Hindawi
Journal of Computer Networks and Communications
Volume 2022, Article ID 7368691, 18 pages
https://doi.org/10.1155/2022/7368691

Research Article

Using IoT Protocols in Real-Time Systems: Protocol Analysis and Evaluation of Data Transmission Characteristics

Andrei Gavrilov,[1] Marlen Bergaliyev,[1] Sergey Tinyakov,[1] Kirill Krinkin,[1] and Pavel Popov[2]

[1]*Department of Software Engineering and Computer Applications, Saint Petersburg Electrotechnical University "LETI", Saint Petersburg, Russia*
[2]*St. Petersburg Branch JSC NIIAS, Saint Petersburg, Russia*

Correspondence should be addressed to Andrei Gavrilov; gavrilov.andrew1999@yandex.ru and Sergey Tinyakov; mirrin1905@outlook.com

Received 24 January 2022; Accepted 25 July 2022; Published 16 August 2022

Academic Editor: Bilal Khalid

Copyright © 2022 Andrei Gavrilov et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

In the Internet of things, many data transfer protocols are used for various tasks. In this article, we consider the application layer protocols that are the main ones for transmitting messages in the IoT. The main problems are unpredictability, lack of stability of data transmission delays, and non-determinism, which are also important for real-time systems. The purpose of this study is to determine the most appropriate middleware and data transfer protocol for systems with high data transfer requirements, including real-time systems. Therefore, MQTT, RTPS, JMS, and AMQP protocols were analyzed in order to find out what tasks these protocols should be used for and whether they can be used in robotic and autonomous systems where high data transmission requirements are imposed. To evaluate the protocols, the standards were analyzed to determine the pros and cons, and the software implementations of each of them were selected. To assess the characteristics of data transmission, we have developed our own test scenarios that simulate complex situations. The behavior of software solutions is analyzed and a comparative analysis is made based on the obtained data. Together, the theoretical analysis of protocols and the study of software solutions allow us to conclude on the applicability of a particular protocol in real-time systems. As a result of the study, we can conclude that RTPS is the best solution for real-time systems with different traffic and MQTT performs well when transmitting short messages. But none of the protocols under consideration guarantees the determinism of data transmission, so it is better to use specialized link-layer protocols to obtain guarantees.

1. Introduction

In modern systems, the volume of data processed, generated, and transmitted is increasing. Many complex systems consist of many modules or nodes that constantly communicate with each other. These modules can be located on a single computer or distributed in a network. The more automated the system, the greater the requirements for data transmission and real-time communication. In distributed computing systems, transmission affects the performance of the system, while in various robotic systems, constant synchronization may be required and low data exchange rates may lead to errors in the operation or breakage of mechanisms. While in autonomous systems such as self-driving vehicles, unpredictable and long delays can become a threat to people's lives. In this regard, the question arises about the choice of data transmission protocols for reliable and fast data transmission.

1.1. Motivation. Various network technologies are used everywhere in everyday life. Systems are getting smarter and there is always a problem of limited resources. Millions of sensors continuously create and transmit data to manage

real-world infrastructures that use complex Internet of things (IoT) networks. For various spheres of human activity, various technologies and protocols are being developed for the organization of the network and the efficient operation of the infrastructure. The predominant areas of development of such network technologies are smart environment, autonomous transport, and medicine.

1.1.1. Smart Environment. A smart environment includes the automation of various processes of our everyday life. It includes such technologies as smart home, smart city, smart manufacturing, etc. In a smart environment, the question of limited resources also arises, since smart devices and various sensors are also used to create this environment. The resources of IoT devices are very limited, and there is no possibility to perform complex computing tasks locally. At the same time, the bandwidth of data transmission channels is also limited and it is impossible to use the entire resource for the functioning of systems. To solve these problems, various approaches to building infrastructure related to edge computing are being created [1]. At the same time, the basic data transfer technologies are the IoT protocols, and they play a significant role in the operation of a particular smart environment system.

1.1.2. Autonomous Transport. Autonomous transport includes a set of units like sensors, GPU, CPU actuators, etc. The operation of an onboard control system is based on the transmission of information between subsystems. Therefore, it is crucial to realize a reliable exchange of data. The main parameters are a guarantee of data delivery, data integrity, maximum possible latency of data transmitting, and calculation cost. Furthermore, it is vital to balance requirements considering different kinds of data: vast streams of information from sensors, short commands from CPUs, etc. Therefore, the data transfer protocol has to provide the best possible parameters for distributed and complicated autonomous control systems.

1.1.3. Medicine. Modern healthcare involves many personal wearable sensors that constantly monitor a person's condition. There are different views on what the architecture of next-generation medical platforms should be. Researchers agree that the integration of personal sensors into a unified health information system is inevitable. This implies effective ways to communicate in a heterogeneous environment. The communication methods discussed in this article will be useful to those who design medical systems.

2. Literature Overview

This section will present the work done in the field of this study as well as a description of each of the protocols under consideration. After considering the theoretical part of the protocols, the main questions that this study is trying to answer are put forward.

2.1. Related Works. A large number of studies and works have been conducted on the problem of choosing a protocol for real-time systems. Various IoT protocols are analyzed for performance, security, and reliability. Since AMQP and MQTT protocols are popular in the IoT, it is important to understand the security vulnerabilities associated with the each protocol. In [2–5], an analysis of vulnerabilities and cyber threats for these protocols was made. Also, in [6], an analysis of AMQP protocol for industrial IoT was done. For MQTT in [7], a comparison with CoAP on Ponte Eclipse Project implementation was provided. Also, an analysis of MQTT performance was done in [8]. There is a formal approach to model, analyze, and verify the usage of MQTT in the case of communicating vehicles in [9]. The dependence of delay and QoS level in MQTT was analyzed in [10]. An analysis and a comparison of AMQP, MQTT, CoAP, and HTTP protocols without any implementation were done in [11]. Since AMQP uses queues, a detailed analysis of message queue methods and a comparison of RabbitMQ, ActiveMQ, and Kafka implementations were made in [12]. Also, RabbitMQ, which is an implementation of AMQP, was analyzed as message-oriented middleware (MOM) and compared to NATS and Kafka in [13]. An analysis of OPC UA, DDS, and MQTT protocols for Industry 4.0 was done in [14]. More details about DDS and data-centric communication were given in [15]. The DDS standard was designed for real-time systems on mission-critical infrastructures. So, security is necessary to avoid disaster and loss of life. Security issues for DDS were analyzed in [16, 17]. Touching on our previous research, we have analyzed the existing data transmission standards in the IoT [18] and the performance of DDS services [19].

2.2. Protocol Overview. In this section, we will briefly consider the main concepts of the protocols and highlight the basic features that are stated in the specifications. Each protocol was created to solve a certain range of tasks, so it is worth getting acquainted with each of them to understand the relevance of using the protocol in the modern IoT.

In IoT, the most relevant protocols are MQTT, AMQP, JMS, XMPP, CoAP, and RTPS (DDSI-RTPS). XMPP does not support any delivery guarantees, but only allows to request information from the client about the delivery of data [20], so this protocol is not considered in this article, because the real-time system requires these delivery guarantees to work with up-to-date data. Also, the CoAP protocol is not discussed in this article; although the addition of the publish-subscribe model to this protocol is interesting for real-time systems, the standard client-server model is not suitable for real-time data transmissionbecause this model requires sending requests to receive new data. These protocols solve the problem of data exchange in any format, queue processing, and data distribution. Each of the considered protocols provides different guarantees, so it is necessary to understand for which target systems a particular protocol may be suitable or for which data.

2.2.1. Message Queuing Telemetry Transport (MQTT). MQTT (Message Queuing Telemetry Transport) is a lightweight transport protocol that releases a publish-subscribe

model. The protocol is an open OASIS standard and an ISO recommended.

MQTT was invented by Dr Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom (now Eurotech), in 1999. There was a problem with communication between monitoring devices and remote servers in the oil and gas industry. In many cases, it was difficult or impossible to use a landline, wired connection, or radio transmission connection. During that time, satellite communication was the only solution to this problem. But it was very expensive, and it was necessary to pay according to the volume of data transmitted. So, thousands of sensors required a form of communication that could provide reliable data transmission with minimal bandwidth use.

Two developers specified the features of the tool to solve this problem:

(i) Simple implementation
(ii) Lightweight and bandwidth efficient
(iii) Quality of service
(iv) Data agnostic
(v) Continuous session awareness

IBM used protocol after it was created for internal purposes for about 10 years. Then IBM released MQTT 3.1 as a free version in 2010. From that time on, everyone was able to use the protocol. In 2014, MQTT became an officially approved OASIS standard.

In 2019, OASIS ratified the new MQTT 5 specification. The new version specified features that are required in the IoT industry, such as more reliability and error handling.

It works on top of any ordered, lossless, bidirectional network protocol, for example TCP or SSL. In MQTT, there are two entities: the server (broker) and the clients. All messages from senders who are called publishers are passed by the broker to receivers who are called subscribers. All data are divided into topics—the label attached to the message. The server matches the topic with subscription and forwards messages if they match. This is how clients behave:

(i) Connect to the broker
(ii) Subscribe to a topic and wait for incoming messages or send messages to the topic(s)
(iii) Close the connection

MQTT provides topics with wildcard, shared subscriptions and quality of service.

There is also a variation of the protocol—MQTT-SN (MQTT for sensor networks). It is designed especially for sensor networks. MQTT-SN can work on top of non-TCP/IP network protocols such as UDP. Header size has also been reduced, and the UTF-8 topic string has been replaced by an integer topic id.

Quality of services defines the delivery guarantee. There are three levels:

(i) QoS 0: At-most-once delivery. Message is sent only one time. The receiver sends no response and the sender performs no retry.
(ii) QoS 1: At least once delivery. Message is sent until it arrives at the subscriber at least once. The subscriber sends an acknowledgment to the publisher after getting the message. The publisher performs a retry until an acknowledgment is received. The subscriber processes any copy of the message as a new unique message.
(iii) QoS 2: Exactly one delivery. The subscriber receives only one copy of the message by a two-step acknowledgment process. Like in QoS 1, the subscriber sends the acknowledgment, but the publisher sends a second acknowledgment back, and after getting it, the subscriber sends back a final acknowledgment that completes message transmission. During the acknowledgment exchange, the receiver drops any message that has the same packet identifier as the current message.

As described earlier, the topic is a label attached to the message. Topic names can form a hierarchical topic tree by separating each level with a forward slash. Subscribers use a topic filter to let the broker know which topics are of interest to them. Wildcards can be used in the topic filter. The number sign is a multilevel wildcard that matches any number of topic levels, including zero topics. The plus sign is a single-level wildcard that matches to only one topic level, including zero topics.

MQTT defines two types of subscriptions: shared and non-shared. The main difference is the number of copies of the message. With the non-shared subscription, all receivers who subscribe to the same topic get their own copy of the message. With the shared subscription, only one copy of the message is sent to the one shared group. That way, a load-balanced network architecture can be created. Shared subscriptions are also useful for parallel processing of publications by multiple clients.

