2023-09 ROS 2 RMW alternate

Abstract

The ROS MiddleWare interface (RMW) is an abstraction layer that allows ROS 2 to swap out its underlying communication mechanism (middleware) at both compile time and runtime. All current Tier 1 implementations of RMW are based on DDS. At the time that this solution was chosen, it met many of the initial requirements.

Over the last 8 years of use, based on user feedback a number of problems have been identified, including:

1. The RMW interface is meant to be generic, but because all the Tier 1 implementations are DDS, details about DDS can leak through the interface.

2. DDS has a fully-connected graph of participants, which limits the number of network entities in the system.
3. DDS discovery relies on multicast UDP by default. Many deployment environments do not support multicast UDP, or limit how much of the network can be reached, leading to silent discovery failures. While individual robots often use a single LAN, there are many domains of robotics that need more scaling.
4. DDS can struggle with large messages (images, point clouds, etc.) that are fundamental to many robot software systems.
5. DDS can have difficulty on WiFi networks, particularly when UDP multicast is disabled or network connectivity is spotty. This is a problem since WiFi is required in many robotics domains.
6. DDS can be complex to configure for non-ideal networks.

The previous issues can be addressed, in whole or in part, by expert configuration of DDS and/or vendor-specific extensions to the DDS standard. However, many in the ROS user community believe that an alternate middleware option that "just works" for many robotics applications is worthwhile, even if some corner cases or particular applications are not addressed. For demanding applications, DDS will always exist, and expert tuning will provide the necessary flexibility. For the rest, improving the "out of the box" experience while meeting the system requirements is the guiding design criterion.

This paper will present the challenges of the current system, then derive and propose a set of requirements that a middleware must meet in order to be considered for a new RMW. It will investigate currently available middlewares, and evaluate them against the requirements. The conclusion will identify a single option that the ROS 2 core team will develop into a non-DDS RMW to demonstrate the feasibility.

User Challenges with DDS

ROS 2 has used DDS as the middleware since the beginning of ROS 2 development around 2015. DDS was chosen because it addressed many of the same goals as ROS, and it had a long history of large-scale, mission-critical deployments across a variety of industries.

Thus, the ROS 2 community has 8 years of experience using DDS as the middleware. In that time, a number of issues have emerged as consistent sources of difficulty:

DDS has a fully-connected graph of participants

The graph of "DDS participants" refers to the ROS nodes, publishers, subscribers, service clients, and service servers in the network. In ROS 1, the full graph was only known by **`roscore`**; individual nodes only received the graph edges necessary to establish their

connections. As a result, ROS programs tend to create many topics and services, and expect that unconnected topics and services are inexpensive in terms of CPU usage, memory usage, and network usage.

However, by design DDS creates and maintains a fully connected graph. All participants, topics, and services in the network must be discovered by all other participants. This causes an n^2 amount of network traffic for discovery, and "packet storms" when new participants enter large networks. In contrast to ROS 1, the creation of new topics in the network is relatively expensive, which limits the number of nodes and topics in the network. The problem becomes worse as the network size grows, leading to frustration as roboticists move beyond toy examples and attempt to scale their systems. There are ways to work around this limitation by multiplexing data over a smaller number of topics or by using a discovery server to act as a central discovery service, but a fully-distributed graph is inherent to the design of DDS.

DDS uses UDP multicast for discovery

By default, DDS relies on UDP multicast for discovery. This means that nodes can discover each other without having a centralized discovery service like ROS 1. The idea is appealing; it's one less process to manage. However, this approach is not always viable, because not all networks have UDP multicast enabled.

Many institutionally-managed networks disable multicast UDP for security reasons, or to prevent accidental "network flooding" by misconfigured applications. In addition, large WiFi networks often disable multicast by default for performance reasons, because there is no physical "multicast" in modern WiFi connections and the behavior must be emulated by the wireless controller. Further, even for networks that do have UDP multicast enabled, the network segments that UDP multicast can traverse is often limited. This means that it can be difficult to configure DDS discovery to work across complex network setups.

The lack of multicast can be worked around by reconfiguring the network to support multicast in a particular subnet, or by configuring DDS to use a predefined list of peers, or by using a discovery service. However, the failure mode is "silent": participants simply don't find each other. This is challenging for new and experienced users alike, and requires them to learn a considerable amount about DDS discovery and multicast UDP.

