IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 16, 2021, accepted December 8, 2021, date of publication December 20, 2021,
date of current version December 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3136855

A Highly Reliable Communication System for Internet of Robotic Things and Implementation in RT-Middleware With AMQP Communication Interfaces

DAISHI YOSHINO, YUTAKA WATANOBE, (Member, IEEE), AND KEITARO NARUSE
School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu-shi, Fukushima 965-8580, Japan
Corresponding author: Daishi Yoshino (daishi-y@u-aizu.ac.jp)

ABSTRACT Since the Internet of Robotic Things (IoRT) is composed of robots with actuators, interferences with the real-world activities are necessary, and safety is essential. In addition, some IoRT services may require bidirectional communication between multiple machines. One of the communication protocols that satisfy these requirements is AMQP, a broker architecture that emphasizes reliability and high functionality in communication. Therefore, using AMQP as the communication infrastructure between components in an IoRT system, it can contribute to improving the reliability of robot operations in IoRT and diversifying messaging between robots. To verify AMQP's communication advantages, we have implemented the communication interface of AMQP in RT-Middleware, which is one of the robot middlewares, and have conducted performance and reliability tests on the effectiveness of RT-Middleware as a platform for constructing a reliable IoRT system. In the tests, we compared the communication performance of the platform with CORBA and MQTT communication interfaces implemented in RT-Middleware. The results show that although AMQP causes a small amount of latency compared to other communication interfaces, the distribution range of the latency is small, and relatively stable communication is performed. Furthermore, in the messaging quality test results, the data loss during communication by AMQP is almost a hundred times better than CORBA, and ten times better than MQTT, which mean that highly reliable messaging is possible. Based on this study's findings, we conclude that AMQP should be fully used as a communication protocol for constructing IoRT systems.

INDEX TERMS AMQP (advanced message queuing protocol), brokered publish/subscribe messaging, highly reliable Internet of Robotic Things (IoRT), MQTT (message queuing telemetry transport), RT-middleware.

I. INTRODUCTION

In general, robots make decisions about their next actions based on environmental data obtained from onboard sensors and their internal states. However, because robots' physical payload and available computational resources are limited in a closed system, they only partially demonstrate their capability. A way to expand robots' applications is to use resources existing outside the robots' system. The Internet of Robotic Things (IoRT) attempts to broaden Internet use by incorporating external resources into robots via the Internet. IoRT systems are composed of multiple components, such as robots, sensors, and clouds, connected to the Internet. Communication protocols connect each component, making them a system of systems. According to P. P. Ray, IoRT is divided into five major layers, one of which is the layer responsible for establishing connections on the Internet. He introduced several Internet of Things (IoT) protocols that enable it [1], some of which exchange messages via a messaging architecture with a broker. The messaging protocol with the broker removes the communication constraints imposed by network address translation (NAT) mechanisms and firewalls and facilitates bi-directional communication between endpoints on the Internet, making it an essential technology to meet the first proposition of the IoT, ''Connecting all things.'' In particular, message queuing telemetry transport (MQTT)

The associate editor coordinating the review of this manuscript and approving it for publication was Fan-Hsun Tseng.

is a representative protocol for publishing/subscribing (pub/sub)-type messaging and has simple specifications and lightweight messaging as its main features [2]. Both features contribute to reducing the communication overhead, thereby improving communication performance and power saving. In IoT systems, some field-side devices may have limited machine resources, and securing a stable power supply may be difficult. For this reason, MQTT is useful for increasing the connectivity and scalability of IoT systems, independent of the performance aspects of field-side devices. Because of the above, constructing IoT systems with MQTT has become a worldwide best practice.

However, it is important to note that there is almost a trade-off between performance and reliability improvement in a system. It raises the concern that MQTT, which has low latency and high performance, may not support mission-critical systems. MQTT's high-throughput messaging is due to its simplicity, and messaging quality assurance is minimized. As a result, messaging service quality is degraded, especially in unstable networks.

The objective of this study is to develop a reliable and robust IoRT system. We discuss the state of communication in IoRT systems and propose a communication model and system configuration method for realizing it. We provide an implementation and show the verification results to verify the model and method's effectiveness. As a concrete implementation, we use advanced message queuing protocol (AMQP), a pub/sub messaging protocol with a broker that focuses on reliability, and extend it as a communication interface for robotic technology (RT) middleware (RT-Middleware), one of the robot middlewares. By combining the AMQP communication interface with the RabbitMQ server, AMQP's main message broker, it is possible to construct a more reliable and robust IoRT system using only RT-Middleware. It implies that RT-Middleware extends beyond middleware and into IoRT platforms. In addition, if a system is implemented with RT-Middleware, reliability can be realized by constructing a robust AMQP bridge between robot systems developed with previously developed functions.

However, while improving reliability, there is a concern that communication performance, to some degree, particularly in real-time environments, may be sacrificed. In this study, we have conducted actual communication experiments and evaluated communication performance. As a result, we have confirmed that the implementation of AMQP has many advantages and few disadvantages.

This study is organized as follows. Following our discussion on the differences between IoRT and IoT in Section 2, we present the advantages of AMQP in IoRT and the features of the RabbitMQ server, a message broker that augments AMQP, with a functional comparison with MQTT and related research. In Section 3, we discuss issues in constructing an AMQP-based IoRT system using existing IoT platforms and robot middleware and explain how and why we came to choose RT-Middleware as the infrastructure for constructing an IoRT system. In Section 4, we explain the basic concepts of RT-Middleware specifications and discuss why they apply to systems other than robot systems. In Section 5, we show a communication structure with AMQP on RT-Middleware. In Section 6, we evaluate the performance of various communication interfaces, including AMQP on RT-Middleware with actual devices, and discuss the effectiveness of the developed AMQP communication interface. Finally, Section 7 summarizes this study and discusses future research.

II. THE NEED FOR RELIABLE PUB/SUB MESSAGING IN IORT

MQTT, a lightweight messaging solution, which has become almost a de facto standard in the field of IoT, has been introduced for various research and commercial purposes. AMQP, by contrast, has a poor track record of adoption in the field of IoT and IoRT. The reason could be a high communication overhead caused by the protocol's complexity because of the emphasis on reliability. In the following, we will contrast the need for lightweight pub/sub messaging represented by MQTT and reliable messaging represented by AMQP.

Most performance evaluation tests in related studies have concluded that MQTT has higher throughput and lower latency than AMQP, indicating a negative report for AMQP. For example, D. Happ *et al.* conducted a performance evaluation test using a pub/sub messaging system consisting of multiple virtual machines launched on a public cloud and reported that AMQP has higher latency than MQTT for small- and medium-sized payloads [3]. However, it has also been reported that there is almost no difference in latency between the two protocols for large payloads. This finding is probably due to the difference in the header size in a message. If the header is compact, as in MQTT, the smaller the payload size, the greater the benefit in the communication performance. Conversely, as the payload size increases, the benefit decreases, and the protocol loses its advantage. However, for small payloads, MQTT outperforms AMQP in terms of communication performance, as evidenced by the results of protocol performance evaluation tests in a smart factory environment conducted by D. Bezerra *et al.* [4]. In the tests, a scenario was implemented for each protocol and the total processing time of a series of processes was measured. AMQP did not outperform MQTT in the tests, particularly in low-bandwidth networks. They concluded that this result was due to the different message sizes specified in the protocols. AMQP's protocol complexity over MQTT improves robustness, but more resources are needed to achieve it. Because devices with small machine resources are typically introduced to IoT systems, we are interested in the suitability of AMQP, which requires more resources, as an IoT protocol. A. Chaudhary *et al.* have pointed out that there is no AMQP library for microcontrollers with small memory, such as Arduino, and they have called for the establishment of an open-source community to expand the application of AMQP [5].

Thus, there is a view in the literature that AMQP is inferior to MQTT in terms of communication performance.

However, its suitability as a protocol for IoT and IoRT cannot be determined by improving communication performance through lightweight messaging. Rather, in some cases, the robustness of systems and the provision of reliable services such as high-quality messaging may also be evaluated. With AMQP, as a system constructed on the Internet, which represents a network with a high degree of uncertainty, there is little demand for real-time communication. Therefore, we need to discuss in depth the necessity and applicability of AMQP in IoRT, contrasting with the differences from the IoT.