2.2.2. Real-Time Publish-Subscribe Protocol (RTPS). The RTPS data exchange protocol, also known as DDSI-RTPS, is developed by the Object Management Group (OMG), as part of the DDS standard, responsible for data transfer and ensuring compatibility between different vendors as well as ensuring platform independence of the DDS service. The first version of the DDS specification was released in 2004 and the RTPS protocol specification version in 2009.

DDSI-RTPS protocol is developed for reliable and high-speed transmission via unreliable data transfer protocols, for example via UDP.

DDSI-RTPS is based on the concept described in the DDS standard. The model describes entities such as:

(i) Domain
(ii) Topic
(iii) Publisher who sends data to the topic via the DataWriter
(iv) Subscriber who gets data from the DataReader

The publisher and subscriber are the end nodes who sends and receives data, respectively, and the subject is an

entity that describes the channel to which data are sent, which subscribers can track. A domain is an entity that divides topics, publishers, and subscribers into different name domains. Therefore, the topic, publisher, and subscriber belong to a specific domain and cannot interact with nodes from another domain. At the same time, subscribers and publishers do not know anything about each other, but are connected only by the topic through which the data are transmitted.

The RTPS protocol is based on four main modules:

(i) A structural module that describes the structure of entities;

(ii) A message module that describes the structure of transmitted messages;

(iii) A behavior module that describes the sequence of message transmission and the time constraints imposed on the transmission of each message;

(iv) A discovery module that describes how nodes find each other to interact with.

To ensure reliable data delivery, the protocol describes an entity such as CacheChanges. Each node (DataReader or DataWriter) has such an entity. When sending and receiving data, each node makes a note about this in the Cache Changes.

The protocol has three message types: DATA, HEARTBEAT, and ACKNACK. DATA is a message with data, HEARTBEAT is a message with changes in CacheChanges of DataWriter, and ACKNACK is a message with changes in CacheChanges of DataReader. With the help of HEARTBEAT and ACKNACK, it is possible to determine what data were sent but not delivered, which will allow resending these data. But since the DataWriter only knows about data loss when it receives changes from the DataReader, which in turn are sent only when the message is received, the packet loss will only be known when the DataReader receives the next message, which generally violates the sequence of receiving messages. To ensure that the sequence is correct, the protocol makes messages unavailable until all the necessary data are received.

The discovery module contains two protocols that implement the search for nodes in the network. The first protocol is the Participant Discovery Protocol (PDP), which describes the algorithm for searching for nodes in a network belonging to a specific domain, and the second protocol is the Endpoint Discovery Protocol (EDP), which describes the search for specific DataReader and DataWriter in a domain.

Thus, we can distinguish the following qualities of this protocol:

(i) Providing automatic search by nodes of each other;

(ii) Providing data transmission with various QoS that the DDS standard describes;

(iii) High data transfer rate due to the UDP protocol, instead of TCP.

2.2.3. Java Message Service (JMS). A new programming language Java was introduced in 1995. As there was not a mechanism that provided communication between a few programs, JMS (Java Message Service) was created in 1998 to solve this problem. The last version (2.0a) of the JMS standard was updated in 2015.

JMS is similar to other middleware. It has two types of communication: point to point and publish and subscribe. In both communication styles, there is a JMS provider that is used to manage connections, queues, and resources. Opening the first connection creates a provider outside Java virtual machine.

Point-to-point messaging means that there are two processes communicating with each other. This communication type uses message queues. Each process has a message queue as a "mailbox." Queues accept all types of messages and are not individual for every message sender, so there is no need in having more than one queue on each side. Each queue can be used by a few clients, but they will not receive all messages in case after delivery every message becomes consumed and cannot be retrieved by any client. In JMS, there are two types of queues: temporary and static. Temporary queues can be used only in one connection, because they exist only during the lifetime of the connection. Static queues are most commonly used. They can be used in a few connections and also they may still exist after the client program's end, so while restarting, the client can retrieve messages from this queue.

Publish and subscribe messaging is a common publish-subscribe model, where few processes communicate through a topic. Reliability depends on the subscriber's durability and publishing persistence. Nonpersistent publishing results in at-most-once reliability, while persistent publishing results in once-and-only-once reliability. Nondurable subscribers can miss messages if they are inactive, while durable subscribers cannot miss messages. Persistent publishing is mostly implemented by static queues. There is no description of how redelivery must be implemented.

As additional functionalities, there is a message selector. It allows users to filter messages by processing their header fields. It uses SQL syntax to do that. Also, some providers allow the use of topic hierarchy, which helps to send a message to a few topics immediately.

2.2.4. Advanced Message Queuing Protocol (AMQP). AMQP was developed by John O'Hara in 2003. One of the first implementations of this protocol was Apache Qpid. Red Hat started developing it in 2005. In 2007, Rabbit Technologies released their implementation of AMQP called RabbitMQ.

AMQP defines three types of nodes: producers, consumers, and queues. Producers and consumers are parts of the application that generate and proceed messages. Each node is attached to a container which can be a client or broker. The difference between clients and brokers is only in expected capabilities.

In AMQP to start communication firstly required to create a connection between containers. Then the session can be started. Every connection may contain multiple sessions. Thereafter, a link can be established between two nodes. All links are unidirectional. On each side of the link, there is a terminus that controls message flow through the link.

Messages can be transmitted only through a link. In relation to nodes, messages can be divided into a few types, namely originate from, terminate at, or relay by nodes. Queues are used to store messages and make them available to few consumers. So, the main purpose of brokers is to manage queues to receive messages from producers, store them, and transmit messages to consumers.

Every message delivery has a settlement state, which controls if a message has been delivered, and delivery state. To set settlement, nodes send frames. When the receiver sends a settlement frame of delivery, the sender removes delivery from its unsettled map. It can be used to implement different reliability policies. If the sender sets the settlement of delivery after message sending, then the message will be delivered "at most once." If the receiver sends a settlement of delivery and the sender waits for settlement, then it implements the "at least once" guarantee. To perform "exactly once," the delivery sender has to set settlement only after the receiver sets the terminal delivery state, and the receiver has to set settlements after the sender. But in this case, a few frames should be sent: at least one for termination and one for settlement. There is no other way to implement reliable transfer without duplications in AMQP.

Additionally, AMQP provides security protocol support, such as TLS and SASL. Also, it has a transactional messaging model. Transactional works consist of three operations: posting, acquiring, and retiring. Target performs posting and makes a message unavailable at the destination until it will be fully discharged. Source performs retirement to associate an outcome of delivery with the transactionSource acquires transaction message to perform operation.

2.3. Conclusion of the Overview. A lot of work has been done in the field of IoT protocol research. Different studies investigate different characteristics of data transmission and use different methodologies. Each protocol has many implementations that can be better or worse relative to each other, and the research done can also use various protocol implementations for analysis. In addition, it is necessary to consider all protocols under the same conditions in order to achieve objective comparison results. Depending on the methodology, selected software implementations, and the range of protocols under consideration, the results of the study may vary.

All these factors complicate the choice of a solution and protocol for a specific task. This article examines the best software implementations based on the work done and creates the same conditions for evaluating middleware.

3. Materials and Methods

To analyze and compare the protocols selected for consideration, it is necessary to formulate criteria that will show their characteristics, select software implementations that use the protocols in data transmission, and develop a testing methodology that would allow studying these technologies from different angles.

In this section, we will consider the methods developed in our work for studying software solutions for data transmission that emit various conditions. Such conditions can occur in different systems, so it is necessary to evaluate not only the maximum capabilities of these software tools, but also their behavior in critical situations that can be created by developers during the development of the system. We will also define the criteria for selecting specific data transfer libraries for each protocol.

3.1. Description of Comparison Methods and Testing Methods. Every protocol has its own reason for creation, which lies in its feature. It has both pros and cons. It is difficult to predict the performance of protocol based on analysis only. The performance also depends not least on the implementation of the protocol. In addition, machines have different power, which also affects performance. It is necessary to know data transmission characteristics for objective evaluation and comparison. Comparison of different protocols can be difficult because they work differently and are designed for different purposes. Therefore, it is good to have a universal value characteristic that shows the practical side of protocols. This makes it possible to compare and select implementations without knowing their internal structure.

Test scenarios have been developed to evaluate different implementations. These tests simulate situations with different environments: message interval, number of subscribers or exchanging pairs, frequency, and message size. The limits in the tests can be viewed as hard real-time situations. All this gives an insight into the limitations and problems of the protocol and/or the implementation. This helps to decide if the protocol is a good solution for a certain problem.

For each selected implementation, corresponding programs were written that have identical logic. The difference between the programs is only in the functions of sending and receiving data, which use the API of the selected middleware directly. The general logic of the tests has the following three types of interaction:

(i) One-way data transfer, when the program performs the role of either only the sender or only the receiver with a specified frequency and with a specified payload size;

(ii) Two-way data transfer, where one node sends messages only when it receives a response from the other, thus measuring RTT and jitter;

(iii) Two-way data transfer, when one node does not wait for a response, but sends messages at a certain interval, while it is possible to artificially limit the queue by indexing received and sent messages.

To obtain objective results of data processing by the selected technologies, programs are assigned to a separate core with exclusive use of its resources, which guarantees that the programs work in the same conditions.

The characteristics of the system that was tested are shown in Table 1. All tests were performed locally on a single computer. The test scenarios used in the study will be described below.

Table 1: System configuration.

OS	Linux-5.4.54-rt with Ubuntu 18.04 bionic
CPU	Intel Core i5, 3.0 GHz
RAM	8 GB

3.1.1. Evaluating Queue Processing Performance. The test is aimed at investigating message queue processing. In the test, the first node sends messages to the other node without any interval. Thus, the queue accumulates.

The test is performed with message sizes of 50 and 60,000 bytes. The process priority is set to the highest in Linux–99. Both the nodes are bound to the CPU cores. So, each process is separate and works without interruptions during queue processing, such conditions help to get the most accurate data about the studied characteristic.

3.1.2. Evaluating the Impact of the Number of Nodes on Performance. The test is aimed at investigating the dependence of the total latency on the number of subscribers. In the test, one node (publisher) sends messages to other nodes—subscribers. The message size with sending frequency takes up 1.25 MB per second of bandwidth.

The test is performed with the number of subscribers varying from 1 to 20. As in test 1, all processes have the highest priority in Linux–99. No node is bound to the CPU core. A separate test is run for each number of subscribers.

3.1.3. Evaluating the Impact of Message Size on Latency. The test is aimed at investigating the total latency in dependence on message size. In the test, the first node sends messages to the other node at a given interval. The message size increases during the test.

The test is performed on different frequencies which in combination with the message sizes take from 2.5 kB to 2 GB per second of bandwidth. As in previous tests, all processes have the highest priority in Linux–99. The nodes are not bound to CPU cores.

