Reliable Event Routing in the Cloud and on the Edge

An Internet-of-Things Solution in the AgeTech Domain

Linus Basig[1,2], Fabrizio Lazzaretti[1], Reto Aebersold[2], and Olaf Zimmermann[1]

[1] University of Applied Sciences of Eastern Switzerland (OST), Oberseestrasse 10, 8640 Rapperswil, Switzerland
{linus.basig,fabrizio.lazzaretti}@lifetime.hsr.ch, olaf.zimmermann@ost.ch
[2] CARU AG, Weberstrasse 3, 8004 Zürich, Switzerland
{linus.basig,reto.aebersold}@caruhome.com

Abstract. The AgeTech domain poses both opportunities and challenges for the architects of Internet of Things (IoT) solutions. Key requirements concern data privacy and cost-efficiency, as well as guaranteed, interoperable delivery of event messages from resource-constrained edge devices to the cloud and consumer applications; public cloud deployments and asynchronous, queue-based messaging are common building blocks of such solutions. This experience report features the conceptual, pattern-oriented design and open source implementation of a custom-built embeddable event router that natively supports the CloudEvents specification from the Cloud Native Computing Foundation. Our CloudEvents Router introduces standardized event routing targeting the software landscape of the AgeTech startup CARU; this landscape reaches from an embedded microcontroller to cloud-scale services running on Amazon Web Services. The report further analyses the message delivery guarantees given in messaging protocol specifications and their implementations (for instance, MQTT libraries) and presents an alternative delivery management approach, implemented and validated in the open sourced Rust microkernel of the CloudEvents router. The report concludes with lessons learned about messaging edge cases and protocol particularities, in MQTT in particular.

Keywords: Asynchronous Messaging · Enterprise Application Integration · Cloud Native Computing · Internet of Things · Quality of Service Guarantees

1 Introduction

The AgeTech domain poses both opportunities and challenges for the architects of Internet of Things (IoT) solutions. Key requirements concern data privacy and cost-efficiency, as well as guaranteed, cross-platform delivery of event messages from edge devices to the cloud and applications; asynchronous, queue-based messaging and public cloud deployments are common building blocks of such

solutions. CARU AG is a Swiss startup targeting this domain. One of its main products is a digital flatmate, an IoT device comprising many sensors connected to Amazon Web Services (AWS) cloud offerings and consumer applications.

This experience report features the domain analysis, design, and implementation of the content-based CloudEvents Router, whose main design goal is to provide a unified event plane to route standardized events conveniently and reliably over multiple messaging protocols such as Message Queuing Telemetry Transport (MQTT), Amazon Simple Queue Service (SQS), and Advanced Message Queuing Protocol (AMQP). We evaluated existing solutions, but did not find any solution that can be deployed both to low-power devices and to the cloud while fitting into the overall software architecture of CARU. Our novel CloudEvents Router is written in Rust and open sourced under the Apache 2 license.[3]

The remainder of the paper is structured in the following way: Section 2 introduces architecture design challenges in the AgeTech domain, evidenced and exemplified in the business model, products, and software architecture of CARU. Section 3 specifies the CloudEvents router architecture that addresses these challenges; Section 4 covers reliability and delivery guarantees and their implementation in Rust and MQTT. Sections 5 and 6 present lessons learned and related work; Section 7 summarizes and concludes the paper.

2 Business Context: AgeTech Startup with IoT Device

CARU is an AgeTech startup with the mission to help the elderly live independently for longer by providing a digital flatmate in the form of an IoT device. It aims at increasing the safety of its human users by a) allowing these users to call for help in an emergency, b) facilitating communication with relatives and caretakers, and c) integrating services that foster a comfortable life at home.

When comparing the CARU Device against traditional social care alarm systems, its main differentiator is that the alarm call can be triggered by the embedded voice recognition model. The CARU Device also integrates the relatives into the care process by allowing them to exchange voice messages with the elderly user via the "CARU Family Chat" smartphone app. The device is also equipped with several sensors that support data-driven functionalities like air-quality monitoring, sleep quality analysis or activity detection that can be used by professional caretakers to optimize the care they provide.

2.1 Software Architecture Overview

Fig. 1 shows an overview of the current software architecture of CARU. It consists of four main components: (1) the *CARU Device* running embedded Linux, (2) the *CARU Cloud* backend running on AWS, (3) the *myCARU* Web App(lication), and (4) the *CARU Family Chat* smartphone app. The two apps

[3] https://github.com/ce-rust/cerk

communicate with the backend over a GraphQL endpoint managed by AWS AppSync; the MQTT protocol connects the devices and the backend via AWS IoT Core.