A. DIFFERENCES BETWEEN IOT AND IORT, AND COMMUNICATION PROTOCOLS FOR ESTABLISHING IORT SYSTEM

Assuming that IoT and IoRT both consist of cloud services on an infrastructure side and a group of devices on a field side across the Internet, the critical differences between IoRT and IoT can be narrowed down to the following two points.

1) Field-side robots interface with the physical world by receiving commands from the cloud or other devices and operating the drive system according to those commands.
2) There is interactive communication between robots on the field side or via the Internet (multi-robot system).

As for point 1), the difference between IoRT and IoT is whether sensors or actuators are at the center of the system. In IoT, which is sensor-centered, there are few situations where actuators influence objects directly in physical space. By contrast, in IoRT, since robots, including drive systems, are at the center, the interaction between physical entities is essential and physical safety must be ensured. For this reason, the integrity of external commands to robots should be ensured. Any deficiencies or inconsistencies in commands delivered to them can make robots run out of control.

AMQP (ver. 0.9.1) includes two features to achieve messaging integrity: transaction control and message queue [6], [7]. The former guarantees the full reachability of messages, whereas the latter guarantees the arrival order of messages as well as their reachability. In transaction control, a series of messages between clients from a sender to a receiver are handled synchronously as a single process. This single process introduces a significant communication delay due to the locking of the message queue in the process, but it makes the exchange of messages between clients more robust. Conversely, in AMQP, the output-side queue corresponding to the receiving client is defined in the broker, and messages are processed one by one in a first-in–first-out (FIFO) format. The FIFO format guarantees the reachability and arrival order of messages originating from the sender. The high-reliability messaging capabilities of AMQP will be of great value in mission-critical systems such as social infrastructure, where constant operation is essential. It has been used as the messaging infrastructure for the financial trading systems of Deutsche Börse and J.P. Morgan [8].

Meanwhile, the MQTT protocol has no provision for queues, and message reachability by quality of service (QoS) is guaranteed only by acknowledgments between clients and brokers [9]. In other words, MQTT has no way to confirm if messages have been exchanged reliably between clients at the protocol level. Since there is no queue corresponding to the receiving client, the arrival order of messages is not guaranteed. Table 1 summarizes the differences in characteristics between AMQP and MQTT, mainly regarding reliability in messaging.

TABLE 1. Differences in reliability-related features between AMQP and MQTT.

	AMQP (ver. 0.9.1)	MQTT (ver. 3.1.1)
Architecture	Brokered messaging with queue	Brokered messaging
Minimum size of the header in one message	7 bytes (7 octet)	2 bytes
Specification of output-side queue	Yes	No
Transaction control in messaging	Yes	No
Message routing capabilities	Routing with "exchanges" and "queues" bindings, including topic-based routing	Topic-based routing

N. Q. Uy *et al.* [10] demonstrated the high reliability of AMQP messaging through communication performance evaluation tests in an Internet-like test environment by comparing AMQP with MQTT. One of the tests is designed to measure the delay by continuously sending packets without setting the frequency in a network environment loaded by the tool. In the test, MQTT exhibits packet loss, regardless of QoS, whereas AMQP has no packet loss in any of the test cases.

The stability of AMQP in high frequency communication has been clearly demonstrated in the results of performance evaluation tests of various messaging middleware used in the Industrial Internet of Things (IIoT) field conducted by G. Andrei *et al.* [11]. The two messaging specifications of the data distribution service (DDS) [12] and AMQP are employed in the performance evaluation. Both of the specifications are mainly for pub/sub messaging, but DDS is brokerless, while AMQP is brokered. Nevertheless, the results of the performance evaluation tests show that AMQP is able to maintain higher messaging throughput even at high frequencies, and the number of messages that stay in the queue is stayed at the minimum throughout the tests. In contrast, the DDS implementations have unstable communication, with sudden and/or large delays from the beginning. Based on the above, AMQP is reported to provide more stable communication than the two DDS implementations.

Several studies, however, have shown that AMQP's messaging is unreliable on highly unstable networks, where connection loss between the sender client and the broker occurs frequently. In the protocol performance evaluation test conducted by T. Moraes *et al.* assuming a network failure, two communication paths were set up between the sender client and the broker, and at regular intervals, a network failure occurred on one of the paths, and the effective path was switched [13]. AMQP explicitly states that when a connection is lost because of a network failure, the sender client takes a long time to establish another connection on a new communication path, resulting in a large amount of packet loss. However, this is the result obtained by the broker with default settings, and it can be resolved by making appropriate adjustments in the configuration.

Nevertheless, considering that AMQP uses message queues in the broker to guarantee the arrival order of messages, reliability depends on the condition that messages arrive at the broker from the sender client. Therefore, we cannot completely address the concern that the advantage of AMQP reliability may deteriorate in a highly unstable network environment, where maintaining connections is difficult. However, from the opposite perspective, AMQP can only ensure reliable messaging if messages from the sender client to the broker are delivered consistently. In other words, deploying a broker for each endpoint, including robots, is sufficient. The testbed developed by D. Lu *et al.* for cloud computing for mobile robots is based on AMQP, and the implementation is exactly following the above deploy [14]. A broker is deployed not only on the cloud but also on mobile robots in the field, which contributes to the improvement of messaging quality while making good use of the protocol's special message routing to enable broadcast distribution between robots without going through the cloud.

In addition to the protocol specifications, there are also examples of messaging middleware that extends the protocol and applies it to IoRT. The RabbitMQ server provided by RabbitMQ [15] messaging middleware supports messaging by MQTT not only the main one by AMQP. This makes it possible to operate a system that combines highly reliable communication using AMQP and bandwidth-economical one using MQTT. The human fall detection system in the smart home developed by L. Killian *et al.* is also based on this [16]. This system consists of three software components: a surveillance camera to detect falls and room clutter, a communication robot to alert the user about falls, and a server-side logger. These can be used in different ways via RabbitMQ, such as MQTT for urgent communication and AMQP for more reliable one.

Thus, in many cases, IoRT systems constructed using AMQP focus more on the high functionality and multifunctionality of the protocol and messaging middleware than on reliability. The previous example is an application of the broker's message routing function in AMQP. In addition to the topic routing in MQTT, in AMQP, various message delivery modes can be selected, such as one-to-one push/pull type and broadcast delivery. The direct reason for this capability of expressing multiple delivery modes is that each component (''exchange'' and ''message queue'') associated with each sender client and receiver client is provided for the broker. In AMQP, as shown in Figure 1, various communication routes are represented by the binding method of ''exchange'' and ''message queue'' and by matching the ''binding key'' associated with the binding with the ''routing key'' in the message [17]. This diverse message routing capability has been applied as a control plane, a key element of network control in cloud services provided by NASA and Red Hat [8]. The network robotics framework implemented by J. Lim *et al.* on smartphones is another example of how AMQP can be used successfully [18]. In AMQP, when multiple consumers are assigned to a message queue, messages are passed around to the consumers in a round-robin order. In the framework, this function is used to offload tasks generated on the robot side, distributing them in turn to multiple task processing prepared on the server side.

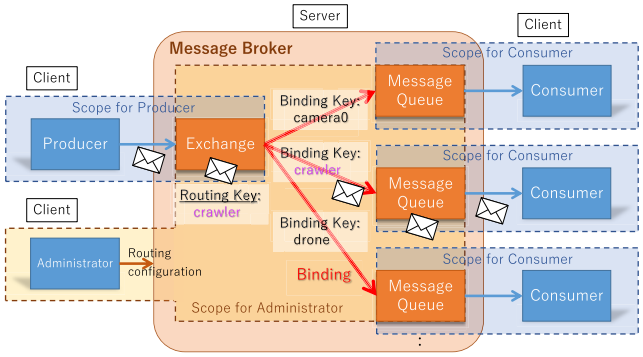


FIGURE 1. Overview of AMQP ver. 0.9.1 messaging architecture.