3.1.4. Estimation of the Jitter Value and Minimum Delays. The test is aimed at investigating the total latency, jitter, and RTT (round trip time) of each message of the minimum size. The test also shows the number of copies between the user space and the kernel space. In this scenario, the ping-pong model is used: one node sends a message to the other, and then both exchange messages only after receiving a message from the other node. The test stops when the set number of transmitted messages is reached. The size of each message is 10 bytes. Nodes are not bound to CPU cores and process priorities are not set.

3.1.5. Evaluation of the Effect of the Number of Interacting Processes in Different Conditions on Delays. The test is aimed at investigating the dependence of the total latency on the message size at different message sending frequencies and on the number of process pairs. In this scenario, the ping-pong model is used, but the node that sends the first message, also called the first node, does not wait to receive a message from the other node, as in the previous scenario, and sends new messages on a given frequency. The other node works as in the previous test. The test stops when the set number of transmitted messages is reached.

The test consists of two types of subtests: (1) one pair exchanges messages on a given frequency; (2) at a fixed frequency, several pairs exchange messages. The message size increases during the test in both subtests. In the first subtest, messages take up a bandwidth in the range of 5 kB–4 GB per second for both the nodes. In the second subtest, messages take up a bandwidth in the range of 100 kB–1.5 GB per second for each pair. The number of pairs in the second subtest is 1, 2, and 3. Nodes are not bound to CPU cores and process priorities are not set.

3.1.6. Investigation of Delays in Artificially Fixing the Message Queue. The test is aimed at investigating the total latency of processing a limited queue. This scenario uses the limited queue ping-pong model from test 5. If the first node has a difference between the number of the sent message and the last received one greater than the watermark value of 50, no message will be sent. It means that messages in the queue are not more than the value of the watermark. Due to the limited queue, the latency will be constant or does not exceed some constant value. The size of each message is 10 bytes. Nodes are not bound to CPU cores and process priorities are not set.

3.2. Description of the Framework Selection. The main criteria for framework selection are open source and C/C++/Java programming language for clients.

The JMS is very abstract, so this research could have been any Java client with or without any server. But in this article, there is an idea of comparing with Java realization, so the JMQ has been selected. The server is called GlassFish, and it was developed by Sun Microsystems and later passed under the wing of the Eclipse Foundation. The auto-acknowledge of message property is used for testing. The nonpersistent property is also set. If the persistent property is set, the message is copied to the hardware to avoid losing it.

There are many implementations of the DDS standard: OpenDDS, OpenSplice, FastRTPS, and Cyclone DDS. There are evaluations of FastRTPS on the eProsima website [21]. We also have another close research comparing all these four DDS frameworks. In it, FastRTPS was the winner. For example, in Figure 1, there is a delay graph for messages with increasing size (we use "Evaluating the impact of message size on latency" test scenario). The results of OpenSplice (the public version of this solution) were not added to the image, as this solution showed a terrible result with delays in seconds after 200 messages, where the size was 512 kB. As pictured, FastRTPS shows the best result. Therefore, FastRTPS has been selected for this research. For guarantee delivery, the following message properties are set: to achieve a delivery guarantee, parameters are set for storing all messages that have not yet been received by subscribers and a QoS-reliable policy is selected, which guarantees data delivery.

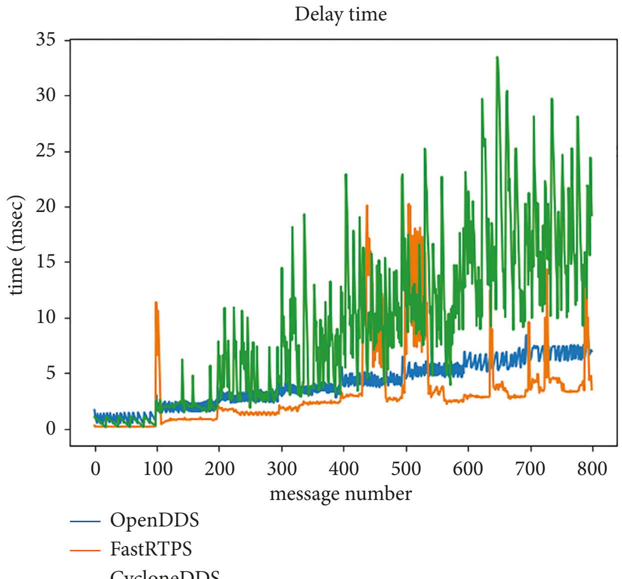

Figure 1: OpenDDS's, FastRTPS's, and Cyclone's delay in the case of increasing the payload size every 100 messages.

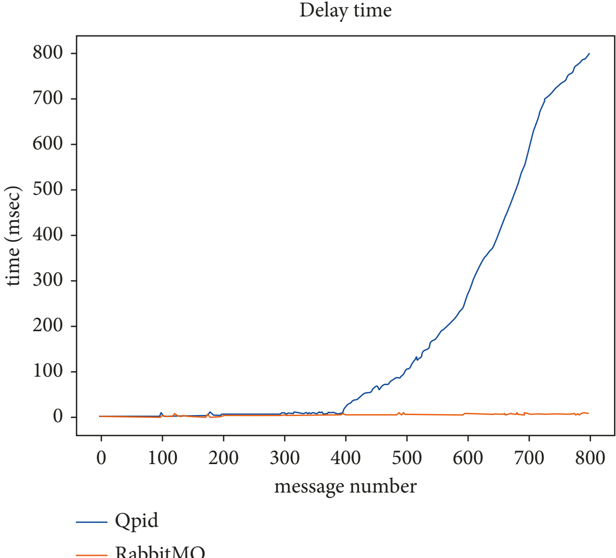

Figure 2: Qpid's and RabbitMQ's delay in the case of increasing the payload size every 100 messages.

There are not many implementations of AMQP. Three popular servers are RabbitMQ, ActiveMQ, and Apache Qpid. A comparison of the first two was made in [22], where RabbitMQ showed better results. In [23], a comparison was made between RabbitMQ and Kafka, where the former showed good results. The Apache Qpid has been tested in the first three tests. In Figure 2, there are results of RabbitMQ and Qpid for the test "Evaluating the impact of message size on latency." As pictured, Qpid is worse. In other tests, the Qpid result is close to or worse than RabbitMQ. Therefore, RabbitMQ has been selected for this research.

There are many MQTT servers since the IoT becomes more and more a part of our life. Evaluation, analysis, and comparison of MQTT brokers have been done in [24, 25]. In these researches, the best result was shown by the Mosquitto broker. Therefore, this broker is used in this research. PahoMQTT has been selected for the client side. It is developed by Eclipse Foundation. For the tests, the QoS level is set to 1–at least once delivery.

4. Results and Discussion

The analysis of protocols should be divided into two sections—theoretical and performance evaluation—since the analysis of specifications will not give information about the performance of data transmission, but only helps to understand the principles of operation and guarantees provided by the protocol. At the same time, protocol implementations from different vendors may show different characteristics, so a practical assessment of the performance of a particular solution does not give an accurate assessment of the protocol.

Later in the article, we will try to analyze what these protocols provide and consider the characteristics obtained as a result of testing.

4.1. Analysis of the Operation of the Protocols (Specification-Based Analysis). To analyze the protocols, we studied the specifications of these protocols, which are in the public domain. In this part of the study, the advantages and disadvantages of the algorithm of the protocols and the technologies used were identified.

4.1.1. MQTT. MQTT packets are lightweight. Each packet is divided into three sections: a fixed header, a variable header, and a payload. The fixed header consists of the packet type, flags, and remaining length. Summary of the first two gives 1 byte. The variable header consists of a 2-byte packet identifier and properties which have a variable length. Some packets have specific required properties. The payload is a user's message sent. Packets do not take up much space. Also, many properties are used in the packet for opening connection. Only one required property in publish packet can take up a lot of space—the topic name. The topic is a UTF-8 string. If needed, the topic can be replaced by, for example, the topic ID, like in MQTT-SN. But that would cause problems with wildcards in topics. Therefore, if they are not used, it is possible to reduce packet size by replacing the topic with something else.

As said before there is no queue in MQTT. So, it becomes a problem that the application may not be able to process a message. This is not unusual on the network with several thousand clients and thousand messages per second. QoS was created for solving this problem. But it also has other problems. With QoS 1, the subscriber spends time processing every message as a new one. With QoS 2, there are three acknowledgment messages where each size is about 6 bytes. So, every second, each of a thousand subscribers sends a thousand acknowledgments. It reduces the bandwidth by at least 6 MB. MQTT is good for real time when it is required to get actual data, but for data transfer with a delivery

guarantee, MQTT may be worse than other solutions based on message queues.

In addition, MQTT is centralized. So, it has a single point of failure which is the broker. This can be a problem in some cases.

The quality of service that defines guarantee delivery is described in the MQTT. Is a lossless network protocol like TCP required? TCP takes a lot of bandwidth. If the message is dropped by the server, for example, due to performance limitation, the message will be sent again, which will take up much bandwidth again. It may be better to use UDP or other UDP-based protocols, because they do not take much bandwidth. QoS levels 1 and 2 guarantee delivery, so the excess requirement of lossless to the network protocol is not necessary.

Advantages:

(i) Lightweight packets;
(ii) QoS levels;
(iii) Ability to filter messages.

Disadvantages:

(i) MQTT uses TCP/IP;
(ii) Centralized model;
(iii) No queues.

4.1.2. RTPS. The RTPS protocol has many advantages over other protocols. As described in the review, the protocol is positioned as a reliable data transfer protocol, using unreliable transport layer protocols-UDP/UDPM. Using these protocols, the highest channel throughput and data transfer rate are provided.

With the help of the Simple Participant Discovery Protocol, the DDS standard provides the ability to dynamically connect to the network without having to know the addresses of senders/recipients. This provides a plug and play connection, in which the nodes find each other by themselves. But the protocol does not guarantee compatibility between different implementations from different vendors.

The protocol supports QoS levels for the best effort or reliable connection. A reliable connection is achieved by resending data if the packet has not been delivered, but information about the delivery of the packet will not be received using special messages, as in TCP, but only when the next packet with data is received. This algorithm blocks the receipt of the packet until all the lost packets are delivered. This can increase delivery delays quite significantly in some situations.

Additional data that help in ensuring the reliability of the connection (guarantees of data delivery) can also set limits on the data storage time of this message, limits on the time of message delivery, and other parameters that may be necessary for real-time data transmission.

The protocol implies a modular system that guarantees the absence of a single point of failure, allows for expansion without loss of compatibility, and uses separately from the DDS standard.

As a result, the following advantages of the protocol can be distinguished:

(i) Availability of QoS and flexible configuration of delivery and data storage parameters for each message;
(ii) Plug and play connection;
(iii) Using UDP for fast data transfer;
(iv) No single point of failure.

The disadvantages are as follows:

(i) Increased latency in the case of packet loss;
(ii) Possible incompatibility between vendors.

4.1.3. JMS. JMS is a good solution for systems where reliability can be implemented as static queues. But the usage in real-time systems can be failing in case of using static queues in persistent storage.

