2019-09

PERFORMANCE ANALYSIS OF ROS 2 NETWORKS USING VARIABLE QUALITY OF SERVICE AND SECURITY CONSTRAINTS FOR AUTONOMOUS SYSTEMS

Chen, Zhaolin

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PERFORMANCE ANALYSIS OF ROS 2 NETWORKS USING VARIABLE QUALITY OF SERVICE AND SECURITY CONSTRAINTS FOR AUTONOMOUS SYSTEMS

by

Zhaolin Chen

September 2019

Thesis Advisor:	Preetha Thulasiraman
Second Reader:	Brian S. Bingham

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2019	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE PERFORMANCE ANALYSIS OF ROS 2 NETWORKS USING VARIABLE QUALITY OF SERVICE AND SECURITY CONSTRAINTS FOR AUTONOMOUS SYSTEMS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Zhaolin Chen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This thesis studies the network performance of the Robot Operating System (ROS) 2 when used in a network of nodes similar to how a group of unmanned assets would operate. Specifically, this thesis evaluates the impact of combining varying Quality of Service (QoS) and security settings in the ROS 2. It also explores the effect that scaling to multiple nodes has on network performance. This is the first work to comprehensively study ROS 2 network performance using QoS and security classification as a function of scale and message size. Network performance metrics include latency and message drop rate between nodes. Our research uniquely integrates ROS 2 with NS-3, developing a simulation architecture that is effective for rapidly studying ROS 2 network performance. Our simulation results demonstrated the trade-offs in choosing different QoS policies as well as the trade-offs in performance when security settings were enabled. We found that enabling security resulted in a higher message drop rate across all QoS profiles. We also found that scaling the network to more nodes resulted in various consequences with the use of different QoS settings. Scaling up to more nodes in a network also resulted in an equivalent increase in the average latency of messages. This work contributes to evaluating and configuring ROS 2 parameters for different unmanned system use cases while providing a simulation framework on which tests can be run.

14. SUBJECT TERMS ROS2, Robot Operating System 2, NS3 network simulator, QOS, quality of service, network performance	**15. NUMBER OF PAGES** 87
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

PERFORMANCE ANALYSIS OF ROS 2 NETWORKS USING VARIABLE QUALITY OF SERVICE AND SECURITY CONSTRAINTS FOR AUTONOMOUS SYSTEMS

Zhaolin Chen
Major, Republic of Singapore Air Force
BEE, National University of Singapore, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2019

Approved by: Preetha Thulasiraman
Advisor

Brian S. Bingham
Second Reader

Douglas J. Fouts
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis studies the network performance of the Robot Operating System (ROS) 2 when used in a network of nodes similar to how a group of unmanned assets would operate. Specifically, this thesis evaluates the impact of combining varying Quality of Service (QoS) and security settings in the ROS 2. It also explores the effect that scaling to multiple nodes has on network performance. This is the first work to comprehensively study ROS 2 network performance using QoS and security classification as a function of scale and message size. Network performance metrics include latency and message drop rate between nodes. Our research uniquely integrates ROS 2 with NS-3, developing a simulation architecture that is effective for rapidly studying ROS 2 network performance. Our simulation results demonstrated the trade-offs in choosing different QoS policies as well as the trade-offs in performance when security settings were enabled. We found that enabling security resulted in a higher message drop rate across all QoS profiles. We also found that scaling the network to more nodes resulted in various consequences with the use of different QoS settings. Scaling up to more nodes in a network also resulted in an equivalent increase in the average latency of messages. This work contributes to evaluating and configuring ROS 2 parameters for different unmanned system use cases while providing a simulation framework on which tests can be run.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
CA	Certificate Authority
DDS	Data Distribution Service
DoD	Department of Defense
EDP	Endpoint Discovery Protocol
IDL	Interface Definition Language
IPV4	Internet Protocol version 4
LTS	Long Term Support
MAC	Message Authentication
PDP	Participant Discovery Protocol
QoS	Quality of Service
RMW	ROS Middleware
ROS	Robot Operating System
RTI	Real Time Innovations
RTPS	Real-Time Publish-Subscribe
SPI	Service Plugin Interfaces
SROS	Secure Robot Operating System
TCP	Transmission Control Protocol
UAV	unmanned aerial vehicles
UDP	User Datagram Protocol
UxS	unmanned systems
VPN	Virtual Private Network
XML	Xtensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

In the process of writing of this thesis, I have received much guidance and support from many people. I would first like to thank my thesis advisor, Professor Preetha Thulasiraman, whose expertise and guidance were invaluable in enabling me to successfully complete this thesis.

To Professor Brian Bingham and Bruce Allen: the weekly meetings were especially helpful in ensuring progress in this thesis. I started off not knowing what the Robotic Operating System and NS-3 were, and to have conquered the learning curve necessary for this thesis would not have been possible without the both of you.

Finally, I must express my profound gratitude to my wife, Kai Ling. Her support and encouragement, especially in raising our children, Matthias and Ruth, were invaluable throughout my year of study. This thesis would not have been possible without her.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. UNMANNED SYSTEMS

Unmanned aerial vehicles (UAV) were first used in a military application in 1849, where unmanned balloons were used to attack the city of Venice, Italy. Modern-day unmanned systems (UxS) gained prominence with the use of UAVs by the Israeli Air Force in their victory over the Syrian Air Force [1]. Since then, the military application for UxS has continued to grow in popularity, with UxS used in the air, surface, and subsurface domains.

Advances in the technology of UxS have enabled their use in a growing variety of military applications. One such advance is the development of mini and micro UxS. Their small size makes them cheaper, easier to deploy, and able to access physical areas which larger UxS cannot. The small size of these UxS also brings with them their own limitations. These include a small payload size and a small communication range. These limitations can be overcome through the use of swarming tactics, which allows the interaction between all UxS entities to achieve a common goal. In [2], the author demonstrates multiple situations where the limitations of UxS could be overcome through the use of swarm tactics. An example of swarming tactics is the use of multiple micro UxS to conduct search or reconnaissance of a single large area.

In [3], the authors explain how UxS are seen as game changers for the military in the additional capabilities that this technology brings. Yet, the introduction of UxS technology for both military and civilian applications has brought with it its own set of challenges. The Unmanned System Integrated Roadmap (2017–2042) released by the U.S. Department of Defense (DoD), listed interoperability and network security as critical UxS needs [4]. Interoperability is described as allowing for interactions between systems and allowing for information to be transmitted in a timely fashion between different users. A common or open architecture is seen as a key enabler for interoperability. Network security is described as being vital to protect the integrity, availability, and confidentiality of information flow between UxS assets.

B. ROBOT OPERATING SYSTEM AS A COMMON FRAMEWORK

One of the difficulties in the development of any new robotics or UxS program lies in the amount of resources required to establish the software infrastructure. Code has to be written to interface and drive the hardware within the system being developed. This means that across multiple programs, the software infrastructure has to be re-developed instead of being reused. The use of a common software infrastructure would mitigate this wastage of resources, as well as any interoperability issues.

The Robot Operating System (ROS) was developed as a framework that provided the software infrastructure on which others could build their UxS. As a framework, ROS provides a set of tools and libraries that simplify the task of creating a new robot. ROS is now managed by Open Robotics as a free and open-source software. It has built up a large set of tools supported by an equally large ROS community, which any developer can utilize.

The second version of ROS, ROS 2, was initiated by the Open Source Robotics Foundation, the predecessor of Open Robotics, to address the shortcomings of ROS 1 that were identified by the industry. Deficiencies of ROS 1 include the fact that it is dependent on a central node (roscore), which is seen as single point of failure for any UxS. ROS 1 was also built without cyber security in mind, and while there were attempts to secure the software, it was concluded that a redevelopment of the framework from the ground-up would be required to address cyber security concerns. ROS 2 was first released in 2015, while the first version with long term support (LTS) was released in June 2019.

ROS-M is the militarized version of ROS and is built upon the ROS 2 framework. ROS-M is meant to address the specific needs of military UxS. Similar to the purpose of ROS, ROS-M seeks to reduce development cost by promoting code sharing and reuse. It also seeks to meet security requirements of the military. In addition to the lower cost associated with the use of a common platform, the use of ROS 2 would enable ROS-M to tap into the existing work executed by many researchers and industry participants.

C. THESIS CONTRIBUTION

In this thesis, we assess the network performance of ROS 2 when used in a network of nodes similar to how a group of UxS would operate. Given the recent introduction of ROS 2, there have not been many studies of the network performance of ROS 2. Information regarding network performance when different Quality of Service (QoS) and security settings are enabled contributes to evaluating and configuring ROS 2 for different UxS use cases.

For this thesis, simulations are performed using NS-3, which is a discrete-event network simulator. Our tests focus on the effect of QoS network settings, security settings and scalability, on network performance. Network performance is evaluated based on message transmission latency and message loss rate.

The contributions of this thesis are:

- Development of a simulation framework that integrates NS-3 and ROS 2, allowing for the network performance of multiple ROS 2 nodes to be evaluated without the need for multiple hardware to host the ROS 2 nodes.
- Simulation and evaluation of the network performance with ROS 2 nodes under varying QoS profiles.
- Simulation and evaluation of the network performance with ROS 2 nodes with security settings turned on and off.
- Simulation and evaluation of the network performance of two ROS 2 nodes versus five ROS 2 nodes.

To the best of our knowledge, this is the first work that comprehensively studies ROS 2 network performance using QoS and security classifications as a function of scale and message size.

D. THESIS ORGANIZATION

The remainder of this thesis is organized as follows: In Chapter II, we provide an overview of the ROS 2 architecture, as well as the available QoS and security settings. We also study the related work on the network performance of ROS 2. In Chapter III, we describe the simulation setup of how NS-3 is used to simulate the network between multiple ROS 2 nodes. In addition, the chapter details the expected effect of each QoS and security setting. In Chapter IV, we present the simulations results and discuss the significant implications for each of the performance metrics studied. Finally, we draw conclusions and propose recommendations for future work in Chapter V.

II. BACKGROUND AND RESEARCH WORK

A. ROS 2 ARCHITECTURE

ROS 1 was first developed to be used with the Willow Garage PR2 robot. The PR2 robot enabled the development team to showcase the capabilities of what ROS 1 could do. This meant that development choices of ROS 1 were partly guided by the use case of the PR2 robot as explained in [5]. This resulted in development decisions such as the use of a central node as well as assuming that optimum network connectivity existed, given that all nodes were physically on a single robot. Also, communication between nodes was not secured from a cyber security perspective. The vulnerability of ROS 1 was shown in [6], when the authors searched the entire Internet Protocol version 4 (IPV4) space of the internet for ROS 1 instances that were exposed to the public. The authors were then able to read data and control multiple ROS 1 nodes given the lack of security of the communications.

ROS 2 addresses many of the shortcomings of ROS 1. One significant change is the use of the Data Distribution Service (DDS) for communication between nodes in ROS 2. DDS is a middleware framework to address the need for real-time data exchange by various applications. As part of the framework, messages are exchanged between nodes using the Real-Time Publish-Subscribe (RTPS) protocol. The use of DDS means that ROS 2 is able to make use of features implemented by supported DDS vendors. The implementation of QoS and security settings is handled within the DDS application. Users do not have to change their code should there be any programming change in the DDS layer as it would be taken care of by the ROS interface between the application and the DDS layer. All ROS code would be agnostic to the DDS implementation, while all DDS code would be agnostic to the ROS code, with the intra-process application programming interface (API) handling the interface between the two. Figure 1 illustrates how the ROS application layer works with the DDS middleware.

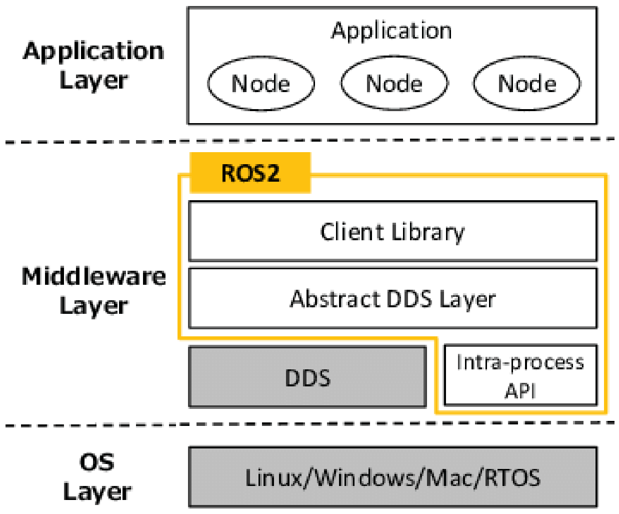


Figure 1. ROS 2 architecture and DDS. Adapted from [7].

ROS 2 currently supports three different DDS vendors as shown in Table 1. The different DDS vendors are expected to be compatible, as they are implementations of the same DDS framework. As such, each node in a network could be using a different DDS vendor and still be able to communicate with the others.

Table 1. List of DDS vendors supported by ROS. Adapted from [8].

Product Name	**License**	**Status**
eProsima Fast RTPS	Commercial, Research	Full support available.
RTI Connext	Commercial, Research	Full support available. Needs to be installed separately
ADLINK Opensplice	Apache 2, Commercial	Only partial support available. Needs to be installed separately
OSRF FreeRTPS	Apache 2	Only partial support available. Development paused.

B. ROS 2 QUALITY OF SERVICE SETTINGS

ROS 1 originally made use of the Transmission Control Protocol (TCP) as its transport protocol. TCP, however, is unsuitable for use in a lossy wireless network. ROS 2's use of DDS makes it more suitable for use in a lossy network. DDS uses the RTPS protocol, which was designed to run over an unreliable network, using the User Datagram Protocol (UDP).

On top of the UDP layer, DDS allows for different QoS policies, providing the user with control over the behavior of the network. These policies address four main aspects of network performance: Real-time Delivery, Bandwidth, Redundancy, and Persistence. Although DDS supports a multitude of QoS policies, as of the ROS 2 Dashing Diademata release, ROS 2 only supports seven different policies. The policies are: History, Depth, Reliability, Durability, Deadline, Lifespan, and Liveliness. The last three policies are newly supported in the Dashing Diademata release, with the first three releases only supporting the first four policies. Table 2 provides a description of each QoS policy supported by ROS 2. This thesis focuses on the first four policies: History, Depth, Reliability, and Durability.

Table 2. Description of ROS 2 QoS policies

QoS Policy	**Description**
History	There are two settings, “Keep last” and “Keep all.” History serves to configure the number of messages that the Publisher or Subscriber will keep in its cache.
Depth	Size of the queue if “Keep last” is configured as the history setting.
Reliability	There are two settings, “Reliable” and “Best effort.” The Reliable setting helps ensure that all messages are delivered. Best effort attempts to send each message only once.
Durability	This policy determines whether the Publisher sends past messages to a newly joined Subscriber.
Deadline	Ensures that messages are sent or received within a specified duration.
Lifespan	Determines the duration for which a message is valid.
Liveliness	Configures the Publisher and the Subscriber to check that the connection is still valid.

In [7], the authors compared the network performance of ROS 1 and ROS 2. The authors showed that the use of DDS middleware did indeed help address the short-falls in ROS 1. This included the fact that a user no longer had to launch a Subscriber node before the Publisher node, as the Durability QoS policy in DDS ensured that the Publisher still receives messages published before it was launched. ROS 2, however, showed large latency increases when the message size was increased. This was attributed to the fact that RTPS was designed for lightweight communication. As such, messages were divided into small packet sizes, with a corresponding increase in overhead when the message size was large.

In [9], the authors measured the latency for messages using three different DDS vendor implementations. There were minimal differences among each implementation, with all three implementations demonstrating similar impact on network performance depending on the QoS policies used.

