OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols

Stefan Profanter, Ayhun Tekat, Kirill Dorofeev, Markus Rickert, Alois Knoll

Abstract—**Ethernet-based protocols are getting more and more important for Industry 4.0 and the Internet of Things. In this paper, we compare the features, package overhead, and performance of some of the most important protocols in this area. First, we present a general feature comparison of OPC UA, ROS, DDS, and MQTT, followed by a more detailed wire protocol evaluation, which gives an overview over the protocol overhead for establishing a connection and sending data. In the performance tests we evaluate open-source implementations of these protocols by measuring the round trip time of messages in different system states: idle, high CPU load, and high network load. The performance analysis concludes with a test measuring the round trip time for 500 nodes on the same host.**

I. Introduction

Industry 4.0 is a current trend in industrial automation which is often referred to as the fourth industrial revolution. Its main goal is to modernize the way manufacturing and production work by easing the integration of devices and by improving the communication between all the devices on the shop floor. Components from different manufacturers need to communicate in a common language by using standardized communication protocols. Current manufacturing systems use are a lot of vendor-dependent field bus protocols that handle this task. As a result, an integrated system can suffer from many different communication paradigms which reduce the system's overall flexibility and adaptability.

In this paper, we evaluate the performance and resource usage of some of the most common protocols in the area of industrial automation and Internet of Things (IoT): OPC UA, DDS, ROS, MQTT. We chose these protocols since they are commonly used in our research projects. Other possible candidates for further evaluation could be CoAP, XMPP, and AMQP.

According to [1], these protocols are among the most important ones to be considered in IoT applications. All presented protocols try to deal with the problem described above by providing a standardized, open, and manufacturer-independent protocol. With new and improved real-time features added to Ethernet-based protocols over the last few years, they are slowly replacing conventional and proprietary field bus communication [2]. One of the major advantages of Ethernet-based protocols is the well known standard which is implemented in many different operating systems and microcontrollers, thus resulting in lower hardware costs. Field bus protocols on the other hand are either implemented as closed source or only on specific hardware in combination with licensing fees.

Today, there is a growing demand in the industry for distributed applications, where multiple devices from different vendors control different types of objects [3]. Selecting an appropriate middleware, a software layer located between applications and operating system, becomes one of the most crucial tasks to simplify and speed up a development of distributed systems [4]. As these industrial systems usually require hard real-time constraints, the performance of a middleware layer becomes a critical issue when designing such a system.

This paper is split into multiple sections: Section II gives a short overview on related work. The main concepts and application domains of the different middlewares are described in Section III followed by a more detailed evaluation of the package overhead of the underlying protocol in Section IV. Section V introduces our architecture for performance testing and lists some of the tests we used for comparison. This section also describes some of the tools used to test the protocols under heavy system and network load. In Section VI the most important results of the comparison are shown and discussed. In Section VII we give a summary of the results and an outlook on future work.

II. Related Work

Numerous performance tests have already been conducted in various domains. In the IoT domain, where a usual scenario includes resource-constrained devices communicating with each other over low-bandwith or probably unreliable wireless networks, the question of the communication protocol's performance evaluation is seen as a crucial one. [5] compares bandwidth consumption and latency of most common IoT protocols, including MQTT, CoAP, and DDS. [6] evaluates CoAP, MQTT, MQTT-SN, TCP, and Websockets, also comparing energy performance and CPU power consumption for each of the protocols. A performance comparison of data usage and the time spent to send and receive messages for MQTT and OPC UA can be found in [7].

A survey of supported communication paradigms between OPC UA and DDS is presented in [8], with a focus on how both protocols can be run in hybrid deployments.

Some papers present performance evaluations for a specific protocol: [9] shows how an OPC UA server performs in a field device, measuring response times, memory, and CPU utilization of an OPC UA Server running on an Altera Cyclone I FPGA. [10] evaluates the performance of different

S. Profanter, K. Dorofeev, M. Rickert, A. Knoll are with fortiss, An-Institut Technische Universität München, Munich, Germany.

A. Tekat is with Technische Universität München, Munich, Germany.

OPC UA features, such as security, binary transport, and SOAP transport, while [11] gives an overview over the features of different OPC UA implementations, including open62541, an open source C (C99) implementation of OPC UA used in our tests. A more detailed comparison between ROS and ROS2, especially considering different DDS implementations such as Connext, OpenSplice, and FastRTPS is evaluated in [12]. They show that using DDS for ROS2 gives a significant performance improvement compared to ROS1.

As of now previous related work mainly focuses on the comparison of one specific middleware, or on the performance evaluation between a subset of the protocols presented in this paper. For our evaluation we focus on some of the most used protocols in Industry 4.0 and present a side-by-side comparison for OPC UA, DDS, MQTT and ROS. We also evaluate the performance of multiple nodes on the same host, showing significant differences between the presented protocols.

III. Middleware Comparison

A middleware provides a communication abstraction layer between different components in a distributed system. It is able to bridge the gap between individual subsystems running on different hardware platforms, operating systems, and programming languages. The development of a distributed system composed of heterogeneous devices from various vendors becomes a complicated task that, among others, must ensure communication between numerous devices. Middlewares enable the developer to manage the complexity of this task by introducing an intermediate software layer that provides a high-level API with an abstraction of the low-level details related to communication and application distribution.

