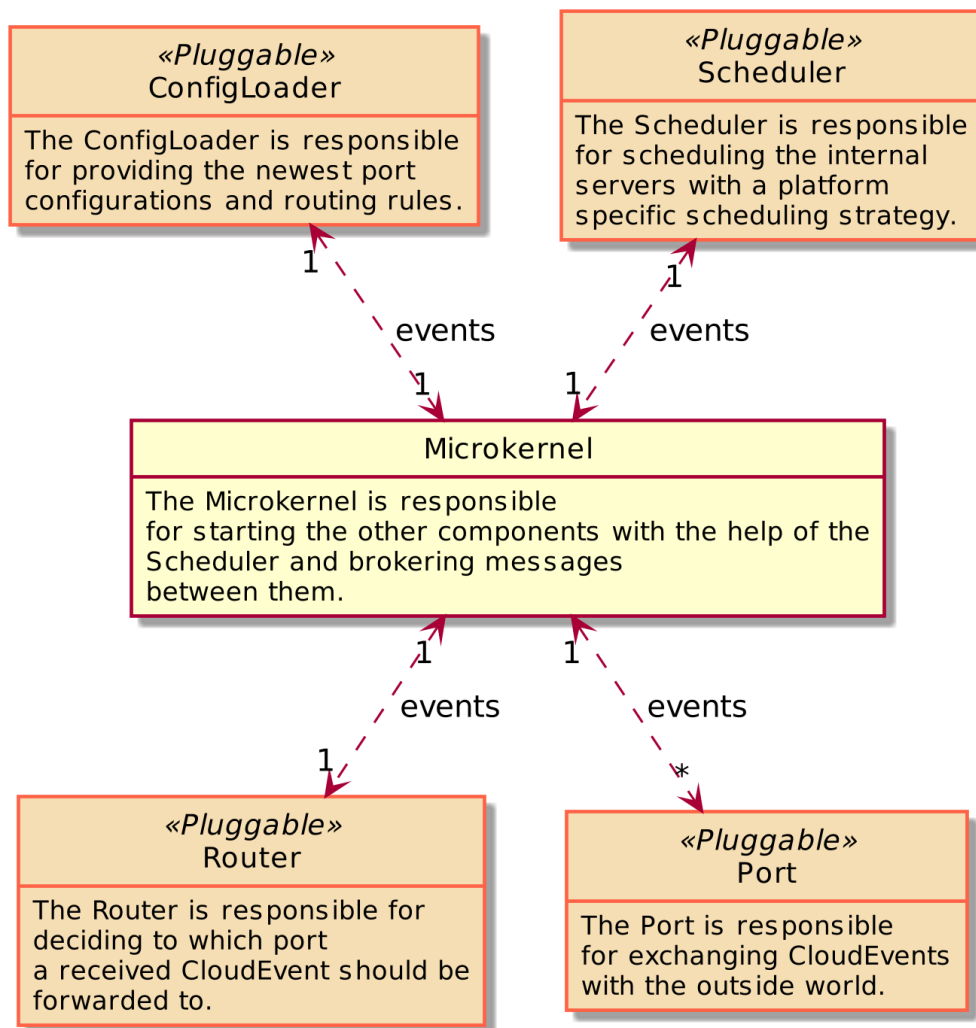


Figure 1.3.: Plugins of the CloudEvents Router after the Student Research Project
(This image is explained in detail in the arc42 documentation)

Chapter 3 "Architecture Design".

The reason for choosing the Microkernel approach was that we wanted to be able to adapt the router to different deployment environments easily. For example, in the embedded Linux environment of the CARU Device, a **ConfigLoader** implementation could be used that loads the configuration from a file, while in a cloud environment, a **ConfigLoader** implementation could be used that loads the configuration from a database. The modular system also allows adding support for new messaging protocols by writing plugins that implement the **Port** interface. It also helps to optimize the size of the executable by only including the plugins that are required for a specific use case. In Figure 1.2, for example, the CloudEvents Router on the device is only required to communicate with the MQTT protocol, while the CloudEvents Router in the cloud has to support both MQTT and Simple Queue Service (SQS).

At the end of the student research project, we had a proof-of-concept implementation of the CloudEvents Router that was able to route events over UNIX sockets and MQTT. Figure 1.4 shows the deployment diagram of the setup of our final presentation. The setup consisted of a CARU Device, an MQTT broker deployed in the cloud, and a laptop running a CloudEvents Router. In the demo we held for our student research project, the audience could trigger the Voice Recognition Service on the CARU Device by saying one of two keywords. When the service was triggered, it sent an event in the CloudEvents format containing the keyword to the router on the CARU Device. The router then

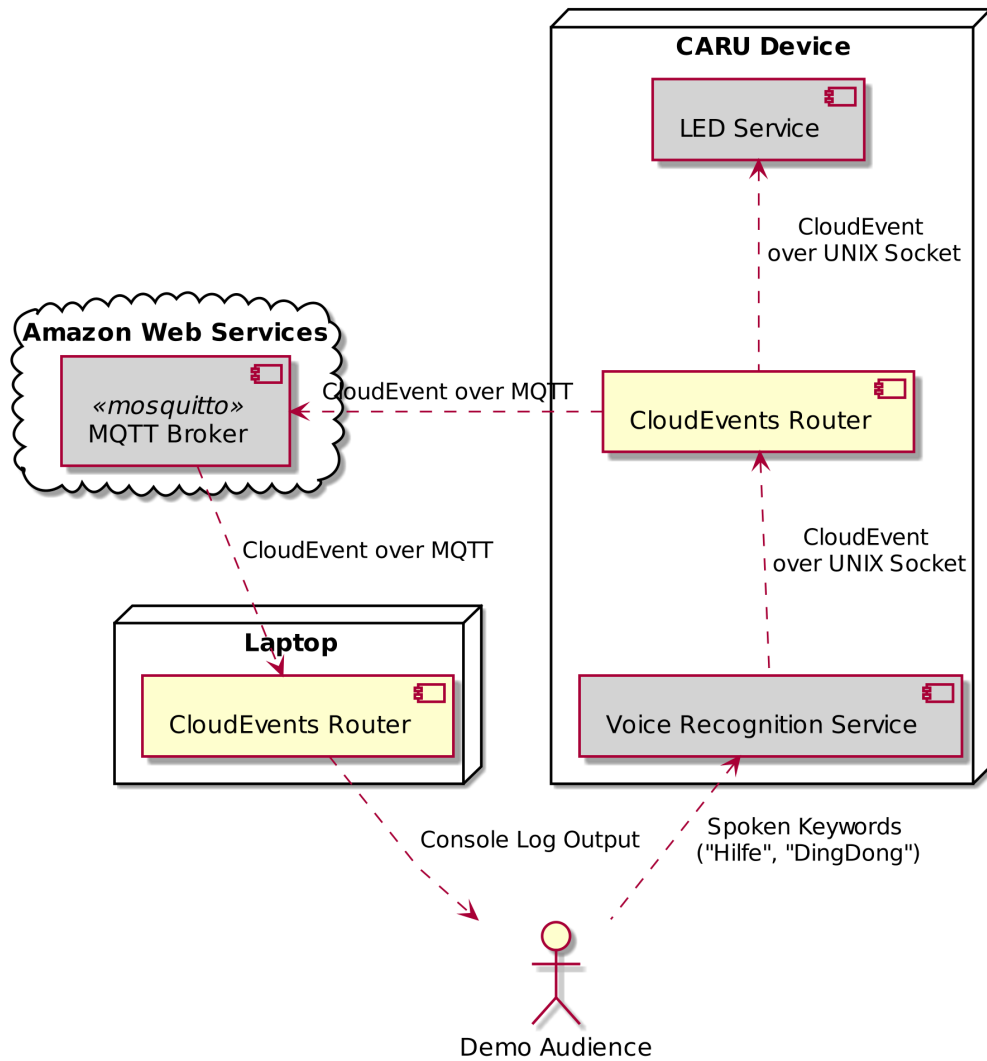


Figure 1.4.: Demo Deployment of the Student Research Project

decided, based on the content of the event, to either send the event only to the LED Service or to the LED Service and the cloud. When the LED Service received the event, the LEDs of the device lit up in the color assigned to the keyword. If the event was sent to the cloud, it was also received by the instance of the CloudEvents Router on the laptop. The router on the laptop forwarded the event to a special `Port` that output the content of the event to the console.

1.3. Creation of the Official CloudEvents Rust SDK

During the student research project, we used an existing library to work with CloudEvents. Unfortunately, this library only supported version 0.2 of the CloudEvents specification, and we needed to work with events in version 1.0. Consequently, we extended the library to support both versions of the CloudEvents specification. We wanted to contribute our code changes to the used library and opened a pull request to ask the owner of the library to integrate them. However, the owner of the library did not respond to our pull request² before the student research project was submitted. On account of this, we finished the project with our own copy of the library.

After our changes were integrated into the library, we got in touch with Doug Davis through Prof. Dr. Olaf Zimmermann. Doug Davis is an active member of the CNCF Serverless Working Group,

²<https://github.com/kichristensen/rust-cloudevents/pull/1>

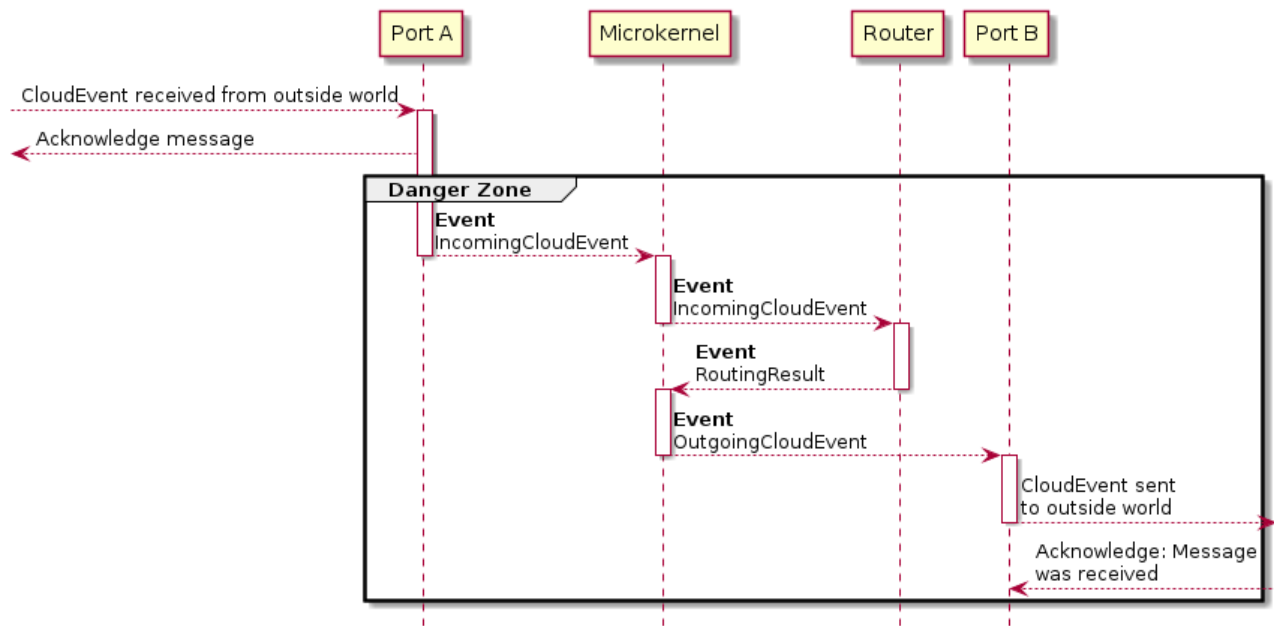


Figure 1.5.: Danger Zone of the Student Research Project CloudEvents Router

which owns the CloudEvents specification.³ Doug Davis connected us with Francesco Guardiani from Red Hat. Together with him, we designed the official Rust CloudEvents Software Development Kit (SDK).⁴

Even though we were involved in the design phase, the implementation itself was mainly done by Guardiani, who is also the active maintainer. Our role in the project is more passive and only includes the review of some pull requests.

1.4. The Problem

One shortcoming that we identified during the student research project was that the CloudEvents Router did not preserve the delivery guarantees of the protocols used to transfer the event to the router. In the case of MQTT, the protocol defines three levels of delivery guarantees called Quality of Service (QoS). MQTT supports QoS 0 ("At Most Once"), QoS 1 ("At Least Once"), and QoS 2 ("Exactly Once").^[6] At the end of the student research project, the **Ports** of the router were able to accept events with any of these delivery guarantees. However, there was a period of time after the event was accepted from a source until the event was delivered to the destination, in which the event was vulnerable to data loss.

Figure 1.5 visualizes this time period. The event would be lost if the CloudEvents Router or its host would crash while the routing process is in the "Danger Zone". Even worse, the sender of the event would receive an acknowledgment that would let it believe that the event delivery was successful.

For CARU to be able to entrust critical events, like alarm call events, to the CloudEvents Router, the router must guarantee the delivery of these events. This thesis aims to get a good understanding of how reliable messaging is defined and implemented in the industry and how these definitions can be matched with the context of CARU. Based on this knowledge, the design of the router should be adapted to guarantee the delivery for critical events while staying within the resource constraints of the CARU Device. Finally, a proof of concept that demonstrates the design in action shall be

³Doug Davis is the main contributor to the CloudEvents specification until now. <https://github.com/cloudevents/spec/graphs/contributors?from=2017-12-03&to=2020-12-26&type=c>

⁴<https://github.com/cloudevents/sdk-rust/>

implemented. A detailed list of the agreed-upon deliverables can be found in Appendix A "Problem Statement".

1.5. The Solution Strategy

To solve this problem, we started with extensive literature research in Chapter 2 "Reliable Messaging". We wanted to answer how reliable messaging is defined, what patterns exist to achieve it, and how different messaging protocols implement these patterns. The goal was to apply this knowledge to the context of CARU and the CloudEvents Router to guide us to a solution.

Equipped with that knowledge, we needed to adapt the architecture design of the CloudEvents Router. In Chapter 3 "Architecture Design", we documented the most significant architecture changes that are required to provide a delivery guarantee for the messages in the custody of the router.

After the design was adapted to the new requirements, the code has to follow. Chapter 4 "Implementation" focuses on this step and reports on the major obstacles that were discovered and overcome while translating the design changes to code changes. Additionally, the chapter summarizes other code changes that were made to make it easier for new users and contributors to work with the CloudEvents Router.

To validate our implementation, we set up two integration test scenarios that challenge the ability of the router to deliver all messages reliably. The setup and results of both test cases are discussed in detail in Chapter 5 "Validation".

Finally, in Chapter 6 "Conclusion and Outlook", we look at how well our solution matches the requirements, what the main findings on the journey to the solutions were, and what our recommendations for the future of the CloudEvents Router are.

2. Reliable Messaging

In this chapter, we present our research on reliable messaging, and discuss why we decided to implement "At Least Once" in our CloudEvents Router. Our research focused on two questions:

1. "Which patterns and strategies exist for reliable messaging, and how are they applicable to our problem?"
2. "Are delivery guarantee semantics implemented similar enough by common messaging protocols to create a stateless, multi-protocol Message Router that preserves these semantics across protocols?"

We did extensive literature research for each of these questions and summarized our findings in two mini-papers. These two papers built the basis for adapting the CloudEvents Router architecture design to support reliable messaging.

First, we introduce a common definition of what reliable messaging means. Then, we give a short summary of our two papers, and at the end of the chapter, we discuss the findings and what these mean for the design of our CloudEvents Router.

The full papers can be found in Appendix B "Reliable Messaging – Patterns and Strategies for Message Routing" and Appendix C "Protocol Interoperability in a Stateless Message Router".

2.1. What is Reliable Messaging?

Reliability is a non-functional requirement of a software system that describes to which degree an action is completed as intended. For messaging, that means that the message is delivered and processed as intended.[7–10]

A more detailed analysis of the term reliability is given in the paper "Reliable Messaging – Patterns and Strategies for Message Routing".[10]

2.2. Reliable Messaging – Patterns and Strategies for Message Routing

In "Reliable Messaging – Patterns and Strategies for Message Routing", we presented an overview of the patterns and strategies that exist for reliable messaging and how they apply to our problem of routing messages reliably in a protocol-independent way.

Our two primary sources for existing messaging patterns were:

- "Enterprise Integration Patterns" by Hohpe and Woolf[11]
- "Cloud Computing Patterns" by Fehling, Leymann, Retter, Schupeck, and Arbitter[12]

We proposed a simple approach to define one delivery guarantee per channel without any additions. The protocol-independent delivery guarantees are named "At Least Once", "At Most Once", and "Exactly Once".

Using the introduced delivery guarantees, we discussed how different architecture description languages (ADLs) and Application Programming Interface (API) description languages can integrate these delivery guarantees to add a clear and simple way to describe delivery guarantees. We examined

the necessary changes for AsyncAPI, asyncMDSL, and the CloudSubscriptions Discovery API.

The author of asyncMDSL has been informed about our proposal and has already implemented the new option into the Microservice Domain-Specific Language (MDSL) syntax.¹ The biggest flaw that we have found in the AsyncAPI specification has also been fixed by our pull request.²

Finally, we showed how each described delivery guarantee could be implemented in a router that routes messages reliably. We demonstrated that the overhead from routing with an "Exactly Once" guarantee can be removed from the router by transferring this task to a message filter that removes duplicated messages before they are handed over to a consumer.

2.3. Protocol Interoperability in a Stateless Message Router

For "Protocol Interoperability in a Stateless Message Router", we analyzed the specifications of the five messaging protocols that have protocol bindings defined in the CloudEvents specification (Advanced Message Queuing Protocol (AMQP), WebHooks over HyperText Transfer Protocol (HTTP), Kafka, MQTT and NATS). The goal of the analysis was to determine if their delivery guarantee concepts are interoperable.

The main conclusion was that the way how most examined protocols implement the "At Most Once" and "At Least Once" delivery guarantees is interoperable and allows the implementation of a cross-protocol event router. The implementation of a cross-protocol router that provides an "Exactly Once" delivery guarantee is not generally possible in a stateless router.

Not all analyzed protocols implement an "Exactly Once" delivery semantics. The ones which include it, employ such different approaches that it is only possible to use them in a stateful Message Router.

2.4. Focus on Stateless Routing

In both papers, we focused on stateless solutions. The reason for this is because in the age of the "infinitely" scalable cloud and ephemeral containers, the cost that comes with statefulness makes it less attractive. With the rise of the cloud, systems got more distributed, and a stateful service is expected to store its data not only on its local disk but to distribute copies of the data to multiple machines. This distribution of the state increases the cost of a stateful service significantly.

Because we focus on a Message Router that could be easily deployed in a cloud environment, we want to avoid the cost of statefulness. Unfortunately, preserving an "Exactly Once" semantics is not possible with the additional constraint of statelessness.^[10]

In a stateful routing scenario, the reception and the forwarding of a message are decoupled by persisting the message to reliable but expensive storage. In a stateless routing scenario, the acknowledgment to the sender is delayed until the message is forwarded successfully to all receivers. We are willing to trade an increased coupling between the sender and the receivers for statelessness.

¹The new MDSL specification version 5 includes the option "delivery guarantee" on the channel as proposed in the paper "Reliable Messaging – Patterns and Strategies for Message Routing".<https://github.com/Microservice-API-Patterns/MDSL-Specification/blob/79ef5011591a05859a0c63c425e835e026d0a2ec/dsl-core/io.mdsl/src/io/mdsl/APIDescription.xtext#L107-L118>

²The description for the MQTT binding has been changed with the pull request <https://github.com/asynccapi/bindings/pull/42>

2.5. Decision on Delivery Guarantee Selection

The decisions on how to implement delivery guarantees were mainly done during our research phase and were based on the insights of "Reliable Messaging – Patterns and Strategies for Message Routing" and "Protocol Interoperability in a Stateless Message Router". In "Reliable Messaging – Patterns and Strategies for Message Routing", we described how a message router that supports each of the discussed delivery guarantees could look like. We distinguished between four routing scenarios:[10]

- Routing with "At Most Once" semantics
- Routing with "At Least Once" semantics
- Routing with "Exactly Once" semantics
- Routing with "At Least Once" semantics and a message filter to achieve "Exactly Once" semantics

As the goal of the industry partner is to ensure that every message reaches its destination, the first solution is not appropriate. So the decision for the routing was between "At Least Once" and "Exactly Once".

As described in Section 2.4 "Focus on Stateless Routing", we aim to keep our router stateless. This comes with the cost of not being able to route messages with an "Exactly Once" delivery guarantee semantics. However, statelessness helps us to have a router with a small footprint, no need for storage management, and fast communication. Furthermore, the shortcomings of "At Least Once" can be compensated by using a message filter. Ultimately, this led us to the decision to only implement "At Least Once".

3. Architecture Design

In this chapter, we are looking into the design changes made to the CloudEvents Router to make reliable messaging possible. We decided to go with the "At Least Once" delivery guarantee as discussed in the previous chapter. In this chapter, we will focus on the most significant design changes that were necessary to provide this delivery guarantee. A more in-depth view of all design decisions and thoughts can be found in the arc42 documentation in Appendix D "Architecture Documentation (arc42)". The implementation of the design changes will be discussed in the next chapter.

3.1. Design Changes for Reliable Messaging

The CloudEvents Router that was implemented during our student research project was not aware of any delivery guarantees.[1] We decided to name the old behavior of the router "Best Effort". In this mode, the router routes events from A to B, but an event can get lost in the process if an error happens. The mode could still be useful for applications that benefit from a simpler, faster, and more lightweight runtime behavior. The other benefit of implementing the new delivery guarantee on top of the current router, and not replacing it, is that the already existing plugins still work with the updated kernel. The plugins only need to be changed if they need to support the new delivery guarantee. An additional benefit is that a good before-after comparison can be made to show how the behavior of the router changes with the new mode added. This comparison will be introduced and discussed in Chapter 5 "Validation".

In this section, we show the differences between the old "Best Effort" and the new "At Least Once" routing behavior with an example. In the example, an event gets routed from an input port called "Port A" to two output ports: "Port B" and "Port C".

3.1.1. Routing with "Best Effort"

First, we start with the already existing "Best Effort" mode as a baseline. This mode is shown in Figure 3.1.

Port A receives a CloudEvent from the outside world. This port deserializes and forwards the event to the Microkernel, which forwards it again to the router plugin. The router plugin decides which ports the event should be routed to. In this example, the router plugin is configured to route the events to Port B and Port C. The routing result is sent to the Microkernel, which forwards the message to the designated ports. These ports serialize and forward the event to the outside world.

The scenario would look exactly the same for "At Most Once". However, in "At Most Once" there is an additional guarantee that no message duplication happens. This is not guaranteed in the "Best Effort" mode.[10]

The routing scenario "Best Effort" had to be slightly modified in the context of this thesis: Before the implementation of "At Least Once" mode, the router was returning each routing result as an `OutgoingCloudEvent` message. Now it returns a `RoutingResult`. This makes the implementation of the router simpler, as it is exactly the same for the "Best Effort" and "At Least Once" mode. The necessity of the `RoutingResult` message will be explained in Section 3.1 "Design Changes for Reliable Messaging".

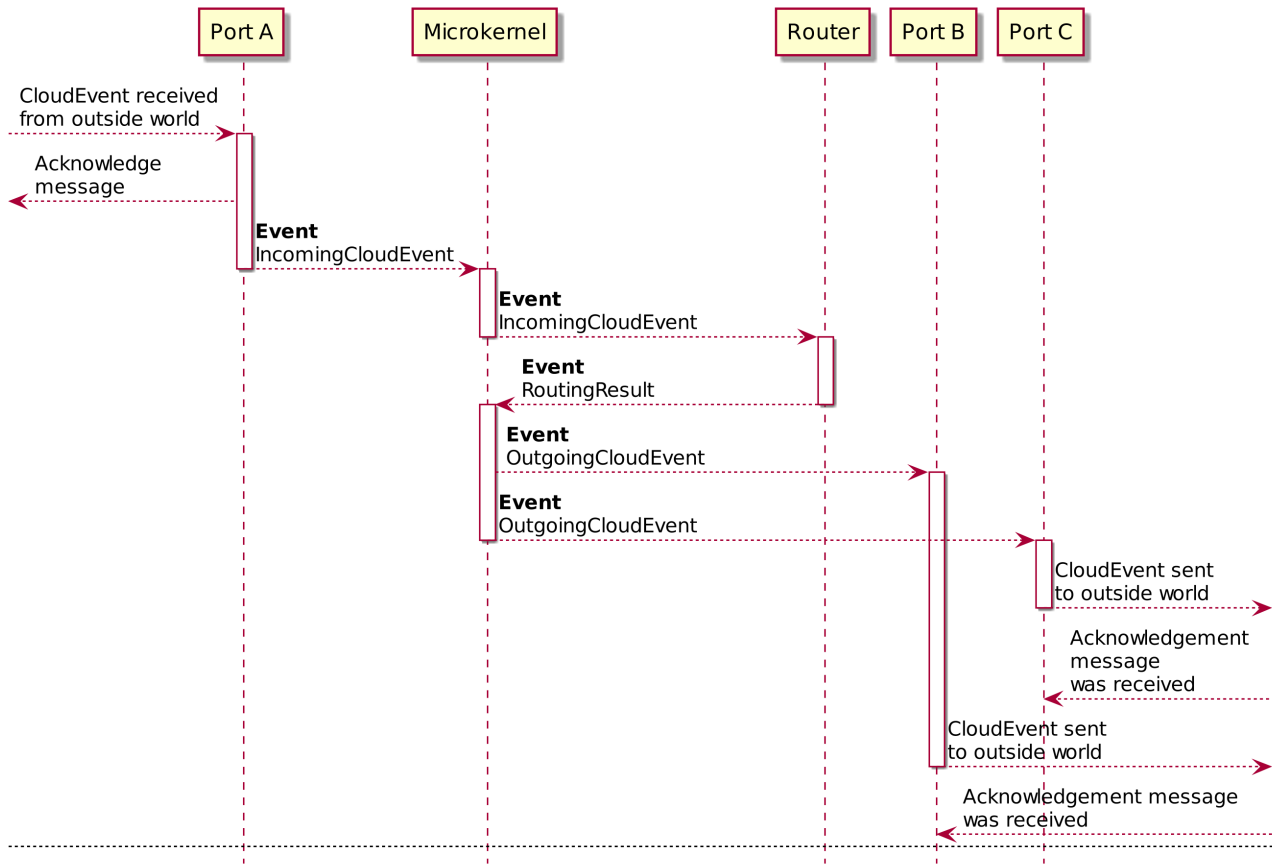


Figure 3.1.: Routing with "Best Effort"

3.1.2. Routing with an "At Least Once" Delivery Guarantee

Figure 3.2 shows the routing scenario with an "At Least Once" delivery guarantee. Port A receives a CloudEvent from the outside world, but it will not yet acknowledge it.

In "Best Effort" mode the acknowledgment would be sent immediately or there is no acknowledgment at all. The decision to send an acknowledgment depends on the underlying protocol and the implementation of each port. We have not specified a correct behavior, as it is only the "Best Effort" flow. Port A deserializes and forwards the event to the Microkernel, which forwards it again to the router plugin. The router plugin decides which ports the event should be routed to. In this example, it routes the event to Port B and C. The routing result is sent to the Microkernel.

The Microkernel knows that the incoming event needs an acknowledgment, so it stores the metadata about the event and the routing decision (the state is indicated with a bar over the dotted line in the figure). After the Microkernel stored the metadata, it forwards the event to the designated ports.

These ports serialize and forward the event to the outside world. Then they inform the Microkernel about the delivery result (`OutgoingCloudEventProcessed`).

If the Microkernel receives all expected delivery confirmations, it will inform the sender port (Port A) about the outcome. Port A will then acknowledge the successful processing of the message to the sending message-oriented middleware (MOM).

3.2. Error Scenarios

In this section, we will look at some error scenarios that can happen during the routing and how they should be handled.

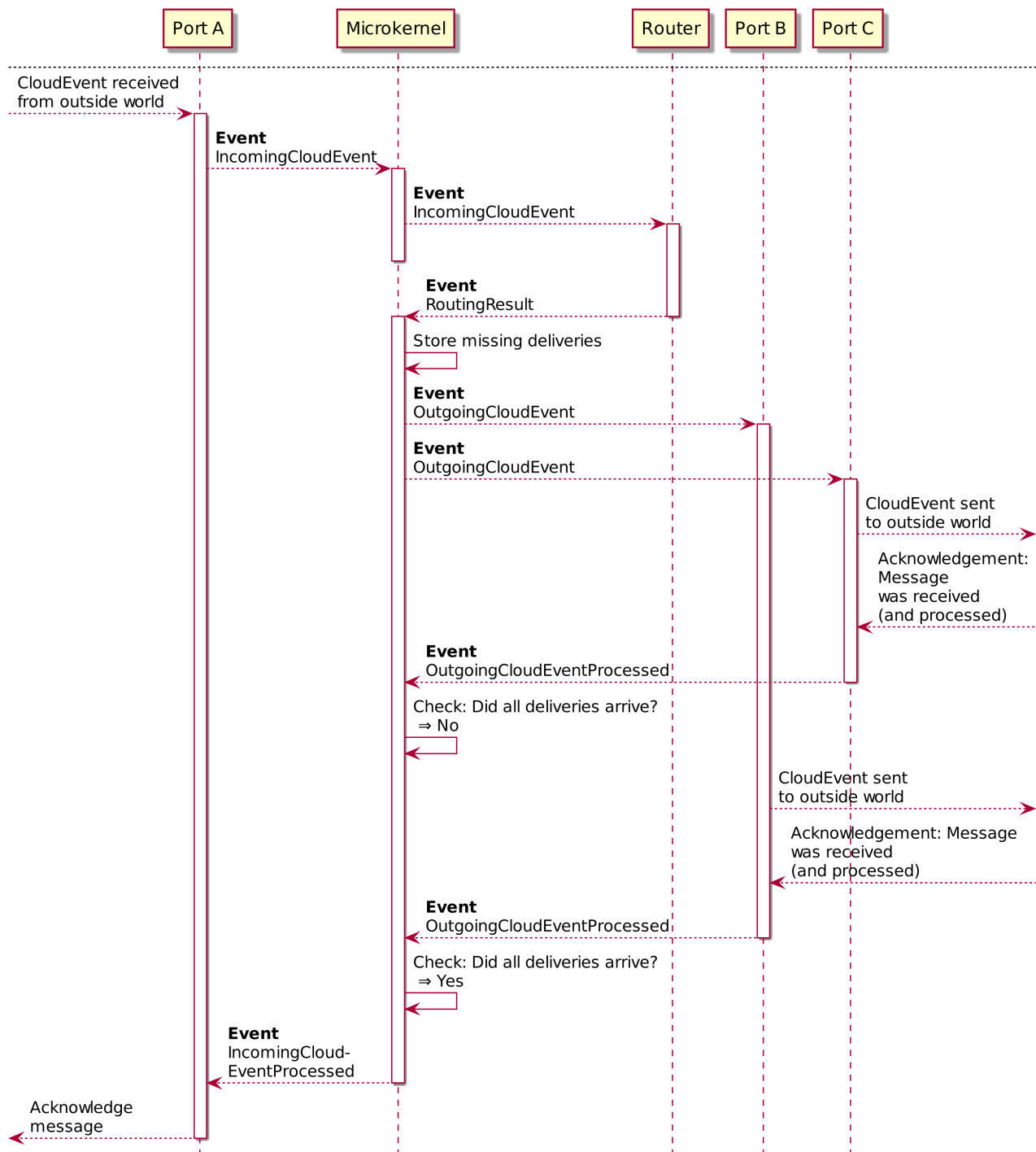


Figure 3.2.: Routing with an "At Least Once" Delivery Guarantee

In the "Best Effort" routing scenario, each plugin has to handle errors individually and errors are never communicated to other plugins. An unrecoverable failure can lead to a message being dropped.

In the "At Least Once" routing scenario, by design, every error that is not communicated back to the input port results in a missing acknowledgment. A missing acknowledgment results in a delivery timeout on the MOM, which therefore results in a resend from the MOM to the Port A (the exact behavior varies depending on the underlying protocol, MOM product, and configuration).

Additional steps were taken to provide a faster error recovery on failed routings:

- When a routing fails on the output ports (Port B or C in our example), and no `OutgoingCloudEventProcessed` events arrive at the Microkernel, the metadata that is stored by the Microkernel would be stored there forever and cause a memory leak. To prevent this situation from happening, a time to live (TTL) is set for the metadata of each event stored by the Microkernel. The Microkernel deletes the metadata when its TTL ends. The timeout is also sent to the input port (Port A in our example) to inform it that the message could not be delivered.
- If an output port cannot deliver the CloudEvent, it should directly inform the Microkernel and not wait for the TTL to run out. This makes the routing outcome faster, the memory usage smaller, and gives the sender port additional information.
We added a mechanism for the outgoing port to inform the kernel and then the incoming port about a failed delivery and if the error is recoverable. With this information, events that are undeliverable because of an unrecoverable issue can be isolated and may be moved to a dead letter queue.

These steps result in faster routing, however, the message order can not be guaranteed in an error scenario. However, the message order was not a requirement and is not important for CARU.

4. Implementation

In this chapter, we are going to take a look at the implementation of the design changes discussed in the previous chapter and highlight some interesting insights.

The implementation of the design changes was rather straight forward. We planned and documented the changes mostly ahead of the implementation phase and then carried out our ideas according to our drafts. We will summarize all changes in Section 4.1 "Overview" and then highlight some of the more noteworthy ones in Section 4.2 "Noteworthy Modifications".

4.1. Overview

The changes made directly to the CloudEvents Router can be summarized as followed:

- Changes to the Microkernel and the whole CloudEvents Router
 - Update the Microkernel to support "At Least Once"
 - Update the bootstrapping process to make the loading process more dynamic
 - Move to the new CloudEvents Rust SDK
- Extensions to the **Ports** (New communication ports and extended functionality)
 - Update the Eclipse Paho MQTT based MQTT port
 - Add an Eclipse Mosquitto base MQTT port
 - Add an AMQP port
 - Add "At Least Once" acknowledgment features to the UNIX port
- Simplification of the router usage by adding configuration options
 - Add a loader component
 - Add a new config loader component
 - Add new functions to simplify the interaction with the dynamic configuration setup
 - Simplify some examples by using the new loader and config loader
- Simplification of the router usage by making it available on registries and other platforms
 - Make all router components available on the Rust package registry
 - Make the router available as a Docker image
- Addition of health check functionality for reliable usage in a cloud setup
 - Add health check functionality
 - Add a health check port
- Changes and extensions for testing
 - Extend the functionality of the sequence generator port
 - Add a sequence validator port
 - Add more tests
 - Add some large scale integration tests
 - Add more examples

- Code quality improvements and minor changes
 - Small refactorings, e.g., to simplify the error handling with the new question mark operator
 - Improve the documentation
 - Update the developer setup instruction

More details can be found in the arc42 documentation in Appendix D "Architecture Documentation (arc42)". All changes and the source code can be found on GitHub¹.

In addition, the following changes were made to support the CloudEvents Router:

- Modifications to the Eclipse Mosquitto library
 - Add an option to the library to delay the acknowledgment until after the processing of a message is done
 - Open a pull request to contribute the changes back to the original library²
- Modifications to the Rust wrapper of the C based Eclipse Mosquitto library³
 - Fork and publish the library with a new name.
(The library was not actively maintained, and pull requests were never reviewed)
 - Update the library to work with the newest version of the Eclipse Mosquitto client library
 - Automate the creation of the Rust bindings from the C header files
 - Download, build, and link the Eclipse Mosquitto library directly during the build of the Rust library. (Originally, the Eclipse Mosquitto shared library had to be installed separately)
 - Add an automated build pipeline (Continuous Integration (CI))

A list of all code repositories that code was contributed to can be found in Appendix I "Code Appendixes Listings".

4.2. Noteworthy Modifications

Our biggest concern was the update of the already existing MQTT port to support the new delivery guarantee "At Least Once". During our research on different protocols, we learned that MQTT handles some guarantees quite differently than the other researched protocols.[13]

For that reason, we chose to start with this port.

An additional important focus in our implementation phase was to improve the "Getting Started" experience of new users. With the steps explained in Section 4.2.2 "Improve the Getting Started Experience", it should be much easier for interested people to start working with the CloudEvents Router.

4.2.1. Adaption of the MQTT Port

As mentioned before, our biggest concern was the adaption of the MQTT port. In the research phase, we realized that the MQTT specification does not define if the acknowledgment for a message should be sent before or after the application has processed the message. This affects messages sent with QoS 1 ("At Least Once" delivery guarantee) and QoS 2 ("Exactly Once" delivery guarantee). To implement our design changes, it should be ensured that a message is only acknowledged after the application has processed the message successfully. Otherwise, messages that are sent with an "At Least Once" delivery guarantee could be lost (as described in Section 3.1 "Design Changes for Reliable Messaging").

¹<https://github.com/ce-rust/cerk/compare/2019-12-19...2021-01-15>

²<https://github.com/eclipse/mosquitto/pull/1932>

³<https://github.com/ce-rust/mosquitto-client-wrapper>

Library	ACK before Processing	ACK after Processing
Eclipse Paho MQTT C	YES	NO, open issue on GitHub ^a
Eclipse Paho MQTT Java	YES	YES ^b
Eclipse Paho MQTT Python	YES	NO, open issue on GitHub ^c
Eclipse Paho MQTT Rust ^d	Same functionality as Paho C	
MQTT.js	YES	YES ^{e,f}
rumqtt	YES	NO ^g

Table 4.1.: MQTT libraries and there acknowledgment (ACK) capability for QoS 1

^a<https://github.com/eclipse/paho.mqtt.c/issues/522>^b<https://github.com/eclipse/paho.mqtt.java/blob/v1.2.5/org.eclipse.paho.mqttv5.client/src/main/java/org/eclipse/paho/mqttv5/client/IMqttAsyncClient.java#L823-L833>^c<https://github.com/eclipse/paho.mqtt.python/issues/348>^dOnly a wrapper around the C library^e<https://github.com/mqttjs/MQTT.js/pull/697>^f<https://github.com/mqttjs/MQTT.js/issues/691>^g<https://github.com/bytebeamio/rumqtt/blob/master/rumqttd/src/state.rs#L161-L211>

During the student research project, we already implemented an unreliable MQTT port. At the time, we decided to use the Eclipse Paho MQTT library because it was, and still is, the most downloaded general-purpose MQTT library⁴. In addition, CARU already had experience with the Python version of it.

When we learned about the mentioned imprecision in the MQTT specification, we checked the implementation of the Eclipse Paho MQTT library, and discovered that the library acknowledges a message before handing it over to the application. In Section 4.2.1 "First Attempt: Patch the Eclipse Paho MQTT Library", we explain how we tried to change the behavior of the library and why our attempt failed.

After our unsuccessful attempt to adapt the Eclipse Paho MQTT library, we had to find an alternative approach. We decided to try the Eclipse Mosquitto client library, which is not as popular as the Eclipse Paho MQTT library, but as part of the popular Eclipse Mosquitto MQTT broker, it is battle-tested and well maintained. Unfortunately, it also acknowledges the message before processing it. Luckily, its architecture allowed us to adapt it to our requirements without running into the same issues as we did with the Eclipse Paho MQTT library. All the details about the required changes can be found in Section 4.2.1 "Second Attempt: Patch the Eclipse Mosquitto Library".

First Attempt: Patch the Eclipse Paho MQTT Library

Our first attempt to fix the problem was to look for MQTT libraries that send the acknowledgment for a QoS 1 message after the processing. Because the documentation of the libraries normally does not explain this behavior in detail, we had to look at the source code directly or try to find published discussions about it. The researched libraries and their acknowledgment behavior are listed in Table 4.1. The focus was set on Eclipse Paho MQTT libraries, as we hoped that they implement it similarly. Additionally, we followed the conversations in the GitHub issues of these libraries. These conversations often pointed to other libraries facing the same problem or libraries that already have solved the problem.

The research showed that many libraries of the Eclipse Paho MQTT project acknowledge the messages before they process it. However, we realized through a documented issue in the C-based Eclipse Paho MQTT library that a patch to change this behavior should be possible.

⁴<https://crates.io/keywords/mqtt?sort=downloads>

We implemented the patch in the C library and changed the Rust wrapper library to use our patch. The content of the patch is shown in Listing 2 and only contains a really small change that switches the order of the function calls for the processing and the acknowledgment of the message. The first tests showed that the change was effective; however, later integration tests showed that there was an error: The new code worked as expected when the router was just receiving or sending over the MQTT port. But sending and receiving at the same time was not possible. The problem was that the library code uses a semaphore to lock the access to resources in a multi-threaded environment. Additionally, the library shares these semaphores between multiple instances of the library. This resulted in a deadlock situation and made it impossible for us to use the library the way we had first intended.

To fix this issue, the small patch in Listing 2 is not sufficient. The whole locking mechanism would need to be changed, which is a big task involving much risk.

Testing the necessary changes is also not trivial because bugs often only appear in some special concurrency scenarios. Another problem we see is that the maintainer of the library would probably not accept a pull request containing such a big change.

```
diff --git a/src/MQTTProtocolClient.c b/src/MQTTProtocolClient.c
index 5dd0112b..6c62dc74 100644
--- a/src/MQTTProtocolClient.c
+++ b/src/MQTTProtocolClient.c
@@ -317,10 +317,9 @@ int MQTTProtocol_handlePublishes(void* pack, int sock)
rc = SOCKET_ERROR; /* queue acks? */
else if (publish->header.bits.qos == 1)
{
-    /* send puback before processing the publications because a lot of return
-   ↪ publications could fill up the socket buffer */
+    Protocol_processPublication(publish, client, 1);
+    /* send puback AFTER processing the publications */
rc = MQTTPacket_send_puback(publish->MQTTVersion, publish->msgId, &client->net,
-   ↪ client->clientID);
-    /* if we get a socket error from sending the puback, should we ignore the
-   ↪ publication? */
-    Protocol_processPublication(publish, client, 1);
}
else if (publish->header.bits.qos == 2)
{
```

Listing 2: Patch for the Eclipse Paho MQTT C Library.

The changes were made on a fork of the GitHub repository.

<https://github.com/ce-rust/paho.mqtt.c/commit/92051a6d98a8a56c666797a438246fc477accd77>

For these reasons, we decided to try a different route and saw the following options:

Implementing a process-based runtime The locking problem of the Eclipse Paho MQTT library is caused by the different instances of the MQTT client sharing the same resources like threads and locks. If each of the instances would run in a separate process, they would not share these resources, and the deadlock could not occur. Our modular architecture would allow the implementation of a process-based runtime plugin that runs each **Port** in its own process, but this approach has two main disadvantages:

First, it would couple the MQTT **Port** to a specific runtime. Second, a process-based runtime creates a big communication-overhead because the data needs to be serialized and deserialized when sent between processes. This is not ideal for software that is intended to run on an embedded device with resource restrictions.

Switching to another MQTT library The other option was to switch to another MQTT library. For this solution, we would need to implement a new **Port**. The additional problem was that we did not find another Rust-based MQTT library that implemented the needed behavior. Therefore, either a fix had to be implemented in a yet to us unknown Rust-based library, or a new Rust wrapper for an already existing non-Rust-based library had to be written.

We decided to try the latter option and adapt another library available in Rust.

Second Attempt: Patch the Eclipse Mosquitto Library

We decided to implement a new port based on the Eclipse Mosquitto client library. The library was selected together with CARU, as they already use the Eclipse Mosquitto broker.

Eclipse Mosquitto is mainly an MQTT broker. However, the project also includes an MQTT client library. The library is implemented in C, but there is also a Rust wrapper available. Unfortunately, the Rust wrapper is out of date and not well maintained. Thankfully it is really lightweight and can therefore easily be maintained by ourselves.⁵

The patch for the Eclipse Mosquitto client library is quite similar to the patch for the Eclipse Paho MQTT library (Listing 3).

We tested our changes and then implemented a new **Port** with the Eclipse Mosquitto client library. The proposed solution worked and was published as a new **Port** named `cerk_port_mqtt_mosquitto`⁶. Fortunately, it did not suffer from the same limitations as our first attempt with Eclipse Paho MQTT. We decided to implement the acknowledgement behavior as a configurable option in the library so that our changes could be merged back to the Mosquitto library with a pull request.⁷

The pull request was received with positive feedback. However, at the moment, it is unclear how to proceed with QoS 2 and we are awaiting feedback from the maintainers.

This means that, at the moment, we have to work with a forked library for our **Port**. Our workaround is to install the binary of the patched Eclipse Mosquitto client library on the machine where the router runs. As soon as the pull request is merged, we will be able to switch back to the official library.

Results

Patching the Eclipse Mosquitto library worked out well. The old **Port** based on the Eclipse Paho MQTT library is still in the project, but we have documented its limitations.

4.2.2. Improve the Getting Started Experience

After we submitted the student research project, we open-sourced the router. The project contained multiple usage examples of the router and a developer starting guide for Ubuntu Linux and Arch Linux.

These examples were helpful and facilitated the start; however, there was space for improvement. The router was only accessible via the GitHub source-code, and the users had to build the router from the source.

In addition, the configuration of the router defining at which port a MOM is reachable had to be defined in the source code. This meant that a change of the configuration was not possible without recompiling the whole router.

⁵The Rust wrapper was forked and published under a new name. Our wrapper fixed some problems with the old wrapper. In addition, the new wrapper is able to compile and statically link the underlying C library. In the old wrapper, the C library was dynamically linked and had to be installed on the target machine. The wrapper is publicly available at <https://github.com/ce-rust/mosquitto-client-wrapper>

⁶https://github.com/ce-rust/cerk/tree/master/cerk_port_mqtt_mosquitto

⁷<https://github.com/eclipse/mosquitto/pull/1932>

```

diff --git a/lib/handle_publish.c b/lib/handle_publish.c
index 31d2dba81..8255ba9b7 100644
--- a/lib/handle_publish.c
+++ b/lib/handle_publish.c
@@ -135,7 +135,9 @@ int handle__publish(struct mosquitto *mosq)
     return MOSQ_ERR_SUCCESS;
     case 1:
         util__decrement_receive_quota(mosq);
-        rc = send__puback(mosq, mid, 0, NULL);
+        if(!mosq->delayed_puback){
+            rc = send__puback(mosq, mid, 0, NULL);
+        }
         pthread_mutex_lock(&mosq->callback_mutex);
         if(mosq->on_message){
             mosq->in_callback = true;
@@ -148,6 +150,9 @@ int handle__publish(struct mosquitto *mosq)
             mosq->in_callback = false;
         }
         pthread_mutex_unlock(&mosq->callback_mutex);
+        if(mosq->delayed_puback){
+            rc = send__puback(mosq, mid, 0, NULL);
+        }
         message__cleanup(&message);
         mosquitto_property_free_all(&properties);
         return rc;

```

Listing 3: Patch in `handle_publish.c` for the Eclipse Mosquitto C Library.

The changes were made on a fork of the GitHub repository and are subject of a pull request.
<https://github.com/eclipse/mosquitto/pull/1932>

Our goal was to make the router more easily accessible for developers of new **Ports** (or components for the CloudEvents Router in general) as well as for users wanting to use the product without any knowledge of Rust.

Configuration Loader

The first step was to create a new configuration loader that reads the configuration from a configuration file. Before this thesis, there was only one configuration loader type: The `config_loader_static`. This plugin had to be implemented for each version of the router and required to statically define all configurations in code.

To solve this shortcoming, we implemented a new `config_loader_file` that loads the configuration from a JSON file.

The configuration loader was already defined as a plugin type of the CloudEvents Router, and therefore, the kernel and other plugins did not need to change.

Loader

Inspired by the `config_loader_file`, we also implemented a `cerk_loader_file`. `cerk_loader_file` is a loader that helps the users start the router only with the plugins they need. Before this helper was introduced, all plugins had to be manually defined in the bootstrap function. This meant that all changes needed to be done in code and required a recompilation to take effect.

This new loader introduces an additional configuration file, which allows the user to specify the plugins of the CloudEvents Router in a JSON file. However, there is still the need to define all plugins in the code, but now the plugins that get instantiated during runtime can be chosen in the configuration file. The loader is not yet capable of loading plugins that were not statically linked in the code.

Published Rust Packages

As a next step, we published all components of the router on crates.io, which is the package registry of the Rust package manager Cargo. With this step, the developers can now use the router without depending on the Git repository. They are now able to define the needed components of the router as a dependency of their project, and the components will be downloaded and compiled automatically. Another benefit is that the update process is standardized.[14]

The Official CloudEvents Router Docker Image

Now, with a configuration loader, a loader for the plugins, and easy to install packages via the package manager, a ready-to-use Docker image can be built and published. The Docker image contains a variety of plugins and can be downloaded and deployed by anyone. With that step, the router is now even accessible to people that do not know anything about Rust and have not installed any of the tools needed for Rust development. Thanks to the two configuration files, the router can still be configured for a multitude of use cases: A custom set of port instances and corresponding configurations can be set and custom routing rules can be defined.

Results

Our validation results and early adopter feedback from CARU developers suggest that it is now easier to start working with the router for both Rust developers and end-users. The router is now accessible over the following ways:

- All router components are now published on the crates.io (the Rust package registry) and can be downloaded with Cargo (the Rust package manager).⁸
- The documentation is now available on docs.rs.⁹
- The router is published as a ready-to-use Docker image on the Docker Hub that can be deployed in seconds.¹⁰

With these access possibilities, the CloudEvents Router is now as accessible as well-established libraries and products.

⁸<https://crates.io/keywords/cerk>

⁹https://docs.rs/cerk/*/cerk/

¹⁰<https://hub.docker.com/repository/docker/cloudeventsrouter/cerk>

5. Validation

In this chapter, we discuss our method of testing the new behavior of the router and showcase the improved developer experience.