Advantages:

(i) It is easy to use. It does not use additional structures, such as for identifying messages that were received. There are no special messages defined for some purposes, such as preventing message duplications;
(ii) It does not require a broker to manage message flow. It uses queues without a special manager that exists in another process, which makes it fault-tolerant;
(iii) May store messages after client failure. In the case of using a static queue, messages can be stored as long as it is required. So, after all, every node can receive all messages sent to them.

Disadvantages:

(i) Standard is too uncertain in many details. It can be interpreted in different ways. So, implementations may have very different characteristics;
(ii) It does not pretend to deliver messages quickly. Other standards may say that they were developed for quick messaging. But JMS was created to allow communication in Java, and it does not say about delivery speed.

4.1.4. AMQP. AMQP cares about message transportation on every transition. This can be used in distributed systems where nodes are grouped in difficult architecture. Also, systems using transactions can be interested in this protocol.

Advantages:

(i) Strong control of message flow on every system node. It uses maps of settled messages and additional frames that allow to ensure that every message has been delivered to each transition node;
(ii) It has a transactional model. It allows the control of full completion of operations. In case of failure or canceling operation, operations will not be

completed. It is possible because messages can be processed only after the completion of a transaction.

Disadvantages:

(i) It does not have a Pub-Sub model that results in using the same way to send messages from publishers to subscribers and to send messages from point to point;
(ii) It has increased frame flow. It uses frames to settle messages, terminate deliveries, etc. So, it can cause a decrease in the latency of common messages because of frame transmission and processing.

4.2. Evaluating the Performance of Protocol Implementations

4.2.1. Evaluating Queue Processing Performance. There are two ascending lines, namely RabbitMQ and PahoMQTT with Mosquitto broker (see Figure 3). As shown in the graph, the delay for these frameworks grows during the whole test. The growth starts at the begging of the test. The peak of delay at the end of the test is several seconds. This is not a good result. Messages are sent more quickly than they can be delivered. The queue is constantly accumulating and the time to receive each packet is delayed. There is a horizontal line. It is FastRTPS. This framework demonstrates a very stable data transmission for this message size. It is the best result. The last one left is GlassFish. Its graph is very similar to FastRTPS, but has delayed growth in the middle of the test. But the delay decreases in the second half of the test. It decreases because messages stop being sent and the queue stops being replenished, messages start being processed until they run out. GlassFish also has a higher delay than FastRTPS but less than 1 second. So, this is a good result.

In Figure 4, there is a graph of delays for a message size of 60000 bytes. There are three rising lines. Two of them are RabbitMQ and PahoMQTT with Mosquitto broker. They do not change their behavior unlike FastRTPS. This is a third ascending line. The situation is more deplorable than with the other two. FastRTPS has the highest delay, which grows through the tests. This may be because FastRTPS (Fast-DDS) uses UDP for data transmission, which imposes a limit on the packet size and the message must be transmitted in several parts if it is larger than 65 kB, which greatly increases the data transfer delay of 1 message. GlassFish does not change its behavior. Its delay grows until the middle of the test and then decreases. The delay is also under 1 second, which is the best result. Thus, according to the results of the two subtests, GlassFish shows the best stable and fastest queue processing performance.

4.2.2. Evaluating the Impact of the Number of Nodes on Performance. In Figures 5–8, minimum and maximum delay and its mean and interquartile range for a different number of subscribers can be seen. Also, mean and maximum delays are presented for each fifth number of subscribers in Table 2.

GlassFish has no delay dependence on the number of subscribers. The maximum delay varies and is more than 100

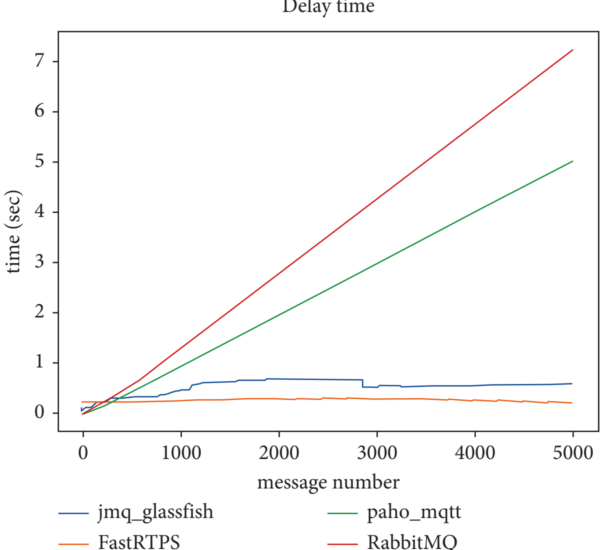

Figure 3: Delay of messages with the size of 50 bytes.

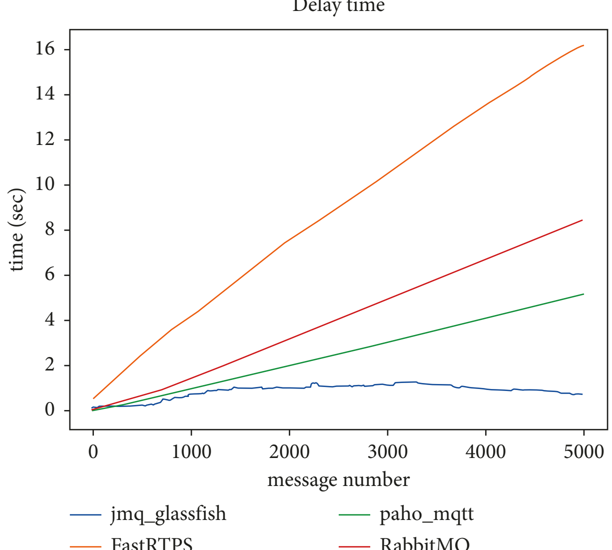

Figure 4: Delay of messages with the size of 60 kB.

milliseconds but without dependence. The mean and the mid-spread are less than 25 milliseconds. But the IQR (interquartile range) is large, and its size is about a few milliseconds.

FastRTPS also has no delay dependence on the number of subscribers. The minimum and maximum are the same during the test. The delay ranges from 1 second or less to 10–12 milliseconds. The mean and mid-spread have no dependence and are in range the from 1 to 4 milliseconds. The IQR is less than 1 millisecond.

PahoMQTT has abnormal jumps of up to 60 and 100 milliseconds on 15 and 18 subscribers. The mean and mid-spread are less than 5 milliseconds. There is no delay dependence on the number of subscribers. RabbitMQ has similar results. There is one jump up to 50 milliseconds on 18

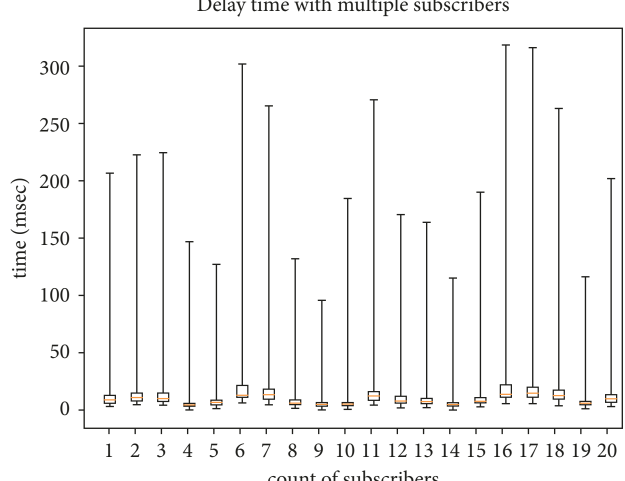

Figure 5: GlassFish's delay boxes with different subscriber counts.

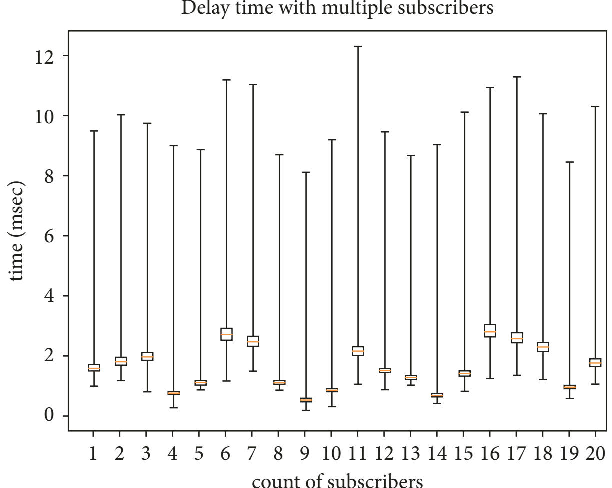

Figure 6: FastRTPS's delay boxes with different subscriber counts.

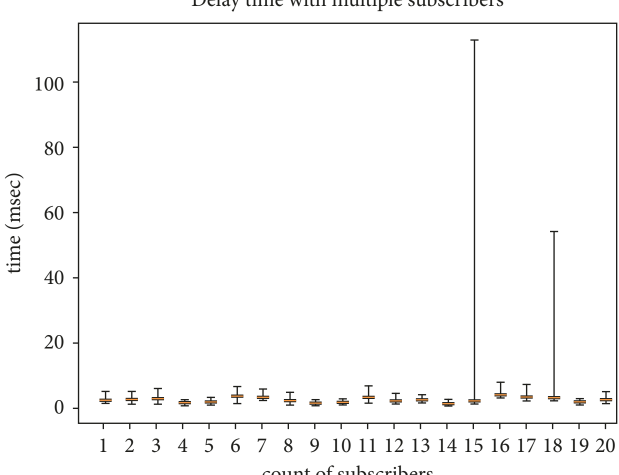

Figure 7: Paho + Mosquitto MQTT's delay boxes with different subscriber counts.

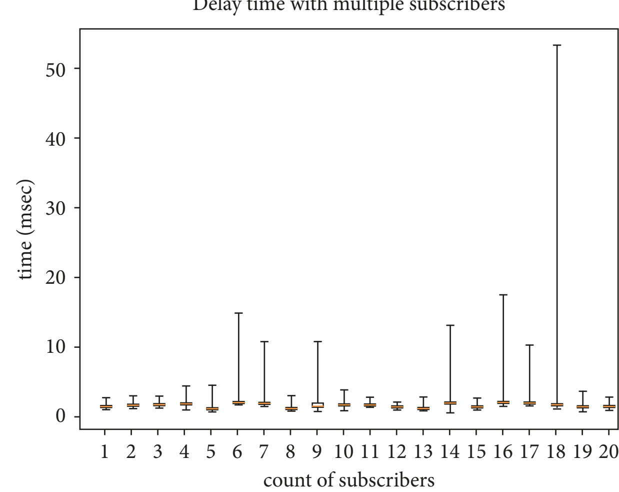

Figure 8: RabbitMQ's delay boxes with different subscriber counts.

subscribers and jumps up to 20 milliseconds on several numbers of subscribers. There is also no delay dependence.

According to the results, no framework has a dependency on the number of subscribers. This means that the software processing of the queues of these implementations takes quite a short time and does not affect the data transfer delay under these conditions. It is a good result because data transfer time does not change with the number of subscribers. In addition, GlassFish shows the worst results, with a mean value of about 10 milliseconds, while the maximum delays reach a few hundred milliseconds. The other three have similar means and mid-spreads, but FastRTPS results are better. However, the maximum delay of FastRTPS is worse than RabbitMQ and PahoMQTT, but the last two have abnormal unknown delay jumps.

