Achieving reliable communication between Kafka and ROS through bridge codes

Luan Lucas Lourenço[1], George Oliveira[2], Patricia Della Méa Plentz[2] and Juha Röning[3]

***Abstract*— The number of smart warehouses has increased in the last few years, mainly because of the enormous sales volume in e-commerce systems. Such physical environments are fully equipped with robots that solve the orders that arrive there. Most research in the literature presents results just about the path planning and allocation tasks as the main problems related to this area. However, it is also essential to consider how robotic platforms, such as ROS (Robot Operating System), deal with integrating with streaming platforms that manage the orders that arrive into the smart warehouses. This paper presents a simple way to integrate ROS with Kafka, a streaming platform highly used in e-commerce systems. We develop a bridge code to glue these two platforms and test it within three realist simulation scenarios. As we will show, the use of the QoS profiles provided by the ROS Data Distribution Service (DDS) allows us to achieve a higher level of reliability.**

I. Introduction

Robot Operating System (ROS) has been widely used for robotics applications [1] and was created as part of STAIR project [2] at Stanford University and the Personal Robots Program [3] at Willow Garage. As explained by [4] it was designed to act as an open-source middleware providing hardware abstraction, low-level device control, interprocesses message-passing and package management. As an abstraction it frequently makes developers' lives easier as they don't have to deal with hardware drivers and interfaces and can focus on the development. [1] summarizes ROS goals as:

- Peer-to-peer: components run independently of each other, possibly on different hosts. In order to achieve this degree of freedom it becomes necessary to have a lookup mechanism in-place to allow different processes to find each other at runtime;
- Multi-lingual: people have different preferences and needs over different programming languages and with that in mind ROS is designed to be language-neutral by using Interface Definition Language (IDL) in order to describe messages sent between modules to make cross-language development possible;
- Tools-based: instead of building a monolithic system, components are built around small tools that later are combined to perform tasks;
- Thin: algorithms become portable as modular builds are performed and therefore making it possible to have libraries that have no dependencies on ROS (standalone libraries) placing all its complexities within the library itself. Then it's just a matter of creating a small executable that can be read by ROS and expose the library's functionalities;
- Free and Open-Source: source code is publicly available under BSD license allowing both non-commercial and commercial projects to benefit from ROS.

As pointed out by [5], the team behind ROS initially had a much narrower scope however today it is possible to see that it is being used in a much broader context that they anticipated, going beyond academic research and thus having also commercial systems built on top of it. With new demands arriving such as real-time systems adding support for inter-process and inter-machine communication, running in not reliable networks allowing it to behave as well as possible when connectivity degrades, and aiming the future growth of the platform, a new version of the platform has been created and it is known as ROS 2.

From this point forward, every time ROS is mentioned, the authors are referencing ROS 2.

Nowadays, ROS should be integrated with streaming platforms, like Kafka [6], for example. This integration is necessary because robots entirely operate the smart warehouses, and the e-commerce enterprises use streaming platforms to manage the orders that arrive in the system, which as shown by [7] achieved a peak of 583,000 orders per second in the last quarter of December and [8] processed an average of 121.1 million orders per month, both in 2020.

Considering this important research topic, the paper presented here seeks to provide the simplest way to integrate Kafka and ROS. We develop a bridge code to glue these two platforms and test it within three realist simulation scenarios. As we will show, the use of the Quality of Service (QoS) profiles provided by the ROS Data Distribution Service (DDS) allows us to achieve a higher level of reliability.

This work is organized as follows: Section II presents the related works. Section III describes ROS 2 Design, and Section IV details the Data Distribution Service (DDS) which is the communication protocol employed by ROS 2. The software architecture used in this paper is shown in Section V. Section VI shows the problem formulation and the proposed solution. The experimental results is presented at Section VII and Section VIII concludes our work.