We ran several different types of tests, which we are all going to explain separately: First, we started with single message routing integration tests to deterministically validate the newly implemented behavior (Section 5.1 "Single Message Routing Integration Tests"). Next, we also tested with a heuristic approach, which verifies the behavior of the router in a more realistic, heavy-load scenario (Section 5.2 "Load Test"). We use the heavy-load to also take a performance snapshot of the new router (Section 5.3 "Performance Test"). Finally, we showcased the improved developer experience by implementing a customized version of the router for the use in a real-world case of CARU (Section 5.4 "Developer Experience Showcase").

5.1. Single Message Routing Integration Tests

To verify that no messages can get lost in an "At Least Once" routing scenario, we designed two simple integration tests. We then implemented these integration tests for both supported messaging protocols: MQTT and AMQP.

5.1.1. Test Method

Both integration tests are visualized in Figure 5.1:

Successful Routing Scenario The first test case validates the successful routing of one event from the input to the output channel.

Failed Routing Scenario The second test also tries to route a message from the input to the output channel. However, now the output channel will not accept the message and the message should consequently not be acknowledged on the input channel, which will result in an unconsumed message.

For both integration tests, the same setup is needed:

- Our CloudEvents Router that routes messages from one input to one output channel with the new "At Least Once" delivery guarantee.
- Two channels: One for the input and one for the output of the router.
- A test-executer that configures the behavior of the output channel. It places the message on the input channel and checks the successful outcome of the test.

We implemented the integration tests twice: once for MQTT and once for AMQP. The tests have some minor implementation differences to accommodate the individual protocols:

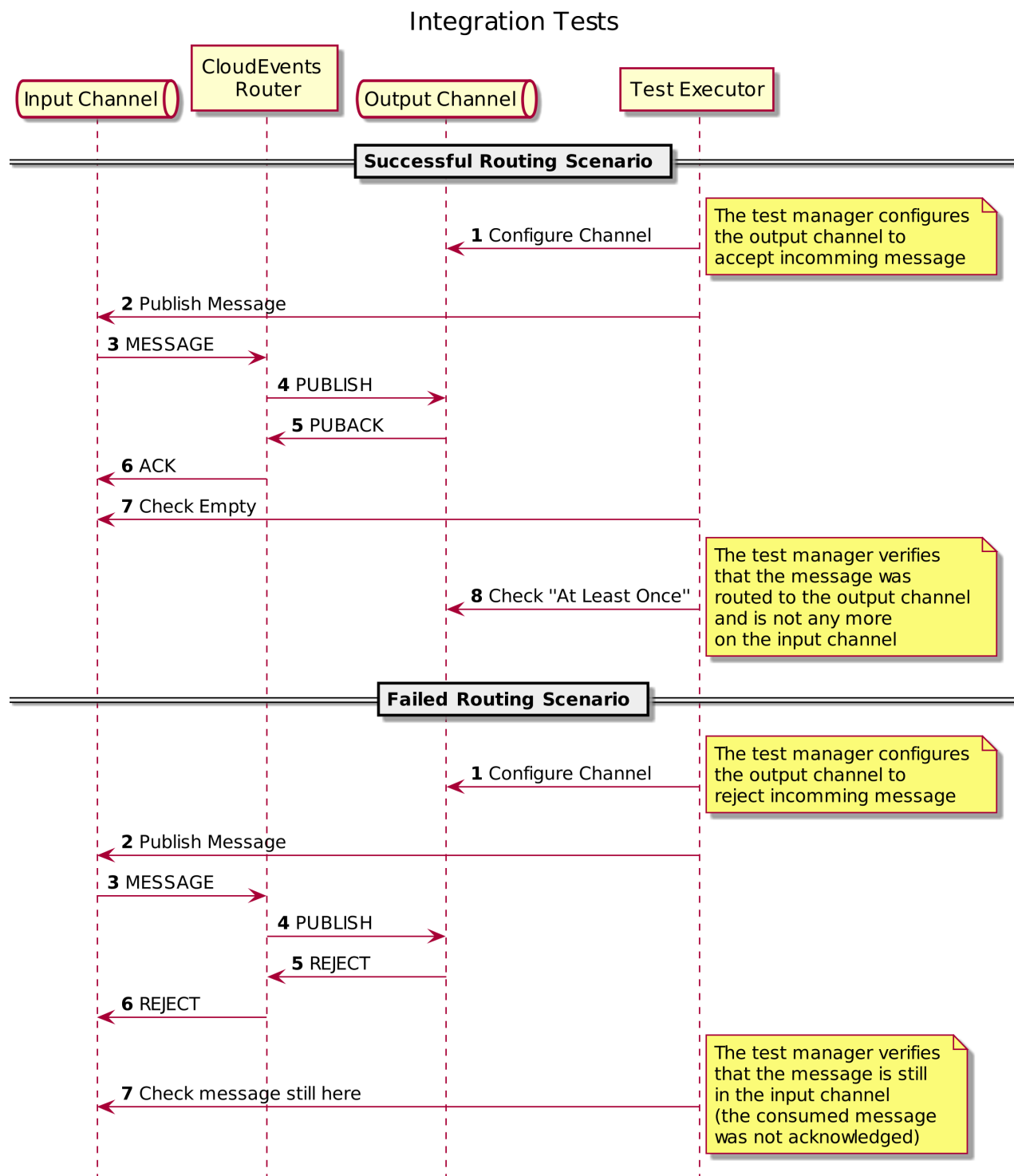


Figure 5.1.: Single Message Routing Integration Tests

MQTT The rejection of the messages was implemented using the `message_size_limit`¹ configuration of the Mosquitto broker. This parameter defines the maximum message size that the broker accepts. The parameter differs between the two test cases: For one, the size is set to allow the message, while it needs to reject the message for the other.

AMQP In the AMQP test we used RabbitMQ as our MOM. RabbitMQ has an option called `max-length`² that limits the maximum amount of messages a queue can hold. If `max-length` is set to zero and the option `overflow`² is set to `reject-publish`², RabbitMQ can be configured to reject all messages.

5.1.2. Test Result

In the Successful Routing Scenario, both the MQTT and the AMQP port were able to send the messages to the output channel and acknowledged them on the input channel. In the Failed Routing Scenario, they did not acknowledge the message, and the message remained stored on the input channel.

The test setup is documented and published on GitHub.³

5.2. Load Test

After testing the router in a deterministic way, we also tried a heuristic approach. The integration tests discussed in the previous section show that the ports work as intended; however, it remained unclear if they would be able to handle real workloads in a reliable fashion. To dismiss this concern, the load test was created.

5.2.1. Test Method

This test aims to evaluate if the delivery guarantee is upheld even under heavy load and challenging conditions. For this, we let our router transmit a large number of messages from an input channel to an output channel. At the same time, we simulate system crashes by repeatedly restarting the CloudEvents Router. In the end, we check if all messages have arrived on the output channel.

To simulate a real-world production environment, we deploy all components of the load test on a Kubernetes cluster. The deployment is visualized in Figure 5.2 and consists of the following components:

- A MOM with two channels: one as the input of the router and one as the output of the router.
- Our CloudEvents Router that routes from the input channel to the output channel. The router is either configured to use the new delivery guarantee "At Least Once" or the "Best Effort" mode from the student research project.
- A service that sends 100'000 messages to the input channel, each with a unique identifier (ID). We used our CloudEvents Router with a special port that generates a specified amount of messages.⁴

¹<https://mosquitto.org/man/mosquitto-conf-5.html>

²<https://www.rabbitmq.com/maxlength.html>

³<https://github.com/ce-rust/cerk/tree/master/integration-tests>

⁴https://docs.rs/cerk_port_dummies/0.2.11/cerk_port_dummies/fn.port_sequence_generator_start.html

- A service that validates if all 100'000 messages were routed to the output channel. It is important to verify each message instead of only checking the number of arrivals, as messages could be duplicated.

For this verification, we also used an instance of our CloudEvents Router. This instance has a special sequence validator port that checks if all messages have arrived.⁵

- chaoskube, an implementation of a Chaos Monkey for Kubernetes. The Chaos Monkey is a tool that randomly stops a service without any notice. The tool was created by Netflix to help with the creation of a resilient infrastructure where any service can be interrupted at any time without impacting the user experience.[15] This service is configured to terminate our CloudEvents Router regularly. Kubernetes will then automatically start a new instance of the router.

Normally, the Chaos Monkey terminates a random service of the whole deployment in a certain time interval. However, for the repeatability and reproducibility of the test, our Chaos Monkey terminates only the CloudEvents Router under test, in 10-second intervals. The validation service, the generator service, or the MOM with the two channels are exempt from the restart process.

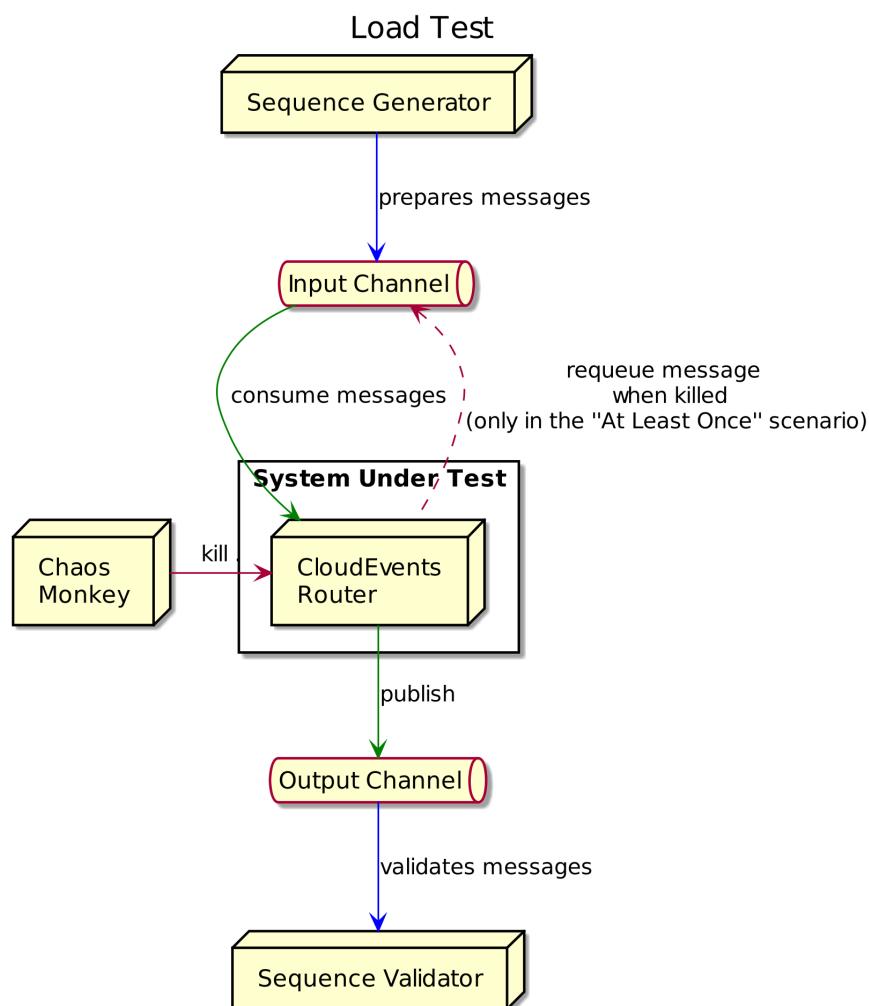


Figure 5.2.: Load Test

⁵https://docs.rs/cerk_port_dummies/0.2.11/cerk_port_dummies/fn.port_sequence_validator_start.html

Queues

► All queues (4)

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
input	classic	D DIL DLX	running	21,766	30	21,796	0.00/s	15,648/s	15,651/s	
input-dlx	classic	D	idle	0	0	0				
output	classic	D DIL DLX	running	0	16	16	15,645/s	15,681/s	15,682/s	
output-dlx	classic	D	idle	0	0	0				

► Add a new queue

Figure 5.3.: Performance Test of the Router in the "At Least Once" Mode
Screenshot taken from the RabbitMQ web interface

5.2.2. Test Result

We ran the test with the "At Least Once" and the "Best Effort" routing mode multiple times. With the "At Least Once" delivery guarantee, all messages were routed successfully to the output channel. Not a single of the 100'000 messages was lost. In the "Best Effort" routing mode, on average 7000 messages were lost per test run. The results differed widely depending on the used protocol ports. The test is explained and analyzed in more detail in Appendix E "Load Test Statistics".

5.3. Performance Test

Additionally, the load test setup allows us to easily test the performance of the router. With the help of RabbitMQ, the performance of the router under test can be measured (see Figure 5.3). The "At Least Once" router manages to route 15'000 messages per second in this setup and thus exceeds our performance needs.⁶

5.4. Developer Experience Showcase

To validate the effectiveness of the changes made to improve the developer experience, we met with an engineer from CARU to put together a version of the CloudEvents Router that is tailored to the CARU Device and is able to provide value to the company. Our main goal was to see how smooth the developer experience is to adapt and configure the CloudEvents Router. As the baseline, we used the experience of adapting and configuring the CloudEvents Router for the final presentation of the student research project. For the live demo during this presentation, we had to implement a custom UNIX socket port⁷ and configure the router⁸. In addition, the CloudEvents Router built for this experiment will be evaluated by CARU for productive use on the CARU Device.

Section 5.4.1 "Test Method" introduces the specifics of the target environment where this version of the router will be deployed to. Then, in Section 5.4.2 "Test Result", we conclude this experiment and discuss steps that could be taken to improve the developer experience further. The technical details are explained in a step-by-step guide that can be found in Appendix G "Developer Experience Showcase: Implementation Steps". These steps are very low-level and require some familiarity with Rust. The source code is published on GitHub.⁹

⁶The target performance was defined as 500 messages per second, more details can be found in the Appendix D "Architecture Documentation (arc42)"

⁷https://github.com/ce-rust/cerk/tree/2019-12-19/cerk_port_unix_socket

⁸https://github.com/ce-rust/cerk/tree/2019-12-19/examples/src/unix_socket_and_mqtt_on_armv7

⁹https://github.com/caruhome/cortex_router

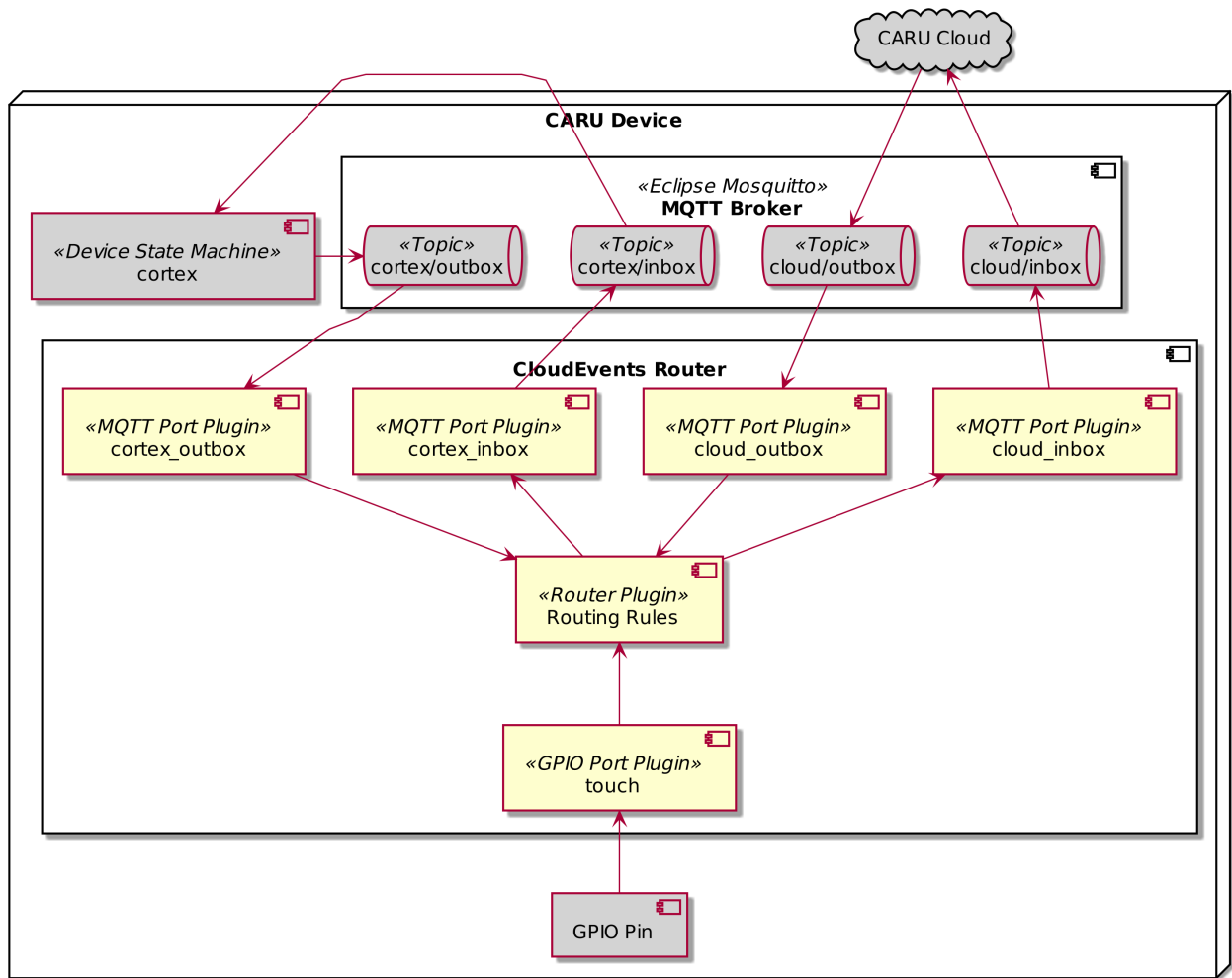


Figure 5.4.: Target Environment of the Developer Experience Showcase

5.4.1. Test Method

Figure 5.4 shows what was built during this experiment. The gray components already existed on the CARU Device while the yellow components were newly introduced.

On the CARU Device, an Eclipse Mosquitto MQTT broker runs as the central communication hub for the services on the device. It is also responsible for communication with the cloud and buffers messages if the device is temporarily disconnected from the internet.

Another important service running on the device is the Device State Machine, which is internally called Cortex. The name is inspired by the cerebral cortex, which is an integral part of the brain. On the CARU Device, the Cortex is responsible for managing the user interface of the device and is implemented as a state machine. For example, it needs to react to the touch button being pressed by updating the device state and triggering reactions like turning on a specific light pattern or play a specific sound file.

The last important component for this experiment, besides the CloudEvents Router, is the General Purpose Input/Output (GPIO) pin that physically connects the central processing unit (CPU) to the controller-chip of the touch button. The touch button is a touch-sensitive area on top of the device. It is the only user input method besides the microphone, which detects voice commands from users. The GPIO pin is used to transmit the current state of the touch button; if the touch button is being pressed, the pin is set to one. When the button is not being pressed, the pin is set to zero. Until now, a small C program read the state of the GPIO pin in short intervals, and whenever the value flipped, it sent a notification over a UNIX socket to the Cortex.

The role of the CloudEvents Router is to link all these components together and replace the C program that monitored the GPIO pin. To fulfill its role, the router is composed of five `Port` instances: Four MQTT port instances and a GPIO port instance. The four MQTT ports are used to communicate with the Device State Machine and the CARU Cloud. This communication is facilitated by the MQTT broker, which provides a channel pair for each communication partner. Channels are called topics in MQTT. The topic pairs consist of an `inbox` topic to transfer events from the router to the communication partner and an `outbox` topic for the other direction. The MQTT broker is configured to bridge the `cloud/inbox` and `cloud/outbox` topic with the corresponding topics on an MQTT broker in the cloud, which is powered by the Amazon Web Services (AWS) IoT Core service.

The CloudEvents Router could also be communicating directly with the topics in the cloud, but, unfortunately, AWS IoT Core bills per open MQTT connection. Because other services are still using the Eclipse Mosquitto broker to communicate with the cloud, directly connecting the router to the cloud was not an option for CARU. Though, this could be changed in the future.

Additionally, a custom port to read the GPIO pin has to be implemented. The purpose of this port is to translate the pin state to CloudEvents. Because the GPIO protocol is extremely simple (the pin can have the state zero or one), this is straightforward to implement and perfectly fits the scope of this experiment.

As the last step, routing rules need to be set up to control the flow of CloudEvents between the ports.

By building this version of the CloudEvents Router, CARU wants to add another important stepping stone on its journey to fulfill the vision of the unified event plane. The CloudEvents Router will help them accelerate the adoption of CloudEvents for the events exchanged on the CARU Device. Additional services could be integrated into the unified event plane by implementing them as custom ports or communicating with them via the MQTT broker.

With the GPIO port, the first steps are taken to link hardware-dependent and hardware-independent services solely over the unified event plane. In the future, this will enable the faster development of hardware-independent services by running them on the developer computer while they exchange events with hardware-dependent services over the network. It will also help to automate integration tests by allowing the simulation of hardware-dependent services. With the old, C-based GPIO reader, interested services needed access to the UNIX socket, which required them to run on the CARU Device.

5.4.2. Test Result

The experiment showed that we could improve the developer experience when comparing it to the one right after the student research project. We could significantly reduce the boilerplate code required to configure and extend the CloudEvents Router. However, during the implementation of this tailored CloudEvents Router, we realized that there are areas that could be improved upon further.

We see the greatest potential for improvement in the configuration management. The newly added loaders¹⁰ enhance the configuration experience significantly, but they also add multiple configuration files that need to be managed. This inconvenience could be resolved by a more powerful configuration framework that provides configuration from one place to all components with a single, well-structured configuration file.

An additional factor that diminishes the developer experience is the MQTT port, more precisely, the underlying Eclipse Mosquitto library. In theory, a port instance is able to send and receive messages at the same time. However, because of the limitations of the Eclipse Mosquitto library and the way we are using it, each MQTT port instance is only usable for one direction at a time. In detail, it comes down to the way the library uses resources like threads and locks, and how we are blocking callbacks to delay the acknowledgment. This limitation means that we needed four instead of two port instances for this use case. This, in turn, results in unnecessary network overhead because each

¹⁰see Section 4.2.2 "Configuration Loader" and Section 4.2.2 "Loader"

port instance opens its own connection.

Additionally, as mentioned in Section 4.2.1 "Second Attempt: Patch the Eclipse Mosquitto Library", we needed to fork and patch the Eclipse Mosquitto library to provide delivery guarantees. The use of this fork adds additional steps that hamper the developer experience. We hope that the fork will soon be merged back into the main version, erasing this overhead.

5.5. Summary

We showed that the "At Least Once" routing works as intended and fixed the shortcomings of the "Best Effort" routing.

While we could improve the developer experience and reduce the required boilerplate code, we still see room to further improve the maintainability and extendability of the CloudEvents Router.

6. Conclusion and Outlook

In this chapter, we draw the conclusion of this thesis, summarize our findings, and discuss the next steps that could be taken to advance the CloudEvents Router further. Finally, we will present an outlook on topics worth exploring in future work.

6.1. Outcome

We started with a substantial research task where we researched common patterns for reliable messaging and how they work. We found and discussed many patterns and techniques to provide the common delivery guarantees "At Least Once", "At Most Once", and "Exactly Once".

In addition, we analyzed the messaging protocols that have protocol bindings defined in the CloudEvents specification. The goal of the analysis was to understand how these protocols implement delivery guarantees and how far these implementations are interoperable.

We used this information to theoretically show how each delivery guarantee can be provided in a routing scenario. The analysis showed that an "At Least Once" delivery guarantee semantics would make most sense for the CloudEvents Router in the context of CARU.

In the next step, we implemented the "At Least Once" delivery guarantee for the CloudEvents Router and tested the correctness of the implementation in different test scenarios. The implementation was done in the Microkernel and in two messaging protocol ports: MQTT and AMQP.

The addition of a delivery guarantee required only a small architectural change, which showed that the original router design was flexible enough.

Implementing the AMQP port and the upgrade from the "Best Effort" delivery guarantee to the "At Least Once" delivery guarantee in the Microkernel was straightforward. However, the upgrade from the "Best Effort" delivery guarantee to the "At Least Once" delivery guarantee on the MQTT port was very difficult.

To achieve a reliable MQTT port, we had to change the underlying library from Paho to Mosquitto and create a fork of the library.

The implementational changes to the CloudEvents Router solved the main shortcoming of the CloudEvents Router version from the student research project: Now the router is able to route messages from one channel to 0 – N other channels while providing an "At Least Once" delivery guarantee for each destination, which means that we can be sure that no events get lost. However, "sure" means that no messages were lost in our specific test scenarios. These results give us, CARU, and other potential users confidence that the CloudEvents Router works properly. Still, our test scenarios are not proofs in a mathematical sense.

The "At Least Once" delivery guarantee ensures that no messages are lost while still providing a good balance between reliability and performance.

In addition to the main goal, we made proposals in the context of the paper "Reliable Messaging – Patterns and Strategies for Message Routing" on how delivery guarantees could be integrated into AsyncAPI, asyncMDSL, and the CloudSubscriptions Discovery API. We also made additional improvements to our CloudEvents Router so that it is easier to find and use the CloudEvents Router for new users.

6.2. Findings

We learned many things that need to be considered while implementing reliable messaging. "Exactly Once" proved to be especially hard to implement when striving for statelessness. It showed us that this guarantee is not achievable in general without a state that survives a process or system crash.

Nevertheless, the biggest lesson for us, as well as for CARU, is that MQTT behaves differently from most other protocols when looking at the "At Least Once" and "Exactly Once" delivery guarantees. In MQTT, the "At Least Once" and "Exactly Once" semantics are not defined as strictly as in most analyzed protocols. MQTT does not define if messages should be acknowledged before or after the application processed them. The specification for the protocol only contains a none-normative example in which the messages are acknowledged before they are processed. As a result, many MQTT libraries also acknowledge the messages before they are processed and do not offer an option to change that behavior. Acknowledging a message before it is successfully processed can result in message loss if the application crashes before or during the processing of the message.

The mentioned problem existed in the library we used in our student research project (Eclipse Paho MQTT) as well as in the library we moved to during this thesis (Eclipse Mosquitto). However, we implemented a patch to fix this shortcoming in the Eclipse Mosquitto library and created a pull request to contribute these changes back to the library.

6.3. Outlook

For future work, we propose to look at augmenting the delivery guarantee with resilience capabilities like support for the monitoring and alerting solutions Prometheus of the CNCF. The current router has a health check component to provide some observability. However, there is still room for improvement.

Additionally, it would be interesting to integrate the asynchronous programming APIs of Rust into the whole CloudEvents Router. The APIs to support the asynchronous programming paradigm were not stable when we started implementing the CloudEvents Router. In the meantime, these APIs were stabilized and are now an official part of the language. Asynchronous Rust would help to reduce the resource usage by requiring only one single thread to run.[16–18]

Another feature that would help to integrate the CloudEvents Router into the CloudEvents ecosystem would be to support the new CloudSubscriptions Discovery API and CloudEvents Subscription API specification. With these APIs, other services would be able to find and subscribe to events via standardized APIs. This addition would make the whole configuration approach more dynamic.

Another place for improvements is the structure of the configurations. We suggest investing in streamlining the configuration management. Currently, the different components of the router have to implement their own strategies, which hampers the maintainability of the configuration.

List of Figures

1.1. The CARU Device	12
1.2. CARU's Envisioned System Context	13
1.3. Plugins of the CloudEvents Router after the Student Research Project	14
1.4. Demo Deployment of the Student Research Project	15
1.5. Danger Zone of the Student Research Project CloudEvents Router	16
3.1. Routing with "Best Effort"	24
3.2. Routing with an "At Least Once" Delivery Guarantee	25
5.1. Single Message Routing Integration Tests	36
5.2. Load Test	38
5.3. Performance Test of the Router in the "At Least Once" Mode	39
5.4. Target Environment of the Developer Experience Showcase	40
E.1. Load Test	188

List of Tables

4.1. MQTT libraries and there acknowledgment (ACK) capability for QoS 1	29
---	----

List of Listings

1.	A CloudEvent in the JSON Serialization Format	11
2.	Patch for the Eclipse Paho MQTT C Library	30
3.	Patch in handle_publish.c for the Eclipse Mosquitto C Library	32
4.	GPIO Port Source Code	202
5.	Device Router Source Code	203
6.	Device Router Init Config	203
7.	Routing Rules and Port Configs	204
8.	Device Router Routing Rules and Port Configs	204

Glossary

There are some overlapping glossary entries between the documents of this thesis. Each document contains only the glossary entries which are referred to in that document.

Some glossary entries were originally created during the thesis "CloudEvents Router"[1]. However, because the context is similar they are inherited from there.

AWS IoT Core IoT Core is a service from AWS that helps with securely connecting IoT devices to the cloud. At its core is a serverless MQTT broker that integrates with other AWS services like message queues, databases and more.[19]. 13, 41

CloudEvents Subscription API The CloudEvents Subscription API is a vendor-neutral API specification that defines how consumers could subscribe events from a producer.[20]. 5, 44, see also CloudEvents & API

NATS Streaming NATS Streaming is an extension for the NATS messaging system that adds a "At Least Once" delivery semantics and message persistence.[21]. see also NATS

AWS Lambda AWS Lambda is the Function as a Service (FaaS) offering from AWS. It is tightly integrated with the rest of the AWS ecosystem and allows the implementation of such functions in many programming languages, like Python, Go, or Java.[22]. 13, see also AWS & FaaS

Academic Free License Academic Free License is a open-source license provided by the Open Source Initiative (OSI). It is not a copyleft license.[23]. 55, see also copyleft, OSI & open-source

AgeTech "AgeTech is about digitally-enabling the Longevity economy". The products could be divided into "4 categories of digital-enablement in AgeTech: services purchased by older people; services purchased on behalf of older people; services traded between older and younger people; and services delivered to future older people." [24]. 11

Amazon Elastic Compute Cloud Amazon Elastic Compute Cloud (EC2) is a Virtual Machine (VM) solution from AWS.[25]. see also VM & AWS

AMQP The Advanced Message Queuing Protocol (AMQP) is an open-source messaging protocol with the goal of standardizing the message exchange between enterprises on the wire level. Before version 1.0, the AMQP specification also defined the behavior of the message broker. In version 1.0, the authors narrowed down the scope of the specification to the exchange of messages between two nodes. Because of that, the pre-1.0 and 1.0 version of the protocol are not compatible.[26, 27]. 57

Apache 2 Apache 2 is an open-source license from the Apache Software Foundation. The license allow modifications and redistribution of the software. It has no copyleft.[28]. 55, see also copyleft & open-source

API Is a interface to communicate from one system to an other[29]. 57

arc42 arc42 is a template for documentation and communication of a system or software. It focus on lean and agile development approaches.[30] The template is available in available in different format, one of them is L^AT_EX.[31] It is open-source licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.[32]. 14, 23, 28, see also L^AT_EX

AsyncAPI AsyncAPI is an open-source initiative to make messaging and event-driven architecture easier, similar to what OpenAPI (formerly known as Swagger) does for REST APIs. At the center of AsyncAPI is the specification which is similarly structured to the OpenAPI specification.[33] It is quite new and growing in popularity.. 20, 43

asyncMDSL AsyncMDSL is part of MDSL and handles asynchronous contracts between systems, mostly by Message Channels. The asynchronous messaging part of MDSL was first introduced by De Liberali. The syntax is derived from the patterns described in "Enterprise Integration Patterns".[34, 35]. 20, 43, see also MDSL & AsyncAPI

"At Least Once" To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for a message in a certain time frame, the message is retransmitted.[12]. 5, 16, 19–21, 23, 24, 26–28, 35, 37, 39, 42–45, 51, 53, 181, 183, 184

"At Most Once" "At Most Once" describes that the message is delivered once or if it fails, it will not at all be delivered at all.

This approach is more an anti-pattern for reliable messaging and can not be found in any of our sources. However, Hohpe and Woolf describe something similar in the pattern "Fire-and-Forget", this pattern is part of the "Conversation Patterns" which the author represent as an addition to "Enterprise Integration Patterns".[36, 37]. 16, 19–21, 23, 43, 52

AWS Amazon Web Services (AWS) is a cloud platform provided by Amazon[38]. 57

"Best Effort" "Best Effort" in the contrast to any guarantee. In this context "Best Effort" means that none of the defined delivery guarantees are claimed.. 23, 24, 26, 37, 39, 42, 43, 181, 183

Buildkite Buildkite is a service for coordinating buildpipelines which get executed on own infrastructure. They provide a build agent that can easily be deployed to AWS and other platforms. Buildkite is well integrated with GitHub.[39]. 199, 201, see also AWS, CI & GitHub

bytecode "Bytecode is program code that has been compiled from source code into low-level code designed for a software interpreter. It may be executed by a virtual machine (such as a Java Virtual Machine (JVM)) or further compiled into machine code, which is recognized by the processor."[40]. see also Java & JVM

C C is a programming language developed by Dennis Ritchie. C++ is a superset of C. Later it was standardized by AMSI and ISO.[41]. 28–31, 40, 41, 53–56, 194, 201, see also C++

C++ C++ is a general-purpose programming language designed by Bjarne Stroustrup. Later it was standardized by AMSI and ISO.[41]. 52

C++ Bindings C++ Bindings are wrappers for a programming language to use C++ function in it.[42]. see also C++

Cargo Cargo is the default package manager for Rust. It is really simple, compiles code, installs packages, runs tests, formats source files and generates the documentation. Cargo can be extended with additional subcommands by installing packages via Cargo itself.. 33, 56, 201, see also Rust

CARU CARU AG is an AgeTech startup with the mission to help the elderly to live independently for longer. It is the industry partner of this thesis.. 5, 11, 12, 16, 17, 26, 29, 31, 33, 35, 39, 41, 43, 44, see also CARU Device

CARU Device The CARU Device is a product placed in the living environment of an elderly person to collect various room parameters, learn the behaviors of the inhabitant and alarm relatives or caretakers in case of deviations. Its user experience is optimized for elderly people. It was formerly known as CARU SmartSensor.[2]

The device has two CPUs with two different operating systems on it.. 11–14, 16, 39–41, 193, 194, see also FreeRTOS, Linux & CARU

- Chaos Monkey** The Chaos Monkey is a tool that randomly stops a service without any notice. The tool was created by Netflix to help with the creation of a resilient infrastructure where any service can be interrupted at any time without impacting the user experience.[15]. 38, 183
- CI** "Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early."[43]. 57
- Cloud Native** "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."[44]. 53, see also CNCF
- CloudEvents** The CloudEvents specification is a standard from the CNCF with the goal to "describe event data in common formats to provide interoperability across services, platforms and systems."[4] The standard reached version 1.0 on the 24th of October 2019.. 5, 11–16, 20, 23, 24, 26, 27, 41, 43, 44, 51, 53, 54, 181, 193, see also CNCF
- CloudEvents Router** The CloudEvents Router is the open-source Router that was deployed during the thesis "CloudEvents Router".[1]. 5, 12–17, 19, 23, 27, 28, 32, 33, 35, 37–45, 55, 181, 183, 184, 193–195
- CloudSubscriptions Discovery API** The CloudSubscriptions Discovery API is a vendor-neutral API specification to discover services and the events published by these services. The specification is part of the CloudEvents specification and is intended to be used with the CloudEvents message format.[45]. 5, 20, 43, 44, see also CloudEvents & API
- CNCF** The Cloud Native Computing Foundation is part of the Linux Foundation. The goal of the foundation is to empower the usage of the cloud by providing standards and open-source projects, the use the term Cloud Native.[44] The foundation host project like Kubernetes[46] and Prometheus[47]. Apple, Microsoft, Google and Amazon Web Services are only a couple of the most famous partners.[48]. 57, see also Cloud Native
- copyleft** Copyleft is a method for making a program including all versions and modifications free.[49]. 51, 54
- CSV** comma-separated values. 183, 199
- Docker** "Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises."[50]. 33, 183, 202
- DSL** A Domain Specific Language is a custom language for a specific purpose. It is an abstract language that should solve a description problem in a specific domain.[51]. 57
- Eclipse Mosquitto** Eclipse Mosquitto is an open-source MQTT message broker with an additional command line tool to publish and subscribe from and to MQTT queues.[52]. 13, 27–29, 31, 40–42, 44, 181, 194
- Eclipse Paho MQTT** Paho is an open-source MQTT library from the Eclipse foundation. The Paho library exists in multiple programming languages such as Rust, Java and C. 27, 29–31, 43, 44
- EKS** Amazon Elastic Kubernetes Service is a managed Kubernetes cluster as a service provided by Amazon.[53]. see also Kubernetes
- "Exactly Once"** In "Exactly Once" delivery, the middleware only transfers the message once to the consumer. The message exchange should be reliable, as a retransmit is not possible if an acknowledgment is missed. To fulfill that, often a transactional message exchange is used.[12] However, "Exactly Once" is challenging to implement, if not even impossible. It depends on

the exact definition.[12, 54, 55] Mostly it is more economical and enough to handle "At Least Once".[54]. 16, 19–21, 28, 43, 44, 53

FaaS "Function as a service (FaaS) is a cloud computing model that enables users to develop applications and deploy functionalities without maintaining a server, increasing process efficiency. The concept behind FaaS is serverless computing and architecture, meaning the developer does not have to take server operations into consideration, as they are hosted externally. This is typically utilized when creating microservices such as web applications, data processors, chatbots and IT automation."[56]. 57

FreeRTOS FreeRTOS is a Real Time Operating System (RTOS) written in C distributed under the MIT license. The kernel binary is around 6KB to 12KB.[57]. see also RTOS

Git Git is an open-source distributed version control system. The tool was created in 2005 by the Linux community.[58]. 33, 54, 183, 194, 199, 200

GitHub GitHub is a source code management and collaboration platform based on Git. It is a well known platform for open-source code and was acquired in 2018 by Microsoft.[59]. 28–32, 37, 39, 52, 55, 56, 183, 199–201, see also Git

GNU 2 The GNU 2 or GNU General Public License v2 is a copyleft license of the Free Software Foundation, the latest version is version 3.[60]. 54, 55, 183, 202, see also GNU 3, copyleft & open-source

GNU 3 The GNU 3 or GNU General Public License v3 is a copyleft license of the Free Software Foundation, the latest version is version 3.[61]. 54, 55, 183, 202, see also GNU 2, copyleft & open-source

Go Go is a statically typed general purpose programming language.[62] The language was developed by Google and is now open-sourced under the BSD-style license.[63]. 51

GPIO A General Purpose Input/Output (GPIO) is a low-level interface that is used to connect microcontrollers to other electronic components.[64]. 40, 57

HTTP "The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers"[65]. 57

Industry 4.0 "The term "Industry 4.0" refers to the transformation of the manufacturing sector by digital technologies, such as the internet of things (IoT), artificial intelligence (AI), machine learning, robotics, 3-D printing, visualization, virtual/augmented reality and analytics."[66]. see also IoT

IoT "The internet of Things, or "IoT" for short, is about extending the power of the internet beyond computers and smartphones to a whole range of other things, processes and environments. Those "connected" things are used to gather information, send information back, or both."[67]. 54, 57

Java "Java is a high-level programming language developed by Sun Microsystems."[68] Java code is compiled to java bytecode and executed with a JVM. 29, 51, 52, 55, 57, 200, see also JVM & bytecode

JavaScript "JavaScript is a programming language commonly used in web development. It was originally developed by Netscape as a means to add dynamic and interactive elements to websites."[69] It is a scripting language based on the ECMAScript standard.[70]. 11, 55, 57

- JDK** The Java Development Kit is a combination of the Java runtime, the compiler and other tooling that is needed for the development of Java applications.[71]. 57
- JSON** "JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON."[72]. 57
- JVM** The Java Virtual Machine is a virtual machine running a Java program. It implements a concrete commands for the abstract Java specification. It executes the Java bytecode and manages the memory.[73]. 57, see also Java
- Kafka** Apache Kafka is a stream processing software and a message broker. Kafka provides queuing and publish-subscribe.[74]. 20
- Knative** Knative is a FaaS platform on top of Kubernetes.. see also Kubernetes & FaaS
- Knative Eventing** Knative Eventing is a set of loosely coupled services that route CloudEvents from producers to interested consumers. It is tightly integrated into Kubernetes.[75]. see also Knative & Kubernetes
- Kubernetes** Kubernetes is an open-source container orchestration platform hosted by CNCF.[76]. 37, 38, 53–55, 181, 183
- LaTeX** "LaTeX, which is pronounced «Lah-tech» or «Lay-tech» (to rhyme with «blech» or «Bertolt Brecht»), is a document preparation system for high-quality typesetting. It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing."[77]
LaTeX is distributed under the LaTeX project public license. It is a free software license.[78]. 51, 200
- Linux** Linux is a operating system written in C with a microkernel architecture and licenced under the GNU 2 license.[41, 79, 80]. 13, 14, 31, 53, 56, 183
- Lucidchart** Lucidchart is a graphical online tool to draw diagrams, processes, and systems.[81]. 200
- MDSL** Microservice Domain-Specific Language (MDSL) is a DSL to describe the interactions between systems. The language describes APIs as contracts in a vendor and protocol neutral way. The language was first proposed by O. Zimmermann and is tightly coupled with the Microservice API Patterns.[82, 83]. see also asyncMDSL & DSL
- Microkernel** "The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration."[5]. 13, 14, 23, 24, 26, 27, 43, 193, 194
- Minikube** Minikube is a tool to easily install Kubernetes on a personal computer.[84]. 183
- MIT license** The MIT License is a open-source license originally created by the Massachusetts Institute of Technology and documented by the OSI.[85]. 53, 55, see also OSI & open-source
- MQTT** MQTT is a lightweight publish/subscribe messaging protocol for machine to machine communication.[86, 87]. 57
- MVP** A product is at the state of a Minimum Viable Product when it gives the user an additional value, but as little extra feature as possible to get a good feedback loop for additional features[88]. 57

NATS NATS is a hyper-scalable messaging system hosted by the CNCF. Its design goal is to provide globally deployable, multi-tenant messaging infrastructure that supports publish/subscribe and request/reply messaging semantics. NATS was originally built by Synadia for their global communications system (NGS) but then donated to the CNCF.[89, 90]. 20, 51, see also CNCF

Node.js Node.js is an open-source JavaScript runtime build on top of the Google Chrome V8 JavaScript engine.[91]. see also JavaScript

open-source "Generally, open-source software is software that can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone. Open-source software is made by many people, and distributed under licenses that comply with the open-source definition. The internationally recognized open-source Definition provides 10 criteria that must be met for any software license, and the software distributed under that license, to be labeled 'open-source software.' "[92]

GNU 2, GNU 3, Apache 2, Academic Free License and MIT license are all open-source licenses which have been approved by OSI. 3, 31, 51, 53–55, 199, see also OSI

OSI OSI is a California public non-profit corporation with the goal of educate about open-source.[93]. 58, see also open-source

"ownership model" The "ownership model" is the most unique feature of Rust. In Rust memory is allocated and freed explicit, but in contrast to C or other low level programming languages without the need of an garbage collector, Rust enforces a set of rules on this actions, to ensure memory safety."[94]. 56, see also Rust

PlantUML PlantUML is a tool to generate different Unified Modeling Language (UML) and non UML diagrams. The diagrams are generated by writing with the PlantUML Language. The tool is written in Java and has various output formats, one of the im JPGE.[95]. 200, see also UML

Port (Plugin) This is a plugin type of the CloudEvents Router. The Port is responsible for exchanging CloudEvents with the outside world.. 13–15, 27, 30–32, 41, see also Microkernel, CloudEvents, CloudEvents Router & CERK

pull request Pull requests are a feature of GitHub to disusse the differences between two branches with the goal to merge the changes of one branch into the other branch.[96]. 15, 16, 20, 28, 30–32, 44, 200, see also Git & GitHub

Python Python is a high-level general purpose programming language; It is used as scripting language, for Web application and many other things[97]. 13, 29, 51

QoS Quality of Service is defined in the term of telecommunication as "Totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service."[98]. 58

R R is a programming language for statistical compuations. It is licenced under the GNU 2 and GNU 3 language.[99, 100]. 183, 202

RTOS Real Time Operating Systems are a type of operating systems where "The scheduler ... is designed to provide a predictable (normally described as deterministic) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline). A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted (and is therefore deterministic)."[101]. 58

- Rust** Rust is a low-level programming language with guaranteed thread and memory safety. It achieves that with its unique "ownership model". The goal is to build safe and reliable software.[94, 102] Rust is used by Mozilla Firefox, Dropbox and many others.[103, 104] In 2019, Rust has also been elected in the well-known survey of Stack Overflow for being the "most loved" programming language.[105]. 5, 12, 13, 16, 27–33, 39, 44, 52, 55, 56, 193, 194, 199–201, see also "ownership model" & Stack Overflow
- rustc** rustc is the Rust compiler provided by the Rust Team. It can be used directly (`rustc`) or with Cargo (`cargo build`).[106]. 201, see also Rust & Cargo
- rustup** rustup is a tool to install different versions of Rust, like stable, beta and nightly, and for different compilation targets. It is the recommended way for developers to install Rust.[107]. 201, see also Rust & Cargo
- SaaS** Software as a service (SaaS) is a software distribution model in which a third-party provider hosts applications and makes them available to customers over the Internet.[108]. 58
- SDK** Software Development Kits are libraries which ease the access to APIs. 58, see also API
- semaphore** A semaphore is a kind of variable that is used to synchronize processes.[109]. 30
- Serverless Framework** The Serverless Framework is a FaaS abstraction layer.. 56, see also FaaS
- SQS** Amazon Simple Queue Service (SQS) is a fully managed message queuing service from Amazon with first-in, first-out (FIFO) support[110]. 58
- Stack Overflow** Stack Overflow is primary a Q&A forum for coding related problems.. 56
- STOMP** A Simple Text Oriented Message Protocol, designed for asynchronous message passing[111]. 58
- T-Metric** T-Metric is a time tracking app with a functionally limited free plan for up to 5 people. It has integrations for various products, such as GitHub.[112]. 199
- TCP** The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.[113]. 58
- UML** UML is a standard for diagramming notation. It was created by Booch and Rumbaugh in 1994.[114]. 55, 58, 200
- UNIX** UNIX a general-purpose, mutli-user operation system. It is written in C and is the root of the Linux operating system.[41, 115]. 27, 39–41
- UNIX socket** "The `AF_UNIX` socket family is used to communicate between processes on the same machine efficiently."[116] . 14, 56, see also UNIX
- WAMP** "WAMP is a routed protocol that provides two messaging patterns: Publish & Subscribe and routed Remote Procedure Calls. It is intended to connect application components in distributed applications. WAMP uses WebSocket as its default transport, but can be transmitted via any other protocol that allows for ordered, reliable, bi-directional, and message-oriented communications."[117]. 58
- WebHook** WebHooks are a popular pattern to deliver events to HTTP-enabled services. Unfortunately, there is no formal specification for WebHooks, and most services implement their own flavour of it.[118, 119]. 20, see also HTTP

Acronyms

ADL architecture description language. 19

ADR ARD (Architecture Decision Record). [Glossary](#) ADR

AMQP Advanced Message Queuing Protocol[27]. 20, 27, 35, 37, 43, 183, [Glossary](#) AMQP

AMSI American National Standards Institute. 52

API Application Programming Interface. 19, 44, 51, 55, 194, [Glossary](#) API

AWS Amazon Web Services[38]. 13, 41, 51, 52, 201, [Glossary](#) AWS

CERK CloudEvents Router with a MicroKernel. 13, 183

CI Continuous Integration. 28, 199–201, [Glossary](#) CI

CNCF Cloud Native Computing Foundation . 11, 15, 44, 53, 54, [Glossary](#) CNCF

CPU central processing unit. 40, 52, 194

DSL Domain Specific Language. [Glossary](#) DSL

EC2 Elastic Compute Cloud. 13, 57, [Glossary](#) Amazon Elastic Compute Cloud

FaaS Function as a Service. 51, 54, 56, [Glossary](#) FaaS

GB Megabytes, equal to 1000 Megabytes. 183

GPIO General Purpose Input/Output. 40, 41, 193–195, [Glossary](#) GPIO

HTTP HyperText Transfer Protocol. 20, [Glossary](#) HTTP

ID identifier. 37, 181, 183

IDE Integrated Development Environment. 199, 201, 202

IoT Internet of Things. 11, 51, [Glossary](#) IoT

ISO International Organization for Standardization. 52

JDK Java Development Kit. [Glossary](#) JDK

JSON JavaScript Object Notation. 11, 32, 33, 49, 195, [Glossary](#) JSON

JVM Java Virtual Machine. 52, 54, [Glossary](#) JVM

KB Kilobytes, equal to 1000 Bytes. 53

MDSL Microservice Domain-Specific Language. 20, 52, [Glossary](#) MDSL

MOM message-oriented middleware. 24, 26, 31, 37, 38, 181, 183

MQTT Message Queuing Telemetry Transport[87]. 12–14, 16, 20, 27–31, 35, 37, 40, 41, 43, 44, 51, 53, 183, 194, 195, [Glossary MQTT](#)

MVP Minimum Viable Product[88]. [Glossary MVP](#)

OASIS Organization for the Advancement of Structured Information Standards. [Glossary OASIS](#)

OSI Open Source Initiative. 51, 55, [Glossary OSI](#)

QoS Quality of Service. 16, 28, 29, 31, [Glossary QoS](#)

RTOS Real Time Operating System. 53, [Glossary RTOS](#)

SaaS Software as a Service. 11, [Glossary SaaS](#)

SDK Software Development Kit. 16, 27, [Glossary SDK](#)

SQS Simple Queue Service[110] . 13, 14, [Glossary SQS](#)

STOMP Simple Text Oriented Messaging Protocol[111]. [Glossary STOMP](#)

TCP Transmission Control Protocol[113]. [Glossary TCP](#)

TTL time to live. 26

UML Unified Modeling Language. 55, 200, [Glossary UML](#)

VM Virtual Machine. 51, 183

WAMP Web Application Messaging Protocol[117]. [Glossary WAMP](#)

Bibliography

- [1] L. Basig and F. Lazzaretti, CloudEvents Router, HSR, 2020.
[Online]. Available: <https://eprints.hsr.ch/832/>.
- [2] CARU | Das intelligente Notrufsystem, 2020.
[Online]. Available: <https://www.caruhome.com/>.
- [3] CloudEvents | A specification for describing event data in a common way.
[Online]. Available: <https://cloudevents.io/>.
- [4] CloudEvents Specification v1.0, GitHub, 2019.
[Online]. Available: <https://github.com/cloudevents/spec/tree/v1.0>.
- [5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-oriented software architecture. Jul. 12, 1996, 476 pp., ISBN: 0471958697.
- [6] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, MQTT Version 5.0, OASIS Standard Incorporating, 2019.
[Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>.
- [7] Z. Ming and M. Yan, A modeling and computational method for QoS in IOT, in 2012 IEEE International Conference on Computer Science and Automation Engineering, IEEE, 2012. DOI: 10.1109/icsess.2012.6269459.
- [8] ISO/IEC 25010. [Online]. Available: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&start=3>.
- [9] ISO/IEC 25010:2011: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.
[Online]. Available: <https://www.iso.org/standard/35733.html>.
- [10] L. Basig and F. Lazzaretti, Reliable Messaging – Patterns and Strategies for Message Routing, research rep., 2021.
- [11] G. Hohpe and B. Woolf, Enterprise integration patterns. Addison Wesley, Jan. 1, 2004, 480 pp., ISBN: 0321200683.
- [12] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, Cloud computing patterns. Springer Publishing Company, Incorporated, Feb. 18, 2014, ISBN: 978-3-7091-1567-1.
- [13] L. Basig and F. Lazzaretti, Protocol Interoperability in a Stateless Message Router, 2021.
- [14] The cargo book. 2020. [Online]. Available: <https://doc.rust-lang.org/cargo/index.html>.
- [15] N. T. Blog, The Netflix Simian Army, Medium,
[Online]. Available: <https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116>.
- [16] Why Async? 2019. [Online]. Available: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html.
- [17] Async-await hits beta!
[Online]. Available: <https://blog.rust-lang.org/2019/09/30/Async-await-hits-beta.html>.
- [18] N. Matsakis, Async-await on stable Rust! 2019.
[Online]. Available: <https://blog.rust-lang.org/2019/11/07/Async-await-stable.html>.
- [19] AWS IoT Core. [Online]. Available: <https://aws.amazon.com/iot-core/>.
- [20] CloudSubscriptions: Subscriptions API, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0.1/subscriptions-api.md>.