AMQP's messaging functions can be used in the communication in the multi-robot system described in point 2). In addition to the broadcast communication described above, AMQP can create a variety of data flows by changing a robot's communication partner according to the broker's routing settings. Furthermore, since AMQP uses a broker as its communication architecture, even if each robot belongs to a different local network, message routing between robots is possible if they can connect to the broker, which can work as a server. In other words, AMQP facilitates crossing NAT and firewalls and encourages connectivity between robots on the Internet. In addition, by allocating multiple consumers to a single message queue, it is possible to divide a task assigned to a single robot into small tasks and allocate them to other robots, thereby distributing the workload. However, in IoT systems, where the field side is mainly composed of sensors, there are few situations where communication between sensors occurs, and only MQTT, a lightweight messaging system, is sufficient for sensor-to-cloud communication.

B. ENHANCE SYSTEM RELIABILITY AND ROBUSTNESS BY CLUSTERING THE BROKER ARCHITECTURE

The advantage of AMQP in IoRT has been found in the characteristics of the protocol, but high-quality and advanced messaging can be realized only with system sustainability. Traditionally, the broker is located alone in the center of the system, and it has been a cause of system failure in the event of a server outage. One way to solve this problem is to cluster the brokers. In clustering, multiple servers with deployed brokers are linked together to provide redundancy and risk distribution of broker functions. Therefore, even if some of the linked servers fail, the system can continue running as long as at least a single server with a broker function is active.

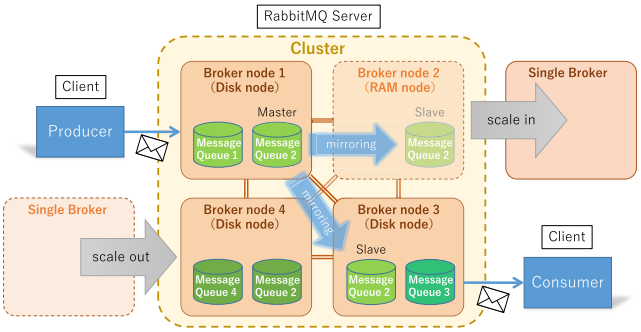


FIGURE 2. RabbitMQ cluster consisting of four broker nodes.

The RabbitMQ server also uses multiple brokers, offering a solution to the single point of failure problem inherent in the traditional broker architecture by clustering a group of broker nodes. An example of the clustering configuration in the RabbitMQ server is shown in Figure 2. A cluster consists of two types of broker nodes: disk nodes, which use both memory and storage for storing runtime information other than message queues, and RAM nodes, which use only memory. Disk nodes are the default nodes, and at least one node is required in a cluster. RAM nodes, by contrast, contribute to throughput improvement but are typically not used because they require synchronization with disk nodes at startup and highly depend on the disk nodes. By coordinating broker nodes in a cluster, it is possible to achieve redundancy and risk distribution in a system. As long as one disk node is running, the cluster does not stop, promoting system sustainability in the event of a failure in the server. Furthermore, the ability to scale in and out of the cluster without stopping the system also promises system sustainability during maintenance. In this way, the RabbitMQ server promises to enhance safety and further improve reliability in IoRT systems, in addition to realizing a fault-tolerant system by clustering a group of the broker nodes. In other words, the combination of AMQP and the RabbitMQ server can have a synergistic effect on a system's robustness and reliability improvement.

III. REALIZATION OF HIGHLY RELIABLE IORT PLATFORM

As shown in the previous section, one characteristic of IoRT systems that differs from IoT systems is that they require flexible and diverse messaging functions for more advanced communication among robots while ensuring safety so that robot actions do not endanger humans in the real world. AMQP satisfies these two requirements through reliable and highly functional messaging, contributing to the construction of more reliable IoRT systems. However, to construct an actual IoRT system using AMQP, there must be an environment in which AMQP's broker service is provided on the cloud on the infrastructure side, and AMQP's client library can be easily used in a robot system on the field side. In this section, we first summarize the issues in using AMQP at each layer when IoRT is largely divided into the infrastructure side and the field side.

As a prerequisite for constructing an AMQP-based IoRT system, it is necessary to prepare AMQP broker services on the infrastructure side. The easiest way to fulfill this prerequisite is to use IoT platforms provided by major vendors.[1] If you do not use them, you need to deploy an AMQP broker on a virtual server on Infrastructure as a Service or prepare an AMQP broker service yourself, as on-premises. Although the latter requires more development labor time, it allows for greater flexibility in system development and flexible customization of various business logic.

By contrast, to realize data communication by AMQP in the field-side robot, the simplest way is to embed an AMQP client in the desired robot system. In robot operating system (ROS) [23], which is the de facto standard for robot middleware, the transport for communication between the nodes that make up the robot system is fixed at TCPROS or UDPROS, which are ROS's original specifications, and it is difficult to switch to other communication protocols, including AMQP. Therefore, one way to use AMQP in an ROS robotic system is to incorporate an AMQP client into the ROS node. However, in this method, IoRT is managed by separate solutions, that is, ROS on the field side and AMQP messaging middleware on the Internet side, which may increase labor time and cause complications in maintenance and operation.

In contrast, in ROS2 [24], which is attracting attention as the next generation of robot middleware, communication functions are separated from the middleware infrastructure, and the ROS middleware (rmw) layer that abstracts communication functions, mainly pub/sub communication, including QoS, is provided. In ROS2, the communication interface is typically selected from DDS implementations of each vendor using the application programming interface provided by rmw. However, if communication functions equivalent to rmw are provided, it is possible to switch to other implementations. This suggests that rmw implementation using AMQP is also possible, but ROS2 does not support switching communication interfaces for every communication port in

[1]Typical examples of IoT platforms, such as Amazon's AWS IoT Core [19], Google's Cloud IoT Core [20], IBM's Watson IoT Platform [21], and Microsoft's Azure IoT Hub [22], all support HTTP and MQTT, but as of July 2021, only Azure IoT Hub supports AMQP. Therefore, Azure IoT Hub is the only choice for constructing an AMQP-based IoRT system using IoT platforms from major vendors.

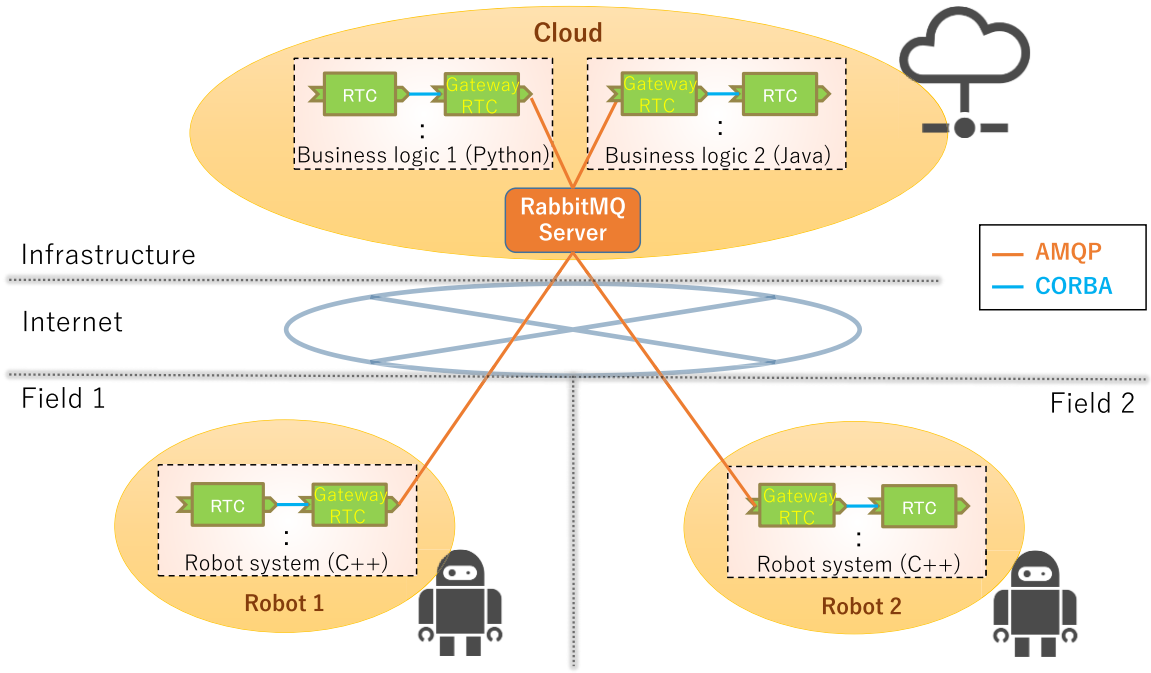


FIGURE 3. An IoRT system model expressed on RT-Middleware.

each node, and different types of communication interfaces cannot be contained in a single node. Therefore, the conventional way to use AMQP and DDS in ROS2 is to embed an AMQP client in the node, as in ROS. However, because ROS2 can effectively control the messaging quality through DDS-based QoS, which may provide the same reliability as AMQP, the system cannot be connected effectively as the broker architecture. This is because DDS is optimized for peer-to-peer communication. To apply it to different local networks through the Internet, we need to design and tune the routing settings. In contrast, in the broker architecture such as AMQP, each client in a different local network only needs to connect to a broker on the Internet, and there is no need to configure the router for NAT crossing. In other words, it is possible to develop a reliable IoRT system using only ROS2 and DDS, but it is not as simple as developing a system using AMQP.