[1]Luan Lucas Lourenço is with Robotizando UFSC, Joinville, Brazil luaan.lucas@gmail.com

[2]George Oliveira is with the Graduate Program in Computer Science, Federal University of Santa Catarina, Florianópolis, Brazil geo.soliveira@gmail.com

[2]Patricia Della Méa Plentz is with the Graduate Program in Computer Science, Federal University of Santa Catarina, Florianópolis, Brazil patricia.plentz@ufsc.br

[3]Juha Röning is with the Biomimetics and Intelligent Systems Group - BISG, University of Oulu, Oulu, Finland juha.roning@oulu.fi

II. Related Works

Streaming platforms like Kafka are highly used in e-commerce systems. Most of them should connect with ROS-like system to apply some Multi-Robot Task Allocation (MRTA) technique. Bridge codes serve for integrating these kind of distributed systems, reducing the effort and time spent on this integration. Few published works targets the aspects related to the integration problem.

A robot software bridge for interconnecting Open Platform for Robotic Services (OPRoS) [9] with ROS without modification of these source codes is proposed in [10]. A remote robot control application was used to evaluate the proposed bridge implementation, which was developed in C++ language. A test was used to observe the bridge code's feasibility however there are no comments about difficulties related to this proposed solution.

An industrial use-case of a collaborative and intelligent automation system with a ROS 2 based communication architecture is described [11]. ROS and ROS 2 communication layer is explained, but the bridge code which connects those systems with streaming data systems, like Kafka, was not presented. These kinds of streaming systems should deal with many external data; integrating these systems with software robotic platforms is not trivial. That is why it is essential to clarify how to connect them.

In [12], ROS and Kafka are applied to a fog computing server model that generates a high definition (HD) map node of the ROS environment. This architecture encompasses edge, fog, and cloud technologies for IoT applications but the authors do not describe the code developed to connect these two important software platforms.

In [13], authors propose a distributed cloud architecture for robotics computation offloading and ROS is used as the framework to provide communication across different robots whilst Kafka is used to distribute messages to the cloud environment and vice-versa. There is no direct message exchange between ROS and Kafka, therefore being necessary the use of another dependency called rosbridge server, which uses WebSockets to enable non-ROS clients to publish and subscribe to ROS topics.

Considering this set of papers, is it possible to observe the lack of explanation of how to connect ROS with streaming platforms. The study of this kind of software component to improve integration between these computational systems is often a topic of high relevance in the scientific community. Considering this interest, we propose a software component that interconnects ROS and a streaming platform Kafka-like.

III. ROS 2 Design

ROS applications consist of independent computing processes called nodes, which promote fault isolation, faster development, modularity, and code reusability [14]. Nodes communication is mainly based on a publish/subscribe model therefore they communicate by sending messages through a topic with a message being a strictly typed data structure composed by primitive types (char, integer, floating point and boolean), arrays of primitive types, constants, other messages and arrays of other messages [1][15]. Nodes interested in those messages will subscribe to the respective topic in order to receive the data. One single topic might have multiple publishers and subscribers, and it's also possible for a single node to publish and subscribe to multiple topics. Fig. 1 [16] illustrates a typical system built on top of ROS where nodes (ellipsis) communicate with each other through topics (rectangles). For example, *robot_state_publisher* is a node that receives messages from topic *joint_states* and publishes messages following the strictly typed data structure defined by *tf* topic.