4.2.3. Evaluating the Impact of Message Size on Latency. In this test, message size changes every 100 messages. Figure 9 shows delays for all frameworks at a frequency of 100 messages per second. The means and maximums of delays for several message sizes for the frameworks are presented in Table 3.

There is no difference between messages on the GlassFish graph. It is as if messages are sent with a static size instead of increasing ones. The delay and its maximum increase and decrease during the test. The maximum delay reaches 100 milliseconds. The mean grows while the message size is less than 1 MB. After means are similar, they are about 18 milliseconds. Thus, GlassFish delay is not dependent on message size so much.

FastRTPS graph shows that the frequency of delay jumps increases with message size. These jumps are repeated when the test is repeated, which indicates that they are caused by the testing logic. But for other implementations, there are no such jumps, which can be regarded as unstable operation of FastRTPS in these conditions. Also, the mean and maximum delay increase with message size. The mean reaches 14.5

Table 2: Mean and maximum delay with different subscriber counts.

N	GlassFish (JMS)	FastRTPS	PahoMQTT client + Mosquitto broker (MQTT)	RabbitMQ (AMQP)
1	13.78/207.6	1.7/9.4	2.13/4.39	1.38/2.62
5	9.5/128.86	1.18/8.85	1.6/3.32	1.26/4.55
10	8.13/186.0	0.96/9.22	1.39/2.62	1.75/3.7
15	12.2/192.4	1.49/10.11	2.0/112.94	1.25/2.7
20	14.7/202.8	1.86/10.2	2.54/4.83	1.38/2.82

Each column contains the mean/max value in ms.

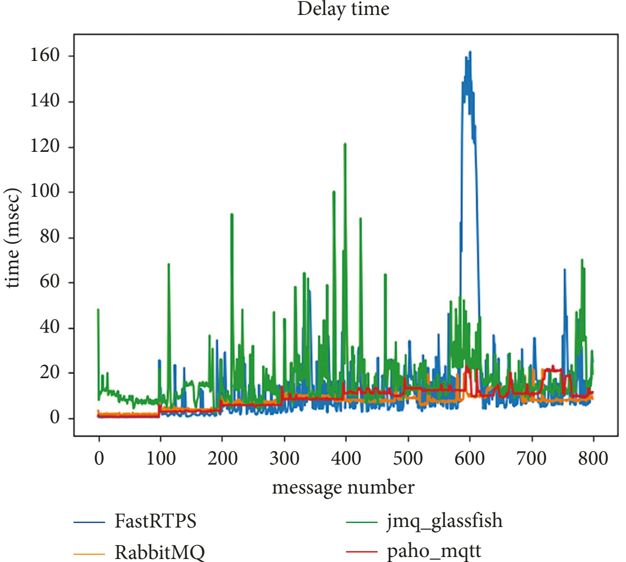

Figure 9: Delay of messages with 100 msgs/sec frequency.

milliseconds and the maximum reaches 65.5 milliseconds at the maximal message size. At a message size of 1.5 MB, there is an abnormal spike. The maximum delay is 160 milliseconds. Thus, FastRTPS delay depends on message size. It increases with size, but it is a linear dependence.

The PahoMQTT and RabbitMQ graphs look like stairs. Each step represents the moment of increase in message size. So, the delay of these two frameworks depends on the message size. The steps are clear while the size is under 1 MB. Thereafter, the delay becomes more chaotic. There is an interesting feature: PahoMQTT shows a better result until the message size reaches 1 MB. If the message size is greater than 1 MB, the mean of RabbitMQ delay is less than the PahoMQTT delay. It can be seen in Figure 9 that the RabbitMQ graph is under the PahoMQTT graph. But the maximum values of these frameworks are quite close. The difference is a few milliseconds.

According to the results, all frameworks have a significant delay dependence on message size, except GlassFish. But the last one has a very high delay. The means are the same as maximums, for example RabbitMQ. As described earlier, if the message size is less than 1 MB, PahoMQTT is better than RabbitMQ, otherwise vice versa. FastRTPS delay is similar to PahoMQTT delay, but has more scatter. The maximums of the former are several times larger than the latter. Thereby, RabbitMQ shows the best results. GlassFish shows message size-independent results, but with high delays.

4.2.4. Estimation of the Jitter Value and Minimum Delays. In Figures 10 and 11, a graph of the delay of the minimum message size is shown. That delay corresponds to the minimal one because there is no or minimal user data. In Table 4, the latency, jitter, and RTT for all frameworks are shown. As can be seen from the graph and table, GlassFish has the highest delay of about 2.5 milliseconds with jumps over 20 milliseconds. Jitter is also high, at about half a millisecond. This looks like a bold line on the graph. This is a bad result, because there are no other messages or any large data transmission. So, only one message containing protocol information is processed, and it takes a few milliseconds. It is also very unstable, because the delay scatter is about 1 millisecond. RTT is also high, over 5 milliseconds on a mean. That means that the receiver has to wait about 5 milliseconds to get a reply from another client. Hence, it is also a poor result. FastRTPS has the lowest delay of 300 microseconds and jitter of 30 microseconds, which is the best result. But there are jumps over 2 milliseconds. This is not good and can be critical in some cases. RTT is about 2.5 milliseconds on a mean. But the maximum of RTT is below the GlassFish RTT mean, so this again indicates that the GlassFish results are poor. PahoMQTT + Mosquitto graph is similar to the FastRTPS graph. Both of them are thin lines in the graph. But PahoMQTT has twice the delay on the mean. The jitter maximum of PahoMQTT is also higher. The RTT mean of PahoMQTT is lower than the mean of FastRTPS, but the maximum of the former is higher than the latter. The RabbitMQ graph shows frequent jumps over 2 milliseconds while the mean is about 1 millisecond. This is bad because the real mean ranges up to 2 milliseconds which is similar to GlassFish. There are also abnormal jumps of up to 16 milliseconds. RabbitMQ RTT is about 3 milliseconds which is neither good nor bad. But the maximum of RTT is up to 20 milliseconds, which is also a problem.

According to the results, FastRTPS shows the best minimal delays that are less than 0.5 milliseconds. It also has the best jitter value of a few tens of microseconds. PahoMQTT with Mosquitto broker also shows good minimal delays and jitter values that are very close to those of FastRTPS. GlassFish has worse minimal delay and jitter values. RabbitMQ has better results than the previous one, but it has a large scatter, which can be seen on the graph.

4.2.5. Evaluation of the Effect of the Number of Interacting Processes in Different Conditions on Delays. In this test, first of all, we will consider the effect of the frequency of sending messages on delays. Figure 12 shows the result for FastRTPS,

Table 3: Mean and maximum delay with different message sizes.

Implementation	128 B	524 kB	1 MB	1.5 MB	1.8 MB
GlassFish (JMS)	9.40/47.8	14.45/90.16	18.86/88.22	17.14/44.67	17.70/69.96
FastRTPS	0.57/25.54	6.49/29.03	10.18/31.37	29.66/161.97	14.57/65.64
PahoMQTT client + Mosquitto broker (MQTT)	0.64/5.49	5.83/13.74	11.35/17.77	11.34/21.57	14.53/23.07
RabbitMQ (AMQP)	1.40/6.85	7.09/13.30	8.09/15.63	11.65/18.89	8.25/21.48

Mean/max latency values are in ms for messages of different sizes.

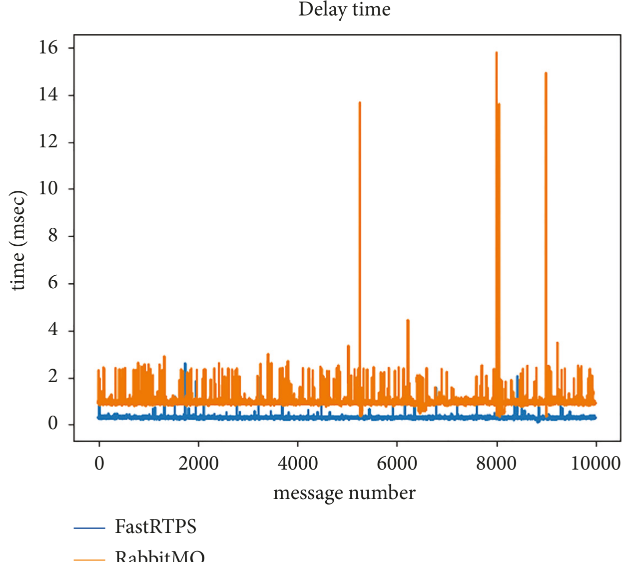

Figure 10: FastRTPS's and RabbitMQ's delay of minimum message size.

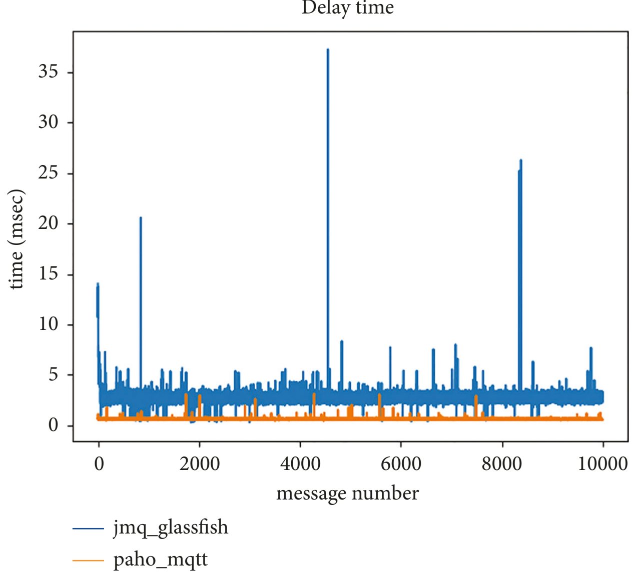

Figure 11: GlassFish's and Paho + Mosquitto MQTT's delay of minimum message size.

RabbitMQ, and GlassFish. The results are quite different since FastRTPS and RabbitMQ handle such a data flow and do not accumulate a queue at any message size. And in the case of GlassFish, the queue appears at the smallest message size and varies throughout the test in the range from 0 to 14 messages. The queue appears already at 400 messages per second. That is, problems with GlassFish already appear when transmitting about 1.2 MB per second.

In the case of Mosquitto and PahoMQTT, the results are very ambiguous, because, at a frequency of more than 400 messages per second, the delay increases to seconds, as can be replaced in Figure13. This is due to the long execution of the message receiving function *consume_message* (see Figure 14). Figure 15 shows a graph of the execution time of the read function provided by the PahoMQTT library API.

Due to the execution of the message receiving function, the receiving time is set later, since our system examines the time when the message is transmitted from the user's sending point to the receiving point by another user, and not the time when the message is received by the operating system or the broker, or when the message appears in the queue at the destination point. The time at the beginning of the test is about 1 ms, which in the context of a large density of received messages greatly increases the queue and reduces the data transfer rate. The testing system does not affect these delays in any way, but only created conditions under which this implementation shows a terrible result.