- [21] NATS Streaming Introduction, 2020.
[Online]. Available: <https://docs.nats.io/nats-streaming-concepts/intro>.
- [22] AWS Lambda – Serverless Compute – Amazon Web Services, 2021.
[Online]. Available: <https://aws.amazon.com/lambda/>.
- [23] Academic Free License ("AFL") v. 3.0 | Open Source Initiative.
[Online]. Available: <https://opensource.org/licenses/AFL-3.0>.
- [24] T. Woods, 'Age-Tech': The Next Frontier Market For Technology Disruption, Forbes, 2019.
[Online]. Available: <https://www.forbes.com/sites/tinawoods/2019/02/01/age-tech-the-next-frontier-market-for-technology-disruption/#567c4dea6c84>.
- [25] Amazon EC2, 2021. [Online]. Available: <https://aws.amazon.com/ec2>.
- [26] 1.3. AMQP - Advanced Message Queuing Protocol Red Hat Enterprise MRG 3, 2019.
[Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_mrg/3/html/messaging_programming_reference/amqp__advanced_message_queuing_protocol.
- [27] R. Godfrey, D. Ingham, and R. Schloming, OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, OASIS, 2012.
[Online]. Available: <https://www.oasis-open.org/standards#amqp1.0>.
- [28] Apache License, Version 2.0, 2019.
[Online]. Available: <https://www.apache.org/licenses/LICENSE-2.0>.
- [29] P. Eising, What exactly IS an API? - Perry Eising - Medium, Medium, 2017.
[Online]. Available: <https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f>.
- [30] arc42 - arc42. [Online]. Available: <https://arc42.org/>.
- [31] Download arc42 - arc42. [Online]. Available: <https://arc42.org/download>.
- [32] arc42 License - arc42. [Online]. Available: <https://arc42.org/license>.
- [33] AsyncAPI, 2020. [Online]. Available: <https://www.asyncapi.com/docs/getting-started/>.
- [34] G. De Liberali, AsyncMDSL: a domain-specific language for modeling message-based systems, Master's thesis, Università di Pisa, 2020.
[Online]. Available: <https://etd.adm.unipi.it/t/etd-06222020-100504/>.
- [35] AsyncMDSL extension (DSL and Eclipse plugin), 2020.
[Online]. Available: <https://github.com/Microservice-API-Patterns/MDSL-Specification/tree/3cf15b9c866c1086da7e8cd354b2e991055f8d3d/examples/asyncMDSL>.
- [36] G. Hohpe and B. Woolf, Fire-and-Forget, 2017.
- [37] —, Conversation Patterns, 2017. [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html>.
- [38] What is AWS. [Online]. Available: <https://aws.amazon.com/what-is-aws/>.
- [39] Buildkite Features, 2019. [Online]. Available: <https://buildkite.com/features>.
- [40] Knative Serving License. [Online]. Available: <https://techterms.com/definition/bytecode>.
- [41] B. Stroustrup, Programming. Addison Wesley, May 15, 2014, 1312 pp., ISBN: 0321992784.
- [42] H. Patel, Python - C++ bindings | Hardik Patel, 2017.
[Online]. Available: <https://www.hardikp.com/2017/12/30/python-cpp/>.
- [43] Continuous integration | ThoughtWorks, 2019.
[Online]. Available: <https://www.thoughtworks.com/continuous-integration>.
- [44] CNCF Cloud Native Definition v1.0, 2019.
[Online]. Available: <https://github.com/cncf/toc/blob/master/DEFINITION.md>.
- [45] CloudSubscriptions: Discovery, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0.1/discovery.md>.

- [46] [Online]. Available: <https://www.cncf.io/cncf-kubernetes-project-journey/>.
- [47] CNCF Prometheus Project Journey - Cloud Native Computing Foundation, 2019. [Online]. Available: <https://www.cncf.io/cncf-prometheus-project-journey/>.
- [48] Members - Cloud Native Computing Foundation. [Online]. Available: <https://www.cncf.io/about/members/>.
- [49] What is Copyleft? [Online]. Available: <https://www.gnu.org/licenses/copyleft.en.html>.
- [50] What is Docker? 2018. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined>.
- [51] S. Efftinge, M. Volter, A. Haase, and B. Kolb, The Pragmatic Code Generator Programmer, 2006. [Online]. Available: <https://www.theserverside.com/news/1365073/The-Pragmatic-Code-Generator-Programmer>.
- [52] Eclipse Mosquitto - An open source MQTT broker. [Online]. Available: <https://mosquitto.org/>.
- [53] Amazon Elastic Kubernetes Service. [Online]. Available: <https://aws.amazon.com/eks/>.
- [54] T. Treat, You Cannot Have Exactly-Once Delivery, 2015. [Online]. Available: <https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/>.
- [55] M. Steen and A. Tanenbaum, Distributed systems. Place of publication not identified: Maarten van Steen, 2017, ISBN: 9781543057386.
- [56] M. Rouse, What is function as a service (FaaS)? - Definition from WhatIs.com, TechTarget, 2018.
- [57] FreeRTOS. [Online]. Available: <https://www.freertos.org/>.
- [58] S. Chacon and B. Straub, Pro git. Apress, 2014.
- [59] K. Johnson, GitHub passes 100 million repositories, 2018. [Online]. Available: <https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/>.
- [60] GNU General Public License v2.0 - GNU Project - Free Software Foundation, 2017. [Online]. Available: <https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html>.
- [61] The GNU General Public License v3.0 - GNU Project - Free Software Foundation, 2016. [Online]. Available: <https://www.gnu.org/licenses/gpl-3.0.html>.
- [62] C. Doxsey, An introduction to programming in go. 2012, ISBN: 978-1478355823. [Online]. Available: <http://www.golang-book.com/books/intro>.
- [63] The Go Project - The Go Programming Language, 2019. [Online]. Available: <https://golang.org/project/>.
- [64] What is GPIO? 2021. [Online]. Available: <https://community.estimote.com/hc/en-us/articles/217429867-What-is-GPIO->.
- [65] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999. doi: 10.17487/RFC2616. [Online]. Available: <https://rfc-editor.org/rfc/rfc2616.txt>.
- [66] P. Mukhopadhyay, Sales Transformations For "Industry 4.0", Forbes, 2019. [Online]. Available: <https://www.forbes.com/sites/forbestechcouncil/2019/10/16/sales-transformations-for-industry-4-0/#23aafb725a42>.
- [67] C. McClelland, What Is IoT? - A Simple Explanation of the Internet of Things, IoT For All, 2019. [Online]. Available: <https://www.iotforall.com/what-is-iot-simple-explanation/>.
- [68] Java Definition, 2012. [Online]. Available: <https://techterms.com/definition/java>.
- [69] JavaScript Definition, 2014. [Online]. Available: <https://techterms.com/definition/javascript>.

- [70] ECMAScript® 2019 Language Specification, Ecma International.
[Online]. Available: <https://www.ecma-international.org/ecma-262/10.0/index.html>.
- [71] JDK Definition from PC Magazine Encyclopedia.
[Online]. Available: <https://www.pcmag.com/encyclopedia/term/45608/jdk>.
- [72] JSON - JavaScript | MDN, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.
- [73] B. Venners, Inside the java virtual machine. McGraw Hill, 1999, ISBN: 0-07-135093-4.
- [74] What is Apache Kafka? [Online]. Available: <https://aws.amazon.com/msk/what-is-kafka/>.
- [75] Knative Eventing, GitHub. [Online]. Available: <https://github.com/knative/eventing/>.
- [76] CNCF Kubernetes Project Journey.
[Online]. Available: <https://www.cncf.io/cncf-kubernetes-project-journey/>.
- [77] Introduction to LaTeX. [Online]. Available: <https://www.latex-project.org/about/>.
- [78] The LaTeX project public license. [Online]. Available: <https://www.latex-project.org/lppl/>.
- [79] Linux from FOLDOC, 2000. [Online]. Available: <http://foldoc.org/linux>.
- [80] The Linux Kernel Archives - FAQ.
[Online]. Available: <https://www.kernel.org/category/faq.html>.
- [81] Learn About Lucid Software, 2021. [Online]. Available: <https://lucid.co/about>.
- [82] Microservice DSL (MDSL), 2020.
[Online]. Available: <https://microservice-api-patterns.github.io/MDSL-Specification/>.
- [83] Microservice API Patterns, 2020. [Online]. Available: <https://microservice-api-patterns.org/>.
- [84] Install Tools, 2020. [Online]. Available: <https://kubernetes.io/docs/tasks/tools/>.
- [85] G. Haff, The mysterious history of the MIT License | Opensource.com, 2019.
[Online]. Available: <https://opensource.com/article/19/4/history-mit-license>.
- [86] MQTT Homepage. [Online]. Available: <http://mqtt.org/>.
- [87] A. Banks and R. Gupta, MQTT Version 3.1.1 Plus Errata 01, OASIS Standard Incorporating, 2015. [Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html>.
- [88] Minimum Viable Product (MVP), [Online]. Available: <https://www.techopedia.com/definition/27809/minimum-viable-product-mvp>.
- [89] NATS Introduction, 2020. [Online]. Available: <https://docs.nats.io/>.
- [90] Synadia - Connect Everything, 2020. [Online]. Available: <https://synadia.com/>.
- [91] Introduction to Node.js. [Online]. Available: <https://nodejs.dev/>.
- [92] Frequently Answered Questions: What is "Open Source" software?
[Online]. Available: <https://opensource.org/faq#osd>.
- [93] About the Open Source Initiative | Open Source Initiative.
[Online]. Available: <https://opensource.org/about>.
- [94] C. N. Steve Klabnik, The rust programming language. Random House LCC US, Aug. 12, 2019, ISBN: 1718500440.
- [95] Drawing UML with PlantUML, PlantUML Language Reference Guide, 2019.
[Online]. Available: <http://plantuml.com/guide>.
- [96] About pull requests. [Online]. Available: <https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests>.
- [97] D. Kuhlman, A python book: Beginning python, advanced python, and python exercises, 1.1a. 2012. [Online]. Available: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html.

- [98] E.800 : Definitions of terms related to quality of service, International Telecommunication Union (ITU), 2008.
[Online]. Available: <https://www.itu.int/rec/T-REC-E.800-200809-I/en>.
- [99] What is R? [Online]. Available: <https://www.r-project.org/Licenses/>.
- [100] R Licenses. [Online]. Available: <https://www.r-project.org/about.html>.
- [101] Why RTOS and What is RTOS?
[Online]. Available: <https://www.freertos.org/about-RTOS.html>.
- [102] D. Bryant, A Quantum Leap for the Web, Medium, 2016. [Online]. Available: <https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12>.
- [103] Rust. [Online]. Available: <https://research.mozilla.org/rust/>.
- [104] Production users. [Online]. Available: <https://www.rust-lang.org/production/users>.
- [105] Stack Overflow Developer Survey 2019, Tech. Rep., 2019.
[Online]. Available: <https://insights.stackoverflow.com/survey/2019>.
- [106] The rustc book. [Online]. Available: <https://doc.rust-lang.org/rustc/what-is-rustc.html>.
- [107] [Online]. Available: <https://www.rust-lang.org/tools/install>.
- [108] M. Rouse, What is Software as a Service (SaaS)? - Definition from WhatIs.com, TechTarget, 2019.
- [109] E. Glatz, Betriebssysteme: Grundlagen, konzepte, systemprogrammierung. Place of publication not identified: Bookwire GmbH, 2015, ISBN: 9783864902222.
- [110] Amazon Simple Queue Service. [Online]. Available: <https://aws.amazon.com/sqs/>.
- [111] STOMP Protocol Specification, Version 1.2, 2012.
- [112] Free Time Tracking Software & App - TMetric. [Online]. Available: <https://tmetric.com/>.
- [113] J. Postel, Transmission Control Protocol, RFC 793, 1981.
DOI: 10.17487/RFC0793. [Online]. Available: <https://rfc-editor.org/rfc/rfc793.txt>.
- [114] C. Larman, Applying uml and patterns. Prentice Hall, 2004, vol. 3, ISBN: 9780131489066.
- [115] D. Ritchie and K. Thompson, The UNIX Time-Sharing System, 1978.
[Online]. Available: <https://archive.org/details/bstj57-6-1905>.
- [116] unix(7) - Linux manual page.
[Online]. Available: <http://man7.org/linux/man-pages/man7/unix.7.html>.
- [117] T. Oberstein and A. Goedde, The Web Application Messaging Protocol, IETF, 2019.
[Online]. Available: https://wamp-proto.org/_static/gen/wamp_latest_ietf.html#rfc.authors.
- [118] HTTP 1.1 Web Hooks for Event Delivery, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/master/http-webhook.md>.
- [119] REST Hooks, 2020. [Online]. Available: <https://resthooks.org/>.
- [120] G. M. Poore, 2017.
[Online]. Available: <http://mirrors.ctan.org/macros/latex/contrib/minted/minted.pdf>.
- [121] N. Friedman, GitHub Actions now supports CI/CD, free for public repositories, GitHub Blog, 2019.
[Online]. Available: <https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/>.
- [122] Rust support for Visual Studio Code, GitHub, 2019.
[Online]. Available: <https://github.com/rust-lang/rls-vscode>.

A. Problem Statement

Aufgabenstellung Bachelorarbeit

Linus Basig, Fabrizio Lazzaretti

Reliable Messaging Using the CloudEvents Router

1. Auftraggeber und Betreuer

Diese Studienarbeit wird in Zusammenarbeit mit CARU AG durchgeführt.

Ansprechpartner CARU AG:

Dr. Thomas Helbling thomas.helbling@caruhome.com (administrative Fragen)

Reto Aebersold reto.aebersold@caruhome.com (technische Fragen)

Betreuer (HSR/OST):

Prof. Dr. Olaf Zimmermann, Institut für Software, olaf.zimmermann@ost.ch

2. Ausgangslage

CARU AG is an AgeTech company that enables the elderly to live independently through technology. It provides security and social integration without being intrusive. CARU's product is composed of an IoT device, which collects sensor data and provides a simple user interface (touch and voice), and a web page on which more services are provided. CARU leverages a highly event-driven architecture that uses CloudEvents, a Specification from the Cloud Native Computing Foundation (CNCF), for the events produced and consumed by its systems.

In a preceding project ("Studienarbeit"), the students designed and implemented a CloudEvents Router that provides seamless information exchange between subsystems. It routes events that are formatted as CloudEvents between ports, which represent the binding to messaging protocols. The routing leverages the content of the fields defined by the CloudEvents Specification.

The CloudEvent Router currently supports MQTT and has a plugin system that allows adding additional messaging protocols like HTTP, AMQP, NATS, and Kafka. These messaging protocols offer different delivery guarantees. Currently, the CloudEvents Router does not respect the delivery guarantees of the messaging protocols it talks to. However, to be entrusted with critical events, the CloudEvents Router must respect the delivery guarantees of the connected messaging protocols by preserving the protocol-specific delivery guarantees when routing events from one messaging protocol to another. AsyncAPI (<https://www.asyncapi.com/>) is an emerging industry standard for defining asynchronous APIs that abstracts from protocol specifics.

3. Ziele der Arbeit und Liefergegenstände (Goals and Deliverables)

CARU would like to address the above problem in this bachelor thesis project. The delivery guarantees of the most relevant messaging protocols and their interoperability shall be analyzed. Next, a design that enables the CloudEvents Router to preserve the delivery guarantees when routing

messages shall be proposed. And finally, a proof of concept that demonstrates the design in action shall be implemented.

Deliverables. The expected deliverables are:

- Analysis of the description of the delivery guarantees in AsyncAPI, including implications on the envisioned design proposal
- Analysis and comparison of the delivery guarantees provided by the messaging protocols that are most relevant for CARU (including MQTT, HTTP, AMQP and NATS).
- (optional) Analysis of existing products (Apache Camel, Apache Kafka, RabbitMQ, Apache Mosquitto, etc) w.r.t. their capabilities to express and provide messaging reliability
- A sound design proposal for the implementation of advanced delivery guarantees between different messaging protocols (i.e., those exceeding the best-effort capability), possibly leveraging AsyncAPI concepts
- A minimal implementation of the preservation of extended delivery guarantees in the CloudEvents Router with at least one of the above protocols
- (optional) Analysis of the description of the delivery guarantees in AsyncMDSL, an emerging service contract DSL that embraces the Enterprise Integration Patterns by Hohpe/Woolf (which may have implications on the above design proposal, and yield feedback to the language designers)
- (optional) Analysis of related CloudEvents Discovery API capabilities

Critical Success Factors. The following quality criteria are particularly relevant for this thesis:

- Feature set as well as usability, quality of documentation and ability to integrate
- Maintainability, extensibility and ability to operate the solution
- Software engineering hygiene: code quality as well as version control, suited automated tests and builds

4. Unterstützung

Die erwartete und effektiv erhaltene Unterstützung wird durch die Studierenden protokolliert.

5. Zur Durchführung

Mit dem Betreuer finden in der Regel wöchentlich Besprechungen statt (Treffen an der OST oder Telefon- bzw. Webkonferenz). Zusätzliche Besprechungen sind nach Bedarf zu veranlassen.

Alle Besprechungen, bei denen eine Vorbereitung durch den Betreuer nötig ist, sind von den Studierenden mit einer Traktandenliste vorzubereiten. Beschlüsse sind in einem Protokoll zu dokumentieren.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. Arbeitszeiten sind zu dokumentieren.

Die Spezifikation der Anforderungen geschieht durch die Studierenden in Absprache mit dem Betreuer. Bei Disputen entscheidet der Betreuer in Rücksprache mit den Studierenden und dem Auftraggeber über die definitiv für die Bachelorarbeit relevanten Anforderungen.

Vorstudie, Anforderungsdokumentation und Architekturdokumentation sollten im Laufe des Projektes mittels Milestones mit dem Auftraggeber und dem Betreuer in einem stabilen Zustand abgenommen werden. Zu den abgegebenen Arbeitsergebnissen wird ein vorläufiges Feedback abgegeben. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

Die Rechte an den Ergebnissen der Bachelorarbeit werden in einer separaten Vereinbarung definiert (Bericht öffentlich).

6. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

Bei der Projektdokumentation und Ihrer Abgabe sind die „Allgemeine Informationen zu Studien- und Bachelorarbeiten“ sowie die „Anleitung: Dokumentation Studien- und Bachelorarbeiten“ inklusive Anhängen zu beachten.

7. Termine

Siehe HSR/OST-Webseiten. Allfällige weitere Termine sind am Sekretariat der Abteilung Informatik zu erfragen und sollten entsprechend in einem Sitzungsprotokoll dokumentiert werden.

8. Beurteilung

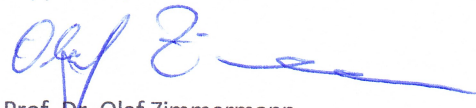
Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS-Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert. Siehe auch Modulbeschreibung der Bachelorarbeiten, http://studien.hsr.ch/allModules/24809_M_BAI14.html.

Gesichtspunkt	Gewicht
1. Organisation, Durchführung	1/6
2. Berichte (Abstract, Management Summary, technische u. persönliche Berichte) sowie Gliederung, Darstellung und Sprache der gesamten Dokumentation.	1/6
3. Inhalt*)	3/6
4. Mündliche Prüfung zur Bachelorarbeit	1/6

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt.

Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Bachelorarbeiten.

Rapperswil, 16. 09. 2020



Prof. Dr. Olaf Zimmermann

Institut für Software

University of Applied Sciences of Eastern Switzerland (OST)

B. Reliable Messaging – Patterns and Strategies for Message Routing

Reliable Messaging – Patterns and Strategies for Message Routing

Linus Basig, Fabrizio Lazzaretti
Advisor: Prof. Dr. Olaf Zimmermann

Department of Computer Science
OST – University of Applied Sciences of Eastern Switzerland
Campus Rapperswil-Jona

Abstract

Many distributed systems use asynchronous messaging patterns to achieve decoupling, scaling, and reliability. An important issue that must be addressed in order to utilize asynchronous messaging for mission-critical workloads is reliability.

In this paper, we present an overview of existing patterns and strategies for reliable messaging and how they apply to the problem of routing messages reliably in a protocol-independent and stateless way. We analyze existing messaging patterns from different sources with a focus on the books "Enterprise Integration Patterns" by Hohpe et al. and "Cloud Computing Patterns" by Fehling et al.

We propose a simple approach to define one delivery guarantee per channel without any additions, such as a specific sender or receiver option regarding the delivery guarantee.

The protocol-independent delivery guarantees are named "At Least Once", "At Most Once" and "Exactly Once".

Then, we discuss how AsyncApi, AsyncMDSL, and the CloudSubscriptions Discovery API could integrate these delivery guarantees to add a clear and simple way to describe the delivery guarantee that is used to transmit messages.

Finally, we propose a platform-independent conceptual design and demonstrate the usage in a stateless router that routes messages reliably. We show that the overhead from routing with an "Exactly Once" guarantee can be removed from the router by transferring this task to a message filter.

The proposal for AsyncMDSL was already implemented and is publicly available.

Keywords: messaging, quality of service, reliability, distributed systems

Contents

1	Introduction	1
2	Reliable Messaging Patterns	2
3	Evolution of the Patterns	6
4	Defining Reliability	6
5	Proposal for a Router Implementation	9
6	Related Work	11
7	Discussion	14
8	Conclusion	14

1 Introduction

In the world of distributed systems, asynchronous messaging patterns are a common tool to achieve decoupling, scaling, and reliability. Asynchronous messaging has the advantage over synchronous request methods that many parallel requests can be buffered by a message-oriented middleware (MOM) (e.g., a message broker, or a queue). However, asynchronous messaging has the draw-back that the caller is not waiting for a response and cannot directly react to a lost connection issue as it would be possible with a synchronous call. This is why reliability, the ability to be sure that a message arrives at its destination, is extremely important in asynchronous messaging.[1]

This work aims to explore how an asynchronous message communication between two or multiple partners could be made more reliable in a protocol-independent way. For this purpose, we analyze different patterns, put them into context, and evaluate them. Most used patterns are from the books "Enterprise Integration Patterns" and "Cloud Computing Patterns". In a second step, we look at the insights from the patterns in different Domain Specific Languages (DSLs) and investigate whether reliable messaging is contained within them or how they need to be adapted in order to implement them. Finally, we try to create a protocol independent design for a stateless router to route messages reliably. This should be done on the specification and the implementation side, with a concept for a router implementation, and we are therefore not looking into replication or fail-over scenarios.

This paper was written in the context of the bachelor thesis "Reliable Messaging using the CloudEvents Router" that discusses the design of a Message Router specialized in routing events structured according to the CloudEvents specification.[4, 5] The CloudEvents specification is a standard from the Cloud Native Computing Foundation (CNCF) with the goal to "describe event data in common formats to provide interoperability across services, platforms and systems." [5] The standard reached version 1.0 on the 24th of October 2019.

The Message Router pattern describes "a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel, depending on a set of conditions." [2]

Before we dive into reliability we are taking a look at how reliability is defined in the context of software engineering in general and in the context of messaging specifically.

Definition of Reliability in Software Engineering

"ISO/IEC 25010:2011: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models" is a standard that defines quality characteristics for software engineering and systems. They define eight characteristics and some sub-characteristics for product quality. The goal is to define a consistent terminology and to establish a complete set of attributes for comparison.

Reliability is one of the eight system/software product quality cornerstones of ISO 25010. Availability is a subcharacteristic of reliability and reflects the degree to which a system is operational and accessible. Meaning, how often is a users' request processed within the defined time frame. Reliability requires that an action is completed as intended, while availability just requires that the system reacts to the request but does not necessarily has to provide the intended result.[6–8]

Definition of Reliable Messaging

In messaging, reliability is also often defined as part of the non-functional requirements. It can also be expressed with the term Quality of Service (QoS), as is used in Message Queuing Telemetry Transport (MQTT).[9]

Other sources describe QoS with the performance, dependability (which describes reliability and availability), scalability, capacity, robustness and other criteria.[7, 10]. Ming and Yan also add price and reputation as subjective QoS metrics.[7]

There are multiple publications on QoS classification for webservices; they mainly use performance, dependability (reliability and availability), scalability, capacity. However, there is currently no broadly accepted standard.[11]

There is no strict definition of QoS. A general one is provided by "ISO/IEC 13236:1998(E): Information technology – Quality of service: Framework". The standard recommends a definition for the scope of "Open distributed processing" as "A set of qualities related to the collective behavior of one or more objects".[12]

In this paper, we would like to describe reliable messaging with patterns and define a notation for the different levels of reliability. We will only focus on reliability and no other proposed properties of QoS. It is, however, important to keep in mind that other above-mentioned properties of QoS implicitly impact performance and robustness of the software.

We also only define reliability in an abstract way and never use concrete numbers. For the purpose of this investigation, we shall not be considering the performance of individual processes, e.g., the reliability of the used storage, but rather conceptually analyze the underlying system.

One advantage in messaging over other similar resource based technologies is the fact that messages are by definition atomic. This means that it is impossible for an incomplete message to be received and the message is never in an eventually persisted state.[2]

Overview

We start our paper by discussing common patterns and concepts for reliable messaging.

With this knowledge, we present a proposal to describe reliability for messaging in a protocol-independent manner. Next, we analyze different DSLs that are designed to describe messaging contracts. We examine their options to describe reliable messaging and show how to extend these languages if the current language syntax lacks the ability to describe reliable messaging options as needed.

Then we show how the presented concepts can be used to implement a reliable event router.

Lastly, we discuss related work, do a critical review of our proposals, and draw our conclusion.

2 Reliable Messaging Patterns

In this section, we are looking at common messaging patterns and concepts to achieve reliable messaging.

A Message Channel is a central element of every asynchronous messaging system. It is used to exchange messages: A number of applications put data on a channel and a number of applications take the messages from the channel. Mostly channels are placed on MOMs. There are two kinds of channels: point-to-point channels and publish-subscribe channels.[2] Sometimes a channel is also called a queue.

All concepts try to make the messaging exchange from a sender to a receiver through one or multiple Message Channels more reliable.

If they are correctly implemented, they increase the reliability of messaging significantly. However, all concepts have advantages and disadvantages. They should be analyzed in more detail (e.g., in the mentioned sources) before they are implemented. The goal of this section is to give an introduction on the different concepts available and to give a broad overview of the used patterns later on.

First, we present the three common delivery guarantee levels. They will accompany us throughout the rest of the paper.

Next, we discuss some message storage strategies. Then, we will look into the binding between the MOM and the receiving and the sending components and how they could interact with each other to transmit messages reliably. We put specific focus on the Two-Phase Commit, an important protocol to handle a distributed transaction. It is especially relevant for the "Exactly Once" delivery guarantee. Another focus is on how to design idempotent messages, a property which makes reliable messaging significantly easier.

Delivery Guarantees

First, we would like to discuss three delivery guarantees. The strategies differ in the number of message copies that a consumer will get from a single initially sent message. "At Most Once" will deliver one or zero copies of a message to the receiver, "At Least Once" will send one or more copies to the receiver, and "Exactly Once" will always try to send precisely one message to the receiver.

"At Most Once"

When using "At Most Once", a message will normally be delivered once from the sender to the receiver. However, if something fails in between, e.g., a crash of a MOM or a connection error, the message might not be received by the receiver at all.

This approach is more an anti-pattern for reliable messaging and can not be found in any of our primary sources. However, Hohpe and Woolf describe something similar in the pattern "Fire-and-Forget", this pattern is part of the "Conversation Patterns" which the authors present as an addition to "Enterprise Integration Patterns".[13, 14] However, the pattern "Fire-and-Forget" is not the same as "At Most Once", "Fire-and-Forget" is a more general pattern and has not necessarily something to do with messaging. In addition, "Fire-and-Forget" does not guarantee that a message could not also be duplicated. So "Fire-and-Forget" is more a "best-effort" semantics than a "At Most Once".

"At Least Once"

To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for

a message in a certain time frame, the message is re-transmitted.

This approach guarantees a message's delivery to the receiver, but duplicates are possible (see Figure 1).

The re-transmission of a message can have multiple reasons:[3]

1. the original message got lost before the receiver received it (Figure 1 scenario 1 step 11)
2. the receiver crashed during the processing of the message before an acknowledgment was sent. (Figure 1 scenario 2 step 21/22)
3. the acknowledgment was lost (Figure 1 scenario 3 step 32)

This pattern can be combined with the timeout-based delivery pattern (see Section "Timeout-based delivery"), especially when a receiver interacts with a MOM.[3]

"Exactly Once"

In "Exactly Once" delivery, the middleware only transfers the message once to the consumer. Therefore the message exchange must be reliable, as a retransmission, in case an acknowledgment is lost, is not possible. To achieve that, often a transactional message exchange is used.[3]

However, "Exactly Once" is challenging to implement, if not even impossible. It depends on the exact definition.[3, 15, 16] Mostly it is more economical and enough to handle "At Least Once".[15]

Durable Message Storage

To guarantee delivery, the messaging middleware must store the message as long as the transmission is not completed. To endure a restart or crash of the middleware, the message must be persisted to disk.[2, 3]

This pattern is described by Hohpe and Woolf as "store-and-forward". "store-and-forward" is a process where each computer involved in the message transfer stores the message locally before forwarding it to the next computer.[2]

In the pattern Guaranteed Delivery, the message has to be persisted before the message gets acknowledged by the receiver.[2]

If an error occurs, all messages, that could not be delivered, have to be kept in the middleware's storage. This can result in high disk space usage. To avoid running out of disk space, they suggest using a "retry timeout" (Message Expiration Pattern) to be able to discard the messages after a while.[2]

Interacting with the Sender and Receiver

In this part, patterns for the implementation of a receiver are discussed. We start with transaction based approaches and then go on to alternatives.

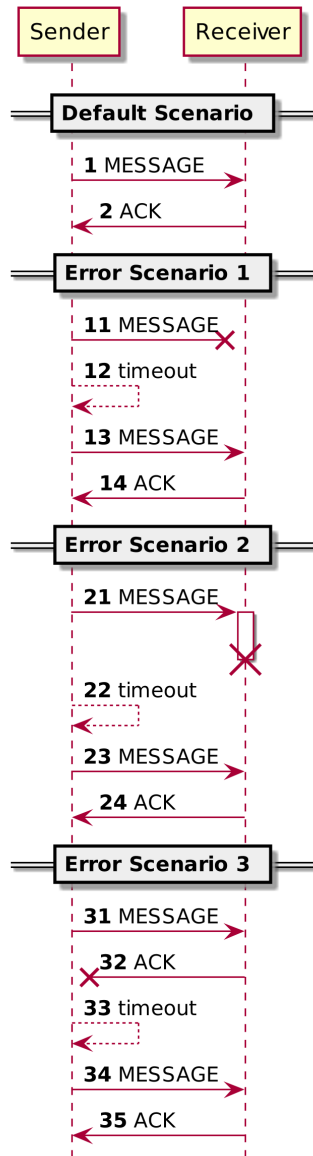
"At Least Once" delivery

Fig. 1: "At Least Once" delivery and the error scenarios

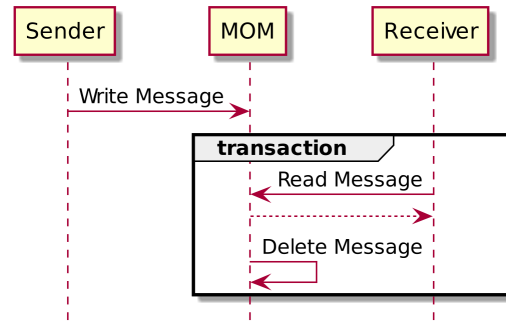
Transaction-Based Delivery

Fig. 2: Transaction-based delivery Flow

Transaction-based Delivery

The transaction-based delivery pattern by Fehling, Leymann, Retter, Schupeck, and Arbitter or transaction-based delivery by Hohpe and Woolf ensures that atomicity, consistency, isolation, durability (ACID) not only happens at the MOM, but it also gets extended to the receiver.

This approach ensures that messages are always received successfully.

This pattern can not only be applied to the receiver but also to the sender.[2, 3]

The patterns describes the following steps, visualized in Figure 2:

1. First, a message gets written to the MOM by the sender.
2. The receiver starts to read the message. The transaction starts.
3. The message gets deleted on the MOM

Transaction-Based Processor

The transaction-based processor by Fehling, Leymann, Retter, Schupeck, and Arbitter or transactional client by Hohpe and Woolf extends the transaction-based delivery pattern to include the processing of the event at the receiver. It ensures that a client not only receives a message "At Least Once" or "Exactly Once" but also processes the message "At Least Once" or "Exactly Once" successfully.[3]

This is ensured by a transaction context around the receiving component. The message is only deleted from the sending MOM after an acknowledgment was received. This acknowledgment is only sent after the successful processing at the receiver. The whole process, from getting the message until deleting the message on the sending MOM, is processed in one transaction. So now the transaction includes not only the MOM but also spans the client and potentially its storage provider.[3]

The processing flow of the pattern transaction-based processor is visualized in Figure 3.

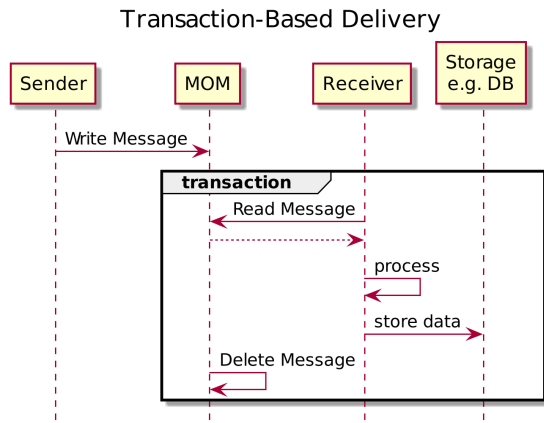


Fig. 3: Transaction-based processor Flow

Two-Phase Commit

The Two-Phase Commit is not only used in messaging but also for databases and other transactional systems. It is a broader concept than the other patterns.

The Two-Phase Commit describes how two or more transactional resources on different machines (without shared storage) can be updated or rolled back by a coordinator so that they end up in the same state.[17]

The two-phase is handled in multiple steps as described below and visualized in Figure 4.

Phase 1

All resources have to be prepared.

1. The coordinator is sending a request to all resources, a "REQUEST-TO-PREPARE"
2. Then the coordinator is waiting for the participants to respond.

Phase 2 If not all participants are responding with "PREPARED" (either with a "NO" or no response at all) the coordinator is sending an "ABORT" (Figure 4 Abort Scenario).

If all participants are responding with a "PREPARED" the protocol continues.:

1. the coordinator sends a "COMMIT".
2. the participants acknowledge with a "DONE".
3. The coordinator waits until it receives "DONE" from all participants.

If not all participants are acknowledging the transaction, a reminder could be sent. (Figure 4 Reminder Scenario)

The previous steps were mostly based on a Two-Phase Commit between different MOMs. However, the distributed commit can also involve a receiver or a sender component. This is described by the pattern transaction-based delivery (see Section "Transaction-based Delivery").

The X/Open already specified the Two-Phase Commit protocol in 1991 in the XA Specification, primarily

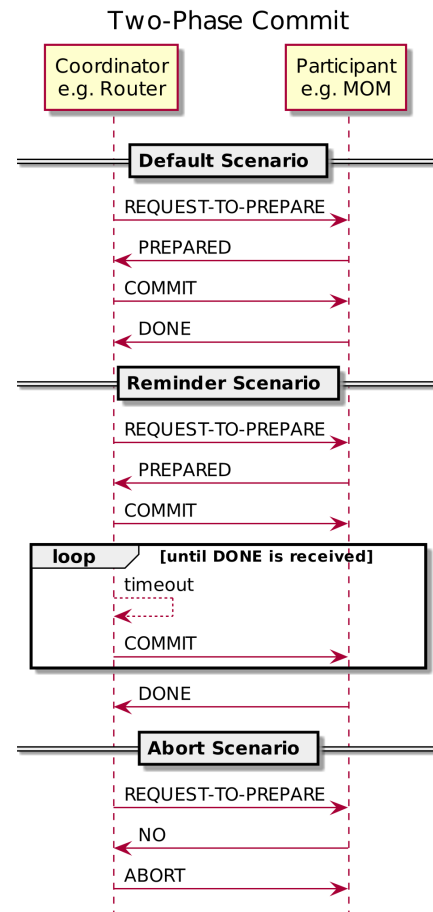


Fig. 4: Default flow of the Two-Phase Commit Protocol. To make the flow easier to understand only one participant is shown. However, a Two-Phase Commit needs at least two distributed participants; without that, a standard transaction occurs. The graphic is inspired by the protocol described by Bernstein and Newcomer.[17]

supposed for database usage but usable for all kinds of transactional systems. They use the wordings Transaction Manager (TM) and Resource Manager (RM). However, this standard does not handle network or communication failures in depth. They write: "More significant failures may disrupt the commitment protocol. The TM typically senses the failure when an expected reply does not arrive." [18] However, when the RMs committed the transaction, they are allowed to lose the knowledge of the transaction. So the same error scenario, as shown in Figure 4 Reminder Scenario, can happen.

Timeout-based delivery

Instead of using a transaction-based delivery method, a message could also be delivered without any transaction and just be acknowledged when the processing is done. For this strategy, the message has to be marked as invisible in the MOM. While the message is invisible, no other receiver can process it until the receiver's

acknowledgment arrives and the message is deleted.[3]

To handle the error case see the pattern in Section "Timeout-based message processor".

Timeout-based message processor

A message delivered to a receiver from the MOM is not visible to the other receivers anymore. The message is kept in that invisible state as long as the acknowledgment is not received. Now there could be a case where the receiver crashes or hangs (e.g., deadlock), and this will result in a situation where the message will never leave the invisible state at the MOM. To solve this issue, the message needs to have a timeout to return the message into the visible state.[3]

Idempotent Processor

Idempotence is a mathematical property where a function can be applied multiple times and the result still keeps the same value, e.g. $f(f(x)) = f(x)$. [2]

In messaging, this means that a message can be processed one or more times with the same effect.

"At Least Once" can lead to multiple executions of application function or similar.

The pattern Idempotent Receiver from Hohpe and Woolf describes how a receiver can be made idempotent. The pattern Idempotent Processor is based on the pattern of Hohpe and Woolf but Fehling, Leymann, Retter, Schupeck, and Arbitter expand it to a broader pattern that is not limited to messaging. Both patterns describe two strategies:[2, 3]

- Drop duplicated messages, with the help of a unique message identifier and a storage to keep them.
- Use a message semantics that is idempotent.

Dropping duplicated messages can be done with a message filter or directly in the receiver. The problem with this approach lies in how to define how long the message identifiers (IDs) should be stored. Often a timestamp in the message is used to define the retaining time, in consideration of the following points: If the storage time is chosen too high, it can lead to high storage costs and performance problems. If the number is chosen too low, duplicates are not detected because the message IDs are not in the storage anymore.

However, this problem can be solved with an expiration date defined per message. If the expiration date is reached, the message will always be dropped.[2, 3]

An example of a message with an idempotent semantics is, in the context of a temperature sensor, to send the actual temperature (e.g., 10° C) instead of the change of the temperature (e.g., +1° C).

When using an Idempotent Receiver, we can safely work with an "At Least Once" delivery to achieve an "Exactly Once" semantics.[2]

3 Evolution of the Patterns

The older patterns by Hohpe and Woolf use techniques like "store-and-forward" on each computer that the message passes.[2] However, in newer books like "Cloud Computing Patterns", in the spirit of the micro-services architecture only some computers need to store data persistently. Components in between only have to work with acknowledgments. Also, the concept of "At Least Once" is now more important because of the more distributed cloud computing concept, instead of bigger enterprise applications.[3]

"Exactly Once" is nowadays less relevant. In the past, most enterprise-grade, on-premise solutions supported "Exactly Once" delivery. However, many cloud-native solutions only offer "At Least Once". This is often because of the more distributed nature of modern, cloud-native systems when compared to traditional, monolithic systems.[3, 15]

The older book "Enterprise Integration Patterns" by Hohpe and Woolf does assume "Exactly Once" delivery as the default. Interestingly, they sometimes explicitly talk about the intentional and the unintentional violation of the "Exactly Once" property, but never strictly defined a concept like "At Least Once".

4 Defining Reliability

Our goal is to specify the reliability goals independent of the protocol. When this succeeds, our specification can be easily implemented and used with most messaging protocols. However, there are still protocols that have some limitations, for example if they do not support "At Least Once".

We propose a protocol-independent definition of the message delivery guarantees on the channel-level. An option called "delivery guarantee" should be placed on the channel to remove any confusion as to what guarantees can be expected from any given channel.

The following four values are possible:

- best-effort
- at-most-once
- exactly-once
- at-least-once

The default value is "best-effort" which means that the channel does not provide any delivery guarantee.

This new option should be used to specify how reliable a channel is. It can be used in a declarative way in a messaging contract, to signal what the receiver can expect.

This option allows all parties to simply see, what delivery guarantee a consumer can expect. In addition, a producer, knows how he should publish messages to the channel. However, not all protocols support all options, and so in practice, not all "delivery guarantee" values make sense for each protocol.

This new "delivery guarantee" option should be mainly used in software architecture to design new asynchronous messaging contracts. Additionally, it could be used to discover new asynchronous messaging Application Programming Interfaces (APIs) on the fly and subscribe to them with the correct "delivery guarantee".

To show possible usages of the "delivery guarantee", we assess three different DSLs and show how the DSLs could be extended to add this option if they are not yet able to perform this task.

A Domain Specific Language is a custom language for a specific purpose. It is an abstract language that should solve a description problem in a specific domain.[19]

Selection of the Languages

To select the DSLs to analyze we used the following criteria:

First, it was important, that the DSLs are message protocol independent. It only makes sense to declare the reliability in a protocol independent way if the same applies to the language. Otherwise, the protocol-specific description would need to be used.

Second, it has to be an open standard.

We started with AsyncAPI, which is a newer, already widely used language, that's growing in popularity.

Next, we chose to investigate asyncMDSL which is part of Microservice Domain-Specific Language (MDSL) and tries to address some shortcomings of AsyncAPI. Its authors use a more pattern oriented approach which is strongly influenced by the patterns described in "Enterprise Integration Patterns".[20]

As the third DSL, we chose the schema of the CloudSubscriptions Discovery API, which is quite new and only available as a preview version. Our previous work ([4, 21]) builds on the CloudEvents standard and therefore the CloudSubscriptions Discovery API holds special interest for us.

The mix of the three languages are also interesting, because they have some different focuses:

AsyncAPI is mainly contract focused, asyncMDSL is more architectural focused and CloudSubscriptions Discovery API is, as the name suggests, a discovery API and so, more interesting after the deployment of a system.

More details about the differences between the languages and how the "delivery guarantee" option could help in each case, is discussed in the coming sections.

AsyncAPI

AsyncAPI is an open-source initiative to make messaging and event-driven architecture easier, similar to what OpenAPI (formerly known as Swagger) does for REST APIs. At the center of AsyncAPI is the specification which is similarly structured to the OpenAPI

specification.[22]

It is quite new and growing in popularity.

An AsyncAPI document is a declaration on how to subscribe or publish to a channel of a MOM or different MOMs with different protocols in one document. Subsequently, in order to support multiple protocols in one schema the AsyncAPI specification needs to be protocol-independent.

The authors of AsyncAPI declare their solution as protocol-independent (they use the term "protocol-agnostic"[23]).[24] However, on the implementation level, most messaging protocols handle reliability a little bit differently and use varying options.[25]

In the previous version of AsyncAPI, version 1.2, they were not quite sensitive to these differences between the protocols. They had no concept to protocol-independently set a delivery guarantee nor had they a protocol-specific solution in place to declare the expected delivery guarantee per protocol. This shortcoming is already visible in one of their official examples: Figure 5 shows that limitation: The protocol-specific header fields (`qos` in the image) are put directly in the AsyncAPI for one protocol schema and so does not support multiple protocols anymore. `qos` is only a MQTT specific field, not a Advanced Message Queuing Protocol (AMQP) header field.[9] But the protocol should support both schemas: MQTT and AMQP, as visible in Figure 5.

In version 2.0 of the AsyncAPI specification, the authors addressed this shortcoming and differentiate between general channel options and protocol-specific ones:

The specification defines "bindings" as "... a mechanism to define protocol-specific information. Therefore, a protocol binding MUST define protocol-specific information only."[23]

This approach solves the shortcoming of not being able to specify protocol-specific fields for multiple protocols in one AsyncAPI document.

They also defined different fields, such as the `QoS` option for MQTT in the protocol specific bindings. The new flexibility for protocol-specific bindings is certainly a plus for protocol-specific options in general but comes with the drawback of making the solution significantly more complex, as shown in Listing 1.

It also makes the AsyncAPI less strict. Now, some options can be defined per protocol and do not have to be part of the AsyncAPI specification itself. As said before this makes the specification more flexible and powerful, but also less strict.

The delivery guarantee is one of the fields that should not be in the protocol-specific section: Research has shown that the delivery guarantee, as an indicator for the reliability, is, to a certain degree, not protocol-specific and should therefore be in the main AsyncAPI specification and therefore not only be part of the protocol bindings.[25]

As discussed in "Protocol Interoperability in a Stateless Message Router", most protocol delivery guarantees are interoperable, but not all.[25] So

for certain use cases, it makes sense to be able to set protocol-specific reliability options in the protocol-bindings. Only this way the AsyncAPI can model every possible scenario and be usable for also the most exotic use case. However, to fulfill their vision of a protocol-independent specification and to make the default case lean and simple, they should introduce a protocol-independent option for describing the reliability.[22, 23]

We want to propose the following compromise: We would like to introduce an additional, non protocol-specific option to the publish and subscribe declaration of the AsyncAPI specification called "delivery guarantee". This option defines a protocol-independent and elegant way that solves most cases in which defining the reliability is required. To support all use cases, it should still be possible to override the "delivery guarantee" with a protocol-specific option in the protocol-bindings. This way it is possible to have an easy and straightforward solution for simple cases, but the architect still has the possibility to take full control.

As the AsyncAPI defines some code generators that generate code from a specification, it is mandatory that a mapping from the simple "delivery guarantee" to the specific-protocol options exists. The definition of this mapping is out of scope for this paper.

The key to success for this solution depends significantly on the mapping from the "delivery guarantee" to the protocol-specific options. The mapping from the protocol-independent delivery guarantees to each protocol-binding's fields requires special considerations.

However, some protocol-bindings would need to be fixed, as they are not correctly describing all protocol-specific fields.

Such as this example in the protocol-binding of the MQTT protocol:

The description of the `qos` field in the AsyncAPI specification does not match the description of QoS in the MQTT specification. MQTT supports the values 0 ("At Most Once"), 1 ("At Least Once") and 2 ("Exactly Once").[9] AsyncAPI describes the value of the `qos` field as "Defines how hard the broker/client will try to ensure that a message is received. Its value MUST be either 0, 1 or 2."[26], which is clearly not a good description: QoS 1 is similar to our definition of "At Least Once", where a message could arrive at the consumer multiple times, while QoS 2 is similar to "Exactly Once".

AsyncMDSL

AsyncMDSL is part of MDSL and handles asynchronous contracts between systems, mostly by Message Channels. The asynchronous messaging part of MDSL was first introduced by De Liberali. The syntax is derived from the patterns described in "Enterprise Integration Patterns".[20, 27]

Microservice Domain-Specific Language (MDSL) is a DSL to describe the interactions between systems.

AsyncAPI Sample 1.0.0

This is a simple example of an AsyncAPI document.

