JOHNS HOPKINS UNIVERSITY

DDS Middleware for Medical Robotics

Contents

INTRODUCTION

BACKGROUND

Data Distribution Service (DDS) is a real-time, brokerless middleware standard designed for distributed systems that require low-latency, high-reliability, and scalable communication. By enabling direct peer-to-peer communication through the Real-Time Publish-Subscribe (RTPS) protocol, DDS avoids centralized bottlenecks and supports fine-grained Quality of Service (QoS) control for deterministic performance.

The da Vinci Research Kit (dVRK) currently relies on the cisstMultiTask software framework, which manages modular, real-time execution through its core building block, *mtsComponent*. [1] This component-based design enables structured data flow and command exchange through well-defined interfaces. For interprocess and interdevice communication, the system uses the Robot Operating System (ROS) via the *mtsROSBridge*. However, the ROS middleware layer introduces communication latency, system overhead, and limited scalability, making it less suitable for the stringent real-time demands of surgical robotics.

GOAL

As shown in *Figure 1*, the goal of this project is to replace the ROS-based middleware in the dVRK software stack with DDS, enabling more efficient and deterministic interprocess communication. This requires developing a DDS communication layer that conforms to the existing *mtsComponent* architecture in cisstMultiTask, which promotes modularity, thread safety, and real-time responsiveness through its interface-based design. Preserving this architecture is essential to ensure seamless integration and consistent system behavior.

Beyond internal communication, the project also aims to retain compatibility with the broader ROS 2 ecosystem. Tools such as RViz, Gazebo, and MoveIt rely on standardized ROS 2 naming conventions, and since ROS 2 itself is built on top of DDS providers (e.g., RTI Connext, Cyclone DDS), aligning DDS message definitions with ROS 2 conventions allows for interoperability. This hybrid approach enables the system to benefit from the performance of native DDS while continuing to leverage the tooling and visualization capabilities of ROS 2.

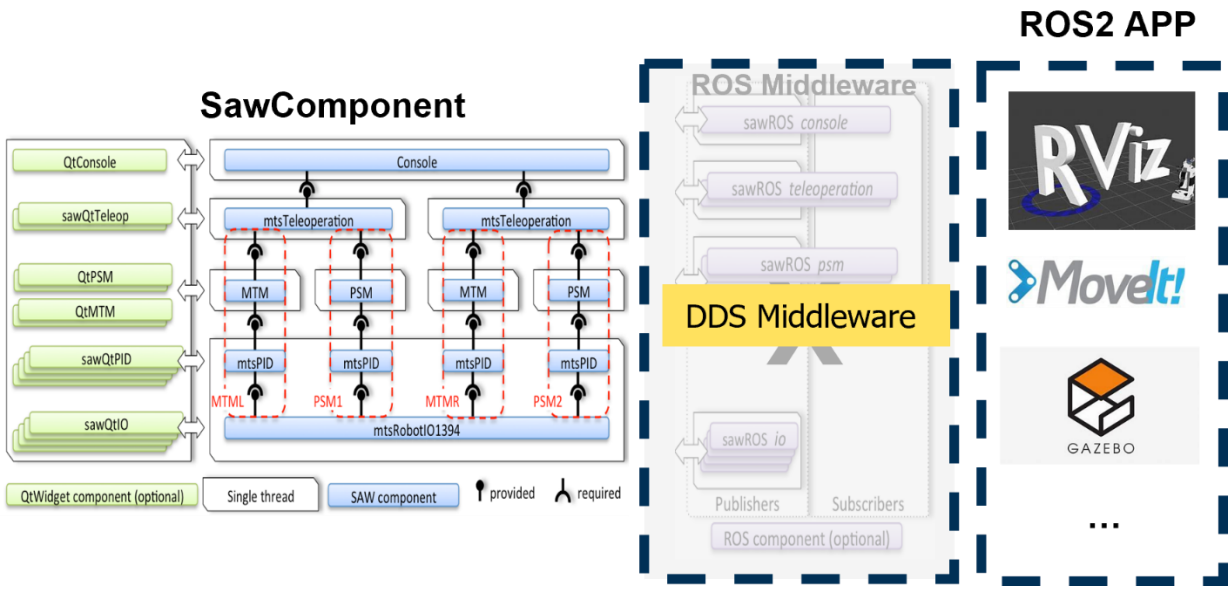

Figure 1. dVRK Software Architecture. The IPC/IDC communication is achieved by ROS which also allows integration with popular tools such as RViz, Gazebo, and MoveIt. The new DDS middleware is proposed as the replacement for ROS Middleware.

SIGNIFICANCE

Replacing ROS's default middleware with a native DDS-based solution brings several key advantages to real-time robotic systems, particularly in surgical applications where performance and determinism are non-negotiable.

First, DDS offers ease of implementation and deployment. ROS 2 installations often involve complex dependencies, fragile package versions, and lengthy setup processes—especially when building from source or enabling real-time capabilities. In contrast, DDS implementations such as RTI Connext DDS are packaged as standalone middleware libraries that are easier to integrate into custom robotic software stacks without needing to install the full ROS ecosystem.