As described above, ROS and ROS2 do not provide an environment that allows easy use of the AMQP client library, and if the AMQP client is embedded in a ROS node, the system is separated into a robot system in the field and an IoT system on the Internet. It is necessary to manage the entire system through two solutions, ROS and AMQP. In contrast, another robot middleware, RT-Middleware, allows each component of a robot system to have a different communication protocol, such as AMQP, implemented without modifying the underlying software. In other words, as shown in the example in Figure 3, RT-Middleware has the potential to become an IoRT platform that can cover all layers of system construction in IoRT, from cloud computing and robot systems to communication systems that connect them. It means that in IoRT systems, where separate solutions are typically applied to each layer, a single RT-Middleware solution can be used to construct and operate the entire system. Integrated management of IoRT simplifies the maintenance and operation of the IoRT system and reduces design, development, and operation costs. Table 2 summarizes the differences in the functions of each robot middleware in integrating heterogeneous systems that comprise IoRT. In this study, we introduce RT-Middleware, which can provide such benefits and extend the AMQP communication interface to RT-Middleware, making it a platform for constructing a reliable IoRT system.

IV. OVERVIEW OF RT-MIDDLEWARE AND RT-COMPONENTS

RT-Middleware is the middleware for constructing robot systems and has been specified mainly by the National Institute of Advanced Industrial Science and Technology (AIST), Japan. A robot system constructed by RT-Middleware is called an RT-System, and an RT-System consists of one or more components called RT-Components (RTCs). In other words, RT-Middleware is a component-oriented middleware based on a distributed control system [25]. The component models and interfaces in RTCs have been standardized by the Object Management Group (OMG), a standardization consortium, and the specifications are given by a platform-independent model, which does not depend on a specific platform [26]. Therefore, regardless of the platform, RT-Middleware is used in robot implementation as long as it adheres to the RTC specifications. As a result, although

TABLE 2. Available functions of robot middleware in IoRT system construction.

	ROS	ROS2	RT-Middleware
Base standard for robot middleware	TCPROS or UDPROS	DDS (OMG standard)	CORBA (OMG standard)
Names of the components that comprise a robot system	ROS nodes	ROS2 nodes	RT-Components (RTCs)
Standardization of robot middleware	No	No	Partially yes, RT-Component is an OMG standard
Commutativity of communication interfaces in components	No	Yes	Yes
Capabilities of including heterogeneous communication interfaces in components	No	No	Yes

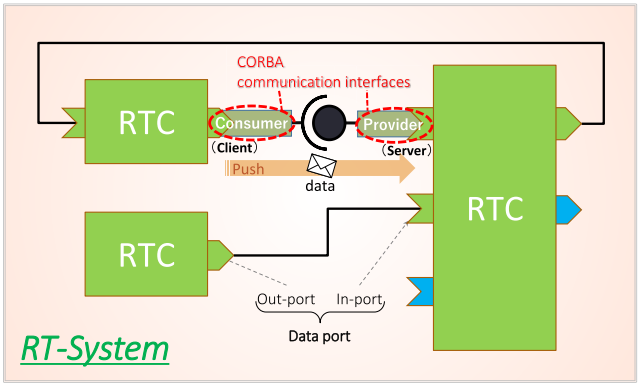


FIGURE 4. Example of RT system on OpenRTM-aist and overview of communication architecture between data ports of RTC.

it depends on the base platform, it can be a framework to construct more advanced systems beyond the middleware of robot systems.

However, the RTC specification advocates the use of common object request broker architecture (CORBA) [27] as a platform-specific model for distributed systems on networks [26]. CORBA is also used in OpenRTM-aist [28], a reference implementation of RT-Middleware [29], and it is the default communication interface for port-to-port communication in RTCs. There are two types of ports in RTCs, and we extended the communication interface for the data port, which is used for continuous data transmission and reception, in this study.

Figure 4 shows an example of RT-system construction using OpenRTM-aist and the communication structure of data port-to-data communication using the CORBA communication interface. As shown in the figure, in the CORBA communication interface, the pairs of data ports that are the targets of data transmission and reception have a client/server relationship. In other words, the consumer module of the sending out-port implements push-type communication using the service provided by the provider module of the receiving in-port. This tightly coupled client/server architecture is a communication model suitable for robotic systems, as it is expected to ensure real-time performance through direct communication between endpoints. However, if either the client or the server is missing, the system is no longer viable, making the system difficult to handle to operate on unstable networks such as the Internet as well as less scalable. On an unstable network, pub/sub messaging with loosely coupled architecture is the technology for improving the continuity and scalability of the system. In addition, AMQP, a protocol designed to maintain the quality of messaging, is adopted as a new communication interface in this study.

V. SYSTEM ARCHITECTURE WITH AMQP COMMUNICATION INTERFACE

To add the AMQP communication interface for data port communication to OpenRTM-aist by C++, we have developed two new communication modules. One of the communication modules is the producer module for data transmission, which is embedded in the out-port. The other is the consumer module for receiving data, which will be embedded in the in-port. These communication modules are developed based on AMQP-CPP [30], which is an open-source C++ library for building AMQP clients.

For example, the message transmission function in the producer module is shown below.

Function 1 Message Transmission Function in the Producer Module

```
InPortConsumer::ReturnCode OutPortAmqpcppProducer::
put(cdrMemoryStream& data){
  const int bufleng = static_cast<int>(data.bufSize());
  const void* data_p = data.bufPtr();
  const char* cstr_data = static_cast<const char*>(data_p);

  try{
   amqp_publish(cstr_data, (uint64_t)bufleng);
   return InPortConsumer::PORT_OK;
  }
  catch (...){
    return CONNECTION_LOST;
  }
  return UNKNOWN_ERROR;
}
```

The message transmission function "put" is called from the RT-Middleware side and is responsible for periodic message transmission using AMQP. The function "put" first obtains the data length from the CORBA data object received as an argument from RT-Middleware side, and then performs data type conversion (lines 3-5 in Function 1). This is so that the data can be sent according to the AMQP-CPP specifications. After that, the message is sent by AMQP using an originally implemented function (line 8 in Function 1). In the default OpenRTM-aist, in the function "put," the client/server type data transmission procedure using CORBA is performed. The CORBA communication interface is implemented in such a way that the client sends and receives data by using the server-side service. However, in AMQP, the data receiver is also a client, so it is necessary to implement a callback function that is automatically called when a message is received. For example, the callback function (message receiving function) in the consumer module is shown below.