DDS can have difficulty transferring large data

According to the standard, DDS uses UDP as the default underlying network protocol for packet delivery. This has some advantages, in that it allows DDS to have total control of many aspects of message delivery (also referred to as “Quality of Service”). This means, for instance, that users can configure certain messages to be best-effort while other ones are reliable.

However, UDP is not nearly as ubiquitous or well-tuned as TCP in modern computing. Because most computing applications are TCP-based, enormous effort has been spent at all levels of the technology stack to optimize TCP performance, including operating system kernels and network stacks, chipset accelerators, network switches, and routers at both LAN and WAN scales. In contrast, UDP requires careful application-specific configuration. The Linux kernel, in particular, has very small default buffer sizes for UDP (typically around 256 KB). Many DDS implementations also request small UDP buffers by default. This is insufficient for the range of message sizes that are seen in robotics applications: for example, when trying to transfer large images or point clouds with DDS, the kernel and userland UDP buffers can fill up. This is particularly problematic on WiFi networks where the connection may be unreliable. In that scenario, a chunk of data may be retransmitted over and over, but not have enough space in the receiving system's UDP buffers. This issue can be worked around by configuring the DDS layer to ask for larger buffer sizes and tuning the kernel UDP buffer sizes. However, this requires users to learn the nuances of configuring systems for demanding UDP applications, and sometimes to use low-level network diagnostic tools such as Wireshark.

DDS can struggle on some WiFi networks

In general, making DDS work on WiFi can be challenging. If the WiFi network allows UDP multicast, and the connection tends to be good, DDS will operate fairly well. But if either of those conditions is not true, DDS can struggle to deliver data. Since ROS 2 is very often used on mobile robots and on laptops used for debugging, it is imperative that it works well "out of the box" for as many networks as possible.

DDS tends to have complex tuning parameters

As mentioned in several of the items above, it is often possible to tune DDS to work in a variety of environments. The various DDS implementations have many configuration points. But it can be overwhelming for users, especially new users, to figure out which tuning parameters they need and for which reason. This leads to user frustration since the tuning parameters that work in one network may not work well for another one. This could be mitigated with more documentation or by exposing more of the tuning parameters from ROS itself. However, that is moving the complexity around rather than addressing it directly. ROS 1 users have come to expect a minimal amount of configuration and maintenance burden; tuning complex network parameters is unfamiliar to most ROS users.

Vendor specific non-standard DDS extensions

As mentioned above, many DDS vendors have developed custom extensions and tools which can be used to work around some of the challenges identified above. These vendor-specific tools are outside the DDS standard, add more complexity for the end user, and potential differences between vendor-specific implementations must be learned (for example, several

vendors offer a DDS discovery service). Additionally, some of these tools are proprietary, which means that they create vendor lock-in and cannot be used in an open-source framework like ROS 2.

Next Steps

The DDS stack works well when it is carefully tuned and operated on a well-managed network, as evidenced by the successful use of ROS 2 in mission-critical systems around the world. The issues described in the previous sections are surmountable in any particular deployment, but they often require expert application-specific DDS configuration.

Given the problems identified above, and the availability of several new middlewares since 2015, now is a good time to explore other options. The main goal is to create an RMW alternative that has a great “out of the box” experience for users, while still retaining the DDS middleware for those with specific needs in demanding applications. The RMW interface will allow ROS 2 to support both the existing DDS middlewares, as well as a new alternate "general-purpose" middleware.

Requirements gathering

The core team gathered a list of requirements for the ROS 2 transport/middleware. These requirements came from three key sources: known use cases of ROS 2, targeted interviews with key stakeholders in the ROS 2 community, and a user survey sent out to the ROS 2 community at large.

User Survey

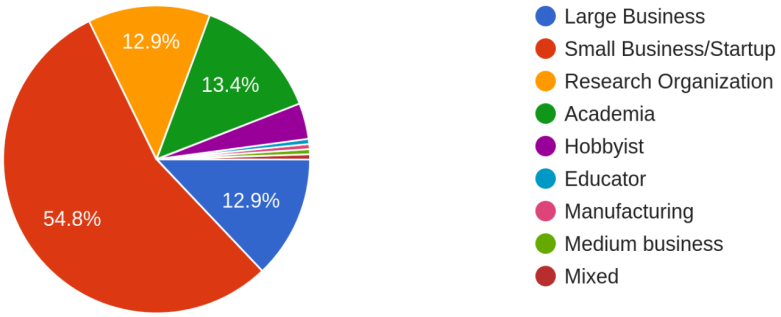
On July 31, 2023, the ROS 2 core team sent a discourse post stating the intention to create an alternative RMW implementation, and asking the community for feedback in the form of a survey.