Second, DDS provides broader platform compatibility. Although ROS 2 supports multiple operating systems in theory, its real-time support, tooling, and package stability are largely optimized for Linux—often limiting deployment options for embedded systems, real-time operating systems (RTOS), or Windows-based environments. In contrast, DDS middleware like RTI Connext DDS is designed for seamless deployment across Linux, Windows, QNX, VxWorks, and other platforms, making it a better choice for cross-platform and production-grade deployments.

Third, DDS enables superior performance and deterministic communication. Studies have shown that DDS can outperform traditional message brokers and ROS 2's RMW by over 50% in terms of latency[2], which is essential in applications requiring microsecond-level responsiveness. DDS's peer-to-peer architecture eliminates centralized bottlenecks, ensuring scalable, real-time data flow with predictable timing.

Finally, RTI Connext was selected for this project not only for its high performance and reliability but also because it offers a free academic license and a rich ecosystem of development tools. These include traditional utilities like the Admin Console, Monitor, and Recording Service for real-time introspection, as well as modern AI-powered features that significantly accelerate development. For instance, developers can now integrate Connext with GitHub Copilot in Visual Studio Code, enabling automatic code generation, intelligent error detection, and contextual assistance. RTI's AI-powered System Designer allows users to prototype and configure DDS XML profiles using natural language, while the Connext AI Chatbot provides real-time, product-specific help with QoS tuning, code validation, and system design. These features not only streamline the development workflow but also lower the learning curve for new users—making DDS a more accessible and scalable solution for building complex, real-time robotic systems.

Together, these advantages position DDS not only as a technically superior middleware for high-performance medical robotics but also as a practical and scalable solution for long-term, cross-platform deployment.

TECHNICAL APPROACH

DESIGN OVERVIEW

To integrate DDS into the dVRK communication framework, I developed a new component, *mtsDDSBridge*, based on the cisstMultiTask architecture as shown in *Figure 2*. The system bridges CISST components with DDS-based interprocess communication (IPC), replacing ROS-based IPC while maintaining compatibility with existing ROS-based applications. Hence, legacy ROS scripts remain fully compatible after this upgrade.

The *mtsDDSBridge* class inherits from *mtsTaskPeriodic*, enabling it to operate as a thread-safe component with built-in support for real-time execution, connection to other CISST components,

and runtime services such as logging and debugging. This design ensures that DDS communication can be seamlessly embedded into the existing dVRK software infrastructure.

Another core function of the bridge is data type conversion. Since CISST uses its own encapsulated data types that wrap standard C++ types, the *mtsDDSBridge* implements conversion routines that translate these types into DDS-compatible and ROS-compatible formats. Crucially, the conversion preserves the data timestamps, which are essential for functions or applications that process data from different sources. To facilitate interoperability with ROS, the system adheres to ROS-compatible topic naming and message type conventions. As a result, real-time joint position updates from the robot can be visualized in RViz, bypassing the ROS middleware entirely.

From a user perspective, the implementation of *mtsDDSBridge* brings significant flexibility and usability improvements. Users can now choose to access the dVRK system either through standard ROS nodes or by creating native DDS nodes. Plus, users can greatly benefit from the AI-assisted tools to create their DDS application framework using System Designer and Connext Chatbot, allowing them to concentrate more on the core code.