OPC UA (Open Platform Communications Unified Architecture) is a service-oriented machine-to-machine communication protocol mainly used in industrial automation and defined in the IEC 62541 specification. Its main goals are to provide a cross-platform communication protocol while using an information model to describe the transferred data. The various features and components of OPC UA are described in different specification parts released and publicly available by the OPC Foundation[1]. OPC UA is mainly driven by the European manufacturing industry and thus gains more and more importance in that area while also becoming one of the more important protocols worldwide. The major strength of OPC UA is the semantic description of the address space model together with various companion specifications which extend the basic semantic descriptions for various domains like PLCopen, robotics, or computer vision. OPC UA recently released a Publish/Subscribe specification. It enriches OPC UA with a Publish/Subscribe concept similar to DDS, where servers can publish data and clients can subscribe to this data, independent of the data origin. OPC UA Publish/Subscribe does not include any quality of service (QoS) mechanisms itself. In combination with, e.g., Time Sensitive Networking (TSN) on layer 2, OPC UA Publish/Subscribe or MQTT can also support additional QoS principles.

DDS (Data Distribution Service) is a data-centric publish-subscribe middleware for highly dynamic distributed systems. It is standardized by the Object Management Group (OMG)[2]. Compared to OPC UA, DDS is more data centric: data is published into the DDS domain and subscribers can subscribe to data from that domain without knowing where the information came from or how it is structured, as the information package already describes itself. In OPC UA every node is described in the address space of the server. A client can query this information and use it together with the received data. DDS provides an extensive set of QoS parameters, e.g., durability, lifespan, presentation, reliability, and deadlines[3]. Similar to OPC UA, DDS also supports dynamic discovery without a central instance. According to the OMG's website, DDS is one of many protocols used in industry sectors such as railway networks, air traffic control, smart energy, medical services, military and aerospace, and industrial automation[4].

ROS (Robot Operating System) is an open-source software framework originally developed by Willow Garage and supported by the Open Source Robotics Foundation (OSRF) together with a large community[5]. Its main target are research institutes in various areas with a focus on encouraging collaborative robotics software development within a large ecosystem. ROS industrial is trying to extend ROS by capabilities for the industrial manufacturing area. The successor of ROS, ROS2 is currently under heavy development. Instead of using the proprietary message formats from ROS, it is built on top of DDS. As we also include DDS in our evaluation, specifically eProsima Fast RTPS as the default DDS implementation for ROS2, the results from this evaluation can also be transferred to ROS2. A more detailed comparison between DDS and ROS2 and the different DDS implementations will be conducted at a later stage.

MQTT (Message Queuing Telemetry Transport) declares itself as an extremely lightweight publish/subscribe machine-to-machine and Internet of Things connectivity protocol[6]. It is an open message protocol which mainly focuses on a small code footprint and low network bandwidth usage, while handling high latency or bad network connections. Communication between sensors via satellite link is therefore one of its use cases. Since 2013, MQTT is standardized by the Organization for the Advancement of Structured Information Standards (OASIS) as the protocol for the Internet of Things[7]. MQTT uses the concept of an MQTT-Server, also known as a broker, which holds the complete data of all of its

[1]https://opcfoundation.org/about/opc-technologies/opc-ua/

[2]https://portals.omg.org/dds/

[3]http://download.prismtech.com/docs/Vortex/html/ospl/DDSTutorial/qos.html

[4]http://portals.omg.org/dds/who-is-using-dds-2/

[5]http://www.ros.org/

[6]http://mqtt.org/

[7]https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard

communication partners. With this concept, small devices simply report data to the Broker and do not need to store the data themselves. The devices can also be controlled through the broker. The published data is grouped hierarchically similar to DDS and multiple devices can publish to the same topic. MQTT supports some basic QoS to define if and how often a message should be re-sent until it is acknowledged by the broker and if the server should cache topic data.

OPC UA's device-centric approach focuses on device interoperability, where devices may be used in different systems, while DDS focuses on software integration mainly in a single type of system. This may be one of the reasons why OPC UA is more likely to be adopted in the manufacturing industry: a typical production line is built up using many different types of systems, whereas systems like air traffic control come from one vendor and all system components are fine-tuned to each other. The focus of ROS is hardware abstraction, the collaboration between different research groups, and reusability of software components. MQTT's focus is on small code footprint and low network bandwidth, which is one of the reasons MQTT does not offer as many extra features such as Remote Procedure Call (RPC) or Discovery.

Table I gives an overview of the basic features for each of the presented middleware protocols. Every protocol can be used on top of TCP, whereas MQTT does not support UDP communication. In ROS, all clients have to support TCP (TCPROS), optionally they can also support UDP (UDPROS) which is currently only implemented by roscpp. Some DDS implementations also support shared memory (SHM): if multiple nodes are running on the same host, the data is exchanged via shared memory instead of the loopback network adapter. Additionally, DDS standardizes the user API, which means that in theory it should be possible to exchange different DDS implementations without changing the source code. Practically this is hard to achieve since every DDS implementation adds additional specific methods to configure the stack.

All presented middlewares support the Publish/Subscribe (Pub/Sub) pattern. Remote Procedure Calls (RPC) are natively supported by OPC UA and ROS. RPC for DDS is standardized but not implemented in many implementations. MQTT does not support RPC at all.