The work in this thesis extends the existing research by exploring the impact of varying QoS profiles within a lossy network. We also explore the specific impact that different QoS policies have on network performance. Given the similarities in DDS implementation, eProsima Fast RTPS was used for our simulations, as it’s the default middleware chosen by ROS2.

C. SECURE ROS 2

DDS security features are made available for use with ROS 2 through a set of tools named Secure ROS 2 (SROS 2). Through SROS 2, ROS 2 as the application layer checks for security settings in the application layer and executes the appropriate security plugins in the middleware layer.

1. DDS-Security

Figure 2 depicts how the ROS 2 application, DDS middleware and security plugins interact with one another.

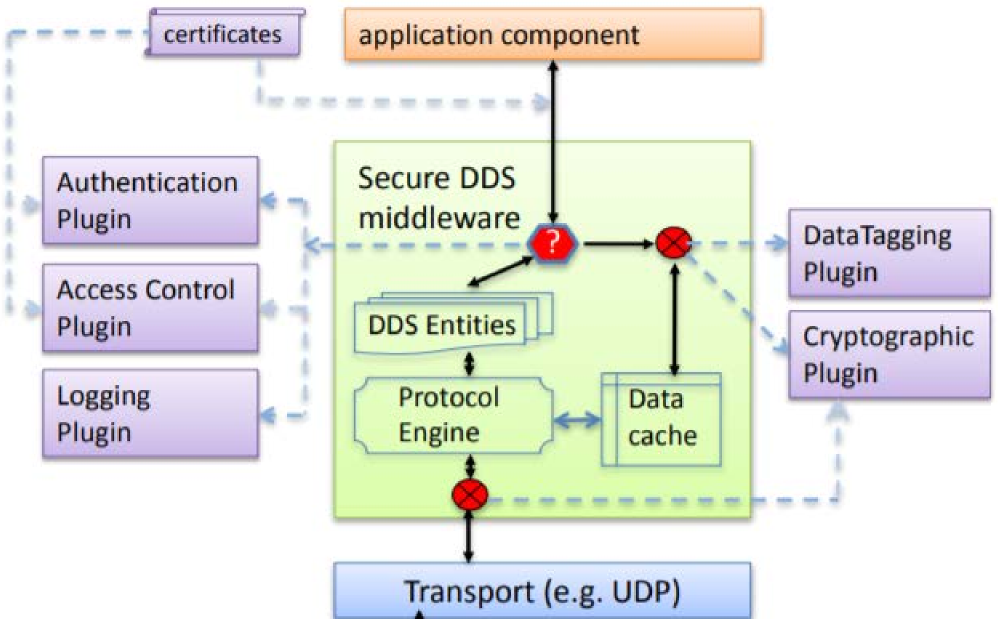


Figure 2. Interaction between application component, DDS, and security plugins. Source: [10].

DDS-Security is a set of specifications that expands on the original DDS and includes a set of Service Plugin Interfaces (SPI). SPIs implement the security model as defined or required by the user [10]. As of the Dashing Diademata version, ROS 2 makes use of only three SPIs. The three SPIs are:

- Authentication: Verification of the identity of the Publisher/Subscriber nodes.
- Access Control: Enforces to which topics the authenticated nodes can publish or subscribe.
- Cryptography: Implementation of cryptographic operations. DDS has separate SPIs that perform encryption, signing as well as hashing.

A user can choose to use one of the SPIs or all of them as part of the security model desired.

2. ROS 2 Security Settings and Roadmap

The security features in DDS-Security are only accessible to the user in ROS 2 via SROS 2. SROS 2 currently does not allow the user to define the SPIs used by the DDS. The settings available to the user are limited to the following:

- A global setting to apply security to all nodes in the network. The setting would apply authentication, access control, and encryption to communications between all nodes.
- Controlling access of each specific node through authentication and access control.
- Allowing the user to control whether the security settings are to be enforced. If the settings are not to be enforced, security would only be applied when the related security files are found. If security needs to be enforced, the node would not be initiated when the security files cannot be found.

The roadmap for ROS 2 development includes plans to allow the user to have finer control over the security settings in the future [11]. This includes the ability to define the SPIs that are used for cryptography (allowing the user to perform only signing, for example). Control over the type of SPIs used will allow a user to understand the appropriate trade-offs between performance and security requirements. ROS 2 also currently stores the security keys in file storage, with the roadmap planning to improve the means of key generation and storage.

3. Performance

At the ROSCON 2018 conference, authors from the DDS vendor Real Time Innovations (RTI) presented the results from the study that they did on the impact of network performance at the DDS layer itself [12]. The study made use of RTI Connext DDS and explored the impact that turning on security settings had on latency and throughput. Simulations were performed using eight different message sizes (32 Bytes, 256 Bytes, 2 Kbytes, 16 Kbytes, 128 Kbytes, and 1 Mbytes) and four security settings (No security, Sign message, Sign message + Encryption, Sign message + Encryption + Origin Authentication). Results showed that using sign message, encryption, and origin authentication, similar to what is used for ROS 2, caused a 25%–41% overhead in terms of latency. It also resulted in an overhead of 1%–32% in terms of throughput. There was also no noticeable difference in impact when scalability was taken into consideration, with simulations run on one, two, and four Subscribers.

In [13], the authors conducted experiments with ROS 2 to examine the overhead incurred when security was enabled. When performed using two nodes in a single computer with a lossless network, having security enabled incurred an overhead of 32% in throughput and 37% in latency. The authors also presented the results from running the experiments performed using two computers with a lossy wireless network and QoS profile set as Reliable. The authors found that the overhead increased linearly as a function of message size.

In [14], the authors compared the impact on performance of using SROS 2 versus using a Virtual Private Network (VPN) to secure communications between two nodes. The authors noted that the overhead for SROS 2 was significantly higher than that of a VPN in terms of latency. A VPN also had a minimal impact on throughput, while SROS 2 caused a significant decrease in throughput. Hardware constraints, however, could limit the use of a VPN. Most VPN protocols also do not inherently support multicast features, which makes it difficult to implement in a swarm network if each node needed to communicate with the others.

Existing work focuses on the impact of SROS 2 on network performance between two nodes. The work in this thesis extends this to exploring the network performance when scaled up to more nodes, with varying QoS profiles and with SROS 2.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTAL DESIGN AND SETUP

In this chapter, we describe the simulation architecture used to evaluate the network performance of ROS 2. The architecture allows for the simulations to be carried out with different network settings like antenna and transmit power, as well as QoS and security settings, as required.

A. ROS 2 COMMUNICATIONS

ROS 1 utilizes a Publisher/Subscriber architecture to pass messages between nodes. Nodes register with a ROS master through which the nodes can discover other nodes. Depending on the topic that the nodes are subscribed to, they then establish communications with the relevant nodes that are subscribed to the same topics.

ROS 2 utilizes a similar Publisher/Subscriber architecture, but without a ROS master. Instead ROS 2 makes use of DDS as its messaging layer. To do so, the ROS Middleware (RMW) in ROS 2 translates ROS messages into Interface Definition Language (IDL) messages that are supported by all DDS vendor implementations. To establish communications between nodes, the DDS framework provides for a set of discovery services, which enable nodes that subscribe to the same topic to dynamically discover one another. When a ROS node is initialized, it broadcasts its presence to all nodes on the same domain by using the Participant Discovery Protocol (PDP). Nodes then respond to this broadcast with their own data, including the type of QoS settings that they are using, via the Endpoint Discovery Protocol (EDP). Connections are then made between nodes if they are compatible in settings. Nodes also continue to periodically send out messages about their own data in order to allow for discovery by nodes that are newly created.

Each DDS participant consists of a data reader and a data writer. The data reader and data writer are responsible for subscribing to the relevant topics and to read and write messages to other participants as necessary. DDS then makes use of the RTPS protocol to send messages on multicast and best-effort transports such as UDP [15]. QoS profiles are used to configure how the messages are sent and ensure that messages can still be sent in a reliable manner over the RTPS protocol if needed.

B. SIMULATION ARCHITECTURE

The intent of the simulations is to evaluate the network performance of ROS 2 with different QoS and security settings. Different simulations are performed with a varying number of ROS 2 nodes in order to evaluate the impact of network performance when the number of ROS 2 nodes is scaled up. Studies involving ROS 2 network performance have previously involved either communications through a network loopback, or two separate computers connected by a wired or wireless network, as described in Chapter II.

The use of a simulator allows the performance of ROS 2 on a wireless network to be studied without requiring a corresponding hardware. This allows us to study the impact of multiple ROS 2 nodes running in the same network. Our simulations utilize NS-3 to simulate a lossy wireless network between the ROS 2 nodes. NS-3 is a discrete-event network simulator that supports the use of different simulation models, allowing it to be used as a real-time network emulator [16]. An example of how virtual hosts can be used to simulate a virtual Wi-Fi network is shown in Figure 3.

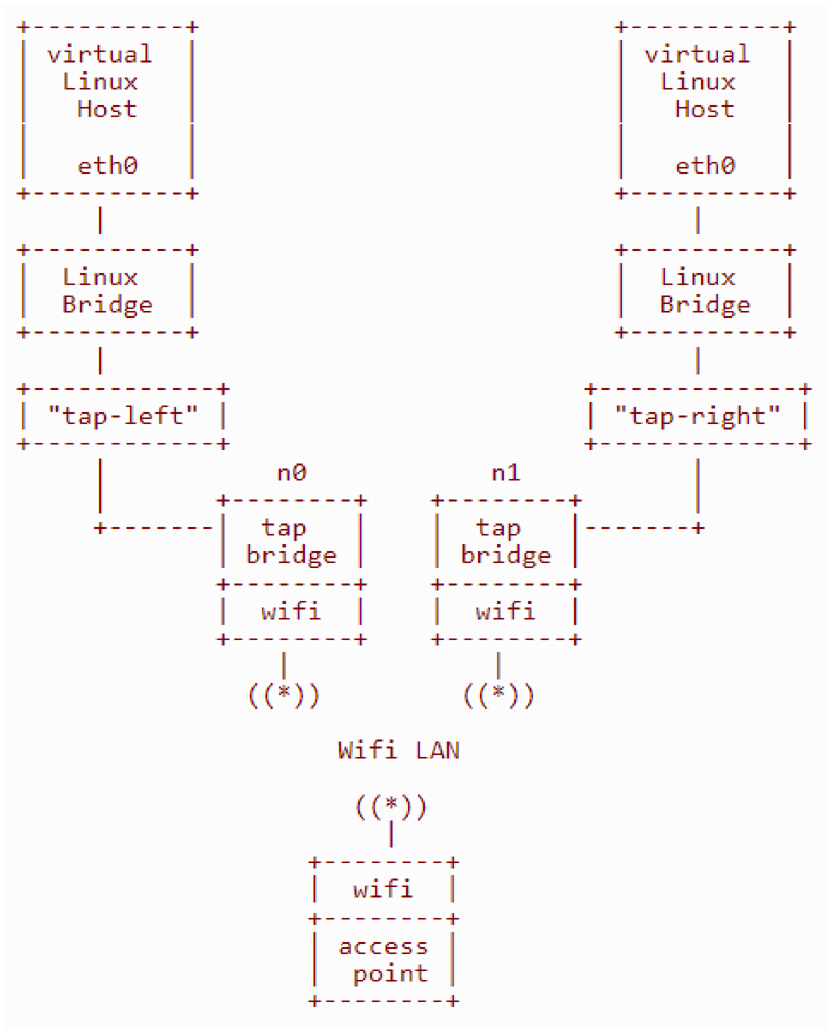


Figure 3. NS-3 network configuration for two nodes. Source: [17].

Our simulation architecture makes use of network namespaces to virtualize the network stack. Each network namespace creates its own network stack for processes within each unique network namespace, including its own network interfaces. For our simulations, a Wi-Fi network interface is created for each individual network namespace. Each ROS 2 node is then executed within its own network namespace, with NS-3 simulating a Wi-Fi network connecting each ROS 2 node. Figure 4 depicts how five ROS 2 nodes within their own network namespace communicate with each other via the simulated NS-3 Wi-Fi. The instructions and script to initialize each network namespace is provided in the appendices.

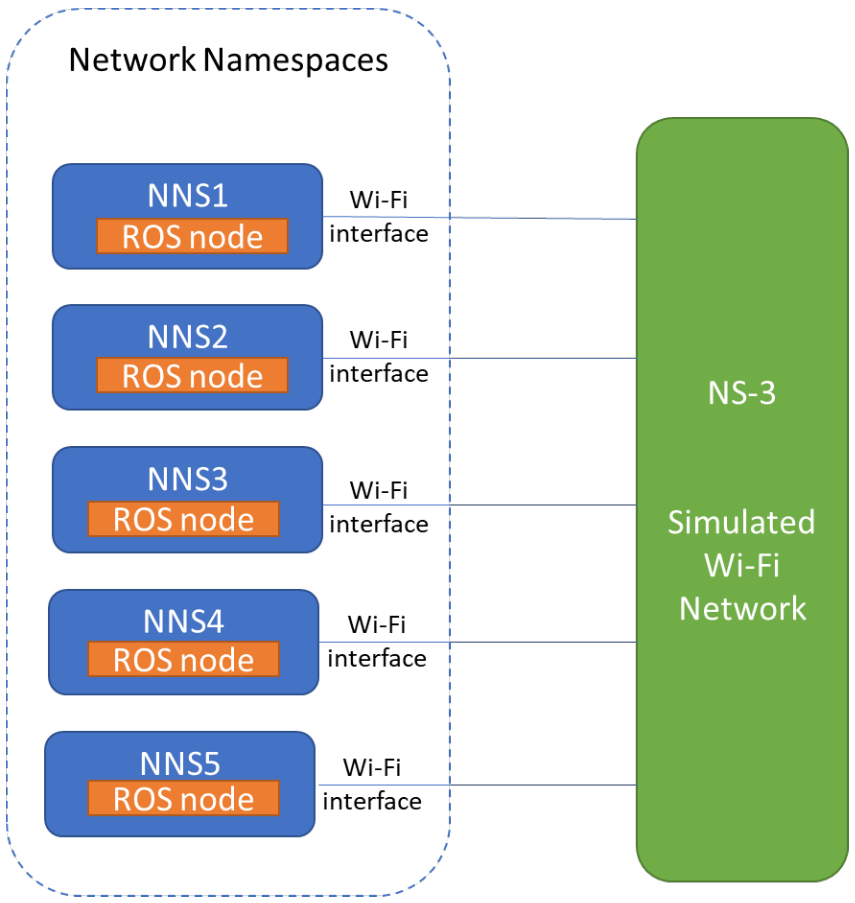


Figure 4. Simulation architecture showing simulation of five ROS 2 nodes

C. NS-3 SETTINGS

NS-3 allows for simulations of different Wi-Fi models. The models used as well as the settings such as antenna strength and throughput can be changed as required. For our simulations, we used the following settings:

- Wi-Fi standard: 802.11a
- Type of network: Ad-hoc
- Data mode: Constant rate OFDM 54 Mbps
- Mobility Model: Constant Position

The remaining settings were the default values provided by NS-3.

In NS-3, nodes are positioned using a 2D Cartesian coordinate system. Using the constant position model, the NS-3 simulator starts with nodes at a distance x from a central node. Figure 5 depicts how four ROS 2 Subscriber nodes are positioned around the central node, which contains the ROS 2 Publisher node. The simulation is run for two minutes, during which the data for the network performance is collected. After collection of the required data, the simulation is re-started with a new distance. Through multiple iterations of this process, network performance at different distances is measured. This provides data regarding network performance of ROS 2 in a lossy network for analysis. Although the position of the four Subscriber nodes does not affect network performance for the simulations in this thesis, the 2D Cartesian coordinate system allows for nodes to be appropriately positioned as required in future simulations.