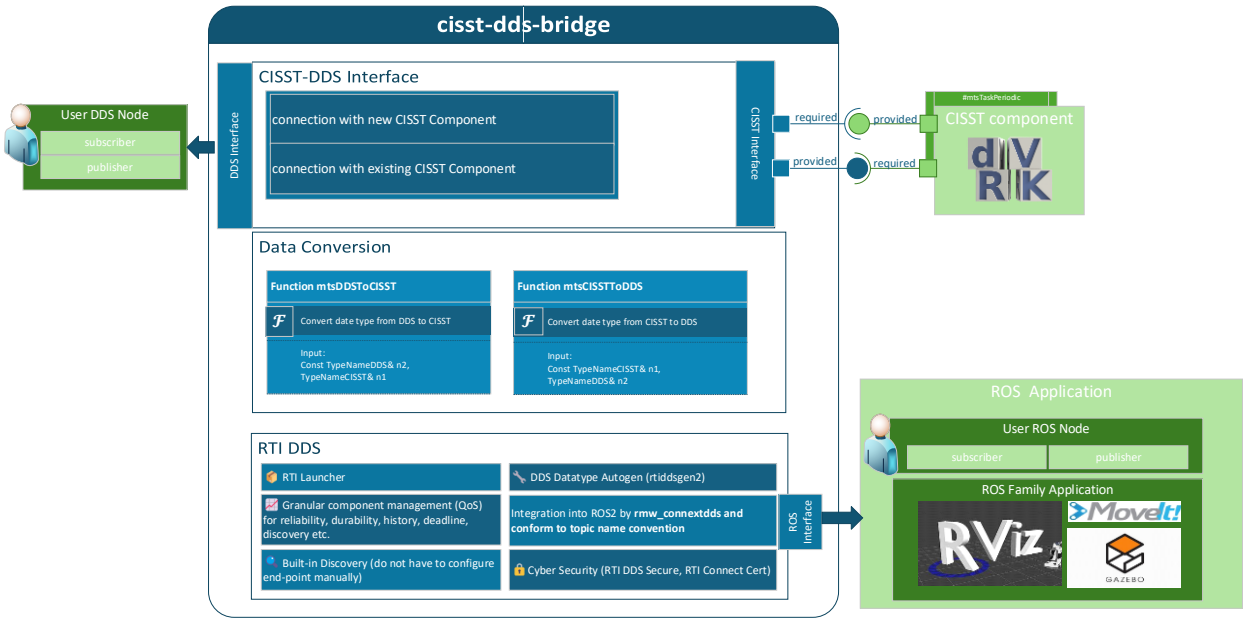

Figure 2. *mtsDDSBridge* middleware architecture. The middleware provides interfaces to the ROS family applications and to the CISST component. The user can also access those interfaces either from a DDS node or ROS node with publisher-subscribe model.

KEY DESIGN PATTERN

In this section, I will further explain the design philosophy of *mtsDDSBridge* with respect to three aspects: the communication model, DDS class instantiation (participant, publisher, etc.), and ROS-compatible topic design. The *mtsDDSBridge* is designed to mirror the modular and interface-based architecture of the cisstMultiTask framework while exposing its communication channels to the DDS network. As shown in *Figure 3*, *mtsDDSBridge* connects to a given CISST component (e.g., dVRK) via its required and provided interfaces, enabling external DDS nodes to interact with internal CISST commands, events, and data streams. In CISST, events and commands use named functions (event handlers and command interfaces) to drive logic, while data transmission uses StateTable integration to handle continuous, timestamped data exchange. You can find a more intuitive example from Chocolate Factory.

These represent different patterns of data flow and control, and users can employ a DDS node using specific function templates that instantiate DDS publishers or subscribers accordingly.

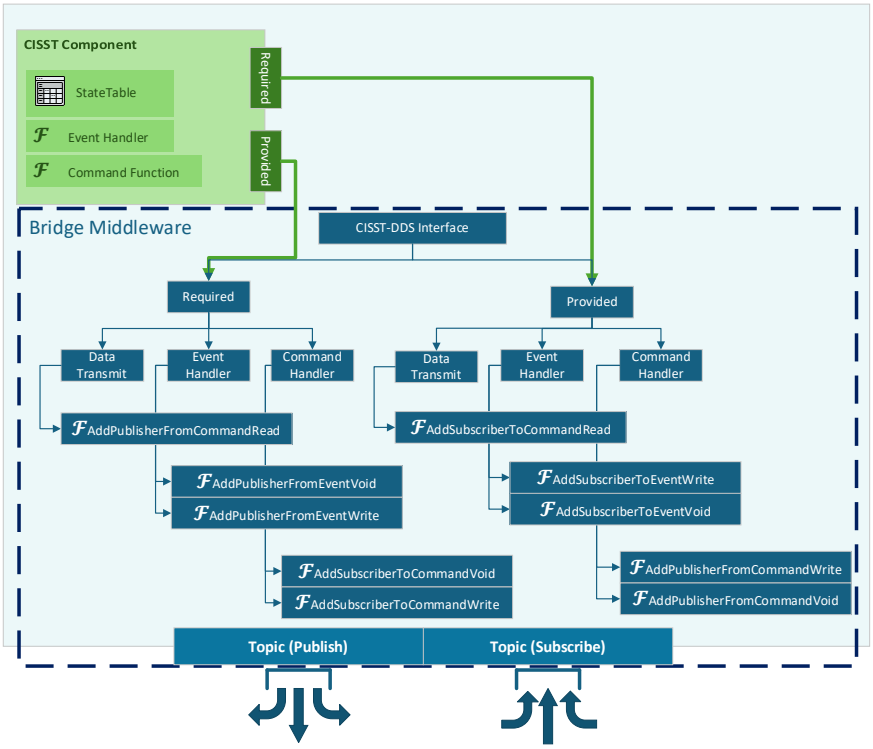

Figure 3. Data/Signal interfaces between CISST component and *mtsDDSBridge*

The *mtsDDSBridge* follows the recommended DDS communication structure outlined by RTI Connext [3], using a hierarchical design pattern to instantiate and manage DDS entities. This

structured approach ensures modularity, scalability, and consistent application of QoS (Quality of Service) policies across components. As shown in *Figure 4*, the foundational building block is the Participant, which represents a single application instance in a DDS domain. All communication entities—Publishers, Subscribers, DataWriters, and DataReaders—are created from this participant. Each level supports its own class of QoS policies.