The use of a public cloud such as AWS allows CARU to build upon its many managed services and therefore minimize the operational burden on their small DevOps team. AWS IoT Core, in particular, provides many of the building blocks required to successfully manage an IoT device at scale. This includes device identity management (e.g., certificate management), connectivity (e.g., serverless MQTT broker), remote device configuration, and data management. AWS AppSync was chosen because it lets CARU run a fully serverless backend that automatically scales and does not require infrastructure maintenance.

As the CARU Device is a safety-critical device and its users depend on it in case of a fall or another medical issue, measures are built into the architecture to reduce the risk of a device going offline. The main challenge is that the device is deployed in the home of its user and relies on infrastructure outside the control of CARU. Two factors in particular can cause a device to go offline: power outages and issues with the mobile network.

To mitigate the risk of a power outage, the CARU Device is equipped with a battery that allows it to operate normally for a few hours without an active power supply. The main strategy to reduce the risk of mobile network-related issues is the use of a roaming SIM that automatically connects to the best network available at the location of the device. Additionally, the CARU Device is also able to connect to WiFi networks for added redundancy; that said, most users of the CARU Device do not have one installed in their home at present.

Because of the uncontrollable environment of the device, strong emphasis is put on monitoring it from the cloud. If a device goes offline or connectivity issues are recognized, the relatives or caretakers are alerted immediately and provided with instructions how to resolve the issue.

Finally, the safety-critical functionality (emergency call) does not depend on an active Internet connection. The recognition of the keyword to trigger the emergency call runs on the device, and the call itself can be made over the 2G/GSM mobile network as a fallback. The 2G/GSM network has a better coverage (at least for now; in Switzerland some providers started to turn it off) and is able to place calls even if the signal is not strong enough to transport usable Internet traffic. Important events that are generated while the device is offline are stored and sent to the cloud as soon as the device comes back online.

2.2 Vision: Unified Event Plane

The remainder of this experience report focuses on two components of the architecture of CARU: the CARU Device and CARU Cloud (Fig. 1). In the future, the concepts introduced in this paper could be applied to the other architectural components as well.

Because the CARU Device can place calls even if the signal quality is too bad to connect to the Internet, it was decided early on to give the device as much independence from the cloud as possible. To support this decision and because

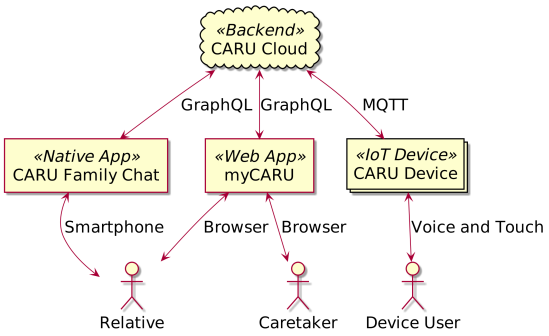


Fig. 1. Architecture Overview of the CARU System Landscape

of the nature of the data that the device produces (i.e., domain events reporting that something business-relevant has happened), an event-driven architecture was chosen. This design worked well in the beginning; with a growing number of events, several issues arose.

Event design and process mismatches. The first issue arose because only minimal effort was put into the design of the events when CARU started the development. This was great for iterating fast on the product but caused some pain when the number of productive devices grew. This pain was amplified because the events are the only interface between the CARU Device and CARU Cloud, and these two systems have vastly different deployment cycles: while the CARU Cloud is developed with a continuous deployment approach, the deployment cycle of the CARU Device is longer because the cost of each deployment is significantly higher. The higher cost is caused by the more manual testing process, the more involved communication with the elderly device users, and the expensive bandwidth required to bring the update onto the devices.

Routing intricacies. The second issue was that the event routing patterns in each component turned out to be quite different. Hence, unnecessary friction occurred when a developer switched from working on one system to another: between the CARU Device and CARU Cloud, the topic-based subscription mechanism of the MQTT protocol was used. To keep the design of the CARU Device simple, an in-process message bus with type-based event routing distributed the events. In the CARU Cloud, AWS EventBridge with its content-based subscriptions was responsible for event routing. The mental friction caused by switching from one component to the other slowed down the development because CARU has a small DevOps team that often has to work on different systems in parallel.

Adoption of CloudEvents specification to the remedy. To make the content-based event routing in/with AWS EventBridge more convenient, CARU introduced CloudEvents as its standardized event format for events exchanged inside the CARU Cloud. CloudEvents is a specification proposed by the Serverless Working Group of the Cloud Native Computing Foundation (CNCF)[6]. The goal of the specification is to simplify event declaration and delivery across

```
{
        "type": "caru.sensor.voice",
        "specversion": "1.0",
        "source": "crn:eu:::device:wvugxd5t",
        "id": "e30dc55b-b872-40dc-b53e-82cddfea454c",
        "contenttype": "text/plain",
        "data": "help"
}
```

Listing 1: A CloudEvent in the JSON Serialization Format

services, platforms, and vendors. To achieve this goal, the specification defines the event fields, different serialization formats, and multiple protocol bindings. The specification has attracted 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) format.