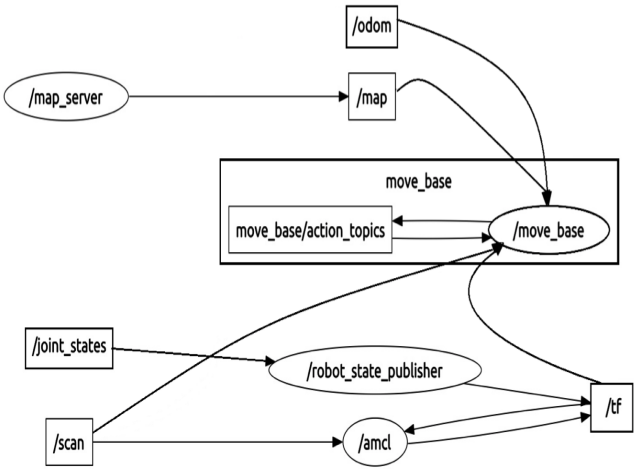

Fig. 1: Typical ROS System

Although nodes communicate mainly asynchronously by publishing/listening to messages, a synchronous form of communication is also possible, which is called service. [1][15] state that a service is defined by a string name, and the package it is in, and a pair of strictly typed messages, being one for the request and one for the response (resembling web services). It is important to highlight that like topics, services must be uniquely named (there can't be more than one service with the same name).

Nodes can be written in different programming languages requiring each of them to have its own client library that provides a language-specific Application Programming Interfaces (APIs). These libraries are in turn built on top of ROS Client Libraries (RCL), a language agnostic model to provide consistency. Officially C++ and Python are supported, with community-provided support for other languages therefore, as explained by [16].

In Fig. 2 [16] it is possible to visualize that ROS is composed of multiple layers where each one is responsible to provide APIs to access underlying implementations.

IV. Data Distribution Service (DDS)

For distributed communication, ROS is built on top of DDS and as explained by [14] DDS meets the requirements of distributed systems for safety, resilience, scalability, fault-tolerance and security, and by reducing library sizes and memory footprints. DDS is a communication protocol

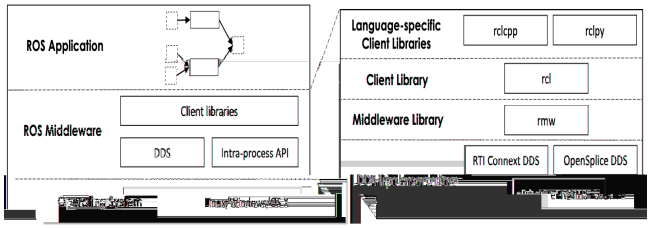

Fig. 2: ROS Layered Structure.

defined by the Object Management Group (OMG) [17] which aims to describe APIs and Communication Semantics that enable communication between data providers and data consumers (publish/subscribe data-distribution system) [14][18]. eProsima Fast DDS is the default implementation used by ROS [18]. As illustrated in Fig. 2, in the DDS Implementations layer, Connext by RTI [19] and Vortex OpenSplice by Adlink [20] are supported as well.

The core of DDS is a Data-Centric Publish-Subscribe (DCPS) model, designed to provide efficient data transport between processes even in distributed heterogeneous platforms, and creates an abstract space called Global Data Space where topics are stored [14][18]. Every entity that is part of the DCPS domain has a QoS Policy attached to itself defining behavioral characteristics. As shown in Fig. 3 [16] DCPS model is defined by the following entities [14][18]:

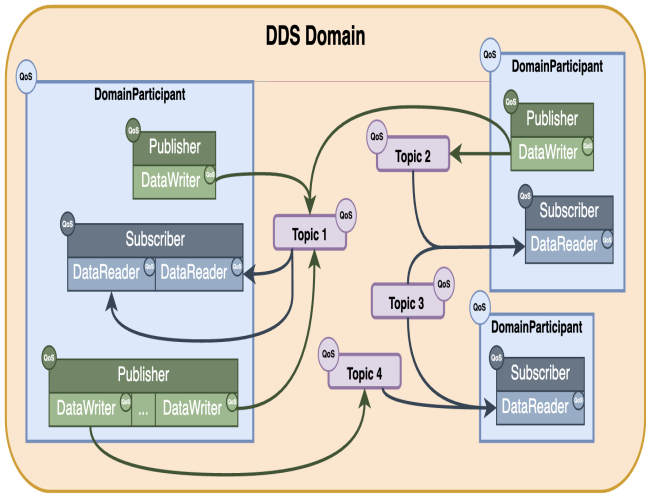

Fig. 3: DCPS Model.