OPC UA uses the address model to provide data to the clients. This address model includes a semantic annotation of the data which allows clients to automatically infer the meaning of specific data values without previous knowledge. In ROS, DDS, and MQTT, data values are published on specific topics, which means that clients have to know the topic they need to subscribe to in advance. DDS additionally uses domains and partitions for the information scope. The only way to infer some additional meaning is to use the topic name, which is not as powerful as it is in OPC UA. In OPC UA, every node can have multiple typed references to other nodes, which is similar to a triple storage database as it is commonly used for semantic modeling.

Security is an important feature for every middleware which

TABLE I: Comparison of the protocols used in the evaluation and their main features.

	OPC UA	**ROS**	**DDS**	**MQTT**
Communication	TCP, UDP	TCP, UDP	TCP, UDP, SHM	TCP
Patterns	RPC, Pub/Sub	RPC, Pub/Sub	(RPC), Pub/Sub	Pub/Sub
QoS	No	No	Yes	Yes
Authentication	User, PKI	(Mac)	PKI	User, PKI
Encryption	Yes	No	Yes	Yes
Std. API	No	No	Yes	No
Semantic Data	Yes	No	No	No

is used in the domain of industrial automation and IoT. Table I shows that OPC UA and MQTT support authentication via username and password or by using a private key infrastructure (PKI). DDS only supports PKI authentication, ROS only supports authentication via MAC Address using third-party packages. ROS is also the only protocol which does not support application layer encryption. The additional efforts (network overhead, CPU load) for encrypting ROS communication are evaluated in [13].

In our tests, we used the following implementations of each protocol, all of which are available as open source software. Commercial stacks exist for OPC UA, DDS, and MQTT.

- **open62541:** OPC UA implementation. License MPL2.0. Version 0.3-rc2 (for Pub/Sub: current master df58cf8)
- **ROS C++:** ROS implementation. License BSD. Version Kinetic Kame
- **eProsima Fast RTPS:** DDS implementation. License Apache 2, Version 1.6.0
- **Eclipse Paho MQTT C:** MQTT implementation. License EPL1.0. Version 1.2.1

IV. Package Overhead

As a first step, we compared the different protocols from a theoretical point of view by looking at the overhead for each transmitted package. Every protocol needs some kind of package header for each data it transmits as a payload of the TCP package so the remote side knows which kind of data is sent. This package header adds additional overhead to each data message transmitted on the network and thus limits the possible maximal bandwidth which can be reached.

To compare the protocol headers, we had a look at every protocol specification and compared the protocol header for different payload sizes of 0 byte, 100 bytes, 1000 bytes and 10 000 bytes as shown in Table II. The table shows the protocol payload size that is passed from the middleware (OSI layer 5, Session) to the UDP/TCP transport (OSI layer 4). Note, that these values do not include TCP or UDP header sizes and are therefore not influenced by the Ethernet frame size, as the middleware payload size is independent from this if the frames are split at a lower layer. Additionally, we verified the values using Wireshark[8].

For OPC UA, we evaluated the overhead for a variable write request from the client to the server without encryption, which would add additional overhead. In OPC UA, the client first needs to open a secure channel (OpenSecure ChannelRequest, 132 bytes), get the available endpoints (Get EndpointsRequest, 93 bytes), create a session (CreateSession Request, 138 bytes, depending on the hostname, here `localhost:4840`), and then activate the session (Activate SessionRequest, 137 bytes, depending on the identity token length, here `open62541-anonymous-policy`), before a write request can be sent. At the end, the client should close the session (CloseSessionRequest, 75 bytes) and the secure channel (CloseSecureChannelRequest, 57 bytes). The total sum of these TCP payload bytes for opening and closing the connection in OPC UA without the WriteRequest sums up to $132 + 93 + 138 + 137 + 75 + 57 = 632$ bytes.

MQTT is using its own lightweight TCP-based binary protocol. The payload of the connect command to the MQTT broker depends on the publisher name. In our case, we only used one character for this, which resulted in a payload of 15 bytes for the connect command and 2 bytes for the disconnect request. The size of the publish message also depends on the topic name. With one character as the topic name, the overall overhead for connection handling in MQTT sums up to $15 + 2 = 17$ bytes.

In ROS, the first step is to start the ROS core on one machine. Every started ROS node requires the ROS core's IP address. On startup, the node sends XMLRPC requests to the core to exchange information about the current system state and any subscribers and publishers for the node's topics. Only if there is a matching subscriber/publisher pair, the two nodes are connected via a separate TCP connection and the TCPROS protocol specifically for this subscription. On shutdown, the ROS node unregisters itself from the core via XMLRPC[9]. In our test, the overall sum of the TCP package's payload size for outgoing XMLRPC requests—connecting and exchanging information with the core (5693 bytes) and disconnecting at the end (3046 bytes)—was 8739 bytes. This does not include the TCPROS messages that are exchanged between the nodes. On the separate TCPROS channel, the subscriber node connects to the publisher node with additional information about the desired topic. The publisher node then returns its own publishing info of size 176 bytes for setting up the connection. If there is no subscriber for a specific topic, the data published by the node will not be sent through the network at all. The final sum of TCP payload bytes for opening and closing the connection in ROS without published data is $8739 + 176 = 8915$ bytes.