We can calculate the amount of data transmitted at a frequency of 1000 messages per second on the last 300 messages, where the payload size reaches the maximum (see equation (1)).

$$\text{Bandwidth} = 1.8\,\text{MB} \cdot \frac{1000\,\text{msgs}}{\text{sec}} = \frac{1.8\,\text{GB}}{\text{sec}}. \quad (1)$$

FastRTPS and RabbitMQ cope with this load, although the delays vary, and in the case of FastRTPS, it is no more than 8 ms, and in the case of RabbitMQ, it is no more than 30 ms. The rest of the implementations cannot withstand this load and do not have time to process the queue, although all implementations run in a loop with an interval of 1 ms. That is, checking for new data with an interval of 1 ms, which in the worst case will give an additional delay of 1 ms. This can happen if the message is located in the recipient exactly after the end of the check for the presence of the message. This interval was chosen to reduce the load on the CPU.

GlassFish simply does not have time to transmit and process such a data stream, and in the case of MQTT, the reasons for the long wait to receive data have not been investigated, but may be related to the load on the broker or the long processing of data on the client. It is worth noting that these results may not apply to any situation where messages are transmitted with a given frequency and a given message size, but rather to this particular test scenario in which complex conditions are recreated.

Table 4: Latency, jitter, and RTT of minimum message size.

Implementation	Latency	Jitter	RTT
GlassFish (JMS)	2.62/37.26	0.57/34.64	5.25/73.49
FastRTPS	0.30/2.60	0.03/2.30	2.67/5.15
PahoMQTT client + Mosquitto broker (MQTT)	0.63/3.11	0.039/2.48	2.45/8.42
RabbitMQ (AMQP)	0.96/15.79	0.10/14.83	3.11/19.41

Mean/max latency, jitter, and RTT values are in ms.

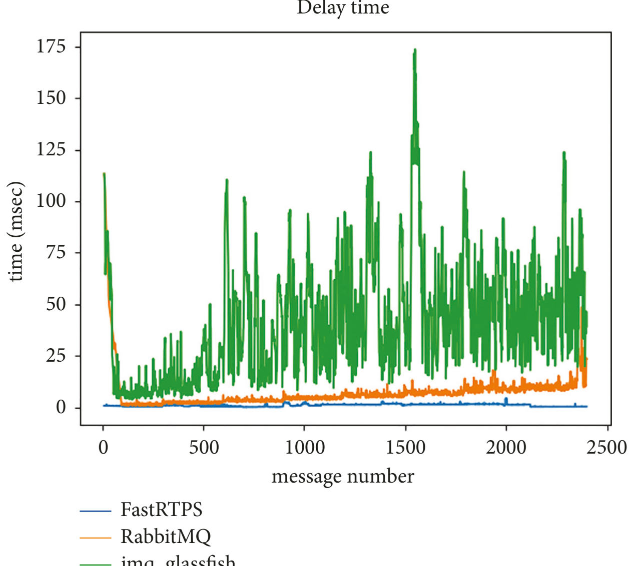

Figure 12: FastRTPS's, RabbitMQ's, GlassFish's latency with a frequency of 1000 messages per second and different message sizes.

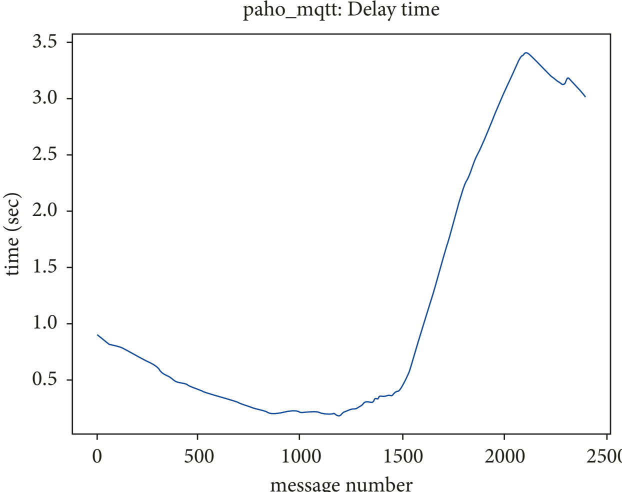

Figure 13: PahoMQTT's latency with a frequency of 1000 messages per second and different message sizes.

Next, the dependence of RTT on the number of pairs of interacting processes is considered. All pairs are identical and do not have any priority of data transmission in comparison with each other. From the test, we want to

```
auto start_timestamp = duration_cast<nanoseconds>
        (high_resolution_clock : : now () . time_since_epoch () ) . count () ;
auto msg = _client_sub . consume_message () ;
unsigned long proc_time = duration_cast<nanoseconds>
        (high_resolution_clock : : now () . time_since_epoch () ) . count () - start_timestamp ;
```

Figure 14: Measuring the execution time of the PahoMQTT library read function.

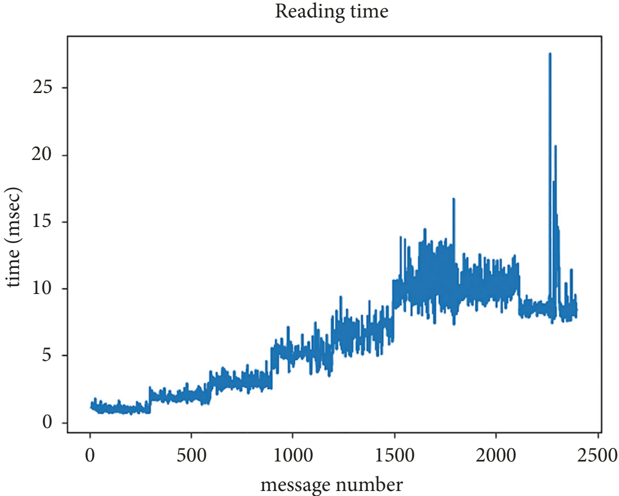

Figure 15: PahoMQTT's data processing time.

consider the behavior of different pairs of processes and the effect of the number of pairs on delays. The frequency of sending messages is 400 messages per second, which gives a fairly large data stream, but this does not create conditions to accumulate a huge queue for each of the considered implementations.

Figure 16 shows the RTT graph for FastRTPS with three pairs of interacting processes. The jumps on the charts are particularly noticeable in the area of every 300 messages, since it is with this frequency that the message size increases. Jumps are associated with the reallocation of memory for the buffer, which creates additional delays. The RTT time for any message size does not exceed 6 ms, with the greatest delays being achieved in the middle of the test with a message size of 0.7 MB. By the end of sending data of a certain size, the delays are reduced. This phenomenon can be explained by the fact that when the message size changes, a small queue of several messages appears, which adds a significant part of the RTT, and by the end of sending the next 300 messages, the queue goes away and the delays fall to their real values. At the end of the test, the delay is less than in the middle, since there

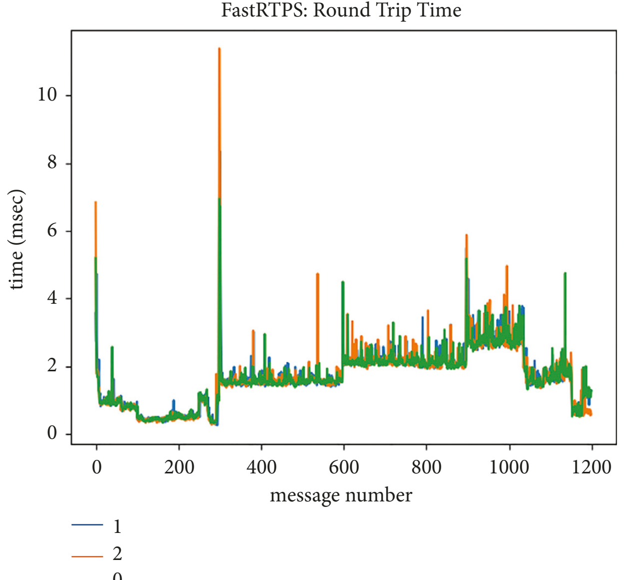

Figure 16: FastRTPS's RTT with three communicating process pairs.

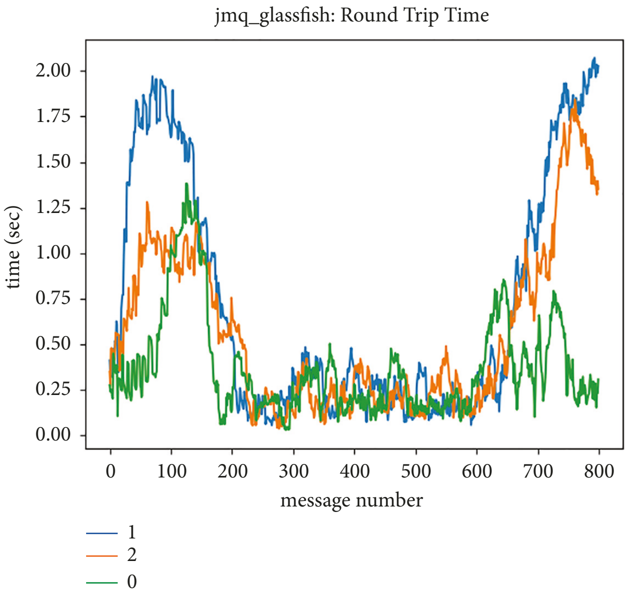

Figure 17: GlassFish's RTT with three communicating process pairs.

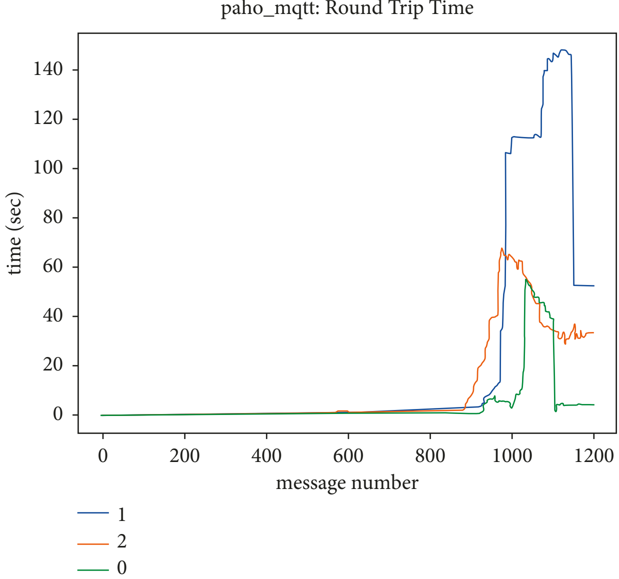

Figure 18: PahoMQTT + Mosquitto broker RTT with three communicating process pairs.