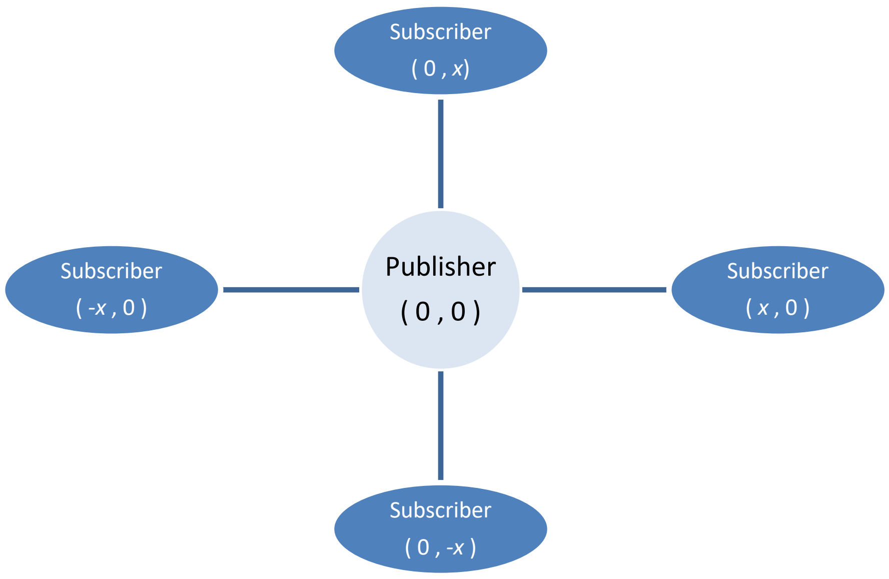


Figure 5. Top down view of the position of Subscriber nodes relative to Publisher node

D. DDS VENDOR

As described in Chapter II, DDS is a middleware framework that has been implemented by multiple vendors. Table 1 lists the different vendor implementations currently supported by ROS. The choice of RMW implementation is left to the user and can be made based on considerations such as license, platform availability, or an implementation that was designed to be targeted at a specific platform. Otherwise, the choice of DDS is considered agnostic to the running of ROS 2. The RMW used by ROS can be switched by changing the RMW_IMPLEMENTATION environment variable. For our experiments, we made use of eProsima Fast RTPS, an open-source DDS implementation. For the purpose of the simulations, only DDS settings, which can be accessed via the ROS 2 application layer, are used as variables.

E. QOS POLICIES IN ROS 2

QoS policies allow communication to occur in a reliable manner over a lossy wireless network even though messages are transmitted over the UDP transport layer. The policies can be applied specifically to each ROS node. This allows each node to be configured flexibly to meet requirements, depending on the type of network that is being used. A QoS profile is made up of different QoS policies. ROS 2 defines four default profiles: Default, Sensors, Parameters, and Services, defined with four policies each. If required, however, ROS 2 also allows specific policies in each QoS profile to be amended as necessary. Table 3 lists the details of the QoS policies used in the QoS profiles defined within ROS 2. The Services profile has the same policies as the Default profile, and thus is not specifically tested for in this thesis.

Table 3. QoS policies for specific QoS profiles in ROS 2

QoS profile	**History**	**Depth**	**Reliability**	**Durability**
Default	KEEP_LAST	10	RELIABLE	VOLATILE
Sensors	KEEP_LAST	5	BEST_EFFORT	VOLATILE
Parameters	KEEP_ALL	1000	RELIABLE	VOLATILE
Services	KEEP_LAST	10	RELIABLE	VOLATILE

The QoS policies of Reliability, History, and Depth are used together to determine the overall reliability with which messages are sent between nodes. These policies affect the reliability of delivery of messages sent from Publisher to Subscriber nodes, especially in a lossy network.

The Reliability policy affects the level of reliability enforced by DDS in delivering messages to Subscribers. Within the Reliability policy, there are two sub-policies: RELIABLE and BEST_EFFORT. If the RELIABLE policy is used, the Publisher waits for an acknowledgment from the Subscriber after each message. If an acknowledgment is not received, the original message is re-transmitted by the Publisher until the Subscriber receives the message. If a BEST_EFFORT policy is used, the Publisher does not listen for any acknowledgment message, and transmits new messages as required.

History controls whether messages are stored in the cache of the data writer or the data reader of a node. Within the History policy, there are two sub-policies: KEEP_LAST and KEEP_ALL. If a KEEP_ALL policy is used, all messages transmitted by a node are stored in the cache of the data writer, up to the system resource limit. If a KEEP_LAST policy is used, then the DEPTH parameter is used to determine the number of messages that are kept in the cache

After nodes discover each other as part of the PDP, they exchange data about each other as part of the EDP. This data includes the index of the messages stored in the cache of the data writer and reader. Publisher nodes and Subscriber nodes compare the messages stored in their data writer and reader respectively, and the messages that are in the writer cache but not the reader cache are transmitted over the network. If a RELIABLE policy is used, this comparison of the writer and reader caches is performed as part of each message acknowledgment. If a BEST_EFFORT policy is used, this comparison is only done during EDP as part of discovery. Subsequently, only the latest message will be sent.

Durability policies control what to do with nodes that join the network late. Durability has two sub-policies: TRANSIENT_LOCAL and VOLATILE. If the TRANSIENT_LOCAL policy is used, all messages stored in the cache of the data writer are sent over to the Subscriber. If the VOLATILE policy is used, data is not stored in the cache, and is not sent to any nodes that join the network later. In the case of our simulations, the Durability policies are not changed between simulations as all nodes are initialized and join the network together. Nevertheless, the policies are included as part of the QoS profiles that are shipped with ROS 2.

Before ROS 2 nodes are able to communicate, their QoS policies need to be compatible. With regards to the Durability policy, both Publisher and Subscriber need to be TRANSIENT_LOCAL before messages can be sent using the TRANSIENT_LOCAL policy. If a Subscriber is using a VOLATILE policy, both Subscriber and Publisher need to communicate using a VOLATILE policy, regardless of the policy used by the Subscriber. Similarly, for the Reliability policy, both Publisher and Subscriber need to be RELIABLE before messages can be sent using the RELIABLE policy. If a Subscriber is using a BEST_EFFORT policy, both Subscriber and Publisher communicate using a BEST_EFFORT policy, regardless of the policy used by the Subscriber. Table 4 summarizes the compatibility of the QoS policies.

Table 4. Summary of compatibility of QoS policies

Policy	**Publisher**	**Subscriber**	**Result**
Durability	VOLATILE	VOLATILE	VOLATILE
	VOLATILE	TRANSIENT_LOCAL	Not compatible
	TRANSIENT_LOCAL	VOLATILE	VOLATILE
	TRANSIENT_LOCAL	TRANSIENT_LOCAL	TRANSIENT_LOCAL
Reliability	BEST_EFFORT	BEST_EFFORT	BEST_EFFORT
	BEST_EFFORT	RELIABLE	Not compatible
	RELIABLE	BEST_EFFORT	BEST_EFFORT
	RELIABLE	RELIABLE	RELIABLE

F. SECURITY SETTINGS

SROS 2 makes use of three DDS-security SPIs, namely Authentication, Access Control and Cryptography. The DDS-security specifications provide compliance points associated with the implementation of the SPIs by vendors. Each vendor implements the specifications using plugins, which can be replaced by the user if so desired. However, all plugins must conform to the DDS-security specifications [10]. DDS-security provides the option of choosing which security plugins to use, including allowing configuration of which parts of the RTPS messages need to be encrypted. SROS 2, however, currently mandates the use of all three SPIs, without providing for finer control over the security plugins that are used. A user can only either turn on all three SPIs on (turn security on), or not have any security at all (turn security off).

The following paragraphs elaborate on the security plugins provided by the Fast-RTPS DDS that is shipped by default with ROS 2. Nonetheless, the implementation of the SPIs is similar to other DDS vendors. By default, security support is not compiled for Fast-RTPS, and must be activated by adding “-DSECURITY=ON” when compiling Fast-RTPS.

1. Authentication

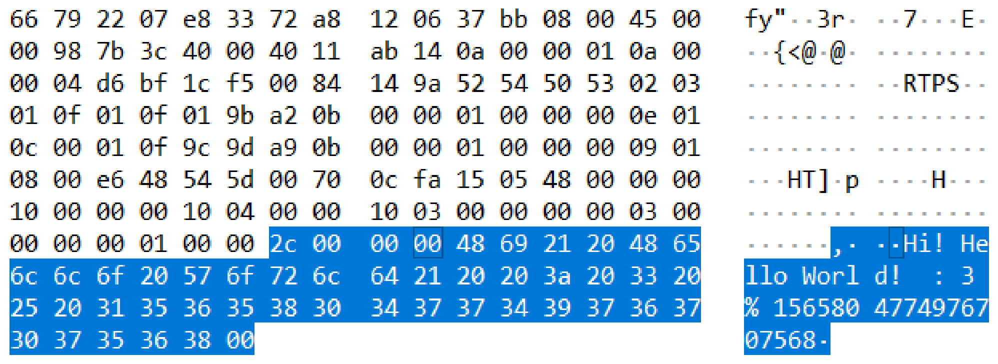
The authentication plugin allows for mutual authentication between discovered nodes. After initial discovery, authentication must be completed before information can be exchanged between nodes. A trusted Certificate Authority (CA) is used as part of the authentication process. The Elliptic Curve Digital Signature Algorithm is used to generate the public key [18]. The Elliptic Curve Diffie-Hellman Key Agreement Method is then used to derive a shared key between both nodes.

2. Access Control

After a node is authenticated, validation of its permission is performed. Permissions for each node are configured through the use of two XML documents in the filesystem: governance.p7s and permissions.p7s. Both documents must be signed by the trusted CA using a X.509 certificate. The governance document provides control over how access control is enforced, while the permissions document expresses the type of access granted to each type of node for each specific topic.

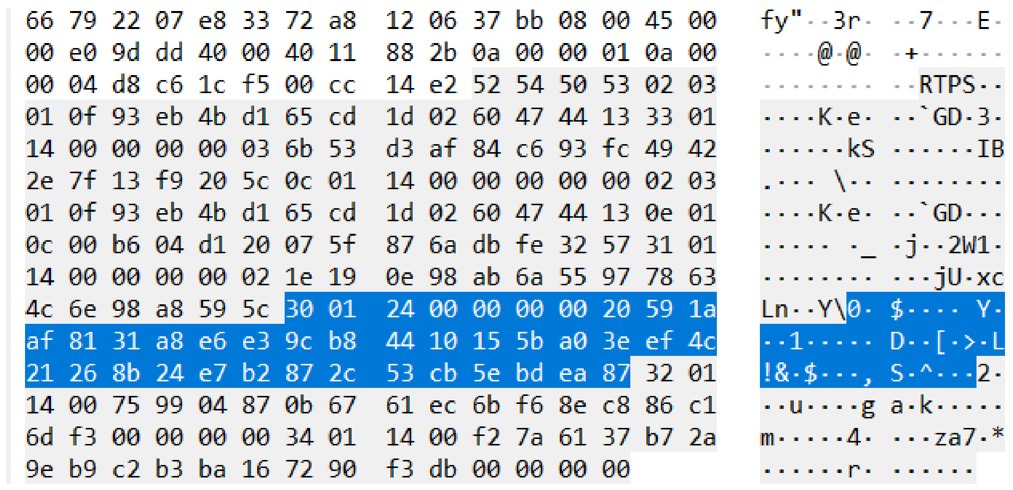
3. Cryptography

The cryptography plugin used by Fast-RTPS provides authenticated encryption using Advanced Encryption Standard in Galois Counter Mode [18]. Message authentication is provided through message authentication codes (MACs) using Galois MAC. In Figure 6, a Wireshark capture of a message being sent in the clear is shown, while Figure 7 illustrates the Wireshark capture of the same message being encrypted.



```
66 79 22 07 e8 33 72 a8  12 06 37 bb 08 00 45 00   fy"··3r· ··7···E·
00 98 7b 3c 40 00 40 11  ab 14 0a 00 00 01 0a 00   ··{<@·@· ········
00 04 d6 bf 1c f5 00 84  14 9a 52 54 50 53 02 03   ········ ··RTPS··
01 0f 01 0f 01 9b a2 0b  00 00 01 00 00 00 0e 01   ········ ········
0c 00 01 0f 9c 9d a9 0b  00 00 01 00 00 00 09 01   ········ ········
08 00 e6 48 54 5d 00 70  0c fa 15 05 48 00 00 00   ···HT]·p ····H···
10 00 00 00 10 04 00 00  10 03 00 00 00 00 03 00   ········ ········
00 00 00 01 00 00 2c 00  00 00 48 69 21 20 48 65   ······,· ··Hi! He
6c 6c 6f 20 57 6f 72 6c  64 21 20 20 3a 20 33 20   llo Worl d!  : 3 
25 20 31 35 36 35 38 30  34 37 37 34 39 37 36 37   % 156580 47749767
30 37 35 36 38 00                                  07568·
```

Figure 6. Wireshark capture of message sent in the clear

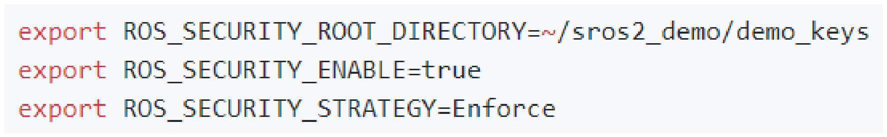


```
66 79 22 07 e8 33 72 a8  12 06 37 bb 08 00 45 00   fy"··3r· ··7···E·
00 e0 9d dd 40 00 40 11  88 2b 0a 00 00 01 0a 00   ····@·@· ·+······
00 04 d8 c6 1c f5 00 cc  14 e2 52 54 50 53 02 03   ········ ··RTPS··
01 0f 93 eb 4b d1 65 cd  1d 02 60 47 44 13 33 01   ····K·e· ··`GD·3·
14 00 00 00 00 03 6b 53  d3 af 84 c6 93 fc 49 42   ······kS ······IB
2e 7f 13 f9 20 5c 0c 01  14 00 00 00 00 00 02 03   .··· \·· ········
01 0f 93 eb 4b d1 65 cd  1d 02 60 47 44 13 0e 01   ····K·e· ··`GD···
0c 00 b6 04 d1 20 07 5f  87 6a db fe 32 57 31 01   ····· ·_ ·j··2W1·
14 00 00 00 00 02 1e 19  0e 98 ab 6a 55 97 78 63   ········ ···jU·xc
4c 6e 98 a8 59 5c 30 01  24 00 00 00 00 20 59 1a   Ln··Y\0· $···· Y·
af 81 31 a8 e6 e3 9c b8  44 10 15 5b a0 3e ef 4c   ··1····· D··[·>·L
21 26 8b 24 e7 b2 87 2c  53 cb 5e bd ea 87 32 01   !&·$···, S·^···2·
14 00 75 99 04 87 0b 67  61 ec 6b f6 8e c8 86 c1   ··u····g a·k·····
6d f3 00 00 00 00 34 01  14 00 f2 7a 61 37 b7 2a   m·····4· ···za7·*
9e b9 c2 b3 ba 16 72 90  f3 db 00 00 00 00         ······r· ······
```

Figure 7. Wireshark capture of message encrypted

G. CONFIGURING SECURITY SETTINGS IN ROS 2

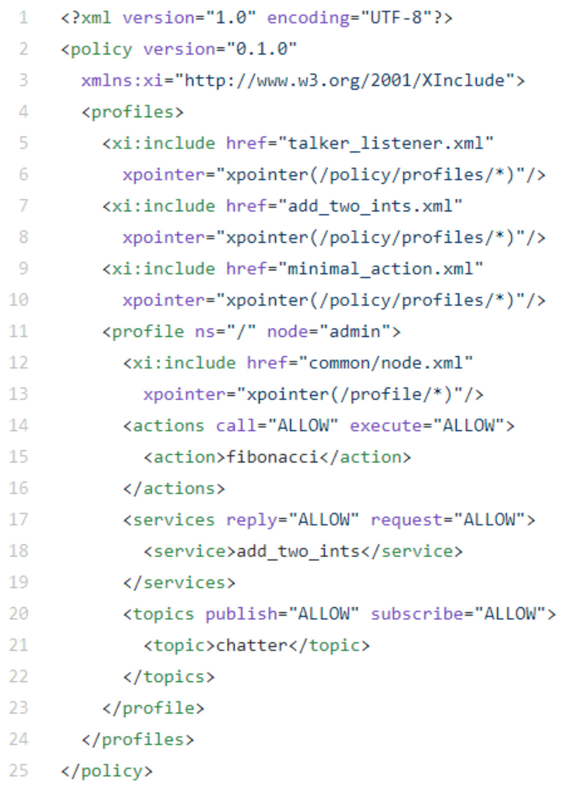
Security in ROS 2 can be turned on and off by configuring the ROS 2 environment variables. Figure 8 shows the environment variables that need to be configured. ROS_SECURITY_ROOT_DIRECTORY states the location where all security policies and keys are stored. ROS_SECURITY_ENABLE toggles whether security is turned on or off. If ROS_SECURITY_STRATEGY is set to Enforced, the node checks to ensure that the security keys are present. If it is not set as enforced, it allows the node to fall back to no security when the security keys are absent.



```
export ROS_SECURITY_ROOT_DIRECTORY=~/sros2_demo/demo_keys
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

Figure 8. Commands to configure ROS 2 environment variables to enable security

The “ros2 security create_permission” command is used to generate relevant security XML files expected by the DDS-security. A sample security policy file used to generate the relevant security XML keys is shown in Figure 9. The security policy allows for configuration of which nodes have access to which specific topics. In the Dashing Diademata release, keys are simply stored in the filesystem, with the ROS 2 roadmap promising to address key storage security [4]. Manually attempting to amend the security XML files to get finer control over the security settings used by the DDS would result in errors in the ROS 2 application.