- Domain: it is a concept, identified by an ID, used to group all publishers and subscribers, which exchange data under different topics, whether they belong to one or multiple applications;
- DomainParticipant: individual applications are called DomainParticipants in which its purpose is to act as a container for other entities in the domain. In DDS, these applications can only communicate with each other within the domain;
- Publisher: it is the entity responsible for the creation and configuration of DataWriters. Even though a publisher is allowed to have multiple DataWriters, a DataWriter is assigned to only one topic where it can then publish messages;
- Subscriber: a subscriber is in charge of receiving data published under topics to which it subscribes and hosts one or more DataReaders, which in turn notify the application whenever new data is available;
- Topic: it is an unique entity within the domain responsible to connect publishers and subscribers, or DataWriters to DataReaders, by defining a name and a data type.

V. Software Architecture

Software architecture for mobile robots in smart warehouses should integrate much information from the environment, robots, and orders in the system. Although this paper deals with the bridge code for integration between streaming platforms and open robotic platforms, it is essential to understand the entire system architecture. A way to do this integration is to join task allocation and path planning strategies with data streaming systems to reach better results in increasing the number of fulfilled tasks.

The distributed robotic software architecture used in this paper is situated over the ROS [21]. It comprises six modules: Task Organizer, Cost Estimator, Path Planner, Monitor, Execution Profiles, and Robot Failure Profile. As shown in Fig 4, the primary execution cycle of this architecture begins in the Orders module, where the data streaming system delivers orders lists. The Task Organizer module receives these lists and breaks them into tasks. The Cost Estimator module calculates the cost for each robot-task pair through data from three sources: Task Organizer module, Monitor, and Execution Profiles.

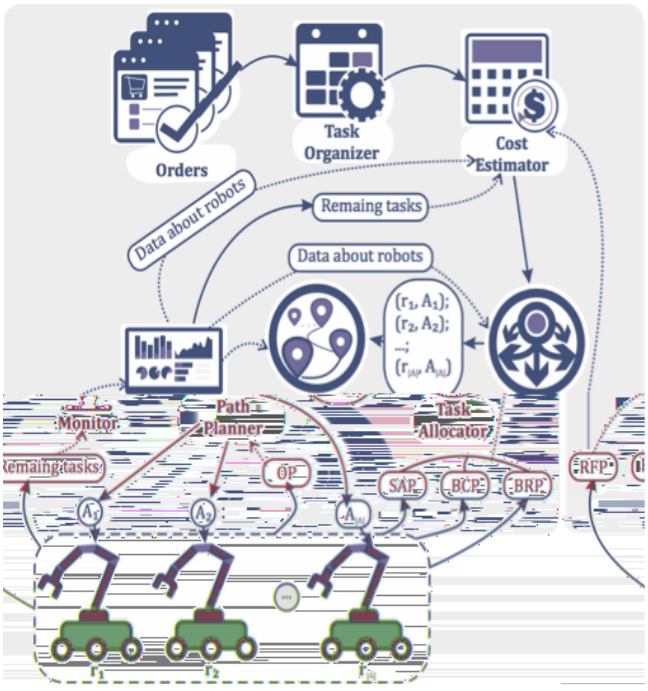

Fig. 4: Software Architecture Used Over the ROS.

The Task Allocator module defines a set of actions that each robot must perform. This module also collects information about robots and tasks from the Monitor module and the Task Organizer module. The Path Planner module gets this set of actions, computes trajectories for each robot, and delivers a path to each robot. Each robot receives its path and tries to execute. If the path is performed with success, the robot finishes its task. On the other hand, if some problem occurs, a message is triggered by the robot to the Monitor, which will check the Robot Failure Profile (RFP) to identify the problem and determine some solution based on the Execution Profiles.

