Summer Student Program

CERN

Start Date: 05/06/2023

Student Name:

Daniel Kosc

Project Title:

Evaluation of message-oriented middleware in the context of a future run control system for CMS

CERN Supervisors:

Philipp Brummer
Hannes Sakulin

Abstract:

This report offers a comparison between Message-Oriented Middleware (MOM) and Representational State Transfer (REST) as applied to the run control system of the CMS experiment. This assessment is grounded in a configurable prototype, which has been designed to emulate node structures within the run control system, particularly in the form of a hierarchical tree. The insights and empirical findings acquired during the inception, development, and measured results of this prototype can serve as a valuable knowledge base for informing future deliberations regarding the appropriateness and efficacy of choosing between REST and MOM as the preferred communication architecture for the CMS run control system.

Contents

1 Motivation

There is a need to adapt Run Control System (RCMS) [1] for the CMS 2 upgrade. The main objectives are to simplify maintainability and further development for control of applications in the Kubernetes cluster. Moreover, it is noteworthy that the communication paradigm between the RCMS (depicted in green) and XDAQ (represented in blue) is transitioning from SOAP to REST, as illustrated in Figure 1. That is one option for RCMS, but we should also consider Message Oriented Middleware (MOM) mainly because communication patterns in RCMS are mostly asynchronous. Therefore, the project is focused on the implementation of a tree node structure, which is similar to the actual RCMS. This tree with configurable attributes will result in a better assessment of what is a more suitable fit for RCMS needs.

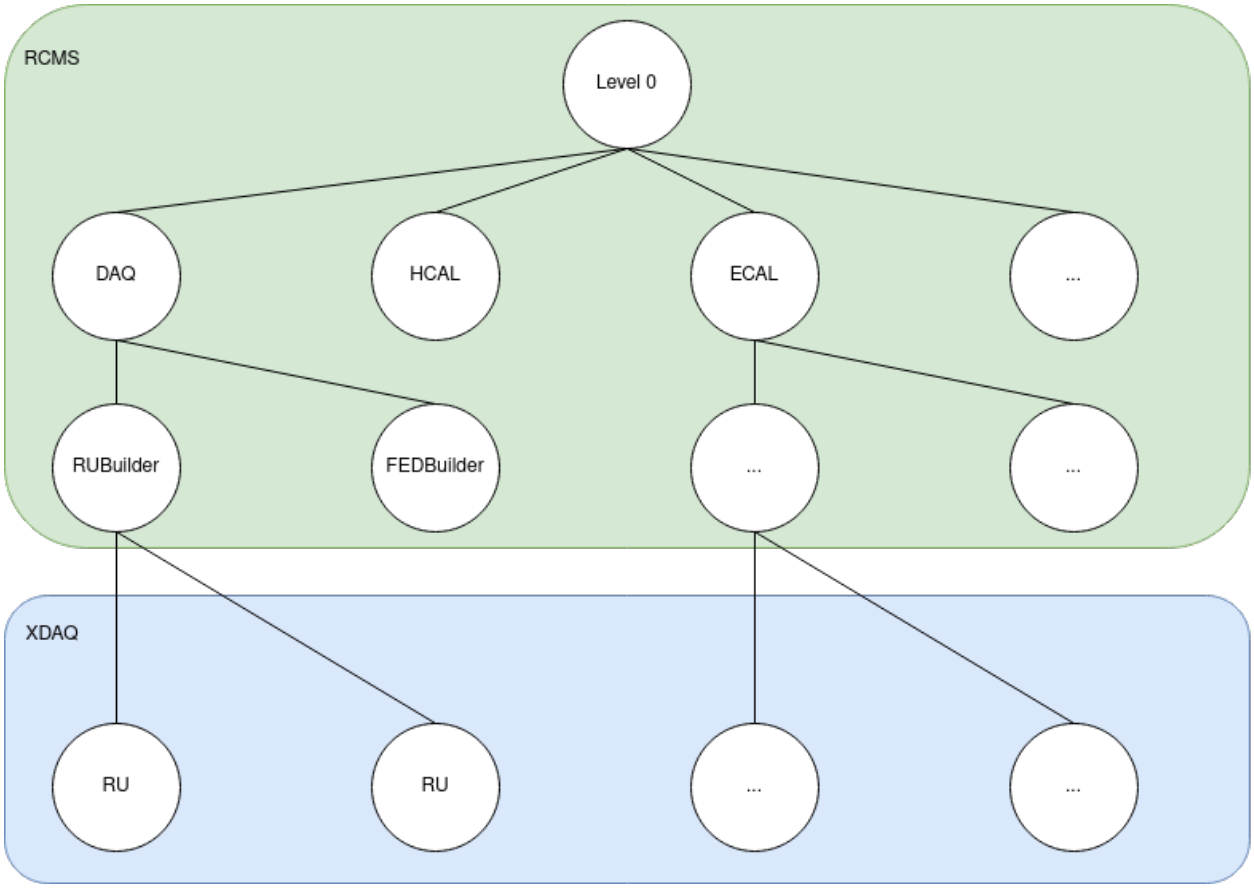

Figure 1: Run control hierarchy

2 Node

Each node is an independent unit with a unique ID that can communicate only to direct neighbours in a tree structure. There exist three types of messages:

- revealing the current state - triggered by an external request
- sending input to the children - triggered by internal logic or external request
- sending notifications to the parent - triggered by internal logic

Each node is always keeping a record of its own internal state. The state can be changed by receiving input (command to change a state) or after receiving a notification from the child node that it has changed the its state.

The node starts its life cycle in the Initialisation state, where it remains until the internal logic is fully prepared for handling communication. Once it is ready, it will change its state to Stopped and notify its parents. It will remain in this state until it receives a input from its parent with an argument specifying the probability for a node to enter its Error state. It will immediately switch to the starting state and propagate this message to all its children after a predefined transition time. Once it receives notification from its children, it will change the state accordingly to Running or Error. In case it is a leave node, it changes its state to Error with a probability defined in the argument, otherwise it enters a Running state. In the Running state it runs in the loop with a predefined period where it can transition to Error state with the probability defined in the argument of input command, or it can receive a change state input command and immediately change its state to the Stopped and propagate this message to its children.