Function 2 Message Receiving Function in the Consumer Module

```
1: void InPortAmqpcppConsumer::amqp_consume(){
2:   channel->consume(amqp_get_qname())
3:     .onReceived([&](const AMQP::Message &msg, uint64_t
deliveryTag, bool redelivered){
4:     const char* pld = msg.body();
5:     void* part = (void*)pld;
6:     const uint64_t len = msg.bodySize();
7:
8:     ::OpenRTM::CdrData tmp((CORBA::ULong)(len),
9:                            (CORBA::ULong)(len),
10:          static_cast<CORBA::Octet*>(part), 0);
11:
12:    cdrMemoryStream cdr;
13:    bool endian_type = m_connector->isLittleEndian();
14:    cdr.setByteSwapFlag(endian_type);
15:    cdr.put_octet_array(&(tmp[0]), tmp.length());
16:
17:    BufferStatus::Enum ret = m_buffer->write(cdr);
18:  });
19: }
```

The callback function for message reception is responsible for passing the contents of messages received through AMQP messaging to the RT-Middleware side. The callback function first retrieves the data and data length from the message (lines 4-6 in Function 2), and then converts the retrieved data into a CORBA data object (lines 8-15 in Function 2). After that, data passing to RT-Middleware is completed by writing to the in-port buffer (line 17 in Function 2). With the two functions introduced above, AMQP messaging via message broker between out-port and in-port, and thus between RT-Components, is established. RabbitMQ Server is used as the message broker, and it runs independently from RT-Middleware. Figure 5 depicts the communication structure of the RT-System when the AMQP communication interface is used.

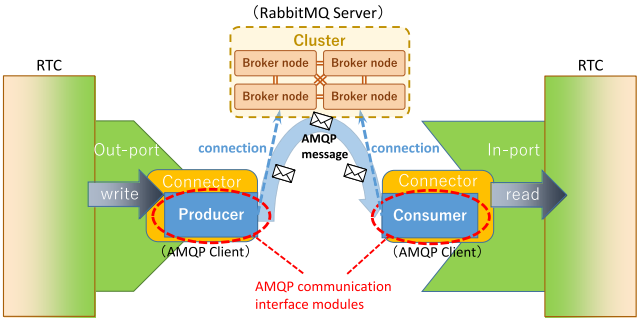


FIGURE 5. System architecture with AMQP communication interface on OpenRTM-aist.

Each external module includes a client for management so that message routing in the RabbitMQ server can be configured from either the out-port side or the in-port side. If RTSystemEditor is used, one of the RT-System management tools in OpenRTM-aist, configuration values can be passed from RT-Middleware to the RabbitMQ server by entering properties in the connector profile screen. However, the RabbitMQ server cannot be configured as a cluster via the AMQP communication interface and must be configured separately on the server side. Although the configuration is separate, the AMQP communication interface supports communication using both single and cluster brokers. In this way, the flexibility of the communication interface enables the efficient construction of AMQP communication systems on OpenRTM-aist.

VI. PERFORMANCE EVALUATION OF COMMUNICATION INTERFACES

Based on the messaging characteristics of AMQP, two types of performance evaluation tests are conducted in this section: one is a performance evaluation test on real-time performance and the other is one on messaging quality. In the former, we assume a cloud-to-robot communication and measure the round-trip delay time between endpoints on a TCP/IP network. In the latter, we measure the number of message arrival order errors and missing messages in one-to-many communication in a high-load experimental environment. The results of the latter experiment are used to verify the effectiveness of AMQP's advantages, and the results are compared with those of the former experiment to see how well the balance with real-time performance is maintained.

A. TEST ENVIRONMENT AND TOOLS

For each of the performance evaluation tests, we have developed several new test RTCs on OpenRTM-aist ver. 1.1.2 and configured a test system by combining them. In the test system, each of the test results can be obtained by switching the communication interface of the data port in the RTC to various protocols. The communication interfaces to be tested are as follows. As a side note, the modules corresponding to any of the communication interfaces are implemented

in C++. Therefore, all test systems are constructed on the OpenRTM-aist C++ version.

1) CORBA communication interface
2) MQTT ver. 3.1.1 communication interface
3) AMQP ver. 0.9.1 communication interface

Interface 1) is the default communication interface provided by OpenRTM-aist. Originally, CORBA uses client/server-type synchronous communication, but in OpenRTM-aist, asynchronous communication can be supported by selecting "new" as the subscription type in the settings when connecting data ports in RTCs. In the tests, we introduce this setting for asynchronous communication MQTT and AMQP. The implementation module for 2) was originally developed by us for research [31], and it uses mosquittopp [32], a C++ client library for Eclipse Mosquitto. In MQTT, you can choose 0, 1, or 2 as the QoS for messaging, in the order of decreasing quality. The value of QoS is directly related to the number of acknowledgments in messaging between the client and the broker. Specifically, when QoS is 0, 1, and 2, there are 0, 1, and 2 acknowledgment(s) respectively. As the value of QoS increases, the possibility of reaching the message increases, while the increase in the number of communications causes the delay. For the real-time performance test, QoS $= 0$ is selected to improve the throughput of the broker. However, for the messaging quality test, we attempt each QoS value ranging from 0 to 2 and verify whether there is a difference in messaging quality. During the tests, the QoS between each client of publisher and subscriber should be matched. The last one, 3), is a communication interface that we newly developed in this study, as mentioned in Section 5. In each of the two tests, no transaction control is performed, and the default messaging control is used. As for the routing settings, the push/pull type is selected for the real-time performance test, and the broadcast type is selected for the messaging quality test. A single message queue is assigned to each receiver client.

TABLE 3. Default buffer capacity for each messaging broker.

Protocol	Broker	Buffer capacity
MQTT ver. 3.1.1	Mosquitto ver. 1.6.4	When QoS is 0, there is no buffer; when QoS is 1 or 2, the maximum number of messages that can be processed at the same time is 20, and queue waiting for processing is allowed up to 100 messages per client.
AMQP ver. 0.9.1	RabbitMQ ver. 3.7.17	Allow up to 48 MB of disk space per server, regardless of the number of queue settings.

Message brokers are required when configuring a test system using MQTT and AMQP communication interfaces, and we use Mosquitto MQTT broker [33] and RabbitMQ AMQP server [15]. Both are open-source software brokers. Since the Mosquitto broker cannot be clustered, the RabbitMQ server has been configured as a single disk node instead of a cluster. The configuration of both brokers is left as default and no changes are made. The broker for each protocol and the buffer capacity of the default setting are summarized in Table 3. Although MQTT does not have a provision for queues in the protocol specification, Mosquitto has implemented its own queue and buffers messages when QoS is 1 or 2.

TABLE 4. Overview of machine specifications.

	OS	CPU	RAM
Machine A	Ubuntu 16.04	Intel Core i7 6200U 2.5 GHz	8 GB
Machine B	Ubuntu 16.04	Intel Core i7 8550U 1.8 GHz	8 GB

The equipment used for the test system is two PCs and a router. In the test, each PC belongs to the same local network formed on the router. Either a wired or wireless local area network (LAN) is selected as the connection method between the PC and the router, depending on the type of performance test. When constructing a network with a wireless LAN, a 5-GHz band is used to avoid radio interference. If we consider the two PCs as Machines A and B, the simplified specifications are as shown in Table 4.

B. EXPERIMENTAL CONFIGURATIONS

An overview of the configuration of the performance test is shown in Figure 6. In the figure, the configuration for the real-time performance test is shown in (a). Of the equipment prepared for the test, Machine A and the router are connected via a wired LAN, and machine B and the router are connected via either a wired or wireless LAN. In the test, the effect of differences in the network environment on latency is also verified via two test cases: the first is the test case (a-1) and the last is the test case (a-2). In both test cases, RTC1, which measures the round-trip delay, is placed in Machine A, and RTC2, which returns the received message, is placed in Machine B, forming a test system. Each MQTT broker and AMQP broker is deployed in Machine A only. Therefore, each MQTT and AMQP client in Machine B is connected to the broker in Machine A through the router. To handle the payload in a message under the tests, five types of small string-type dummy data of 8, 16, 32, 64, and 128 bytes are prepared. In the tests, for each dummy data size, the message is sent 1,000 times at 100 Hz and repeated 100 times at 10-s intervals. It means that a total of 100,000 round-trip delay times are measured for each test. In the test results, the mean, first quartile, median (second quartile), third quartile, maximum, and minimum values are measured and summarized as a box-and-whisker diagram. The buffer size of the data port in the RTC is fixed at 1,000 according to the number of consecutive transmissions, regardless of the communication interface or test case.