In *mtsDDSBridge*, this structure is reflected in the add_dds_publisher() and add_dds_subscriber() functions, which instantiate Publisher and Subscriber objects tied to the central participant. These functions first check whether an instance already exists for the given name, and if so, they reuse the existing object rather than creating a new one. Reusing publishers and subscribers helps to conserve system resources and maintain consistent QoS policy application. Writers and readers are then created per topic through helper classes, following a one-to-one mapping with CISST communication endpoints.

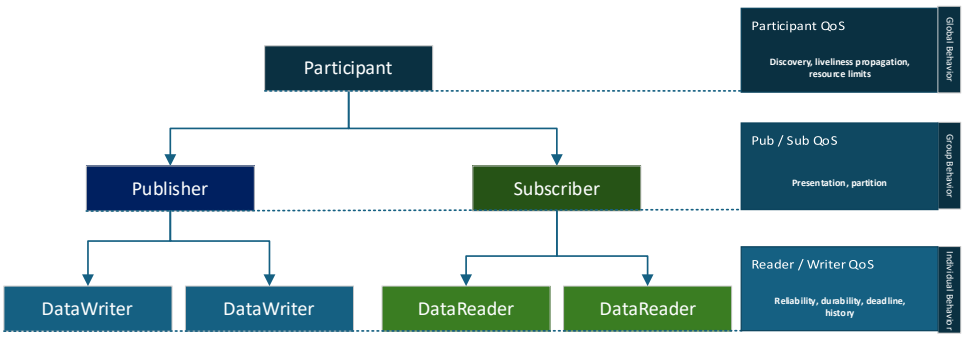

Figure 4. DDS Entity Hierarchy and QoS Configuration Layers

To ensure interoperability between DDS nodes and ROS applications, the *mtsDDSBridge* adheres to a two-step compatibility process that enables seamless data exchange and topic discovery within the ROS ecosystem.

Step 1: ROS Topic Name Convention: ROS 2 relies on a specific naming convention for DDS topics. To make DDS topics discoverable and interpretable by ROS applications, topic names must conform to ROS's namespace and prefixing rules.

- Regular ROS topics are discovered if they are prefixed with "rt/" (i.e., "ROS topic") as shown in *Figure 5a*.
- Topics related to services, actions, and parameters follow the "rq/" (request) and "rr/" (response) prefix patterns.

Step 2: ROS-Compatible Data Types: DDS message types must also match ROS expectations for communication to succeed. ROS message types are translated into DDS IDL-generated types, often nested under vendor-specific namespaces as shown in *figure 5b*. For example:

- A ROS 2 std_msgs/String message is converted to the DDS type: std_msgs::msg::dds_::String_

A ROS-DDS communication test script was developed as shown in *figure 5c* to show the messages can successfully stream on the topic "PSM1/measured_js".