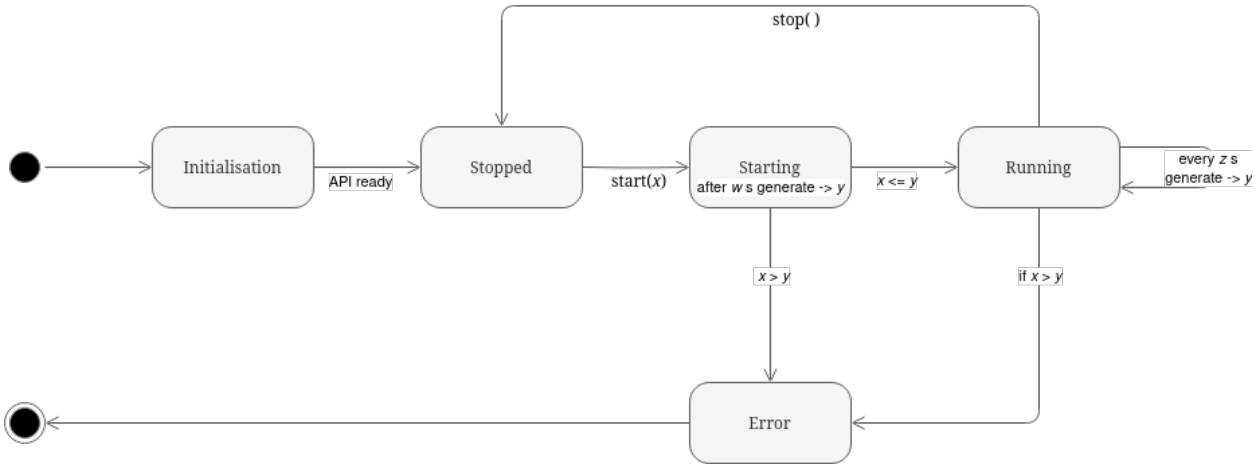

Figure 2: State diagram

3 Implementation description

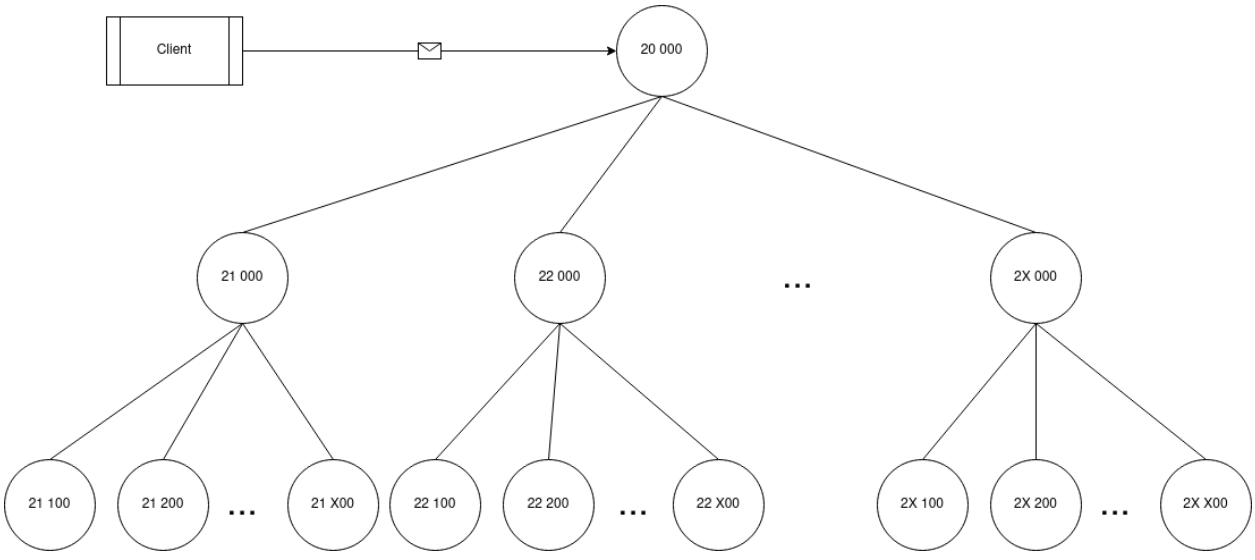

Figure 3: Example of tree node hierarchy using REST with depth 2 and arity X

The tree structure is built based on parameters defined in the configuration file:

- tree depth and node arity
- REST or MOM
- JSON or Protocol Buffers [2]
- message validation on/off
- time durations (timeout, transitions, loop period, processing time)
- endpoints path

3.1 REST

REST implementation is based on the FastAPI [3] framework, which automatically generates an OpenAPI schema. That makes it possible to generate clients for many programming languages. Additionally, the framework's default configuration facilitates the automatic generation of web-based API documentation, accompanied by an integrated REST client, as illustrated in Figure 4.