The survey consisted of a number of questions that aimed to determine how people are using ROS 2, the networks that are in use, issues faced while using ROS 2, and alternative middlewares to explore. Over 180 community responses were recorded from that survey. For the sake of brevity, not all questions asked on the survey will be discussed here; the entire anonymous survey results are available here.

Demographics

Which best describes your organization?

186 responses

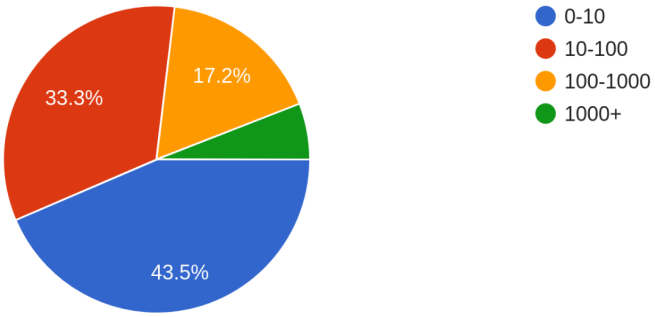


The majority of the respondents were from small businesses, with a pretty even mix of large businesses, medium businesses, and research organizations making up most of the rest.

Technical Data

What is the scale of your robotics application in terms of number of robots?

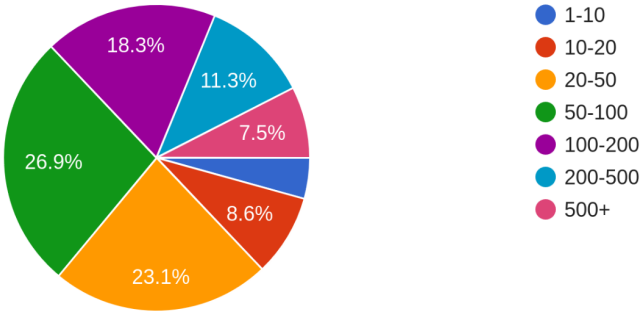
186 responses



Almost half of the respondents have small fleets of robots, less than 10. Exactly half of the respondents have between 10 and 1000 robots, with a few respondents having much larger fleets.

For your application, what is the typical number of topics in your network?