The introduction of the CloudEvents event format also helped with the management and the safe evolution of the events by encouraging strict definition and versioning of schemas defining the event payload.

The Unified Event Plane. After the successful introduction of CloudEvents in the CARU Cloud, the vision of a *Unified Event Plane* emerged. The *Unified Event Plane* aims at letting events flow with as little friction as possible between the different components of the CARU architecture, which was introduced in Section 2.1. This should be achieved by introducing the CloudEvents event format and a content-based event router (CloudEvents Router) in all systems.

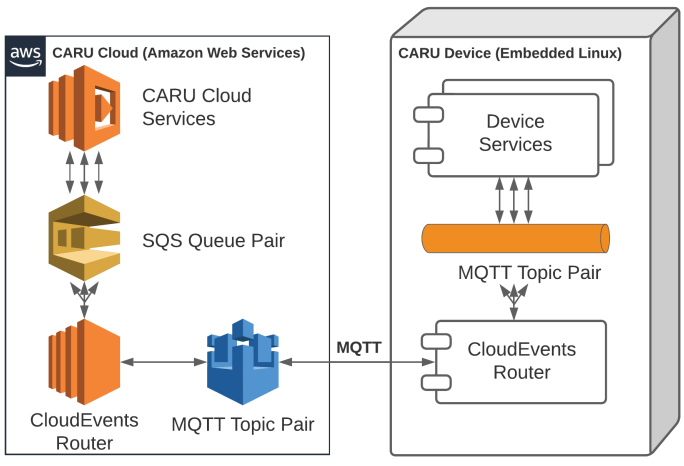


Fig. 2. CARU's Unified Event Plane

Fig. 2 shows the envisioned Unified Event Plane. Whenever events are exchanged, they are formatted according to the CloudEvents specification. CloudEvents Router instances appear in the CARU Cloud and the CARU Device systems. The CloudEvents Routers connect to each other as well as to the different services of their respective systems. To keep complexity minimal, the CloudEvents Routers are stateless and use existing services to handle the event transport: on the device, an embedded Eclipse Mosquitto MQTT Broker has this duty; in the cloud, Amazon SQS queues are responsible for reliable event delivery. This also means that the CloudEvents Routers must be able to route messages between the protocols supported by these services reliably and in an interoperable manner. For each CARU Device and CARU Cloud Service, an inbox and an outbox channel pair are used to communicate with the CloudEvents Router. In the cloud, this is implemented by an SQS Queue Pair; on the device and between the device and the cloud, an MQTT Topic Pair is used.

3 Design and Implementation of the CloudEvents Router

This section presents the architecture and implementation of the CloudEvents Router and shows how the design goals in Section 2.2 are reached. From the beginning, the project had a focus on a lean and easily extendable architecture as well as a flexible deployment on IoT devices and in the cloud. Later on, the router was extended to provide an *At Least Once* delivery guarantee.

3.1 Technical Constraints and Requirements

Before the decision to design and implement the CloudEvents Router was taken, we evaluated existing solutions. The following non-functional requirements and constraints guided the evaluation:

CloudEvents Support. CARU has already invested in CloudEvents and has gained a positive experience with it so far. The solution must be able to make routing decisions based on the fields defined in the CloudEvents specification.

Reliable Messaging. The CARU Device is a safety-critical device. Therefore the solution must be able to give some delivery guarantees for the events it processes. The current delivery guarantee provided by the used infrastructure (AWS IoT Core) is *At Least Once*. The CloudEvents Router must provide the same or better delivery guarantees.

Support for MQTT 3.1.1. To leverage existing messaging infrastructure on the CARU Device (Eclipse Mosquitto MQTT Broker) and in the cloud (AWS IoT Core), the solution must be able to interact with MQTT 3.1.1 endpoints. This enables a smoother transition from the current state into the direction of the unified event plane.

Runs on the CARU Device. The device has an ARM Cortex-A7 (armv7l) 1 Core CPU clocked at 198 - 528 MHz, 500 MB RAM, and runs a Yocto Linux.

Modularity. The solution should be adaptable to different deployment environments. Additional messaging protocols should be easy to add.

Rust Programming Language (optional). CARU wanted to gain experience with Rust as an alternative to C/C++.

After the evaluation criteria had been defined, ten existing solutions were analyzed.[4] Out of these ten solutions, two (Apache Camel and Node-RED) looked promising. However, both of them do not support CloudEvents out-of-the-box; it is possible to extend them to do so. Unfortunately, their resource requirements were incompatible with the available resources on the CARU Device. As the hardware was already in production and used by customers, upgrading it to satisfy the resource requirements was not an option.