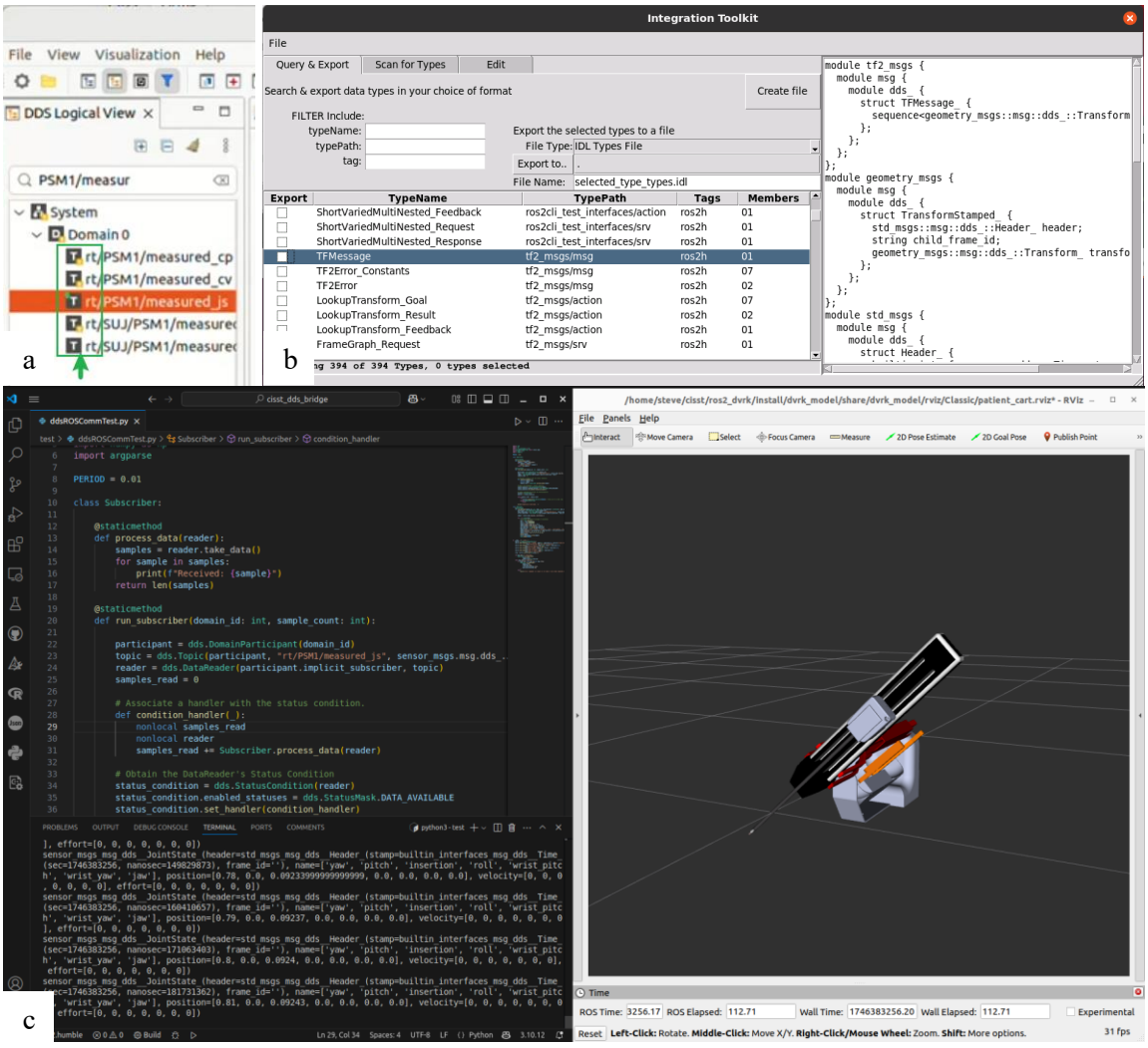

Figure 5 a. ROS 2 Topic Naming Convention with DDS Prefixes (rt/, rq/, rr/) b. ros-integration-tool. This tool can scan the path for ROS data type and automatically convert them with a legal DDS .IDL file. c. Demonstrate a DDS node can communicate with a pure ROS node (RViz). A

ROS-DDS Communication Test script is used to control the trajectory of PSM1 and collect the joint state data. [Here is a complete demo video]

IMPLEMENTATION

The *mtsDDSBridge* middleware is instantiated and equipped to a CISST component in two ways:

Method 1 manual integration within a sample project: In this method, both the target CISST and the *mtsDDSBridge* component are created within the same project. Developers can explicitly register both components using *mtsManagerLocal*, and manually connect the required and provided interfaces through the code. [Method1 example]

Method 2: Library Integration for Existing Executables: When integrating the bridge into an existing CISST-based application—such as dvrk_console_json or sawIntuitiveResearchKitQtJSON—the *mtsDDSBridge* component is compiled into a shared library In this case, interface connections are automatically configured at runtime using a .json configuration file. This file defines both the library reference and the interface connection schema, allowing the executable to dynamically load and link the bridge without modifying its source code. [Method2 example]

EXPERIMENT SETUP

To evaluate the performance of the *mtsDDSBridge*, two experiments were conducted using different integration methods and application contexts. Each experiment is designed to assess specific performance aspects of the bridge, including throughput, latency, and trajectory-following fidelity. All experiments were performed on a single computer (Intel(R) Core(TM) i7-14700KF 3.40 GHz, 32GB RAM), using ROS or DDS to provide inter-process communication.

The first experiment aims to compare the throughput capability of *mtsDDSBridge* versus the existing *mtsROSBridge*. As illustrated in *Figure 6*, both bridges were integrated into the same application: dvrk_console_json. A designed frequency ranging from 100 Hz to 100 kHz was configured in the main application loop. The system's server-side publishing rate (DDS or ROS bridge frequency) and client-side subscription rate (DDS or ROS subscriber frequency) were

measured to determine how efficiently data is transmitted and received across the two middleware systems.

Although such high frequencies exceed those used in practical robot control scenarios, the purpose of this experiment is to stress the system and identify maximum achievable throughput.