The discovery process in DDS depends on the used implementation stack. For our tests, we used the eProsima Fast RTPS implementation. It uses two phases to discover other nodes in the network: in the participant discovery phase, the node sends out multicast messages to discover other participants in the network and periodically sends heartbeat packages. When two nodes have found each other, they exchange information about published and provided topics with the other endpoint in the endpoint discovery phase. During the endpoint discovery phase, the subscriber and publisher have to acknowledge the other side. Only then can they exchange information. During this whole setup process in our test, the server sent a total amount of 8348 bytes, encapsulated in RTPS (Real-Time Publish Subscribe) packages and delivered as UDP payload. In OpenDDS, another open source DDS implementation used in our tests, the discovery process is implemented by using a central discovery repository. Upon start, the publisher uses IIOP (Internet Inter-ORB Protocol) to exchange information with the directory service (add domain participant, assert topic, add subscription, add publication) on its own publish data and on the current subscribers. This procedure sends a total amount of 13 TCP messages with a sum of 4522 bytes. Unregistering (remove publication/subscription, remove topic, remove domain participant) results in another 1580 outbound bytes. The total sum of outbound bytes without exchanging any messages in OpenDDS is 6102. When a message is published, DDS uses the RTPS protocol, which is sent through UDP unicast in our setting. The RTPS protocol can also be configured to use UDP multicast or TCP.

TABLE II: Package size in bytes transmitted as TCP/UDP payload for each protocol with given payload in bytes. The difference between the final package size and the protocol payload shows the protocol overhead. The last column summarizes the total bytes for the connection setup in the evaluation.

Payload	0	100	1000	10 000	Connection
OPC UA C/S	96	196	1096	10 096	632
MQTT	5	105	1006	10 006	17
ROS	8	108	1008	10 008	8915
DDS	88	188	1088	10 088	8348

Overall, the evaluation shows that MQTT not only adds the smallest amount of additional data during the connection initialization, it also has the smallest overhead when sending out data messages, as can be seen in Table II. OPC UA with Client/Server is in second place when looking at the connection setup overhead, but comes last regarding the message overhead. ROS requires the largest amount of extra bytes for setting up the connection between two nodes. This is due to the fact that it uses XMLRPC, which is sending non-compressed XML text to the ROS core. On the other hand, ROS is very efficient when looking at the payload overhead. DDS has a slightly smaller payload overhead compared to OPC UA and requires a few bytes less than ROS for setting up the connection between publisher and subscriber. It can also be seen, that the protocol overhead is almost constant for all investigated protocols, independent of the payload size. MQTT requires an additional byte for messages larger than 127 bytes: the packet length header field in MQTT is a variable between 1

[8]https://www.wireshark.org/

[9]http://wiki.ros.org/ROS/Technical%20Overview

and 4 bytes (7 bits plus continuation bit) and is adapted to the corresponding payload size.

One important reason for the significant differences between MQTT and ROS compared to OPC UA and DDS lies in the fact, that MQTT and ROS create a dedicated TCP connection for every subscriber and publisher pair. They do not need to include additional information about the published data in the transmitted package. In OPC UA and DDS different kind of data can be sent in the same connection, therefore the payload needs additional identification data.

DDS and OPC UA include the possibility to add additional diagnostics information for every transmitted data package, while for the lightweight protocols (ROS, MQTT) this data needs to be collected separately.

V. Testing Setup

Our testing environment is composed of the following components: a client machine is connected to the tested Linux server machine acting as a remote host for the latency and throughput tests via a gigabit network switch (TP-Link TL-SG1024DE). Both Linux machines have the following specification: Intel i7-8700K CPU with 3.70 GHz and 64 GB RAM running on Ubuntu 16.04.4 with Preempt-RT Kernel 4.14.59-rt37. We used a real-time kernel to ensure high performance and reproducibility on the tests. A basic evaluation of the setup results in an average round-trip time (RTT) ping of 0.35 ms between the two Linux machines. A measurement of the bandwidth using the Linux iperf command resulted in a value of up to 724 Mbit/s.

The evaluated middlewares have different feature sets as described in Section III. To compare the performances of OPC UA, DDS, ROS, and MQTT, we used the request-reply pattern as it can be implemented in all protocols and is an important and often-used feature. In this setup, the server is responsible for providing the methods a client can call. The client is responsible for calling the methods on the server and measuring the server's performance. For MQTT, DDS, and OPC UA Pub/Sub, we implemented a request-reply pattern using two topics: one topic where the data is sent and another topic where the response is returned. All the tests are designed to be reproducible with a single command, which also logs the timing values in a file. Our test suite developed for this evaluation is open source and available on GitHub[10].

To compare the performance and system load during the tests, we employed the following metrics: CPU usage, RAM usage, messages per second, and round-trip time (RTT).

For the tests, we used two distinct modes: the acknowledge (ACK) and echo mode. In the ACK mode, a client sends a request and waits for a simple acknowledgment message (1 byte) from a server. This mode can be used to measure the response time for a single message. In the echo mode, a server replies with the same data bytes, which is useful to measure the throughput of a middleware. Echo and ACK modes are executed one after the other. For each mode, requests are made continuously without waiting in between until 5000 requests are sent. In the next step, the payload size is doubled, starting from 2 bytes and going up to 32 768 bytes in one request. This process is repeated for both test modes (ACK and echo).

In addition to the ACK and echo tests, we also tested the resting RTT. This is the time it takes the server to respond to a request when it is in a resting state. Within 30 seconds, a client waits for a random amount of time between 0 and 3 seconds, sends a single request in ACK mode, and then measures the RTT.