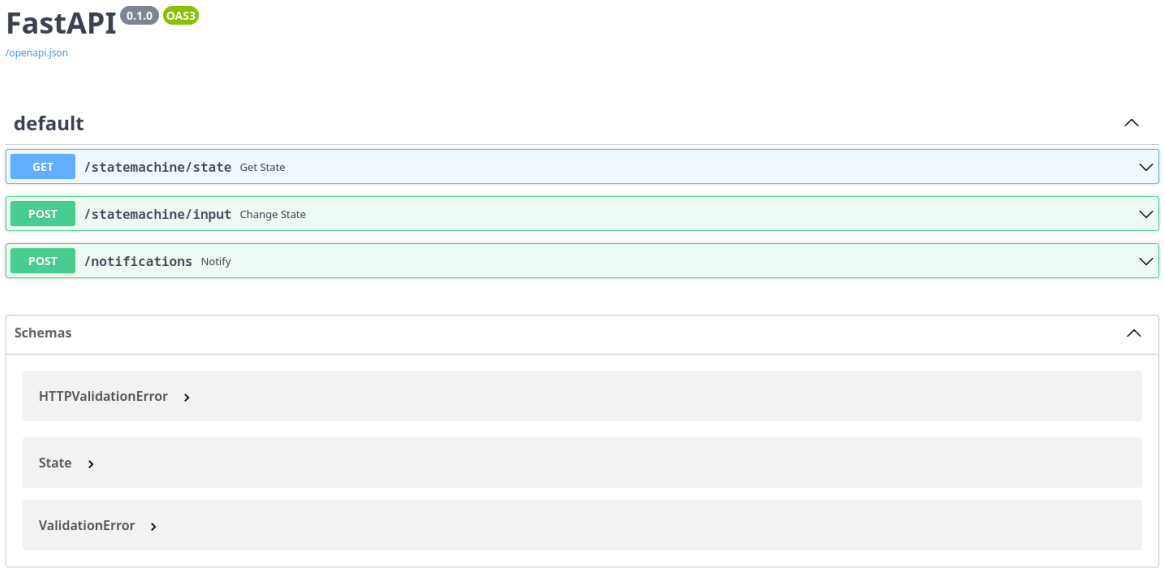

Figure 4: web-based API documentation with a built-in REST client

3.2 MOM

RabbitMQ [4] was selected as the representative of Message Oriented Middleware for this project because it is the "most widely deployed open-source message broker" [4]. It offers several different delivery strategies, with the best fitting strategies for the RCMS use-case being Topic and RPC Server. In general, a big advantage of Message Oriented Middleware is the fact that both sides (sender and receiver) communicate with the broker, leading to the decoupling of communicating nodes.

In the present context, the entity responsible for transmitting a message is referred to as the "Producer" node, while the entity tasked with receiving the message is denoted as the "Consumer" node. The whole process uses acknowledgment on top of TCP to ensure that message was correctly consumed. By default, acknowledgment exchanges are confined to interactions solely between the "Broker" and the "Consumer".

The "Consumer" is capable of sending one of: ACK or NACK (resend again) or REJECT (delete it from the queue)

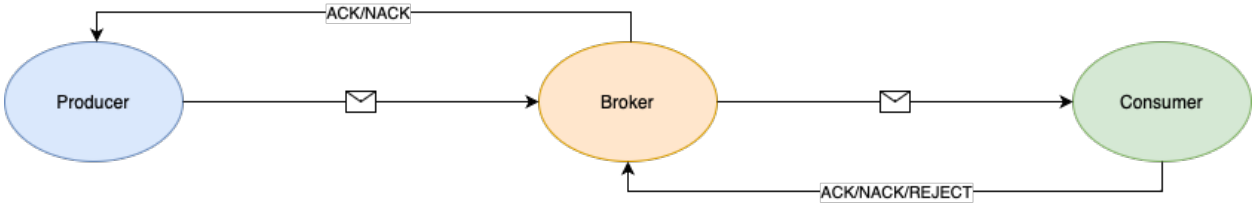

Figure 5: Run control hierarchy

3.2.1 Topic messaging pattern

This technology is employed to transmit notifications or input (change state commands). It is important to underscore that this mechanism does not return a response indicating the successful processing of the transmitted message. Additionally, it is noteworthy that the utilization of topics permits the the use of wildcards for matching multiple topic names, facilitating the delivery of a single message to a group of nodes (not implemented in the prototype developed during the program).

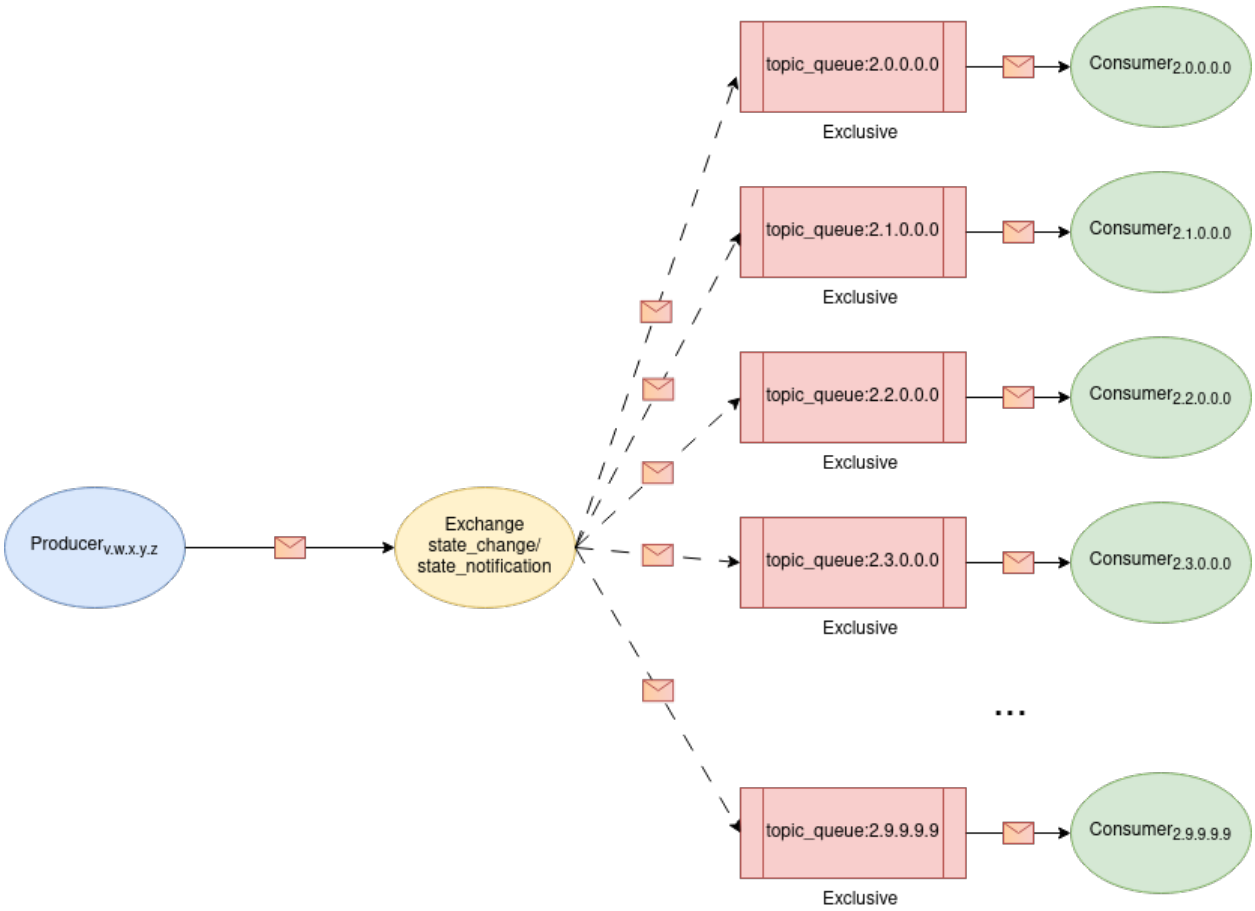

Figure 6: Topic delivering strategy

3.2.2 RPC Server messaging pattern

This strategy enables the establishment of bidirectional message flow, facilitating the receipt of responses to dispatched messages. The Client (also referred to as the Producer) is required to augment its message transmission with supplementary information, specifically the designation of a queue, wherein the response shall be directed. This is used for receiving the current state of the requested node.