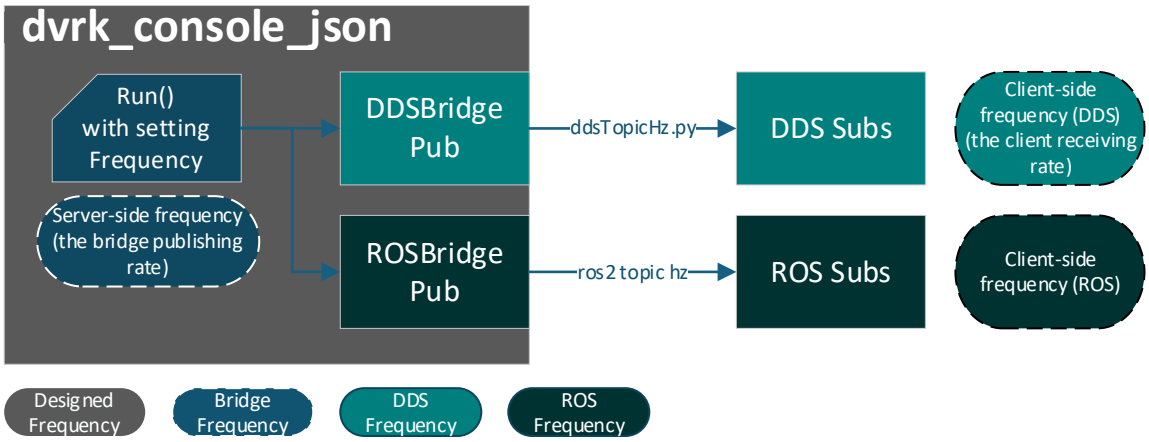

Figure 6. Maximum Throughput Performance Evaluation Experiment Setup

The second experiment evaluates communication latency and trajectory-following performance using *mtsDDSBridge* in a closed-loop robotic control task. As shown in *Figure 7*, the experiment is executed using sawIntuitiveResearchKitQtConsoleJSON with the *mtsDDSBridge* linked as a library. An external Python script, ddsPSMSinMoveTest.py, is used to publish a predefined cosine joint trajectory via the servo_jp topic. During execution, the script also subscribes to setpoint_js and measured_js topics to collect feedback.

Latency measurement: Extract timestamps from setpoint_js messages and compare them to timestamps in servo_jp commands to calculate roundtrip latency.

Trajectory deviation analysis: Analyze joint position values from measured_js and compare them against expected cosine wave values to compute trajectory error under different bridge publishing frequencies.

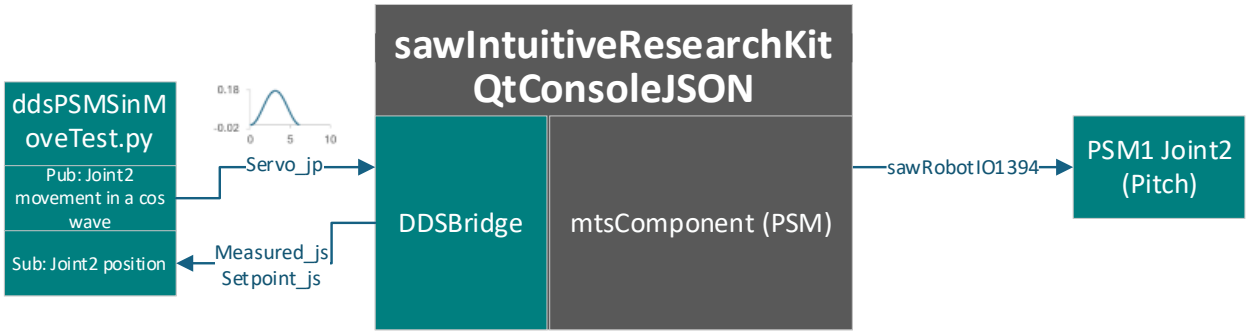

Figure 7. Trajectory Following Performance Evaluation and Latency Measurement Experiment Setup. The sinusoidal trajectory is configured to run by 5 cycles within 5s under different frequencies. The measured_js topic publishes sensor measured joint positions. The setpoint_js topic publishes the command that was received by the arm controller.

RESULT

The throughput experiment revealed key performance thresholds for both DDS and ROS bridges under increasing publishing frequency:

- Below 1.6 kHz: Both the server-side publisher (dvrk_console_json) and the client-side subscriber (DDS or ROS) were able to sustain the designed frequency without degradation as shown in *Figure 8a*.
- At 3.2 kHz and above: The server-side bridge could no longer maintain the target frequency. The limiting factor could be the computation speed of the application, which could not generate and publish messages fast enough—regardless of whether DDS or ROS was used.
- At 50 kHz and above: Packet loss occurred on the server side for both DDS and ROS. ROS exhibited more pronounced loss, indicating that DDS is comparatively more resilient under high-frequency load, though both approaches exceeded the practical throughput capacity of the system as shown in *Figure 8b*.

These findings highlight that while both bridges are suitable for typical control frequencies (<1 kHz), DDS offers greater robustness under computational stress and higher messaging rates.

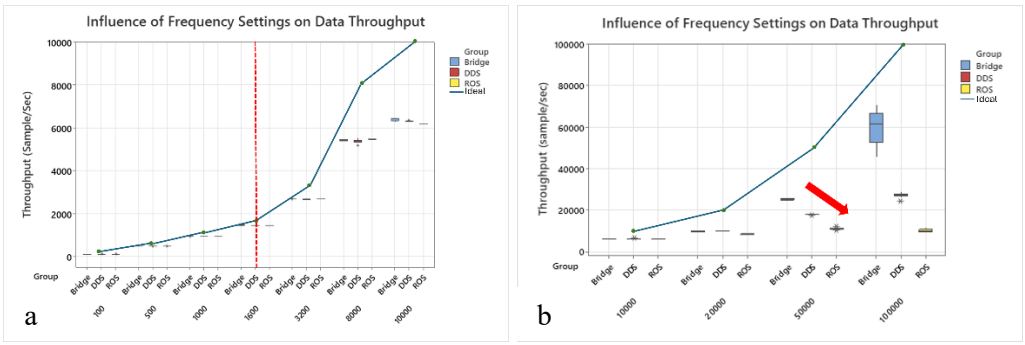

Figure 8 a. Influence of Frequency Settings on Data Throughput (frequency from 100Hz – 10kHz) b. Influence of Frequency Settings on Data Throughput (frequency from 10kHz – 100kHz)