To evaluate the performance in non-ideal scenarios, where other processes on the end-device cause high CPU load or the network is working at full capacity, we used third-party tools to generate extensive network traffic and CPU usage. For network traffic, we used the Ostinato[11] traffic generator, which can create sequential and interleaved streams of different protocols at different rates to generate artificial network load [14]. This results in a full capacity utilization of the network stack, but the test should still succeed. The *stress*[12] tool was used to create artificial CPU load. It is designed to apply configurable CPU, memory, I/O, and disk stress on the system.

This setup results in 7 tests per middleware: echo and ACK when the system is in idle state, echo and ACK for CPU load, echo and ACK for network load, and the resting RTT test.

As a final step, we also evaluated the performance of the middleware when 500 server instances of the same protocol are started at the same time on the server machine. The client machine will then simultaneously send 10 packages with a payload of 10 240 bytes to 10 nodes running on the server machine. These 10 nodes will then immediately forward the received package to the next node on the same machine. The last step is repeated 50 times, which results in 10 sending streams running in parallel. In the last node, the messages are sent back to the client, which will then measure the complete RTT of all 10 messages. The whole process is repeated 100 times to average the results. This test shows the suitability of the protocol for the use case, where a lot of nodes are running on the same host and data is exchanged primarily between these nodes. This setup is often used with ROS, e.g., where a ROS node is reading a camera image, another node subscribes to this image, applies preprocessing, and outputs the result in a new topic.

In the next section, we evaluate some of the most interesting test results and present the performance of each middleware implementation.

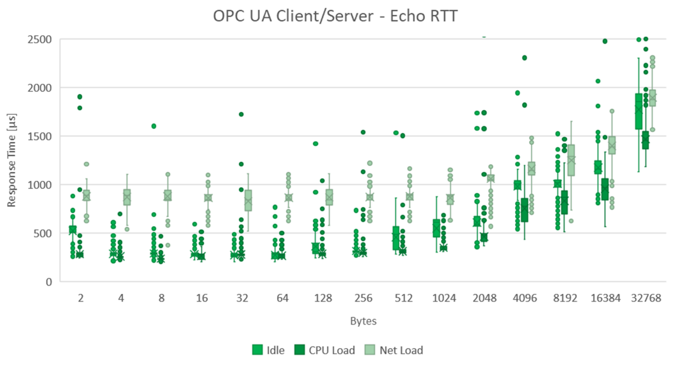
VI. Performance Evaluation

This section presents the results of our tests as described in Section V. Note, that for conducting the tests we are using the most common C/C++ open-source implementations of every protocol. Therefore, the results may also be influenced by the corresponding performance of the implementation itself and may not apply to other implementations of the same protocol.

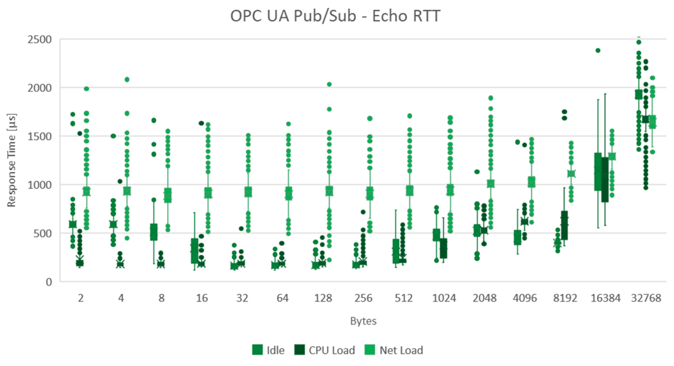
[10]https://github.com/Pro/middleware_evaluation

[11]https://ostinato.org/

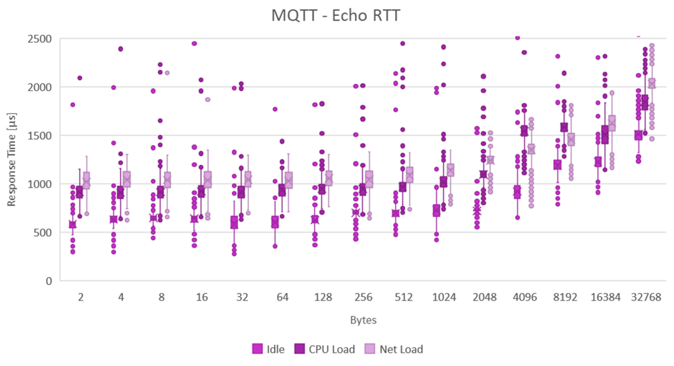
[12]https://people.seas.harvard.edu/~apw/stress/



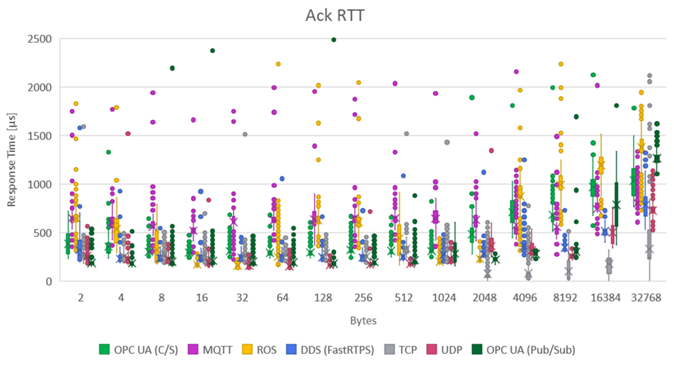
(a) Echo RTT for different data message size using OPC UA Client/Server via TCP on Idle, CPU Load, and Network Load.