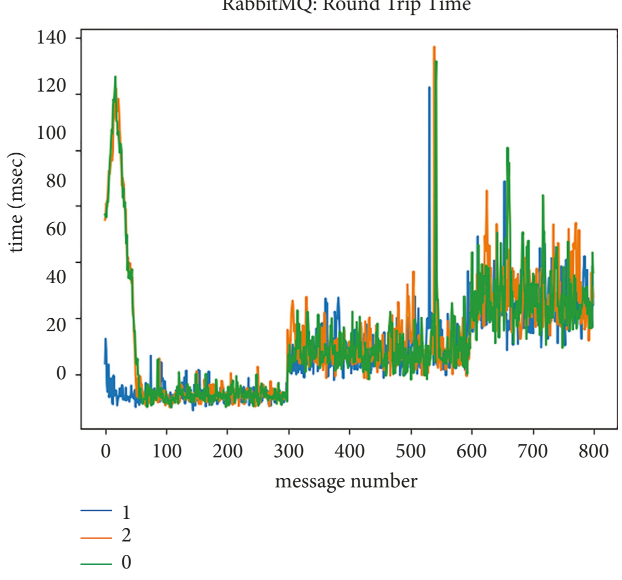

Figure 19: RabbitMQ's RTT with three communicating process pairs.

is no queue and new messages stop being sent, which reduces RTT.

When studying the influence of the number of pairs, there is no obvious dependence, but with a large number of interacting processes, the maximum RTT value in some cases increases. These maximum values are not repeated from test to test and are random, with a random message size. As shown for three pairs, the RTT graphs overlap each other, which shows the uniform distribution of library resources between processes. This is a good indicator, as it is a sign of the predictability of delays, which is very important for real-time systems. It is very important to evenly distribute resources between equal priority data transmission channels because there is no need to think about the fact that any node in the network will take part of the resources for data transmission. There will be no need to reduce the amount of data transmitted over one channel in favor of another. The only exception to think about is limited bandwidth.

Analyzing the results of GlassFish, a direct relationship between the number of pairs and RTT is seen, since the delays when two pairs of processes work simultaneously increase by about two times, and when adding another pair

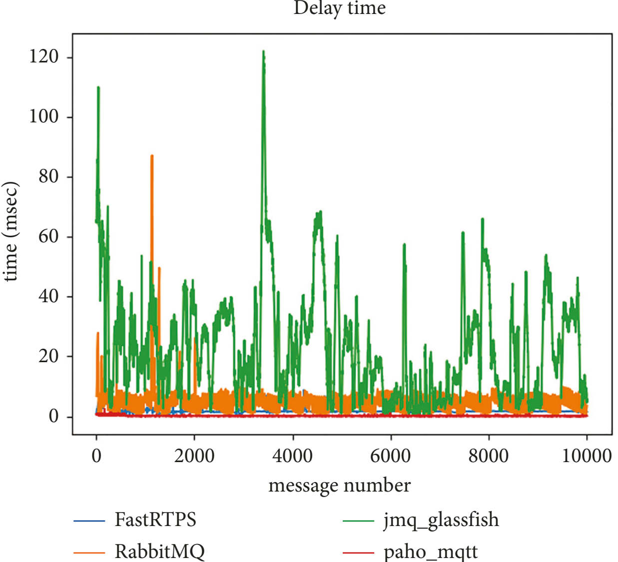

Figure 20: The delay time in the case of a limited queue.

of RTT increases to the order of seconds (see Figure 17). In this case, the behavior of the pairs is very similar, although the RTT value of each pair is not as close as in the case of FastRTPS. This software solution cannot cope with the load of one pair of interacting processes, and as the number of processes increases, the load increases many times and the queue quickly accumulates.

The results of PahoMQTT with Mosquitto are similar, with an increase in the number of pairs, this solution does not cope, and already with two pairs of interacting processes, RTT reaches the order of seconds. Figure 18 shows a graph for three pairs, where with a large message size, RTT reaches hundreds of seconds.

The behavior of these two implementations does not meet the requirements of real-time systems.

RabbitMQ, in turn, showed similar FastRTPS results with random outliers and the same behavior of process pairs, but a noticeable increase in latency from one case to three, which can also be seen in Figure 19.

4.2.6. Investigation of Delays in Artificially Fixing the Message Queue. Consider the case of sending messages of 10 bytes without an interval with the maximum possible frequency, but we limit the queue to 50 messages. First, the state of the queue during the test was considered. As a result, FastRTPS kept the queue state at the level of 50 messages almost throughout the entire test, but there were rare single jumps when the queue was empty and abruptly accumulated back. The state of the GlassFish queue turned out to be quite unstable and constantly fluctuated from a few messages to 50, similar to RabbitMQ, but in the range from 0 to 30 messages. This situation is explained by the fact that the queue is calculated based on the difference between the number of messages sent and the number of messages received, and it takes time to transmit a response message. In addition, the check for new messages occurs at intervals of 1 ms, which also gives time to process the queue.

In the case of PahoMQTT with Mosquitto, the queue did not rise more than six messages and did not have time to accumulate, because one node did not have time to send data with sufficient frequency. This is also because this implementation waits for the message delivery information to be received before sending a new one. Therefore, this solution does not make sense to compare with the others, since it is in different conditions. But when testing the case without waiting for this information, the queue jumps in the range from 0 to 35 messages and the delay does not exceed 2 ms, and the mean delay value was 0.5 ms. This result is the best among all the studied implementations.

Figure 20 shows the delays of each implementation, as the result of FastRTPS showed the best result with a delay of about 6 ms, GlassFish showed a huge spread of values, and if we consider the peak delay and the state of the queue of 50 messages, it turns out that it took 120 ms to deliver 50 small messages, which is a terrible indicator. RabbitMQ turned out to be worse and not as stable as FastRTPS; on average, the delay of this implementation was less than 10 ms with a queue state of less than 30 messages. With a smaller queue, the delays were greater than those of FastRTPS.

4.3. Result Comparison. A lot of papers have pointed out that DDS is the best solution for real-time data transmission. The authors of [26] compared popular protocols based on theoretical information. They concluded that the problem of transferring data many to many in real-time DDS is the best solution.

The authors of [14] showed that DDS is one of the leaders in delivery performance. Their result for DDS at payload size 2 B coincides with our result obtained in the fourth scenario (Section 4.2.4). There is a difference of 2 milliseconds because in our experimental setup each node performs an incoming message check only once every 1 millisecond. Therefore, the additional delay is up to 2 milliseconds for RTT.

A comparison of protocols using the Anglova scenario was conducted in [27]. The results show that the DDS implemented with OpenDDS has a lower latency and jitter compared to RabbitMQ and ZeroMQ.

A comparison of FastRTPS and ROS-MQTT was made in [28]. The results show that the average FastRTPS latency for an empty message is 620 microseconds. This is close to our result in the third scenario (Section 4.2.3), where FastRTPS has an average latency of 570 microseconds for a 128 B message size. But the other results do not match because the authors are experimenting on a more complex architecture with networked data transmission, while our scenarios are run on a single machine. This is also true for the MQTT results. But overall, results show that FastRTPS has an advantage over MQTT which is coincident with our results.

The authors of [29] compared the popular protocols in terms of performance in eHealth solutions. They considered protocols from a different angle, but the results coincide with ours: DDS is the best solution. The graph where the sampling rate is 1000 coincides with the results obtained in the fifth

scenario (Section 4.2.5). There are many differences in the specific values, because the experimental setups are very different. But, in general, the results are the same: DDS showed the best latency, followed by AMQP, then JMS, and finally MQTT, which showed a very poor latency. The authors have summarized their results in a comparative table with the strengths and weaknesses of each protocol. But in this table, JMS has strong points such as "low latency" and "suitable for real-time applications." This looks very strange, because DDS does not have "low latency" in the strengths, but has lower latency in the results, and AMQP has "not suitable for real-time applications" in the weaknesses, but the results show that AMQP is the same or better than JMS.

An analysis of MQTT, CoAP, and XMPP was performed in [30]. In this study, real data, such as temperature, humidity, and light, were sent 100 times for each protocol. The results show that the average delay of MQTT is about 589 microseconds. This is better than the average delay of CoAP (821 microseconds) and XMPP (41 milliseconds). Since the temperature, humidity, and light data are lightweight and the transmission was in one direction, the MQTT result is the same as the results obtained in the third scenario (Section 4.2.3): 640 microseconds for MQTT with a message size of 128 B.

The MQTT result in the third scenario is also the same as the result from [31], where MQTT has a latency of 1 to 1.5 milliseconds. The difference of two times may be caused by the packet transmission time over the network, in contrast to our experimental setup where all components (publisher, broker, and subscriber) are placed on the same machine.

There are a lot of comparisons between AMQP and MQTT. In [32], the authors compared the RTT of AMQP and MQTT. The results show that MQTT is slightly better than AMQP: the RTT of AMQP is about 700 microseconds and the RTT of MQTT is about 500 microseconds. This coincides with the RTT results from the fourth scenario (Section 4.2.4). The difference of a few milliseconds is explained by the fact that in our experimental setup all receiving nodes work in a 1-millisecond wait cycle, which causes an additional delay of up to 2 milliseconds in this scenario. Other small time differences may be caused by the operating system or different versions of the frameworks.

In [33], the authors also compare the RTT for AMQP and MQTT and show that AMQP is better in most cases. Most measured values for AMQP and MQTT range from 2 milliseconds to 6 milliseconds. These results are matched by our results for the fourth scenario (Section 4.2.4). But in our results, MQTT is slightly better than AMQP, in contrast to the conclusion in [33]. The cause for this may be that the authors used the Python programming language, the Raspberry PI, or the pika library for the AMQP broker.

In [34], the authors show that AMQP and MQTT have the same delay (MQTT is slightly better). Since the authors did not provide a detailed description of the test scenario, it is difficult to compare with our results, but it is similar to the results of the third scenario (Section 4.2.3), where AMQP and MQTT have similar delays for message sizes up to 1 MB.

In [35], AMQP and MQTT delays were compared for different scenarios: sending data from weather sensors and sending data from city cameras. The results show that for scenarios with sensors, MQTT has a lower delay (less than 1 millisecond) than AMQP. Since sensor data are lightweight, this is coincident with the results obtained in the third scenario (Section 4.2.3) with a message size of 128 B, where MQTT has an average delay of 640 microseconds. But for the scenario with cameras, the result is reversed, with AMQP having a lower delay (less than 13 milliseconds). This is also coincident with the results obtained in the third scenario (Section 4.2.3): as the message size increases, MQTT shows higher latency compared to AMQP.

The authors of [36] also concluded that MQTT is better suited for small message sizes. Their results show that MQTT has lower latency (from 400 microseconds to 500 microseconds) than AMQP (from 800 microseconds to 1100 microseconds) for message sizes up to 4096 B. This is matched by our results obtained in the third scenario (Section 4.2.3) with a message size of 128 B. The latency in our results is higher because the network speed is less than 1 GB/s and the broker has lower system characteristics. The authors also experimented a 5000-message sending rate and message size of 64 B. This is very close to the first scenario, where messages are sent without intervals. The results are matched: latency increases throughout the scenario.