```
<?xml version="1.0" encoding="UTF-8"?>
<policy version="0.1.0"
  xmlns:xi="http://www.w3.org/2001/XInclude">
  <profiles>
    <xi:include href="talker_listener.xml"
      xpointer="xpointer(/policy/profiles/*)"/>
    <xi:include href="add_two_ints.xml"
      xpointer="xpointer(/policy/profiles/*)"/>
    <xi:include href="minimal_action.xml"
      xpointer="xpointer(/policy/profiles/*)"/>
    <profile ns="/" node="admin">
      <xi:include href="common/node.xml"
        xpointer="xpointer(/profile/*)"/>
      <actions call="ALLOW" execute="ALLOW">
        <action>fibonacci</action>
      </actions>
      <services reply="ALLOW" request="ALLOW">
        <service>add_two_ints</service>
      </services>
      <topics publish="ALLOW" subscribe="ALLOW">
        <topic>chatter</topic>
      </topics>
    </profile>
  </profiles>
</policy>
```

Figure 9. Example of a security policy. Adapted from [19].

H. SYSTEM SETUP

Simulations were performed on a single computer. The computer used for the simulations had the following hardware:

- Processor: Intel(R) Core (TM) i7-8700K CPU @ 4.60GHz (6 cores)
- Memory: 32 GB DDR4
- OS: Ubuntu 18.04
- ROS 2.0 version: Dashing Diademata Patch Release 1
- NS-3 version: NS-3.29

IV. SIMULATION AND ANALYSIS OF RESULTS

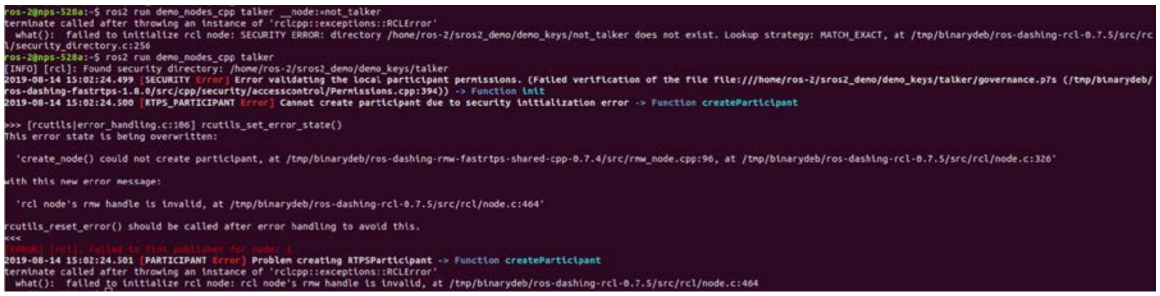
In this chapter, we discuss the results obtained from performing simulations using different QoS policies and with the security setting switched on or off. We also discuss the difference in network performance between having two and five ROS 2 nodes, in order to evaluate the impact of node scaling on the network performance.

A. VALIDATION OF SECURITY SETTINGS

In [20], the author demonstrated that the redesign of ROS 2 would enable it to be effective against rogue nodes and message spoofing. We modify the security certificates to verify that the security settings correctly protect against rogue nodes.

1. Authentication

Figure 10 depicts two scenarios in which a ROS 2 node fails to correctly authenticate itself, and the error messages that ROS 2 produces. The first scenario demonstrates how the node fails to initialize when no security certificate matches to the node. The second scenario demonstrates how the node fails the security initialization when the security keys are amended manually. This simulates a rogue node with a fake certificate.

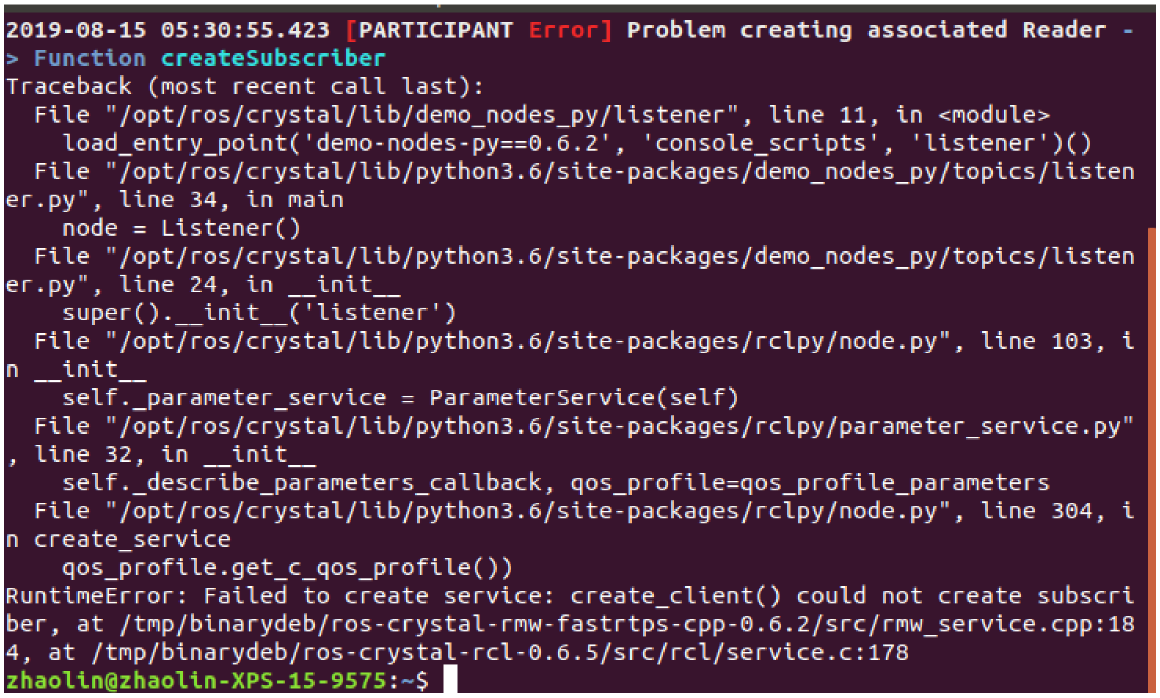


```
ros-2@nps-528a:~$ ros2 run demo_nodes_cpp talker __node:=not_talker
terminate called after throwing an instance of 'rclcpp::exceptions::RCLError'
  what():  failed to initialize rcl node: SECURITY ERROR: directory /home/ros-2/sros2_demo/demo_keys/not_talker does not exist. Lookup strategy: MATCH_EXACT, at /tmp/binarydeb/ros-dashing-rcl-0.7.5/src/rc
l/security_directory.c:256
ros-2@nps-528a:~$ ros2 run demo_nodes_cpp talker
[INFO] [rcl]: Found security directory: /home/ros-2/sros2_demo/demo_keys/talker
2019-08-14 15:02:24.499 [SECURITY Error] Error validating the local participant permissions. (Failed verification of the file file:///home/ros-2/sros2_demo/demo_keys/talker/governance.p7s (/tmp/binarydeb/
ros-dashing-fastrtps-1.8.0/src/cpp/security/accesscontrol/Permissions.cpp:394)) -> Function init
2019-08-14 15:02:24.500 [RTPS_PARTICIPANT Error] Cannot create participant due to security initialization error -> Function createParticipant

>>> [rcutils|error_handling.c:106] rcutils_set_error_state()
This error state is being overwritten:

  'create_node() could not create participant, at /tmp/binarydeb/ros-dashing-rmw-fastrtps-shared-cpp-0.7.4/src/rmw_node.cpp:96, at /tmp/binarydeb/ros-dashing-rcl-0.7.5/src/rcl/node.c:326'

with this new error message:

  'rcl node's rmw handle is invalid, at /tmp/binarydeb/ros-dashing-rcl-0.7.5/src/rcl/node.c:464'

rcutils_reset_error() should be called after error handling to avoid this.
<<<
[ERROR] [rcl]: Failed to fini publisher for node: 1
2019-08-14 15:02:24.501 [PARTICIPANT Error] Problem creating RTPSParticipant -> Function createParticipant
terminate called after throwing an instance of 'rclcpp::exceptions::RCLError'
  what():  failed to initialize rcl node: rcl node's rmw handle is invalid, at /tmp/binarydeb/ros-dashing-rcl-0.7.5/src/rcl/node.c:464
```

Figure 10. Error messages when the node fails to authenticate

2. Access Control

Access to specific keys is dictated by an XML file used to create the security keys for the nodes. The XML file specifies the list of topics that each node has access to, either as a Publisher or as a Subscriber. Figure 11 shows the error given by ROS 2 when the node is attempting to access a node that it is not authorized to access. This results in the node failing to initialize.