The performance evaluation test for messaging quality is shown in Figure 6(b). The equipment used, the location of the brokers, and the type of the dummy data in the messages are the same as in test (a), but the test is conducted in an unstable and high-load environment to verify messaging reliability. First, the test assumes only a wireless LAN test case and does not cover tests on a network built only with

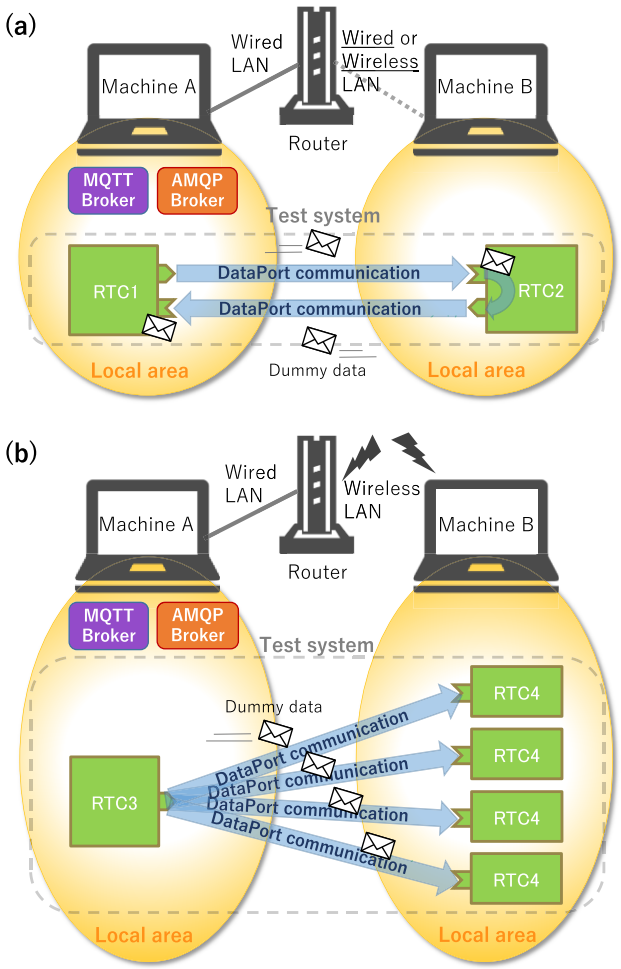


FIGURE 6. (a) Experimental setup for the performance evaluation test. (b) Experimental setup for messaging quality test.

a wired LAN. Only the router and Machine B communicate via a wireless LAN, and as in test (a), Machine A and the router are connected via the wired LAN. Next, in this test case, we configure a test system that performs one-to-many communication by preparing multiple RTCs on the receiving side. Specifically, Machine A has one RTC that sends messages to the destination, and Machine B has four RTCs that receive messages and count the number of arrival order errors and deficiencies, thereby making one-to-four communication possible and increasing the load on the receiving side. In addition, the number of consecutive message transmissions is set to 10,000 at 100 Hz, which is an additional load on the communication path and the broker. The total number of messages to be received by the four RTCs is 400,000 since this continuous transmission is repeated 10 times at 20-s intervals per test. The number of errors in the order of arrival and the number of missing messages for each communication interface are summed up and summarized in plots. The number of arrival order errors is counted when messages are not delivered in sequential order. Therefore, the count is not limited to cases where messages are delivered in the wrong order but also includes cases where the order of messages is skipped.

The above is the test configuration for verifying the reliability of messaging. The setting of the buffer size of the data port in the RTC should be noted. The buffer of the data port is specific to RT-Middleware [34]; in other words, an appropriate buffer size is ensured by verifying the reliability of OpenRTM-aist, regardless of the communication interface. For this reason, the buffer size should be considered separately from the reliability verification of the messaging middleware, which is an implementation of MQTT and AMQP. Therefore, in the tests, we also cross-sectionally verify the reliability of OpenRTM-aist when the size of the buffer in the data port is changed from 0 to 10,000. In addition, the buffer size is the same for both the out-port and in-port.

C. TEST RESULTS AND DISCUSSION

The results of the real-time performance test (a) are summarized in a box-and-whisker plot in Figure 7. (a-1) shows the results when Machine B and the router are on a wired LAN, and (a-2) shows the results when the router is on a wireless LAN. If we only study the mean or median latency in the wired LAN tests in (a-1), latency was lowest for CORBA and highest for AMQP for all payload sizes. However, the difference is small, ranging from 0.2 to 0.3 ms. MQTT has less latency than AMQP, but the mean is drawn to the higher outlier, indicating a slight deviation from the median. In contrast, the latency of the AMQP communication interface is the highest among the three communication interfaces, but its mean and median values are relatively close, indicating that stable communication is possible. The quartile range is also relatively narrow and coherent, and there is no sudden instability and widening of the quartile range owing to changes in the payload size.

In contrast, the wireless LAN test in (a-2) is conducted on an unstable network, and the effect of the instability is evidenced in the results. Although the median value does not show a significant difference between the communication interfaces, the average value shows a large difference between CORBA and the other communication interfaces. There is a difference of approximately 30 ms, depending on the payload size. In the results for CORBA, a large upward deviation of the mean from the median is due to a frequent occurrence of sudden high latency conditions compared to MQTT and AMQP. The interquartile range also shows that CORBA has a wider range of values around the median than the other two interfaces, confirming its instability. In comparison, the mean values of MQTT and AMQP show some tendency to be drawn by outliers, but the median and mean values do not deviate as much as those of CORBA, indicating that relatively stable communication has been achieved. In particular, AMQP has a narrower quartile range than MQTT, and the difference between the median and mean values is narrower in the 64- and 128-bytes results, making AMQP more stable than MQTT.

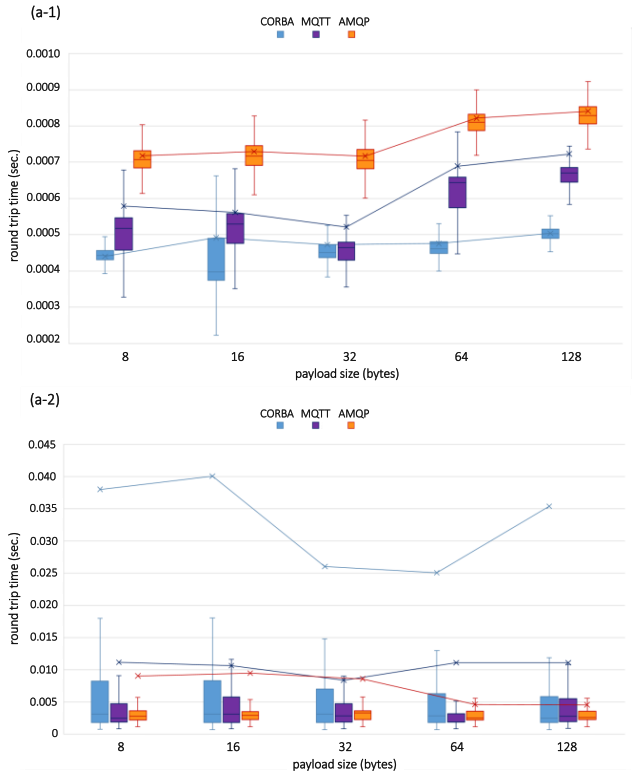


FIGURE 7. **(a-1) Round trip time with wired network environment. (a-2) Round trip time with wireless network environment.**

As described above, the real-time performance tests (a-1) and (a-2) yield extremely different results. The results are significantly influenced by the network environment. (a-1) is tested on a stable network built only with wired LANs, and the reason why MQTT can achieve lower latency than can AMQP is that a small payload is used. We can assume that the small size of the header in the message caused the result. In contrast, we consider that the main reason why CORBA has the lowest latency is the difference in the messaging paradigm. CORBA differs from the other two communication protocols because it uses a client/server communication model, with no medium between the client and server. CORBA also adopts a broker architecture using an object request broker (ORB). However, ORB is also an object and is embedded in each process of the client or server at the endpoint. It is important to note that the system architecture is very different from that of a pub/sub messaging broker, which runs in a separate process, regardless of whether it is inside or outside the endpoint.