A client sends the white envelope with the request to get the state to the exchange, which will forward it to the right queue based on the routing number. When node (RPC Server) receives it, it waits for a predefined processing time and then responds with the blue envelope to the queue defined in the white envelope (see figure 7).

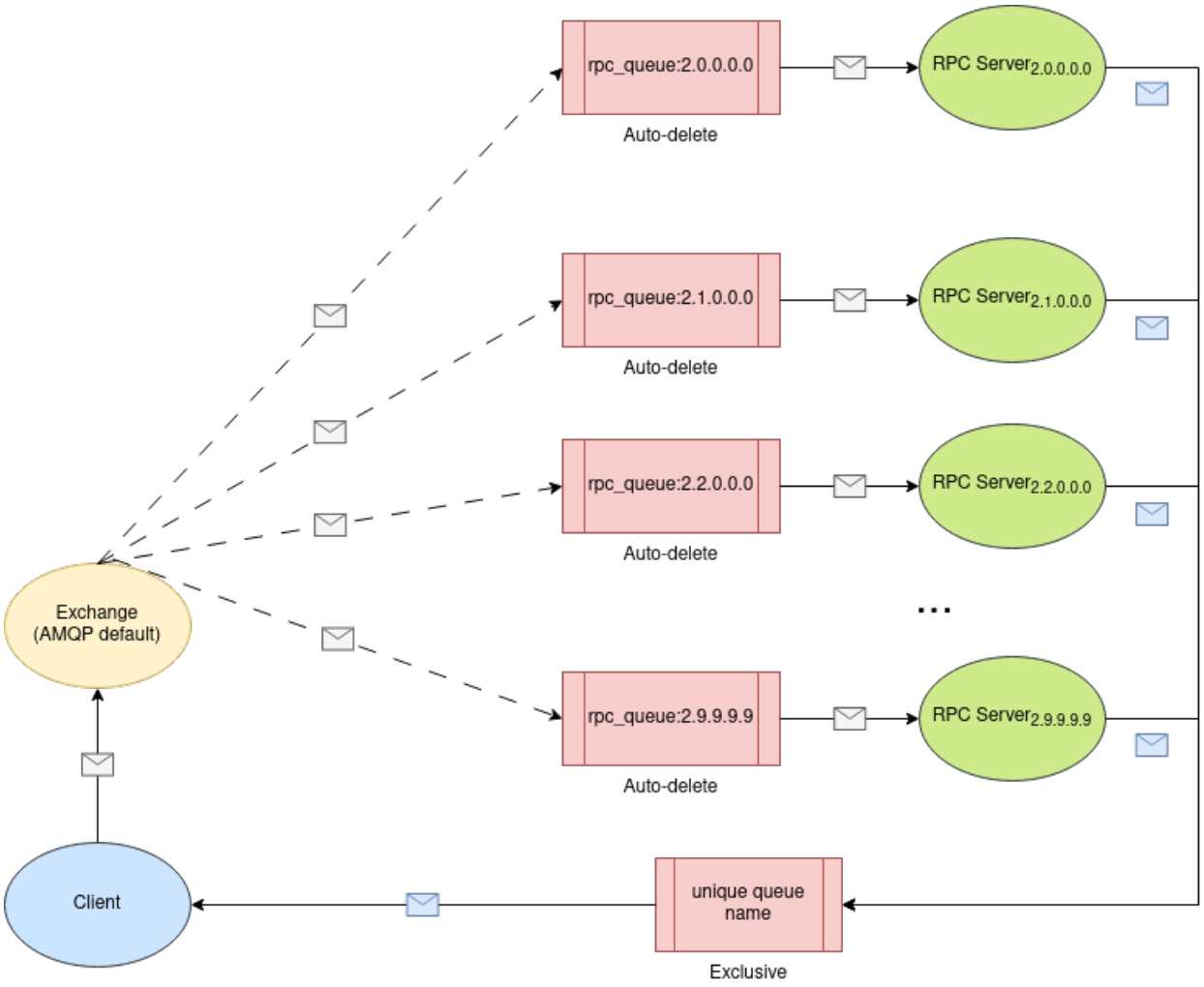

Figure 7: RPC Server delivering strategy

3.3 Validation

Ensuring the robust processing of accurate information is crucial in the control system; therefore, implementation also contains a validation that can be disabled to enhance system performance. Pydantic [5] library was used to validate JSON messages sent via REST. Even though there exist 3rd party libraries for validation of Protocol Buffers, e.g. protobuf-to-pydantic [6], it supports only proto3. Hence, a custom validator for RabbitMQ Protocol Buffers messages was implemented. In essence, both approaches offer identical functionality - check message and raise an error in case data are invalid:

- message structure
- type
- value range

The raised error is then caught by an exception handler. There is one already implemented in FastAPI, so it's convenient to make use of it. In the case of RabbitMQ, there was an implemented custom exception handler. Considering error handling, it's important to point out that REST always has some return value (response code), even when using asynchronous calls, which gives some additional information compared to MOM by default. Nevertheless, it is possible to implement all messages as an RPC Server (see 3.2.2) instead of Topic and thus enabling the receiving a feedback, but this brings more complexity.

3.4 Message format

There was implemented an option for MOM to choose whether messages will be transmitted in JSON or Protocol Buffers. Protocol Buffers, being a schema-based binary data format, possess advantages such as enhanced speed and reduced data size (see Figure 8). On the other hand, data are not human readable, and it requires additional developmental step - compilation of schema definition. However it result in quicker transmission primarily due to the faster conversion of Python objects to Protocol Buffers.