Terms of service

<https://api.company.com/terms>

Connection details

URL	Scheme	Description
▶ api.company.com:{port}/{app-id}	mqtt	Allows you to connect using the MQTT protocol.
▶ api.company.com:{port}/{app-id}	amqp	Allows you to connect using the AMQP protocol.

Topics

PUBLISH `hitch.accounts.1.0.action.user.signup`

Message

Action to sign a user up.

Multiline description of what this action does. It allows Markdown.

Headers

Name	Title	Type	Format	Default	Description
qos	qos	integer	int32	1	Quality of Service
retainFlag	retainFlag	boolean		false	This flag determines if the message will

Fig. 5: AsyncAPI Sample 1.0.0 with QoS in the header field, however, this is not protocol independent! Sample from <http://editor.asyncapi.org/>.

```

1  channels:
2    user/signup:
3      publish:
4        bindings:
5          mqtt:
6            qos: 2
7            retain: true
8            bindingVersion: 0.1.0

```

Listing 1: Example of a Operation Binding Object from the MQTT Bindings of AsyncAPI.[26]

The language describes APIs as contracts in a vendor and protocol neutral way.

The language was first proposed by O. Zimmermann and is tightly coupled with the Microservice API Patterns.[28, 29] In contrast to AsyncAPI, MDSL describes the whole system landscape and not only an interface to one system or service.

We propose to add a `delivery guarantee` option to the channel declaration for this DSL. The field should allow the values `AT_LEAST_ONCE`, `AT_MOST_ONCE`, `EXACTLY_ONCE` and `BEST_EFFORT`. `BEST_EFFORT` should be used as the default. An example of a channel declaration with the proposed field is shown in Listing 2.

CloudSubscriptions Discovery API

The CloudSubscriptions Discovery API is a vendor-neutral API specification to discovers services and the events published by these services. The specification is part of the CloudEvents specification and is intended to be used with the CloudEvents message format.[30]

In contrast to AsyncAPI and asyncMDSL, this is

```

1 channel BanksLoansInsightsChannel
2   of type PUBLISH_SUBSCRIBE,
3     ↪ GUARANTEED_DELIVERY
4   on path "banks/${bankId}/loans"
5   with bankId: int, "The bank from which the
6     ↪ loan has been requested"
7   description "Subscribe to be notified of
8     ↪ new customers requesting loans"
9   accepts and produces message
10    ↪ NewLoanRequested
11   delivering payload {
12     "timeStamp": D<string>?,
13     "success": D<bool>
14   }
15   delivery option AT_LEAST_ONCE

```

Listing 2: Example of a channel declaration in AsyncMDSL with the new option on line 11. The example is originally from "AsyncMDSL extension (DSL and Eclipse plugin)".[27]

not an architecture description language (ADL). This API is used to discover a system's events that other systems can subscribe to. The discovery API publishes information about events produced by a particular system and does not define channels between systems like the other two ADLs. The producer should already be aware of the delivery guarantee ahead of the creation of an event and therefore the definition of the delivery guarantee should occur during event registration. So when an event, that was first discovered via the discover API, gets published to a channel, the channel should be declared with the same "delivery guarantee" as the event.

For example, when an event gets produced by a service in a batch-process scope, the event may get produced and sent out in a particular step. This step could re-run when the batch-process restarts (as a result of a failure in the first run). In this scenario, potentially a duplicated event will be produced. This leads to the fact that the event type has to be declared as "At Least Once" or "Best Effort". "Exactly Once" is not possible anymore.

To add the reliability option to the API, the already predefined `subscriptionconfig` property could be used. This property adds the possibility to add any key-value pair to the API. We suggest to add a key `deliveryoption` to every service, with the values defined on the beginning of this section. An example is shown in Listing 3.

As mention, the `subscriptionconfig` already exists and the additional field is fully conform with the current specification.[30]

5 Proposal for a Router Implementation

In this section, we combine the patterns and findings into a design that could be implemented by a router.

With the rise of the cloud, systems got more distributed, and a stateful service is expected to store its

```

1 {
2   "id": "62e8-c095-11ea-b3de-0242",
3   "epoch": 1,
4   "url": "https://example.com/widgetService",
5   "name": "widgets",
6   "specversions": ["1.0"],
7   "subscriptionconfig": {
8     "deliveryoption": "at-least-once"
9   }
10  "subscriptionurl":
11    ↪ "https://events.example.com",
12  "protocols": ["HTTP"],
13  "events": [
14    {
15      "type": "com.example.widget.create"
16    },
17    {
18      "type": "com.example.widget.delete"
19    }
20  ]
21 }

```

Listing 3: Example of a services in the CloudSubscriptions: Discovery API resource, with the delivery option on the service, on line 8. The sample is originally from "CloudSubscriptions: Discovery".[30]

data not only on its local disk but to distribute copies of the data to multiple machines. This distribution of the state increases the cost of a stateful service significantly.

For this proposal, we focus on a Message Router that could be easily deployed in a cloud environment and therefore we want to avoid the cost of statefulness.

The router should be able to forward messages from a MOM to one or multiple MOMs according to predefined routing rules.

The goal is to be able to make a proposal for all three proposed delivery guarantees from Section "Defining Reliability".

The examples always show the case of one MOM as a message sender and two MOMs as message receivers. The MOMs can be separate instances and even different products, or all three could be the same instance.

"At Most Once"

The flow for a router with an "At Most Once" semantics is shown in Figure 6. The steps are as followed:

1. The incoming messaging component sends the messages to the router, without the need for an acknowledgment.
However, if the protocol used needs an acknowledgment which is not necessary for this concept, it is important that the acknowledgment is sent and received by the sender before a message is sent to the receiver MOM. If this is not the case, there is a possibility that the message gets duplicated.

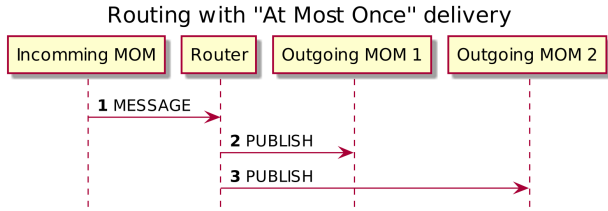


Fig. 6: Routing with "At Most Once" delivery

- 2.,3. The router then publishes the messages to all receivers. An acknowledgment is not necessary.

As shown, this process is quite easy and straightforward. However, this is not reliable because messages can get lost.

"At Least Once"

The flow for a router with "At Least Once" is shown in Figure 7. The steps are as followed:

1. The incoming messaging component sends the messages to the router.
- 2., 4. The message is then sent to the receiver
- 3., 5. The router waits for the acknowledgments of the receiver.
This may take some time, for example, if the consumer does some processing before they acknowledge the message (as described in Section "Transaction-based Delivery").
6. After the acknowledgments are received, the router will know that the message was delivered, enabling it to send the acknowledgment to the sender component.

If any of the steps are not successful or a service crashes during the processing, the process will be restarted from the sender, after a timeout. The timeout is reached when the sender does not receive an acknowledgment in the expected time frame. In this case, the sender will retry to send the message to the receiver, which will result in a restart of this process.

It only ends if the last step, the acknowledgment from the router to the sender (step 6), was successful, leading to the message being deleted from the sender component.

"At Least Once" delivery is much easier to implement compared to "Exactly Once", as shown in the following two chapters. However, it comes with the drawback that every message must either be idempotent or that the receiver has to use a message filter to remove the duplicates and work again with a "Exactly Once" semantics as described in Section "Message Filter for "Exactly Once"". Idempotent was discussed in Section "Idempotent Processor".

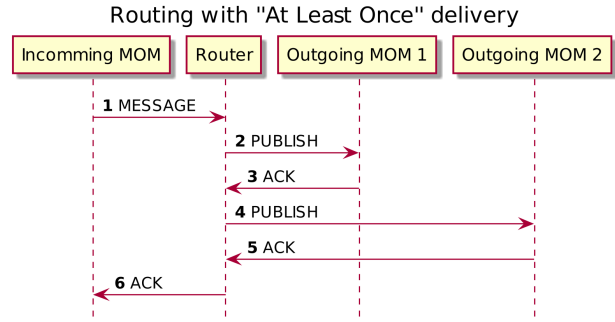


Fig. 7: Routing with "At Least Once" delivery

"Exactly Once"

A suggestion for a flow for a router with "Exactly Once" is shown in Figure 8. The steps are as followed:

1. The incoming messaging component sends the messages to the router.
- 2., 4. The router does a "PREPARE-PUBLISH" and sends the message to all outgoing MOMs.
- 3., 5. The router waits for a "PREPARED" answer from all outgoing components.
- 6., 8., 10. Now the router sends a "COMMIT" acknowledgment to all components (incoming and outgoing).
- 7., 9., 11. The router waits until all components (incoming and outgoing) acknowledge that the transaction was processed successful.
If an acknowledge is missing, the router has to retry it, until he succeeds, as shown in Figure 4 Reminder Scenario.

If any step before 6 fails (during phase 1 of the Two-Phase Commit) then the transaction will be rolled back with an "ABORT" message to all participants.

If something went wrong after this, there is no simple roll-back scenario and the router has to retry to commit until it succeeds. Additionally, all steps in this phase have to be persisted to disk so that the transaction can end successfully even if the router fails and has to restart during the transaction. This requirement would make the router stateful and therefore less interesting for a cloud environment.

We do not recommend this implementation and suggest an "At Least Once" routing with a message filter in front of the receiver that requires "Exactly Once".

However, if we would add an additional requirement, for example that all MOMs are of the same protocol, e.g. Kafka, then the scenario would be much easier to implement. More details about that can be found in "Protocol Interoperability in a Stateless Message Router".[25]

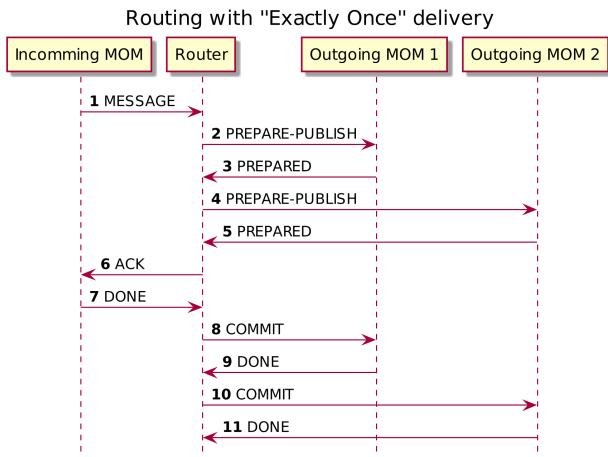


Fig. 8: Routing with "Exactly Once" delivery

Message Filter for "Exactly Once"

Another approach to providing "Exactly Once" delivery semantics is to transfer the message as "At Least Once" and only ensure "Exactly Once" delivery in front of the receiver. This approach allows us to have the easier handling of "At Least Once" before and just change the delivery option at the end, in front of the receiver, if it is really needed.

This could be done with a Stateful Message Filter as described in the Message Filter Pattern by Hohpe and Woolf.[2]

A Message Filter consumes messages from one channel, filters out undesired messages, and forwards the other messages to a different channel. In this case, the filter would have to read a unique identifier of the message (normally in the message body) and store it. If a received message identifier is already in the list of stored messages, the message will not be forwarded. This scenario is described as a possible example by Hohpe and Woolf and solved our problem directly. However, an additional requirement is that each produced message has a unique identifier.[2]

This approach is easier than adding the functionality directly to the router, and support "Exactly Once" there because there is only one output channel. However, this component has to be stateful, as it has to save already received messages.

In addition, there has to be a message expiration. Without that, the message filter has to store the already forwarded message IDs indefinitely.

These two requirements can be solved, when for example, using the CloudEvents message format. This format has a field `id` for a unique message identifier and a field `time` for the time when the event was created; a fixed timeout can then be defined using this field.[31]

An visualization of the message filter for converting an "At Least Once" to an "Exactly Once" channel is visualized in Figure 9.

However, there is still a possibility that a message gets lost, and the commit does not take place on the receiver. This is visualized in Scenario 3 in Figure 9. There the ACK message (number 47) was not received.

This can happen because:

- The DONE (message 46) was not received by the receiver.
- The receiver crashed during the process between message 46 and 47.
- The ACK was not delivered to the filter.

There is also the possibility that the filter crashes during the waiting phase for an ACK (message 46-49), which is not persisted to disk. However, the probability of this happening is extremely low.

This scenario could also be changed, such that the "DONE"/"ACK" to the consumer happens before the "COMMIT" on the filter. With this strategy it would be more an "At Least Once"/"Exactly Once" filter. With the flow shown in Figure 9 it is more an "At Most Once"/"Exactly Once" filter.

However, "Exactly Once" cannot always be implemented. It is only possible for special cases, where only one MOM is needed, with a central agent, that could do a central commit.[25, 32, 33]

6 Related Work

In Simple Object Access Protocol (SOAP) there were multiple attempts to standardize reliability with the standards WS-ReliableMessaging and WS-Reliability.[34, 35]

Ivaki, Laranjeiro, and Araujo present some patterns for reliable one-way messaging with a custom protocol over Transmission Control Protocol (TCP). Also, a sample implementation is provided. This custom protocol is used to handle message transition failures. The authors' solution is a mix of existing asynchronous messaging and one-way synchronous communication patterns. However, they call their proposal "correct", but lack complete proof of correctness; only a few test cases are provided.[36]

Tai, Mikalsen, Rouvellou, and Sutton describe a way of adding conditions, not to the Message Channel, but to the message itself. The application-specific conditions are then handled by the messaging middleware.[37]

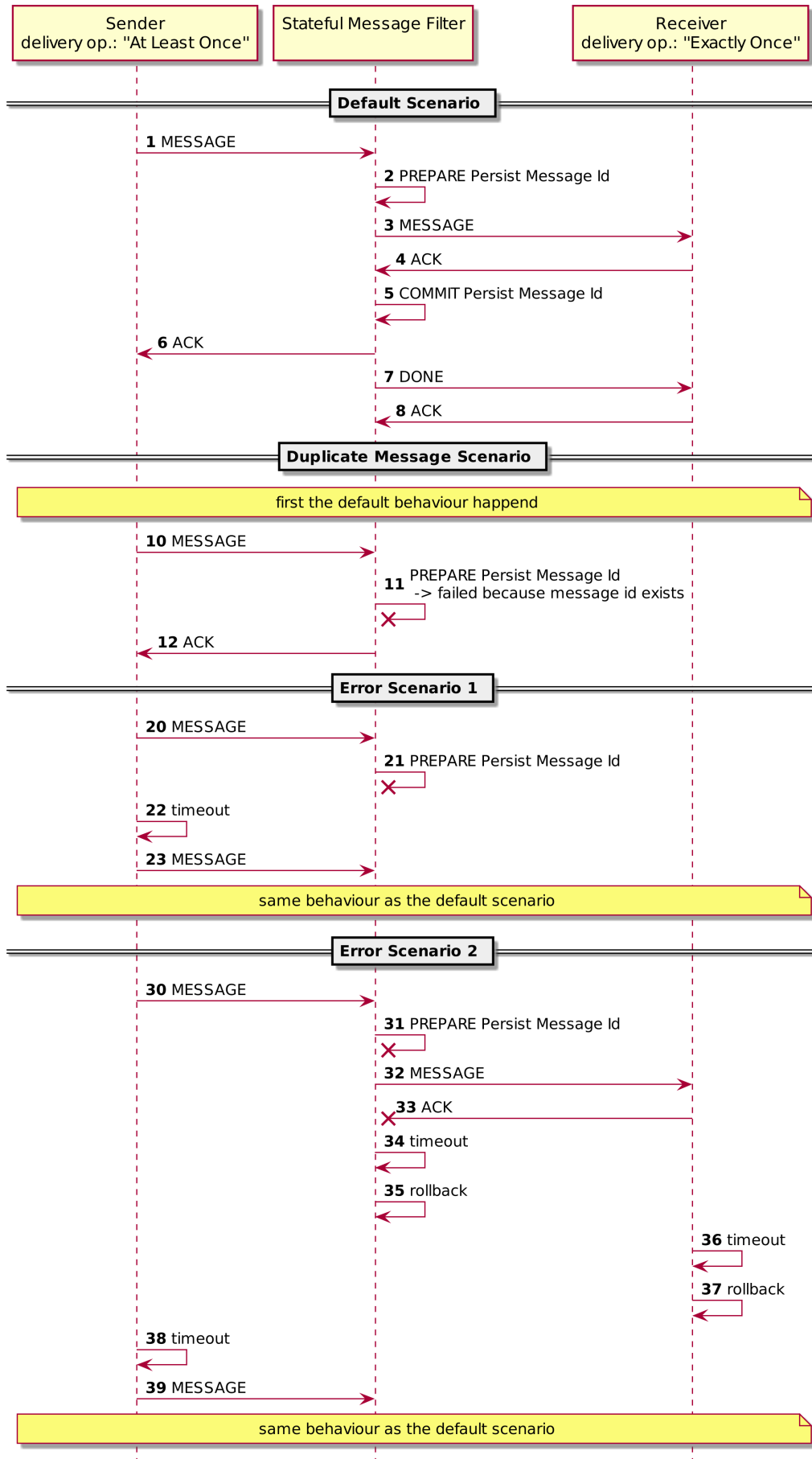
Bernstein and Newcomer describe in an abstract way how a client-server communication over queues can be processed with an "Exactly Once" semantics. The example describes a client that requests some processing via a queue by a server and gets a response in another queue. For this scenario, the possible error states are described.[1]

Latter, interactions of queues with other transactional systems that are irreversible are analyzed and transactional processing is discussed.

The book gives a good and easy understanding of the topic.[1]

The book lacks any references to the mentioned patterns and many other related work in other books.

Message Filter for "Exactly Once" delivery



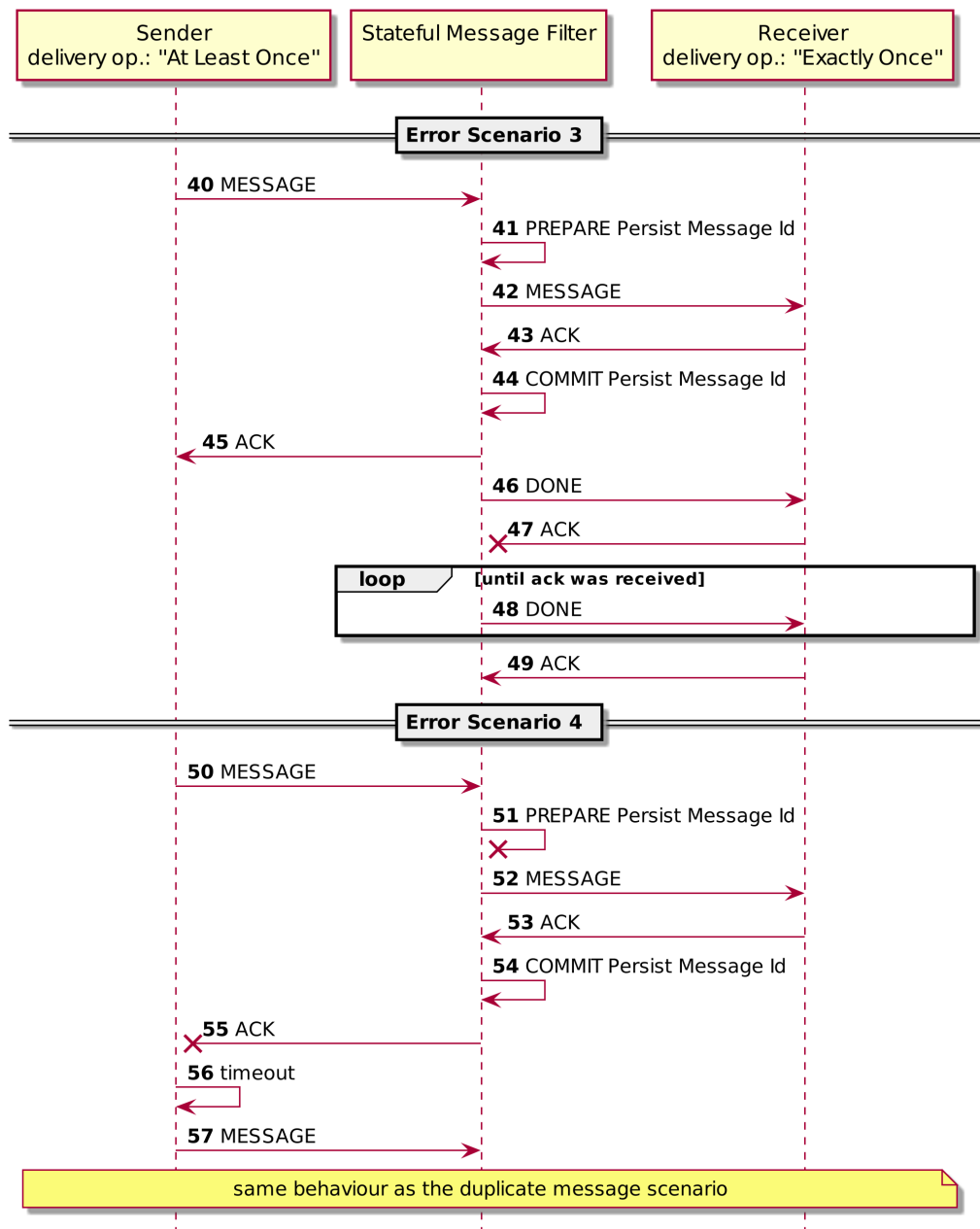


Fig. 9: Message Filter for "Exactly Once" delivery

Steen and Tanenbaum give a broad overview over many distributed system concepts. Their work contains many thoughts on error recovery and fault-tolerance. Even though the first version of the book is very old, much of it is still up to date and the content has often been updated in the newer versions.[16]

Lampson, Lynch, and Sogaard-Andersen show a proof for two "At Most Once" Message Delivery Protocols with additional order guarantee.[38]

In "Exactly Once Delivery and Transactional Messaging in Kafka", the design of Kafka's "Exactly Once" strategy is described. The work is not protocol-independent, but an interesting concept over all.[33]

7 Discussion

In this paper, we proposed a declarative option to describe reliability, which we refer to as "delivery guarantee". Our aim is to have a clear and reliable concept that prevents discrepancies between what a consumer expects and what actually is provided. The drawback of our concept is that it may be limiting for special use cases, where it is not enough to define the reliability options only on the channel level.

The other concept, which is used in many protocol definitions or special MOM implementations, allows defining the reliability options in more detail. They sometimes allow defining delivery options not only on the channel but also on each message, per sender, or per receiver.

We are strictly against this flexibility, as it makes the concept significantly more complicated and unclear. On one hand, it is confusing to determine which definition overrules which other definition when multiple delivery guarantees can be defined in different places (e.g., the channel but also the message itself). On the other hand, this could lead to contract breaches if a sender sends a message with "At Least Once" and a consumer consumes with "Exactly Once". In this case, the consumer expects that every message just arrives once, but because the sender uses "At Least Once" this is not true. If this happens, there is probably a problem in the concept.

For this reason, we decided to use this easy concept of a definition per channel, even if it is less flexible, it is clear and strict. However, if there is the use case of sending some messages from the same sender to the same receiver with different message delivery guarantees, then multiple channels and routing should be defined. This has the advantage that the definitions are still clear. However, the drawback of this approach is that the message order can not be guaranteed.

Other points that were not discussed in this paper: All error scenarios are just taking transient errors into account. Our concept introduced a retry mechanism, however, there are also permanent errors that can not be resolved by just retry to route the message

if a routing failed. This could, for example, be caused by a wrong configuration or a corrupted message. In such a case, the message could never be sent. Such a problem can lead to a crash loop of a receiver, especially when new delivery guarantees are taken into account: Then without a delivery guarantee that requires an acknowledgment, a corrupted message could be dropped, but with a delivery guarantee, such as "At Least Once", the message will be re-consumed after every restart. In this scenario, we need some additional error handling.

To handle persisting errors, additional considerations need to be made. Without these, the whole system could be trying to consume the same corrupted message over and over and will never go on to process other, valid messages.

This part is more about resilience than reliability, but it is crucial when introducing retries to handle failures. For such a situation, a concept like a dead letter queue is probably appropriate (or Dead Letter Channel by Hohpe and Woolf[2]).

Such a pattern, in addition to monitoring and alerting systems, allows us to apply the pattern Maximize Human Participation[39], which is very important in a large asynchronous system.

The pattern Maximize Human Participation by Hanmer describes that system operators should be able to interact with a system, especially in an error scenario; they should be able to monitor the system and help the system to recover in an error case. The expertise of these operators could help to resolve problems quickly. Giving experts some control over the error recovery can eliminate false attempts and focus the recovery down productive paths.[39]

However, these considerations are great topics for future work and are just mentioned here for completeness.

8 Conclusion

In this paper, we have shown some common patterns for reliable messaging and how they work.

In Section "Defining Reliability", we are looking into DSLs that describe messaging contracts. We proposed an option called "delivery guarantee" to specify with what delivery guarantee a message is transferred over a Message Channel. We showed how this option can be integrated into three DSLs. In addition, we have shown how to handle the delivery options in AsyncAPI with the given features.

The author of asyncMDSL was already informed about our proposal and implemented the new option into the MDSL syntax.¹ The flaw concerning the MQTT binding's description of the qos field in the AsyncAPI spec-

¹ The new MDSL specification version 5 includes the option "delivery guarantee" on the channel as proposed in this paper. <https://github.com/Microservice-API-Patterns/MDSL-Specification/blob/79ef5011591a05859a0c63c425e835e026d0a2ec/dsl-core/io.mdsl/src/io/mdsl/APIDescription.xtext#L107-L118>

ification has also been fixed by our pull request.²

In the third part, we have shown how we would apply these three delivery guarantees to a messaging router that forwards messages from one channel to one or multiple other channels. With this approach, all channels could potentially be on different MOMs and even on different protocols.

We showed that "Exactly Once" is hard to achieve and not always guaranteed in all scenarios.

² The description for the MQTT bindig has been changed with the pull request <https://github.com/asynccapi/bindings/pull/42>

References

- [1] P. A. Bernstein and E. Newcomer, Queued Transaction Processing, in Principles of Transaction Processing, Elsevier, 2009, pp. 99–119. DOI: 10.1016/b978-1-55860-623-4.00004-4.
- [2] G. Hohpe and B. Woolf, Enterprise integration patterns. Addison Wesley, Jan. 1, 2004, 480 pp., ISBN: 0321200683.
- [3] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, Cloud computing patterns. Springer Publishing Company, Incorporated, Feb. 18, 2014, ISBN: 978-3-7091-1567-1.
- [4] L. Basig and F. Lazzaretti, Reliable Messaging using the CloudEvents Router, 2021.
- [5] CloudEvents Specification v1.0, GitHub, 2019.
[Online]. Available: <https://github.com/cloudevents/spec/tree/v1.0>.
- [6] ISO/IEC 25010:2011: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2011.
[Online]. Available: <https://www.iso.org/standard/35733.html>.
- [7] Z. Ming and M. Yan, A modeling and computational method for QoS in IOT, in 2012 IEEE International Conference on Computer Science and Automation Engineering, IEEE, 2012. DOI: 10.1109/icse.2012.6269459.
- [8] ISO/IEC 25010. [Online]. Available: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010?limit=3&start=3>.
- [9] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, MQTT Version 5.0, OASIS Standard Incorporating, 2019.
[Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>.
- [10] S.-Y. Hwang, H. Wang, J. Tang, and J. Srivastava, A probabilistic approach to modeling and estimating the QoS of web-services-based workflows, Information Sciences, vol. 177, no. 23, pp. 5484–5503, 2007. DOI: 10.1016/j.ins.2007.07.011.
- [11] D. Petrova-Antonova and S. Ilieva, Towards a Unifying View of QoS-Enhanced Web Service Description and Discovery Approaches, EPTCS 2, 2009, pp. 99–113, Jun. 22, 2009. DOI: 10.4204/EPTCS.2.8. arXiv: 0906.3930 [cs.SE].
- [12] ISO/IEC 13236:1998(E): Information technology – Quality of service: Framework, 1998.
- [13] G. Hohpe and B. Woolf, Fire-and-Forget, 2017.
- [14] —, Conversation Patterns, 2017.
[Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html>.
- [15] T. Treat, You Cannot Have Exactly-Once Delivery, 2015.
[Online]. Available: <https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/>.
- [16] M. Steen and A. Tanenbaum, Distributed systems. Place of publication not identified: Maarten van Steen, 2017, ISBN: 9781543057386.
- [17] P. A. Bernstein and E. Newcomer, Two-Phase Commit, in Principles of Transaction Processing, Elsevier, 2009, pp. 223–244. DOI: 10.1016/b978-1-55860-623-4.00008-1.
- [18] Distributed Transaction Processing: The XA Specification, X/Open Company Limited, Tech. Rep. XO/CAE/91/300, 1991.
[Online]. Available: <http://www.opengroup.org/onlinepubs/009680699/toc.pdf>.
- [19] S. Efftinge, M. Volter, A. Haase, and B. Kolb, The Pragmatic Code Generator Programmer, 2006.
[Online]. Available: <https://www.theserverside.com/news/1365073/The-Pragmatic-Code-Generator-Programmer>.
- [20] G. De Liberali, AsyncMDSL: a domain-specific language for modeling message-based systems, Master’s thesis, Università di Pisa, 2020.
[Online]. Available: <https://etd.adm.unipi.it/t/etd-06222020-100504/>.
- [21] L. Basig and F. Lazzaretti, CloudEvents Router, HSR, 2020.
[Online]. Available: <https://eprints.hsr.ch/832/>.
- [22] AsyncAPI, 2020. [Online]. Available: <https://www.asyncapi.com/docs/getting-started/>.
- [23] AsyncAPI specification 2.0.0, AsyncAPI, 2020.
[Online]. Available: <https://www.asyncapi.com/docs/specifications/2.0.0>.

- [24] AsyncAPI Specification 1.2.0, AsyncAPI, 2019. [Online]. Available: <https://github.com/asynccapi/asynccapi/blob/943c665b9da94995baa4533f4625872b1c1df0c2/versions/1.2.0/asynccapi.md>.
- [25] —, Protocol Interoperability in a Stateless Message Router, 2021.
- [26] MQTT Bindings, 2020. [Online]. Available: <https://github.com/asynccapi/bindings/tree/a39a081e0f0d3da3d768c9af5725735c39116378/mqtt>.
- [27] AsyncMDSL extension (DSL and Eclipse plugin), 2020. [Online]. Available: <https://github.com/Microservice-API-Patterns/MDSL-Specification/tree/3cf15b9c866c1086da7e8cd354b2e991055f8d3d/examples/asynccMDSL>.
- [28] Microservice DSL (MDSL), 2020. [Online]. Available: <https://microservice-api-patterns.github.io/MDSL-Specification/>.
- [29] Microservice API Patterns, 2020. [Online]. Available: <https://microservice-api-patterns.org/>.
- [30] CloudSubscriptions: Discovery, 2020. [Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0.1/discovery.md>.
- [31] CloudEvents - Version 1.0, 2019. [Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/spec.md>.
- [32] N. Narkhede, Exactly-Once Semantics Are Possible: Here's How Kafka Does It, 2017. [Online]. Available: <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>.
- [33] Exactly Once Delivery and Transactional Messaging in Kafka, 2017. [Online]. Available: https://docs.google.com/document/d/11Jqy_GjUGtdXJK94XGsEIK7CP1SnQGdp2eF0wSw9ra8/edit?usp=sharing.
- [34] D. Davis, A. Karmarkar, G. Pilz, S. Winkler, and Ü. Yalçinalp, Web Services Reliable Messaging (WS-ReliableMessaging), OASIS, 2009. [Online]. Available: <http://docs.oasis-open.org/ws-rx/wsrn/200702/wsrn-1.2-spec-os.html>.
- [35] K. Iwasa, J. Durand, T. Rutt, M. Peel, S. Kunisetty, and D. Bunting, Web Services Reliable Messaging TC WS-Reliability 1.1, OASIS, 2004. [Online]. Available: http://docs.oasis-open.org/wsrn/ws-reliability/v1.1/wsrn-ws_reliability-1.1-spec-os.pdf.
- [36] N. Ivaki, N. Laranjeiro, and F. Araujo, Design Patterns for Reliable One-Way Messaging, in 2017 IEEE International Conference on Services Computing (SCC), IEEE, 2017. DOI: 10.1109/scc.2017.40.
- [37] S. Tai, T. Mikalsen, I. Rouvellou, and S. Sutton, Conditional messaging: extending reliable messaging with application conditions, in Proceedings 22nd International Conference on Distributed Computing Systems, IEEE Comput. Soc, 2002. DOI: 10.1109/icdcs.2002.1022249.
- [38] B. W. Lampson, N. A. Lynch, and J. F. Søgaaard-Andersen, Correctness of At-Most-Once Message Delivery Protocols, 1994.
- [39] R. Hanmer, Patterns for Fault Tolerant Software. John Wiley & Sons, Jul. 12, 2013, 320 pp.

C. Protocol Interoperability in a Stateless Message Router

Protocol Interoperability in a Stateless Message Router

Linus Basig, Fabrizio Lazzaretti
Advisor: Prof. Dr. Olaf Zimmermann

Department of Computer Science
OST – University of Applied Sciences of Eastern Switzerland
Campus Rapperswil-Jona

Abstract

In asynchronous messaging, different delivery guarantee semantics exist. Most messaging protocols implement some or all of them. The goal of this paper is to determine if common messaging protocols implement the delivery guarantee semantics similar enough to create a stateless, multi-protocol Message Router that preserves these semantics across protocols.

We analyzed the specifications of the five messaging protocols that have bindings defined in the CloudEvents specification. These five messaging protocols were MQTT, AMQP, WebHooks over HTTP, NATS, and Kafka.

Our research shows that the five messaging protocols implement the "At Most Once" and "At Least Once" semantics similar enough to allow the creation of a multi-protocol and delivery-guarantee-preserving Message Router. Unfortunately, preserving an "Exactly Once" semantics is not possible with the constraint of statelessness.

While analyzing the specifications of the five protocols, we encountered many particularities that affect the reliability of these messaging protocols. The biggest finding was that an imprecision in the MQTT specification could lead to unexpected data-loss if the user is not aware of it.

Keywords: messaging, quality of service, reliability, MQTT, AMQP, WebHooks, NATS, Kafka

Contents

1	Introduction	1
2	Analyzed Protocols	2
3	Results	4
4	Conclusion	8

1 Introduction

Many distributed systems use asynchronous messaging patterns to achieve decoupling, scaling, and reliability. One of the most popular books describing these patterns is "Enterprise Integration Patterns" from Hohpe

and Woolf.[1] Their book contains 65 messaging patterns that aim to help enterprises to integrate their systems.

One of these patterns is the Message Router. The pattern describes "a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel, depending on a set of conditions".[1]

This paper was written in the context of the bachelor thesis "Reliable Messaging using the CloudEvents Router" that discusses how a Message Router specialized in routing events structured according to the "CloudEvents Specification v1.0" can preserve the delivery guarantees of the messages it routes. CloudEvents is a specification proposed by the Serverless Working Group of the Cloud Native Computing Foundation (CNCF). This specification seeks to ease event declaration and delivery across services, platforms, and vendors. To achieve this goal, the specification defines the event structure, different serialization formats, and multiple protocol bindings. The specification is attracting a lot of attention and contributions from major cloud providers and Software as a Service (SaaS) companies.[2, 3]

Problem Description

In "Reliable Messaging – Patterns and Strategies for Message Routing", a literature research on reliable messaging, we established that there are three common delivery guarantee semantics. There is the "At Most Once" delivery semantics that expects a message to be delivered once, but in an error case, it might not arrive at all. Messages sent with an "At Least Once" delivery guarantee should be delivered once or multiple times if an error cases appears. The "Exactly Once" delivery guarantee expects that a message is delivered once and only once.[4]

In the framework of the "Enterprise Integration Patterns" the implementation of such a multi-protocol Message Router is trivial. The Message Router pattern described in the book routes messages between abstract Channels and the authors leave it open to the reader to use any messaging protocol to implement these Channels. Unfortunately, the book

does not discuss the previously mentioned delivery guarantee semantics in detail. The authors only describe a Guaranteed Delivery pattern that does not distinguish between "At Least Once" and "Exactly Once" semantics. The Guaranteed Delivery pattern must be implemented by persisting the message to disk on the sender's and the receiver's machine until the message transfer is completed. This technique ensures that the message cannot be lost even if any of the involved processes restart unexpectedly (e.g., after a crash or system reboot).[1]

In the age of the "infinitely" scalable cloud and ephemeral containers, the cost that comes with this statefulness makes the approach described in "Enterprise Integration Patterns" less attractive. With the rise of the cloud, systems became more distributed, and a stateful service is expected to store its data not only on its local disk but to also distribute copies of the data to multiple machines. This distribution of the state increases the cost of a stateful service significantly.

For this paper, we focus on a Message Router that could be easily deployed in a cloud environment, and therefore we want to avoid the cost of statefulness. The primary focus of the Message Router should be routing and it should leave other tasks, like message buffering and persistence, to specialized products.

Figure 1 shows the differences between stateful and stateless routing with an "At Least Once" delivery guarantee. In the Stateful Routing scenario, the reception and the forwarding of a message is decoupled by persisting the message to reliable but expensive storage. In the Stateless Routing scenario, the acknowledgment to the sender is delayed until the message is forwarded successfully to all receivers. We are willing to trade an increased coupling between the sender and the receivers for statelessness.

The implementation of a stateless router is only possible if all the involved messaging protocols have interoperable delivery guarantees. This would mean that the concepts and mechanisms used by the different messaging protocols to implement the delivery guarantees are translatable from one protocol to another.

The main goal of this paper is to examine how common messaging protocols implement the mentioned delivery guarantee semantics, how interoperable their implementations are and if it is theoretically possible to implement a Message Router that routes messages between different protocols while preserving the delivery guarantees of the involved protocols.

In Section "Analyzed Protocols", we define which messaging protocols are relevant for our use case and introduce them. Next, in Section "Results", we look at how these messaging protocols implement the different delivery guarantee semantics and determine how interoperable their approaches are. We also present the particularities that some of the analyzed protocols have regarding their implementation of the delivery guarantee semantics. Finally, we summarize our findings in Section "Conclusion".

2 Analyzed Protocols

Because there are more messaging protocols out there than we can analyze in this paper, we had to limit the selection of protocols for in-depth analysis. Since we wrote this paper in the context of the CloudEvents Router, we consulted the CloudEvents specification for guidance. Version 1.0 of the specification defines five protocol bindings, which define how these protocols should be used to transport CloudEvents. These five messaging protocols are Message Queuing Telemetry Transport (MQTT) 3.1.1 and 5.0, Advanced Message Queuing Protocol (AMQP) 1.0, WebHook over Hypertext Transfer Protocol (HTTP), NATS, and Kafka.¹ We decided to do an in-depth delivery guarantee analysis of these five protocols.[5–9]

For each protocol, we analyzed its specification to find out which delivery guarantees the protocol offers, how the protocol implements these guarantees, and if the protocol specification reveals any non-intuitive peculiarities regarding delivery guarantees. Then, we analyzed the interoperability of the delivery guarantee implementations of the messaging protocols.

In the following sections, we give a short introduction to each of the five protocols. We start with MQTT and AMQP because these are both open and well-established standards. We then look at WebHook over HTTP which is not an established standard but is as widely used as the previous two. After that, we look at the product-specific protocols of NATS and Kafka.

MQTT 3.1.1 and 5.0

Message Queuing Telemetry Transport (MQTT) is an open-source messaging protocol that is specifically designed to be used in an Internet of Things (IoT) environment. Its extremely lightweight design allows remote devices to send data with a publish/subscribe semantics while requiring only a small code footprint and minimal network bandwidth. Currently, Organization for the Advancement of Structured Information Standards (OASIS) maintains two versions of the MQTT standard: Version 3.1.1 and 5.0. While version 3.1.1 already defines all the basic functionalities required to reliably collect telemetry data, version 5.0 adds more advanced messaging patterns like message expiry, request-response flows, and shared subscriptions.

Our analysis is based on the specifications of both versions of the protocol.[10–12]

AMQP 1.0

The Advanced Message Queuing Protocol (AMQP) is an open-source messaging protocol with the goal of standardizing the message exchange between enterprises on the wire level. Before version 1.0, the AMQP specification also defined the behavior of the message broker. In version 1.0, the authors narrowed

¹ The protocols are ordered according to the relevance for the bachelor thesis.

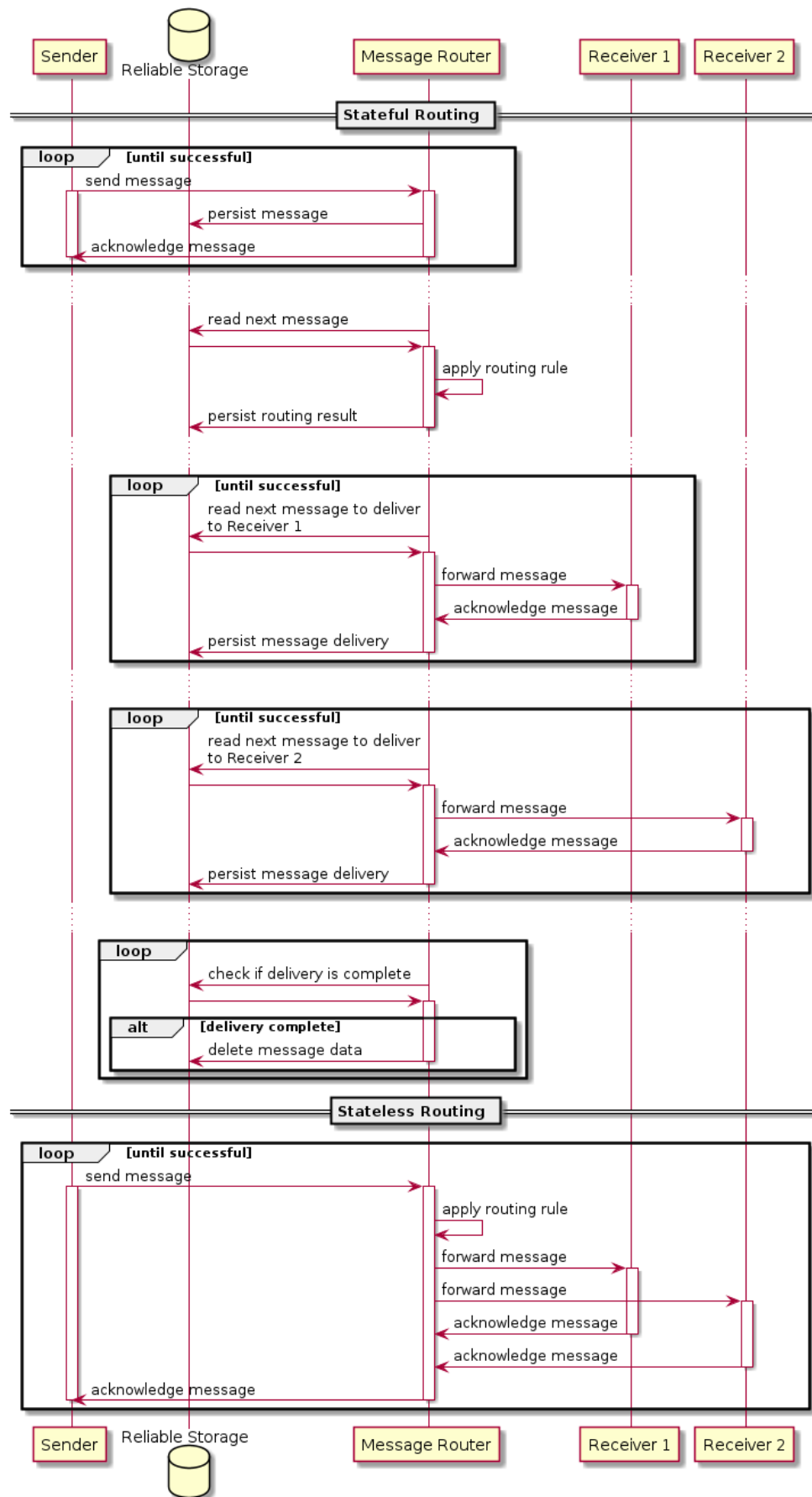


Fig. 1: Stateful versus Stateless Routing

down the scope of the specification to the exchange of messages between two nodes. Because of that, the pre-1.0 and 1.0 version of the protocol are not compatible.[13]

Our analysis is based on the "OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0" specification.[14]

WebHook over HTTP

The use of WebHooks is a widely adopted pattern to deliver events to HTTP-enabled services. Unfortunately, there is no formal specification for WebHooks, and most services implement their own flavor of it. As there is no widespread specification for WebHooks, there are no formally defined delivery guarantees.

Our analysis is based on the patterns described in the working draft of the "CloudEvents: HTTP 1.1 Web Hooks for Event Delivery" specification and the "REST Hooks" pattern collection.[15, 16]

NATS Protocols

NATS is a hyper-scalable messaging system originally built by Synadia for their global communications system (NGS). NATS is open-source and Synadia donated it in 2018 to the CNCF. Its design goal is to provide globally deployable, multi-tenant messaging infrastructure that supports publish/subscribe and request/reply messaging semantics.[17–19]

NATS uses a very simple, text-based protocol (called NATS Client Protocol). Its design goal is to allow the easy and performant implementation of clients in any programming language. To achieve this simplicity, the NATS Client Protocol does not implement any delivery guarantees.

If delivery guarantees are required, NATS expects them to be implemented on top of the NATS Client Protocol. An example of such an implementation is NATS Streaming. NATS Streaming is an extension for the NATS messaging system that adds a "At Least Once" delivery semantics and message persistence. It implements the "At Least Once" delivery guarantees on top of the NATS Client Protocol and is an optional part of the NATS project.[20]

Our analysis is based on the NATS and NATS Streaming documentation version 2.0.[17]

Kafka Protocol

The Kafka Protocol is the product-specific messaging protocol of Apache Kafka. Apache Kafka is a distributed log, originally built at LinkedIn, to enable the processing of activity-events at a grand scale. LinkedIn open-sourced it in 2011, and in 2014 the original authors founded Confluent to commercialize it.[21, 22]

Kafka implements a publish/subscribe pattern based on topics. Each topic represents an ordered log of messages and to achieve the required scalability, each topic is partitioned based on conditions defined by the publisher (called producer). The subscribers

(called consumers) can start reading from any point in the log because all published messages are persisted.[23]

Our analysis is based on the protocol specification for Kafka 2.6.0.[24]

3 Results

In this chapter, we present the results of the analysis of the messaging protocol introduced in the previous chapter and answer the question of their interoperability.

In short, we conclude that the implementations of the "At Most Once" and "At Least Once" semantics of the analyzed protocols are interoperable. We discovered that our goal of statelessness and how the protocols implement the "Exactly Once" semantics prevent us from declaring the implementations of the "Exactly Once" semantics to be interoperable. In the following sections, we discuss each semantics in detail and highlight the similarities and differences of the ways the different protocols implement them. At the end of the chapter, we present the peculiarities we discovered during the analysis of the protocols. We start with an overview of the delivery guarantee semantics each protocol supports (see Table 1).

Protocol	At Most Once	At Least Once	Exactly Once
MQTT 3.1.1 and 5.0[12]	yes ²	yes ³	yes ⁴
AMQP 1.0[14]	yes ⁵	yes ⁶	yes ⁷
WebHook over HTTP[15, 16]	yes ⁸	yes ⁹	no
NATS Protocols[25, 26]	yes ¹⁰	yes ¹¹	no ¹²
Kafka Protocol[24]	yes ¹³	yes ¹⁴	yes ¹⁵

Tab. 1: Supported Delivery Guarantees per Protocol

"At Most Once" Delivery

The "At Most Once" delivery guarantee is the lowest level of delivery guarantee available in all the analyzed protocols. Its implementation is extremely simple as

² by using MQTT Quality of Service (QoS) 0

³ by using MQTT Quality of Service (QoS) 1

⁴ by using MQTT Quality of Service (QoS) 2

⁵ by configure the sender to send settled messages

⁶ by configure the receiver to settle after receiving the message

⁷ by configure the sender to settle after receiving the acknowledgment

⁸ by not retrying in case of a non-2xx HTTP response code

⁹ by retrying in case of a non-2xx HTTP response code

¹⁰ by using the NATS Client Protocol

¹¹ by using the NATS Streaming Protocol

¹² NATS JetStream will support "Exactly Once" but it is currently in technical preview[27]

¹³ by requesting zero acknowledgments when publishing

¹⁴ by requesting an acknowledgments when publishing

¹⁵ by using Idempotent Producer and Kafka Streams

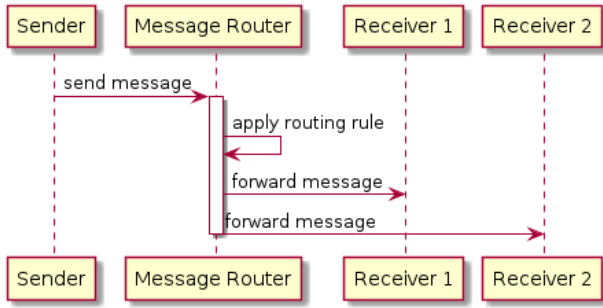


Fig. 2: "At Most Once" Routing

it only requires the sender to send the message on its way and hope it arrives at the receiver. Figure 2 shows how the router could implement the "At Most Once" flow.

The only analyzed protocol that does not follow this simple flow is WebHook over HTTP. This is due to its underlying protocol (HTTP) being request-response-based, forcing the receiver always to respond. This is a performance disadvantage but it does not harm the interoperability. To provide an "At Most Once" delivery guarantee, the sender is not allowed to retry the delivery of a message in any case.¹⁶ In "At Least Once" mode, the sender has to retry the delivery in case the receiver does not answer with a HTTP Status Code in the 2xx range (success).

"At Least Once" Delivery

The default strategy to implement an "At Least Once" delivery guarantee is that the sender stores a copy of the message until the receiver acknowledges the arrival of the message. In case the acknowledgment does not arrive in a certain time frame, the message is resent. Therefore, if the initial message delivery is successful but the acknowledgment is lost on its way back, a duplicated message delivery is possible. Figure 3 shows how a Message Router could implement the "At Least Once" flow.