In contrast, in the test on the unstable network built with the wireless LAN in (a-2), the cause of more sudden delays may be expected in the CORBA specification or its implementation. CORBA uses Internet inter-ORB protocol [27] as the communication protocol between ORBs in TCP/IP networks, but the protocol is not designed for IoT and may not be sufficiently optimized for communication in unstable networks. By contrast, MQTT and AMQP are protocols that are developed for system development on the Internet, and it is assumed that the plots show the optimization results for systems on wireless networks. Among them, the Nagle algorithm is a possible reason why AMQP, which has a higher communication overhead, appears more stable and, in some cases, has less latency than MQTT. The Nagle algorithm is used to reduce the number of packets in a TCP/IP network to improve throughput and is used for congestion control. However, since it sacrifices latency for efficiency, it should not be used in systems, where real-time performance is required. When we assess the configuration of Mosquitto and RabbitMQ, we see that the Nagle algorithm is turned on for Mosquitto and off for RabbitMQ by default. It means that RabbitMQ is better at reducing latency when considering only the Nagle algorithm setting. In the case of the default setting of Mosquitto, it can be inferred that the Nagle algorithm works as a congestion control in unstable network environments and override the original low latency communication. This means that we cannot simply conclude that the communication performance of MQTT and AMQP may be reversed when communicating over wireless networks using small payloads.

Next, we show the results of the messaging quality test (b). To compare and verify the results among the communication interfaces, a clear difference is observed when the buffer size of the data port was zero, and only this result is shown in Figure 8. They are shown in bar plots as the number of errors in the order of message arrival and the number of missing messages at each tested communication interface for each payload size. For the simplicity, they are called as the number of error occurrences in the following.

As shown in the figure, in the test with a buffer size of 0, CORBA has the highest number of errors for any payload size, followed by MQTT and AMQP. The message missing rate for the test with CORBA is approximately 2.2%. Although MQTT, the runner-up, does not have the same poor messaging quality as CORBA, its messaging quality is not improved, even when the QoS is set to a higher number, and in some cases, the number of errors increases with a decrease in QoS value. Meanwhile, AMQP has the lowest number of errors among the three communication interfaces. Although it does not perform transaction control, both message arrival order errors and missing messages rarely occur, and the message missing rate is close to zero. These results show a similar trend for all payload sizes, and although we only tested with small payloads, we do not find any relationship between payload size and the number of errors.

We consider that the reason for the significant occurrence of errors in CORBA at any payload size is the client/server-type communication architecture. In the tests, to perform one-to-four communication, one out-port of the client is connected to each of the four in-ports of the server in the CORBA communication interface, and data transmission and reception are established. It means that the workload on the client side is four times higher than usual, which can be interpreted as simply reflecting the increased load on the out-port. In contrast, in pub/sub messaging with a broker such as

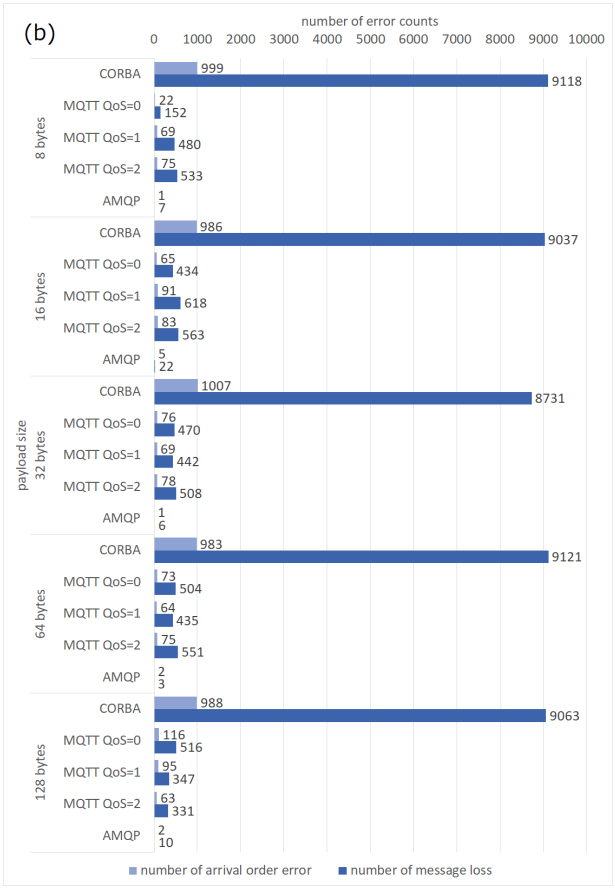


FIGURE 8. Number of arrival order errors and the number of messages lost in wireless and heavily loaded environment.

MQTT and AMQP, the out-port, as the client, only needs to connect to a single broker, as the server. In addition, since the broker is deployed in the same local environment as the RTC with the out-port, messages delivered from the out-port are almost certain to reach the broker, and it is unlikely that the publisher (producer) is the cause of the error. Therefore, we can assume that either the processing performed by the broker or subscriber (consumer) is the bottleneck.

In MQTT, when QoS is set to 0, there is no limit to the number of messages a broker can process simultaneously, so it can be assumed that messages can be sent continuously. However, they are lost if the subscriber cannot fully process them. When QoS is set to 1 or 2, the broker waits for the other party's response when communicating, and in the default state, the maximum number of messages that can be processed simultaneously without waiting for a response is set to 20. Once this limit is reached, other messages are stored in the queue. When the limit is exceeded, the oldest messages are discarded. It indicates that when QoS is set to 1 and 2, the processing by the broker introduces some bottlenecks.

By contrast, AMQP processes messages one by one in a FIFO format in each queue allocated to each consumer, so the order is guaranteed. However, if there is a deadlock in the processing on the consumer side, there should be some message loss. The count in the arrival order error is directly caused by the skipped order due to missing messages.

In the above test results, when the buffer size of the data port was set to 0, a large number of errors related to messaging quality are observed in each communication interface of CORBA and MQTT. For CORBA, although the number of error occurrences decreases, even when the buffer size is set to 100, this trend continues and some missing messages are observed. For both MQTT and AMQP communication interfaces, except CORBA, setting the buffer size to 100 results in zero error occurrences. Furthermore, by setting the buffer size to 10,000, the results show that the error occurrence is zero for all communication interfaces, including CORBA. Therefore, we can conclude that ensuring sufficient buffer size for asynchronous communication at the data port of an RTC improves reliability, regardless of the type of communication interface. In addition, the test results show that the AMQP communication interface can sufficiently maintain high-quality messaging under unstable and high-load network conditions without relying on RT-Middleware's buffer mechanism.

VII. CONCLUSION AND FUTURE WORK

The objective of this study was to develop a reliable and robust IoRT system. We have discussed the state of communication in IoRT systems and have proposed a communication model and system configuration method for realizing it.

As the concrete implementation, we have introduced AMQP, a pub/sub messaging protocol with a broker, because AMQP can ensure the arrival and order of messages as well as AMQP can create a variety of message flows.

In this study, we have chosen RT-Middleware for expanding the AMQP communication interface, because RT-Middleware allows network components to have multiple types of communication interfaces, meanwhile it has been difficult with other IoT platforms, along with robot middlewares such as ROS or ROS2. By combining the AMQP communication interface with the RabbitMQ server, it has become possible to construct a more reliable and robust IoRT system using only RT-Middleware.

We have made the performance analysis of the implementation, and have shown that it can maintain reliable messaging, even in a high-load wireless network environment, at least when sending and receiving data with small payloads. On the other hand, we have also found that maintaining real-time performance is difficult, especially in a wired LAN environment.

Our implementation presented in this study has been released in the GitHub repository[2] as public. In the future, we would like to build and present a reference guide of IoRT systems and promote the AMQP communication interfaces in RT-Middleware as a platform for constructing highly reliable IoRT systems. We are greatly honored if this study helps you develop IoRT systems.

[2]https://github.com/dyubicuoa/OpenRTM_aist_amqpcpp_interface

REFERENCES

[1] P. P. Ray, "Internet of robotic things: Concept, technologies, and challenges," *IEEE Access*, vol. 4, pp. 9489–9500, 2016.

[2] *MQTT Official Web Site*. Accessed: Apr. 12, 2021. [Online]. Available: http://mqtt.org/

[3] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting IoT platform requirements with open pub/sub solutions," *Ann. Telecommun.*, vol. 72, no. 1, pp. 41–52, Feb. 2017.