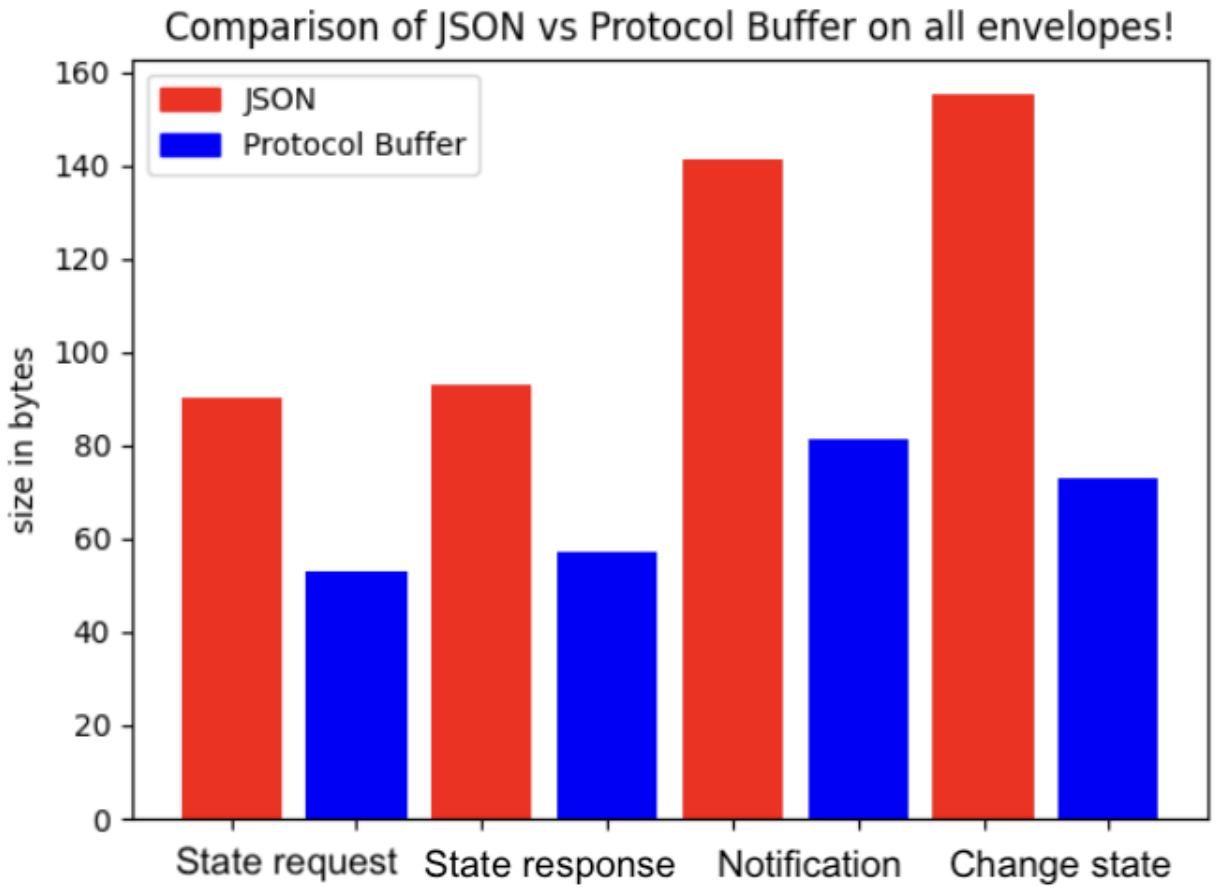

Figure 8: Comparison JSON vs. Protocol Buffers

4 RabbitMQ deployment

Multiple deployment methodologies exist for RabbitMQ, ranging from conventional installation on a physical host to containerization through Docker images or orchestration within a Kubernetes Cluster, as illustrated in Figure 9. In the context of the present use case, the Kubernetes Cluster emerges as the preferred approach, and so, we dive deeper in the available alternatives in following subsections.