All the examined protocols follow this concept. The only difference we observed was that Kafka and NATS Streaming distinguish between a client sending a message to a server and a server sending a message to a client. The other protocols use the same flow for both cases.

The reason for the differentiation can be found in the different underlying philosophies. Kafka and NATS Streaming are not, in comparison to the other protocols, centered around the exchange of messages between systems but around reading and writing to a persistent log of messages. Therefore, they distinguish between writing to the log and reading from it. The underlying concepts of message transfer, acknowledgment and retry are the same and do not reduce the interoperability.

¹⁶ We assume HTTP 1.1 over TCP is used and no request duplication can occur on the network layer.

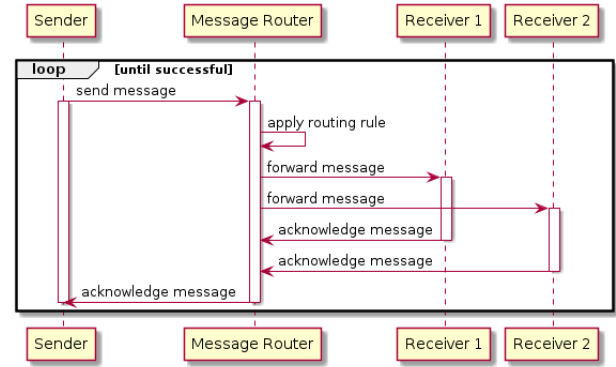


Fig. 3: "At Least Once" Routing

"Exactly Once" Delivery

For the "Exactly Once" delivery, we could observe the most differences between the protocols. NATS and WebHook over HTTP do not support the semantics at all and the others implement it with different strategies. All "Exactly Once" delivery guarantee supporting protocols have in common that they assign each message a unique identifier to track the delivery state and detect duplicates.

MQTT and AMQP

MQTT and AMQP use the same strategy to achieve an "Exactly Once" semantics but the differences in their implementation are big enough to make them worth mentioning. For MQTT the previously mentioned identifier to track the delivery state is called Packet Identifier and for AMQP it is called Delivery Tag. Figure 4 shows the "Exactly Once" message flow for MQTT and AMQP. MQTT uses four interactions between sender and receiver to ensure the message is delivered exactly once. AMQP uses a similar strategy but reduces the required interactions per transferred message from four to three. The protocol achieves this by performing a state-synchronization when a connection is reestablished after an ungraceful disconnect.

The reason for the differences can be found in the use cases the protocols were designed for. MQTT was designed for the collection of telemetry data in an IoT environment where network issues are expected to occur often. AMQP, on the other hand, was designed as a message-exchange protocol between enterprises where network issues should not occur as often while the expected volume of data is much higher. The protocol authors made a trade-off between minimizing the cost of setting up a new network connection and the cost of exchanging one single message.[10, 13]

Kafka

Kafka takes a slightly different approach to "Exactly Once" delivery. As mentioned before, it distinguishes between sending a message from a client (Producer) to the server and sending a message from the server to a client (Consumer). Figure 5 shows the "Exactly Once"

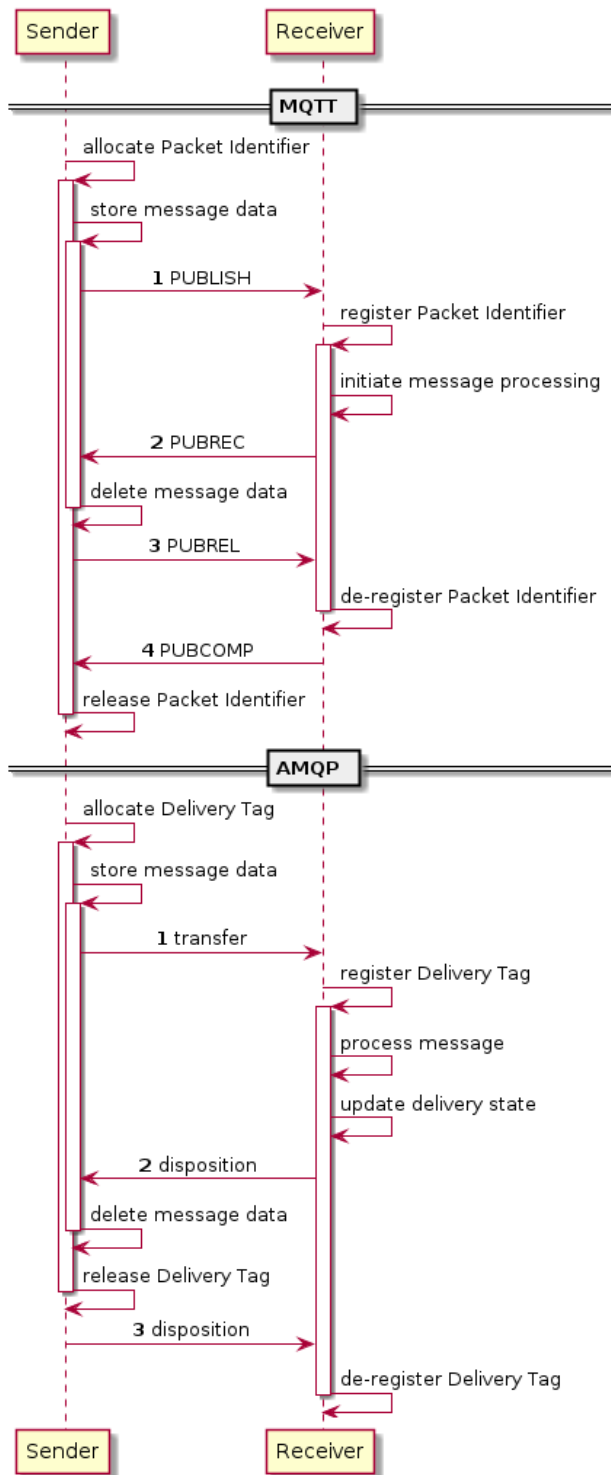


Fig. 4: "Exactly Once" Flow: MQTT vs AMQP

messaging flow with Kafka.

For sending a message from a Producer to the server with an "Exactly Once" delivery guarantee, it offers the concept of an Idempotent Producer. An Idempotent Producer assigns each message a sequence number (epoch) before sending it with an "At Least Once" delivery guarantee to the Kafka server. The server then, before persisting the message, checks if the epoch is higher than the epoch of the last received message. If the epoch is lower, the server assumes the message is a duplicate and ignores it. The advantage of this approach is that the message transfer can be done with only two interactions (same as for "At Least Once" delivery).

To ensure an "Exactly Once" semantics for sending a message from the server to a Consumer, a similar strategy is used. However, instead of using a client-specific sequence number, the offset of the message in the log is used. The main difficulty with this approach is that the client is responsible for reliably storing the last received message's offset. To help with that, Kafka allows a client to store the offset of the last received message on the server.

Built on top of that, Kafka also offers the Kafka Streams framework which extends "Exactly Once" semantics to the processing of the message and storage of the work result. With the Kafka Streams framework, a client can read messages from one or more topics, process these messages, write the work results to other topics and update its offsets all in one transaction. This allows not only the processing of one single message at a time but also operations on streams of messages. These message streams can be merged, split, mapped and aggregated with an "Exactly Once" semantics.[28, 29]

Interoperability

We consider the analyzed protocols to not be interoperable with a stateless router because there are error cases that break the "Exactly Once" semantics.

Figure 6 shows such an error-case: The Message Router crashes and gets restarted during the routing process of a message with an "Exactly Once" delivery guarantee. Because of its stateless design, all the information about ongoing message transfer is lost. When the router comes back online, the sender has a pending message transfer and tries to continue with that. Because the Message Router lost its state but already partially delivered the message, recovery is impossible without breaking the "Exactly Once" semantics. If Kafka¹⁷ or AMQP¹⁸ was used, the Message Router would process the same message again. This would result in a duplicated delivery to Receiver 1 which breaks the "Exactly Once" semantics. If the sender communicated over MQTT, it is not clear from the specification

¹⁷ For Kafka the reason for the reprocessing is that the router did not update the message offset before the crash and the offset stored on the server still points to the already processed message.

¹⁸ For AMQP the reason for the reprocessing is that the router does not recognize the Delivery Tag of in-progress message transfer and fails to realize it already processed the message.

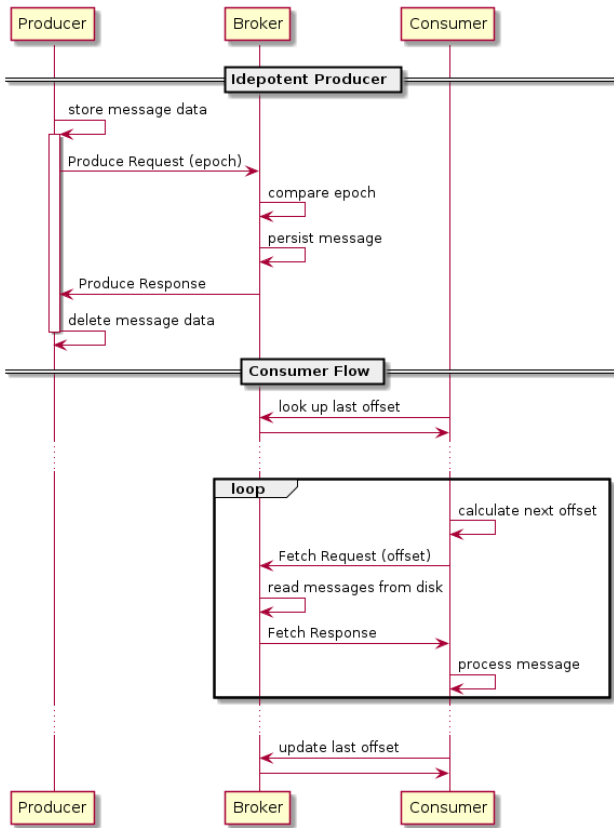


Fig. 5: "Exactly Once" Flow with Kafka

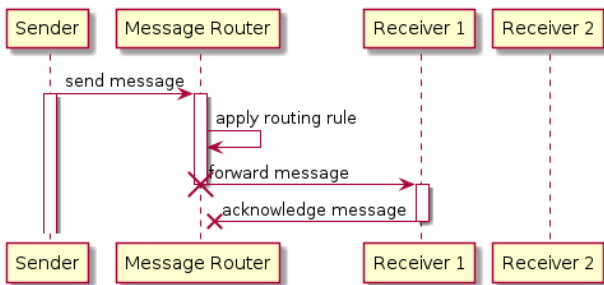


Fig. 6: "Exactly Once" Routing Failure

if there would be a duplicated delivery to Receiver 1¹⁹ or no delivery to Receiver 2 at all²⁰. The reason for this is an imprecision in the specification that leaves it up to the implementation to decide if the first acknowledgment is sent before or after the receiver processed the message. This also breaks the "Exactly Once" semantics. We discuss this issue in detail in Section "Protocol Peculiarities".

Protocol Peculiarities

During the analysis of the different messaging protocols, we discovered some peculiarities regarding reliable

¹⁹ If the MQTT implementation waits with PUBREC until after the message is processed, the router would not recognize a resent PUBLISH and redeliver the message to Receiver 1.

²⁰ If the MQTT implementation sends PUBREC before the message is processed, the router would not recognize a resent PUBREL and assumes the message is part of an already finished message transfer.

messaging that were not intuitive and could be confusing for new user of the protocols. Below, we present these findings.

MQTT

MQTT had some especially surprising peculiarities hidden in its specification that effect the reliability of the protocol. The main peculiarity we want to highlight is "The receiver does not need to complete delivery of the Application Message before sending the PUBACK".[12] This means that the guarantees provided by the protocol do only apply to the transfer of the ownership of a message and not necessarily include the processing of the message. If an application crashes before it has processed a message, the protocol does not guarantee that the message will be resent.

This quote is from a non-normative example in the specification, but it is also the only part of the specification that describes this aspect. The normative parts of the specification leave it open to the implementation to decide when to acknowledge a message.

This results in many MQTT libraries implementing the behavior proposed in the non-normative example of the specification.[30–32] This behavior is extremely unintuitive and even the maintainers of the Eclipse Mosquitto MQTT client library were surprised to rediscover that their library implements it this way.[30]

There are other particularities that effect the reliability of the protocol:

Maximum Packet Size A client can optionally define a Maximum Packet Size, which limits the size of an MQTT packet it is willing to receive. Messages with a bigger packet size will be discarded by the server with no regard to delivery guarantee.

Quality of Service (QoS) for Subscriptions MQTT allows the client to set a QoS for the subscriptions it creates. This does not mean, that all messages that are sent to the client because of this subscription are delivered with this QoS. The set QoS is only intended as a mechanism for the client to declare the maximal QoS level it is willing to receive. If the QoS of a message, that the server wants to forward to a client, is lower than the QoS of the client's subscription, the server has to deliver the message with its original QoS. If the QoS of the message is higher, it has to send it with the subscription's max QoS. To make it even more complicated, the server is allowed to grant a lower QoS for a subscription than the client requested.

Ordering Guarantees are Limited The protocol guarantees that successfully transferred messages from one client are delivered in order. This means messages sent and subscribed to with QoS 0 will arrive in order (if they arrive). However, if the messages are sent with QoS 1 or 2, and a transfer issue occurs, it is possible that messages arrive

out of order. This happens because the protocol allows to have multiple unacknowledged messages in-flight. If only some of these messages get acknowledged, and the unacknowledged messages get resent, the order can no longer be maintained. The protocol only guarantees that the resent messages are sent in the same order as in the previous delivery attempts. There is also an option in the protocol that allows servers to declare specific topics as un-ordered, which frees them from upholding even these limited guarantees.

Shared Subscriptions are Non-Intuitive Version 5.0 of the specification introduced Shared Subscriptions that allow messages that match a subscription to be distributed between multiple clients. Each client that participates in a Shared Subscriptions can set its own QoS level. This means that some messages can be delivered with one QoS to one client and with a different QoS to another client even if both clients participate in the same Shared Subscription.

It gets even more confusing when subscribing with QoS 2 because the meaning changes from "Exactly Once" to "At Most Once". If the delivery of a message with an "Exactly Once" delivery guarantee is interrupted and the delivery cannot be completed until the clients session expires, the unacknowledged message will be discarded by the broker to prevent a duplicated delivery. In the same situation, a message sent with QoS 1 ("At Least Once") would be assigned to another client attached to the Shared Subscription.

Local Storage is Implementation-Specific The protocol leaves it open to the implement to decide how unacknowledged messages are stored. The protocol does not offer any guarantees in case a client or server crashes.

AMQP

The main peculiarity of AMQP is that with version 1.0 of the specification the protocol was completely redesigned. Before 1.0, the protocol did not only define the transfer of a message but also defined the behavior of the message broker. Version 1.0 on the other hand is a pure message transport protocol, which solely handles the transfer from one node to another. These nodes can be message brokers, but if they are, their behavior is implementation-specific and not defined by the specification. In this sense, the pre-1.0 specification covered more aspects of reliable messaging by also defining things like persistence on the broker.[33]

Particularities of version 1.0 are:

Local Storage is Implementation-Specific The specification leaves it open to the implement to decide how unacknowledged messages are stored. In the case of a crashing node, the protocol does not offer any guarantees.

Message Order is Implementation-Specific The specification only defines the transfer of single messages. It is up to the implementation to decide if and how it wants to provide ordering guarantees.

WebHook over HTTP

As WebHook over HTTP is not a standardized mechanism there are no peculiarities to mention. The reliability of this method is highly dependent on the implementation.

NATS

The NATS and NATS Streaming protocol are very light-weighted and have only a few particularities:

Bring your own Delivery Guarantees If delivery guarantees are needed, they must be built on top of the NATS Client Protocol. The NATS project includes the optional NATS Streaming service, which implements an "At Least Once" delivery guarantee on top of the NATS Client Protocol.

Auto-Ack is the Default for NATS Streaming Per default the NATS Streaming client automatically acknowledges a received message without waiting for the application to finish processing the message. The acknowledgment can be delayed by setting the client into manual-ack-mode where the user has to manually acknowledge the message.²¹

Kafka

Because Kafka is not a specification but a mature, highly complex, and widely distributed system, it has plenty of factors and configurations that influence its reliability. Its reliability is not only affected by its protocol, which defines over 50 message types that each can have multiple versions, but also by its deployment configuration. It would go beyond the scope of this paper to capture all the peculiarities of Kafka.[23, 24]

4 Conclusion

The main goal of this paper was to examine how popular messaging protocols implement the common delivery guarantee semantics, how interoperable the implementations of the protocols are and if it is theoretically possible to implement a Message Router that routes messages between different protocols while preserving the delivery guarantees of the involved protocols.

To answer this question, we looked at five messaging protocols and analyzed their specifications. We determined that the implementations of the "At Most Once" and "At Least Once" semantics of all the analyzed protocols are interoperable. We also determined that the implementations of "Exactly Once" are not

²¹ <https://docs.nats.io/developing-with-nats-streaming/acks>

interoperable. Not all protocols implement it and the protocols that do would require the router to become stateful to be able to preserve the delivery guarantee in the case of an unexpected restart.

Based on our results we conclude that it is theoretically possible to implement a statless Message Router that routes messages between the analyzed protocols while preserving "At Most Once" and "At Least Once" semantics.

Main Finding

The most interesting aspect of this paper for us was to learn about the imprecision in the MQTT specification regarding the definition of a successful message transfer. In the specification, it is not clearly defined if a message must be processed before it is acknowledged in an "At Least Once" scenario. A non-normative example in the specification even explicitly proposes to send the acknowledgment before the message is processed by the application. This behavior can lead to message-loss in case of a crash and is not the expected behavior when "At Least Once" delivery guarantee comes into play. Unfortunately, many MQTT libraries implement this unsafe and unexpected behavior.[30–32]

Future Work

This paper only covers a small subset of the existing messaging protocols and it would be interesting to see how other protocols implement the delivery guarantee semantics. We would also be curious about the possibility of integrating other protocols into our stateless, multi-protocol, and delivery-guarantee-preserving Message Router scenario.

References

- [1] G. Hohpe and B. Woolf, Enterprise integration patterns. Addison Wesley, Jan. 1, 2004, 480 pp., ISBN: 0321200683.
- [2] L. Basig and F. Lazzaretti, Reliable Messaging using the CloudEvents Router, 2021.
- [3] CloudEvents Specification v1.0, GitHub, 2019.
[Online]. Available: <https://github.com/cloudevents/spec/tree/v1.0>.
- [4] —, Reliable Messaging – Patterns and Strategies for Message Routing, research rep., 2021.
- [5] MQTT Protocol Binding for CloudEvents - Version 1.0, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/mqtt-protocol-binding.md>.
- [6] AMQP Protocol Binding for CloudEvents - Version 1.0, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/amqp-protocol-binding.md>.
- [7] HTTP 1.1 Web Hooks for Event Delivery - Version 1.0, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/http-protocol-binding.md>.
- [8] NATS Protocol Binding for CloudEvents - Version 1.0, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/nats-protocol-binding.md>.
- [9] Kafka Protocol Binding for CloudEvents - Version 1.0, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0/kafka-protocol-binding.md>.
- [10] MQTT Homepage. [Online]. Available: <http://mqtt.org/>.
- [11] A. Banks and R. Gupta, MQTT Version 3.1.1, OASIS Standard Incorporating, 2014.
[Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [12] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, MQTT Version 5.0, OASIS Standard Incorporating, 2019.
[Online]. Available: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>.
- [13] 1.3. AMQP - Advanced Message Queuing Protocol Red Hat Enterprise MRG 3, 2019.
[Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_mrg/3/html/messaging_programming_reference/amqp_advanced_message_queuing_protocol.
- [14] R. Jeyaraman, A. Telfer, R. Godfrey, D. Ingham, and R. Schloming, OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, 2012.
[Online]. Available: <http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf>.
- [15] HTTP 1.1 Web Hooks for Event Delivery, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/master/http-webhook.md>.
- [16] REST Hooks, 2020. [Online]. Available: <https://resthooks.org/>.
- [17] NATS Introduction, 2020. [Online]. Available: <https://docs.nats.io/>.
- [18] Synadia - Connect Everything, 2020. [Online]. Available: <https://synadia.com/>.
- [19] CNCF to host NATS, 2020.
[Online]. Available: <https://www.cncf.io/blog/2018/03/15/cncf-to-host-nats/>.
- [20] NATS Streaming Introduction, 2020.
[Online]. Available: <https://docs.nats.io/nats-streaming-concepts/intro>.
- [21] Open-sourcing Kafka, LinkedIn's distributed message queue, 2011.
[Online]. Available: <https://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka>.
- [22] Announcing Confluent, a Company for Apache Kafka and Realtime Data, 2014. [Online]. Available: <https://www.confluent.io/blog/announcing-confluent-a-company-for-apache-kafka-and-real-time-data/>.
- [23] Apache Kafka, 2020. [Online]. Available: <https://kafka.apache.org>.
- [24] Kafka Protocol Guide, 2020. [Online]. Available: <https://kafka.apache.org/protocol>.
- [25] NATS Client Protocol, 2020. [Online]. Available: <https://docs.nats.io/nats-protocol/nats-protocol>.
- [26] NATS Streaming Protocol, 2020.
[Online]. Available: <https://docs.nats.io/developing-with-nats-streaming/protocol>.
- [27] JetStream (Technical Preview), 2020.
[Online]. Available: <https://github.com/nats-io/jetstream/tree/v0.0.19>.
- [28] N. Narkhede, Exactly-Once Semantics Are Possible: Here's How Kafka Does It, 2017.
[Online]. Available: <https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>.

-
- [29] Exactly Once Delivery and Transactional Messaging in Kafka, 2017. [Online]. Available: https://docs.google.com/document/d/11Jqy_GjUGtdXJK94XGsEIK7CP1SnQGdp2eF0wSw9ra8/edit?usp=sharing.
 - [30] add MOSQ_OPT_DELAYED_ACK option, 2020. [Online]. Available: <https://github.com/eclipse/mosquitto/pull/1932>.
 - [31] Prevent ack of received message until final, 2020. [Online]. Available: <https://github.com/eclipse/paho.mqtt.python/issues/348>.
 - [32] QoS 1 and QoS 2 ack control and persistence, 2020. [Online]. Available: <https://github.com/eclipse/paho.mqtt.c/issues/522>.
 - [33] Differences between AMQP 0-10 and AMQP 1.0, 2020. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_mrg/3/html/messaging_programming_reference/differences_between_amqp_0-10_and_amqp_1.0.

D. Architecture Documentation (arc42)

Reliable Messaging using the CloudEvents Router

arc42

Bachelor Thesis

Department of Computer Science
OST – University of Applied Sciences of Eastern Switzerland
Campus Rapperswil-Jona

Fall Term 2020/2021

Authors	Linus Basig, Fabrizio Lazzaretti
Advisor	Prof. Dr. Olaf Zimmermann
Project Partner	CARU AG
External Co-Examiner	Dr. Gerald Reif
Internal Co-Examiner	Ivan Bütler
Printing Date	January 15, 2021

About arc42

arc42, the Template for documentation of software and system architecture.

By Dr. Gernot Starke, Dr. Peter Hruschka and contributors.

Template Revision: 7.0 EN (based on asciidoc), January 2017

© We acknowledge that this document uses material from the arc42 architecture template, <https://www.arc42.de>. Created by Dr. Peter Hruschka & Dr. Gernot Starke.

Contents

0	Changes	5
1	Introduction and Goals	7
1.1	Functional Requirements (Business Goals)	7
1.1.1	Use Cases	7
1.2	Non-Functional Requirements (Quality Goals)	11
1.3	Stakeholders	11
2	Architecture Constraints	13
2.1	Technical Constraints	13
2.1.1	Programming Language	13
2.1.2	Version Control	13
2.2	Organizational Constraints	13
2.2.1	Apache 2 License	13
3	System Scope and Context	15
3.1	On the CARU Device.	15
3.2	In the Cloud	15
4	Solution Strategy	19
4.1	Content-Based Router Pattern	19
4.2	Rust	19
4.3	Microkernel Pattern	19
4.4	Reliable Messaging with "At Least Once"	20
4.5	Prototyping	20
4.6	Pair-Programming	20
4.7	Code Review with GitHub pull request	20
4.8	GitHub	20
4.9	Apache 2 License	21
4.10	Continuous Integration	21
5	Building Block View	23
5.1	Microkernel Events	24
6	Runtime View	27
6.1	Runtime Scenario 1: Loading and Bootstrapping	27
6.2	Runtime Scenario 2: Initialization	28
6.3	Runtime Scenario 3: Initial Configuration	29
6.4	Runtime Scenario 4a: Routing with "Best Effort"	30
6.5	Runtime Scenario 4b: Routing with "At Least Once"	31
6.6	Runtime Scenario 5: Configuration Update	32
7	Deployment View	37
7.1	Target Platforms at CARU	37
7.1.1	CARU Device	37
7.1.2	Virtual Machine in the Cloud	38

8	Cross-Cutting Concepts	39
8.1	Safety and Security Concepts	39
8.1.1	Starting Position	39
8.1.2	Minimize the Attack Surface Area	39
8.1.3	Handling Security Issues	40
8.2	Architecture Patterns	40
8.2.1	Microkernel Pattern	40
8.2.2	"At Least Once" Delivery Guarantee	40
8.3	Implementation Rules	42
9	Architecture Decisions	43
9.1	ADR-001: MADR for Architecture Decision Documentation	43
9.2	ADR-002: Rust as the Programming Language	45
9.3	ADR-003: Use of the Microkernel Pattern	48
9.4	ADR-004: Use of the Threading Model	50
9.5	ADR-005: Use of the Broker Pattern with Channels	52
9.6	ADR-006: "At Least Once" as the next delivery guarantee to implement	54
10	Quality Requirements	57
10.1	Quality Scenarios and Quality Tree	57
10.2	Landing Zones	59
11	Risks and Technical Debts	61
11.1	Single Unit of Mitigation	61
	List of Figures	63
	List of Tables	65
	Glossary	67
	Acronyms	75

0 Changes

Version 1.0 - 20.12.2019

The initial version of this document was written in the context of the student research project "CloudEvents Router".[1]

Version 2.0 - 15.01.2021

Version 2.0 of this document was written in the context of the bachelor thesis "Reliable Messaging using the CloudEvents Router".[2] Table 0.1 lists all changes made to version 1.0 to incorporate the new state of the project. Specifically, the changes required to make the router reliable.

Chapter	Section	Change Description
1 "Introduction and Goals"	1.1 "Functional Requirements (Business Goals)"	Add functional requirement FR-7.
1 "Introduction and Goals"	1.1 "Functional Requirements (Business Goals)"	Add non functional requirement NFR-4.
1 "Introduction and Goals"	1.3 "Stakeholders"	Add External & Internal Co-Examiner and update university name.
4 "Solution Strategy"	4.4 "Reliable Messaging with "At Least Once""	Add Reliable Messaging with "At Least Once" as a new important solution strategy.
4 "Solution Strategy"	Section 4.9 "Apache 2 License"	Add license constraints for included dependencies.
3 "System Scope and Context"	-	Overhaul CARU's Envisioned System Context.
5 "Building Block View"	-	Add the new building block "Scheduler" to Figure 5.1.
5 "Building Block View"	5.1 "Microkernel Events"	Add events of V2.0 and update description of OutgoingCloudEvent.
6 "Runtime View"	-	Introduce new Runtime Views: 6.1 "Runtime Scenario 1: Loading and Bootstrapping" as well as splitting the Runtime Scenario Routing into 6.4 "Runtime Scenario 4a: Routing with "Best Effort"" and 6.5 "Runtime Scenario 4b: Routing with "At Least Once"".
8 "Cross-Cutting Concepts"	8.1 "Safety and Security Concepts"	Add security measures and concepts.
8 "Cross-Cutting Concepts"	8.2.2 ""At Least Once" Delivery Guarantee"	Add "At Least Once" architecture pattern.
9 "Architecture Decisions"	9.6 "ADR-006: "At Least Once" as the next delivery guarantee to implement"	Add new ARR-006
10 "Quality Requirements"	10.1 "Quality Scenarios and Quality Tree"	Add nodes "Reliable Messaging" under the node "Reliability" to the tree and add "Operability" under the node "Usability". Matching leaves were added, too.
11 "Risks and Technical Debts"	-	Removed technical dept "Limited QoS Support". And update risk 11.1 "Single Unit of Mitigation".
-	-	Minor editorial changes

Table 0.1: Document Changes

1 Introduction and Goals

The goal of this project is to create a CloudEvents Router to bring CloudEvents to the CARU Device. The router would allow the subsystems of the device and the cloud exchange information seamlessly.

The name of the product is CERK. CERK stands for CloudEvents Router with a MicroKernel.

1.1 Functional Requirements (Business Goals)

Table 1.1 shows a list of the functional-requirements distilled from the use cases in subsection 1.1.1. Only the functional requirements in Table 1.1 are part of the scope of the project.

Version ¹	No	Requirement
1.0	FR-1	The router must understand the CloudEvents specification version 0.3 1.0. ²
1.0	FR-2	The router must be able to run on a modern Linux distribution
1.0	FR-3	A simple way to define routing rules based on the CloudEvents attributes must be designed
1.0	FR-4	The router should offer a UNIX socket interface to send and receive CloudEvents
1.0	FR-5	The router should offer a MQTT version 3.1.1 interface to send and receive CloudEvents
1.0	FR-6	The router should provide a simple interface to access debugging information
2.0	FR-7	Messages should be routed to multiple ports while preserving the delivery guarantee requested by the message source. The router should support an "At Least Once" delivery guarantee. ³

Table 1.1: Functional Requirements

1.1.1 Use Cases

The use cases give insights where these requirements are coming from and give context to the problems they should address. Only the main success scenarios are relevant for the requirements. The extensions are meant to be kept in mind when designing the system and are not part of the project's scope. The implementation and testing on real CARU Device hardware or in the cloud is not part of the project's scope.

¹Version of the project in which the requirement was introduced

²CloudEvents specification version 1.0 was released on the 24. of October and it was decided with the customer to support it.

³This requirement comes from the project proposal.[3]

Routing on the Edge (Inter-Process)

Context There are multiple services running on the CARU Device's main processor. Each of these services has a specific task like interfacing with a hardware component, communicating over the network or coordinating other services with the help of a state machine. The services are communicating with each other by emitting events and consuming the events of the other services.

Currently, most of these services are running in one single Python process and use an in-process broadcast to publish their events. There are already a few services which, for performance or tooling reasons, are implemented in another programming language and run in a separate process. For each of these external processes, there needs to be a service in the Python process to communicate with the external process and make its events available. This is an unnecessary step and adds undesired complexity.

Main Success Scenario The CloudEvents Router enables all services to run in separate processes by facilitating the controlled exchange of events between them. The exchange is defined by the routing rules.

1. Service A, B, C and D are communicating to each other via the CloudEvents Router.
2. Service A publishes an event by writing it to a UNIX socket it shares with the CloudEvents Router.
3. The CloudEvents Router receives the event and loads relevant routing rules from a configuration file.
4. The CloudEvents Router applies the rules to the CloudEvents attributes of the event to figure out which other services are interested in the event. Service C and D are interested in the event.
5. The CloudEvents Router forwards the event to the UNIX sockets associated to service C and D.
6. Service C and D receive the event on their socket and do something useful with it.

Extensions

Hardware Emulation The CloudEvents Router enables services, which interface with hardware components, to be easily replaced by a faker services which simulates the hardware and emits the corresponding events.

1. A developer wants to test something without deploying his changes to a CARU Device.
2. The developer runs a command and the CloudEvents Router, the hardware-independent services and the faker services get started.
3. All the services start communicate via the CloudEvents Router.
4. The developer interacts with a web-page provided by the faker service to simulate the interaction with the device.

Local Debugging The CloudEvents Router enables the easy inspection of the events passed between the services.

1. A developer needs to debug an unexpected behavior of a CARU Device.
2. He logs into the device and uses a tool to connect to the CloudEvents Router using a special debug UNIX socket
3. The CloudEvents Router starts sending the received events, the routing decisions and performance metrics over the socket.
4. The tool displays this information and helps the developer to find the issue.

Routing in the Cloud

Context CARU AG uses the public cloud to offer a variety of services related to their CARU Device. The functionality is similarly organized as on the device and the different services are also communicating with each other by emitting events.

Currently, all the events exchanged between the cloud services are structured according to the CloudEvents specification version 0.3. They use a proprietary product of the cloud provider to route the events between the different services. The routing capabilities of the proprietary product are quite limited. (Whitelisting, Blacklisting and Prefix-Matching)⁴

Main Success Scenario The CloudEvents Router enables the services in the cloud to receive CloudEvents based on more complex routing rules than currently available.

Extensions

Dynamic Routing The CloudEvents Router offers an api to dynamically adapt the configuration and routing rules without an interruption.

1. A customer wants some specific events emitted from his CARU Devices to be forwarded to a message broker he owns.
2. He enters the filter conditions and the configuration for his broker into the configuration console.
3. The service owning the configuration console calls the api of the CloudEvents Router to configure the event transport and the routing rule.
4. The deployed CloudEvents Router instances pick up the new configuration and routing rule and start forwarding the relevant events.

Routing between the Edge and the Cloud

Context The CARU Device is connected to the cloud via the MQTT protocol version 3.1.1. One of the services on the main processor of the CARU Device is responsible for sending and receiving messages over this connection.[4]

Currently, the events exchanged between the services on the CARU Device and the events exchanged between the device and the cloud are not following the CloudEvents specifications. When they are received by the cloud they have to be converted into a cloud event and republished.

⁴docs.aws.amazon.com/sns/latest/dg/sns-message-filtering.html

Main Success Scenario The CloudEvents Router enables events to be seamlessly exchanged between services on the CARU Device and services in the cloud.

1. Service A and B are connected to a CloudEvents Router on the device via MQTT. Cloud Service C and D are connected to a CloudEvents Router in the cloud. The CloudEvents Router on the device is connected to the CloudEvents Router in the cloud via MQTT broker.
2. Service A publishes an event to the CloudEvents Router on the device via UNIX socket.
3. The CloudEvents Router on the device applies the routing rules and knows service B and the CloudEvents Router in the cloud is interested in the event.
4. The CloudEvents Router on the device forwards the event to the local service B via UNIX socket and to the cloud via MQTT.
5. The CloudEvents Router in the cloud receives via MQTT and applies its routing rules. It knows service D is interested in the event.
6. The CloudEvents Router in the cloud forwards the event to service D over the appropriate event transport.

Extensions

Remote Debugging The CloudEvents Router enables the remote inspection of the events passed between the services on the CARU Device.

1. A developer needs to debug an unexpected behavior of a CARU Device.
2. He logs into the debug cloud console and selects the device he wants to debug.
3. The debug cloud console sends a command via MQTT to the CloudEvents Router on the device set it into debug mode.
4. The CloudEvents Router on the device starts sending the received events, the routing decisions and performance metrics over MQTT to the cloud.
5. The debug cloud console displays this information and helps the developer to find the issue.

MQTT Quality of Service (QoS) The CloudEvents Router respects the QoS properties of the messaging protocols.

1. The CloudEvents Router receives an event from via MQTT. The MQTT message has the QoS set to 1. This means the message should be delivered at least once.
2. The CloudEvents Router applies the routing rules and knows that services A and B are interested in the event.
3. The CloudEvents Router forwards the event to service A and B.
4. After successfully sending the event to service A and B, the CloudEvents Router acknowledges that it has processed the MQTT message.

Embedded Routing

Context The CARU Device has, besides the main processor, an additional low power processor running a Real Time Operating System (RTOS). The task of the low power processor is interacting with the environment sensors and managing the power of the device.

Currently, the main processor communicates with the low power processor over a serial line using the Modbus⁵.

Main Success Scenario The CloudEvents Router enables events to be seamlessly exchanged between the low power processor, the main processor and the cloud.

1.2 Non-Functional Requirements (Quality Goals)

The most important quality goals can be found in Table 1.2 and are derived from the project proposal and problem definition document.[5, 6]

Broader quality requirements are listed in the quality tree in Figure 10.1.

Refer to section 10.2 for the quantification of the non-functional requirements with landing zones.

Version ⁶	No	Goal	Scenario
1.0	NFR-1	Modularity	It should be easy to implement additional ports and configuration loaders.
1.0	NFR-2	Cross Platform	It should be easy to deploy the router to different environments especially to the cloud and embedded Linux platforms.
1.0	NFR-3	Open-Source	Easy findable and followable documentation to setup the project and run it.
2.0	NFR-4	Reliable Messaging	Route messages reliable from input to output. It should be guaranteed that no message can get lost.

Table 1.2: Nonfunctional Requirements

1.3 Stakeholders

In Table 1.3 lists the stakeholders of this project.

⁵modbus.org

⁶Version of the project in which the requirement was introduced

Role	Stakeholder	Expectation
Project Team Member	Fabrizio Lazzaretti (Student)	co-project management, co-system engineer, co-development
Project Team Member	Linus Basig (Student)	co-project management, co-system engineer, co-development
Project Team Supervisor	Prof. Dr. Olaf Zimmermann (HSR/OST)	easy understanding of architecture
Sponsor	Thomas Helbling (CARU)	understanding of the product idea
Customer	Reto Aebersold (CARU)	Knowledge of the external interfaces, easy understanding of architecture
External Co-Examiner	Dr. Gerald Reif	easy understanding of architecture
Internal Co-Examiner	Ivan Bütler (HSR/OST)	easy understanding of architecture

Table 1.3: Stakeholders

2 Architecture Constraints

All the constraints are derived from the project proposal and problem definition document.[5, 6]

2.1 Technical Constraints

2.1.1 Programming Language

Rust should be used as the main programming language.

Description Rust is a low-level programming language with guaranteed thread and memory safety. It achieves that with its unique "ownership model". The goal is to build safe and reliable software.[7, 8]
Rust is used by Mozilla Firefox, Dropbox and many others.[9, 10]
In 2019, Rust has also been elected in the well-known survey of Stack Overflow for being the "most loved" programming language.[11]

Reason During the student research project, CARU was interested in receiving a first-hand experience report on getting started with Rust.

2.1.2 Version Control

Git with GitHub should be used as the version control system.

Descriptions

Git Git is an open-source distributed version control system. The tool was created in 2005 by the Linux community.[12]

GitHub GitHub is a source code management and collaboration platform based on Git. It is a well known platform for open-source code and was acquired in 2018 by Microsoft.[13]

Reason A goal of the student research project was to open-source the project and GitHub is a widely used hosting platform for open-source projects.

2.2 Organizational Constraints

2.2.1 Apache 2 License

The Product should be licensed under the Apache 2 license.

Description Apache 2 is an open-source license from the Apache Software Foundation. The license allow modifications and redistribution of the software. It has no copyleft.[14]

Reason CARU would like to see the product available to the public under the Apache 2 license.

3 System Scope and Context

Figure 3.1 shows CARU’s envisioned system context with the unified event plane implemented. Whenever events are exchanged, they are formatted according to the CloudEvents specification. The CloudEvents Routers are placed in each system and connected over an appropriate messaging protocol to its neighbors. Consequently, the services are then also connected to their CloudEvents Router over an appropriate messaging protocol. The figure shows an exemplary system context diagram derived from the use cases (see Chapter 1 “Introduction and Goals”). Because our solution should be configurable and extendable, there is an infinite amount of possible system contexts.

In the following section, we take a look at the two environments in which the router will be placed.

3.1 On the CARU Device.

Figure 3.2 shows the envisioned communication on the CARU Device. The CloudEvents Router is the center of communication on the device. It is responsible to facilitate the communication between the services on the device and the communication between the services on the device and the cloud.

For the communication on the device, the router communicates over an embedded MQTT broker with the different services on the device. This embedded broker is not strictly required but frees the router and the services from buffering events. This is especially useful in case a service or the cloud is temporarily not available.

For each service on the device, two topics are configured on the embedded MQTT broker. The Service Outbox Topic is used to buffer events from the service to the router and the Service Inbox Topic is used for events in the other direction.

3.2 In the Cloud

Figure 3.3 shows the envisioned communication in the in the Amazon Web Services (AWS) cloud.

The CloudEvents Router in the cloud routes events between CARU’s Cloud Services which mostly run on AWS Lambda and also between the Cloud Services and CARU Devices. AWS Lambdas is the Function as a Service (FaaS) offering from AWS. It is tightly integrated with the rest of the AWS ecosystem and allows the implementation of such functions in many programming languages, like Python, Go, or Java.[15] The main difference is that in the cloud Simple Queue Service (SQS) is used in place of the embedded MQTT broker.

Additionally, the cloud hosts the event broker that is used to buffer the events exchanged between the device and the cloud. For this exchange, the AWS IoT Core broker is used which communicates over the MQTT protocol. In the future, the here located router can be used to also route events over other protocols to integrate partners and other services that do not use MQTT.

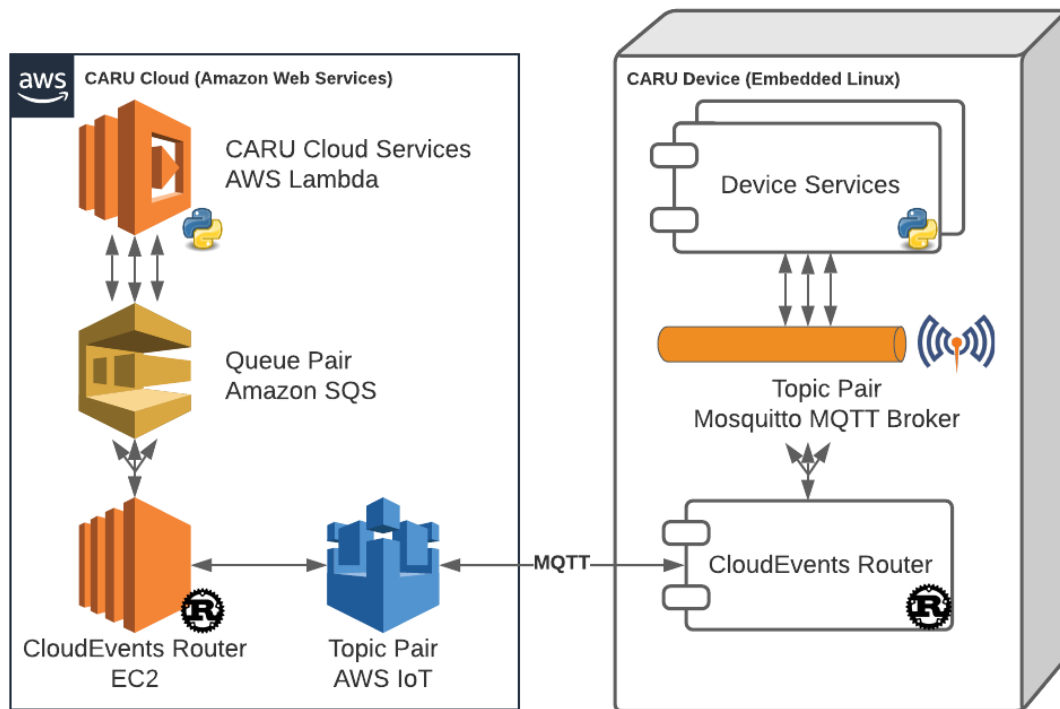


Figure 3.1: CARU's Envisioned System Context

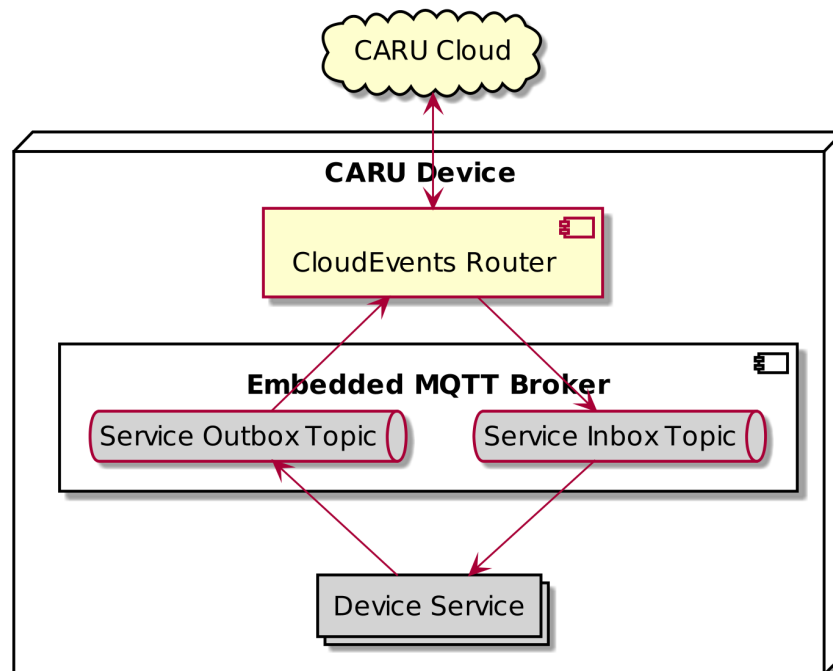


Figure 3.2: CARU's Envisioned Communication on the Device

Possible Deployment in the CARU Cloud

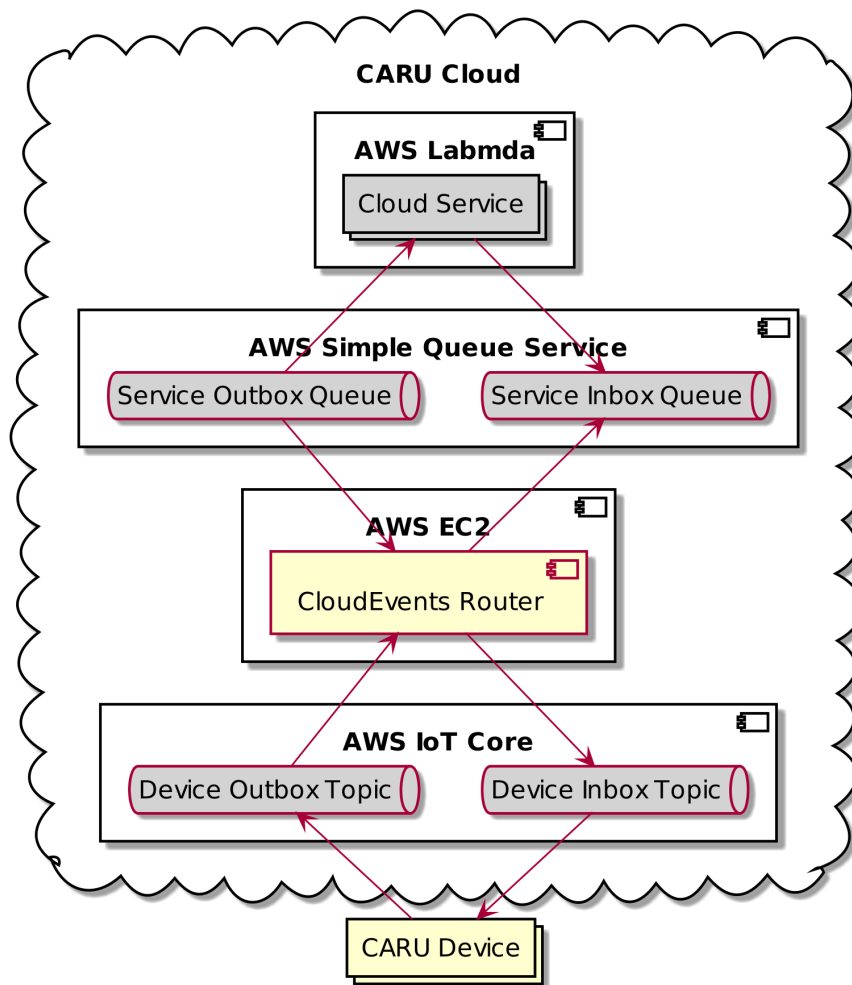


Figure 3.3: CARU's Envisioned Communication in the Cloud

4 Solution Strategy

The most important Solution Strategies are explained in the following sections. The strategies are sorted by their influence on the outcome of the project.

4.1 Content-Based Router Pattern

The router should be implemented according to the Content-Based Router pattern described in "Enterprise Integration Patterns" by Hohpe and Wolf.[16]

Helps with FR-3

Details "The Content-Based Router examines the message content and routes the message onto a different channel based on data contained in the message. The routing can be based on a number of criteria such as existence of fields, specific field values etc."[16]

4.2 Rust

The Rust programming language was requested by the customer.

Helps with Constraints

Details ADR-002

Rust is a low-level programming language with guaranteed thread and memory safety. It achieves that with its unique "ownership model". The goal is to build safe and reliable software.[7, 8]

Rust is used by Mozilla Firefox, Dropbox and many others.[9, 10]

In 2019, Rust has also been elected in the well-known survey of Stack Overflow for being the "most loved" programming language.[11]

4.3 Microkernel Pattern

The Microkernel Pattern helps with the modularization and portability of our solution.

Helps with NFR-1 and NFR-2

Details ADR-003 and MicroKernel

4.4 Reliable Messaging with "At Least Once"

Reliability of the router, especially message delivery guarantees were the main focus of version 2 of the project. It was a requirement from the customer to extend the router with a guarantee of message delivery over different messaging protocols. In our research, we found out that the delivery guarantee "At Least Once" fits the use cases of the project best.[17, 18] More details about the decision for "At Least Once" could be found in the attached Architecture Decision Record (ARD).

Helps with FR-7 and NFR-4

Details ADR-006

4.5 Prototyping

Before the final product is built, multiple prototypes were created to minimize the risk and decide which concepts should be used for the final product.

4.6 Pair-Programming

The main components of the router were implemented with pair-programming style to bring the best possible quality.

Details Pair-programming is a style of programming where two programmers working collaboratively on the same code, design or test at the same time. The desired goal behind this strategy is to have better code quality and better know-how distribution.[19, 20]

4.7 Code Review with GitHub pull request

The master branches of the repositories were setup up as protected so that new code could be only merged into the master branch by pull request. Every pull request needs at least one reviewer. With this strategy, we can guarantee that every code was reviewed before it was merged into the master branch.