For these reasons, we decided to design and implement a specialized, stateless CloudEvents Router in Rust. Messages are only stored by the incoming and outgoing message channels and never persisted in the CloudEvents Router itself. This eases the deployment of the CloudEvents Router, especially in the cloud; its horizontal scaling is simplified. Furthermore, this design reduces the complexity of the router itself. A negative consequence of this architectural decision is that all important messages have to be persisted before they are processed by the CloudEvents Router, e.g., by an underlying message broker.

3.2 CloudEvents Router Architecture

For our implementation of the CloudEvents Router, we chose the Microkernel architecture[5]. The Microkernel pattern is commonly used as the architectural foundation of modern operating systems. A prominent example is Linux. Our Microkernel defines the interfaces to the four plugin types that implement the bulk of the functionality. The four plugin types are Scheduler, Configuration-Loader, Router, and Protocol-Port.

Besides the four 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, a specific plugin type can be implemented in different ways. The users can then choose the implementation for each plugin type based on their specific use cases.

One reason for choosing the Microkernel pattern was the ability to adapt the router to different deployment environments easily. In the embedded Linux environment of the CARU Device, for example, the router could use a different Configuration-Loader plugin implementation than in the cloud. On the device, the loader implementation could read the configuration from a file while the loader implementation in the cloud fetches it from a database. The modular system design also allows adding support for new messaging protocols by writing plugins that implement the Protocol-Port plugin interface. It also helps to

[4] The ten analyzed messaging products are: CloudEvent Router and Gateway, Knative Eventing v0.9, Pacifica Dispatcher v0.2.3, Serverless Event Gateway v0.9.1, Amazon Simple Notification Service, Apache Camel v2.24.2, Crossbar.io v19.10.1, D-Bus v1.12, Node-RED v1.0.1, RabbitMQ v3.8[2]

optimize the size of the executable by only including the plugins that are required for a specific use case. In Fig. 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 Amazon SQS.

3.3 Implementation: Rust SDK for CloudEvents

When our project started, there was no suitable Rust Software Development Kit (SDK) to work with CloudEvents, so we built our own to implement the open source CloudEvents Router. Later on, we had the opportunity to evolve this SDK and co-design the official CloudEvents Rust SDK[5] in the name of the CNCF Serverless Working Group.

4 Delivery Guarantees and their Implementations

This section investigates the reliable routing process of the CloudEvents Router and explains why the *At Least Once* delivery guarantee was chosen. Before doing so, we recapitulate how reliability is defined in the context of software engineering in general and messaging systems in particular.

ISO 25010 is a suite of standards defining eight characteristics of quality for software engineering and systems[14]. Reliability is one of these eight system/ software product quality characteristics. Availability is a sub-characteristic of reliability and reflects the degree to which a system is operational and accessible. Reliability requires that an action is completed as intended, while availability just requires that the system is operational and reacts to requests somehow[14,17].

In messaging systems, reliability is often defined as a non-functional property, e.g., in MQTT as Quality of Service (QoS)[1]. QoS, in a broader context, often describes much more than just reliability[12,13,17,19]. In our context, reliability is primarily concerned with not losing messages during transport, and with preserving the order of subsequently sent messages during delivery. Dirty reads or other inconsistencies of messages usually are not in scope.

4.1 Reliable Event Routing with the CloudEvents Router

To address the business requirements in the AgeTech domain (Section 2), the CloudEvents Router should be able to route messages between different channels (e.g., MQTT topics, Amazon SQS queues) in a reliable fashion. This means that messages are routed whenever possible (i.e., the configuration is correct, and the destination is ready to accept messages). They are not lost even in case the router crashes.

Fig. 3 shows a simplified example in which it is critical that events are not lost. The embedded classifier of the *Voice Recognition Service* detects the keyword

[5] https://github.com/cloudevents/sdk-rust

`Help` and emits the *Keyword Recognized Event* (see Listing 1). The *CloudEvents Router on the Device* then applies its routing rules and forwards the event to the *Call Service* on the device and the *CloudEvents Router in the Cloud*. The router in the cloud applies its routing rules and forwards the event to the *Alarm Receiving Center* (ARC) that assigns the alarm to a call taker and prepares him/her for the incoming call. In the meantime on the device, the *Call Service* initiates a call to the ARC over the cellular network.