In the second experiment, the *mtsDDSBridge* was evaluated under varying frequencies from 100 Hz to 1.6 kHz to assess its impact on trajectory-following accuracy and communication latency. The best trajectory performance was observed at 100 Hz, with:

- Mean error: 0.00231 rad (0.13°)
- Max error: 0.00648 rad (0.37°)
- Standard deviation: 0.00140 rad (0.08°)

These results demonstrate that higher frequency does not improve trajectory-following performance. In fact, increasing the frequency introduces more system overhead, which in turn leads to greater communication latency and delays in completing the trajectory.

This confirms that an optimal frequency must balance responsiveness with system processing capacity, and that overly high update rates can degrade rather than enhance performance in real-time robotic control.

It can also be observed from *Figure 9c* that the received trajectory is offset to the right of the published trajectory. The Latency was measured by extracting timestamps from setpoint_js messages and comparing them to the corresponding servo_jp command timestamps, representing the one-way delay in command transmission and system response. The measured communication latency was:

DDS	ROS
• Mean: 6.62 ms • Standard deviation: 2.92 ms • Maximum latency: 11.56 ms • Reliability*: 98.4%	• Mean: 5.23 ms • Standard deviation: 3.82 ms • Maximum latency: 12.94 ms • Reliability: 94%

*Reliability: the received data is the same as the published data

Although the latency results show that ROS achieved a slightly lower mean latency (5.23 ms vs. 6.62 ms for DDS), it is important to consider data completeness and reliability. Out of 500 expected samples, ROS received only 492, and just 94% of the data were considered valid for analysis.

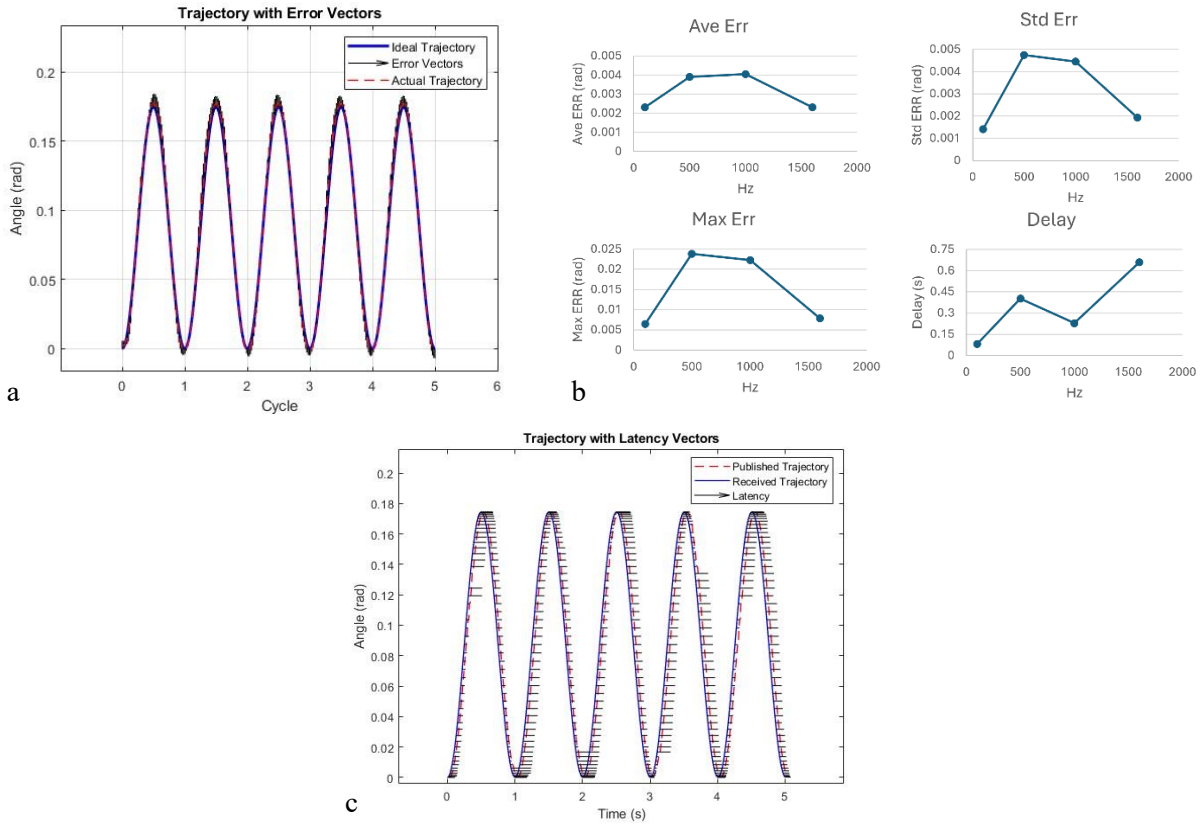

Figure 9 a. Trajectory-following performance evaluation result (100Hz). b. Trajectory-following performance statistical results (100Hz, 500Hz, 1kHz, 1.6kHz). c. Trajectory-following one-way latency evaluation result (100Hz).