4.8 GitHub

GitHub helps us to make an open-source product that can be found and used by other programmers.

Helps with NFR-3

Details <https://github.com/ce-rust/cerk>

GitHub is a source code management and collaboration platform based on Git. It is a well known platform for open-source code and was acquired in 2018 by Microsoft.[13]

4.9 Apache 2 License

The Apache 2 license helps us to make an open-source product that is interesting for companies to use.

Helps with NFR-3

Details Apache 2 is an open-source license from the Apache Software Foundation. The license allow modifications and redistribution of the software. It has no copyleft.[14]

However, it is not feasible that all used libraries are licensed under the same Apache 2 license.

The used libraries are allowed to be licensed under one of the following licenses:

- Apache-2.0
- BSD-2-Clause
- BSD-3-Clause
- EPL-1.0
- MIT
- ISC

The license selection is enforced by the crate "cargo-deny".¹ A non matching license of a dependency will be detected in the Continuous Integration (CI).

4.10 Continuous Integration

The CI helps us to always have a compiling and runnable product.

An example user interface of the output is provided in Figure 4.1.

Details Buildkite was used as the CI. Buildkite is a service for coordinating buildpipelines which get executed on own infrastructure. They provide a build agent that can easily be deployed to AWS and other platforms. Buildkite is well integrated with GitHub.[21]

¹<https://github.com/EmbarkStudios/cargo-deny>

The screenshot displays the Buildkite CI interface for the 'ce-rust / cerk' repository. The top navigation bar includes links for Pipelines, Agents (5), Users, Settings, Reports, My Builds, Documentation, Support, and a user profile for F. Lazzaretti. The main header shows the repository name 'ce-rust / cerk' with a 'Public' badge, a GitHub icon, and statistics: 215 Builds, 0 Running, and 0 Scheduled. There are buttons for 'New Build' and 'Pipeline Settings'.

The current build is for 'Merge pull request #46 from ce-rust/fix/routing-fields-doc', Build #212, on the 'master' branch, with commit 'db472f2'. It passed in 5m 25s. The build steps are listed below:

- Upload Pipeline**: buildkite-agent pipeline upload. Ran in 15s, Waited 5s. ID: CERK-i-049b635f6f6cc3567-2.
- test**: Ran in 3m 36s, Waited 2s. ID: CERK-i-049b635f6f6cc3567-2.
- rust doc**: Ran in 4m 12s, Waited 4s. ID: CERK-i-03e8036327389119e-5.
- check readme**: Ran in 25s, Waited 4s. ID: CERK-i-03e8036327389119e-4.
- example: execute hello world**: cd examples/src/hello_world && docker-compose run hello-worl... Ran in 3m 16s, Waited 7s. ID: CERK-i-049b635f6f6cc3567-4.
- example: build UNIX Socket and MQTT for armv7**: cd examples/src/unix_socket_and_mqtt_on_armv... Ran in 5m 5s, Waited 7s. ID: CERK-i-049b635f6f6cc3567-5.
- example: execute rule based routing**: cd examples/src/rule_based_routing && docker-compose run ... Ran in 15s, Waited 7s. ID: CERK-i-049b635f6f6cc3567-3.
- setup: ubuntu/linux**: cd setup && docker-compose run ubuntu/linux Ran in 3m 39s, Waited 3s. ID: CERK-i-03e8036327389119e-3.
- setup: archlinux**: cd setup && docker-compose run archlinux Ran in 3m 39s, Waited 4s. ID: CERK-i-03e8036327389119e-2.

The build was triggered by F. Lazzaretti at 10:45 AM via a Webhook. A 'Rebuild' button is available.

Figure 4.1: The CI of CERK. The figure shows a passed CI job on Buildkite. The job for the CERK CI contains 8 separate jobs.

5 Building Block View

In Figure 5.1 the high level decomposition is shown. In addition to the decomposition, the responsibilities are included in the diagram.

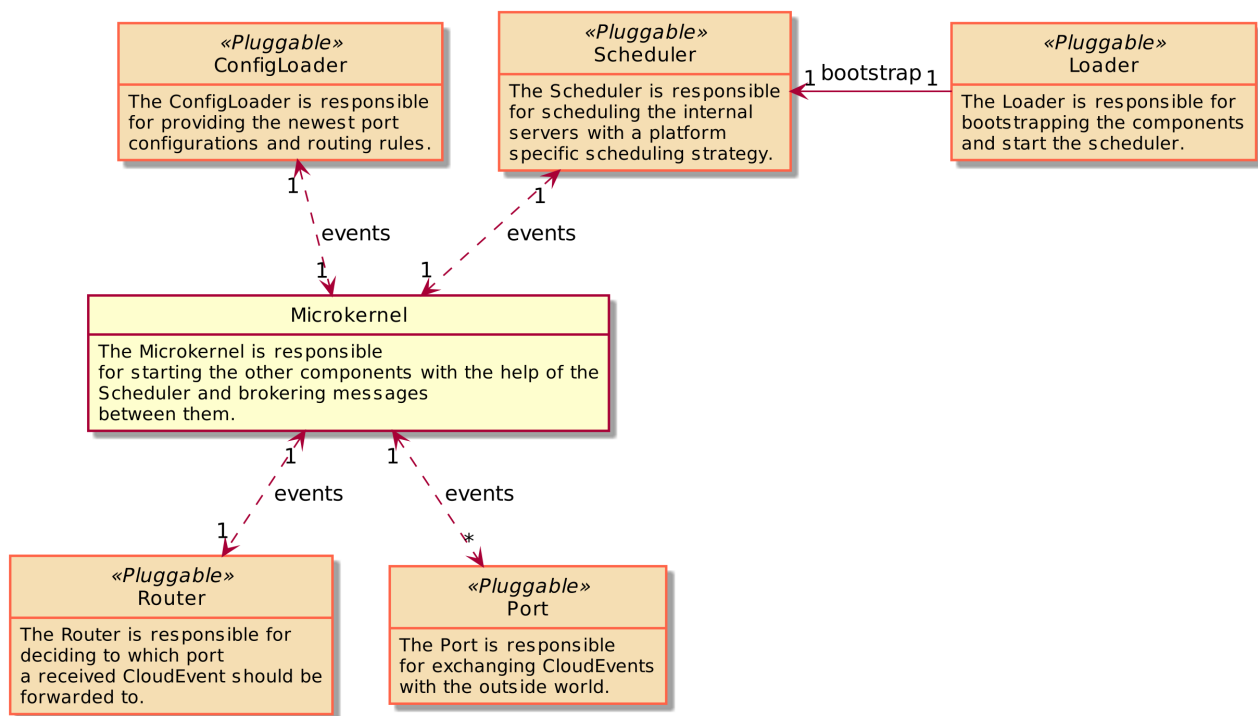


Figure 5.1: Building Blocks with their Responsibilities

5.1 Microkernel Events

Table 5.1 gives an overview over all events which are exchanged between the components.

Version ¹	Name	Description
1.0	Init	The Init event indicates to the receiver that it should start interacting with the outside world. The event is produced by the kernel when all components are scheduled.
1.0	ScheduleInternalServer	The ScheduleInternalServer event tells the Scheduler to schedule a new internal server. The event is produced by the kernel, one event for each component.
1.0	InernalServerScheduled	The InernalServerScheduled event indicates to the receiver that a new internal server was successfully scheduled. The event get produced by the scheduler, after a component was scheduled (because of a ScheduleInternalServer event), the receiver is the kernel.
1.0	IncommingCloudEvent	The IncommingCloudEvent event indicates to the receiver that a new CloudEvent has been received from the outside world. The event is produced by an input port and is sent to the kernel. The kernel sends the same Event to the router.
1.0	OutgoingCloudEvent	The OutgoingCloudEvent event indicates to the receiver that a CloudEvents has been routed and is ready to be forwarded to the outside world. The event is created by the router, send to ² the kernel and then to the output adapter. An event is created for each outgoing port the message is routed to.
1.0	ConfigUpdated	The ConfigUpdated event indicates to the receiver that the config has changed and a configuration update should be applied. The event is produced by the router to the kernel and then to the component.
1.0	Batch	The Batch event can be used to make sure a collection of events are processed by the Microkernel in one batch to prevent race conditions.
2.0	RoutingResult	The RoutingResult is the result of a routing from one IncomingCloudEvent. The event is sent from the router to the kernel and there forwarded as OutgoingCloudEvent to the ports.

²Starting in Version 2, the router responds with a single RoutingResult event. With this change, the kernel can keep track of the number of outgoing events per routed event. This is necessary for providing an "At Least Once" delivery guarantee.

Version ¹	Name	Description
2.0	OutgoingCloudEventProcessed	The OutgoingCloudEvent event was processed. The OutgoingCloudEventProcessed event notifies the kernel about the end of the processing and indicates whether the outcome was successful. This response should only be sent if the OutgoingCloudEvent event indicates that an acknowledgment is required.
2.0	IncomingCloudEventProcessed	The IncomingCloudEvent was processed. The IncomingCloudEventProcessed notifies the receiver port that the routing is completed and an acknowledgment to the sender can be sent. This response should only be sent if the IncomingCloudEvent event indicates that an acknowledgment is required.
2.0	HealthCheckRequest	A health check port sends HealthCheckRequest events to some components. The components should respond with a HealthCheckResponse. The response to the health check request is an indicator of the health state of the router. No answer will be rated as unhealthy.
2.0	HealthCheckResponse	The event is a response for the HealthCheckRequest event. It should go to a health check component.

Table 5.1: Microkernel Events; The events that are exchanged between the components

¹Version of the project in which the event was introduced

6 Runtime View

In this chapter we will describe the interactions of the building blocks during five runtime scenarios. They are sorted in the order of their occurrence during normal operations. Some scenarios can happen multiple times. Every section describes one runtime scenario, at the end all scenarios are shown combined in Figure 6.7.

6.1 Runtime Scenario 1: Loading and Bootstrapping

The rust `main` function should call the `start` function of the loader (Figure 6.1). The loader loads the needed components and passes them to the `bootstrap` function.

The `bootstrap` function call initiates the start sequence of the application. The `bootstrap` function starts the scheduler, which then schedules the Microkernel.

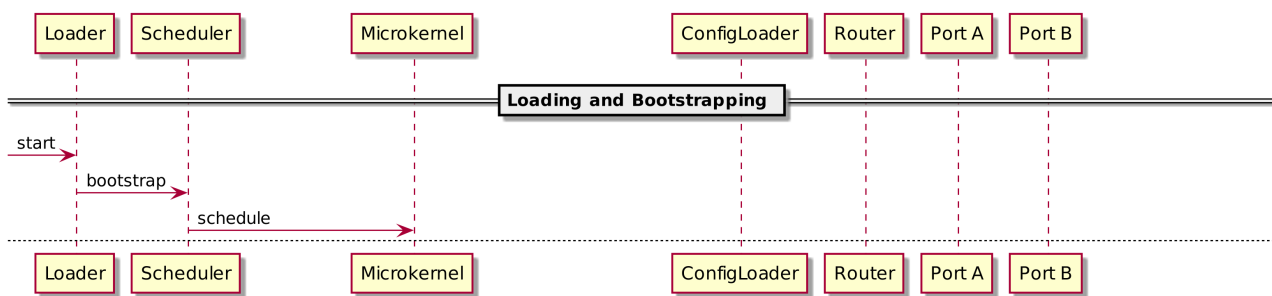


Figure 6.1: Runtime Scenario: Loading and Bootstrapping

6.2 Runtime Scenario 2: Initialization

After the bootstrap sequence ended, the Microkernel schedules all the other building blocks with the help of the Scheduler (Figure 6.2). The list of the components to schedule is provided by the `start_options`.

For each component the Microkernel sends an event to the Scheduler. The Scheduler then schedules the component. A handle for the inbox channel of the Microkernel is provided as a parameter. With this handle the new component can send messages to the Microkernel. After the component is scheduled, the Scheduler sends an event back to the Microkernel to let it know that the component is scheduled. In this message, a handle for the inbox channel of the scheduled component is provided. With this handle the Microkernel can send messages to the component.

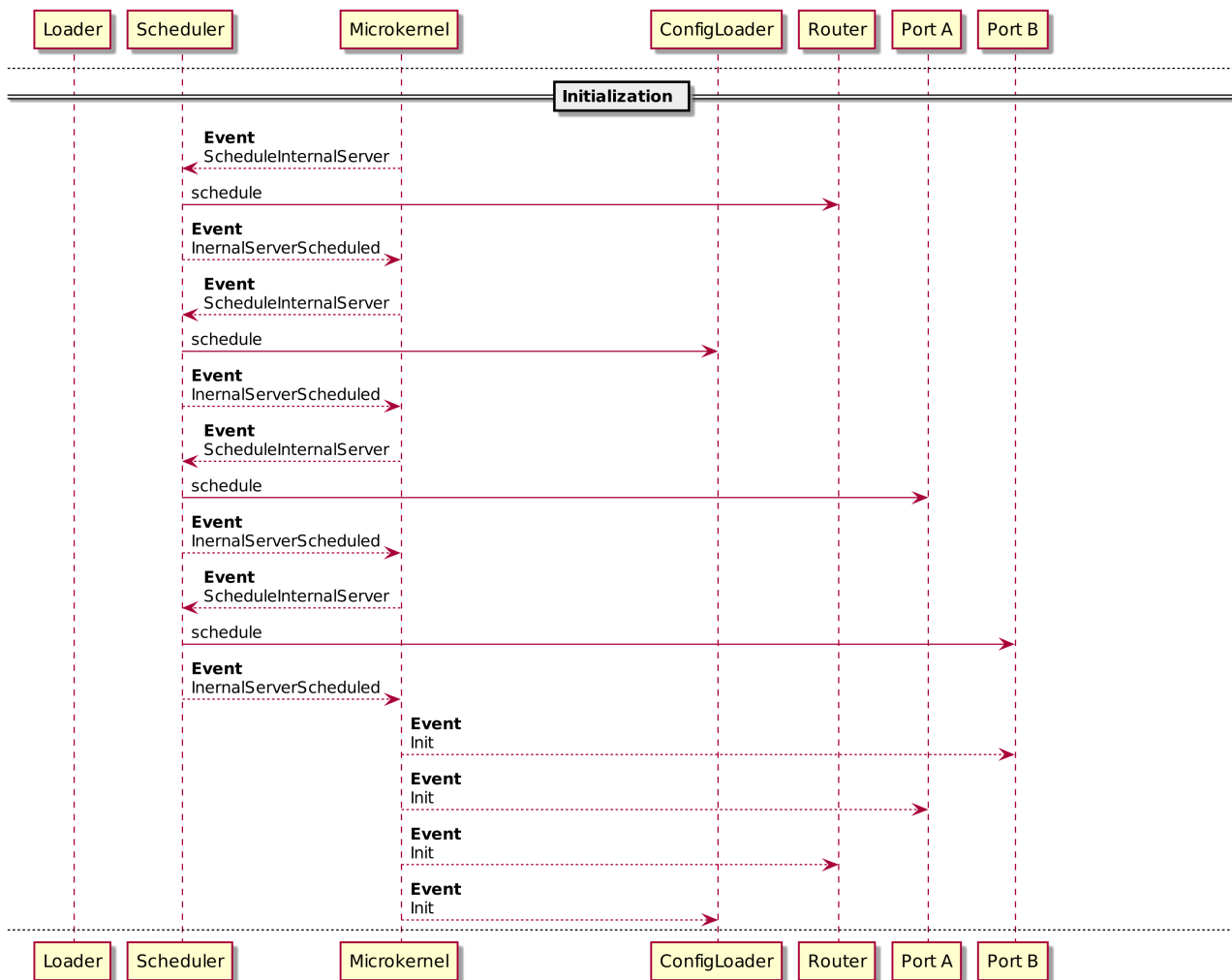


Figure 6.2: Runtime Scenario: Initialization

6.3 Runtime Scenario 3: Initial Configuration

In the scenario shown in Figure 6.3 the initialization has already happened. Now the ConfigLoader loads the configuration and sends it to the Microkernel. The Microkernel forwards the received configuration to the relevant component.

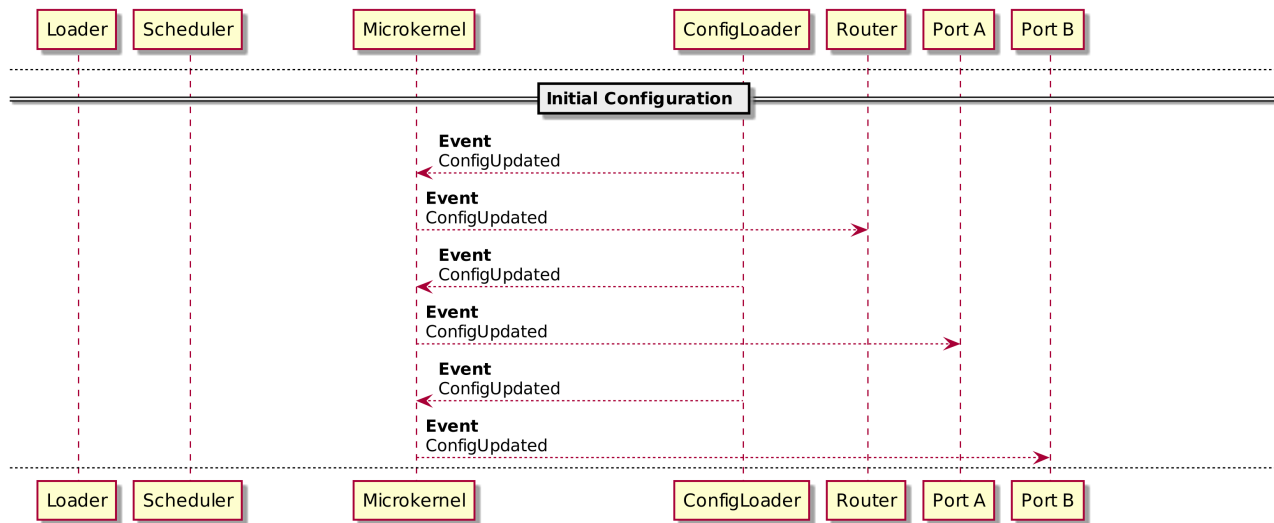


Figure 6.3: Runtime Scenario: Initial Configuration

6.4 Runtime Scenario 4a: Routing with "Best Effort"

"Best Effort" is the contrast to any guarantee. In this context "Best Effort" means that none of the defined delivery guarantees are claimed.

Figure 6.4 shows the routing scenario with "Best Effort" delivery. One of the ports receive a CloudEvent from the outside world. This port deserializes and forwards the event to the Microkernel which forwards it again to the router. The router decides, based on the configuration it received earlier, to which ports the event should be routed to. After the router has decided the event gets sent to the designated ports via the Microkernel. The ports serialize and forward the event to the outside world.

The scenario in Figure 6.4 would look exactly the same for "At Most Once". However, in "At Most Once" there is an additional guarantee, that no message duplication happens. This is not guaranteed in the "Best Effort" routing.[18]

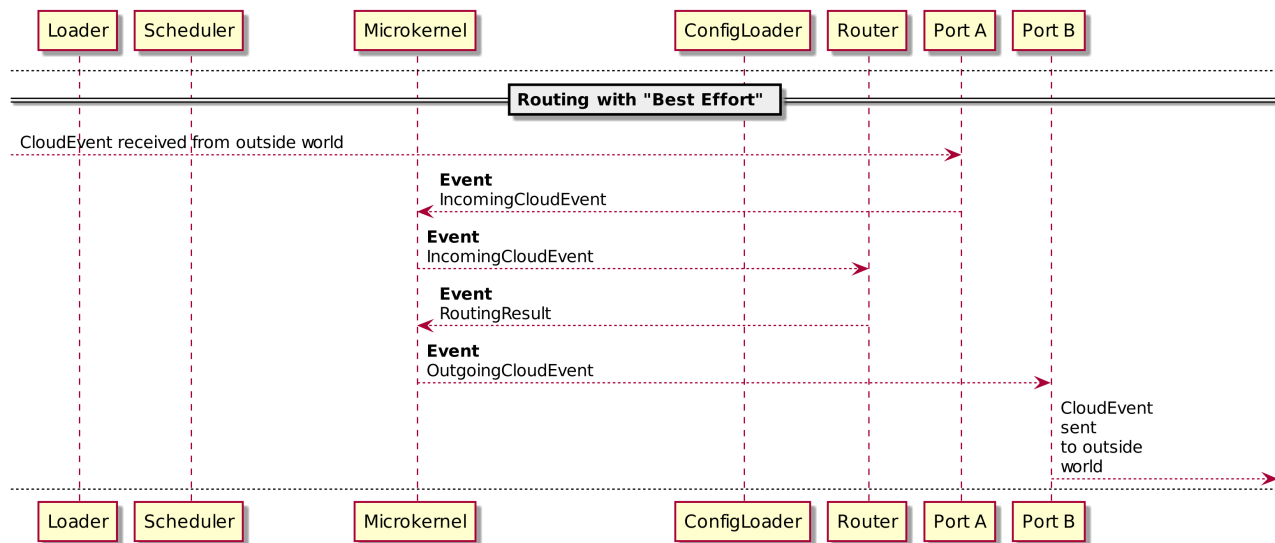


Figure 6.4: Runtime Scenario: Routing with "Best Effort"

6.5 Runtime Scenario 4b: Routing with "At Least Once"

To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for a message in a certain time frame, the message is retransmitted.[22]

Figure 6.5 shows the routing scenario with an "At Least Once" delivery guarantee. Port A receives a CloudEvent from the outside world but does not acknowledge it yet. This port deserializes and forwards the event to the Microkernel, which forwards it again to the Router.

The Router decides, based on the configuration it received earlier, to which ports the event should be routed to. The routing result is sent to the Microkernel that extracts the information about the delivery guarantee. In this case, the event has an "At Least Once" delivery guarantee. The Microkernel stores metadata about the event, such as the original sender port and the recipient port list. Then it forwards the event to the recipient ports.

The recipient ports serialize and forward the event to the outside world. They report the success or failure of the delivery back to the Microkernel (`OutgoingCloudEventProcessed`).

The Microkernel checks the result and, if the delivery was successful, removes the port from the pending recipient list. When the pending recipient list is empty, then the Microkernel informs the sender port (Port A) about the successful event delivery (`IncomingCloudEventProcessed`).

Port A receives the result from the Microkernel and acknowledges the message.

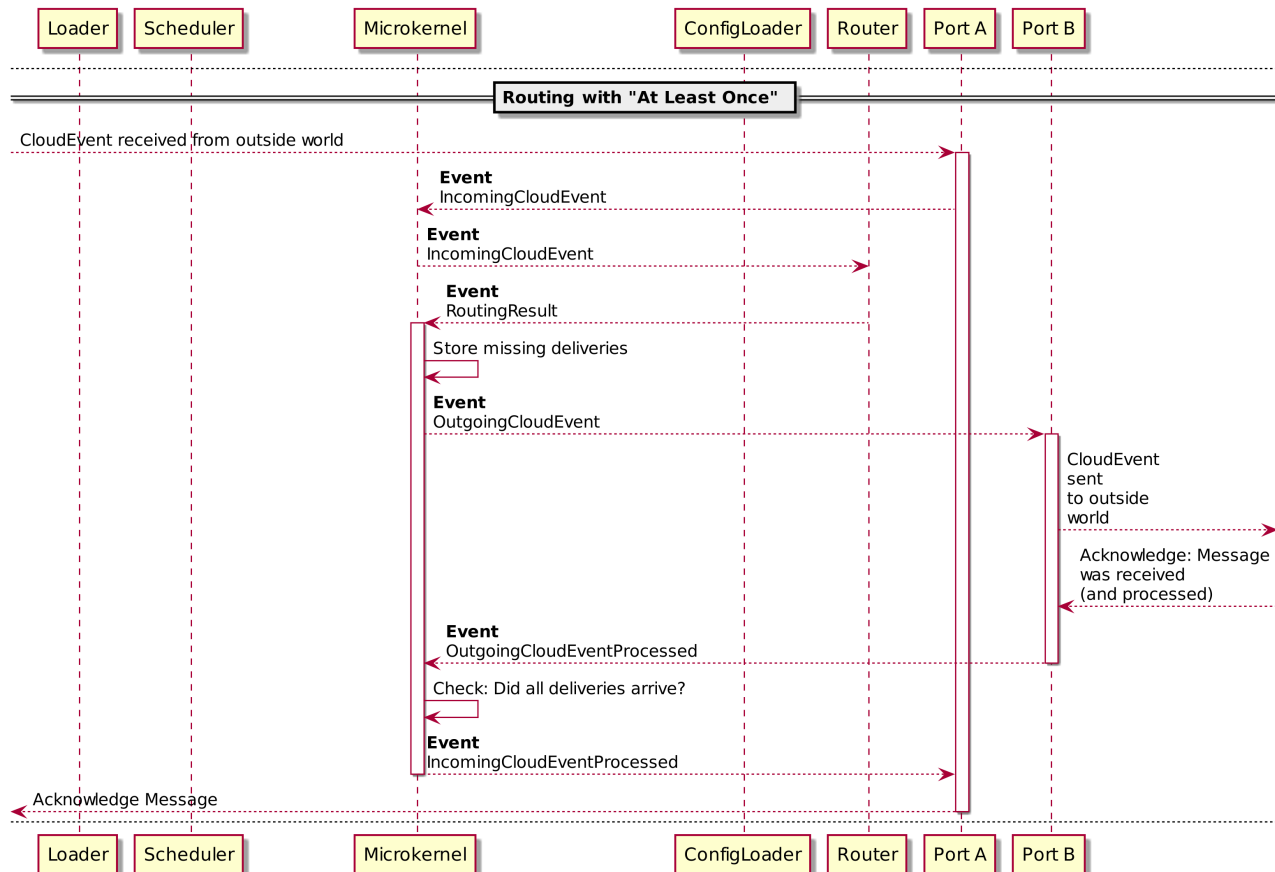


Figure 6.5: Runtime Scenario: Routing with "At Least Once"

6.6 Runtime Scenario 5: Configuration Update

The ConfigLoader can receive configuration updates during runtime from the outside world as shown in Figure 6.6. When the ConfigLoader receives an update, it sends the update to the affected port or router. The affected port or router reconfigures itself and acts according to the new configuration.

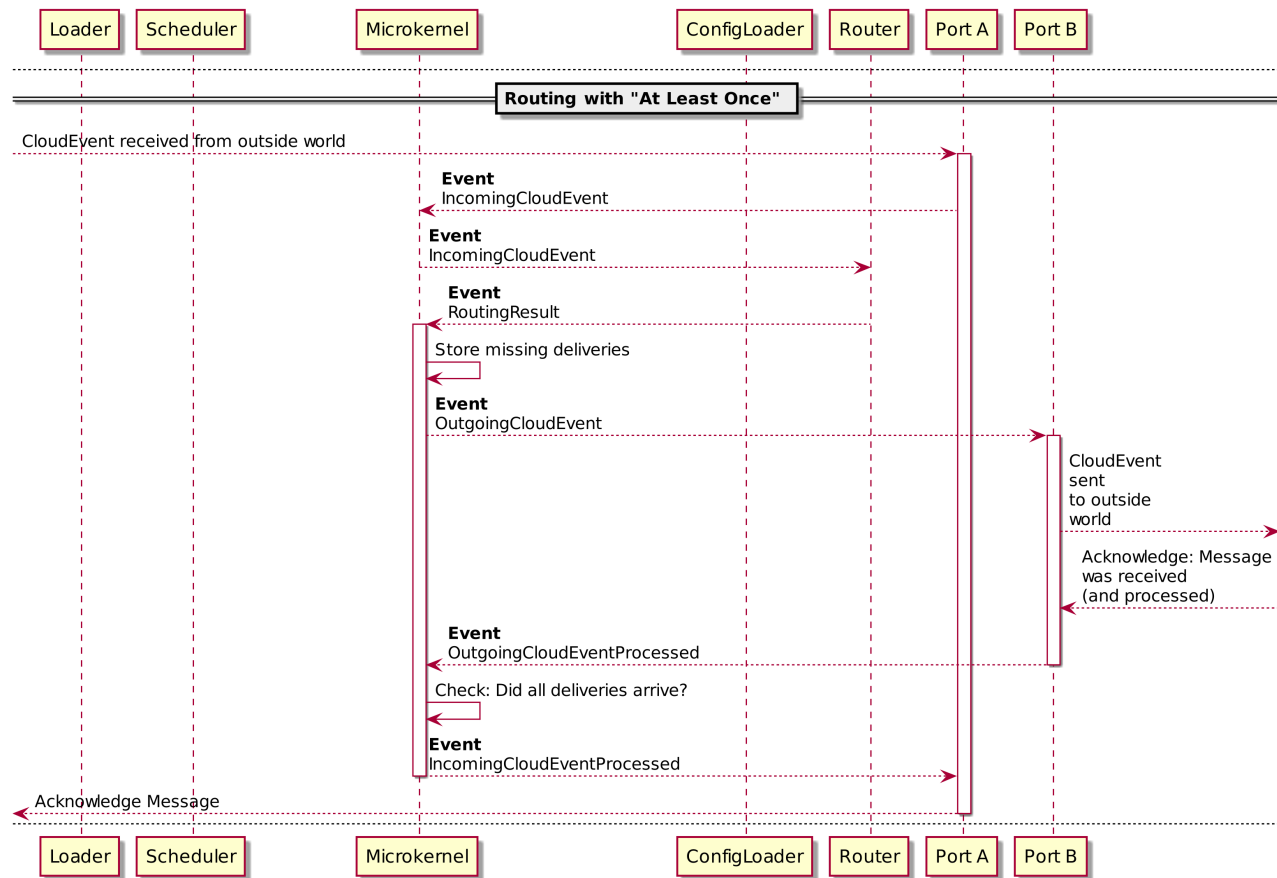
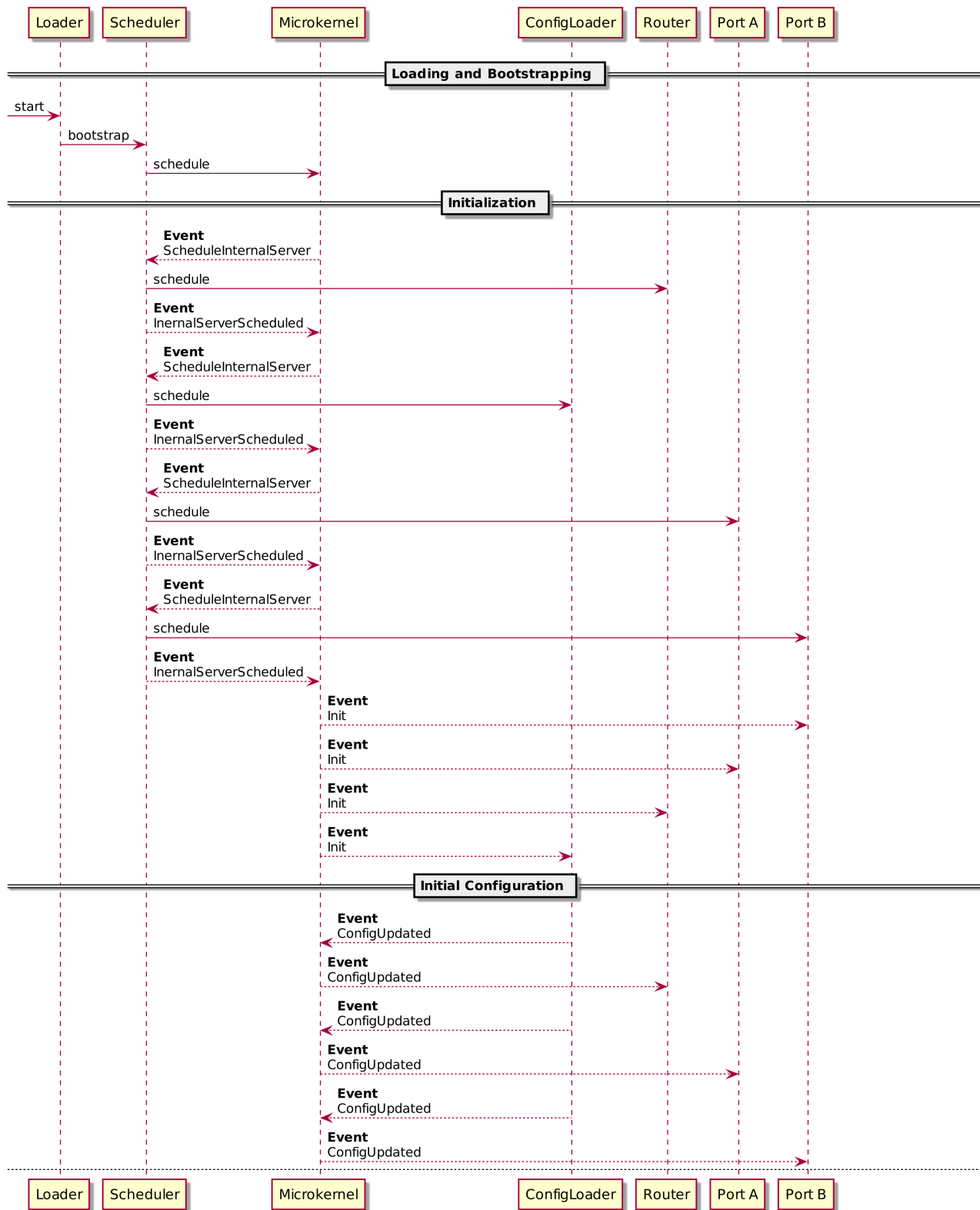


Figure 6.6: Runtime Scenario: Configuration Update



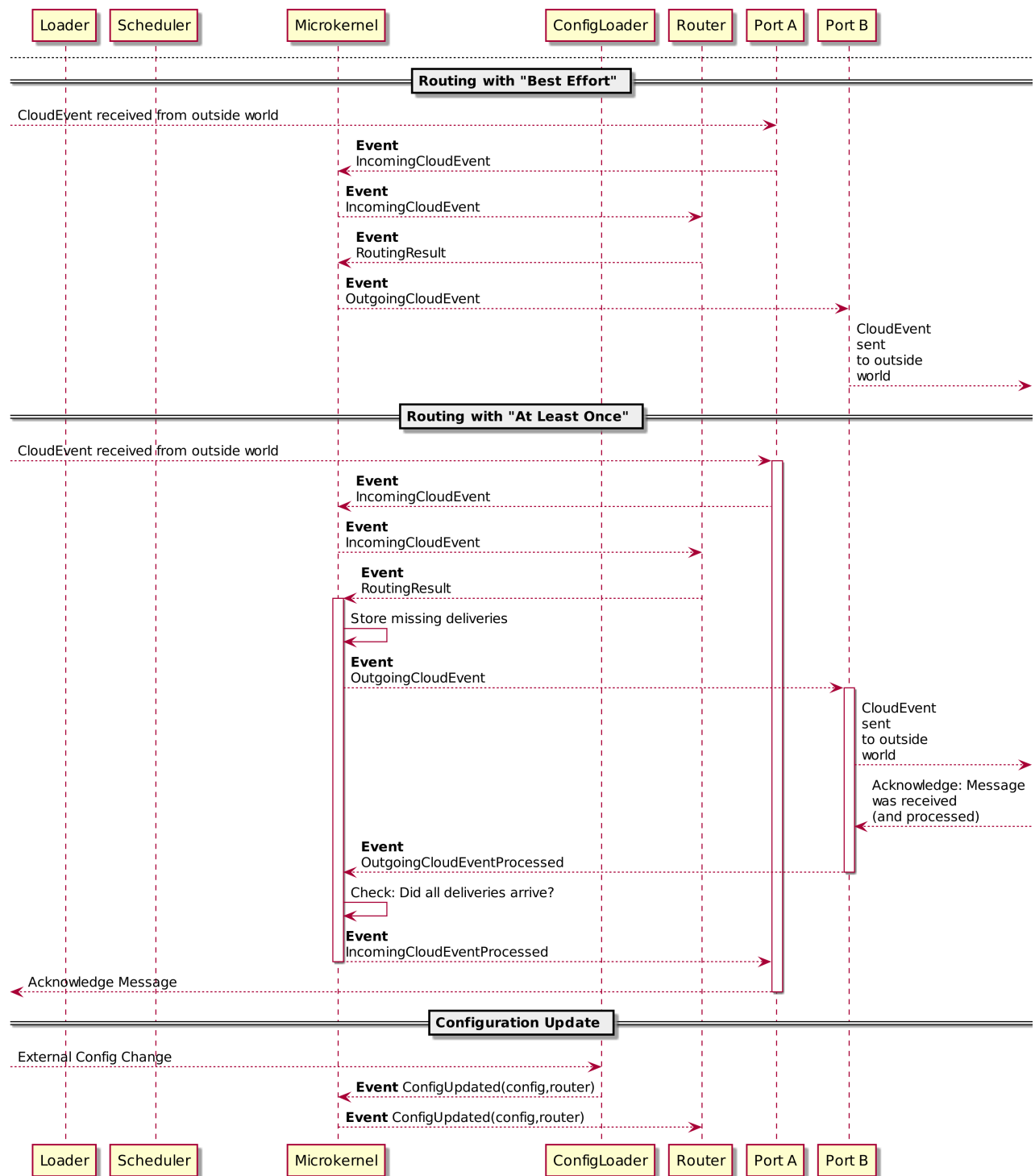


Figure 6.7: Runtime Scenarios showing the interaction between the components of CERK and the outside world

7 Deployment View

The detailed deployment of the CloudEvents Router is not part of the scope of this project. The only guardrails to keep in mind during development are the platforms CARU intends to use the router on.

7.1 Target Platforms at CARU

The following subsections provide an overview of the platforms used by CARU. The router should be able to run on the main processor of the CARU Device and in the cloud. The co-processor of the CARU Device is not the main target and the router may never be used on it.

7.1.1 CARU Device

The CARU Device is a product placed in the living environment of an elderly person to collect various room parameters, learn the behaviors of the inhabitant and alarm relatives or caretakers in case of deviations. Its user experience is optimized for elderly people. It was formerly known as CARU SmartSensor.[23]

The device has two central processing units (CPUs) with two different operating systems on it.

Main Processor of the CARU Device

The CARU Device main processor has limited capabilities:

CPU Architecture ARM Cortex-A7 (armv7l)

CPU Cores 1

CPU Threads 1

CPU Clock 198 - 528 MHz

RAM 500 MB

Operating System Linux

Co-Processor of the CARU Device

The CARU Device low power co-processor has limited capabilities:

CPU Architecture ARM Cortex-M4 (Armv7E-M)

CPU Cores 1

CPU Threads 1

CPU Clock 80 MHz

RAM 160 KB

Operating System FreeRTOS

7.1.2 Virtual Machine in the Cloud

The virtual machines are hosted by AWS and have almost unlimited capabilities[24]:

CPU Architecture ARM, Intel, AMD

CPU Cores 1 - 224

CPU Threads 1 - 448

CPU Clock 2.3 GHz - 4.0 GHz

RAM 500 MB - 24 TB

Operating System Linux, Windows

8 Cross-Cutting Concepts

8.1 Safety and Security Concepts

In this section, we describe some security measures that are in place and some known shortcomings.

8.1.1 Starting Position

The CERK is at the moment only designed to be run in a secure environment, without any connections over the public internet.

If the router should communicate over the public internet, first support for secure connections (e.g., Transport Layer Security (TLS)) connection for all used ports must be added. Currently, neither the MQTT port, nor the Advanced Message Queuing Protocol (AMQP) port are using a secure connection.¹

8.1.2 Minimize the Attack Surface Area

The following list shows some identified attack surfaces of the router and how they are or could be prevented:

Operatingsystem The only ready-to-use deployment that the project offers is a Docker deployment. To minimize the attack surface a "Distroless" base image is used.² The "Distroless" Docker Images is an image provided by Google, which contains only the most essential parts. There are no shells or other tools included that are not required for a productive deployment.[25]

Software Supply Chain (Cargo) We used Cargo to manage and install dependencies. Installing dependencies via a public registry and automatically updating them has become a growing risk in the last years.[26]

To mitigate this risk, multiple measures are in place:

- Cargo has a "yank" mechanism to flag package versions with security issues.³
- We use the "cargo-deny" tool to check for vulnerabilities in used crates.⁴ The command uses the "RustSec Advisory Database" to check against known vulnerabilities.⁵ However, we are only printing warnings and do not stop the CI on any findings. Here, space for improvement exists.
- We have integrated the GitHub Security Advisories and Dependabot security updates. If GitHub Security Advisories finds a vulnerability, the Dependabot will automatically create a pull request to update the affected dependency to a safer version.[27]

¹<https://github.com/ce-rust/cerk/issues/97>

²<https://github.com/GoogleContainerTools/distroless>

³<https://doc.rust-lang.org/cargo/reference/publishing.html#cargo-yank>

⁴<https://github.com/EmbarkStudios/cargo-deny>

⁵<https://github.com/rustsec/advisory-db>

Buffer Overflow Rust has secure memory allocation by design.[28] However, the underlying C libraries (used in some ports) can suffer from a buffer overflow. There, we have to ensure that only well-established libraries are used and that they are up to date.

Logs One potential security risk, is the information contained in the logs of the router. They give quite some details about the system and the data that passes through it. However, the detailed logs are mostly coming from the plugins and have to be fixed individually.

With the currently implemented log statements, the log level should not be set to debug in productive environments because the debug logs can contain sensitive data.

8.1.3 Handling Security Issues

Security issues should be reported and discussed in GitHub issues. With that measure they are documented and comprehensible.

However, if the security problem opens an attack surface, they should be discussed privately in the GitHub "Security Advisories" section.

8.2 Architecture Patterns

8.2.1 Microkernel Pattern

"The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration."[29]

We took the pattern description from "Pattern-Oriented Software Architecture" as reference.

The Microkernel pattern defines five kinds of participating components: internal servers, external servers, adapters, clients and the Microkernel itself. It is the Microkernel's task to provide the core mechanisms, offer communication facilities and manage the resources. Internal servers implement additional services and encapsulate some system specifics. External servers provide programming interfaces to its clients. Both, internal and external servers, collaborate with the Microkernel component. In addition to these internal components, the pattern also defines clients and adapters. Clients are separate applications that consume services from the Microkernel. An adapter is code that is part of the client, but is provided by the creators of the Microkernel. Nowadays, adapters are often called Software Development Kits (SDK). SDKs help the developers to interact with external server's APIs by adding an abstraction layer to it.[29]

The only significant difference between our interpretation of the Microkernel Pattern and the description from the authors is that we merged the internal and external servers. We call both of them internal servers because, in our context, we did not see any differences in how we interact with them. Treating them the same but naming them differently seemed unnecessary to us.

8.2.2 "At Least Once" Delivery Guarantee

To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for a message in a certain time frame, the message is retransmitted.[22]

Figure 8.1 shows the happy case (Default Scenario) and three error cases. More details about the pattern and how it can be used in conjunction with other patterns can be found in our paper "Reliable Messaging – Patterns and Strategies for Message Routing".[18]

"At Least Once" delivery

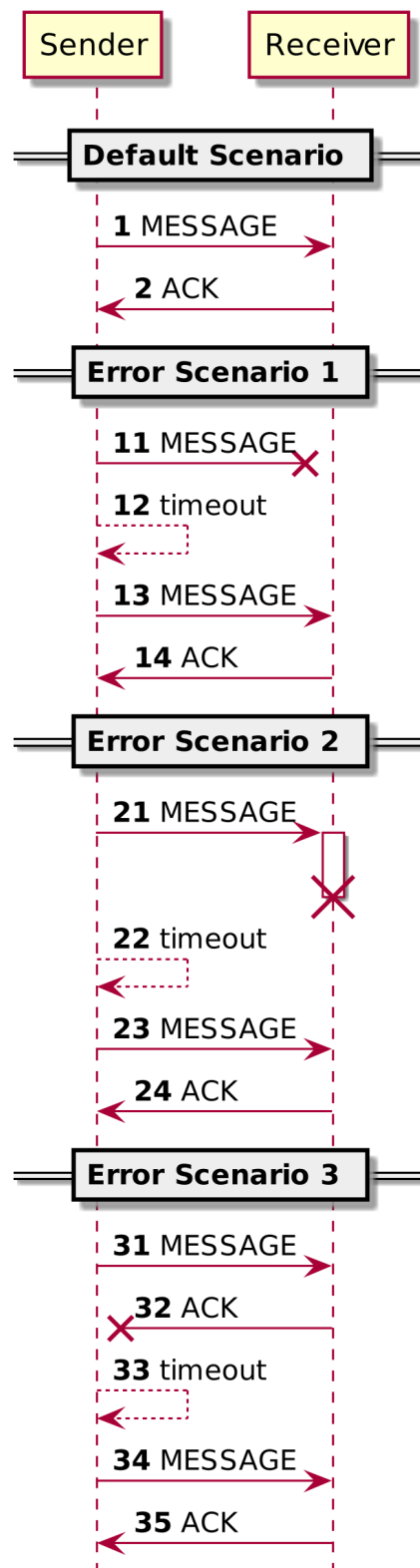


Figure 8.1: Architecture Patterns: At Least Once

8.3 Implementation Rules

Rust has build in macros to check the code for different aspects while compiling. We used the `missing_docs` check as macro `#![deny(missing_docs)]`. This macro requires the programmer to add a documentation to every public element. If the programmer does not add a documentation, the code will not compile.[7]

The CI compiles the product on every Git push to GitHub. A successful CI pipeline run is required to finish a pull request. So it is not possible to merge a undocumented pice of library code to the master.

We also require that the code is well formatted. This is done with `cargo fmt`. Sadly this is not tested in the CI pipeline, yet. The pull request reviewer have to check this manually.

9 Architecture Decisions

9.1 ADR-001: MADR for Architecture Decision Documentation

Status accepted

Date 03.11.2019

Context and Problem Statement

We need to document our architecture in a way that a reader understands our thought process behind it.

Decision Drivers

lean The documentation should be lean.

simple The documentation should be easy to understand.

complete The documentation should reveal the thought process which lead to the decisions.

Considered Options

ADR Architecture Decision Record[30]

MADR Markdown Architectural Decision Records (version 2.1.2)[31]

Y-Statement Sustainable Architectural Decisions[32]

Decision Outcome

We decided to use Markdown Architectural Decision Records (MADR) because of the dedicated section about the comparison of the considered options, the good documentation and open license (CC0).

Because we are already using \LaTeX for this document we will only use the structure of MADR and use it with \LaTeX instead of Markdown.

Positive Consequences

- Our architecture decision documentation is structured.
- Our architecture decisions are reconstructable.
- Architecture documentation and decisions are in the same document.

Negative Consequences

- MADR related tooling not usable because we are using it with \LaTeX .
- Architecture decisions and code in different repositories.[33, 34]
- The \LaTeX format is not as easy to publish as a webpage as Markdown.

Pros and Cons of the Options

Option MADR

good considers context, decision and consequences

good considers and compares alternative solutions

good specific instructions (GitHub repository[31])

good open license (CC0)

Option ADR

good considers context, decision and consequences

good good documentation (blog post[30])

bad lacks consideration of alternative solutions

bad no license

Option Y-Statement

good considers context, decision and consequences

good considers alternative solutions

bad distributed documentation (article[32], presentation slides[35], third party summary[36])

bad too compact (in the context of this thesis)

bad no license

References

- <http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions/>
- <https://github.com/adr/madr/tree/2.1.2/>
- <https://www.infoq.com/articles/sustainable-architectural-design-decisions/>

9.2 ADR-002: Rust as the Programming Language

Status accepted

Date 26.10.2019

Context and Problem Statement

We need to select a programming language for the implementation of our solution.

Decision Drivers

customer The customer has a strong preference for Rust because he wants to gain experience with it.[5]

portability The programming language should support multiple platforms. Especially support of resource restricted platforms like the Cortex-M4 processor with FreeRTOS is a wish of the customer. (see NFR-2)

footprint The programs produced by the programming language should have a low resource footprint because of the resource constraint target environments.[5]

stability The produced program should run for a long time without a memory leaks or crashes.

Considered Options

Rust The Rust programming language[37]

C/C++ The C and C++ programming languages[38, 39]

Go The Go programming language[40]

Python The Python programming language[41]

Decision Outcome

We decided to use Rust because the programs it creates have a small footprint and can run on various platforms (including bare metal and FreeRTOS[42, 43]). Its unique "ownership model" and the preferences of the customer gave it the edge over C/C++.

Positive Consequences

- Rust has a small footprint because it has no runtime or garbage collector.[44]
- "Rust's rich type system and ownership model guarantee memory-safety and thread-safety"[37]
- Rust can be compiled for many platforms, including bare metal and FreeRTOS.[42, 43]
- Rust is preferred by the customer.[5]

Negative Consequences

- Rust is new for us and we have no experience with it.
- Rust is a low-level programming language and we have not a lot of experience with low-level programming languages.
- Rust is a young programming language with a smaller ecosystem than others options.

Pros and Cons of the Options

Option Rust

good statically typed

good the "Rust ownership model" guarantees memory-safety and thread-safety[37, 45]

good can be compiled to be run without an operating system[42]

bad young language with small ecosystem (when compared to C/C++/Python)

bad no experience in the team

bad the "Rust ownership model" is unique and has a steep learning curve

Option C/C++

good statically typed

good can be compiled to be run without an operating system

good mature language with big ecosystem

good some experience in the team

good used by the customer

bad easy to make mistakes (memory leaks, stale pointers, ...)[46, 47]

bad old language with lots of legacy because of its backward-compatibility

Option Go

good statically typed

good simple multi-processing model

bad garbage collected

bad requires an operating system to run

bad young language with small ecosystem (when compared to C/C++/Python)

bad no experience in the team

Option Python

good can be run without an operating system (with some limitations)[48]

good experience in the team

good widely used by the customer

bad dynamically typed

bad interpreted language

bad garbage collected

References

- <https://www.rust-lang.org/>
- <https://isocpp.org/>
- <https://golang.org/>
- <https://www.python.org/>

9.3 ADR-003: Use of the Microkernel Pattern

Status accepted

Date 26.10.2019

Context and Problem Statement

We need to select an architecture pattern that allows our solution to support multiple platforms, with and without operating system, and a mechanism to add adapters.