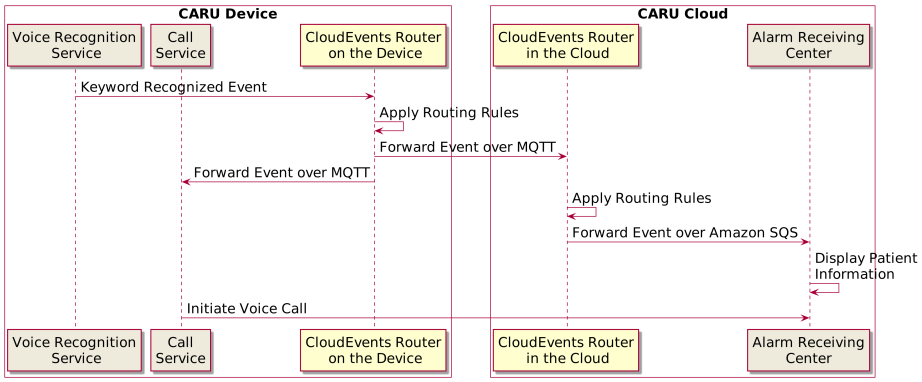


Fig. 3. Example Use Case that Requires Reliability

The literature and most protocols define a common set of delivery guarantees: *At Most Once*, *Exactly Once*, *At Least Once*[9,11]. A first step towards implementing cross-protocol reliable routing was to select a protocol-independent delivery guarantee. With such a delivery guarantee defined, the CloudEvents Router can route messages between message channels that use the same or different messaging protocols while preserving the delivery guarantee.

Most protocols and messaging systems provide subscription- or publication-based delivery guarantees; on the contrary, we propose a single delivery guarantee on the channel-level. A property called "delivery guarantee" should be defined on the channel to remove any confusion as to what guarantees can be expected from any given channel. This property should be used to specify how reliable a channel is. A declarative specification can be added to the messaging contract to signal what the receiver can expect. A producer then knows how it must 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.

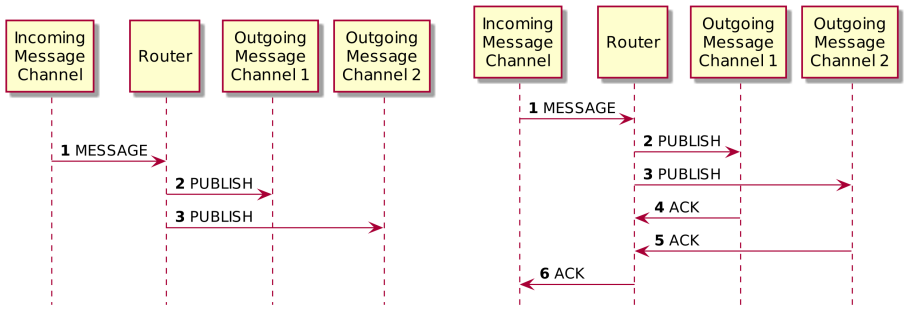
4.2 Different Approaches to Reliable Event Routing (Options)

A stateless router such as the CloudEvents Router should be able to forward event messages from an inbound channel to one or more outbound channels

according to predefined routing rules. All three stated delivery guarantees from Section 4.1 should ideally be supported.

The following examples show the case of one incoming and two outgoing message channels. Message transmission may involve one or more messaging systems (brokers).

At Most Once. We start with the simplest delivery guarantee: *At Most Once*. This delivery guarantee does not satisfy the business need of this router, as the business requires that every message reaches the destination and will be processed there. The flow for a router with *At Most Once* semantics is shown in Fig. 4a: **1.** First, the router gets a message from an incoming channel and then **2., 3.** publishes it to all subscribers.



(a) *At Most Once* Delivery
The message is sent in a fire-and-forget mode; no acknowledgments are sent or waited for.

(b) *At Least Once* Delivery
The acknowledgment (ACK) has to arrive in a certain time frame. If this does not happen, the message will be sent again.

Fig. 4. Routing with Different Delivery Guarantees

At Least Once. This flow guarantees that every message is transmitted to the subscriber at least once; duplicates are permitted. The flow for a router with *At Least Once* is shown in Fig. 4b. The steps are: **1.** The incoming messaging component sends the messages to the router. **2., 3.** The message is then sent to the subscribers. **4., 5.** The router waits for the acknowledgments from the subscribers. **6.** After the acknowledgments are received, the router will know that the message was delivered, enabling it to send the acknowledgment to the sending component.

If any of the steps is not successful or a service crashes during the processing, the sender will restart the process 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. The routing process only ends if the last step, the acknowledgment from the router to the sender (Step 6), was successful. This causes the message to be deleted from the incoming channel, which completes the routing[9].

Exactly Once. This delivery guarantee is hard to achieve, especially when involving different messaging protocols or products. One challenge is to define a point in the transmission process when the transaction of sending a message is completed. In *Exactly Once*, this point has to be coordinated between all participants. If that does not happen (e.g., a participant crashes or does not respond), the message may be submitted multiple times or not at all[9,20,22]. The Two-Phase Commit protocol[23] can help to overcome this challenge. It describes how to persist data on multiple machines with the help of a global system transactions, which comes at a price. The process can still fail for some edge cases and requires a coordinated state that all participants agree upon. This is not feasible in our project and business context. An *Exactly Once* stateless-routing can work for specific protocols, such as Kafka that provide a central coordinator that controls and performs a global commit operation[3,8,18].