VI. Problem Formulation

DDS is a powerful and efficient protocol used for asynchronous communication however in most cases other technologies are widely known and more adopted, such as Apache Kafka, which is a stream-processing software for collecting, processing, storing and analyzing data at scale. According to its own definition "Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.", in addition it provides excellent performance, low latency, fault tolerance, and high throughput, being capable of handling thousands of messages per second [16]. It might become necessary to integrate other systems with the domain managed by ROS (e.g. smart warehouse with logistics demand). One option, and here presented, to achieve the communication between a ROS domain with third-party systems is by having a Kafka broker in-between them. Then, it is possible to create nodes and assign them the responsibility to act as communication bridges between all parts involved.

As illustrated by Algorithm 1 this node is responsible for receiving a message from a consumer listening to a particular Kafka topic. As a first step it is required to get ROS library to manage the actions of the node, then it creates the objects holding the node itself and the Kafka consumer. Once all the objects are created it is possible to subscribe and listen for messages arriving from Kafka and forward them to the respective ROS topic. As a last step, it is necessary to block the current thread's execution and enable the node's callbacks, and finally shut everything down once the thread is unblocked and returns to the running state.

Similarly, as described by Algorithm 2, this node is in charge of reading messages originating in a ROS topic and publishing them in a Kafka topic. Again it is necessary to start by getting ROS to manage the actions of the node, then objects holding the node itself and, this time, the Kafka producer is created. Once the creation of the objects is done it is possible to subscribe and listen for messages arriving from ROS and publish them to the respective Kafka topic. As a last step, it is necessary to block the current thread's execution and enable the node's callbacks, and finally shut everything down once the thread is unblocked and returns to the running state.

Algorithm 1 Kafka-ROS Bridge Node

```
1:  rcl.init()
2:  node ← rcl.create_node(name)
3:  publisher ← node.create_publisher(
            message_type, ros_topic_name)
4:  consumer ← KafkaConsumer(kafka_topic_name)
5:  timer ← node.create_timer(
            period,
            publish(consumer, ros_publisher))
6:  rcl.spin(node)
7:  rcl.shutdown()
8:
9:  procedure PUBLISH(consumer, publisher)
10:     for every message in consumer do
11:         publisher.publish(message)
12:     end for
13: end procedure
```

Algorithm 2 ROS-Kafka Bridge Node

```
1:  rcl.init()
2:  node ← rcl.create_node(name)
3:  publisher ← KafkaPublisher()
4:  node.create_subscription(
            message_type,
            ros_topic_name,
            (message) -> publisher.publish(message))
5:  rcl.spin(node)
6:  rcl.shutdown()
```

The solution here presented is one possible strategy, although there can be others achieving the same goal, and can be described as a direct approach to provide a mechanism capable of exchanging messages between both systems, thus acting like a centralizer avoiding the need to have duplicated code, making it loosely coupled by isolating nodes responsibilities, providing a way to achieve scalability more easily and keeping it simple by not having additional dependencies.

VII. Experiments

This section presents the experiments performed to validate the efficiency of the proposed bridge nodes. Our goal is to have messages being created at one end, a Kafka producer reproducing an external system, get the produced messages to be routed as part of a ROS environment, and then send them back to the other end, reproducing the same external system reading responses from ROS. Fig. 5 shows the communication architecture between the following nodes:

- kafka-producer: produces random orders where the amount of products within 1 order can vary from 1 to 7 with each product having dimensions (height, width, length), weight, material and coordinates (x, y, z), representing its absolute position in a warehouse;
- kafka-broker: typical Apache Kafka broker;

- kafka-ros-bridge: bridge node implemented following Algorithm 1. It subscribes to the same Kafka topic as kafka-producer sends messages to and in addition it forwards the same orders to an internal ROS topic by using ROS String message type;
- ros-kafka-bridge: bridge node implemented following Algorithm 2. It subscribes to the same internal ROS topic as kafka-ros-bridge sends messages to and whenever new messages arrive they are sent to another Kafka topic;
- kafka-consumer: a dummy Kafka consumer that subscribes to the same Kafka topic that ros-kafka-bridge sends messages to.