Decision Drivers

extendability It should be easy to add new adapters. (see NFR-1)

portability The pattern should allow the implementation of platform-specific service. (see NFR-2)

efficiency The pattern should have as little overhead as possible.

Considered Options

Microkernel Microkernel pattern (see pattern summary in Section 8.2.1 "Microkernel Pattern")

Microservices Microservices pattern[49]

Plugin Plugin pattern[50, p. 499-503]

Decision Outcome

We decided to use the Microkernel pattern because it requires no inter-process communication and we do not need the scalability of the Microservice pattern.

The decision was fixed, after the first running Microkernel prototype were running.¹

Positive Consequences

- A Microkernel can be easily ported to different platforms.
- The overhead is low because everything is in one process.
- Core functionality can be shared between platform-specific implementations.

Negative Consequences

- A Microkernel is harder to scale than Microservices.

¹<https://github.com/ce-rust/architecture-prototype/commit/28db2968e01eaddb5b8a349f025387d13d178a9a>

Pros and Cons of the Options

Option Microkernel

good allows platform-specific implementations of services

good can run services in one process

good manages communication between services

bad limited scaling possibilities

Option Microservices

good allows platform-specific implementations of services

good different parts of the system can potentially run on different platforms at the same time

good scalability beyond a single machines

bad services are separate processes by definition

bad overhead of inter-process communication

Option Plugin

good allows dynamic loading of different implementations

good separate binary for program and plugins

bad depends on operating system

bad platform-dependent plugin format (so, dylib, dll)[51]

bad dynamic linking hell[52]

bad **unsafe** Rust code (The Rust compiler can not give its usual safety guarantees because the plugin is dynamically loaded)[53]

References

- see pattern summary in Section 8.2.1 "Microkernel Pattern"
- <https://martinfowler.com/articles/microservices.html>

9.4 ADR-004: Use of the Threading Model

Status accepted

Date 26.10.2019

Context and Problem Statement

We need a way to run multiple blocking operations concurrently to be able to implement the different adapters.

Decision Drivers

overhead The overhead of running blocking operations concurrently should be as low as possible.

compatibility We want to reuse existing libraries to implement the adapters.

performance We want to be informed as fast as possible when new data is available.

portability We want to be able to support multiple platforms.

Considered Options

Async Rust runs an event loop and schedules different parts of the program concurrently.[54]

Polling We check regularly if new data is available.

Threading Rust tells the platform to schedule different parts of the program concurrently.[55]

Decision Outcome

We decided to use the threading model because it allows us to outsource the scheduling to the platform and leverage the full library ecosystem of Rust. There is also the possibility to run non input/output-bound tasks in parallel on platform which support it.

Because the implementation of threads is platform-specific, our microkernel will provide an abstraction (called Internal Server) for it which needs to be implemented for each platform.

The decision was fixed, after the first running Microkernel prototype with a threading model was implemented and did run successfully.²

Positive Consequences

- We can reuse existing libraries.
- The platform takes care of scheduling and wakes threads (Internal Servers) automatically when new data is available.

²<https://github.com/ce-rust/architecture-prototype/commit/28db2968e01eaddb5b8a349f025387d13d178a9a>

Negative Consequences

- There is an overhead for switching between threads.
- The implementation of the threading model differs from platform to platform.
- Our microkernel needs to provide the interface for threads (Internal Servers) that can be implemented for the different platforms.

Pros and Cons of the Options

Option Async

good low overhead because no context switching required[54]

good high performance for io-bound tasks[54]

bad not supported on bare metal[56]

bad only libraries built for async can be used

bad async was a beta feature when we started with the project and not a lot of libraries support it. It is stable since Rust 1.39.0.[57, 58]

bad no parallelism possible

Option Polling

good platform independent

good any library can be used

bad very high overhead if we poll often

bad very low performance if we poll seldomly

bad no parallelism

Option Threading

good most platforms implement the threading model

good any library can be used

good good performance because the platform schedules the thread as soon new data is available

good good scalability on platform with parallel processing capabilities

bad implementation depends on platform

bad overhead because of the switching between thread contexts

References

- <https://rust-lang.github.io/async-book/>
- <https://doc.rust-lang.org/std/thread/>

9.5 ADR-005: Use of the Broker Pattern with Channels

Status accepted

Date 16.10.2019

Context and Problem Statement

Because we decided that we will be using a threading based approach, we need to define how the microkernel's Internal Servers (threads) communicate with each other.

Decision Drivers

idiomaticity The way the Internal Servers communicate should be idiomatic for Rust

overhead The communication should have a small overhead.

Considered Options

Broker with channels a central broker which receives and forwards data over channels[59, 60]

Broker with invocations a central broker which coordinates invocations between Internal Servers[61]

Mesh with channels Internal Servers are directly connected to each other over channels[60]

Mesh with invocations Internal Servers directly invoke each other

Decision Outcome

We decided to use a broker with channels for the communication between Internal Servers because it is the most idiomatic way for Rust.

The implementation of the channel is, like the threading implementation, platform-specific, our microkernel will provide an abstraction for channels (called Channel) which needs to be implemented for each platform.

Positive Consequences

- Channels are the idiomatic way for sharing data between threads in Rust.[62]
- No explicit locking needed.
- Decoupling of the Internal Servers (they only need to know the channel to the broker).

Negative Consequences

- The broker can become a bottleneck.
- Overhead of communicating over the broker.
- The implementation of channels differs from platform to platform.
- Our microkernel needs to provide the interface for channels that can be implemented for the different platforms.

Pros and Cons of the Options

Option Broker with Channels

good channels are idiomatic for Rust[62]

good channels do not need explicit locking

good decoupling (only the channel to the broker needs to be known)

bad communication over the broker is overhead

bad broker can become the bottleneck

bad channels are platform specific

bad request/response scenarios are harder to implement because it is message based

Option Broker with Invocations

good decoupling (only the reference to the broker needs to be known)

good invocations make request/response scenarios easier to implement than channels

bad Internal Servers share reference of broker[63]

bad access to the broker needs to be explicitly secured with locks[64]

bad communication over the broker is overhead

bad the broker can become the bottleneck

bad locks are platform specific

Option Mesh with Channels

good channels are idiomatic for Rust[62]

good channels do not need explicit locking

good no overhead because of indirect communication

bad coupling (channels to all other Internal Servers need to be known)

bad channels are platform specific

bad request/response scenarios are harder to implement because it is message based

Option Mesh with Invocations

good invocations make request/response easy

good no overhead because of indirect communication

bad coupling (references of all other Internal Servers need to be known)

bad Internal Servers share reference to each other[63]

bad access to the Internal Servers needs to be explicitly secured with locks[64]

bad locks are platform specific

References

- <https://doc.rust-lang.org/std/sync/index.html>

9.6 ADR-006: "At Least Once" as the next delivery guarantee to implement

Status accepted

Date 08.11.2020

Context and Problem Statement

The router should be able to deliver messages to the destination(s) without losing any message.

We compared the possible patterns and solution strategies for the different approaches to achieve delivery guarantee in a separate research phase during the "Reliable Messaging using the CloudEvents Router".[2] The output of the research were two papers: "Reliable Messaging – Patterns and Strategies for Message Routing" and "Protocol Interoperability in a Stateless Message Router".[17, 18]

Decision Drivers

stateless The router should not have to save any state. It should be stateless so that it could easily be deployed in a cluster and not have any problems with corrupt data.

interoperability The offered delivery guarantee semantics needs to be protocol-independent so that the delivery guarantee can be provided by different ports and protocols.

overhead The overhead of the delivery guarantee should be as small as possible. Especially in the context of network traffic and memory usage. This is important on embedded devices.

Considered Options

"At Least Once" To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for a message in a certain time frame, the message is retransmitted.[22]

"Exactly Once" In "Exactly Once" delivery, the middleware only transfers the message once to the consumer. The message exchange should be reliable, as a retransmit is not possible if an acknowledgment is missed. To fulfill that, often a transactional message exchange is used.[22] However, "Exactly Once" is challenging to implement, if not even impossible. It depends on the exact definition.[22, 65, 66] Mostly it is more economical and enough to handle "At Least Once".[65]

Decision Outcome

We decided to implement "At Least Once" and to not implement "Exactly Once".

Positive Consequences

- The router will offer guaranteed delivery
- Messages can be routed between different protocols while preserving an "At Least Once" delivery guarantee.[17]
- The router will still be stateless.[18]

Negative Consequences

- Consumers with the requirement of an "Exactly Once" delivery guarantee can not be connected directly to the router or to a channel the router routes messages to. A message de-duplication filter has to be deployed inbetween.[18]

Pros and Cons of the Options

Option "At Least Once"

good We showed in "Reliable Messaging – Patterns and Strategies for Message Routing" that "At Least Once" can be used as a good and protocol-independent solution. "At Least Once" can be implemented without requiring a stateful router.[18]

good In "Protocol Interoperability in a Stateless Message Router", we showed that the ways different protocols implement "At Least Once" message delivery is interoperable.[17]

bad Legacy software that requires the consumption of messages with an "Exactly Once" guarantee will not work out of the box. As a work around a message filter could be implemented to filter duplicates between the router and the "Exactly Once" consumer.[18]

Option "Exactly Once"

good Any consumer can consume messages that are published with an "Exactly Once" delivery guarantee semantics. A message transferred with an "Exactly Once" semantics can also be consumed by a consumer which is expect an "At Least Once" or an "At Most Once" semantics without braking it.

good The definition of "Exactly Once" is the easiest and most intuitive one at the surface.

bad An implementation of a router that offers an "Exactly Once" delivery guarantee requires the router to be stateful and introduces significant overhead in network traffic and memory consumption when compared to "At Least Once" or "Best Effort".

bad Not all protocol offer an "Exactly Once" delivery guarantee and the ones that do implement it use significantly different strategies which make protocol interoperability in general impossible.[17]

References

- L. Basig and F. Lazzaretti, Reliable Messaging – Patterns and Strategies for Message Routing, research rep., 2021[18]
- L. Basig and F. Lazzaretti, Protocol Interoperability in a Stateless Message Router, 2021[17]

10 Quality Requirements

section 10.1 describes the quality requirements with help of the quality tree. The section 10.2 add more details to the non-functional requirements.

10.1 Quality Scenarios and Quality Tree

Figure 10.1 shows our quality tree with the quality scenarios as leafs. The quality tree is included for completeness and is not scope defining. The quality scenarios should be kept in mind while designing the product.

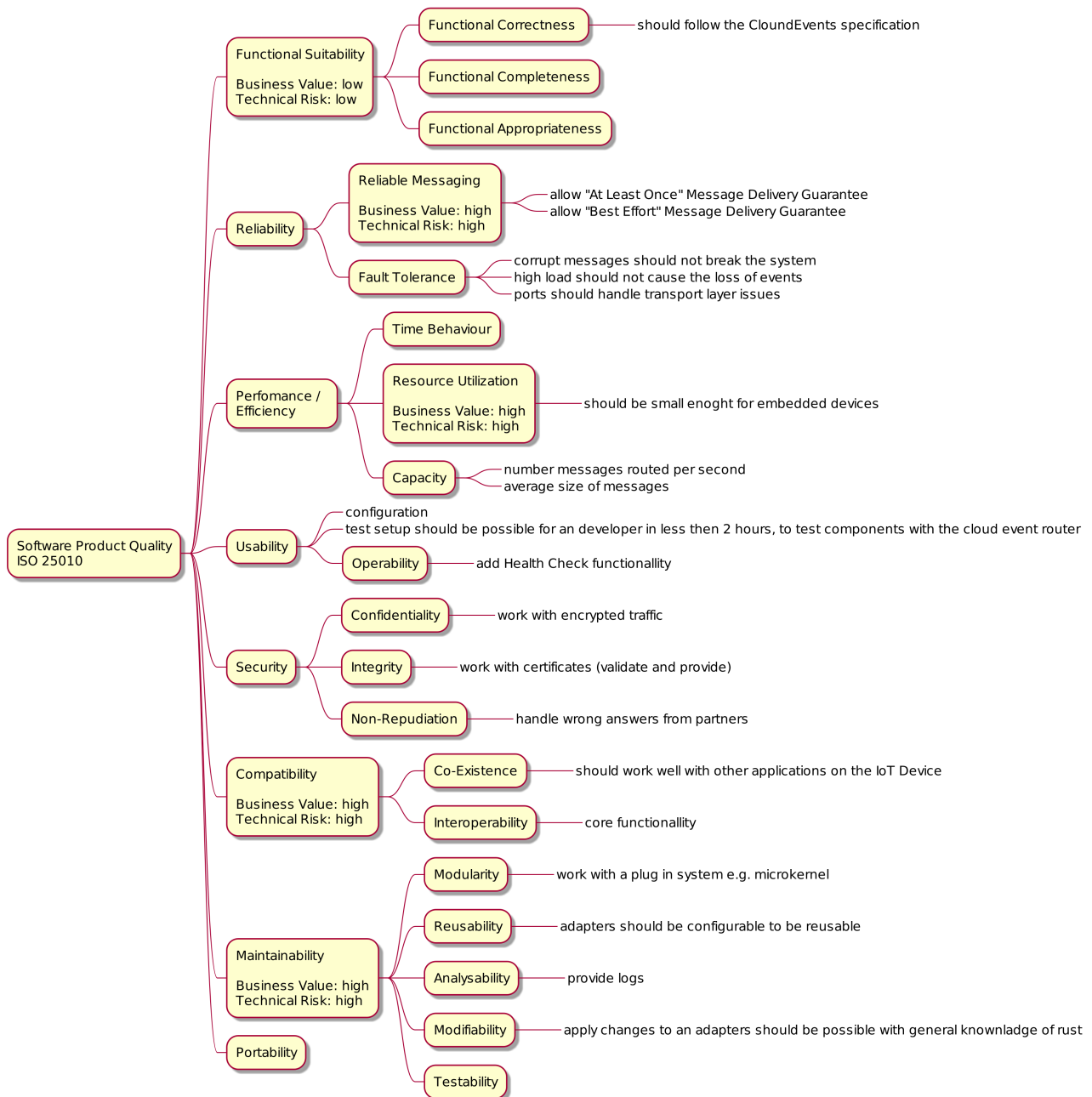


Figure 10.1: Quality Tree according to arc42 and ISO 25010[67, 68]

10.2 Landing Zones

The primary landing zones add more details to the non-functional requirements defined in section 1.2. There are also secondary landing zones which do not have a corresponding non-functional requirement in section 1.2. These landing zones are less important than the ones with a corresponding non-functional requirement. Table 10.1 lists our landing zones with their minimum, target and outstanding result.

Version ¹	No	NFR	Description	minimum	target	outstanding
1.0	LZ-1	NFR-2	Supported operating systems	Ubuntu Linux	Embedded Linux	FreeRTOS
1.0	LZ-2	NFR-3	Easy findable and followable documentation to setup the project and run it.	README document exists	Documentation exist and contains clear setup and run documentation.	Multiple tutorials for different adapter setups are provided.
1.0	LZ-3	-	Events processed on a normal computer	5/s ²	500/s ³	1500/s ⁴
1.0	LZ-4	-	Average event size	1 KByte	64 KByte ⁵	64 KByte

Table 10.1: Landing Zones

¹Version of the project in which the landing zone was introduced

²Current number of events processed per second on the CARU Device

³Estimated number of events processed in the cloud by the end of the year

⁴Expected number of events processed in the cloud per second for 1000 devices

⁵Maximal size the CloudEvents specification allows[69]

11 Risks and Technical Debts

In this chapter, the known and most critical technical debts and risks are listed. In addition to the following sections also a list of issues is maintained on GitHub: <https://github.com/ce-rust/cerk/issues>.

11.1 Single Unit of Mitigation

At the moment no advanced error mitigation concept is implemented; the whole router is one "Unit of Mitigation". This means the whole router shuts down if an error occurs in any component.

Update Version 2.0 A Health Check concept was introduced to monitor the health of the router. With the new delivery guarantee "At Least Once" a restart of the router is always possible. So if the router is unhealthy it could just be restarted. That mitigates the problem of one Unit Of Mitigation.

List of Figures

3.1	CARU's Envisioned System Context	16
3.2	CARU's Envisioned Communication on the Device	16
3.3	CARU's Envisioned Communication in the Cloud	17
4.1	The CI of CERK	22
5.1	Building Blocks	23
6.1	Runtime Scenario: Loading and Bootstrapping	27
6.2	Runtime Scenario: Initialization	28
6.3	Runtime Scenario: Initial Configuration	29
6.4	Runtime Scenario: Routing with "Best Effort"	30
6.5	Runtime Scenario: Routing with "At Least Once"	31
6.6	Runtime Scenario: Configuration Update	32
6.7	Runtime Scenarios	35
8.1	Architecture Patterns: At Least Once	41
10.1	Quality Tree	58

List of Tables

0.1	Document Changes	6
1.1	Functional Requirements	7
1.2	Nonfunctional Requirements	11
1.3	Stakeholders	12
5.1	Microkernel Events	25
10.1	Landing Zones	59

Glossary

There are some overlapping glossary entries between the documents of this thesis. Each document contains only the glossary entries which are referred to in that document.

Some glossary entries were originally created during the thesis "CloudEvents Router"[1]. However, because the context is similar they are inherited from there.

AWS IoT Core IoT Core is a service from AWS that helps with securely connecting Internet of Things (IoT) devices to the cloud. At its core is a serverless MQTT broker that integrates with other AWS services like message queues, databases and more.[70]. 15, 16

CloudEvents Subscription API The CloudEvents Subscription API is a vendor-neutral API specification that defines how consumers could subscribe events from a producer.[71]. see also CloudEvents & API

NATS Streaming NATS Streaming is an extension for the NATS messaging system that adds a "At Least Once" delivery semantics and message persistence.[72]. see also NATS

AWS Lambda AWS Lambdas is the FaaS offering from AWS. It is tightly integrated with the rest of the AWS ecosystem and allows the implementation of such functions in many programming languages, like Python, Go, or Java.[15]. 15, 16, see also AWS & FaaS

Academic Free License Academic Free License is a open-source license provided by the Open Source Initiative (OSI). It is not a copyleft license.[73]. 70, see also copyleft, OSI & open-source

Amazon Elastic Compute Cloud Amazon Elastic Compute Cloud (EC2) is a Virtual Machine (VM) solution from AWS.[74]. see also VM & AWS

AMQP The Advanced Message Queuing Protocol (AMQP) is an open-source messaging protocol with the goal of standardizing the message exchange between enterprises on the wire level. Before version 1.0, the AMQP specification also defined the behavior of the message broker. In version 1.0, the authors narrowed down the scope of the specification to the exchange of messages between two nodes. Because of that, the pre-1.0 and 1.0 version of the protocol are not compatible.[75, 76]. 73

Apache 2 Apache 2 is an open-source license from the Apache Software Foundation. The license allow modifications and redistribution of the software. It has no copyleft.[14]. 13, 21, 70, see also copyleft & open-source

API Is a interface to communicate from one system to an other[77]. 70, 73

arc42 arc42 is a template for documentation and communication of a system or software. It focus on lean and agile development approaches.[78] The template is available in available in different format, one of them is L^AT_EX.[79] It is open-source licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.[80]. 2, 58, see also L^AT_EX

asyncMDSL AsyncMDSL is part of Microservice Domain-Specific Language (MDSL) and handles asynchronous contracts between systems, mostly by Message Channels. The asynchronous messaging part of MDSL was first introduced by De Liberali. The syntax is derived from the patterns described in "Enterprise Integration Patterns".[81, 83]. see also MDSL & AsyncAPI

"At Least Once" To achieve "At Least Once" message delivery, every message has to be acknowledged by the receiver. This acknowledgment is sent back to the sender. If the sender does not receive an acknowledgment for a message in a certain time frame, the message is retransmitted.[22]. 6, 7, 20, 24, 31, 40, 54, 55, 61, 67, 69

"At Most Once" "At Most Once" describes that the message is delivered once or if it fails, it will not at all be delivered at all.

This approach is more an anti-pattern for reliable messaging and can not be found in any of our sources. However, Hohpe and Woolf describe something similar in the pattern "Fire-and-Forget", this pattern is part of the "Conversation Patterns" which the author represents as an addition to "Enterprise Integration Patterns".[84, 85]. 30, 55, 68

AWS Amazon Web Services (AWS) is a cloud platform provided by Amazon[86]. 73

"Best Effort" "Best Effort" in the contrast to any guarantee. In this context "Best Effort" means that none of the defined delivery guarantees are claimed.. 30, 55

Buildkite Buildkite is a service for coordinating buildpipelines which get executed on own infrastructure. They provide a build agent that can easily be deployed to AWS and other platforms. Buildkite is well integrated with GitHub.[21]. 21, 22, see also AWS, CI & GitHub

bytecode "Bytecode is program code that has been compiled from source code into low-level code designed for a software interpreter. It may be executed by a virtual machine (such as a Java Virtual Machine (JVM)) or further compiled into machine code, which is recognized by the processor."[87]. see also Java & JVM

C C is a programming language developed by Dennis Ritchie. C++ is a superset of C. Later it was standardized by AMSI and ISO.[88]. 40, 45, 46, 69–71, see also C++

C++ C++ is a general-purpose programming language designed by Bjarne Stroustrup. Later it was standardized by AMSI and ISO.[88]. 45, 46, 68

C++ Bindings C++ Bindings are wrappers for a programming language to use C++ function in it.[89]. see also C++

Cargo Cargo is the default package manager for Rust. It is really simple, compiles code, installs packages, runs tests, formats source files and generates the documentation. Cargo can be extended with additional subcommands by installing packages via Cargo itself.. 39, 71, see also Rust

CARU CARU AG is an AgeTech startup with the mission to help the elderly to live independently for longer. It is the industry partner of this thesis.. 9, 12, 13, 37, see also CARU Device

CARU Device The CARU Device is a product placed in the living environment of an elderly person to collect various room parameters, learn the behaviors of the inhabitant and alarm relatives or caretakers in case of deviations. Its user experience is optimized for elderly people. It was formerly known as CARU SmartSensor.[23]

The device has two CPUs with two different operating systems on it.. 7–11, 15, 16, 37, 59, see also FreeRTOS, Linux & CARU

CC0 "No Rights Reserved" License[90]. 43, 44

CI "Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early."[91]. 73

Cloud Native "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."[92]. 69, see also CNCF

CloudEvents The CloudEvents specification is a standard from the Cloud Native Computing Foundation (CNCF) with the goal to "describe event data in common formats to provide interoperability across services, platforms and systems." [93] The standard reached version 1.0 on the 24th of October 2019.. 7–11, 15, 24, 30, 37, 59, 67–70, see also CNCF

CloudEvents Router The CloudEvents Router is the open-source Router that was deployed during the thesis "CloudEvents Router". [1]. 15, 71

CloudSubscriptions Discovery API The CloudSubscriptions Discovery API is a vendor-neutral API specification to discover services and the events published by these services. The specification is part of the CloudEvents specification and is intended to be used with the CloudEvents message format. [94]. see also CloudEvents & API

CNCF The Cloud Native Computing Foundation is part of the Linux Foundation. The goal of the foundation is to empower the usage of the cloud by providing standards and open-source projects, the use the term Cloud Native. [92] The foundation hosts projects like Kubernetes [95] and Prometheus [96]. Apple, Microsoft, Google and Amazon Web Services are only a couple of the most famous partners. [97]. 73, see also Cloud Native

copyleft Copyleft is a method for making a program including all versions and modifications free. [98]. 13, 21, 67, 69

Docker "Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises." [99]. 39

DSL A Domain Specific Language is a custom language for a specific purpose. It is an abstract language that should solve a description problem in a specific domain. [100]. 73

Eclipse Mosquitto Eclipse Mosquitto is an open-source MQTT message broker with an additional command line tool to publish and subscribe from and to MQTT queues. [101]. 16

EKS Amazon Elastic Kubernetes Service is a managed Kubernetes cluster as a service provided by Amazon. [102]. see also Kubernetes

"Exactly Once" In "Exactly Once" delivery, the middleware only transfers the message once to the consumer. The message exchange should be reliable, as a retransmit is not possible if an acknowledgment is missed. To fulfill that, often a transactional message exchange is used. [22] However, "Exactly Once" is challenging to implement, if not even impossible. It depends on the exact definition. [22, 65, 66] Mostly it is more economical and enough to handle "At Least Once". [65]. 54, 55, 69

FaaS "Function as a service (FaaS) is a cloud computing model that enables users to develop applications and deploy functionalities without maintaining a server, increasing process efficiency. The concept behind FaaS is serverless computing and architecture, meaning the developer does not have to take server operations into consideration, as they are hosted externally. This is typically utilized when creating microservices such as web applications, data processors, chatbots and IT automation." [103]. 73

FreeRTOS FreeRTOS is a Real Time Operating System (RTOS) written in C distributed under the MIT license. The kernel binary is around 6KB to 12KB. [104]. 37, 45, 59, see also RTOS

Git Git is an open-source distributed version control system. The tool was created in 2005 by the Linux community. [12]. 13, 20, 42, 69

GitHub GitHub is a source code management and collaboration platform based on Git. It is a well known platform for open-source code and was acquired in 2018 by Microsoft. [13]. 13, 20, 21, 39, 40, 42, 44, 61, 68, 71, see also Git

- GNU 2** The GNU 2 or GNU General Public License v2 is a copyleft license of the Free Software Foundation, the latest version is version 3.[105]. 69, 70, see also GNU 3, copyleft & open-source
- GNU 3** The GNU 3 or GNU General Public License v3 is a copyleft license of the Free Software Foundation, the latest version is version 3.[106]. 69, 70, see also GNU 2, copyleft & open-source
- Go** Go is a statically typed general purpose programming language.[107] The language was developed by Google and is now open-sourced under the BSD-style license.[108]. 15, 45, 67
- GPIO** A General Purpose Input/Output (GPIO) is a low-level interface that is used to connect microcontrollers to other electronic components.[109]. 73
- HTTP** "The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers"[110]. 73
- Industry 4.0** "The term "Industry 4.0" refers to the transformation of the manufacturing sector by digital technologies, such as the internet of things (IoT), artificial intelligence (AI), machine learning, robotics, 3-D printing, visualization, virtual/augmented reality and analytics."[111]. see also IoT
- IoT** "The internet of Things, or "IoT" for short, is about extending the power of the internet beyond computers and smartphones to a whole range of other things, processes and environments. Those "connected" things are used to gather information, send information back, or both."[112]. 69, 73
- Java** "Java is a high-level programming language developed by Sun Microsystems."[113] Java code is compiled to java bytecode and executed with a JVM. 15, 67, 68, 71, 73, see also JVM & bytecode
- JavaScript** "JavaScript is a programming language commonly used in web development. It was originally developed by Netscape as a means to add dynamic and interactive elements to websites."[114] It is a scripting language based on the ECMAScript standard.[115]. 70, 73
- JDK** The Java Development Kit is a combination of the Java runtime, the compiler and other tooling that is needed for the development of Java applications.[116]. 73
- JSON** "JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null. It is based upon JavaScript syntax but is distinct from it: some JavaScript is not JSON."[117]. 73
- JVM** The Java Virtual Machine is a virtual machine running a java program. It implements the concrete commands for the abstract java specification. It executes the java bytecode and manages the memory.[118]. 73, see also Java
- Knative** Knative is a FaaS platform on top of Kubernetes.. see also Kubernetes & FaaS
- Knative Eventing** Knative Eventing is a set of loosely coupled services that route CloudEvents from producers to interested consumers. It is tightly integrated into Kubernetes.[119]. see also Knative & Kubernetes
- Kubernetes** Kubernetes is an open-source container orchestration platform hosted by CNCF.[120]. 69, 70

LaTeX "LaTeX, which is pronounced «Lah-tech» or «Lay-tech» (to rhyme with «blech» or «Bertolt Brecht»), is a document preparation system for high-quality typesetting. It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing." [121]

LaTeX is distributed under the LaTeX project public license. It is a free software license. [122]. 43, 44, 67

Linux Linux is a operating system written in C with a microkernel architecture and licenced under the GNU 2 license. [88, 123, 124]. 7, 13, 16, 69

Markdown "Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML)." [125]. 43, 44

MDSL Microservice Domain-Specific Language (MDSL) is a DSL to describe the interactions between systems. The language describes Application Programming Interfaces (APIs) as contracts in a vendor and protocol neutral way.

The language was first proposed by O. Zimmermann and is tightly coupled with the Microservice API Patterns. [126, 127]. see also asyncMDSL & DSL

Microkernel "The Microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The Microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration." [29]. 19, 24, 25, 27–31, 40, 48, 50, 65

MIT license The MIT License is a open-source license originally created by the Massachusetts Institute of Technology and documented by the OSI. [128]. 69, 70, see also OSI & open-source

MQTT MQTT is a lightweight publish/subscribe messaging protocol for machine to machine communication. [129, 130]. 73

MVP A product is at the state of a Minimum Viable Product when it gives the user an additional value, but as little extra feature as possible to get a good feedback loop for additional features [131]. 73

NATS NATS is a hyper-scalable messaging system hosted by the CNCF. Its design goal is to provide globally deployable, multi-tenant messaging infrastructure that supports publish/subscribe and request/reply messaging semantics. NATS was originally built by Synadia for their global communications system (NGS) but then donated to the CNCF. [132, 133]. 67, see also CNCF

Node.js Node.js is an open-source JavaScript runtime build on top of the Google Chrome V8 JavaScript engine. [134]. see also JavaScript

open-source "Generally, open-source software is software that can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone. Open-source software is made by many people, and distributed under licenses that comply with the open-source definition.

The internationally recognized open-source Definition provides 10 criteria that must be met for any software license, and the software distributed under that license, to be labeled 'open-source software.' " [135]

GNU 2, GNU 3, Apache 2, Academic Free License and MIT license are all open-source licenses which have been approved by OSI. 11, 13, 20, 21, 67–70, see also OSI

OSI OSI is a California public non-profit corporation with the goal of educate about open-source. [136]. 74, see also open-source

"ownership model" The "ownership model" is the most unique feature of Rust. In Rust memory is allocated and freed explicit, but in contrast to C or other low level programming languages without the need of an garbage collector, Rust enforces a set of rules on this actions, to ensure memory safety."[7]. 13, 19, 71, see also Rust

pair-programming Pair-programming is a style of programming where two programmers working collaboratively on the same code, design or test at the same time. The desired goal behind this strategy is to have better code quality and better know-how distribution.[19, 20]. 20

PlantUML PlantUML is a tool to generate different Unified Modeling Language (UML) and non UML diagrams. The diagrams are generated by writing with the PlantUML Language. The tool is written in Java and has various output formats, one of the im JPGE.[137]. see also UML

Port (Plugin) This is a plugin type of the CloudEvents Router. The Port is responsible for exchanging CloudEvents with the outside world.. see also Microkernel, CloudEvents, CloudEvents Router & CERK

pull request Pull requests are a feature of GitHub to disusse the differences between two branches with the goal to merge the changes of one branch into the other branch.[138]. 20, 39, 42, see also Git & GitHub

Python Python is a high-level general purpose programming language; It is used as scripting language, for Web application and many other things[139]. 8, 15, 16, 45, 46, 67

QoS Quality of Service is defined in the term of telecommunication as "Totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service."[140]. 74

RTOS Real Time Operating Systems are a type of operating systems where "The scheduler ... is designed to provide a predictable (normally described as deterministic) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline). A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted (and is therefore deterministic)."[141]. 74

Rust Rust is a low-level programming language with guaranteed thread and memory safety. It achieves that with its unique "ownership model". The goal is to build safe and reliable software.[7, 8] Rust is used by Mozilla Firefox, Dropbox and many others.[9, 10] In 2019, Rust has also been elected in the well-known survey of Stack Overflow for being the "most loved" programming language.[11]. 13, 16, 19, 40, 45, 46, 49–53, 68, 71, see also "ownership model" & Stack Overflow

rustc rustc is the Rust compiler provided by the Rust Team. It can be used directly (**rustc**) or with Cargo (**cargo build**).[142]. see also Rust & Cargo

rustup rustup is a tool to install different versions of Rust, like stable, beta and nightly, and for different compilation targets. It is the recommended way for developers to install Rust.[143]. see also Rust & Cargo

SaaS Software as a service (SaaS) is a software distribution model in which a third-party provider hosts applications and makes them available to customers over the Internet.[144]. 74

SDK Software Development Kits are libraries which ease the access to APIs. 74, see also API

Serverless Framework The Serverless Framework is a FaaS abstraction layer.. 71, see also FaaS

SQS Amazon Simple Queue Service (SQS) is a fully managed message queuing service from Amazon with first-in, first-out (FIFO) support[145]. 74

Stack Overflow Stack Overflow is primary a Q&A forum for coding related problems.. 13, 19, 71

STOMP A Simple Text Oriented Message Protocol, designed for asynchronous message passing[146]. 74

TCP The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.[147]. 74

UML UML is a standard for diagramming notation. It was created by Booch and Rumbaugh in 1994.[148]. 71, 74

UNIX socket "The AF_UNIX socket family is used to communicate between processes on the same machine efficiently."[149] . 7–10, 71, see also UNIX

WAMP "WAMP is a routed protocol that provides two messaging patterns: Publish & Subscribe and routed Remote Procedure Calls. It is intended to connect application components in distributed applications. WAMP uses WebSocket as its default transport, but can be transmitted via any other protocol that allows for ordered, reliable, bi-directional, and message-oriented communications."[150]. 74

WebHook WebHooks are a popular pattern to deliver events to HTTP-enabled services. Unfortunately, there is no formal specification for WebHooks, and most services implement their own flavour of it.[151, 152]. see also HTTP

Acronyms

ADR ARD (Architecture Decision Record). 20, 43, 44, [Glossary](#) ADR

AMQP Advanced Message Queuing Protocol[76]. 39, [Glossary](#) AMQP

ANSI American National Standards Institute. 68

API Application Programming Interface. 40, 70, [Glossary](#) API

AWS Amazon Web Services[86]. 15, 16, 21, 38, 67, 68, [Glossary](#) AWS

CERK CloudEvents Router with a MicroKernel. 7, 39

CI Continuous Integration. 21, 22, 39, 42, 63, [Glossary](#) CI

CNCF Cloud Native Computing Foundation . 68, [Glossary](#) CNCF

CPU central processing unit. 37, 68

DSL Domain Specific Language. [Glossary](#) DSL

EC2 Elastic Compute Cloud. 16, 73, [Glossary](#) Amazon Elastic Compute Cloud

FaaS Function as a Service. 15, 67, 70, 71, [Glossary](#) FaaS

FR Functional Requirements. 7

GPIO General Purpose Input/Output. [Glossary](#) GPIO

HSR University of Applied Sciences Rapperswil. 12

HTTP HyperText Transfer Protocol. [Glossary](#) HTTP

IoT Internet of Things. 67, [Glossary](#) IoT

ISO International Organization for Standardization. 68

JDK Java Development Kit. [Glossary](#) JDK

JSON JavaScript Object Notation. [Glossary](#) JSON

JVM Java Virtual Machine. 68, 70, [Glossary](#) JVM

KB Kilobytes, equal to 1000 Bytes. 37, 69

MADR Markdown Architectural Decision Records[31]. 43, 44

MB Megabytes, equal to 1000 Kilobytes. 37

MDSL Microservice Domain-Specific Language. 67, [Glossary](#) MDSL

MQTT Message Queuing Telemetry Transport[130]. 7, 9, 10, 15, 16, 39, 67, 69, [Glossary](#) MQTT

MVP Minimum Viable Product[131]. [Glossary MVP](#)

NFR Non-Functional Requirements. 11, 59

OASIS Organization for the Advancement of Structured Information Standards. [Glossary OASIS](#)

OSI Open Source Initiative. 67, 70, [Glossary OSI](#)

OST University of Applied Sciences of Eastern Switzerland. 12

QoS Quality of Service. 10, [Glossary QoS](#)

RTOS Real Time Operating System. 11, 69, [Glossary RTOS](#)

SaaS Software as a Service. [Glossary SaaS](#)

SDK Software Development Kit. 40, [Glossary SDK](#)

SQS Simple Queue Service[145] . 15, 16, [Glossary SQS](#)

STOMP Simple Text Oriented Messaging Protocol[146]. [Glossary STOMP](#)

TCP Transmission Control Protocol[147]. [Glossary TCP](#)

TLS Transport Layer Security. 39

UML Unified Modeling Language. 71, [Glossary UML](#)

VM Virtual Machine. 67

WAMP Web Application Messaging Protocol[150]. [Glossary WAMP](#)

Bibliography

- [1] L. Basig and F. Lazzaretti, CloudEvents Router, HSR, 2020.
[Online]. Available: <https://eprints.hsr.ch/832/>.
- [2] —, Reliable Messaging using the CloudEvents Router, 2021.
- [3] T. Helbling, Projektantrag: Reliable Messaging Using the CloudEvents Router, 2020.
- [4] R. Aebersold, Requirements Gathering, 2019.
- [5] T. Helbling, Project Thesis Proposal - CloudEvent Router, 2019.
- [6] O. Zimmermann, Aufgabenstellung Studienarbeit, 2019.
- [7] C. N. Steve Klabnik, The rust programming language.
Random House LCC US, Aug. 12, 2019, ISBN: 1718500440.
- [8] D. Bryant, A Quantum Leap for the Web, Medium, 2016. [Online]. Available:
<https://medium.com/mozilla-tech/a-quantum-leap-for-the-web-a3b7174b3c12>.
- [9] Rust. [Online]. Available: <https://research.mozilla.org/rust/>.
- [10] Production users. [Online]. Available: <https://www.rust-lang.org/production/users>.
- [11] Stack Overflow Developer Survey 2019, Tech. Rep., 2019.
[Online]. Available: <https://insights.stackoverflow.com/survey/2019>.
- [12] S. Chacon and B. Straub, Pro git. Apress, 2014.
- [13] K. Johnson, GitHub passes 100 million repositories, 2018. [Online]. Available:
<https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/>.
- [14] Apache License, Version 2.0, 2019.
[Online]. Available: <https://www.apache.org/licenses/LICENSE-2.0>.
- [15] AWS Lambda – Serverless Compute – Amazon Web Services, 2021.
[Online]. Available: <https://aws.amazon.com/lambda/>.
- [16] G. Hohpe and B. Woolf, Enterprise integration patterns, 1, Ed. Addison-Wesley, 2003,
480 pp., ISBN: 978-0321200686. [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html>.
- [17] L. Basig and F. Lazzaretti, Protocol Interoperability in a Stateless Message Router, 2021.
- [18] —, Reliable Messaging – Patterns and Strategies for Message Routing, research rep., 2021.
- [19] L. A. Williams, The Collaborative Software Process,
PhD thesis, The University of Utah, 2000.
[Online]. Available: <http://collaboration.csc.ncsu.edu/laurie/Papers/dissertation.pdf>.
- [20] M. Fowler, PairProgrammingMisconceptions, 2006.
[Online]. Available: <https://martinfowler.com/bliki/PairProgrammingMisconceptions.html>.
- [21] Buildkite Features, 2019. [Online]. Available: <https://buildkite.com/features>.
- [22] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, Cloud computing patterns.
Springer Publishing Company, Incorporated, Feb. 18, 2014, ISBN: 978-3-7091-1567-1.
- [23] CARU | Das intelligente Notrufsystem, 2020.
[Online]. Available: <https://www.caruhome.com/>.
- [24] Amazon EC2 Instance Types.
[Online]. Available: <https://aws.amazon.com/ec2/instance-types>.

- [25] "Distroless" Docker Images, 2021.
[Online]. Available: <https://github.com/GoogleContainerTools/distroless>.
- [26] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages, Feb. 4, 2020.
- [27] GitHub Security Alerts integration.
[Online]. Available: <https://dependabot.com/blog/github-security-alerts/>.
- [28] D. Hosfelt, Fearless Security: Memory Safety, Mozilla Hacks - the Web developer blog, 2019.
[Online]. Available: <https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/>.
- [29] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-oriented software architecture. Jul. 12, 1996, 476 pp., ISBN: 0471958697.
- [30] M. Nygard, Documenting Architecture Decisions, Nov. 5, 2019. [Online]. Available: <http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions>.
- [31] Markdown Architectural Decision Records, Nov. 5, 2019.
[Online]. Available: <https://github.com/adr/madr/tree/2.1.2>.
- [32] U. Zdun, R. Capill, H. Tran, and O. Zimmermann, Sustainable Architectural Design Decisions, Mar. 9, 2014. [Online]. Available: <https://www.infoq.com/articles/sustainable-architectural-design-decisions/>.
- [33] ce-rust: Thesis (Studien Arbeit), Nov. 5, 2019.
[Online]. Available: <https://github.com/ce-rust/sa>.
- [34] cerk (CloudEvent Router Kernel), Nov. 5, 2019.
[Online]. Available: <https://github.com/ce-rust/cerk>.
- [35] O. Zimmermann, Making Architectural Knowledge Sustainable – The Y-Approach, Nov. 5, 2019. [Online]. Available: https://resources.sei.cmu.edu/asset_files/Presentation/2012_017_001_31349.pdf.
- [36] adr.github.io, Nov. 5, 2019.
[Online]. Available: <https://adr.github.io/#sustainable-architectural-decisions>.
- [37] Rust Homepage, 2019. [Online]. Available: <https://www.rust-lang.org>.
- [38] The GNU C Reference Manual.
[Online]. Available: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>.
- [39] Standard C++, 2019. [Online]. Available: <https://isocpp.org/>.
- [40] The Go Programming Language, 2019. [Online]. Available: <https://golang.org/>.
- [41] python.org, 2019. [Online]. Available: <https://www.python.org/>.
- [42] Rust - Embedded Devices, 2019.
[Online]. Available: <https://www.rust-lang.org/what/embedded>.
- [43] freertos_rs. [Online]. Available: https://docs.rs/freertos_rs/0.3.0/freertos_rs/.
- [44] Rust - Validating References with Lifetimes, 2019. [Online]. Available: <https://doc.rust-lang.org/1.30.0/book/2018-edition/ch10-03-lifetime-syntax.html>.
- [45] Rust - What Is Ownership? 2019.
[Online]. Available: <https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html>.
- [46] I. Barland, Why C and C++ are Awful Programming Languages, 2019.
[Online]. Available: <https://www.radford.edu/ibarland/Manifestoes/whyC++isBad.shtml>.
- [47] I. Joyner, C++??: A Critique of C++, 1992.
[Online]. Available: <http://www.literateprogramming.com/c++critique.pdf>.
- [48] MicroPython, Nov. 5, 2019. [Online]. Available: <https://micropython.org/>.

- [49] M. Fowler, Microservices, Mar. 23, 2014.
[Online]. Available: <https://martinfowler.com/articles/microservices.html>.
- [50] M. Fowler, Patterns of enterprise application architecture. Addison Wesley, Nov. 1, 2002, 533 pp., ISBN: 0321127420.
[Online]. Available: <https://www.martinfowler.com/books/ea.html>.
- [51] Linkage - The Rust Reference, Nov. 12, 2019.
[Online]. Available: <https://doc.rust-lang.org/reference/linkage.html>.
- [52] DLL Hell - A Article About DLL Hell, Nov. 12, 2019. [Online]. Available: <https://web.archive.org/web/20080703211550/http://dllcity.com/dll-hell.php>.
- [53] Dynamic Loading & Plugins, Nov. 12, 2019.
[Online]. Available: https://michael-f-bryan.github.io/rust-ffi-guide/dynamic_loading.html.
- [54] Why Async? 2019. [Online]. Available: https://rust-lang.github.io/async-book/01_getting_started/02_why_async.html.
- [55] Using Threads to Run Code Simultaneously, 2019.
[Online]. Available: <https://doc.rust-lang.org/book/ch16-01-threads.html>.
- [56] Cannot define async fn with no_std #56974.
[Online]. Available: <https://github.com/rust-lang/rust/issues/56974>.
- [57] Async-await hits beta!
[Online]. Available: <https://blog.rust-lang.org/2019/09/30/Async-await-hits-beta.html>.
- [58] N. Matsakis, Async-await on stable Rust! 2019.
[Online]. Available: <https://blog.rust-lang.org/2019/11/07/Async-await-stable.html>.
- [59] G. Hohpe and B. Woolf, Message Broker, Nov. 5, 2019. [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageBroker.html>.
- [60] Multi-producer, single-consumer FIFO queue communication primitives, 2019.
[Online]. Available: <https://doc.rust-lang.org/std/sync/mpsc/index.html>.
- [61] R. Hanmer, Solution: Use a Broker, Nov. 5, 2019.
[Online]. Available: <https://www.oreilly.com/library/view/pattern-oriented-software-architecture/9781119963998/chap12-sec005.html>.
- [62] Using Message Passing to Transfer Data Between Threads, 2019.
[Online]. Available: <https://doc.rust-lang.org/book/ch16-02-message-passing.html>.
- [63] A thread-safe reference-counting pointer, 2019.
[Online]. Available: <https://doc.rust-lang.org/std/sync/struct.Arc.html>.
- [64] Useful synchronization primitives, 2019.
[Online]. Available: <https://doc.rust-lang.org/std/sync/index.html>.
- [65] T. Treat, You Cannot Have Exactly-Once Delivery, 2015.
[Online]. Available: <https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/>.
- [66] M. Steen and A. Tanenbaum, Distributed systems.
Place of publication not identified: Maarten van Steen, 2017, ISBN: 9781543057386.
- [67] G. Starke, Beispiele für Qualitätsanforderungen an Software, GitHub, 2017.
[Online]. Available: <https://github.com/arc42/quality-requirements>.
- [68] ISO-25010 in Agile Architecture, 2012. [Online]. Available: <http://a2build.com/architectdagile/Architecte%20Agile.html?ISO25010.html>.
- [69] CloudEvents Specification v0.3, 2019.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v0.3/spec.md>.
- [70] AWS IoT Core. [Online]. Available: <https://aws.amazon.com/iot-core/>.
- [71] CloudSubscriptions: Subscriptions API, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0.1/subscriptions-api.md>.

- [72] NATS Streaming Introduction, 2020.
[Online]. Available: <https://docs.nats.io/nats-streaming-concepts/intro>.
- [73] Academic Free License ("AFL") v. 3.0 | Open Source Initiative.
[Online]. Available: <https://opensource.org/licenses/AFL-3.0>.
- [74] Amazon EC2, 2021. [Online]. Available: <https://aws.amazon.com/ec2>.
- [75] 1.3. AMQP - Advanced Message Queuing Protocol Red Hat Enterprise MRG 3, 2019.
[Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_mrg/3/html/messaging_programming_reference/amqp__advanced_message_queuing_protocol.
- [76] R. Godfrey, D. Ingham, and R. Schloming, OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0, OASIS, 2012.
[Online]. Available: <https://www.oasis-open.org/standards#amqp1.0>.
- [77] P. Eising, What exactly IS an API? - Perry Eising - Medium, Medium, 2017.
[Online]. Available: <https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f>.
- [78] arc42 - arc42. [Online]. Available: <https://arc42.org/>.
- [79] Download arc42 - arc42. [Online]. Available: <https://arc42.org/download>.
- [80] arc42 License - arc42. [Online]. Available: <https://arc42.org/license>.
- [81] G. De Liberali, AsyncMDSL: a domain-specific language for modeling message-based systems, Master's thesis, Università di Pisa, 2020.
[Online]. Available: <https://etd.adm.unipi.it/t/etd-06222020-100504/>.
- [82] G. Hohpe and B. Woolf, Enterprise integration patterns. Addison Wesley, Jan. 1, 2004, 480 pp., ISBN: 0321200683.
- [83] AsyncMDSL extension (DSL and Eclipse plugin), 2020.
[Online]. Available: <https://github.com/Microservice-API-Patterns/MDSL-Specification/tree/3cf15b9c866c1086da7e8cd354b2e991055f8d3d/examples/asyncMDSL>.
- [84] —, Fire-and-Forget, 2017.
- [85] —, Conversation Patterns, 2017. [Online]. Available: <https://www.enterpriseintegrationpatterns.com/patterns/conversation/index.html>.
- [86] What is AWS. [Online]. Available: <https://aws.amazon.com/what-is-aws/>.
- [87] Knative Serving License. [Online]. Available: <https://techterms.com/definition/bytecode>.
- [88] B. Stroustrup, Programming. Addison Wesley, May 15, 2014, 1312 pp., ISBN: 0321992784.
- [89] H. Patel, Python - C++ bindings | Hardik Patel, 2017.
[Online]. Available: <https://www.hardikp.com/2017/12/30/python-cpp/>.
- [90] Creative Commons - CC0, Nov. 5, 2019.
[Online]. Available: <https://creativecommons.org/share-your-work/public-domain/cc0>.
- [91] Continuous integration | ThoughtWorks, 2019.
[Online]. Available: <https://www.thoughtworks.com/continuous-integration>.
- [92] CNCF Cloud Native Definition v1.0, 2019.
[Online]. Available: <https://github.com/cncf/toc/blob/master/DEFINITION.md>.
- [93] CloudEvents Specification v1.0, GitHub, 2019.
[Online]. Available: <https://github.com/cloudevents/spec/tree/v1.0>.
- [94] CloudSubscriptions: Discovery, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/v1.0.1/discovery.md>.
- [95] [Online]. Available: <https://www.cncf.io/cncf-kubernetes-project-journey/>.
- [96] CNCF Prometheus Project Journey - Cloud Native Computing Foundation, 2019.
[Online]. Available: <https://www.cncf.io/cncf-prometheus-project-journey/>.