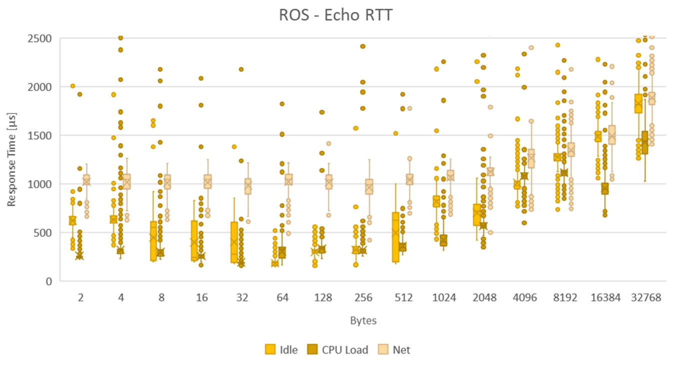
(b) Echo RTT for different data message size using OPC UA Publish/Subscribe via UDP on Idle, CPU Load, and Network Load.



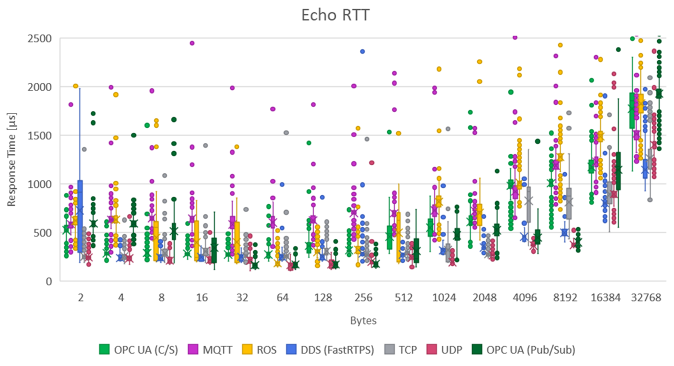
(c) Echo RTT for different data message size using MQTT on Idle, CPU Load, and Network Load.



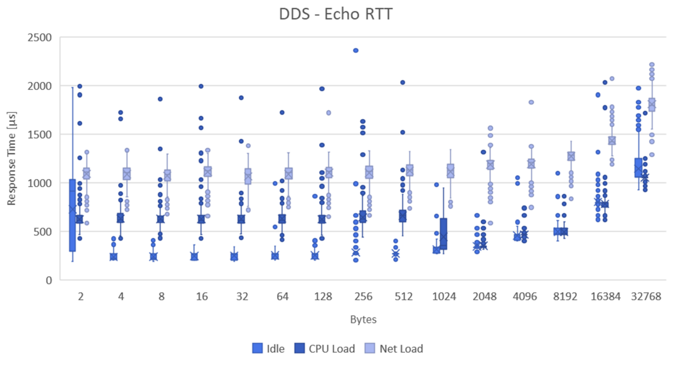
(d) RTT for sending data with a simple ACK message using different protocols.



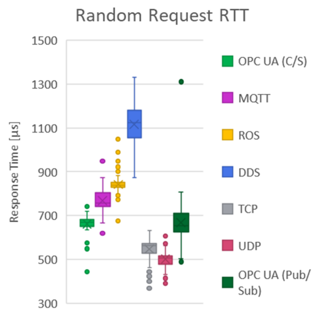
(e) Echo RTT for different data message size using ROS on Idle, CPU Load, and Network Load.



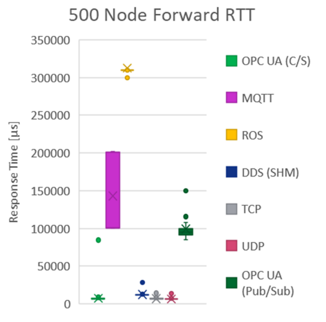
(f) RTT for sending and receiving data messages using different protocols.



(g) Echo RTT for different data message size using DDS on Idle, CPU Load, and Network Load.



(h) Resting RTT for the evaluated protocols sending a payload of 4 bytes waiting randomly between 0 and 3 seconds.



(i) RTT for sending 10 packages with a payload of 10240 bytes through 500 nodes on a different host.

Fig. 1: Plots showing the measurement results for an excerpt of the collected data.

TABLE III: Average RTT in microseconds for echo and ACK mode with various payloads (in bytes) as visualized in Fig. 1.