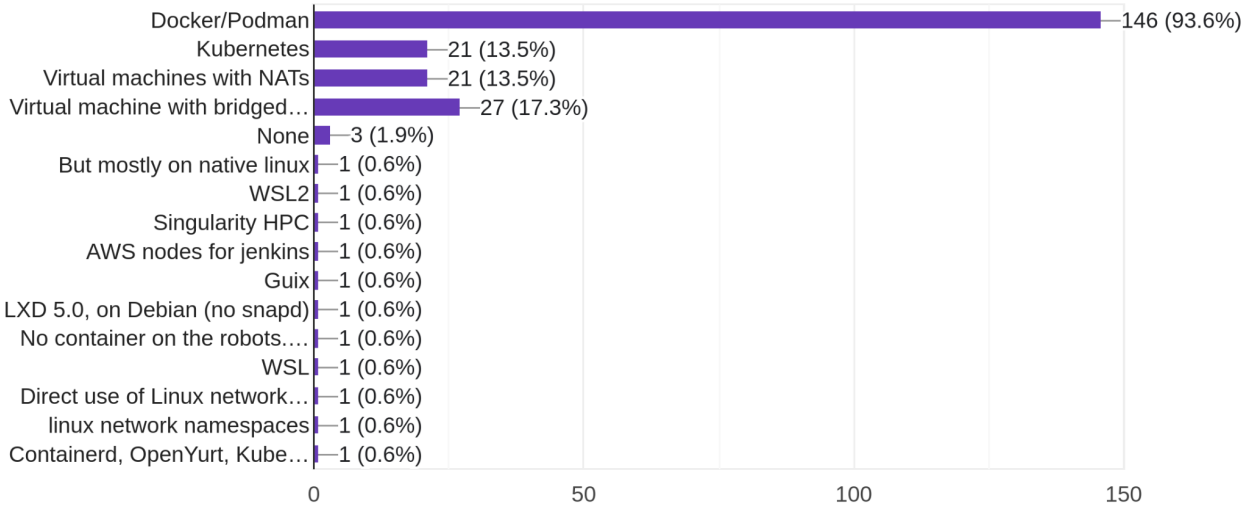
186 responses



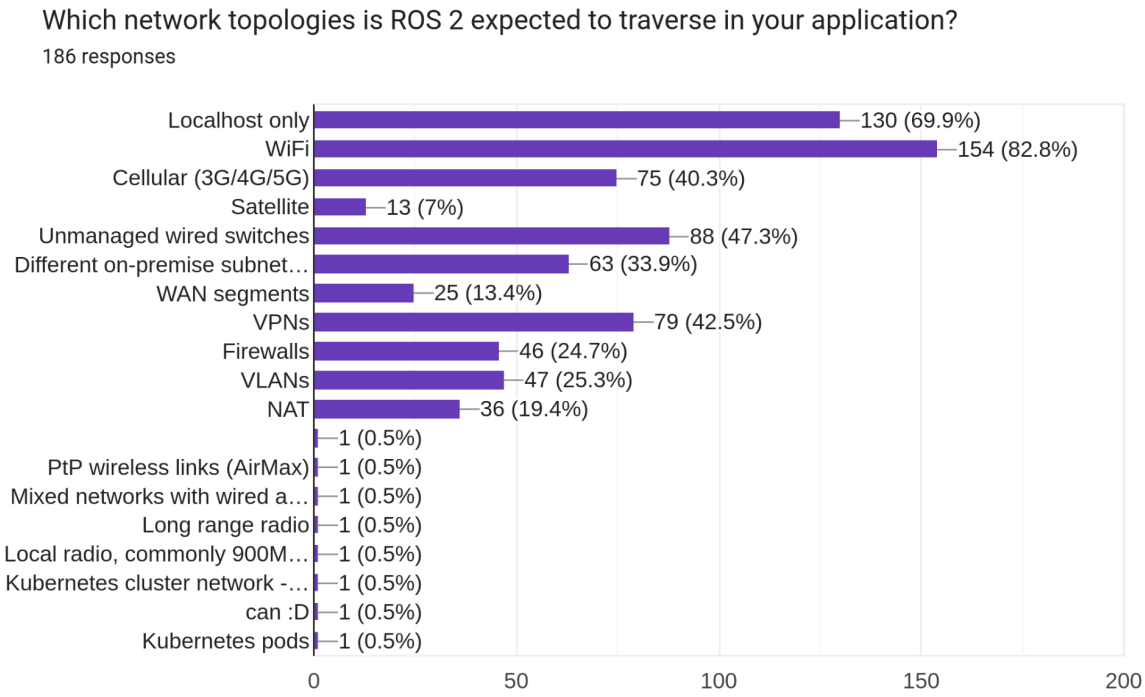
The typical number of topics is dominated by the range 20-200, with a significant portion above 200.

Which container technologies do you use?

156 responses



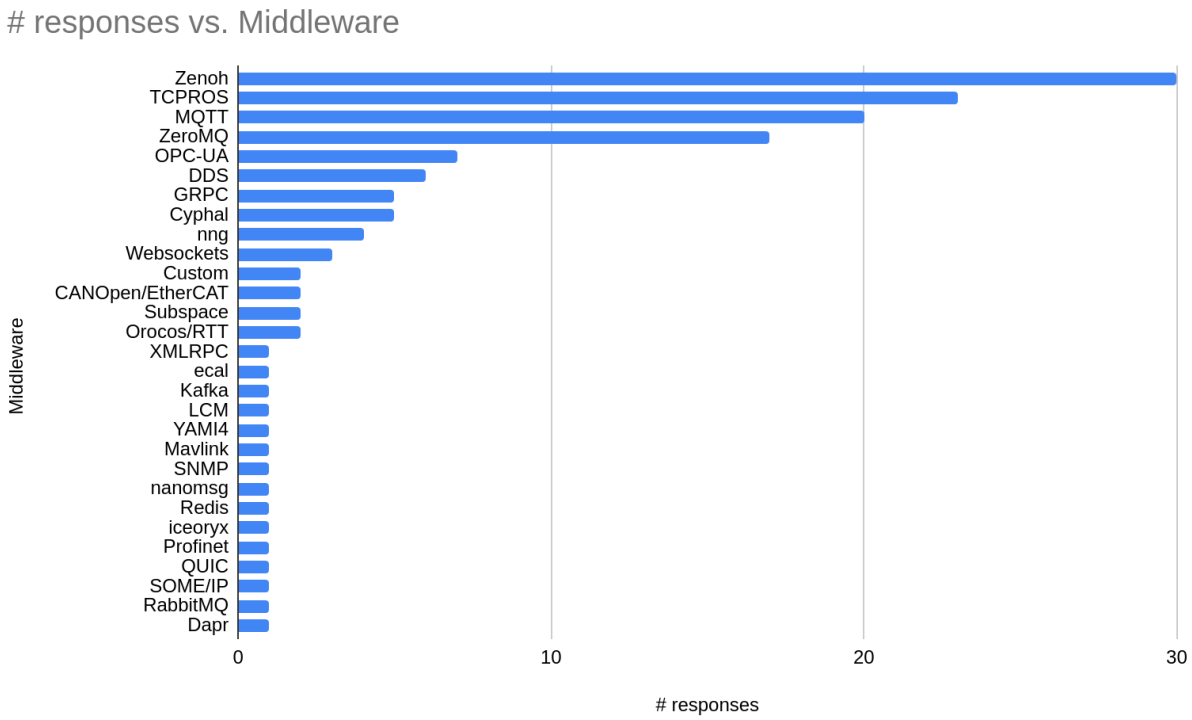
The vast majority of the respondents use Docker or Podman for their robots.