4.3 How Common Protocols Implement Delivery Guarantees

We analyzed the messaging protocols for which a protocol binding is defined in the CloudEvents specification. At the time of writing, the specification contained such bindings for MQTT, AMQP, WebHook over HyperText Transfer Protocol (HTTP), Neural Autonomic Transport System (NATS), and Kafka[6]. During the analysis we compared the strategies for providing reliable messaging in these protocols and studied whether it would be possible to combine them in a reliable CloudEvents Router.

We were able to categorize the five protocols into two categories: message-oriented and stream-oriented protocols. Both categories provide similar delivery guarantees, but they take different approaches[3].

Message-Oriented Protocols. Message-oriented protocols provide delivery guarantees on a per-message basis. The receiver sends an acknowledgment for each transferred message (when guaranteeing *At Least Once* delivery). To achieve an *Exactly Once* delivery guarantee, the protocols require the receiver to remember already received messages to prevent duplicates until extra steps are performed to safely remove the message from both the sender and the receiver. MQTT, AMQP, WebHook over HTTP belong to this category of protocols.

Stream-Oriented Protocols. Stream-oriented protocols provide delivery guarantees on a stream of messages where each message can be identified by its position in the stream. The advantage of a stream-oriented protocol is that not every single message must be acknowledged when reading from the stream.

Instead, the receiver can acknowledge its position in the stream from time to time by sending the position of the last successfully processed message. This results in a lower network overhead but requires a central server that manages the stream. For writing to the stream, the same mechanisms as for the message-oriented protocols are used (acknowledgments for *At Least Once* and de-duplication for *Exactly Once*). The Kafka protocol and NATS Streaming protocol, which is built on top of the NATS protocol, fall into this category.

4.4 Selection of *At Least Once* for the CloudEvents Router

As the goal of CARU is to achieve safe message routing so that every message reaches its destination, an *At Most Once* delivery guarantee is not appropriate. So a decision between *At Least Once* and *Exactly Once* had to be taken.

We decided to implement an *At Least Once* delivery guarantee because of the following reasons: (1) The significant complexity of a Two-Phase Commit that is required to achieve an *Exactly Once* delivery guarantee. (2) The lacking support or non-interoperable implementations of *Exactly Once* in the targeted messaging protocols (see Section 4.3). (3) The statefulness that *Exactly Once* requires. (4) The possibility to deduplicate an event based on the `id` field defined in the CloudEvents specification (see next paragraph).

If the solution requirements call for strict *Exactly Once* delivery semantics, messages can be transferred with an *At Least Once* delivery guarantee, ensuring *Exactly Once* processing at the receiver with the help of a stateful Message Filter[11]. Implementing this pattern is straightforward with CloudEvents because these events contain the `id` field that can be used to detect events that were retransmitted and to prevent their reprocessing[6].

4.5 Implementation and Validation of *At Least Once* Channels

Implementation. Our router is able to route messages from one channel to zero or more channels while providing an *At Least Once* or no delivery guarantee (the latter is called *Best Effort*) for each route.

Implementing the AMQP plugin and upgrading from the *Best Effort* delivery guarantee to the *At Least Once* delivery guarantee in our Microkernel was straightforward. However, the upgrade from the *Best Effort* delivery guarantee to *At Least Once* on the MQTT plugin turned out to be rather challenging (see Section 5 for more information).

Validation. Different test methods were applied, from unit tests to integration tests. The main test objective was to gain confidence that the router works as intended and does not loose any messages even under harsh conditions. The most assuring test was an integration and reliability test that we ran both with the *At Least Once* and with the *Best Effort* delivery guarantee. This test setup is visualized in Fig. 5. The setup was an automated script that deploys the CloudEvents Router in a Kubernetes cluster and generates 100'000 events. The router consumes them on Channel A, and publishes them on Channel B. During

the test, the single router instance was killed every 10 seconds by chaoskube and restarted by Kubernetes. chaoskube is a chaos engineering tool that periodically kills random instances in a Kubernetes cluster[6] (in our case, always the router instance). The messages were generated in a 1ms interval during the test so that the test ran for at least 100 seconds. With the *At Least Once* delivery guarantee, all messages were routed successfully to the output channel. Not a single message was lost. In the *Best Effort* routing mode, an average of 7'000 messages were lost per test run. The results differed widely depending on the used protocol-plugins.

With *At Least Once*, we can be sure that no events get lost while still providing a good balance between reliability and performance. 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.