5. Conclusions

Described protocols are different in initial purposes. JMS is a simple standard for messaging in Java programs, that does not specify messaging mechanics. MQTT and AMQP are protocols for messaging using queues that differ in some details. RTPS has a purpose to make real-time messaging available to its users by having more strict rules. For real-time systems, the best protocols are MQTT and RTPS. MQTT is good at transmitting small messages, while RTPS is a more versatile and flexible protocol with more options for changing the QoS.

The article presents a methodology that allows to investigate the software delays of solutions in various conditions, but the conditions of the study have limitations:

(i) The interval for checking new messages is 1 ms, which in the worst case can give an error of 1 ms;

(ii) Testing was carried out on a single machine using network data transfer protocols, but minimizing the impact of network delays. Thus, it is assumed that the network is stable, although the instability of the network can greatly affect the result [37];

(iii) The test from Section 3.1.2 uses a fairly small number of simultaneously running subscribers;

(iv) All tests are aimed at investigating the delays introduced by the middleware;

(v) In the developed methodology, nonstandard behavior of nodes was stimulated, for example increasing the size of transmitted data during one test, which is not typical.

Considering the results, GlassFish is the best solution for message sending without intervals. It can be used in systems

with a lot of messages, where other solutions will be too slow. But in all other cases, it has the worst results and cannot compete with other solutions.

Paho is very good at transferring messages of minimum size. It could be a good solution for systems with very small messages, such as commands that can be presented as few bytes.

RabbitMQ is one of the fastest solutions. It shows itself good in all cases, except in sending messages without an interval. RabbitMQ is the best at sending messages with a big size and with multiple subscribers.

FastRTPS is almost the best solution. There is only one problem—message processing without an interval between sending messages. FastRTPS has the most stable delays which makes it predictable. It is very important in real-time systems.

In IoT systems almost all investigated cases are presented. RabbitMQ and FastRTPS are good in most cases and should be considered firstly. RabbitMQ should be used in systems with big messages and a lot of consumers of the same data. FastRTPS should be used in real-time systems, where delays have low jitter and are bounded by a constant value. The usage of GlassFish and Paho is more limited. They should be used only in special cases, such as too frequent message sending and low-size data transferring.

When building a system with various data, which includes both small command messages and large amounts of data from sensors, cameras, etc., the RTPS protocol shows itself in the best way. But none of the software solutions considered by us provides guarantees of delivery in isochronous mode, and in all cases, in certain situations, there are unpredictable jumps and the accompanying growth of the message queue. To provide real guarantees, it is necessary to use link-layer protocols that provide appropriate capabilities and guarantees, since the application layer is highly dependent on external factors.

Data Availability

No data were used to support this study.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was funded by the "Development Program of ETU "LETI" within the Framework of the Program of Strategic Academic Leadership" Priority-2030 No 075-15-2021-1318 on 29 September 2021.

References

[1] W. Rafique, L. Qi, I. Yaqoob, M. Imran, R. U. Rasool, and W. Dou, "Complementing IoT services through software defined networking and edge computing: a comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, pp. 1761–1804, 2020.

[2] I. McAteer, M. I. Malik, Z. Baig, and P. Hannay, "Security vulnerabilities and cyber threat analysis of the AMQP protocol for the internet of things," in *Proceedings of the 15th Australian Information Security Management Conference*, pp. 70–80, Perth, Western Australia, 2017.

[3] S. Andy, B. Rahardjo, and B. Hanindhito, "Attack scenarios and security analysis of MQTT communication protocol in IoT system," in *Proceedings of the 2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, pp. 1–6, Yogyakarta, Indonesia, 2017.

[4] M. S. Harsha, B. M. Bhavani, and K. R. Kundhavai, "Analysis of vulnerabilities in MQTT security using Shodan API and implementation of its countermeasures via authentication and ACLs," in *Proceedings of the 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 2244–2250, Bangalore, India, 2018.

[5] A. Mileva, A. Velinov, L. Hartmann, S. Wendzel, and W. Mazurczyk, "Comprehensive analysis of MQTT 5.0 susceptibility to network covert channels," *Computers & Security*, vol. 104, Article ID 102207, 2021.

[6] C. A. Garcia, J. E. Naranjo, and M. V. Garcia, "Analysis of AMQP for industrial internet of things based on low-cost automation," *Smart Innovation, Systems and Technologies*, vol. 201, pp. 235–244, 2019.

[7] M. Collina, M. Bartolucci, A. Vanelli-Coralli, and G. E. Corazza, "Internet of Things application layer protocol analysis over error and delay prone links," in *Proceedings of the 2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pp. 398–404, Livorno, Italy, 2014.

[8] P. Benedick, J. Robert, Y. Le Traon, and S. Kubler, "O-MI/O-DF vs. MQTT: a performance analysis," in *Proceedings of the 2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pp. 153–158, St. Petersburg, Russia, 2018.

[9] S. Chouali, A. Boukerche, and A. Mostefaoui, "Towards a formal analysis of MQtt protocol in the context of communicating vehicles," in *Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access (MobiWac'17)*, pp. 129–136, Miami, FL, USA, 2017.

[10] S. Lee, H. Kim, D. Hong, and H. Ju, "Correlation analysis of MQTT loss and delay according to QoS level," in *Proceedings of the International Conference on Information Networking 2013 (ICOIN)*, pp. 714–717, Bangkok, Thailand, 2013.

[11] N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP," in *Proceedings of the 2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, Vienna, Austria, 2017.

[12] R. Sanika, "Performance comparison of message queue methods," UNLV Theses, Dissertations, Professional Papers, and Capstones, University of Nevada, Las Vegas, NV, USA, 2019.

[13] E. Nilsson and V. Pregén, "Performance evaluation of message-oriented middleware," Dissertation, KTH Royal Institute of Technology, Stockholm, Sweden, 2020.

[14] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: performance evaluation of industry 4.0 protocols," in *Proceedings of the 2019 IEEE International Conference on Industrial Technology (ICIT)*, pp. 955–962, Melbourne, Australia, 2019.

[15] G. Pardo-Castellote, "OMG data-distribution service: architectural overview," in *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops*, Providence, RI, USA, 2003.

[16] T. White, M. N. Johnstone, and M. Peacock, "An investigation into some security issues in the DDS messaging protocol," in *Proceedings of the 15th Australian Information Security Management Conference*, pp. 132–139, Perth, Australia, 2017.

[17] M. J. Michaud, T. Dean, and S. P. Leblanc, "Attacking OMG data distribution service (DDS) based real-time mission critical distributed systems," in *Proceedings of the 2018 13th international Conference on Malicious and Unwanted software (MALWARE)*, pp. 68–77, Nantucket, MA, USA, 2018.

[18] A. Gavrilov, M. Bergaliyev, S. Tinyakov, and K. Krinkin, "Industrial messaging middleware: standards and performance evaluation," in *Proceedings of the 2020 IEEE 14th International Conference on Application of Information and Communication Technologies (AICT)*, Tashkent, Uzbekistan, October 2020.

[19] K. Krinkin, A. Filatov, A. Filatov, O. Kurishev, and A. Lyanguzov, "Data distribution services performance evaluation framework," in *Proceedings of the 2018 22nd Conference of Open Innovations Association (FRUCT)*, Jyvaskyla, Finland, May 2018.

[20] XMPP Standards Foundation, *XEP-0184: Message Delivery Receipts, Version 1.4.0 (2018-08-02). Extensible Messaging and Presence Protocol*, XMPP Standards Foundation, Parker, CO, USA, 2018.

[21] "Eprosima Performance," http://www.eprosima.com/index.php/resources-all/performance/.

[22] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," in *Proceedings of the 2015 14th RoEduNet International Conference–Networking in Education and Research (RoEduNet NER)*, pp. 132–137, Craiova, Romania, 2015.

[23] P. Dobbelaere and K. S. Esmaili, "Kafka versus RabbitMQ: a comparative study of two industry reference publish/subscribe implementations: industry paper," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS'17)*, pp. 227–238, New York, NY, USA, 2017.

[24] B. Mishra and B. Mishra, "Evaluating and analyzing MQTT brokers with stress-testing," in *Proceedings of the (CS)2-12th Conference of PhD Students in Computer Science at: University of Szeged*, Szeged, Hungary, 2020.

[25] B. Mishra, "Performance evaluation of MQTT broker servers," *Computational Science and Its Applications–ICCSA 2018*, vol. 10963, pp. 599–609, 2018.

[26] S. Seleznev and V. Yakovlev, "Industrial application architecture IoT and protocols AMQP, MQTT, JMS, REST, CoAP, XMPP, DDS," *International Journal of Open Information Technologies*, vol. 7, no. 5, pp. 17–28, 2019.

[27] N. Suri, "Experimental evaluation of Group communications protocols for tactical data dissemination," in *Proceedings of the MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pp. 133–139, Los Angeles, CA, USA, 2018.

[28] A. Mokhtarian, A. Kampmann, M. Lueer, S. Kowalewski, and B. Alrifaee, "A cloud architecture for networked and autonomous vehicles," *IFAC-PapersOnLine*, vol. 54, pp. 233–239, 2021, 2.

[29] A. Talaminos-Barroso, M. A. Estudillo-Valderrama, L. M. Roa, J. Reina-Tosina, and F. Ortega-Ruiz, "A machine-to-machine protocol benchmark for eHealth applications–use case: respiratory rehabilitation," *Computer Methods and Programs in Biomedicine*, vol. 129, pp. 1–11, 2016.

[30] B. H. Çorak, F. Y. Okay, M. Güzel, S. Murt, and S. Ozdemir, "Comparative analysis of IoT communication protocols," in *Proceedings of the 2018 international Symposium on networks, Computer and Communications*, pp. 1–6, Rome, Italy, 2018.

[31] M. Iglesias-Urkia, A. Orive, M. Barcelo, A. Moran, J. Bilbao, and A. Urbieta, "Towards a lightweight protocol for industry 4.0: an implementation based benchmark," in *Proceedings of the 2017 IEEE international Workshop of Electronics, control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, Donostia, Spain, May 2017.

[32] D. Yoshino, Y. Watanobe, and K. Naruse, "A highly reliable communication system for internet of robotic things and implementation in RT-middleware with AMQP communication interfaces," *IEEE Access*, vol. 9, pp. 167229–167241, 2021.

[33] N. Q. Uy and V. H. Nam, "A comparison of AMQP and MQTT protocols for internet of things," in *Proceedings of the 2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*, pp. 292–297, Hanoi, Vietnam, 2019.

[34] F. Zujie and K. JaeSoo, "Transmission performance comparison and analysis with different publish/subscribe protocol," in *Proceedings of the Korean Society of Computer Information Conference*, pp. 77–80, Seoul, South Korea, 2020.

[35] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting IoT platform requirements with open pub/sub solutions," *Annales Des Telecommunications*, vol. 72, no. 1-2, pp. 41–52, 2017.

[36] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems," in *Proceedings of the 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pp. 1217–1223, Munich, Germany, 2018.

[37] D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. K. Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," in *Proceedings of the 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, Singapore, 2014.