```
2019-08-15 05:30:55.423 [PARTICIPANT Error] Problem creating associated Reader -
> Function createSubscriber
Traceback (most recent call last):
  File "/opt/ros/crystal/lib/demo_nodes_py/listener", line 11, in <module>
    load_entry_point('demo-nodes-py==0.6.2', 'console_scripts', 'listener')()
  File "/opt/ros/crystal/lib/python3.6/site-packages/demo_nodes_py/topics/listen
er.py", line 34, in main
    node = Listener()
  File "/opt/ros/crystal/lib/python3.6/site-packages/demo_nodes_py/topics/listen
er.py", line 24, in __init__
    super().__init__('listener')
  File "/opt/ros/crystal/lib/python3.6/site-packages/rclpy/node.py", line 103, i
n __init__
    self._parameter_service = ParameterService(self)
  File "/opt/ros/crystal/lib/python3.6/site-packages/rclpy/parameter_service.py"
, line 32, in __init__
    self._describe_parameters_callback, qos_profile=qos_profile_parameters
  File "/opt/ros/crystal/lib/python3.6/site-packages/rclpy/node.py", line 304, i
n create_service
    qos_profile.get_c_qos_profile())
RuntimeError: Failed to create service: create_client() could not create subscri
ber, at /tmp/binarydeb/ros-crystal-rmw-fastrtps-cpp-0.6.2/src/rmw_service.cpp:18
4, at /tmp/binarydeb/ros-crystal-rcl-0.6.5/src/rcl/service.c:178
zhaolin@zhaolin-XPS-15-9575:~$
```

Figure 11. Screenshot of ROS 2 error when a node attempts to connect to an unauthorized topic

B. EVALUATION METHODOLOGY FOR RESULTS

In order to observe whether NS-3 correctly simulates packet loss in a wireless network, we performed a simulation using NS-3 without ROS 2. Packet sizes of 60 bytes were sent between two nodes in NS-3. Figure 12 illustrates the average percentage of packet loss at different distances for an 802.11a ad-hoc network. The results indicate that packet loss begins to increase at the 25 meter mark.

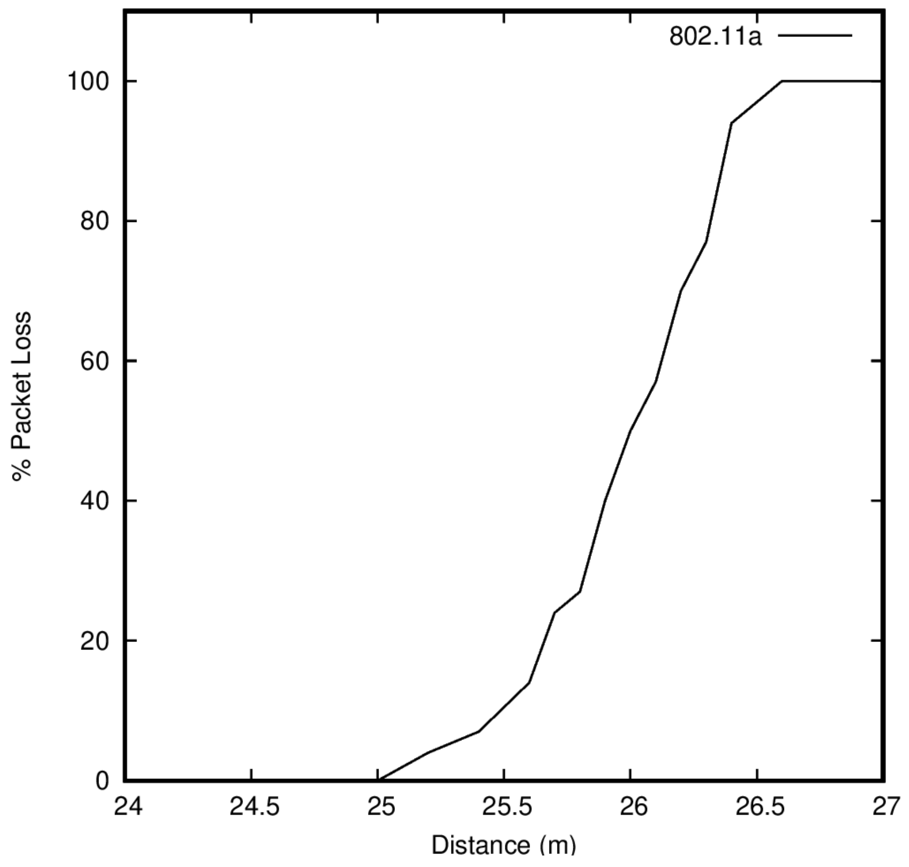


Figure 12. Rate of packet loss versus Distance (m) as simulated in NS-3

Next, sets of simulations were carried out, each with a specific QoS profile and security setting. Each set consisted of a series of simulation runs in which the nodes were placed at different distances from one another. With different distances, the wireless network has a different simulated packet drop rate, which affects the message transmissions in ROS 2.

For each simulation run, ROS 2 messages were published by a single Publisher node at a frequency of 2 Hz. Each simulation ran until either 200 messages were received by the Subscriber nodes or a time-out error was reached. Each message was 45 bytes long and consisted of a generic “Hello World!” string. A counter and time stamp were also appended to the message. For each run, the rate of message loss and the average latency incurred by each message was recorded.

1. Rate of Message Loss

The rate of message loss is defined as the ratio of messages received by the Subscriber to the total messages that the Subscriber was supposed to receive. Each published message is stamped with a counter indicating the index of the message. Based on the counter in the message, the Subscriber node determines whether any message was not properly received. The Subscriber then compares the counter of the message received to that of the last message received. If the counter is not in running sequence, the Subscriber is able to determine the number of messages that were lost.

2. Latency

Latency is defined as the delay between the time that a specific message is published by a Publisher and the time that it is read by the Subscriber. Each message includes the system time of when the message was prepared. When the Subscriber receives a message, it compares the current system time with the system time included in the message. In this way, the measured latency includes the delay incurred by RMW to translate the message, the delay incurred by the DDS middleware to process each message packet, as well as the actual network propagation delay.

C. NETWORK PERFORMANCE WITH DIFFERENT QOS SETTINGS

We used the ROS 2 QoS profiles shown in Table 3 as a baseline to measure network performance. We ran additional simulations with QoS policies set at custom values in order to measure the specific impact of changing the settings of that specific QoS policy. Simulations were initially performed with one Publisher and one Subscriber node.

1. Message Loss Rate

The message loss rates for the three QoS profiles shipped with ROS 2 are shown in Figure 13. With a Sensor profile, the message loss rate starts to increase gradually as the distance between nodes is increased. This is to be expected as the Sensor profile utilizes a Best_Effort policy. As the distance between nodes increases, the rate of packet loss increases, similar to what is shown in Figure 12. Thus, the chance of a message being dropped increases as well.

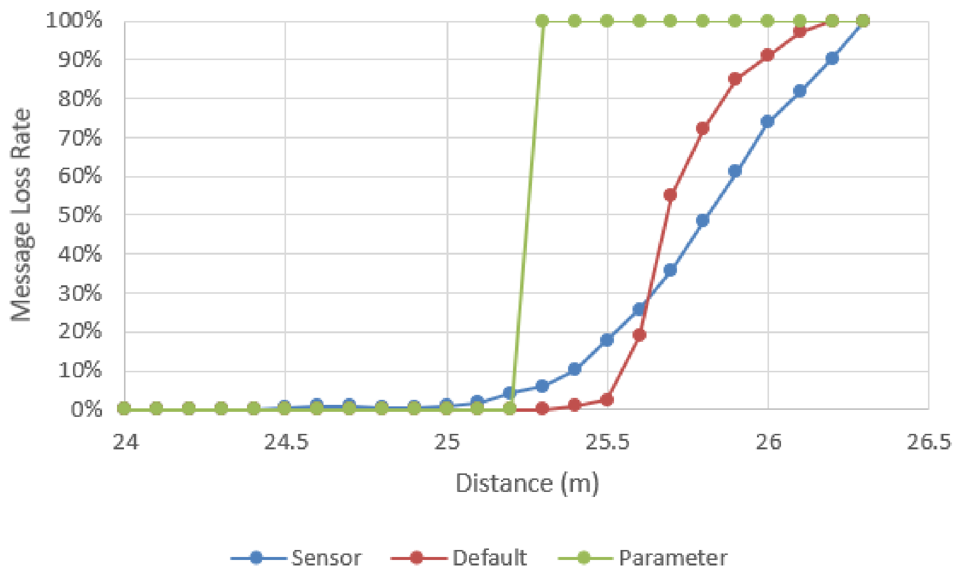
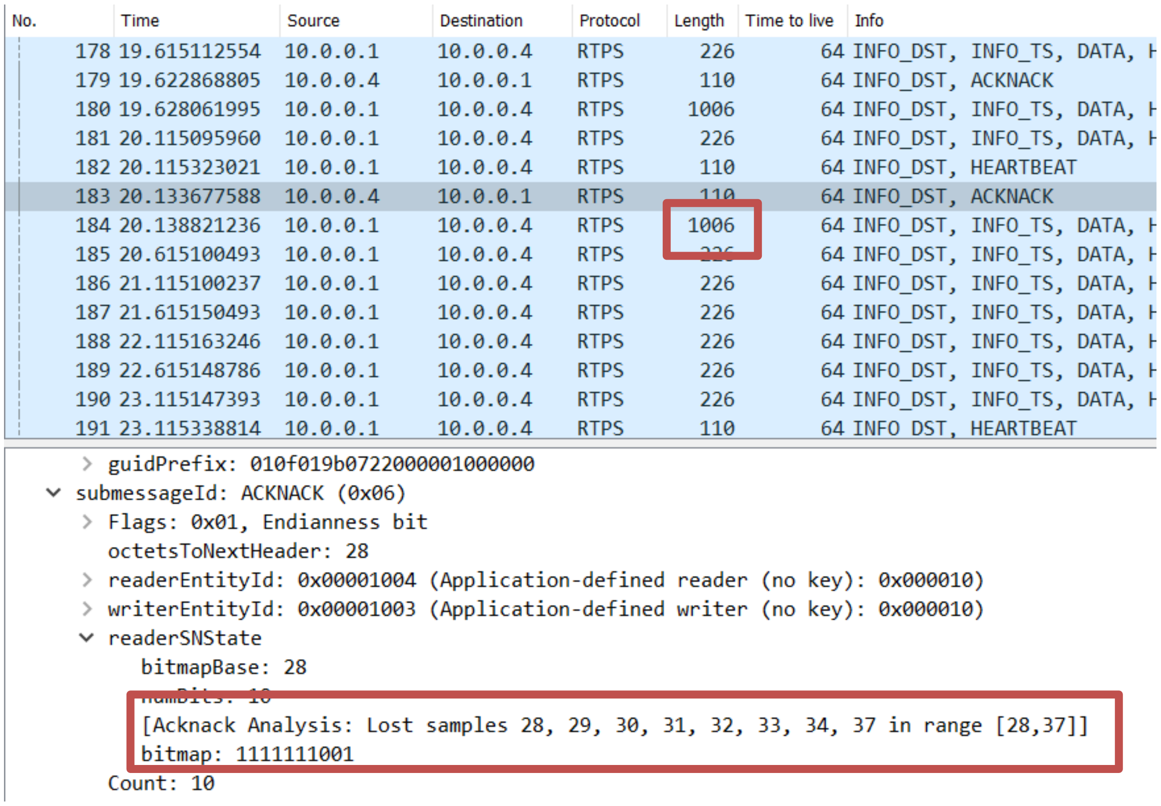


Figure 13. Message Loss Rate versus Distance for QoS profiles shipped with ROS 2

The Default profile has a Depth = 10 (see Table 3). The Default profile produced a lower message drop rate until 25.7m, as compared to using the Sensor profile. The Default profile utilizes a Reliable policy, which resends the messages if the Publisher does not receive an acknowledgment message from the Subscriber. Figure 14 shows the Wireshark capture of messages sent to and from the Publisher node. The Publisher regularly sends a heartbeat message to the Subscriber to check that the Subscriber is still connected to the network. The Subscriber then sends an acknowledgment message, either in the reply to the original message sent, or as a reply to the heartbeat message. The acknowledgment message includes a list of messages that have not been received by the Subscriber. The Publisher then immediately sends out these messages again, as long as the messages are still available in the writer cache.



```
No.   Time          Source    Destination  Protocol  Length  Time to live  Info
  178 19.615112554  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  179 19.622868805  10.0.0.4  10.0.0.1     RTPS         110            64 INFO_DST, ACKNACK
  180 19.628061995  10.0.0.1  10.0.0.4     RTPS        1006            64 INFO_DST, INFO_TS, DATA, H
  181 20.115095960  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  182 20.115323021  10.0.0.1  10.0.0.4     RTPS         110            64 INFO_DST, HEARTBEAT
  183 20.133677588  10.0.0.4  10.0.0.1     RTPS         110            64 INFO_DST, ACKNACK
  184 20.138821236  10.0.0.1  10.0.0.4     RTPS        1006            64 INFO_DST, INFO_TS, DATA, H
  185 20.615100493  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  186 21.115100237  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  187 21.615150493  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  188 22.115163246  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  189 22.615148786  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  190 23.115147393  10.0.0.1  10.0.0.4     RTPS         226            64 INFO_DST, INFO_TS, DATA, H
  191 23.115338814  10.0.0.1  10.0.0.4     RTPS         110            64 INFO_DST, HEARTBEAT