Experiments have been performed on a 2,60 GHz 6-Core Intel (R) Core (TM) i7, with 16 GB 2400 MHz DDR4 of memory running on macOS Big Sur (11.5), with all parts running on different containers, with 1.941GiB of memory each, Kafka topics with a single partition and only 1 replica, and for a period of 10 minutes.

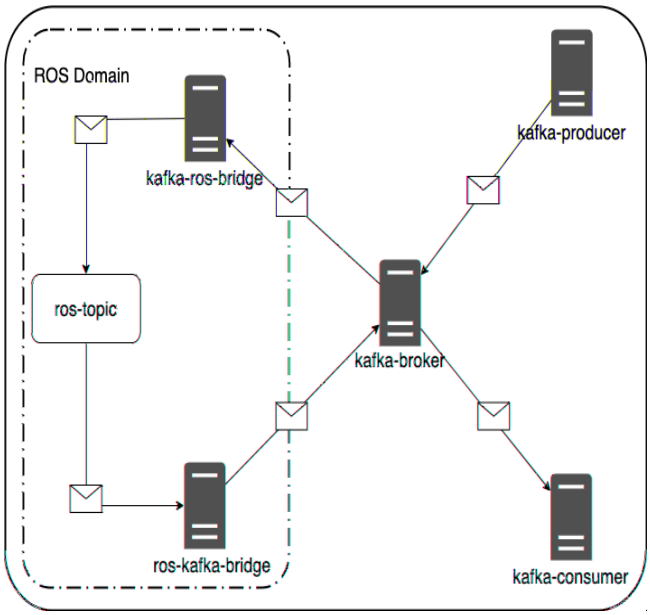

Fig. 5: Experiment Distributed Architecture.

In Table I we can see the results after running the experiment using the default QoS Profile, where the queue size is set to 10, for both publisher and subscriber. For the first case, Kafka-to-ROS reports 1,23% (24.43MiB) of memory usage before starting the experiment and a peak of 1.27% (25.17MiB) during the experiment, whilst ROS-to-Kafka reports 1.22% (24.20MiB) before starting the experiment and a peak of 1.27% (25.2MiB) during the experiment. Fastest message took 0.006791 seconds, slowest took 0.282652 seconds and 0% of the messages were lost. For the second case, Kafka-to-ROS reports 1.23% (24.45MiB) of memory usage before starting the experiment and a peak of 1.32% (26.2MiB) during the experiment, whilst ROS-to-Kafka reports 1.22% (24.32MiB) before starting the experiment and a peak of 1.30% (25.8MiB) during the experiment. Fastest message took 0.006467 seconds, slowest took 0.851981 seconds and $\sim 0,17\%$ (90) of the messages were lost. For the third case, Kafka-to-ROS reports 1.23% (24.43MiB) of memory usage before starting the experiment and a peak of 1.34% (26.54MiB) during the experiment, whilst ROS-to-Kafka reports 1.22% (24.18MiB) before starting the experiment and a peak of 1.34% (26.54MiB) during the experiment. Fastest message took 0.006247 seconds, slowest took 1.107546 seconds and $\sim 3,63\%$ (3469) of the messages were lost.

TABLE I: QoS Profile Depth=10

Producer	Kafka-ROS	ROS-Kafka	Consumer
5787	5787	5787	5787
51683	51683	51593	51593
95664	95664	92195	92195