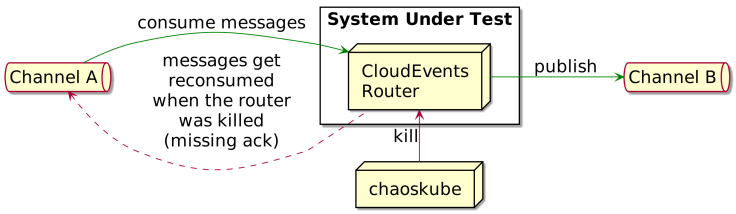


Fig. 5. Integration Test Setup of the Router Inside the Kubernetes Cluster

5 Lessons Learned

The messaging protocols that we worked with handle reliability in slightly different ways. Our router implementation demonstrates that the current design works for MQTT and AMQP; we are confident that it can leverage other messaging protocols as well[2,3]. In this section, we report two of our learnings and findings w.r.t. the delivery guarantees of existing messaging middleware.

Non-Transient Error Handling. All error scenarios merely take transient errors into account. Our router design introduces a retry mechanism; however, there are also permanent errors that can not be resolved by just retrying to route a message if its routing failed. This could, for example, be caused by a wrong configuration or a malformed message. In such a case, the message can never be delivered. Such a problem can lead to a crash loop of a receiver, especially when the delivery guarantees are considered: Without taking the delivery guarantee(s) into account, these messages would be dropped; with a delivery guarantee such as *At Least Once*, the message will be resent and re-consumed over and over. In this

[6] https://github.com/linki/chaoskube

scenario, some additional error handling is required; resilience measurements are an important concern. This is crucial when introducing retries to handle failures. Two already applied mechanisms in the CloudEvents Router for that is a Dead Letter Channel[11] and a health-check endpoint to integrate a Watchdog[9].

MQTT Reliability Underspecified and Not Configurable. Our most significant and somewhat surprising lesson learned was that MQTT behaves differently from most other protocols (see Section 4.3) 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 whether a message should be acknowledged before or after the application has processed it.

The specification of the protocol only contains a non-normative example in which the messages are acknowledged before they are processed[1]. As a result, many MQTT libraries acknowledge the messages before they are processed. Even worse, they 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 CloudEvents Router uses the Eclipse Mosquitto library to communicate over MQTT. Unfortunately, this library also acknowledges messages before they are processed successfully. We had to fix this issue with a small code change which is currently an open pull request on the Eclipse Mosquitto repository.[7]

6 Related Work

Bernstein and Newcomer describe how client-server communication over queues can be processed with *Exactly Once* semantics in an abstract way[4]. Their examples describe 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 specified. Later, interactions of queues with other transactional systems that are irreversible are analyzed, and transactional processing is discussed.

Gruener, Koziolek, and Rückert measured resilience of different MQTT brokers under unreliable network conditions and show how a message can get lost[10]. Tai, Mikalsen, Rouvellou, and Sutton describe a way of adding conditions not to the message channel but to the message itself[21]. Application-specific conditions are then handled by the messaging middleware.

Steen and Tanenbaum give a broad overview of many distributed system concepts. For instance, they present concepts for error recovery and fault-tolerance[20]. Lampson, Lynch and, Søgaard-Andersen show proof for two *At Most Once* Message Delivery Protocols with additional order guarantee[16].

In "Exactly Once Delivery and Transactional Messaging in Kafka", the design of Kafka's *Exactly Once* strategy is described[8]. While this work is not protocol-independent it presents relevant general concepts. In the context of

[7] https://github.com/eclipse/mosquitto/pull/1932

Simple Object Access Protocol (SOAP) based Web services, there were multiple attempts to standardize reliability with the standards WS-ReliableMessaging and WS-Reliability[7,15].

7 Summary and Outlook

In this experience report, we presented the analysis, design, implementation, and validation of a versatile message router that natively supports events structured according to the CloudEvents specification. Our CloudEvents Router introduces standardized event routing targeting the software landscape of the AgeTech startup CARU; this landscape reaches from an embedded microcontroller on the edge to public cloud services from AWS. The introduction of CloudEvents in all systems helps CARU to safely evolve the structure of their events and reduces friction when developers have to work on different components of their system landscape.

In the CloudEvents Router architecture, special attention was paid to modularity, scalability, and reliability. To support flexible deployments while ensuring a minimal footprint, the Microkernel pattern was chosen as architectural foundation. This design makes it possible to select deployment-specific plugins for configuration access and messaging protocol integration to make the executable as small as possible. To ensure scalability and minimize complexity, we decided to leverage existing messaging infrastructure and to keep the CloudEvents Router stateless by tasking services like Amazon SQS, AWS IoT Core, and Eclipse Mosquitto to store and transport messages reliably. As many events produced by the CARU Device are safety-relevant, providing a delivery guarantee for events entrusted to the router was particularly important. Because of the desired statefulness and the limited protocol interoperability of *Exactly Once*, implementing *At Least Once* was the only feasible option.