    > guidPrefix: 010f019b0722000001000000
  v submessageId: ACKNACK (0x06)
    > Flags: 0x01, Endianness bit
      octetsToNextHeader: 28
    > readerEntityId: 0x00001004 (Application-defined reader (no key): 0x000010)
    > writerEntityId: 0x00001003 (Application-defined writer (no key): 0x000010)
    v readerSNState
        bitmapBase: 28
        numBits: 10
        [Acknack Analysis: Lost samples 28, 29, 30, 31, 32, 33, 34, 37 in range [28,37]]
        bitmap: 1111111001
      Count: 10
```

Figure 14. Wireshark capture of packets sent from the Publisher node

Yet, because the Publisher retransmits all the missing messages in a single packet, the message loss rate becomes larger than that experienced by the Sensor profile when the distance between nodes is larger than 25.7m. As seen in the highlighted row of Figure 14, the acknowledgment message in packet number 183 indicates that eight out of the past ten messages were not received. These eight messages were then retransmitted together in the next message, resulting in a packet size of 1006 bytes (shown in Figure 14). This is much larger than if only a single message was sent. In a lossy network, the larger packet size results in a higher chance of packet loss. As such, the retransmission of messages was too large to be successfully transmitted to the Subscriber node.

With retransmission of messages usually not being successful after 25.7 m, the Default profile has a higher message drop rate than the Sensor profile. This is due to the Default profile transmitting the heartbeat message together with the message data in a single packet. This results in packets that are much larger than if the Sensor profile was used. The overall effect is a higher message drop rate using the Default profile.

The Parameter profile has a History policy of Keep_All and a Depth policy of 1000. As such, the packets that are sent are very large, as it would include both the current message as well as all past messages. In this situation, either the Subscriber receives all the messages or all the messages are dropped. It performs slightly better than the Sensor profile for a small set of distances (24.8 m–25.3 m), as the Subscriber can still retrieve any dropped messages from subsequent message deliveries. When the network gets more lossy with increased distance, however, none of the messages manage to be delivered; the packet drop rate for such a large packet size was very high.

In our next experiment, the Depth of the Default profile is changed to one, and the results of the simulations are compared to that of the original Default profile (Depth = 10). The results are shown in Figure 15. With Depth = 1, the network performs similarly to that of the Sensor profile, with only a slightly higher message loss rate. This is due to the fact that with a Depth policy of only one, the Subscriber is unable to retransmit most of the dropped messages, and thus initially exhibits a higher message drop rate. Since the packet sizes are smaller due to the smaller depth size, even with greater distance, more packets are delivered as compared to the profile with a large Depth (Depth = 10).

Although the message drop rate of the Default profile with a Depth of one is similar to that of the Sensor profile, it is overall slightly higher. This is because the packet sizes sent using the Default profile are higher than that of Sensor profile, due to the inclusion of the heartbeat message.

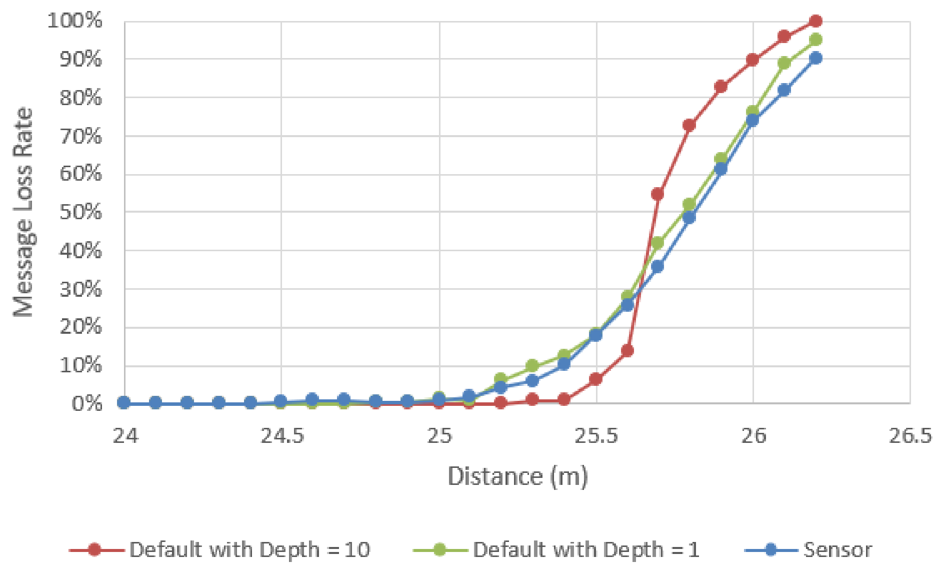


Figure 15. Comparison of Message Loss Rate when Depth = 1 and Depth = 10 for Default

2. Latency

The simulation results that measure the impact of the QoS profiles on latency are plotted on two separate graphs. The first graph plots the latency for distances from 0 to 24 m, while the second graph examines the latency from 24 m and beyond. As seen in Figure 12, given a packet size of 60 bytes, packet loss occurs starting at 25 m. As such, in order to review the impact that the QoS profiles have on a lossless network, we first evaluate the latency from 0 to 24 m for the different QoS profiles. We also separately look at the results between 24 and 26 m to review the impact QoS profiles have in a lossy wireless network.

The latency of the messages when the QoS profiles from Table 3 are used is shown in Figure 16. As distance is increased, the latency increases as well, due to the impact of propagation delay. The Sensor profile demonstrated the least latency, with the Default and Parameters profiles exhibiting a much higher latency. This higher latency is attributable to processing time by the RMW, which translates the messages from ROS 2 to IDL messages transmitted by the DDS middleware.

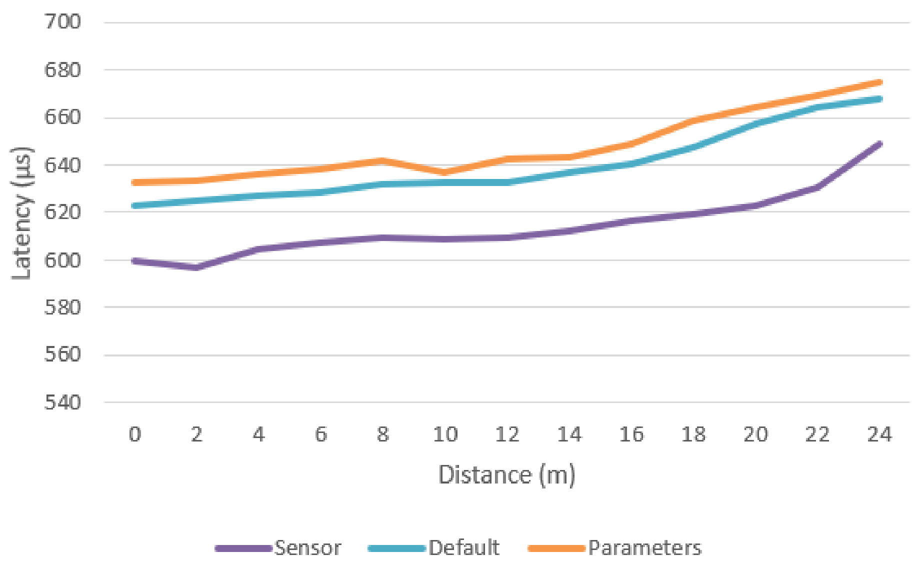


Figure 16. Latency of messages with different QoS profiles

We next compare the impact of the Default profile versus the Sensor profile on latency. The results are shown in Figure 17. As the distance increases, the network experiences increasing packet loss. Accordingly, the latency for messages with the Default profile increases significantly. This is due to the time incurred from the retransmission of messages not received by the Subscriber node. This can take multiple retransmissions in a lossy environment, with the Subscriber node receiving the message much later than when it was originally sent by the Publisher node.

There is no significant trend for using the Parameters profile, as all messages are dropped from distances 25.2 m and beyond (as was shown in Figure 13).

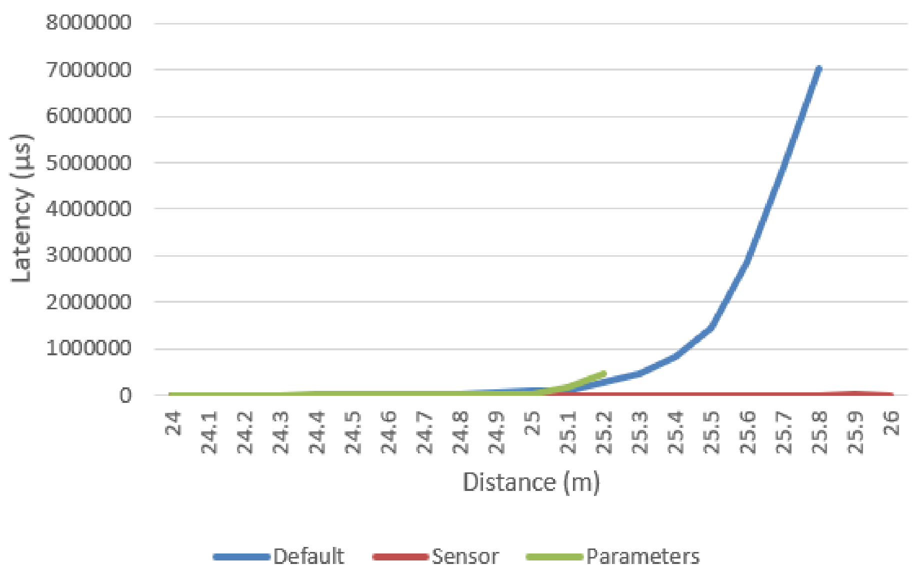


Figure 17. Latency of messages with different QoS profiles in a lossy network

D. SIMULATION RESULTS WITH SECURITY ON AND OFF

We next study the performance of ROS 2 when security is turned off or on. The results shown in Figures 18–20 depict the message loss rates for the QoS profiles from Table 3 with security turned on and off. Simulations were performed with one Publisher and one Subscriber node. It can be seen that for all profiles (Sensor, Default and Parameters), messages are dropped at a shorter distance when security is turned on as compared to when security is turned off. Wireshark captures of the transmitted packets using the Default profile, as shown in Figure 21 and Figure 22, illustrate the overhead incurred when security is turned off or on, respectively. The packet size when security is turned off is 170 bytes versus 330 bytes when security is turned on. The larger the packet size, the more likely the message is to be dropped.

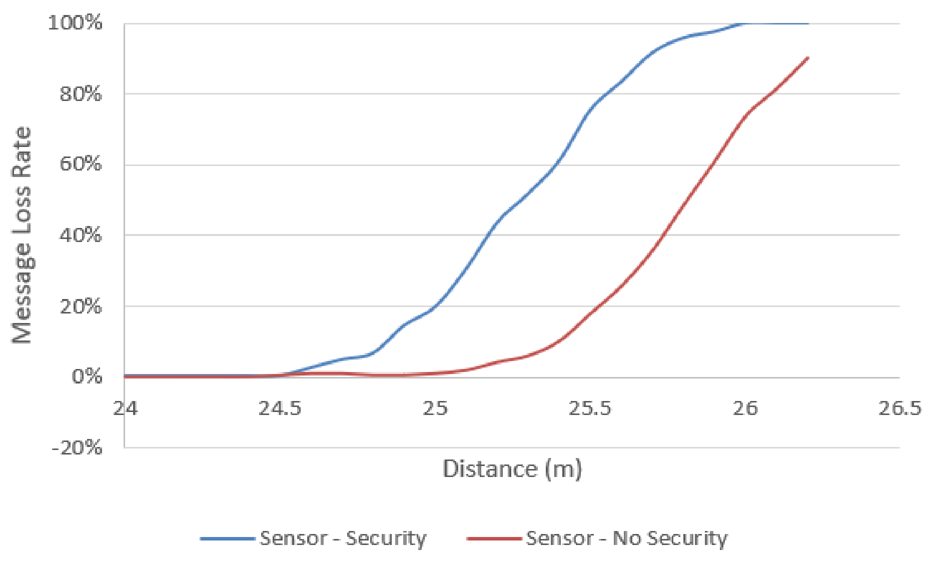


Figure 18. Message loss rate with security turned on and off using the Sensor profile

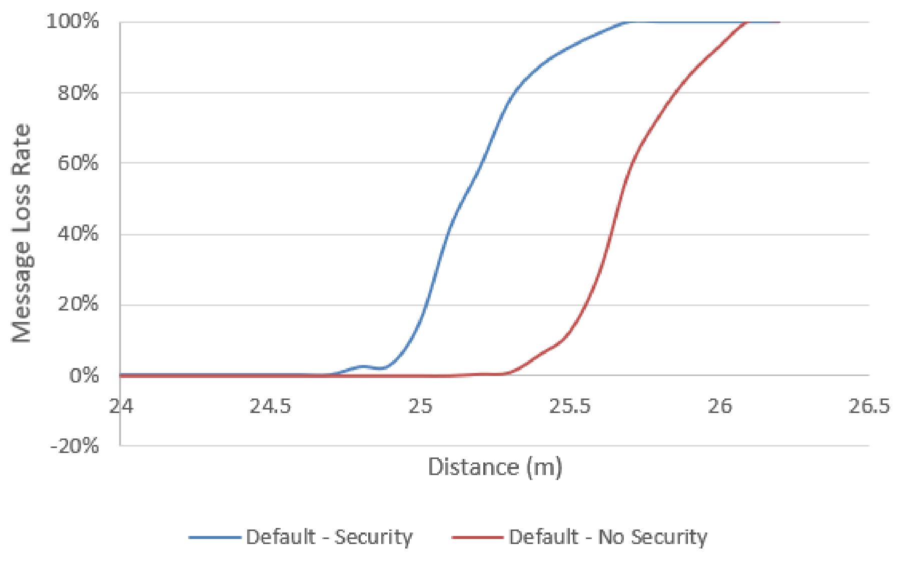


Figure 19. Message loss rate with security turned on and off using the Default profile

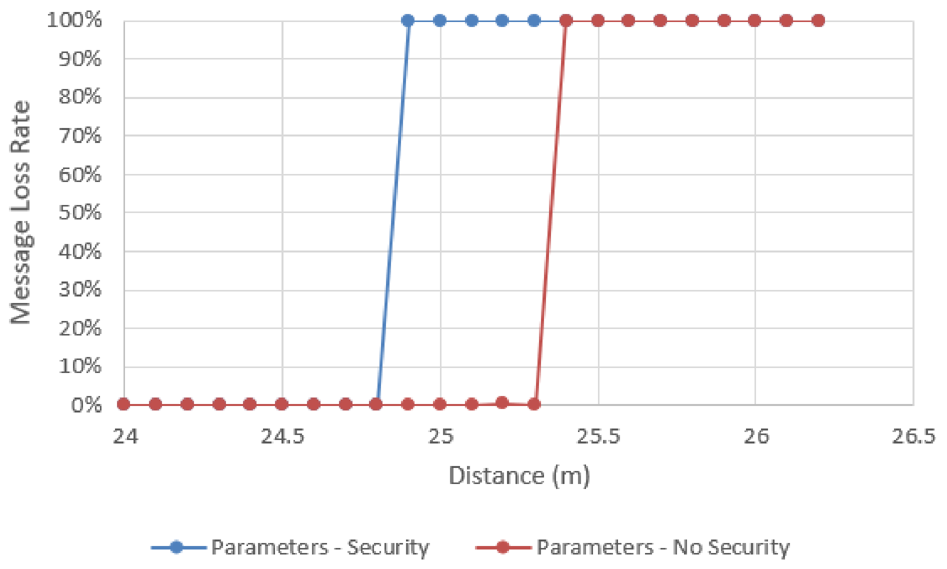
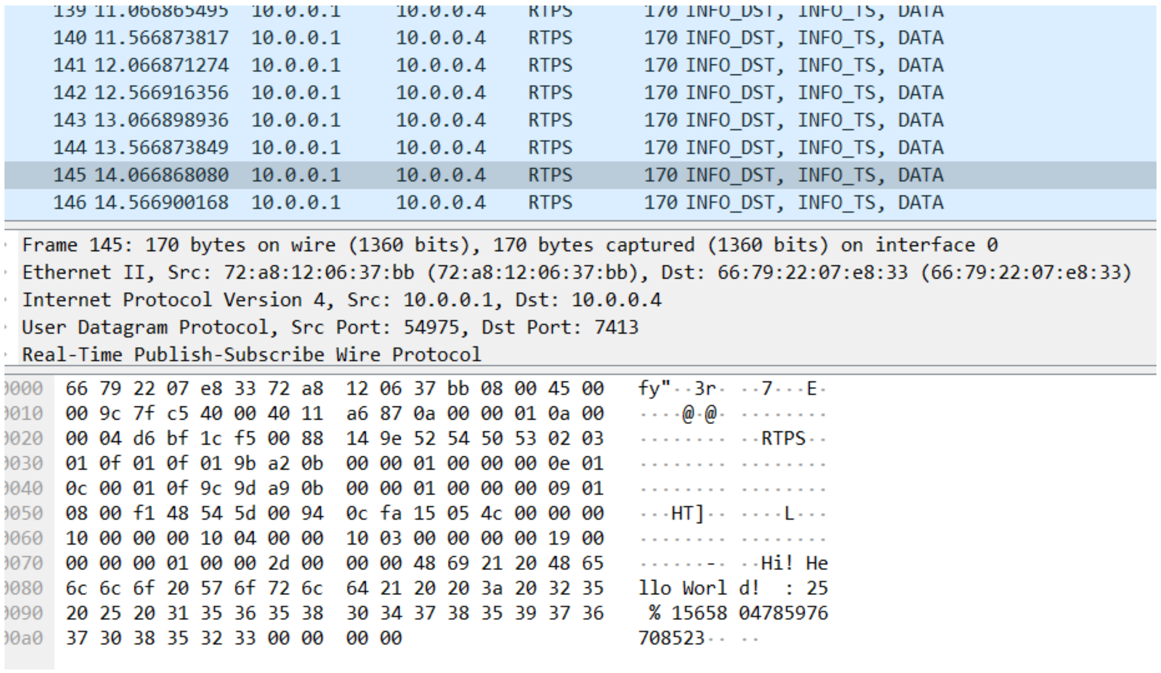
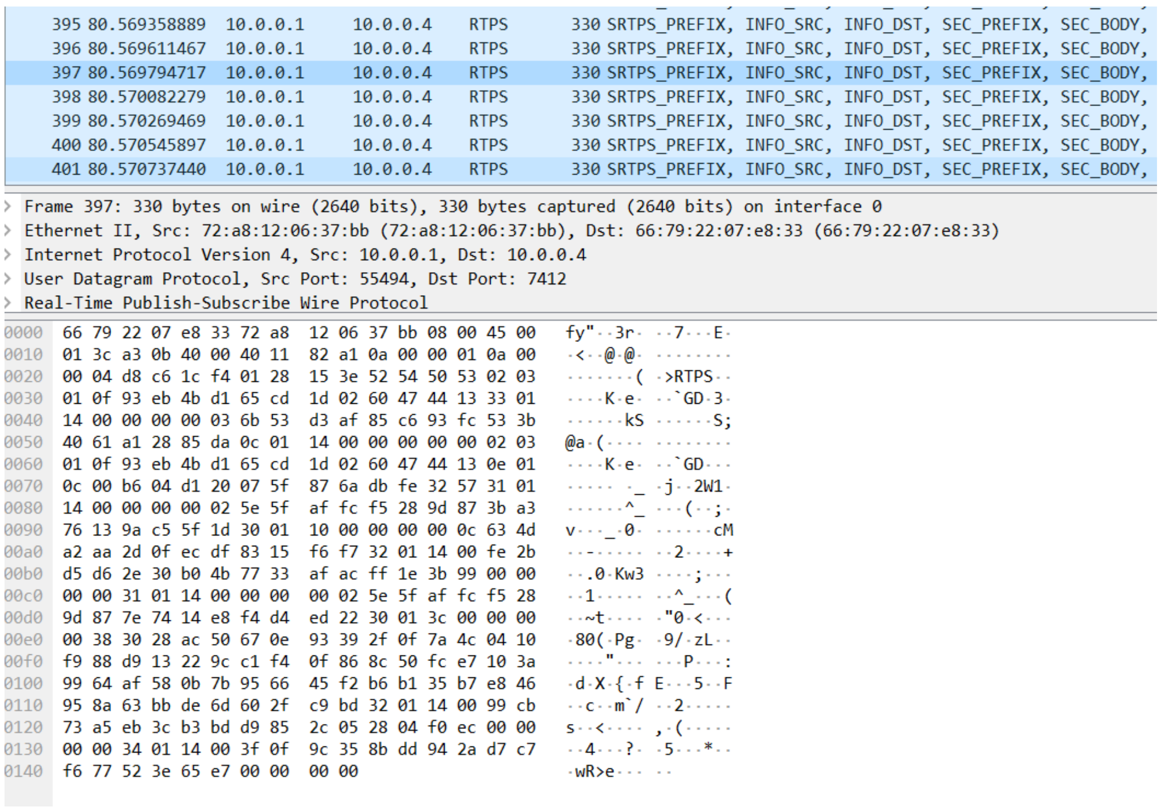


Figure 20. Message loss rate with security turned on and off using the Parameters profile



```
139 11.066865495  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
140 11.566873817  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
141 12.066871274  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
142 12.566916356  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
143 13.066898936  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
144 13.566873849  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
145 14.066868080  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA
146 14.566900168  10.0.0.1      10.0.0.4     RTPS       170 INFO_DST, INFO_TS, DATA

Frame 145: 170 bytes on wire (1360 bits), 170 bytes captured (1360 bits) on interface 0
Ethernet II, Src: 72:a8:12:06:37:bb (72:a8:12:06:37:bb), Dst: 66:79:22:07:e8:33 (66:79:22:07:e8:33)
Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4
User Datagram Protocol, Src Port: 54975, Dst Port: 7413
Real-Time Publish-Subscribe Wire Protocol

0000  66 79 22 07 e8 33 72 a8  12 06 37 bb 08 00 45 00   fy"··3r· ··7···E·
0010  00 9c 7f c5 40 00 40 11  a6 87 0a 00 00 01 0a 00   ····@·@· ········
0020  00 04 d6 bf 1c f5 00 88  14 9e 52 54 50 53 02 03   ········ ··RTPS··
0030  01 0f 01 0f 01 9b a2 0b  00 00 01 00 00 00 0e 01   ········ ········
0040  0c 00 01 0f 9c 9d a9 0b  00 00 01 00 00 00 09 01   ········ ········
0050  08 00 f1 48 54 5d 00 94  0c fa 15 05 4c 00 00 00   ···HT]·· ····L···
0060  10 00 00 00 10 04 00 00  10 03 00 00 00 00 19 00   ········ ········
0070  00 00 00 01 00 00 2d 00  00 00 48 69 21 20 48 65   ······-· ··Hi! He
0080  6c 6c 6f 20 57 6f 72 6c  64 21 20 20 3a 20 32 35   llo Worl d!  : 25
0090  20 25 20 31 35 36 35 38  30 34 37 38 35 39 37 36    % 15658 04785976
00a0  37 30 38 35 32 33 00 00  00 00                     708523·· ··
```

Figure 21. Wireshark capture showing size of messages with security turned off