CONCLUSION

DISCUSSION AND FUTURE WORK

The results from the two experiments provide valuable insights into how developers and system integrators can select appropriate communication frequencies and anticipate system behavior in terms of latency and trajectory-following accuracy when using the *mtsDDSBridge* in real-time robotic systems.

Overall, the experiments suggest that higher communication frequencies do not necessarily lead to better performance. Instead, they may introduce increased latency, computational load, and trajectory deviation. The trajectory-following experiment showed that optimal tracking was achieved at 100 Hz, with low error and minimal delay. This indicates that for most practical applications, particularly those involving moderate-speed manipulation or surgical assistance, frequencies below 1 kHz offer a good balance between responsiveness and system stability.

In the maximum throughput experiment, the system demonstrated reliable performance up to 1.6 kHz. However, at 3.2 kHz and above, the application could no longer maintain the target publishing rate. This highlights a processing bottleneck on the server side, likely due to limitations in computational throughput, thread scheduling, or memory bandwidth. To push the system's capabilities beyond this point, it would be necessary to optimize the computational efficiency of the application. At frequencies above 50 kHz, the subscriber rate was observed to fall behind the publishing rate, indicating packet drops or delayed delivery. Potential causes include operating system scheduler delays and default DDS QoS settings. To address these issues and improve performance at higher frequencies, two strategies are recommended. Optimize DDS QoS settings using a static payload and adjusting deadline and lifespan parameters to suit real-time needs [4]. Additionally, bind critical processes to isolated CPU cores, reducing OS-induced jitter and contention [5], thereby ensuring more consistent execution timing.

MANAGEMENT SUMMARY

KEY DELIVERABLES

This project successfully met the minimum and most of the expected and maximum deliverables. A runnable application using DDS middleware integrated with the CISST library was achieved, the system was validated on physical hardware, and performance was tested on both the physical dVRK platform and in simulation environments. ROS-compatible DDS publishing/subscribing was also implemented to support integration with ROS ecosystem tools. One expected feature (DDS-CRTK) is still under development.

Minimum	• A runnable application using pure DDS middleware with CISST library.	✅

Expected	• Deploy and validate the application on a physical system.	✅
	• Test DDS based application performance in both remote and local console environments.	✅
	• Topic and type auto population in compliance with CRTK naming convention.	⏳
Maximum	• Enable DDS message publishing/subscribing in a ROS2-compatible format to integrate with ROS ecosystem tools.	✅

DEPENDENCIES

All required dependencies, including software libraries, licenses, and dVRK hardware, were acquired on time.

Dependency	**Need**	**Status**	**DDL**
CISST library	Necessary software library	Acquired	02/28
RTI DDS SDK	Necessary software library	Acquired	02/28
RTI DDS license	Necessary license for SDK	Acquired	02/28
ROS2 RMW library	Necessary software library	Acquired	02/28
dVRK hardware	Necessary for performance verification	Acquired	03/30

REFLECTION

I would like to thank the mentors for their background guidance and technical feedback, and Anton for leading two insightful workshops that supported the DDS integration effort. Through this project, I gained foundational knowledge of real-time robotic architecture via the cisstMultiTask framework, and significantly deepened my understanding of DDS middleware, particularly in the areas of performance tuning and experimental validation. On a personal level, I improved my project leadership, time management, and cross-team collaboration skills—strengthening both technical and professional competencies.

REFERENCES

[1] dVRK. (n.d.). *da Vinci Research Kit Documentation – Development Branch*. https://dvrk.readthedocs.io/devel/

[2] Quigley, M., Gerkey, B., & Smart, W. D. (2021). Micro-ROS. In Robot Operating System (ROS): The Complete Reference (Volume 7) (pp. 45–70). Springer. https://doi.org/10.1007/978-3-031-09062-2

[3] RTI Connext DDS. (n.d.). *RTI Connext DDS Core Libraries Getting Started*. Retrieved May 1, 2025, from https://community.rti.com/static/documentation/connext-dds/6.0.1/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_GettingStarted/index.htm

[4] Liang, W. Y., Yuan, Y., & Lin, H. J. (2023). A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds. arXiv preprint arXiv:2303.09419.

[5] Bode, V., Trinitis, C., Schulz, M., Buettner, D., & Preclik, T. (2023, August). DDS Implementations as Real-Time Middleware–A Systematic Evaluation. In *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (pp. 186-195). IEEE.

APPENDIX

Project Management	https://github.com/users/stevenleon99/projects/4
Project Code Repository	https://github.com/stevenleon99/cisst_dds_bridge