[4] D. Bezerra, R. R. Aschoff, G. Szabo, and D. Sadok, "An IoT protocol evaluation in a smart factory environment," in *Proc. Latin Amer. Robot. Symp., Brazilian Symp. Robot. (SBR) Workshop Robot. Educ. (WRE)*, Nov. 2018, pp. 118–123.

[5] A. Chaudhary, S. K. Peddoju, and K. Kadarla, "Study of Internet-of-Things messaging protocols used for exchanging data with external sources," in *Proc. IEEE 14th Int. Conf. Mobile Ad Hoc Sensor Syst. (MASS)*, Oct. 2017, pp. 666–671.

[6] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Comput.*, vol. 10, no. 6, pp. 87–89, Nov. 2006.

[7] AMQP Working Group. *AMQP Version 0.9.1 Protocol Specification*. Accessed: Apr. 12, 2021. [Online]. Available: http://www.amqp.org/sites/amqp.org/files/amqp0-9-1.zip

[8] AMQP Working Group. *Products and Success Strories*. Accessed: Nov. 11, 2021. [Online]. Available: https://www.amqp.org/about/examples

[9] Organization for the Advancement of Structured Information Standards (OASIS). *MQTT Version 3.1.1 Specification*. Accessed: Apr. 12, 2021. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[10] N. Q. Uy and V. H. Nam, "A comparison of AMQP and MQTT protocols for Internet of Things," in *Proc. 6th NAFOSTED Conf. Inf. Comput. Sci. (NICS)*, Dec. 2019, pp. 292–297.

[11] G. Andrei, B. Marlen, T. Sergey, and K. Krinkin, "Industrial messaging middleware: Standards and performance evaluation," in *Proc. IEEE 14th Int. Conf. Appl. Inf. Commun. Technol. (AICT)*, Oct. 2020, pp. 1–6.

[12] Object Management Group (OMG). *Data Distribution Service Specification Version 1.4*. Accessed: Jul. 20, 2021. [Online]. Available: https://www.omg.org/spec/DDS/1.4/About-DDS/

[13] T. Moraes, B. Nogueira, V. Lira, and E. Tavares, "Performance comparison of IoT communication protocols," in *Proc. IEEE Int. Conf. Syst., Man Cybern. (SMC)*, Oct. 2019, pp. 3249–3254.

[14] D. Lu, Z. Li, D. Huang, X. Lu, Y. Deng, A. Chowdhary, and B. Li, "VC-Bots: A vehicular cloud computing testbed with mobile robots," in *Proc. 1st Int. Workshop Internet Vehicles Vehicles Internet (IoV-VoI)*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 31–36.

[15] Pivotal Software. *RabbitMQ Official Web Site*. Accessed: Apr. 12, 2021. [Online]. Available: https://www.rabbitmq.com/

[16] L. Killian, M. Julien, B. Kevin, L. Maxime, B. Carolina, C. Mélanie, B. Nathalie, G. Sylvain, and G. Sebastien, "Fall prevention and detection in smart Homes using monocular cameras and an interactive social robot," in *Proc. Conf. Inf. Technol. Social Good (GoodIT)*. New York, NY, USA: Association for Computing Machinery, Sep. 2021, pp. 7–12.

[17] J. O'Hara, "Toward a commodity enterprise middleware," *Queue*, vol. 5, no. 4, pp. 48–55, May 2007.

[18] J. Lim, G. S. Tewolde, J. Kwon, and S. Choi, "Design and implementation of a network robotic framework using a smartphone-based platform," *IEEE Access*, vol. 7, pp. 59357–59368, 2019.

[19] Amazon Web Services. *AWS IoT Core*. Accessed: Jul. 20, 2021. [Online]. Available: https://aws.amazon.com/iot-core/

[20] Google LLC. *Cloud IoT Core*. Accessed: Jul. 20, 2021. [Online]. Available: https://cloud.google.com/iot-core

[21] International Business Machines Corporation (IBM). *Watson IoT Platform*. Accessed: Jul. 20, 2021. [Online]. Available: https://www.ibm.com/cloud/watson-iot-platform

[22] Microsoft Corporation. *Azure IoT Hub*. Accessed: Jul. 20, 2021. [Online]. Available: https://azure.microsoft.com/en-us/services/iot-hub/

[23] Open Robotics. *ROS Official Web Site*. Accessed: Jul. 20, 2021. [Online]. Available: https://www.ros.org/

[24] Open Robotics. *ROS2 Documentation*. Accessed: Jul. 20, 2021. [Online]. Available: https://docs.ros.org/en/foxy/index.html

[25] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-Middleware: Distributed component middleware for RT (robot technology)," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Aug. 2005, pp. 3555–3560.

[26] Object Management Group (OMG). *Robotic Technology Component Specification Version 1.1, Formal/12-09-01*. Accessed: Apr. 12, 2021. [Online]. Available: http://www.omg.org/spec/RTC

[27] Object Management Group (OMG). *Common Object Request Broker Architecture*. Accessed: Apr. 12, 2021. [Online]. Available: https://www.omg.org/spec/CORBA/About-CORBA/

[28] National Institute of Advanced Industrial Science and Technology (AIST). *OpenRTM-AIST Official Web Site*. Accessed: Apr. 12, 2021. [Online]. Available: https://www.openrtm.org/

[29] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-AIST," in *Proc. Int. Conf. Simulation, Modeling Program. Auton. Robots (SIMPAR)*, Lecture Notes in Computer Science, vol. 5325, 2008, pp. 87–98.

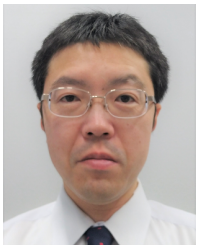
[30] B. V. Coperica. *C++ Library for Asynchronous Non-Blocking Communication With RabbitMQ*. Accessed: Apr. 12, 2021. [Online]. Available: https://github.com/CopernicaMarketingSoftware/AMQP-CPP

[31] D. Yoshino, Y. Watanobe, Y. Yaguchi, K. Nakamura, J. Ogawa, and K. Naruse, "Publish/subscribe messaging interface using bridges among message brokers on RT middleware," presented at the Soc. Instrum. Control Eng. (SICE SI), Sendai, Japan, 2017.

[32] Eclipse Foundation. *C++ Library for MQTT Client With Mosquitto MQTT Broker*. Accessed: Apr. 12, 2021. [Online]. Available: https://github.com/eclipse/mosquitto/tree/master/lib/cpp

[33] Eclipse Foundation. *Eclipse Mosquitto Offical Web Site*. Accessed: Apr. 12, 2021. [Online]. Available: https://mosquitto.org/

[34] G. Biggs and N. Ando, "A formal specification of the RT-middleware data transfer protocol," in *Proc. IEEE Int. Conf. Simulation, Modeling, Program. Auton. Robots (SIMPAR)*, Dec. 2016, pp. 302–309.



DAISHI YOSHINO received the M.S. degree from The University of Aizu, Japan, in 2001. He took a position as a Special Researcher for a Renewable Energy Related Project, from 2012 to 2016 and has been working as a Special Researcher for a Roboticsrelated Business Project, since 2016. His research interests include middleware, frameworks, and communication protocols that are necessary for constructing systems on the internet, including robots. He is currently a Project Researcher with the School of Computer Science and Engineering, The University of Aizu.

YUTAKA WATANOBE (Member, IEEE) received the M.S. and Ph.D. degrees from The University of Aizu, Japan, in 2004 and 2007, respectively. He was a Research Fellow of the Japan Society for the Promotion of Science at The University of Aizu, in 2007, where he is currently a Senior Associate Professor with the School of Computer Science and Engineering. His research interests include data mining, smart learning, cloud robotics, and visual languages.

KEITARO NARUSE is currently a Full Professor with the School of Computer Science and Engineering, The University of Aizu, Japan. He is also working for design, development, and standardization of networked distributed intelligent robot systems with heterogeneous robots and sensors. He has applied the networked distributed intelligence to service robot systems, factory automation systems, and intelligent disaster respond robot system. His research interests include swarm robots and their application to agricultural robotic systems, and interface system for disaster response robots.