There is no one clear dominant networking topology. Respondents reported things ranging from localhost-only to the cell network to WiFi to VPNs.

Alternative middleware options

The survey included a free-form field to suggest middlewares that should be investigated. Since this was free-form, respondents were free to cite as many as they wanted, and thus the data below contains more responses than the number of respondents.



Zenoh was the alternative most suggested by users, though there was substantial support for TCPROS, MQTT, and ZeroMQ.

Requirements

Based on the above requirements gathering, a list of requirements was produced. In the list below, the standard RFC 2119 terminology of “Must”, “Should”, and “May” is used.

Requirement	Level	Notes
Pub/Sub	Must	It is expected that peer-to-peer comms is necessary for performance, but everything is on the table. (As an alternative, for example, could have an MQTT router on every host and shared-memory within each host)
Security - Encryption	Must	For example, TLS.
Security - Authentication	Must	For example, peers use certificates during connection that were generated by some application-specific certificate-authority.
Security - Access Control	Must	Exact definitions may vary, but the requirement is for per-identity access control that is more granular than just a global allow/deny. As in, identity A may publish on topic B but not on topic C.
Gracefully handle disconnect/reconnect	Must	WiFi. Need graceful, fast recovery of pub/sub/services when connection is re-established.

of the network		
Tolerance to bandwidth changes	Must	WiFi connections can dynamically change bandwidth by order(s) of magnitude.
Configure which interface to use	Must	Plenty of scenarios need to use a specific network interface (virtual networks, routing).
Ability to send multi-megabyte messages	Must	Need to send images, pointclouds, etc., usually < 30 Hz for the "heavy" sensors, with reasonable CPU usage especially on lossy networks.
Ability to send fast small messages	Must	Things like robot state updates from real-time subsystems, say 1 kHz, <1 kB messages.
Restart discovery without restarting all nodes	Must	For safety critical scenarios, there can't be a single point of failure that requires a full reboot.
Cross Platform support	Must	The middleware must be supported on the ROS 2 Tier 1 target platforms: Ubuntu Linux amd64 and arm64, and Windows.
OSI approved permissive license	Must	The middleware must use an OSI approved license, and the license must be permissive (not copyleft-style).
Built-in Discovery	Should	The discovery system can be either built-in or added outside the pub/sub middleware, if enough hooks are present to allow this.
Routing across subnets	Should	It's common to have subnets onboard robots, and offboard subnets for debugging or heavy processing. If an option doesn't have this, then something additional needs to provide this (probably a TCP stream).
Shared memory (intra-host)	Should	Some common robotics pub/sub needs (camera images, point clouds) are gigabit+ per second, and with "heavy" usage of USB3 and 10G sensors, sensor bandwidth requirements keep growing.
Peer-to-peer data connections	Should	A brokered-like network tends to amplify the amount of network bandwidth needed, as well as increase latency.
RPC	May	If a middleware doesn't have built-in async RPC, another solution needs to be found to implement it.
Protocol debugging tooling (CLI, GUI, Wireshark plugins, etc)	May	Ability to debug the network protocol.
Prioritization of message streams (small high-priority messages can interrupt large lower-priority messages)	May	Want to be able to express that some streams are more important (for example, joystick teleop commands), when bandwidth is insufficient for everything.
Control over reliability QoS	May	In some scenarios (e.g., sensors) the most recent message is more important than getting all messages.