```
395 80.569358889  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
396 80.569611467  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
397 80.569794717  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
398 80.570082279  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
399 80.570269469  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
400 80.570545897  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,
401 80.570737440  10.0.0.1    10.0.0.4    RTPS     330 SRTPS_PREFIX, INFO_SRC, INFO_DST, SEC_PREFIX, SEC_BODY,

> Frame 397: 330 bytes on wire (2640 bits), 330 bytes captured (2640 bits) on interface 0
> Ethernet II, Src: 72:a8:12:06:37:bb (72:a8:12:06:37:bb), Dst: 66:79:22:07:e8:33 (66:79:22:07:e8:33)
> Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.4
> User Datagram Protocol, Src Port: 55494, Dst Port: 7412
> Real-Time Publish-Subscribe Wire Protocol

0000  66 79 22 07 e8 33 72 a8  12 06 37 bb 08 00 45 00   fy"··3r· ··7···E·
0010  01 3c a3 0b 40 00 40 11  82 a1 0a 00 00 01 0a 00   ·<··@·@· ········
0020  00 04 d8 c6 1c f4 01 28  15 3e 52 54 50 53 02 03   ·······( ·>RTPS··
0030  01 0f 93 eb 4b d1 65 cd  1d 02 60 47 44 13 33 01   ····K·e· ··`GD·3·
0040  14 00 00 00 00 03 6b 53  d3 af 85 c6 93 fc 53 3b   ······kS ······S;
0050  40 61 a1 28 85 da 0c 01  14 00 00 00 00 00 02 03   @a·(···· ········
0060  01 0f 93 eb 4b d1 65 cd  1d 02 60 47 44 13 0e 01   ····K·e· ··`GD···
0070  0c 00 b6 04 d1 20 07 5f  87 6a db fe 32 57 31 01   ····· ·_ ·j··2W1·
0080  14 00 00 00 00 02 5e 5f  af fc f5 28 9d 87 3b a3   ······^_ ···(··;·
0090  76 13 9a c5 5f 1d 30 01  10 00 00 00 00 0c 63 4d   v···_·0· ······cM
00a0  a2 aa 2d 0f ec df 83 15  f6 f7 32 01 14 00 fe 2b   ··-····· ··2····+
00b0  d5 d6 2e 30 b0 4b 77 33  af ac ff 1e 3b 99 00 00   ··.0·Kw3 ····;···
00c0  00 00 31 01 14 00 00 00  00 02 5e 5f af fc f5 28   ··1····· ··^_···(
00d0  9d 87 7e 74 14 e8 f4 d4  ed 22 30 01 3c 00 00 00   ··~t···· ·"0·<···
00e0  00 38 30 28 ac 50 67 0e  93 39 2f 0f 7a 4c 04 10   ·80(·Pg· ·9/·zL··
00f0  f9 88 d9 13 22 9c c1 f4  0f 86 8c 50 fc e7 10 3a   ····"··· ···P···:
0100  99 64 af 58 0b 7b 95 66  45 f2 b6 b1 35 b7 e8 46   ·d·X·{·f E···5··F
0110  95 8a 63 bb de 6d 60 2f  c9 bd 32 01 14 00 99 cb   ··c··m`/ ··2·····
0120  73 a5 eb 3c b3 bd d9 85  2c 05 28 04 f0 ec 00 00   s··<···· ,·(·····
0130  00 00 34 01 14 00 3f 0f  9c 35 8b dd 94 2a d7 c7   ··4···?· ·5···*··
0140  f6 77 52 3e 65 e7 00 00  00 00                     ·wR>e··· ··
```

Figure 22. Wireshark capture showing size of messages with security turned on

In Figure 24 to Figure 26, we show the latency of the messages using the QoS profiles from Table 3 and when security is turned on and off. In Figure 21, it can be seen that from 0 to 24 m, the Sensor profile incurs an average of 19% overhead. The Default profile incurs an average 60% overhead for that same distance (Figure 24) while the Parameter profile incurs an average of 32% overhead for that same distance (Figure 25). There is a higher overhead incurred for the Default profile because the packet size is larger, and therefore, the process to encrypt the packet is more time consuming. The overhead incurred is also dependent on the processing power used to encrypt the messages.

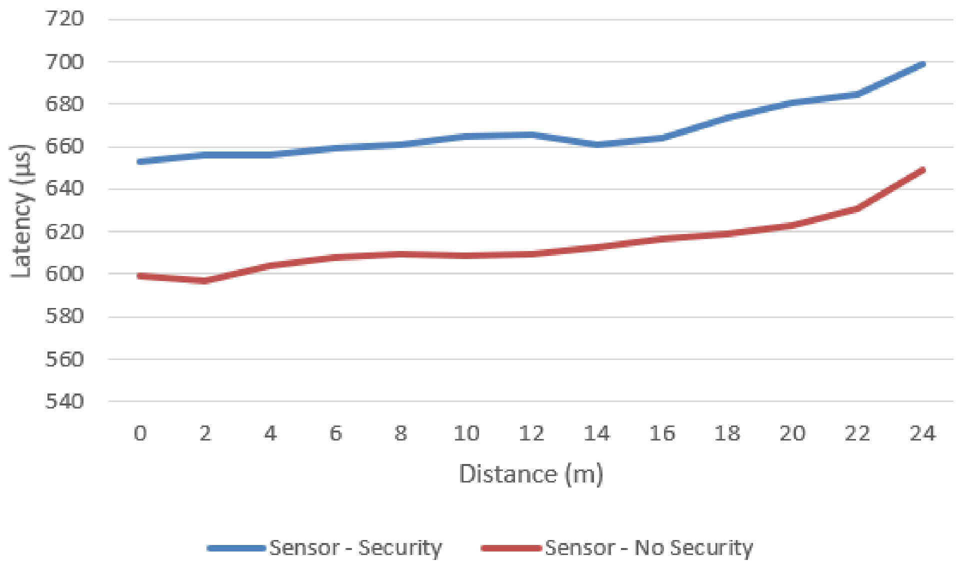


Figure 23. Latency of messages with the Sensor profile and with security turned on and off (0-24 m)

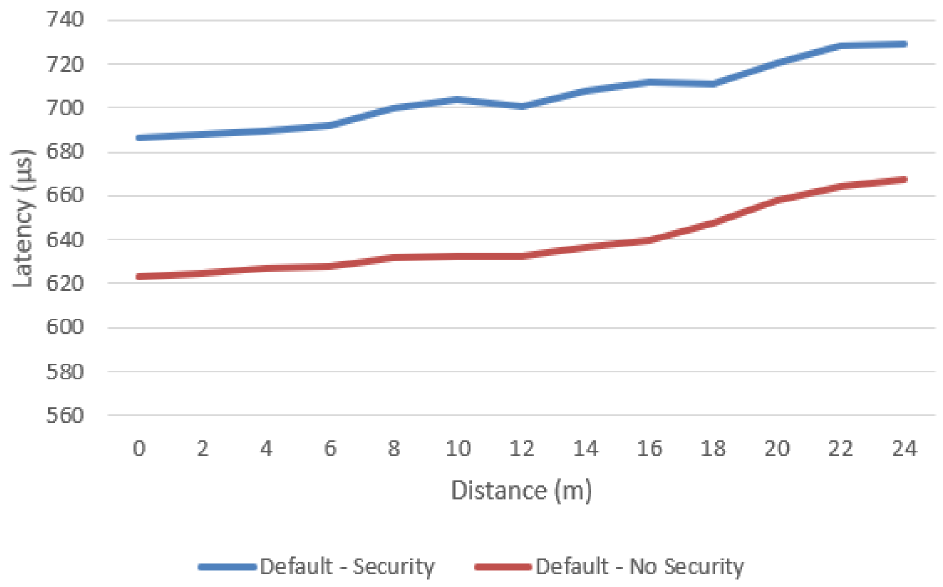


Figure 24. Latency of messages with the Default profile and with security turned on and off (0-24 m)

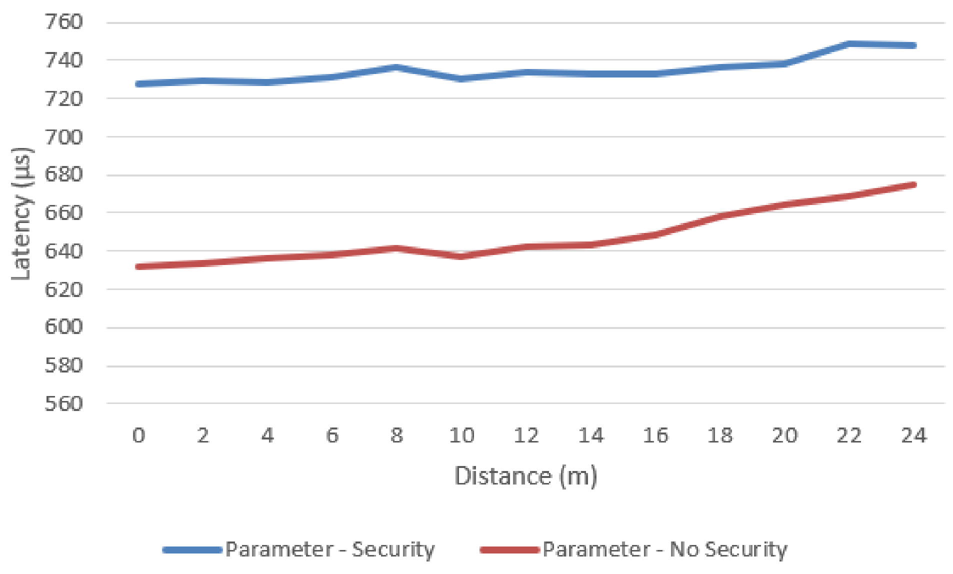


Figure 25. Latency of messages with the Parameter profile and with security turned on and off (0-24 m)

Figure 26 and Figure 27 show the impact on latency with security turned on and off for the Sensor profile and the Default profile in a lossy network, respectively. Latency increases for both profiles when security is turned on. In Figure 26, it can be seen that from 24 m, the latency starts to increase immediately with security turned on, as compared to 24.5 m with security turned off. Using the Default profile, latency starts to increase at 24.7 m with security turned on as compared to 25.2 m with security turned off, as shown in Figure 27. In order to achieve a comparable latency with security turned on for both profiles, the distance between nodes must be decreased by 0.5 m. For example, to achieve a latency of 4000 ms, the distance between nodes can go up to 25.7 m with security turned off. With security turned on, however, the distance between nodes can only go up to 25.2 m to achieve a latency of 4000 ms.

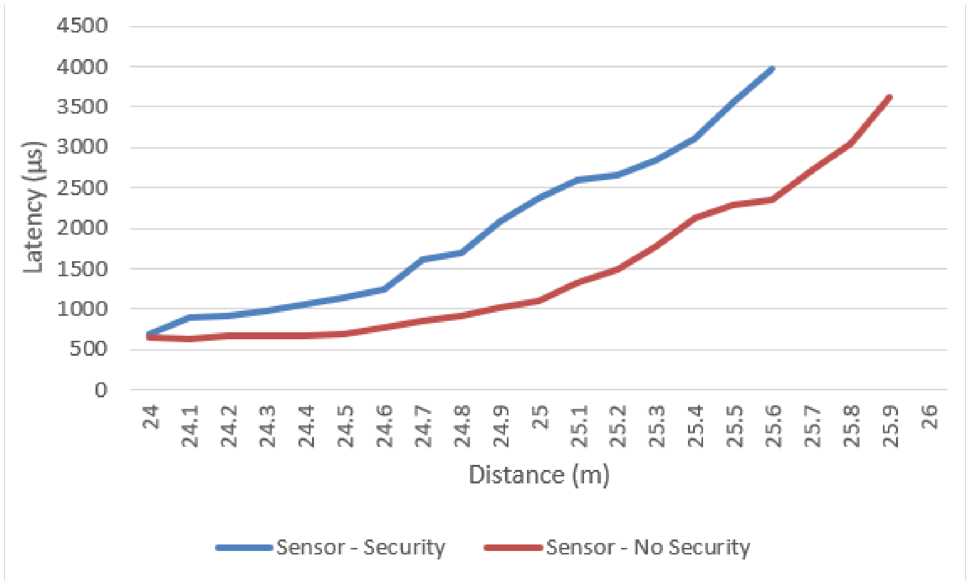


Figure 26. Latency of messages for the Sensor profile with security turned on and off (beyond 24 m)

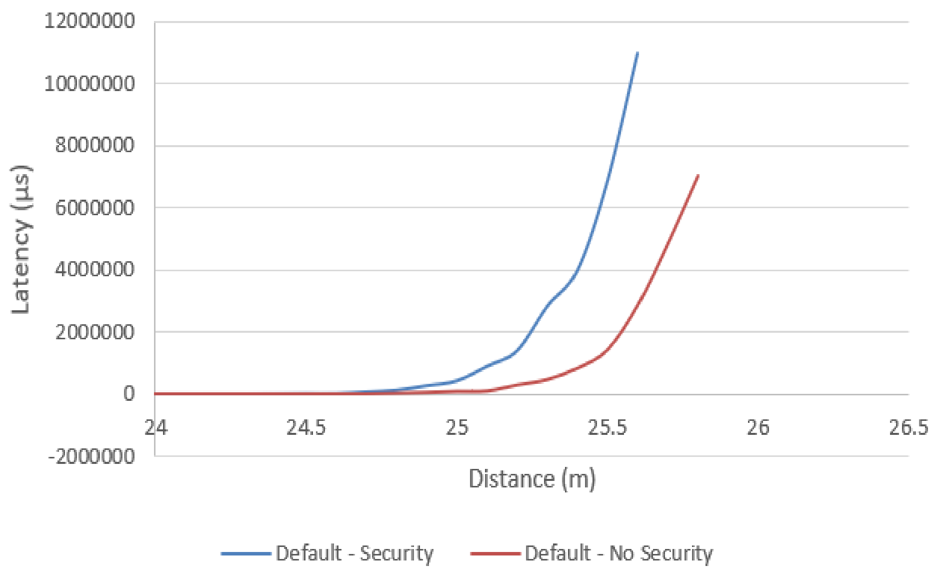


Figure 27. Latency of messages for the Default profile with security turned on and off (beyond 24 m)

E. IMPACT ON NETWORK SCALE ON PERFORMANCE

In this section, we compare the network performance of ROS 2 when the number of nodes in our simulations is increased to five.

1. Message Drop Rate

Figure 28 compares the message drop rates using the Sensor profile when the network has two nodes (one Publisher and one Subscriber) and five nodes (one Publisher and four Subscribers). The results show that as long as the network has sufficient bandwidth, the message drop rate is not affected when scaling up to more nodes when using the Sensor profile. Nevertheless, there is a difference when the Default profile is used. Figure 29 shows that five nodes have a lower message drop rate with the Default profile than when the network has only two nodes with the Default profile. When the network has only two nodes, the Publisher node sends out all unreceived messages in a single packet, as was shown in Figure 14. In Figure 30, the Wireshark capture for the Default profile with five nodes is shown. As seen in Figure 30, after the Publisher receives an acknowledgment message that the Subscriber has not received multiple messages, it sends out those messages in individual packets in sequence. This results in the Publisher resending dropped messages in smaller packets. These smaller packets are less likely to be dropped as compared to the large packets sent by the Publisher when there are only two nodes in the network.

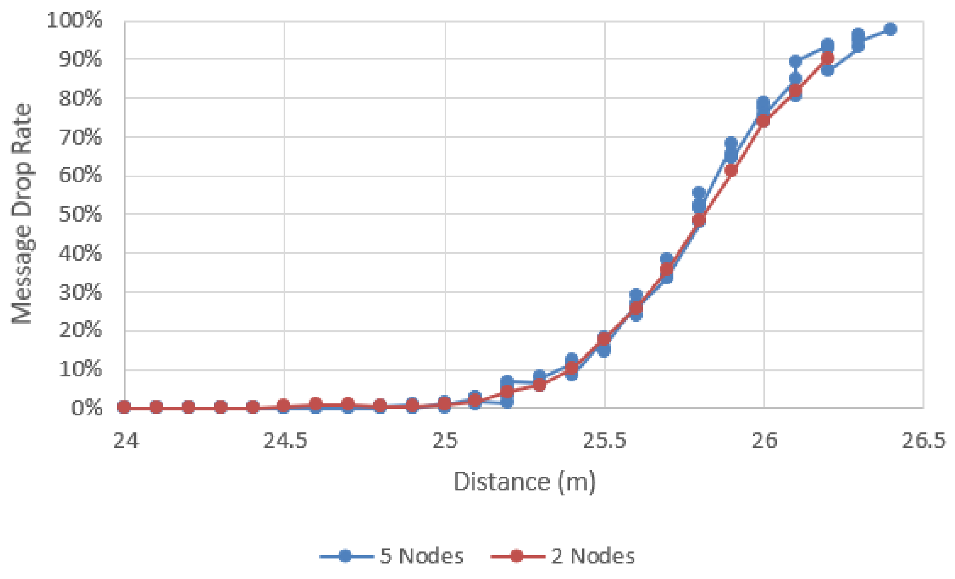


Figure 28. Message drop rates comparing two nodes and five nodes with the Sensor profile

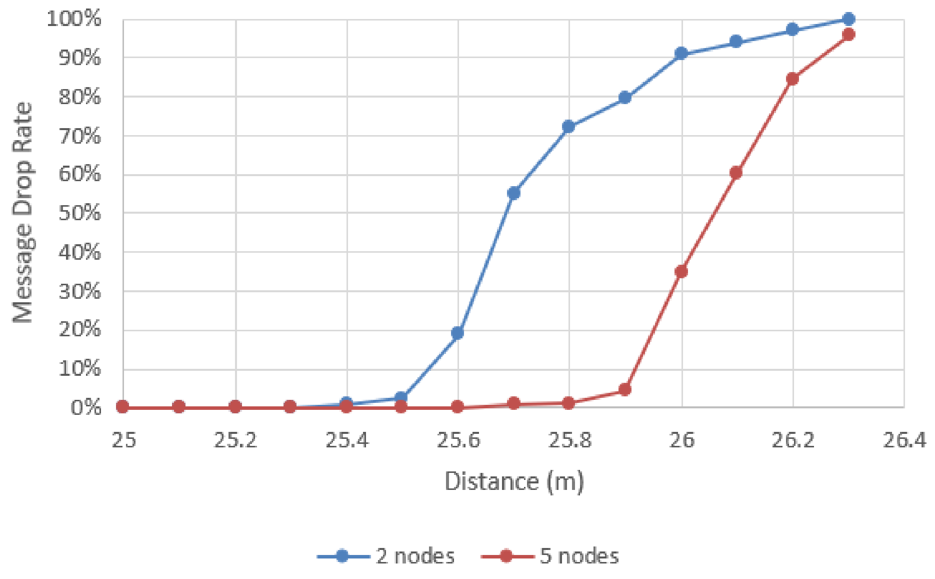
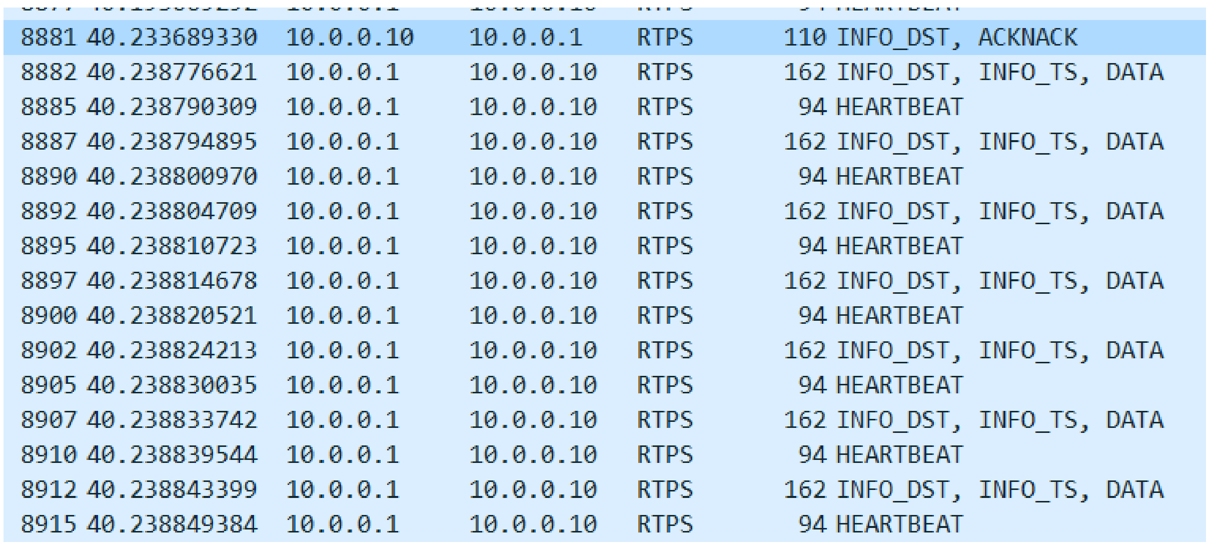


Figure 29. Message drop rates comparing two nodes and five nodes with the Default profile

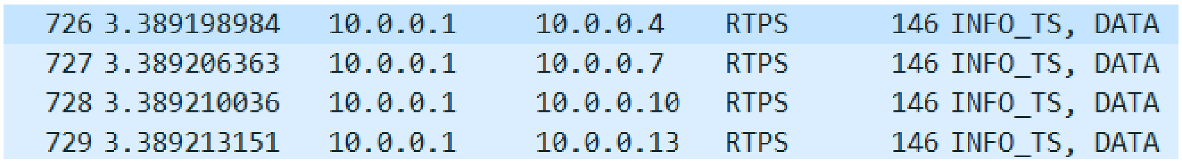