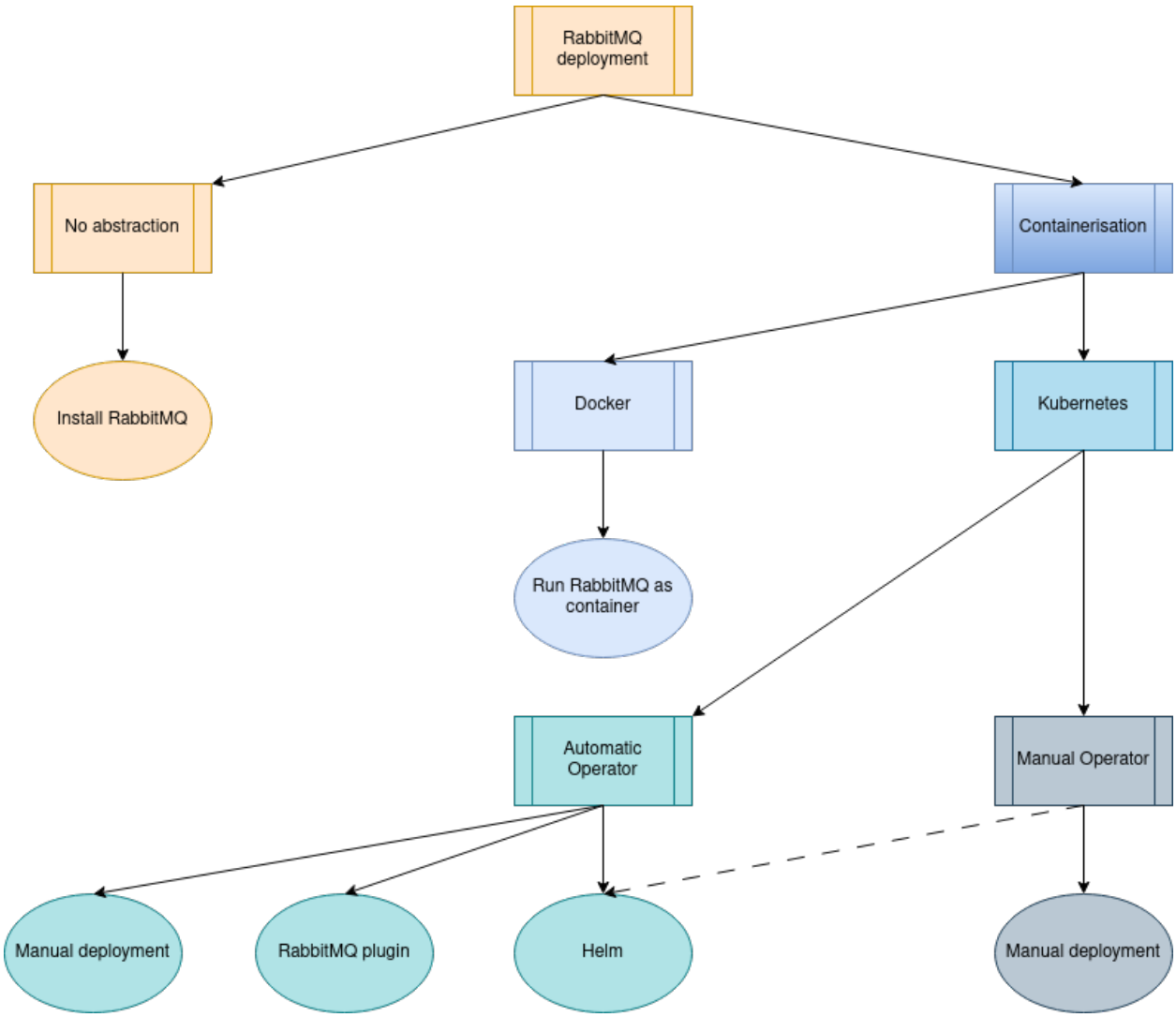

Figure 9: RabbitMQ deployment options

4.1 With Operator

Leveraging Kubernetes Operators for RabbitMQ represents a significant advancement in streamlining the deployment, scaling, and management of RabbitMQ clusters within cloud-native environments. These specialized controllers simplify the integration of RabbitMQ into containerized infrastructures, ensuring consistent and reliable deployments while enabling automatic scaling and high availability. Kubernetes Operators offer a powerful toolset for enhancing RabbitMQ's operational efficiency and resilience in the dynamic landscape of modern cloud-native computing.

4.1.1 Cluster Operator

The RabbitMQ Operator represents an orchestration instrument crafted for Kubernetes ecosystems. Its primary objective is to enhance operational efficiency and expedite the deployment and administration of RabbitMQ clusters within this framework. The operator exhibits a remarkable degree of adaptability concerning deployment techniques, permitting users to configure it either through a dedicated standalone configuration file, utilising the RabbitMQ plugin, or by leveraging the Helm charts functionality.

4.1.2 Messaging Topology Operator

The Messaging Topology Operator is a component responsible for defining and managing the structure and relationships of queues and exchanges within a messaging architecture. However, its utility can vary depending on the specific messaging system's characteristics and requirements. In RCMS topology, which is not expected to change very often, the Messaging Topology Operator may not prove as useful as in more dynamic environments. RabbitMQ's design principles prioritize a static topology, where queues and exchanges are typically configured during the setup phase and are not expected to change frequently. This deliberate choice of stability minimizes potential disruptions and enhances the system's robustness. Consequently, the need for a dynamic Messaging Topology Operator in RabbitMQ is diminished, as it may introduce unnecessary complexity and management overhead to an architecture that values consistency and predictability in its messaging patterns. Therefore, while the Messaging Topology Operator is a valuable tool in many messaging contexts, its applicability in RCMS architecture, which is not expected to change very often, is relatively limited.

4.2 Without Operator

Maintaining RabbitMQ pods in a Kubernetes environment without a provided operator by RabbitMQ requires careful orchestration. In the absence of an operator, the process involves setting up a custom deployment or statefulset for your application that consumes RabbitMQ messages. You'll need to ensure proper handling of error scenarios, and scalability.

4.2.1 My deployment

In order to enhance my proficiency with Kubernetes, I made the deliberate choice to instantiate RabbitMQ without Operator, opting instead for the utilization of the StatefulSet component as displayed in Figure 10. This decision aligns with the nature of RabbitMQ as a stateful application. The deployment encompasses a cluster of four pods, each instantiated with the RabbitMQ image, thereby facilitating the provisioning of services on internal ports. Subsequently, a LoadBalancer component was deployed to facilitate access to the internal ports associated with these pods.

It is important to note that the principal functionality designed to distribute tasks across multiple nodes remains underutilized in this context. This underutilization arises from the inherent constraint of having only a single Node operating within my virtual cluster, which has been provisioned using Kubernetes in Docker (KinD) [7]. For the purposes of load balancing in this specific example, the MetalLB [8] implementation was chosen and employed as the load balancing mechanism.