Control over QoS history	May	Need to gracefully configure what to do with slow subscribers or slow networks.
Control for "latching"	May	ROS 2 graphs often have late-joining subscribers, so there needs to be some way for publishers to "remember" past data to deliver to them.
Ability to configure static peers	May	In a large network, may only want to do discovery with certain peers.

Comparative analysis of currently available middlewares

Besides the requirements, the ROS 2 core team examined currently available middleware alternatives.

Complete list of investigated middlewares

Here is the complete list of investigated middlewares, with some minimal information about each one:

Name	License	Existing RMW implementation?
Eclipse Cyclone DDS	Eclipse Public License 2.0	https://github.com/ros2/rmw_cyclonedds/
eProsima Fast DDS	Apache 2.0	https://github.com/ros2/rmw_fastrtps
RTI Connext	Proprietary	https://github.com/ros2/rmw_connextdds
Gurum DDS	Proprietary	https://github.com/ros2/rmw_gurumdds
Open DDS	OpenDDS license - Open source-ish, but not OSI approved	https://github.com/OpenDDS/rmw_opendds

Zenoh	Apache 2.0 and Eclipse Public License 2.0	https://github.com/atolab/rmw_zenoh
Zenoh-Pico	Apache 2.0	
MQTT	Implementation dependent, see the wikipedia article	
ZeroMQ	MPL 2.0	
nanomsg	BSD-ish	
nng	MIT	
LCM	LGPL	
IceOryx	Apache 2.0	https://github.com/ros2/rmw_iceoryx/
OPC-UA	An MIT-GPLv2-RCL combination See OPC Licenses Open-source implementation in C: https://github.com/open62541/open62541 C++ impl: https://github.com/FreeOpcUa/freeopcua	
eCal	Apache 2.0	https://github.com/eclipse-ecal/rmw_ecal
ROS 1	BSD / Apache	
Kafka	Apache 2.0	
GRPC	Apache 2.0	
Websockets	Implementation dependent	
Subspace	Apache 2.0	
XMLRPC	Implementation dependent	
CAN bus	Implementation dependent	

YAMI4	GPLv3	
SNMP	Implementation dependent	
Mavlink	LGPLv3	
Cyphal (libcanard)	MIT	

Performance

Detailed networking performance was a deliberate omission from the research into available middlewares. Previous experience with performance testing (from the Galactic and Humble middleware selection) has shown that it is extremely time-intensive, and can only give gross differences in performance. The configuration of the machines under test, the network, and the configuration of the middleware play a huge role in determining the fine differences in performance. In ideal conditions, all of the further investigated options are capable of saturating a gigabit network link; the differences primarily rest in ease of middleware configuration for performance across a wide range of applications.

Third parties have performed detailed performance analyses of some of the middlewares:

- Zenoh vs MQTT vs Kafka vs DDS
- Fast-DDS vs CycloneDDS
- Fast-DDS vs ZeroMQ
- Kafka vs ActiveMQ vs RabbitMQ

While the data in the above analyses don't objectively prove the performance, they are a representative sample of why performance testing is difficult to do.

Middlewares X Requirements

From the complete list of middlewares above, a subset that seemed most promising was examined in more detail. In particular, each of the middlewares in that subset were compared against the list of requirements presented in the earlier section.

The resulting spreadsheet is in Appendix A, or is available in an easier-to-read format at https://docs.google.com/spreadsheets/d/18SgD-aFJAiDus5cYMN-ya9greOgXq-ErVEsWuV40Kjw/edit#gid=1189402529. Note that the columns are in alphabetical order.