In an attempt to make the communication between Kafka-ROS-Bridge and ROS-Kafka-Bridge more reliable (less messages loss), Table II shows the results after running the experiment with a QoS profile similar to the default, with the only difference being that the queue size (depth) was increased to 100 for both publisher and subscriber. Now for the second case shown by Table I, Kafka-to-ROS reports 1.23% (24.46MiB) of memory usage before starting the experiment and a peak of 1.33% (26.36MiB) during the experiment, whilst ROS-to-Kafka reports 1.22% (24.21MiB) before starting the experiment and a peak of 1.33% (26.43MiB) during the experiment. Fastest message took 0.006465 seconds, slowest took 1.023807 seconds and 0% of the messages were lost. For the third case shown by Table I, Kafka-to-ROS reports 1.25% (24.82MiB) of memory usage before starting the experiment and a peak of 1.34% (26.60MiB) during the experiment, whilst ROS-to-Kafka reports 1.22% (24.26MiB) before starting the experiment and a peak of 1.35% (26.74MiB) during the experiment. Fastest message took 0.006582 seconds, slowest took 0.746120 seconds and 0,23% (230) of the messages were lost.

TABLE II: QoS Profile Depth=100

Producer	Kafka-ROS	ROS-Kafka	Consumer
50152	50152	50152	50152
98733	98733	98503	98503

Final experiment, described by Table III, shows the results after increasing the depth to 200. Now for the second case shown by Table II, Kafka-to-ROS reports 1.24% (24.64MiB) of memory usage before starting the experiment and a peak of 1.43% (28.44MiB) during the experiment, whilst ROS-to-Kafka reports 1.23% (24.40MiB) before starting the experiment and a peak of 1.43% (28.35MiB) during the experiment. Fastest message took 0.006730 seconds, slowest took 1.029223 seconds and 0% of the messages were lost.

As can be seen by the results gathered after performing the experiments our solution to integrate Kafka with ROS

TABLE III: QoS Profile Depth=200

Producer	Kafka-ROS	ROS-Kafka	Consumer
97995	97995	97995	97995

accomplishes its purpose by delivering messages in both directions, from Kafka to ROS, and from ROS to Kafka. It is important to highlight that the depth values chosen for the experiments were randomly picked, thus making room for better and more optimized techniques in order to select the proper number for that variable in different situations. Although in some scenarios a percentage of messages are lost, which might not be desirable for some systems e.g. smart warehouse processing orders from customers, after increasing the throughput in the Kafka producer, we show that by leveraging the use of QoS profiles provided by DDS it is still possible to achieve a higher level of reliability when the depth of ROS nodes are increased to cope with the throughput of each test-case.

Moreover, even though our results show that ROS can be easily integrated with Kafka and it is even easier when the reliability constraint is not present due to the fact that no additional changes need to be done to the QoS profile used by ROS producer and subscriber, we intend to expand our test-cases, as similar to a real scenario as possible, in order to collect more metrics to confirm that the strategy here presented is reliable.

VIII. Conclusions and Future Work

In this work we present a strategy to integrate ROS with Kafka. The proposed solution relies on the use of two nodes that are part of the ROS environment acting as communication bridges to publish and consume messages between both systems. Results also show that QoS profiles play a key role in the system providing different levels of configuration and that by increasing the queue size used by ROS publisher an subscriber, it is possible to decrease the amount of messages lost hence increasing its reliability.

To show that the proposed strategy successfully accomplishes its purpose, randomly generated data sets were collected. As part of future works, we intend to design real case scenarios to plug in our solution to perform more detailed experiments. Furthermore, cybersecurity is another topic of extreme importance that must be tackled in the next steps of our research.

Acknowledgment

The authors are grateful for the financial support granted by CAPES Brazil to this research at the CAPES/PRINT - Call no. 41/2017 Senior Visiting Professor at Biomimetics and Intelligent Systems Group - BISG, University of Oulu, Finland. The authors would like to thank the Finnish UAV Ecosystem (No. 338080).

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In Proc. of IEEE International Conference on Robotics and Automation Workshop on Open Source Software, volume 3, page 5, 2009.

[2] M. Quigley, E. Berger, and A.Y.Ng. STAIR: Hardware and Software Architecture. In AAAI 2007 Robotics Workshop, Vancouver, B.C, August, 2007.

[3] K. Wyobek, E. Berger, H. V. der Loos, and K. Salisbury. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. In Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA), 2008.