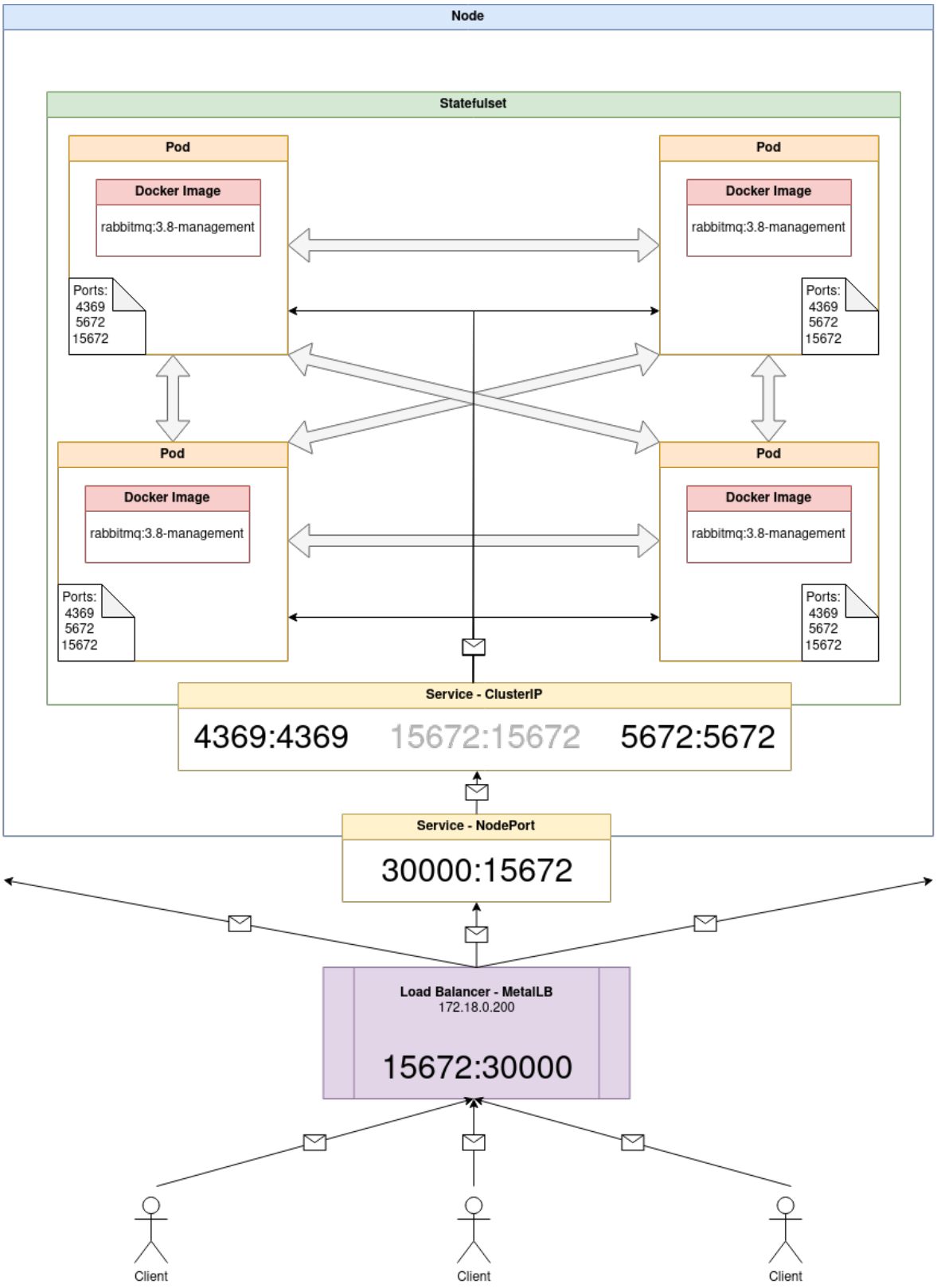

Figure 10: Deployed Kubernetes architecture

5 Comparison

From a developmental perspective, a pivotal distinction arises in the implementation complexity. The successful establishment of the MOM and the effective exploitation of its inherent benefits necessitate a high degree of expertise. This expertise encompasses the complex configuration of various facets, such as message exchanges, inquiry mechanisms, message delivery strategies, logging mechanisms, the deployment of the messaging broker itself, and an array of other parameters.

In strong contrast, Representational State Transfer (REST) does not impose such intricate configuration demands. However, it fosters a higher degree of coupling among nodes within the system. This elevated coupling can become obstacle when considering the scalability of the system or the prospect of making future alterations to the tree-like structure. Notably, in MOM systems, nodes exclusively communicate with the central messaging broker and are freed from any other information about the message recipients, save for their unique identifiers called routing keys.

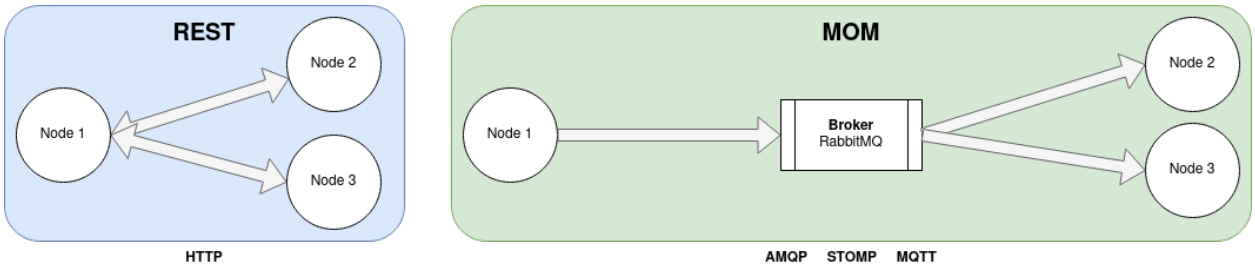

Figure 11: REST compared to MOM

The primary drawback associated with the Message-Oriented Middleware (MOM) in the context of this specific utilization pertains to the absence of an acknowledgment mechanism between the message publisher and consumer. Consequently, the management of errors in this framework introduces a set of challenges that are absent in the Representational State Transfer (REST) architecture, wherein some response is invariably generated. Obviously information value of that response depends on REST implementation. One potential approach to mitigate this MOM limitation is the incorporation of a Remote Procedure Call (RPC) message strategy for all messages, thereby enabling the reception of response codes. However, it is worth noting that this alternative introduces an added layer of complexity to the system.