- [97] Members - Cloud Native Computing Foundation.
[Online]. Available: <https://www.cncf.io/about/members/>.
- [98] What is Copyleft? [Online]. Available: <https://www.gnu.org/licenses/copyleft.en.html>.
- [99] What is Docker? 2018. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined>.
- [100] S. Efftinge, M. Volter, A. Haase, and B. Kolb, The Pragmatic Code Generator Programmer, 2006. [Online]. Available:
<https://www.theserverside.com/news/1365073/The-Pragmatic-Code-Generator-Programmer>.
- [101] Eclipse Mosquitto - An open source MQTT broker.
[Online]. Available: <https://mosquitto.org/>.
- [102] Amazon Elastic Kubernetes Service. [Online]. Available: <https://aws.amazon.com/eks/>.
- [103] M. Rouse, What is function as a service (FaaS)? - Definition from WhatIs.com, TechTarget, 2018.
- [104] FreeRTOS. [Online]. Available: <https://www.freertos.org/>.
- [105] GNU General Public License v2.0 - GNU Project - Free Software Foundation, 2017.
[Online]. Available: <https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html>.
- [106] The GNU General Public License v3.0 - GNU Project - Free Software Foundation, 2016.
[Online]. Available: <https://www.gnu.org/licenses/gpl-3.0.html>.
- [107] C. Doxsey, An introduction to programming in go. 2012, ISBN: 978-1478355823.
[Online]. Available: <http://www.golang-book.com/books/intro>.
- [108] The Go Project - The Go Programming Language, 2019.
[Online]. Available: <https://golang.org/project/>.
- [109] What is GPIO? 2021. [Online]. Available:
<https://community.estimote.com/hc/en-us/articles/217429867-What-is-GPIO->.
- [110] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.
doi: 10.17487/RFC2616. [Online]. Available: <https://rfc-editor.org/rfc/rfc2616.txt>.
- [111] P. Mukhopadhyay, Sales Transformations For "Industry 4.0", Forbes, 2019.
[Online]. Available: <https://www.forbes.com/sites/forbestechcouncil/2019/10/16/sales-transformations-for-industry-4-0/#23aafb725a42>.
- [112] C. McClelland, What Is IoT? - A Simple Explanation of the Internet of Things, IoT For All, 2019.
[Online]. Available: <https://www.iotforall.com/what-is-iot-simple-explanation/>.
- [113] Java Definition, 2012. [Online]. Available: <https://techterms.com/definition/java>.
- [114] JavaScript Definition, 2014. [Online]. Available: <https://techterms.com/definition/javascript>.
- [115] ECMAScript® 2019 Language Specification, Ecma International.
[Online]. Available: <https://www.ecma-international.org/ecma-262/10.0/index.html>.
- [116] JDK Definition from PC Magazine Encyclopedia.
[Online]. Available: <https://www.pcmag.com/encyclopedia/term/45608/jdk>.
- [117] JSON - JavaScript | MDN, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON.
- [118] B. Venners, Inside the java virtual machine. McGraw Hill, 1999, ISBN: 0-07-135093-4.
- [119] Knative Eventing, GitHub. [Online]. Available: <https://github.com/knative/eventing/>.
- [120] CNCF Kubernetes Project Journey.
[Online]. Available: <https://www.cncf.io/cncf-kubernetes-project-journey/>.
- [121] Introduction to LaTeX. [Online]. Available: <https://www.latex-project.org/about/>.

- [122] The LaTeX project public license. [Online]. Available: <https://www.latex-project.org/lppl/>.
- [123] Linux from FOLDOC, 2000. [Online]. Available: <http://foldoc.org/linux>.
- [124] The Linux Kernel Archives - FAQ. [Online]. Available: <https://www.kernel.org/category/faq.html>.
- [125] J. Gruber, Daring Fireball: Markdown. [Online]. Available: <https://daringfireball.net/projects/markdown/>.
- [126] Microservice DSL (MDSL), 2020. [Online]. Available: <https://microservice-api-patterns.github.io/MDSL-Specification/>.
- [127] Microservice API Patterns, 2020. [Online]. Available: <https://microservice-api-patterns.org/>.
- [128] G. Haff, The mysterious history of the MIT License | Opensource.com, 2019. [Online]. Available: <https://opensource.com/article/19/4/history-mit-license>.
- [129] MQTT Homepage. [Online]. Available: <http://mqtt.org/>.
- [130] A. Banks and R. Gupta, MQTT Version 3.1.1 Plus Errata 01, OASIS Standard Incorporating, 2015. [Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html>.
- [131] Minimum Viable Product (MVP), [Online]. Available: <https://www.techopedia.com/definition/27809/minimum-viable-product-mvp>.
- [132] NATS Introduction, 2020. [Online]. Available: <https://docs.nats.io/>.
- [133] Synadia - Connect Everything, 2020. [Online]. Available: <https://synadia.com/>.
- [134] Introduction to Node.js. [Online]. Available: <https://nodejs.dev/>.
- [135] Frequently Answered Questions: What is "Open Source" software? [Online]. Available: <https://opensource.org/faq#osd>.
- [136] About the Open Source Initiative | Open Source Initiative. [Online]. Available: <https://opensource.org/about>.
- [137] Drawing UML with PlantUML, PlantUML Language Reference Guide, 2019. [Online]. Available: <http://plantuml.com/guide>.
- [138] About pull requests. [Online]. Available: <https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-pull-requests>.
- [139] D. Kuhlman, A python book: Beginning python, advanced python, and python exercises, 1.1a. 2012. [Online]. Available: https://web.archive.org/web/20120623165941/http://cutter.rexx.com/~dkuhlman/python_book_01.html.
- [140] E.800 : Definitions of terms related to quality of service, International Telecommunication Union (ITU), 2008. [Online]. Available: <https://www.itu.int/rec/T-REC-E.800-200809-I/en>.
- [141] Why RTOS and What is RTOS? [Online]. Available: <https://www.freertos.org/about-RTOS.html>.
- [142] The rustc book. [Online]. Available: <https://doc.rust-lang.org/rustc/what-is-rustc.html>.
- [143] [Online]. Available: <https://www.rust-lang.org/tools/install>.
- [144] M. Rouse, What is Software as a Service (SaaS)? - Definition from WhatIs.com, TechTarget, 2019.
- [145] Amazon Simple Queue Service. [Online]. Available: <https://aws.amazon.com/sqs/>.
- [146] STOMP Protocol Specification, Version 1.2, 2012.
- [147] J. Postel, Transmission Control Protocol, RFC 793, 1981. doi: 10.17487/RFC0793. [Online]. Available: <https://rfc-editor.org/rfc/rfc793.txt>.
- [148] C. Larman, Applying uml and patterns. Prentice Hall, 2004, vol. 3, ISBN: 9780131489066.

- [149] unix(7) - Linux manual page.
[Online]. Available: <http://man7.org/linux/man-pages/man7/unix.7.html>.
- [150] T. Oberstein and A. Goedde, The Web Application Messaging Protocol, IETF, 2019.
[Online]. Available:
https://wamp-proto.org/_static/gen/wamp_latest_ietf.html#rfc.authors.
- [151] HTTP 1.1 Web Hooks for Event Delivery, 2020.
[Online]. Available: <https://github.com/cloudevents/spec/blob/master/http-webhook.md>.
- [152] REST Hooks, 2020. [Online]. Available: <https://resthooks.org/>.

E. Load Test Statistics

In this appendix, we describe the load test we performed with the CloudEvents Router. We describe the tests and present the statistical analysis of the results. The analysis served to verify that the implementation of the "At Least Once" delivery guarantee semantics works as intended under high-pressure workloads.

Test Method

The load test uses a heuristic approach and works as follows:

- We send a big amount of messages (100'000 CloudEvents) to the input channel.
- We start and then repeatedly restart the CloudEvents Router while it is routing messages.
- We check if all messages have arrived on the output channel. It is important to verify each message, as opposed to only check the number of arrivals, as messages could be duplicated.

To simulate a real-world production environment, we deploy all components on a Kubernetes cluster. The deployment is visualized in Figure E.1. We need the following components:

- Our CloudEvents Router that routes from one input channel to one output channel. The router is either configured to use the new delivery guarantee "At Least Once" or the "Best Effort" mode from the student research project.

The following plugins are configured:

- Depending on the exact test, the ports `cerk_port_amqp`¹ and `cerk_port_mqtt_mosquitto`², or only one of them is used.
- The routing is always done with the `cerk_router_broadcast`³ router.
- Additionally, a health check is configured on Kubernetes to check the `readiness` and `liveness` of the CloudEvents Router. The `cerk_port_health_check_http`⁴ was used for this task.
- A MOM with two channels: One as the input of the router and one as the output of the router. Depending on the protocol, RabbitMQ or Eclipse Mosquitto is used.
- A service to generate the 100'000 events, each with a unique ID. We used our CloudEvents Router with a special port (`port_sequence_generator`), which autonomously generates a specified amount of messages.⁵
The port waits 1 ms between sending two messages, limiting the generation of messages to 1000 per second.

¹https://docs.rs/cerk_port_amqp/0.2.11

²https://docs.rs/crate/cerk_port_mqtt_mosquitto/0.2.11

³https://docs.rs/cerk_router_broadcast/0.2.8

⁴https://docs.rs/cerk_port_health_check_http/0.2.11

⁵https://docs.rs/cerk_port_dummies/0.2.11/cerk_port_dummies/fn.port_sequence_generator_start.html

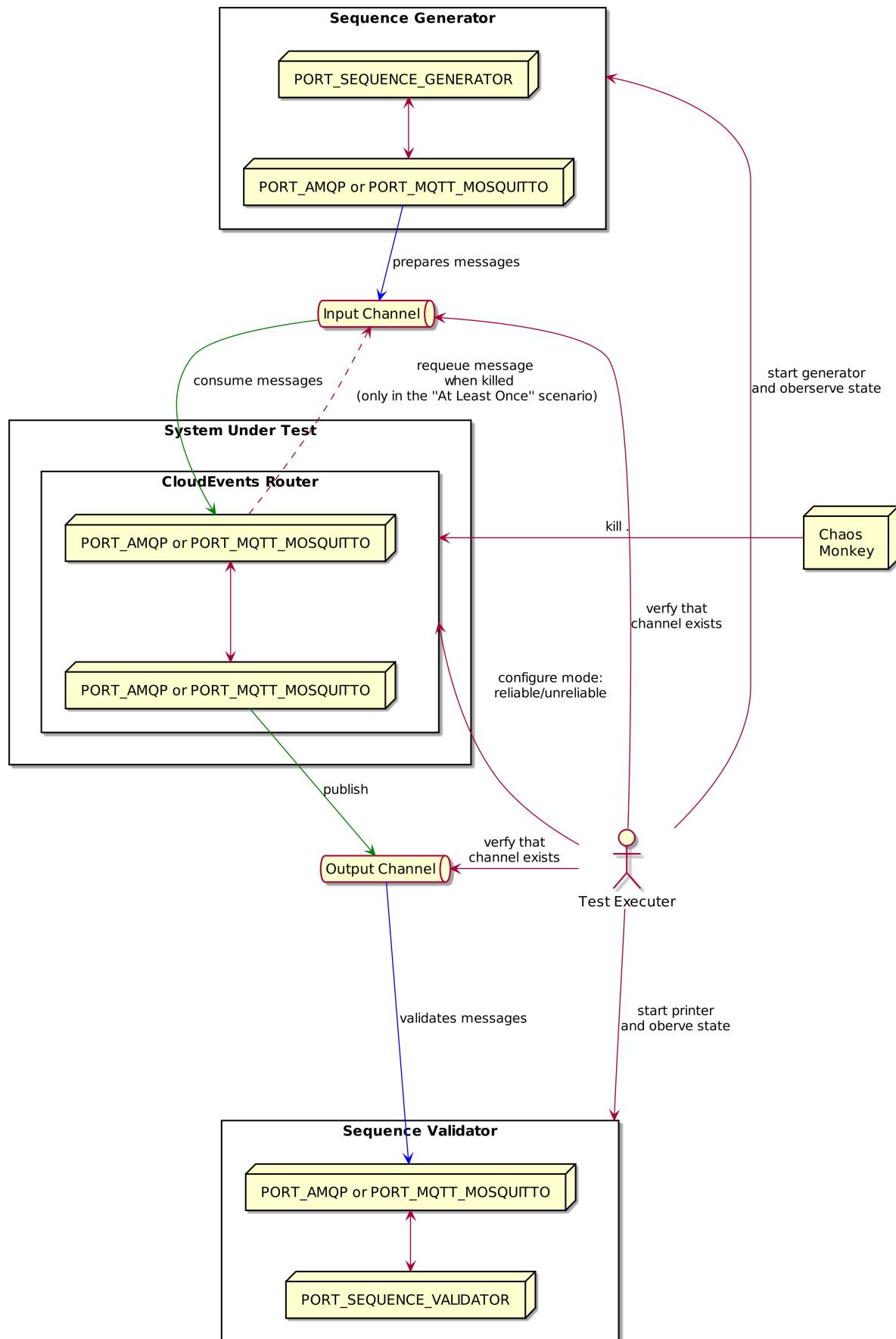


Figure E.1.: Load Test

- A service to validate the arrival of all 100'000 events on the output channel. The validator has to ensure that all 100'000 unique IDs arrived. Just counting the amount is not enough, as it might come to message duplication in the routing process.
For this verification, we also used an instance of our CloudEvents Router. This instance has a special sequence validator port (`port_sequence_validator`), that checks if all messages arrived.⁶
- chaoskube, an implementation of a Chaos Monkey for Kubernetes. The Chaos Monkey is a tool that randomly stops a service without any notice. The tool was created by Netflix to help with the creation of a resilient infrastructure where any service can be interrupted at any time without impacting the user experience.[15] This service is configured to terminate our CloudEvents Router regularly. Kubernetes will then start a new instance of the router.
Normally the Chaos Monkey terminates a random service of the whole deployment in a certain time frame. However, for the repeatability and reproducibility of the test, we changed that behavior: The Chaos Monkey terminates only the CloudEvents Router under test, in 10 seconds intervals. The validation service or the MOM with the two channels will never be restarted.

The load test exists in eight different variations. For each test variation, the CloudEvents Router under test has two ports configured: one to receive events from the input channel and one to send events to the output channel. Because we are testing two protocols (AMQP and MQTT), there are four port combinations. Additionally, we have to run each port combination twice: Once with the new "At Least Once" delivery semantic and once with the old "Best Effort" delivery semantics.

Test Setup

The test ran on a Minikube instance. Minikube is a tool to easily install Kubernetes on a personal computer.[84]

Runtime Minikube was used with Kubernetes version 1.19.4, and Docker as runtime and VM Driver, Docker v19.03.13-ce was used.

Resource Limitations The cluster had 12 respectively 16 cores and 32 GB of Memory as resource limitation (two slightly different machines where used).

Operating System Linux Manjaro, Kernel 5.8.18

The code used to generate these tests are in the CERK GitHub repository. Git Hash `a9fc65a6aed12834359c89c2fd7c33c1d4f15ee9` was used for the tests.⁷

The tests were executed with the script `run-all-in-loop.sh`, which executes all eight test variations in a loop, one after the other.

With the script `build-csv.sh`, the testdata created by the previous script was condensed into a single CSV file. This file was used to analyze our runs.

Test Result

The tests were analyzed with R. R is a programming language for statistical computations. It is licenced under the GNU 2 and GNU 3 language.[99, 100]. The test were written in a R Markdown file. The resulting file can be found in Appendix F "Analyzing the Load Test".

In summary: we ran 160 tests in the "At Least Once" and 160 tests in the "Best Effort" mode. In the "Best Effort" mode, depending on the port combination, we lost between 0.5 and 13'000

⁶https://docs.rs/cerk_port_dummies/0.2.11/cerk_port_dummies/fn.port_sequence_validator_start.html

⁷<https://github.com/ce-rust/cerk/tree/a9fc65a6aed12834359c89c2fd7c33c1d4f15ee9/integration-tests/chaos-monkey>

messages per 100'000 messages.

In the "At Least Once" mode, we lost none of the 16 million messages that were routed by the CloudEvents Router.

F. Analyzing the Load Test

Analyzing the Load Test

Linus Basig, Fabrizio Lazzaretti

1/12/2021

This is an R Markdown document that documents the statistical analysis that was executed on the load test data.

```
library(readr)

dataset_raw = read_csv2("output.csv",
                        col_types = cols_only(
                          machine = "c",
                          setup = "c",
                          mode = "c",
                          missing = "i",
                          start = col_datetime("%Y%m%d%H%M%S"),
                          end = col_datetime("%Y%m%d%H%M%S"),
                          brokercrash = "i",
                          ended = "i"))
```

Analyze Raw Data

```
library(dplyr)

dataset_raw %>% group_by(setup,mode) %>%
  summarise(mean = mean(missing), "test runs" = n())
```

```
## # A tibble: 8 x 4
## # Groups:   setup [4]
##   setup    mode      mean 'test runs'
##   <chr>   <chr>    <dbl>      <int>
## 1 amqp    reliable      0          41
## 2 amqp    unreliable   NA          41
## 3 amqp-mqtt reliable      0          41
## 4 amqp-mqtt unreliable 13325.        41
## 5 mqtt    reliable     NA          40
## 6 mqtt    unreliable  8186.         39
## 7 mqtt-amqp reliable      0          40
## 8 mqtt-amqp unreliable  7661.         40
```

Some meas are NA as some test have probably not finished. To solve this we have to do some data clean up.

Basic Clean Up

We have to filter out all tests, that are not finished or where the broker crashed during the run. These tests had a setup problem.

```
dataset = dataset_raw %>%
  filter(ended == 1) %>%
  filter(brokercrash == 0)

dataset %>% group_by(setup,mode) %>%
  summarise(mean = mean(missing), "test runs" = n())
```

```
## # A tibble: 8 x 4
## # Groups:   setup [4]
##   setup      mode      mean 'test runs'
##   <chr>    <chr>    <dbl>      <int>
## 1 amqp      reliable      0          41
## 2 amqp      unreliable  0.4          40
## 3 amqp-mqtt reliable      0          41
## 4 amqp-mqtt unreliable 13325.         41
## 5 mqtt      reliable      0          39
## 6 mqtt      unreliable  8186.         39
## 7 mqtt-amqp reliable      0          40
## 8 mqtt-amqp unreliable  7661.         40
```

Calculating the Amount of Tests that Were Executed

total:

```
count(dataset)
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   321
```

Reliable Test Runs

```
count(dataset %>% filter(mode == "reliable"))
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   161
```

The total number of routed events:

```
count(dataset %>% filter(mode == "reliable")) * 100000
```

```
##       n
## 1 16100000
```

Unreliable Test Runs

```
count(dataset %>% filter(mode == "unreliable"))
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1   160
```

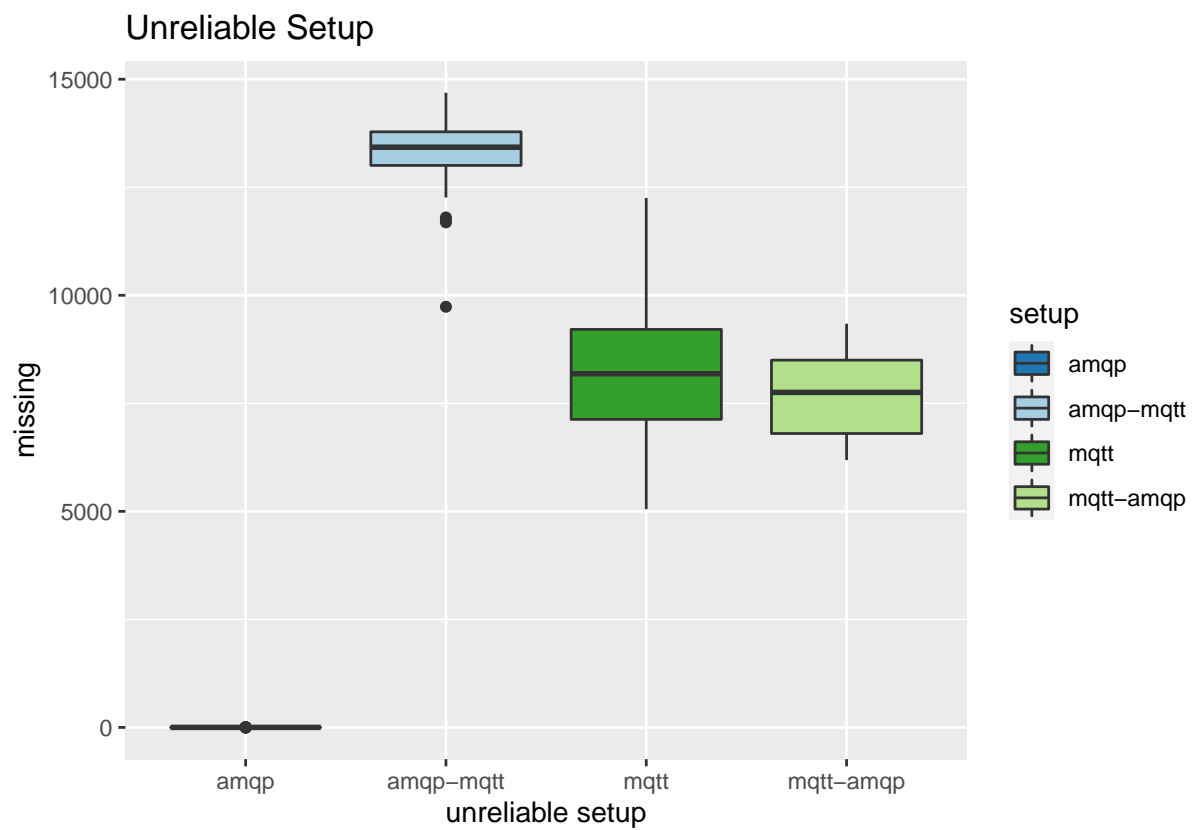
Check If Crashes Are Depending on the Test Type

Broker Crashes

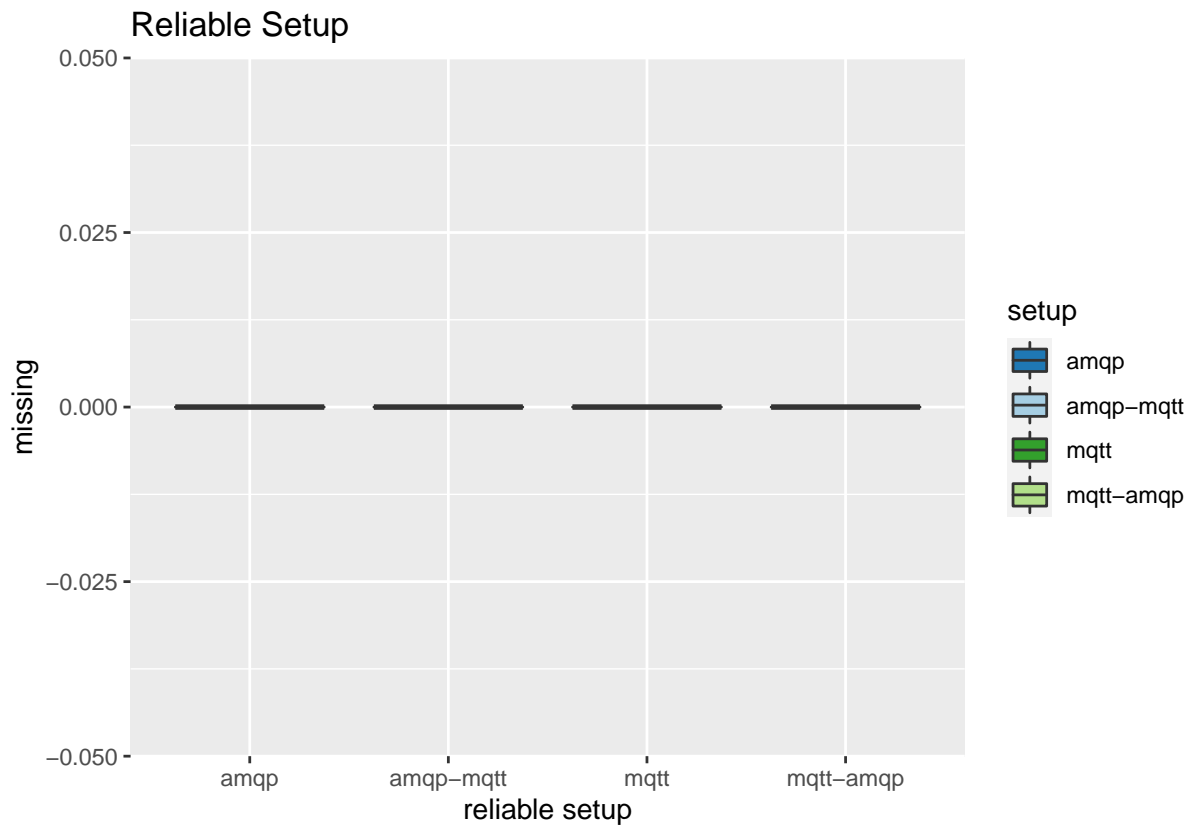
```
dataset_raw %>% group_by(setup,mode) %>%  
  filter(brokercrash != 0) %>%  
  summarise(brokercrash = sum(brokercrash))  
  
## # A tibble: 0 x 3  
## # Groups:   setup [0]  
## # ... with 3 variables: setup <chr>, mode <chr>, brokercrash <int>  
  
looks good! we have not had any crashes!
```

Not Finished

```
dataset_raw %>% group_by(setup,mode) %>% filter(ended == 0) %>% count(ended)  
  
## # A tibble: 2 x 4  
## # Groups:   setup, mode [2]  
##   setup mode   ended     n  
##   <chr> <chr>   <int> <int>  
## 1 amqp  unreliable     0     1  
## 2 mqtt  reliable       0     1  
  
library(ggplot2)  
require(RColorBrewer)  
  
unreliable = dataset %>% filter(mode == "unreliable")  
reliable = dataset %>% filter(mode == "reliable")  
  
condColorSet = brewer.pal(length(unique(unreliable$setup)), name="Paired")  
names(condColorSet) = unique(unreliable$setup)  
  
ggplot(unreliable, aes(x=setup,y=missing, fill=setup)) +  
  geom_boxplot() + scale_fill_manual(values=condColorSet) +  
  scale_x_discrete(name = "unreliable setup") +  
  ggtitle("Unreliable Setup")
```



```
ggplot(reliable, aes(x=setup, y=missing, fill=setup)) +
  geom_boxplot() + scale_fill_manual(values=condColorSet) +
  scale_x_discrete(name = "reliable setup") +
  ggtitle("Reliable Setup")
```



Clean Up of the Data

Now we clean up the data: We remove the 25% outliers on each end.

```
library(tidyverse)

extended = dataset %>%
  mutate(test=paste(setup, mode, sep="_"))

temp = extended %>%
  group_by(test) %>%
  group_modify(~{
    quantile(.x$missing, probs = c(0.25, 0.75)) %>%
    as.integer() %>%
    tibble::enframe()
  }) %>%
  spread(name, value)

colnames(temp)[2:3] = c("lower", "upper")
with_quratiles = left_join(extended, temp, by = "test")

no_outliers = with_quratiles %>%
  filter(missing >= lower & missing <= upper)

no_outliers %>%
  group_by(setup, mode) %>%
```



```
summarise(mean = mean(missing), "test runs" = n())
```

```
## # A tibble: 8 x 4
## # Groups:   setup [4]
##   setup      mode      mean 'test runs'
##   <chr>      <chr>    <dbl>      <int>
## 1 amqp      reliable      0          41
## 2 amqp      unreliable    0          30
## 3 amqp-mqtt reliable      0          41
## 4 amqp-mqtt unreliable 13413.      21
## 5 mqtt      reliable      0          39
## 6 mqtt      unreliable  8046.      19
## 7 mqtt-amqp reliable      0          40
## 8 mqtt-amqp unreliable  7642.      20
```

Now, we make some tests to check if the cleanup is correct.

```
with_quartiles %>%
  filter(setup == "mqtt-amqp" & mode == "unreliable") %>%
  filter(missing < lower | missing > upper) %>%
  select(missing, lower, upper)
```

```
## # A tibble: 20 x 3
##   missing lower upper
##   <int> <int> <int>
## 1   6549  6801  8499
## 2   9341  6801  8499
## 3   6186  6801  8499
## 4   6684  6801  8499
## 5   6696  6801  8499
## 6   6722  6801  8499
## 7   6781  6801  8499
## 8   6726  6801  8499
## 9   6705  6801  8499
## 10  6687  6801  8499
## 11  6504  6801  8499
## 12  8600  6801  8499
## 13  8631  6801  8499
## 14  8888  6801  8499
## 15  8546  6801  8499
## 16  8512  6801  8499
## 17  8733  6801  8499
## 18  8712  6801  8499
## 19  8541  6801  8499
## 20  8846  6801  8499
```


G. Developer Experience Showcase: Implementation Steps

The following steps are required to tailor the CloudEvents Router to the CARU Device:

1. Implementation of the custom GPIO port
2. Creation of a binary with all the needed plugins
3. Creation of the configuration files
4. Definition of the routing rules

We will explain these steps in the following sections.

Implementing the GPIO Port

Listing 4 shows the slightly simplified (the imports are hidden) source code of the GPIO port. We will guide through the code and highlight the developer experience improvements that we made.

Like for any other port implementation, the most important part is the start function (here `port_touch_start`). This function implements the `InternalServerFn` signature which allows it to be called by the Microkernel (via the scheduler plugin). The `InternalServerFn` requires the function to accept three parameters: The `id`, the port's `inbox` and the `sender_to_kernel`. The `id` is a string that identifies the port instance. The `inbox` is an in-memory channel over which the port receives all the messages from the Microkernel. Lastly, the `sender_to_kernel` parameter is an in-memory channel to send messages to the Microkernel (the channel is only an in-memory channel because of the used runtime plugin). The main task of the start function is to listen to the messages that are sent from the Microkernel and react to them.

When the CloudEvents Router is launched, the start function of the port (`port_touch_start`) is invoked and the port starts listening to its `inbox`. The first message that will arrive is the `BrokerEvent::Init` message to indicate that all plugins were initialized and that the Microkernel has started to listen to messages on the `sender_to_kernel` channel. The `BrokerEvent::Init` event is not relevant for this port because it needs to wait until the configuration loader plugin provides the configuration. As soon as the configuration arrives (`BrokerEvent::ConfigUpdated`), the port parses the configuration (`gpio_config()`) and starts listening to the GPIO pin (`listen_to_gpio()`).

The configuration parser function (`gpio_config()`) shows one of the developer experience improvements we added. The configuration is provided to each port as a Rust enum of the type `Config`. This means that a `Config` instance could be a key-value pair list, an array, a number, a string, or a boolean. This gives plenty of flexibility to define the shape of the configuration, but also puts the burden of extracting the configuration values out of this generic `Config` onto the port. To help with that, we introduced the `get_op_val_<type>()` methods, which help to extract values out of a `Config` that is expected to be key-value pairs. Before these helper methods were introduced, multiple lines of code were needed to extract the values in a safe manner. This reduced the readability of the code and made the implementation of a config parser unnecessarily verbose.

The code to listen to the GPIO pin is straightforward: The port reads the pin in a configurable interval and every time the pin flips, the port will publish a CloudEvent to signal the start or the end of a

button press. The only noteworthy detail is that the loop that listens to the GPIO pin is running in a separate thread because otherwise, the loop that listens to messages from the Microkernel would be blocked.

This could change in the future, if the new asynchronous API of Rust would be adopted for the CloudEvents Router. The asynchronous programming model allows multiple input-output-intensive tasks to run in the same thread.

Another improvement we introduced is the `InternalServerFnRefStatic` type. We will see in the next section how the `PORT_TOUCH` function pointer will help to add and remove ports without recompiling the binary.

Compiling the Binary

After the custom GPIO port is implemented, the next step is to build a binary that can be deployed to the CARU Device. The biggest improvement in this step is that the Microkernel and all the plugins are available on the official Rust package registry and do not have to be integrated via Git.

For this use case we need the following packages:

1. The Microkernel¹
2. The loader to add and remove ports with a config file²
3. The threading runtime³
4. The rule-based routing plugin⁴
5. The config file loader plugin⁵
6. The Mosquitto-based MQTT port⁶
7. The custom GPIO port⁷

Listing 4 shows how all these packages are wired together with the help of the new `cerk_loader_file` package. The `cerk_loader_file` package adds the utilities that need to be used with the previously mentioned `InternalServerFnRefStatic` function pointer. This allows to select the actually used plugin from the list of available plugins with the help of a configuration file. We will discuss this configuration file in the next section. Before the `cerk_loader_file` was introduced, the used plugins had to be statically defined and required a recompilation of the binary if they needed to be changed.

Because we are using the Mosquitto-based MQTT port, we also have to add two special environment variables before we build the binary. We have to set `MOSQUITTO_GIT_URL` and `MOSQUITTO_GIT_HASH` to point to our fork of Eclipse Mosquitto. Without this, the router would not be able to provide any delivery guarantees (see Section 4.2.1 "Second Attempt: Patch the Eclipse Mosquitto Library").

The toolchain of Rust makes it extremely easy to cross-compile the program for the ARM-based CPU architecture of the CARU Device. Unfortunately, the Eclipse Mosquitto library is written in C and does not use the Rust toolchain. Although our Rust wrapper library that provides access to the C API compiles the C code for us, it, unfortunately, does not pass the cross-compilation parameter from the Rust toolchain to the C compiler. To make the cross-compilation for the C code work, we have to set an additional environment variable: `MOSQUITTO_CROSS_COMPILER`.

¹<https://crates.io/crates/cerk>

²https://crates.io/crates/cerk_loader_file

³https://crates.io/crates/cerk_runtime_threading

⁴https://crates.io/crates/cerk_router_rule_based

⁵https://crates.io/crates/cerk_config_loader_file

⁶https://crates.io/crates/cerk_port_mqtt_mosquitto

⁷https://github.com/caruhome/cortex_router

Creating the Configuration Files

The CloudEvents Router executable is now compiled and ready for deployment. Only the preparation of two configuration files remains: The first file is `init.json`, which is used by `cerk_loader_file` to determine which of the available plugins should be used and which ports should be instantiated. The second one is `config.json`, which is used by `cerk_config_loader_file` to configure the routing and port plugins.

Listing 6 shows the `init.json` for this use case. The lists `schedulers`, `routers`, and `config_loaders` allow us to define different plugins per type, which could then be loaded by the `cerk_loader_file`. We only defined one per type in the `main()` function, which are the ones we will be using. For the ports, we define the five ports described in Section 5.4.1 "Test Method": `cortex_inbox`, `cortex_outbox`, `cloud_inbox`, `cloud_outbox`, and `touch`.

Listing 7 shows the `config.json` file with its two main sections: One for the routing rules and one for port configurations. For better readability, the routing rules are extracted into Listing 8.

The port section holds the configuration values for each of the ports. For the MQTT ports, all the required information for the connection and interaction with the MQTT broker is provided. For the `touch` port, the number of the GPIO pin and the interval between reads is also provided.

The `cerk_router_rule_based` plugin expects the routing rules to be a JSON object encoded as a string.

The routing rules read as follows: All events from a source that exactly matches `io.caru.cloud` or have a type that starts with `io.caru.device.button_press` must be routed to the `cortex_inbox` port. All events which have a type of `io.caru.alarm` must be routed to the `cloud_inbox` port.

In the bigger picture, this means events that come from the cloud or the GPIO port will be delivered to the State Machine Service, and events that are related to the alarm call feature will be forwarded to the cloud.

```

1  fn parse_config(config: Config) -> Result<GpioConfig> {
2      return Ok(GpioConfig {
3          gpio_num: config.get_op_val_u8("gpio_num")?
4              .unwrap()
5              .into(),
6          interval: config.get_op_val_u32("interval_millis")?
7              .map(|v| Duration::from_millis(v.into()))
8              .unwrap_or(DEFAULT_INTERVAL)
9      });
10 }
11
12 fn listen_to_gpio(id: InternalServerId, config: GpioConfig, sender_to_kernel: BoxedSender) {
13     thread::spawn(move || {
14         let mut gpio = SysFsGpioInput::open(config.gpio_num).unwrap();
15         let mut last_value = GpioValue::Low;
16         loop {
17             let value = gpio.read_value().unwrap();
18             match (value, last_value) {
19                 (GpioValue::High, GpioValue::Low) => sender_to_kernel.send(new_event(&id,
↪ "io.caru.device.button_press.started".into())),
20                 (GpioValue::Low, GpioValue::High) => sender_to_kernel.send(new_event(&id,
↪ "io.caru.device.button_press.ended".into())),
21                 _ => {}
22             }
23             last_value = value;
24             thread::sleep(config.interval);
25         }
26     });
27 }
28
29 fn new_event(id: &InternalServerId, name: String) -> BrokerEvent {
30     let event_id = Uuid::new_v4();
31     BrokerEvent::IncomingCloudEvent(IncomingCloudEvent {
32         routing_id: event_id.to_string(),
33         incoming_id: id.clone(),
34         cloud_event: EventBuilderV10::new()
35             .id(event_id.to_string())
36             .ty(name)
37             .time(Utc::now())
38             .source(format!("io.caru.device.{}.button", DEVICE_ID))
39             .build()
40             .unwrap(),
41         args: CloudEventRoutingArgs {
42             delivery_guarantee: DeliveryGuarantee::BestEffort,
43         },
44     })
45 }
46
47 pub fn port_touch_start(id: InternalServerId, inbox: BoxedReceiver, sender_to_kernel: BoxedSender) {
48     info!("start touch port with id {}", id);
49     loop {
50         match inbox.receive() {
51             BrokerEvent::Init => {
52                 info!("{}", "initiated", id);
53             }
54             BrokerEvent::ConfigUpdated(config, _) => {
55                 info!("{}", "received ConfigUpdated", &id);
56                 let gpio_config = parse_config(config).unwrap();
57                 listen_to_gpio(id.clone(), gpio_config, sender_to_kernel.clone_boxed());
58             }
59             broker_event => warn!("{}", "event {} not implemented", broker_event),
60         }
61     }
62 }
63
64 pub static PORT_TOUCH: InternalServerFnRefStatic = &(port_touch_start as InternalServerFn);

```

Listing 4: GPIO Port Source Code

```

1  #[macro_use]
2  extern crate cerk_loader_file;
3
4  use cerk_loader_file::{start, ComponentStartLinks};
5  use cerk_runtime_threading::THREADING_SCHEDULER;
6  use cerk_router_rule_based::ROUTER_RULE_BASED;
7  use cerk_config_loader_file::CONFIG_LOADER_FILE;
8  use cerk_port_mqtt_mosquitto::PORT_MQTT_MOSQUITTO;
9  use touch_port::PORT_TOUCH;
10
11 fn main() {
12     start(ComponentStartLinks {
13         schedulers: fn_to_links![THREADING_SCHEDULER],
14         routers: fn_to_links![ROUTER_RULE_BASED],
15         config_loaders: fn_to_links![CONFIG_LOADER_FILE],
16         ports: fn_to_links![PORT_MQTT_MOSQUITTO, PORT_TOUCH],
17     });
18 }

```

Listing 5: Device Router Source Code

```

1  {
2      "scheduler": "THREADING_SCHEDULER",
3      "router": "ROUTER_RULE_BASED",
4      "config_loader": "CONFIG_LOADER_FILE",
5      "ports": {
6          "touch": "PORT_TOUCH",
7          "cortex_inbox": "PORT_MQTT_MOSQUITTO",
8          "cortex_outbox": "PORT_MQTT_MOSQUITTO",
9          "cloud_inbox": "PORT_MQTT_MOSQUITTO",
10         "cloud_outbox": "PORT_MQTT_MOSQUITTO"
11     }
12 }

```

Listing 6: Device Router Init Config

```

1  {
2    "routing_rules": "{\\"cortex_inbox\\":{\\"Or\\":[{\\"Exact\\":[...],
3    "ports": {
4      "touch": {
5        "gpio_num": 8,
6        "interval": 50
7      },
8      "cortex_inbox": {
9        "host": "tcp://localhost:1883",
10       "send_topic": "cortex/inbox"
11     },
12     "cortex_outbox": {
13       "host": "tcp://localhost:1883",
14       "subscribe_topic": "cortex/outbox",
15       "subscribe_qos": 1
16     },
17     "cloud_inbox": {
18       "host": "tcp://localhost:1883",
19       "send_topic": "cloud/inbox"
20     },
21     "cloud_outbox": {
22       "host": "tcp://localhost:1883",
23       "subscribe_topic": "cloud/outbox",
24       "subscribe_qos": 1
25     }
26   }
27 }

```

Listing 7: Routing Rules and Port Configs

```

1  {
2    "cortex_inbox": {
3      "Or": [
4        {
5          "Exact": [
6            "Source",
7            "io.caru.cloud"
8          ]
9        },
10       {
11         "StartsWith": [
12           "Type",
13           "io.caru.device.button_press"
14         ]
15       }
16     ]
17   },
18   "cloud_inbox": {
19     "Exact": [
20       "Type",
21       "io.caru.alarm"
22     ]
23   }
24 }

```

Listing 8: Device Router Routing Rules and Port Configs

H. Used Tools

In this appendix we describe the tools used to write the thesis and to create our product. This appendix was originally written in the context of the student research project "CloudEvents Router".[1] As we had the same setup, we reused all the same tools for our thesis. The most significant changes from the student research project to this thesis were:

Documentation CI Our documentation is no longer built by the Buildkite CI but by GitHub Actions. Additionally the CI performed a new task: linting the latex files with check-tex.

IDE In addition to Visual Studio Code IDE, we also used the CLion IDE from JetBrains for writing Rust code.

Diagrams We used Lucidchart for some visually challenging images.

General Tools

The tools in this section were used both for the documentation as well as the product development.

GitHub

For our project hosting we decided to use GitHub.

GitHub is a source code management and collaboration platform based on Git. It is a well known platform for open-source code and was acquired in 2018 by Microsoft.[59]

We decided to use GitHub because it is the most used platform for hosting open-source code repositories.

GitHub's project management functionalities are quite limited. There are issues and milestones. Working with them is quite challenging when working on bigger projects, but it was just enough for the size of our project.

T-Metric

We used T-Metric for time tracking.

T-Metric is a time tracking app with a functionally limited free plan for up to 5 people. It has integrations for various products, such as GitHub.[112]

The tool is quite simple, and so are the reporting functions. The collected data can be exported as a CSV file or a PDF.

Tools for the Documentation

This document was written with \LaTeX .

” \LaTeX , which is pronounced «Lah-tech» or «Lay-tech» (to rhyme with «blech» or «Bertolt Brecht»), is a document preparation system for high-quality typesetting. It is most often used for medium-to-large technical or scientific documents but it can be used for almost any form of publishing.”[77] \LaTeX is distributed under the \LaTeX project public license. It is a free software license.[78]

Both team members were familiar with \LaTeX prior to the start of this thesis, but we both improved our skills in this text writing format.

We prefer \LaTeX over common graphical text processors like Word because the files are plain text and are therefore suitable to be versioned with source control tools like Git. The advantage of using Git for versioning the documentation is that we were able to have the same GitHub pull request based review procedure as we use for our code. This procedure ensures that both partners have read every bit of text before it is added to the master branch.

Code Highlighting

We used the minted \LaTeX extension for code syntax highlighting in our documents. Minted supports Rust syntax highlighting for inline code snippets or code from an external file.[120]

Diagrams

The diagrams in the documentation were not directly built by \LaTeX . Instead, they were created with PlantUML and then included in the \LaTeX documents as images.

PlantUML is a tool to generate different UML and non UML diagrams. The diagrams are generated by writing with the PlantUML Language. The tool is written in Java and has various output formats, one of them is JPGE.[95]

PlantUML source files are also plain text and therefore support our Git versioning strategy and GitHub pull request review procedure.

In addition to PlantUML we added Lucidchart to our diagram tools. Lucidchart is a graphical online tool to draw diagrams, processes, and systems.[81] The primary advantage of Lucidchart is that it is easier to modify the visual output according to our wishes. However, changes cannot be versioned in a comprehensible way by Git because its file format is not text-based.

CI

A disadvantage of writing the documentation in \LaTeX and not in a graphical editor is that it has to be compiled. We encountered a few situations where some source files became corrupted and refused to compile. Resolving these issues can be quite frustrating because the error messages and warnings are very cryptic. \LaTeX source files are not very readable for people who are not familiar with the syntax.

In order to counteract these points we have built a CI to build the PDF files automatically on every Git push.

The CI has two jobs:

build-pdfs Builds the PDFs and the PlantUML diagrams.

1. build the diagrams

2. build the pdfs

check-tex This job does some basic linting actions (static code analysis) of the \LaTeX code. The tool "ChkTeX" was used for this.

This Job was newly introduces in this thesis. We had no linting actions in the last project.

For the CI we used GitHub Actions.

In the last project, we used Buildkite. We changed it because GitHub actions are free, unlike Buildkite. We only now started working with GitHub Actions as it was still in its beta phase during our last project.[121]

Tools for the Product

Our program is written in Rust, so our main tool was the rust compiler: `rustc`.

`rustc` is the Rust compiler provided by the Rust Team. It can be used directly (`rustc`) or with Cargo (`cargo build`).[106]

The package management was done with Cargo.

Cargo is the default package manager for Rust. It is really simple, compiles code, installs packages, runs tests, formats source files and generates the documentation. Cargo can be extended with additional subcommands by installing packages via Cargo itself.

rustup

We installed and switched between Rust versions with `rustup` (Rust has a release cycle of 6 weeks[107]).

`rustup` is a tool to install different versions of Rust, like stable, beta and nightly, and for different compilation targets.

It is the recommended way for developers to install Rust.[107]

IDE

The language is not widely used, which is why IDE support is suboptimal. However, there is a good plugin for Visual Studio Code called "Rust (rls)" which shows error, adds some navigation features and shows some documentation.[122] Additionally, the errors from the compiler are already very helpful, which makes a deep IDE-integration less necessary than for other languages.

In addition to the Visual Studio Code IDE, we started to use the CLion IDE from JetBrains in this thesis. The IDE is primarily designed for the use with C code. However, JetBrains provides an official "Rust" plugin that introduces support for Rust and Cargo. They are adding navigation features, show some documentation, highlight errors, and add some auto completion functionality.

CI

We used Buildkite to automatically run the tests for our code after each push.

Buildkite is a service for coordinating buildpipelines which get executed on own infrastructure. They provide a build agent that can easily be deployed to AWS and other platforms. Buildkite is well integrated with GitHub.[39]

Statistical Tools

We used R with the IDE RStudio. R is a programming language for statistical computations. It is licenced under the GNU 2 and GNU 3 language.[99, 100]

To facilitate the setup and make all steps of the R script reproducible, we used R in a Docker container with RStudio Server.¹ With this workflow the operating system and the version of all packages are always the same.

¹<https://hub.docker.com/r/rocker/verse>

I. Code Appendixes Listings

Code Appendixes Listings

This appendix gives an overview over all code changes that were made during the bachelor thesis. All code changes are versioned with Git and are uploaded on GitHub. However, not all code repositories are public. They are grouped into documentation, CloudEvents Router, and the domain languages repositories that were updated.

Documentation

ce-rust/ba The main bachelor thesis repository.

Visibility Private

URL <https://github.com/ce-rust/ba>

Changes Created during this thesis

ce-rust/docker-latex The Docker image source for the L^AT_EXbuild container.

Visibility Public

URL <https://github.com/ce-rust/docker-latex>

Changes Minimal updates during this thesis

ce-rust/docker-plantuml The Docker image source for the PlantUML build container.

Visibility Public

URL <https://github.com/ce-rust/docker-plantuml>

Changes Minimal updates during this thesis

ce-rust/docker-chktex The Docker image source for the ChkTex L^AT_EXlinter container.

Visibility Public

URL <https://github.com/ce-rust/docker-chktex>

Changes Created during this thesis

CloudEvents Router

ce-rust/cerk The repository of the CloudEvents Router.

Visibility Public

URL <https://github.com/ce-rust/cerk>

Changes Created during this thesis

caruhome/cortex_router The example router use case for the developer experience test.

Visibility Public

URL https://github.com/caruhome/cortex_router

Changes Created during this thesis

ce-rust/mosquitto-client-wrapper **Visibility** Public

URL <https://github.com/ce-rust/mosquitto-client-wrapper>

Changes Forked from GitHub and updated during this thesis

ce-rust/mosquitto The forked Eclipse Mosquitto repository.

Visibility Public

URL <https://github.com/ce-rust/mosquitto>

Changes Forked from GitHub and updated during this thesis

eclipse/mosquitto The official Eclipse Mosquitto repository. Here, a pull request was proposed.

Visibility Public

URL <https://github.com/eclipse/mosquitto>

Changes Pull request created: <https://github.com/eclipse/mosquitto/pull/1932>

ce-rust/docker-rust-cerk An extension of the Rust Docker image.

Visibility Public

URL <https://github.com/ce-rust/docker-rust-cerk>

Changes Minimal updates during this thesis

ce-rust/paho.mqtt.rust The forked Eclipse Paho MQTT wrapper. It has been modified to use the forked Eclipse Paho MQTT library instead of the original.

Visibility Public

URL <https://github.com/ce-rust/paho.mqtt.rust>

Changes Forked from GitHub and updated during this thesis

ce-rust/paho.mqtt.c The forked Eclipse Paho MQTT library. Here, the first Message Queuing Telemetry Transport (MQTT) patch was created.

Visibility Public

URL <https://github.com/ce-rust/paho.mqtt.c>

Changes Forked from GitHub and updated during this thesis

AsyncAPI

asyncapi/bindings The bindings repository of the AsyncAPI specification. Here, a pull request to the MQTT binding was proposed.

Visibility Public

URL <https://github.com/asyncapi/bindings>

Changes Pull request created: <https://github.com/asyncapi/bindings/pull/42>

MDSL

socadk/MDSL In mainline repository of the Microservice Domain-Specific Language (MDSL) specification. Here, the AsyncAPI was reviewed before it was public and changes were proposed.

Visibility Private

URL <https://github.com/socadk/MDSL>

Changes Pull request created: <https://github.com/socadk/MDSL/pull/63>