Some key takeaways of the spreadsheet:

- Zenoh meets most of the requirements. For those requirements it does not currently meet, either a feature is in development, or can be developed using existing mechanisms.
- TCPROS is the underlying communication mechanism in ROS 1. Since it was specifically designed to meet the requirements of a robotic application, it too meets most of the current requirements.
- MQTT meets a number of the requirements, and is heavily used in the IoT world. The size limitation on messages, along with the fact that it is fully brokered, doesn't seem to fit the ROS use case very well.
- ZeroMQ (and its grandchild, nng) meets a number of requirements, and is in active use by Gazebo. Because ZeroMQ is more of a "toolbox" of networking primitives, it would require significant additional development to create a fully-featured middleware.
- OPC-UA meets a number of requirements, but is brokered architecture and doesn't have discovery by default.
- DDS is currently being used by ROS 2, and it meets most of the requirements, though with the problems pointed out earlier in this paper.
- Kafka is an interesting and widely used message service, but is fairly complex and the messaging model doesn't directly map to ROS.

Conclusion

Requirements from ROS 2 users were gathered, and middleware options that are available today were investigated. The research has concluded that Zenoh best meets the requirements, and will be chosen as an alternative middleware. Zenoh was also the most-recommended alternative by users. It can be viewed as a modern version of the TCPROS implementation, and meets most of the ROS 2 requirements. There are still a number of design decisions to be made regarding this implementation; those details will be discussed on https://discourse.ros.org as development begins.

Appendix A

Requirements		Middleware Options										
Item	**Level**	**Cyphal**	**DDS**	**eCal**	**Kafka**	**LCM**	**MQTT**	**NNG**	**OPC-UA**	**TCPROS**	**Zenoh**	**ZeroMQ**
Pub/Sub	Must	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes but requires a special broker	Yes	Yes	Yes
Security - Encryption	Must	No	Yes	No	Yes https://kafka.apache.org/documentation/#security	No	Yes, TLS	Yes, TLS	Yes	Via SROS	Yes, TLS	Not in the default, but CurveMQ builds on top to provide it
Security - Authentication	Must	No	Yes	No	Yes	No	Yes	Yes	Yes	Via SROS	Yes	Not in the default, but CurveMQ builds on top to provide it
Security - Access Control	Must	No	Yes	No	Yes	No	Yes	No	Yes	Via SROS	No, planned for later this year	No, but can be implemented on top

Gracefully handle disconnect/reconnect of the network	Must	Yes	Borderline; in UDP mode, packets may be dropped. If the connection is reliable, data will be resent, but may be fragmented and subject to kernel limits. In best-effort mode, data will be dropped.	Borderline; in UDP mode (the default), packets will be dropped. In TCP mode, it will handle this better, but this is not the default	Yes. Handles failures well too https://kafka.apache.org/documentation/#basic_ops_restarting	Yes (connectivity will recover)	Yes	Yes	Yes	Yes	Yes	Yes
Tolerance to bandwidth changes	Must	Unclear	Borderline: without flow controllers, can struggle to send data if the link degrades.	Borderline; in UDP mode (the default), packets will be dropped. In TCP mode, it	Yes. Esp due to writing to disk	No built-in flow control	Nothing on top of what TCP offers	Nothing on top of what TCP offers	Nothing on top of what TCP offers	Nothing on top of what TCP offers	Yes	TCP

				will handle this better, but this is not the default								
Configure which interface to use	Must	Yes (IP address)	Yes	Yes (via standard Linux configuration)	Yes	Yes	Yes, implementation dependant: See mosquitto	No, there doesn't seem to be an API to indicate interface	Yes	Indirectly through the routing table	Yes, configuration variable	Yes (both tcp and udp during zmq_bind())
Ability to send multi-megabyte messages	Must	Yes (with UDP transport options)	Borderline; in UDP mode, packets are heavily fragmented and sent. Without configuring kernel buffer sizes, very possible to lose some packets along the	Yes, but with caveats depending on which transport layer is in use: - SHM: Yes - UDP: same issues as DDS - TCP: Yes	Yes	Yes	To the limit: 8mb - 10hz (from the Zenoh performance comparison at https://zenoh.io/blog/2023-03-21-zenoh-vs-mqtt-kafka-dds/)	Yes	Yes	Yes	Yes, relying on this analysis	Should be since TCP, see https://github.com/jeffbass/imagezmq#why-use-imagezmq

			way and thus have to retransmit a bunch.									
Ability to send fast small messages	Must	Yes (with UDP transport option)	Yes	Borderline; particularly with the SHM transport (default for localhost), if the publisher is running faster than the subscribers, data will be lost.	Yes. Apparently latency of 2ms is achievable	Yes	Yes	Yes	Maybe not	Yes	Yes, relying on this analysis	Yes
Restart discovery without restarting all nodes	Must	Yes since the channels are statically configured.	Yes	Yes (UDP multicast discovery)	Yes	Yes	Yes	N/A (discovery is not builtin)	N/A	No	Yes: Discovery happens when a zenoh session is created in each applicat	N/A