6 Measurement

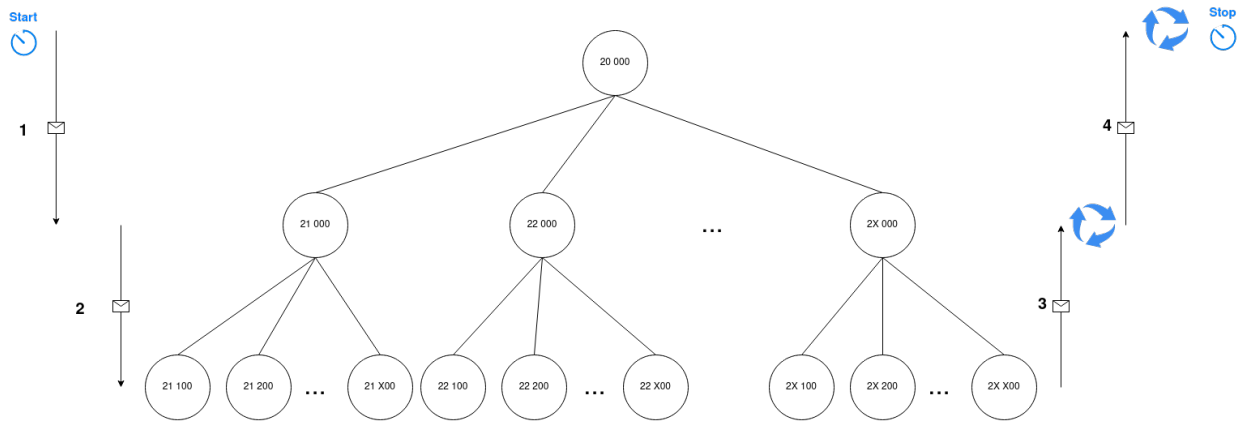

Figure 12: Measurement schema

The time duration required for the propagation of a communication within a tree structure (see Figure 12) depends on numerous influencing factors. Among these factors, the selection of the underlying technology, whether it be Representational State Transfer (REST) or Message-Oriented Middleware (MOM). The measurement of this time starts with the initiation of a transmission event from the root node to its child nodes and ends upon the reception of confirmatory notification from its last child node, indicating the change of their respective states in accordance with a specified input.

In cases where the child nodes of the root are not leaf nodes, they forward received input further down. They first transmit the input to their own child nodes, subsequently awaiting notifications signifying the completion of state changes from their child nodes. When the notification arrives, the node can change its own state and send a notification to the parent.

Measurement was executed on:

- 32 GB DDR4-3200
- i7-1165G7 4 Cores 8 Threads 4.70 GHz
- RabbitMQ running as a Docker container

All nodes are running on a single computer → no real network communication delays included

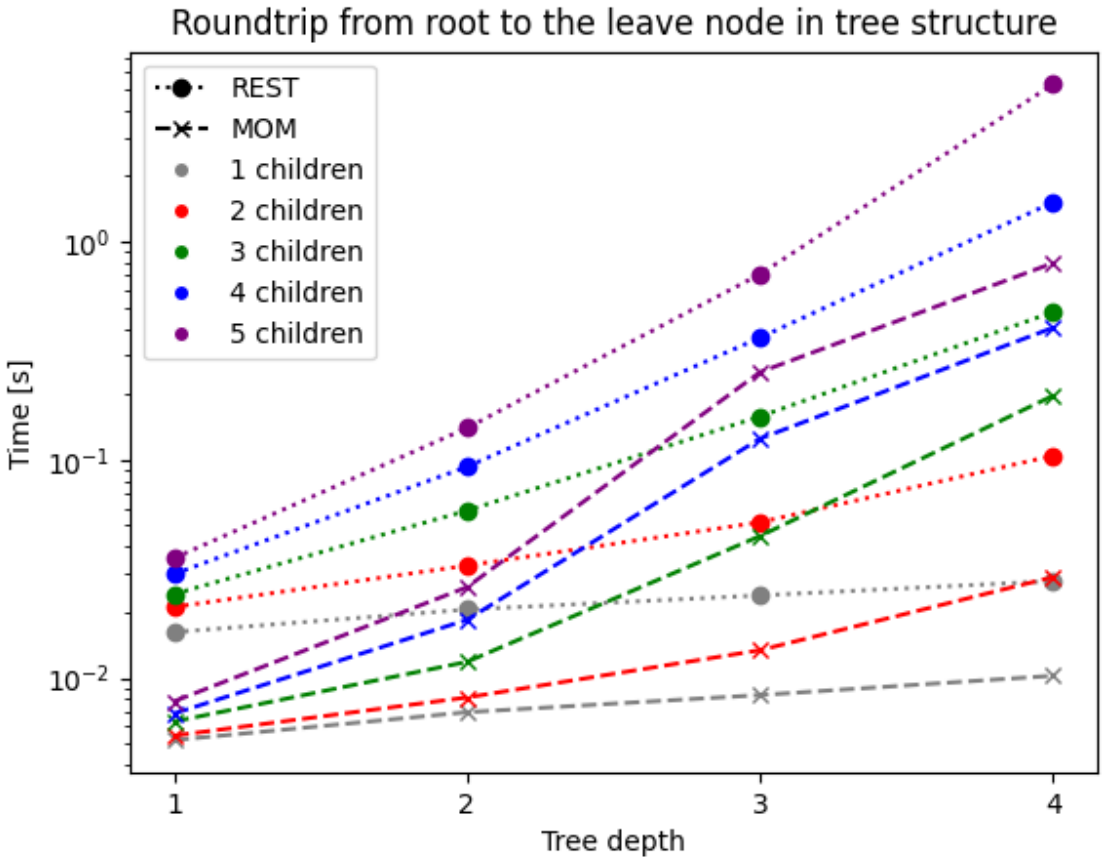

Figure 13: Measurement results

7 Conclusion

MOM brings many advantages that are not fully utilised in the RCMS use case:

- MOM broker can run on several instances in Kubernetes - High availability
- MOM can handle message delivery to nodes that are rebooting without any action from a producer/sender

There is also a benefit which might be useful, but it was not included in this project:

- Using wildcards for sending messages - sending the same message to multiple consumers

In general, it's safe to say that the MOM implementation cost is noticeably higher compared to REST and it needs to be weighed against the benefits for the specific use-case. Advantage of decoupled nodes might be strong enough to choose MOM in RCMS use-case.

8 Future Work

There exists a notable scope for further investigation within the domain of Kubernetes deployment and performance assessment. In the initial phase of further study, it is essential to deploy a node structure within the Kubernetes ecosystem. It is worth noting that the internal operational logic of these nodes shall remain invariant, independent of being executed within a clustered environment. Subsequently, it is essential to conduct a repeated measurement analysis under conditions where nodes operate within clusters, preferably leveraging an infrastructure that closely mimics the actual use-case scenario.

References

[1] P. Brummer, *Evaluation of Technologies for a Future Run Control System for the Compact Muon Solenoid Experiment at CERN*, 2018. Presented 2018. URL: https://cds.cern.ch/record/2727984.

[2] URL: https://protobuf.dev/.

[3] URL: https://fastapi.tiangolo.com.

[4] URL: https://www.rabbitmq.com.

[5] URL: https://docs.pydantic.dev/latest/.

[6] URL: https://pypi.org/project/protobuf-to-pydantic/.

[7] URL: https://kind.sigs.k8s.io.

[8] URL: https://metallb.universe.tf.