[4] A. Koubaa, M. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. M. Shakshuki. COROS: A multi-agent software architecture for cooperative and autonomous service robots. In Cooperative Robots and Sensor Networks 2015, 2015, pp. 3–30.

[5] Why ROS 2?. Available from: https://design.ros2.org/articles/why_ros2.html. Accessed 18-Oct-2021.

[6] Apache Kafka. Available from: https://kafka.apache.org. Accessed 18-Apr-2021.

[7] Alibaba Group Announces December Quarter 2020 Results. Available from: https://www.alibabagroup.com/en/news/press_pdf/p210202.pdf. Accessed 18-Oct-2021.

[8] SHOPIFY INC. 2020 ANNUAL INFORMATION FORM. Available from: https://s27.q4cdn.com/572064924/files/doc_financials/2020/ar/40-F.pdf. Accessed 18-Oct-2021

[9] OPRoS (2021). Available: http://www.opros.or.kr.

[10] Kang, Jeong & Yu, Dong & Park, Hong. (2012). A robot software bridge for interconnecting OPRoS with ROS. 296-297. 10.1109/URAI.2012.6462998

[11] Endre Eros, Martin Dahl, Kristofer Bengtsson, Atieh Hanna, Petter Falkman, (2019). A ROS2 based communication architecture for control in collaborative and intelligent automation systems, Procedia Manufacturing, Volume 38, Pages 349-357, ISSN 2351-9789, https://doi.org/10.1016/j.promfg.2020.01.045

[12] Lee, Junwon; Lee, Kieun; Yoo, Aelee; Moon, Changjoo. 2020. "Design and Implementation of Edge-Fog-Cloud System through HD Map Generation from LiDAR Data of Autonomous Vehicles" Electronics 9, no. 12: 2084. https://doi.org/10.3390/electronics9122084

[13] R. Chaari, O. Cheikhrouhou, A. Koubâa, H. Youssef and H. Hmam, "Towards a Distributed Computation Offloading Architecture for Cloud Robotics," 2019 15th International Wireless Communications & Mobile Computing Conference (IWCMC), 2019, pp. 434-441, doi: 10.1109/IWCMC.2019.8766504.

[14] Y. Maruyama, S. Kato, and Takuya Azumi. Exploring the performance of ROS2. In Proceedings of the 13th International Conference on Embedded Software (EMSOFT). Association for Computing Machinery, New York, NY, USA, Article 5, 1–10, 2016

[15] Interface definition using .msg / .srv / .action files. Available from: https://design.ros2.org/articles/ros_on_dds.html. Accessed 04-Mar-2021.

[16] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), volume 133, pages 1-23, 2019.

[17] Object Management Group (OMG). Available from: https://www.omg.org/spec/DDS/About-DDS/. Accessed 05-Mar-2021.

[18] eProsima fast DDS Documentation. Available from https://fast-dds.docs.eprosima.com/en/latest/index.html. Accessed 05-Mar-2021.

[19] RTI Connext DDS. Available from: https://www.rti.com/products/connext-dds-professional. Accessed 05-Mar-2021.

[20] Vortex OpenSplice. URL: http://www.prismtech.com/vortex/vortex-opensplice. Accessed 05-Mar-2021.

[21] OLIVEIRA, GEORGE; PLENTZ, PATRICIA D. M.; CARVALHO, JONATA TYSKA Multi-Constrained Voronoi-Based Task Allocator for Smart-Warehouses In: 2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics (SACI), 2021, Timisoara, Romania. p.515

[22] eProsima FastRTPS. Available from: https://www.eprosima.com/index.php/products-all/eprosima-fast-rtps. Accessed 05-Mar-2021.
[23] ROS Package: ros_kafka_connector (2021). Available at https://github.com/isakdiaz/ros-kafka-connector.
[24] Turtlesim Simulator (2021). Available at http://wiki.ros.org/turtlesim