											ion. It looks for other peers/routers.	
Cross platform support	Must	Linux, Windows, macOS	Implementation-specific	Linux, Windows, macOS	Java, so in theory runs anywhere a JVM does	Linux, Windows, macOS	Implementation-specific	Linux, Windows, macOS	Implementation-specific	Linux, Windows (third-party), macOS (third-party)	Linux, Windows, macOS	Linux, Windows, macOS
OSI approved permissive license	Must	MIT	Implementation-specific	Apache 2.0	Apache 2.0	LGPL	Implementation-specific	MIT	Implementation-specific	BSD	Apache 2.0 and Eclipse Public License 2.0	MPL 2.0
Built-in Discovery	Should	No	Yes	Yes (UDP multicast)	No	No (pre-defined UDP mcast)	No (need to provide host:port to every client)	No (though survey protocol may be able to do this in the future)	No	Yes	Yes (both gossip and UDP mcast)	No (need to provide host:ip to connect to)

Routing across subnets	Should	No	Borderline; possible, but usually fraught with problems when hitting real-world managed networks, (TTL, multicast UDP, Wifi)	Challenging by default, since uses multicast UDP. Can use TCP to avoid this.	Yes	Challenging in practice; easy to overload switches with mcast UDP	Yes, via connected brokers	No	Yes, as TCP allows.	Yes, as TCP allows.	Via a zenoh router	Yes, provided networks are bridged
Shared memory (intra-host)	Should	No	Yes (implementation-specific, but all current DDS vendors have it)	Yes	Yes. Msgs are always written to pagecache then disk https://kafka.apache.org/documentation/#design_filesystem	No	No	Yes	No	No	Experimental	No

Peer-to-peer data connections	Should	Yes	Yes	Yes (UDP multicast)	Not directly. But possible via distributed brokers, replication and load balancing https://kafka.apache.org/documentation/#intro_concepts_and_terms https://kafka.apache.org/documentation/#design_loadbalancing	Yes (UDP multicast)	No	Yes	Yes	Yes	Yes	Yes
RPC	May	Yes	Not built-in	Yes (TCP)	Yes	Not built-in	Not built-in	Yes, non blocking call available	Yes	Yes (sync)	Yes (async)	Yes, http://zguide2.zeromq.org/py:asyncsrv

Protocol debugging tooling (CLI, GUI, Wireshark plugins, etc)	May	Yes	Yes (implementation specific)	Yes (monitor, recorder, player)	Yes	No	Yes (implementation specific)	No	Yes	Yes	Yes	Not out of the box. But API available. http://api.zeromq.org/4-1:zmq-socket-monitor
Prioritization of message streams (small high-priority messages can interrupt large lower-priority messages)	May	Yes	Yes	No	Need to implement at application layer; see this blog post	No	No	No	Yes, using ProrityLabel values	No	Yes, when declaring a publisher a priority value is used. See options When dealing with "large" payloads it priotizies the messages carrying the splitted payload	No

											so all the parts are recevied earlier in the other end.	
Control over reliability QoS	May	No	Yes	Yes, but only when using UDP mode	Yes	No	Yes, see here	No. Quote: "*Applications that require reliable delivery semantics should consider using nng_req(7) sockets, or implement their own acknowledgement layer on top of pair sockets.*"	No	No	Yes, see here for reliability and congestion control	No, guaranteed once and only once

Control over QoS history	May	No	Yes	Yes	Yes	No	Yes, in the broker side	No	If using OPC HDA	No	Yes, there is a reliability queue in sender and receiver. When full you can choose to drop messages or block sender/ receiver.	Yes, you can configure queue size for a socket
Control for "latching"	May	Not built-in	Yes	Not built-in	Yes	Not built-in	Yes	Not built-in feature	Not built-in	Yes	Not built-in	Not built-in
Ability to configure static peers	May	No	Yes	Yes (via standard Linux tools)	Yes	Yes	Yes	Yes (no discovery is provided)	Yes	Yes	Yes, discovery for a session can be disabled and a list of nodes address	Yes, it's the only way

											es can be explicitly submitted. See zenoh::config	
	Legend:											
	Green - Meets requirement											
	Yellow - borderline, or needs work to meet requirement											
	Red - Show-stopper, would need redesign											