Mode	Middlware	Load	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16 384	32 768
Echo	OPC UA C/S	Idle	520	283	288	280	270	270	360	313	456	546	606	978	1003	1171	1770
		CPU	279	258	233	260	276	265	284	293	310	346	461	752	830	955	1467
		Net	875	869	876	858	827	864	866	871	875	860	1056	1158	1248	1397	1894
	OPC UA P/S	Idle	590	589	510	338	162	270	169	174	305	469	521	446	395	1142	1924
		CPU	224	177	178	182	183	265	188	206	242	373	529	620	618	1083	1676
		Net	919	930	880	902	911	864	921	901	924	936	1003	1019	1110	1286	1641
	MQTT	Idle	584	629	645	641	581	601	622	699	694	725	716	920	1186	1225	1511
		CPU	912	905	912	917	911	934	945	944	964	1015	1094	1540	1582	1504	1843
		Net	1017	1042	1038	1033	1040	1027	1054	1039	1085	1135	1239	1347	1456	1622	2029
	ROS	Idle	618	628	446	396	398	182	296	319	495	814	698	1007	1273	1486	1823
		CPU	256	320	295	252	190	305	332	317	359	421	563	1077	1114	952	1427
		Net	1017	1014	1009	1012	982	1021	1011	963	1040	1063	1124	1263	1343	1488	1884
	DDS	Idle	726	234	236	243	247	252	251	279	270	314	343	454	495	802	1144
		CPU	624	631	626	625	624	628	622	644	659	442	351	464	489	782	1050
		Net	1091	1086	1070	1111	1063	1088	1102	1099	1123	1114	1184	1190	1270	1432	1806
	TCP	Idle	250	253	284	303	254	255	307	270	289	315	273	823	811	892	1261
	UDP	Idle	241	231	208	213	208	162	168	198	238	193	263	263	268	893	1398
ACK	OPC UA C/S	Idle	383	354	301	294	306	291	293	321	310	305	481	710	681	969	1033
	OPC UA P/S	Idle	185	180	199	207	196	190	180	195	215	279	222	277	298	785	1265
	MQTT	Idle	647	602	556	518	606	626	636	646	650	643	629	575	556	740	800
	ROS	Idle	642	537	321	168	157	248	623	608	437	197	226	871	992	1192	1377
	DDS	Idle	320	228	234	236	241	243	244	242	252	272	275	355	343	514	784
	TCP	Idle	235	243	283	187	239	279	196	208	189	237	58	71	81	167	338
	UDP	Idle	259	223	220	190	155	153	164	169	174	215	359	317	255	502	725

Figure 1 shows an excerpt of some of the most interesting results from our tests. The corresponding numeric values for the average RTT are also shown in Table III. In addition to the protocols described in previous sections, we implemented two simple echo/ACK servers in C that listen for TCP and UDP connections and return the received data to the sender or acknowledge the data by sending one single byte as a response. The results of these raw TCP and UDP implementation are included in the figures and show the best reachable RTT for the corresponding tests without any overhead from a specific middleware implementation.

Figures 1a to 1c, 1e, and 1g show the box plots for the RTT of the message being sent to the server side, which immediately returns the same message for all four protocols. Every diagram shows the values for the system idle state with no specific load on the system or the network. The second value row shows the RTT with 100 percent CPU load on the server and the last row shows the RTT under high network load.

Comparing the values for the server in idle state versus the one with high CPU load, the test results show that the RTT of OPC UA and ROS does not correlate to the CPU load of the host and is nearly the same as in the idle state. Note, that we are using a Preempt-RT patched Linux kernel and the processes are set to the highest priority for all protocols. MQTT and DDS show a significant slowdown of approximately 300 µs compared to the idle state. This leads to the conclusion, that the Paho MQTT and eProsima Fast RTPS implementations require more CPU power to process the messages and are thus slowed down when the CPU utilization is at 100 percent.

The results of the RTT during high network load show that protocols using TCP (OPC UA Client/Server, MQTT) were less influenced by a network interface running at maximum capacity than protocols using UDP (OPC UA Pub/Sub, DDS, ROS). All protocols show a slowdown of more than 400 µs.

The combined view in Fig. 1d and 1f shows the direct comparison of the protocols, including a simple TCP and UDP echo/ACK server implemented in C. It can be seen, that the raw UDP implementation is the fastest way for exchanging messages of small sizes, which are either acknowledged or echoed back, closely followed by TCP, which has a better performance for bigger message sizes. Excluding the results of TCP and UDP, the open62541 OPC UA Pub/Sub implementation described in [15] is the fastest middleware for almost all package sizes independent of the mode. eProsima Fast RTPS is in second place, followed by OPC UA Client/Server, ROS, and then MQTT. The diagrams also show that MQTT has the highest number of outliers.

To investigate further which components of a middleware have the highest impact on the RTT, we also ran tests using the OpenDDS middleware. It features a zero-copy or memory allocation mode for sending back the echo message. This has shown that the way messages are read from the socket and then forwarded to the user code has a high impact on the RTT, since memory allocations are an expensive operation. Additionally, OpenDDS was significantly slower than FastRTPS, which shows that the serialization of messages may also lead to higher RTT. The eProsima Fast RTPS and open62541 OPC UA implementations pass constant data pointers to the user code, whereas MQTT and ROS use multiple copy operations to duplicate the data.

Figure 1h shows the RTT for sequential requests, where the client waited a random amount of seconds (between zero and three) to send one single request with 4 bytes of payload and waited for the acknowledgment messages. This test was repeated 100 times. These results indicate, that DDS needs more time to reactivate from the resting state compared to other middleware protocols. For OPC UA, MQTT, and ROS, the values are similar to the ones in Fig. 1d, which shows the ACK RTT.

As a last test, we also evaluated the case where 500 instances are running on the same host and the instances need to exchange data between each other, as described in Section V. For OPC UA Client/Server, we simply started 500 OPC UA servers on the same host, using different endpoint ports. The client from the remote host sends 10 simultaneous write requests with a payload of 10 240 bytes and every server forwards this request to the next server in line, resulting in 50 write requests per package, multiplied by 10 simultaneous streams. The last 10 servers then return the value to the client, which measures the overall time. For OPC UA Pub/Sub, MQTT, ROS, and DDS, the same procedure is achieved by using topics with different names and starting multiple processes. One of the biggest issues for this test was the resource usage of ROS and DDS. Starting 500 ROS forces the CPU to work at full capacity and filled 60 GB of the total 64 GB of installed RAM on that host. Using Fast RTPS, we were not even able to start more than 100 nodes, as the CPU was already working at full capacity and the DDS nodes were not able to discover any topics after some time. We then used the OpenDDS implementation, which also supports shared memory, to conduct the tests. The high load is the result of the discovery process in ROS and DDS: as explained in Section IV, these two protocols produce a higher number of packages with more data compared to OPC UA and MQTT.