When analysing the reliability properties of the messaging protocols in the CloudEvents specification and the implementation of *At Least Once*, we discovered some unexpected properties. MQTT in particular provides rather limited guarantees for *At Least Once* and *Exactly Once*.

The presented CloudEvents Router design and its open source implementation provide a solid foundation for realizing the vision of a unified event plane. To further advance this vision, we have identified three development topics for our future work: (1) Implement the new CloudSubscriptions Discovery API and CloudEvents Subscription API specifications. (2) Integrate the CloudEvents Router into the monitoring solutions from the CNCF. (3) Adopt the asynchronous Application Programming Interface (API) of Rust to lower the resource requirements even further.

References

1. Banks, A., Briggs, E., Borgendale, K., Gupta, R.: MQTT version 5.0 (2019), https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

2. Basig, L., Lazzaretti, F.: CloudEvents Router. HSR (2020), https://eprints.ost.ch/id/eprint/832/
3. Basig, L., Lazzaretti, F.: Reliable messaging using the CloudEvents Router (2021), https://eprints.ost.ch/id/eprint/904/
4. Bernstein, P.A., Newcomer, E.: Queued transaction processing. In: Principles of Transaction Processing, pp. 99–119. Elsevier (2009)
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture. Wiley (1996)
6. CloudEvents specification v1.0. GitHub (2019), https://github.com/cloudevents/spec/tree/v1.0
7. Davis, D., Karmarkar, A., Pilz, G., Winkler, S., Yalcinalp, U.: Web services reliable messaging (WS-ReliableMessaging) (2009)
8. Exactly once delivery and transactional messaging in Kafka (2017), https://docs.google.com/document/d/11Jqy_GjUGtdXJK94XGsEIK7CP1SnQGdp2eF0wSw9ra8
9. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns. Springer Publishing Company, Incorporated (2014)
10. Gruener, S., Koziolek, H., Rückert, J.: Towards resilient IoT messaging: An experience report analyzing MQTT brokers. IEEE International Conference on Software Architecture (ICSA) (2021)
11. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison Wesley (2004)
12. Hwang, S.Y., Wang, H., Tang, J., Srivastava, J.: A probabilistic approach to modeling and estimating the QoS of web-services-based workflows. Information Sciences **177**(23), 5484–5503 (2007). https://doi.org/10.1016/j.ins.2007.07.011
13. ISO/IEC 13236:1998(E): Information technology – quality of service: Framework
14. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models (2011), https://www.iso.org/standard/35733.html
15. Iwasa, K., Durand, J., Rutt, T., Peel, M., Kunisetty, S., Bunting, D.: Web Services Reliable Messaging TC WS-Reliability 1.1 (2004)
16. Lampson, B.W., Lynch, N.A., Søgaard-Andersen, J.F.: Correctness of at-most-once message delivery protocols. In: Proceedings of FORTE 1993, Boston, MA, USA (1993)
17. Ming, Z., Yan, M.: A modeling and computational method for QoS in IOT. In: 2012 IEEE International Conference on Computer Science and Automation Engineering. IEEE (2012). https://doi.org/10.1109/icsess.2012.6269459
18. Narkhede, N.: Exactly-once semantics are possible: Here's how Kafka does it (2017), https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/
19. Petrova-Antonova, D., Ilieva, S.: Towards a unifying view of QoS-enhanced web service description and discovery approaches. EPTCS (2009). https://doi.org/10.4204/EPTCS.2.8
20. van Steen, M., Tanenbaum, A.: Distributed systems. Maarten van Steen (2017)
21. Tai, S., Mikalsen, T., Rouvellou, I., Sutton, S.: Conditional messaging: extending reliable messaging with application conditions. In: Proceedings 22nd International Conference on Distributed Computing Systems. IEEE Computer Society (2002). https://doi.org/10.1109/icdcs.2002.1022249
22. Treat, T.: You cannot have exactly-once delivery (2015), https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/
23. Distributed transaction processing: The XA specification (1991), http://www.opengroup.org/onlinepubs/009680699/toc.pdf

Cite as Basig L., Lazzaretti F., Aebersold R., Zimmermann O. (2021) Reliable Event Routing in the Cloud and on the Edge. In: Biffl S., Navarro E., Löwe W., Sirjani M., Mirandola R., Weyns D. (eds) Software Architecture. ECSA 2021. Lecture Notes in Computer Science, vol 12857. Springer, Cham.
https://doi.org/10.1007/978-3-030-86044-8_17