This test shows that the open62541 OPC UA Client/Server implementation is still the fastest protocol and is even faster than the DDS shared memory implementation. The reason for this huge performance gap to MQTT and ROS is the direct TCP connection between nodes in OPC UA, whereas MQTT uses stateless UDP connections. In [12], the authors also state that for small data, shared memory does not improve the latency compared to local loopback. ROS is slowed down by the high CPU load it produces on the host.

VII. Conclusion

In this paper, we gave an overview on the different features of OPC UA, ROS, DDS, and MQTT and compared their performance in several benchmarks. OPC UA has its strength in the semantic modeling of information. ROS is mainly used on robots for research purposes and provides many different pre-implemented feature packages. DDS has an extensive set of Quality-of-Service settings, whereas MQTT mainly focuses on a lightweight publish/subscribe protocol. The performance comparison of the protocol implementations shows that open62541 for OPC UA and eProsima FastRTPS for DDS deliver high performance, whereas the MQTT and ROS implementations show a significant slowdown in the RTT of packages sent to the server.

Future experiments will evaluate the performance of the protocols on bad network connections, for example on a wireless network. Additionally, we will evaluate the performance of different implementations of the same protocol.

References

[1] I. Ungurean, N. C. Gaitan, and V. G. Gaitan, “A middleware based architecture for the industrial internet of things,” *KSII Transactions on Internet and Information Systems*, vol. 10, no. 7, pp. 2874–2891, 2016.

[2] J.-D. Decotignie, “Ethernet-based real-time and industrial communications,” *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1102–1117, 2005.

[3] K. Bauer, B. Diegner, J. Diemer, W. Dorst, S. Ferber, R. Glatz, A. Hellinger, W. Herfs, M. Horstmann, T. Kaufmann, C. Kobsda, C. Kurz, U. Löwen, V. Stumpf *et al.*, “Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0,” Promotorengruppe Kommunikation der Forschungsunion Wirtschaft - Wissenschaft, acatech, Abschlussbericht des Arbeitskreises Industrie 4.0, 2013.

[4] M. Saghian and R. Ravanmehr, “A survey on middleware approaches for distributed real-time systems,” *Journal of Mobile, Embedded and Distributed Systems*, vol. 6, no. 4, 2014.

[5] Y. Chen and T. Kunz, “Performance evaluation of IoT protocols under a constrained wireless access network,” in *Proceedings of the International Conference on Selected Topics in Mobile & Wireless Networking*, 2016.

[6] D.-H. Mun, M. L. Dinh, and Y.-W. Kwon, “An assessment of Internet of Things protocols for resource-constrained applications,” in *Proceedings of the IEEE Annual Computer Software and Applications Conference*, jun 2016, pp. 555–560.

[7] M. S. Rocha, G. S. Sestito, A. L. Dias, A. C. Turcato, and D. Brandao, “Performance comparison between OPC UA and MQTT for data exchange,” in *Proceedings of the IEEE International Workshop on Metrology for Industry 4.0 and IoT*, 2018, pp. 175–179.

[8] J. Pfrommer, S. Grüner, and F. Palm, “Hybrid OPC UA and DDS: Combining architectural styles for the industrial internet,” in *Proceedings of the IEEE World Conference on Factory Communication Systems*, 2016.

[9] A. Veichtlbauer, M. Ortmayer, and T. Heistracher, “OPC UA integration for field devices,” in *Proceedings of the IEEE International Conference on Industrial Informatics*, 2015, pp. 419–424.

[10] S. Cavalieri and G. Cutuli, “Performance evaluation of OPC UA,” in *Proceedings of the IEEE Conference on Emerging Technologies & Factory Automation*, 2010.

[11] H. Haskamp, M. Meyer, R. Mollmann, F. Orth, and A. W. Colombo, “Benchmarking of existing OPC UA implementations for Industrie 4.0-compliant digitalization solutions,” in *Proceedings of the IEEE International Conference on Industrial Informatics*, 2017, pp. 589–594.

[12] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ROS2,” in *Proceedings of the International Conference on Embedded Software*, 2016.

[13] F. J. Rodríguez-Lera, V. Matellán-Olivera, J. Balsa-Comerón, Á. M. Guerrero-Higueras, and C. Fernández-Llamas, “Message encryption in Robot Operating System: Collateral effects of hardening mobile robots,” *Frontiers in ICT*, vol. 5, no. 2, pp. 1–12, 2018.

[14] S. Srivastava, S. Anmulwar, A. M. Sapkal, T. Batra, A. K. Gupta, and V. Kumar, “Comparative study of various traffic generator tools,” in *Proceedings of the International Conference on Recent Advances in Engineering and Computational Sciences*, 2014.

[15] J. Pfrommer, A. Ebner, S. Ravikumar, and B. Karunakaran, “Open source OPC UA PubSub over TSN for realtime industrial communication,” in *Proceedings of the International Conference on Emerging Technologies and Factory Automation*, 2018.