```
8881 40.233689330  10.0.0.10   10.0.0.1    RTPS     110 INFO_DST, ACKNACK
8882 40.238776621  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8885 40.238790309  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8887 40.238794895  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8890 40.238800970  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8892 40.238804709  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8895 40.238810723  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8897 40.238814678  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8900 40.238820521  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8902 40.238824213  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8905 40.238830035  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8907 40.238833742  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8910 40.238839544  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
8912 40.238843399  10.0.0.1    10.0.0.10   RTPS     162 INFO_DST, INFO_TS, DATA
8915 40.238849384  10.0.0.1    10.0.0.10   RTPS      94 HEARTBEAT
```

Figure 30. Wireshark capture of packets sent by Publisher node with five nodes in the network and using the Default profile

2. Latency

ROS 2 transmits each message sequentially to each individual node. This means that for a single message with four Subscribers, the same message is transmitted four times. This is depicted in the Wireshark capture shown in Figure 31. These sequential transmissions result in a significant increase in latency as more nodes are added to the network. Table 5 shows the latency experienced by each node in a five-node network that has one Publisher and four Subscribers as compared to the latency in a two-node network with only one Subscriber.



```
726 3.389198984   10.0.0.1    10.0.0.4    RTPS     146 INFO_TS, DATA
727 3.389206363   10.0.0.1    10.0.0.7    RTPS     146 INFO_TS, DATA
728 3.389210036   10.0.0.1    10.0.0.10   RTPS     146 INFO_TS, DATA
729 3.389213151   10.0.0.1    10.0.0.13   RTPS     146 INFO_TS, DATA
```

Figure 31. Wireshark capture of messages sent by Publisher

Table 5. Latency of Messages for each node in a one Subscriber network and a four Subscriber network (nanoseconds)

QoS profile	**Two-node network**	**Five-node network**			
	One Subscriber	**Node 1**	**Node 2**	**Node 3**	**Node 4**
Sensor	614 292	684 854	10 645 701	20 617 056	30 396 675
Default	639 480	712 332	10 706 034	20 725 019	30 793 318
Parameters	647 597	716 007	10 724 211	20 734 675	31 384 246

F. SUMMARY OF FINDINGS

The simulation results demonstrated that the integration of NS-3 as a simulation platform for ROS 2 is useful and an effective way to rapidly test network performance. The different QoS profiles affect the network performance in distinctive ways. The results from the various simulations demonstrated the trade-offs in network performance when using different QoS profiles. The Sensor profile delivers messages as quickly as possible, with a minimal impact on latency. It also outperforms the Default profile in terms of message drop rate in a network of high wireless loss. The Parameter profile has a large depth to cater to situations where the Subscriber node is repeatedly unable to reach the Publisher node. This results in a larger latency compared to the other profiles. In addition, the percentage of messages delivered is either 100% or 0% and would be not be suitable for all occasions.

There was also significant overhead when security settings were turned on. Using the Default profile, it incurred a 60% increase in latency, with the overhead likely to be much higher if performed using a slower processor. The overhead from having security turned on also meant a higher message drop rate across all QoS profiles.

Scaling up the number of nodes in the network to five nodes from two nodes resulted in varying consequences with the use of different QoS settings. Of significance is the increase in latency when the Default profile is used. It is likely that in a swarm network with 30–50 unmanned assets, a Reliable QoS policy cannot be used at all as the latency incurred would be too high.

V. CONCLUSION

A. SUMMARY

In this thesis, we proposed and validated a simulation architecture to study the network performance of ROS 2 using varying QoS profiles and security settings. The integration of ROS 2 with the NS-3 network simulator is unique to this thesis, and it was shown that the simulation architecture is effective for rapidly studying ROS 2 network performance.

Using the Default profile allows messages to be delivered in a reliable manner. Nonetheless, this comes at a cost of increased latency as compared to using the Sensor profile. The Sensor profile delivers messages as quickly as possible, with a minimal impact on latency. It also outperforms the Default profile in terms of message drop rate in a network of high packet losses.

Turning on the security features results in significant overhead in terms of latency, while producing a much higher message drop rate. In addition, scaling up the network to five nodes resulted in a significant increase in latency, with messages sent sequentially. On the other hand, it did result in an unexpectedly better performance in terms of message drop rate when the Default profile was used. As ROS 2 continues to evolve, the network performance of ROS 2 in a swarm network needs to be addressed, as it is not viable to scale up the number of nodes to a significant swarm due to the existing impact on latency.

B. FUTURE WORK

The work in this thesis contributes towards evaluating and configuring ROS 2 parameters for different unmanned system use cases, while providing a simulation framework on which tests can be run. Additional research would contribute towards the evaluations of the network performance of ROS 2.

1. Tuning of Additional QoS and Security Settings

There are many DDS settings that can affect ROS 2 network performance. Most of these settings are not available to the user for fine-tuning in the current ROS 2 Dashing

Diademata version. For example, DDS allows for specific security plugins to be turned on or off individually, but ROS 2 only allows security settings to be turned on or off entirely. As these setting controls are exposed in the future through ROS 2, additional work would have to be performed to evaluate their impact on network performance.

2. Use Case for a Swarm UxS Network

This thesis demonstrated the trade-offs in network performance incurred by different QoS and security settings. It would be useful to formulate a use case in a swarm UxS network using the appropriate QoS settings and validate the network performance through the simulation architecture.

3. Performance Testing through Actual Hardware

Simulations through the use of NS-3 allowed us to rapidly test and prototype networks of different settings. It would still be important to test out the network performance using actual hardware. A typical UxS would not have the processing power used in the simulations of this thesis. This will have an impact on the network performance with different QoS and security settings and will need to be validated.

APPENDIX A. SCRIPT TO GENERATE NAMESPACES

The following script is used to generate five different namespaces.

```
#!/usr/bin/env python3

# calculate IP values given index
def _ip_from_index(i):
  if i < 1 or i > pow(2,20)//3:
      raise Exception(“bad”)

  # wifi IP values are contiguous starting at 1
  j=(i-1)*3+1
  k=(i-1)*3+2
  l=(i-1)*3+3
  wifi_veth = “10.%d.%d.%d/9”%(j//65536, (j//256)%256, j%256)
  wifi_vethb = “10.%d.%d.%d/9”%(k//65536, (k//256)%256, k%256)
  wifi_tap = “10.%d.%d.%d/9”%(l//65536, (l//256)%256, l%256)

  # direct IP values are triplets step 8 starting at 1
  j=(i-1)*8+1
  k=(i-1)*8+2
  l=(i-1)*8+3
  direct_br = “10.%d.%d.%d/29”%(128+j//65536, (j//256)%256, j%256)
  direct_vethn = “10.%d.%d.%d/29”%(128+k//65536, (k//256)%256, k%256)
  direct_default = “10.%d.%d.%d”%(128+l//65536, (l//256)%256, l%256)

  return wifi_veth, wifi_vethb, wifi_tap, \
        direct_br, direct_vethn, direct_default

# create tap device: ip tuntap add tap1 mode tap; ip link set dev tap1 up

from argparse import ArgumentParser
import subprocess
import sys

def _run_cmd(cmd):
  print(“Command: %s”%cmd)
  subprocess.run(cmd.split(), stdout=subprocess.PIPE).stdout.decode(‘utf-8’)

def do_setup_nns(i):

  # network namespace
  _run_cmd(“ip netns add nns%d”%i)

  # enable loopback - helps Ctrl-C activate on first rather than second
  _run_cmd(“ip netns exec nns%d ip link set dev lo up”%i)

def do_setup_wifi(i):

  wifi_veth, wifi_vethb, wifi_tap, direct_br, direct_vethn, direct_default = \
                _ip_from_index(i)

  # create veth pair
  _run_cmd(“ip link add wifi_veth%d type veth peer name wifi_vethb%d”%(
                                                                    i,i))
  _run_cmd(“ip address add %s dev wifi_vethb%d”%(wifi_vethb,i))

  # associate wifi_veth with nns
  _run_cmd(“ip link set wifi_veth%d netns nns%d”%(i,i))

  # set wifi_veth IP at nns
  _run_cmd(“ip netns exec nns%d ip addr add %s “
         “ dev wifi_veth%d”%(i, wifi_veth, i))

  # create bridge
  _run_cmd(“ip link add name wifi_br%d type bridge”%i)

  # bring up bridge and veth pair
  _run_cmd(“ip link set wifi_br%d up”%i)
  _run_cmd(“ip link set wifi_vethb%d up”%i)
  _run_cmd(“ip netns exec nns%d ip link set wifi_veth%d up”%(i,i))

  # add wifi_vethb to bridge
  _run_cmd(“ip link set wifi_vethb%d master wifi_br%d”%(i,i))

  # create tap device
  _run_cmd(“ip tuntap add wifi_tap%d mode tap”%i)
  _run_cmd(“ip addr flush dev wifi_tap%d”%i) # clear IP
  _run_cmd(“ip address add %s dev wifi_tap%d”%(wifi_tap, i))
  _run_cmd(“ip link set wifi_tap%d up”%i)

  # add tap to bridge
  _run_cmd(“ip link set wifi_tap%d master wifi_br%d”%(i,i))

def do_setup_direct(i):

  wifi_veth, wifi_vethb, wifi_tap, direct_br, direct_vethn, direct_default = \
                _ip_from_index(i)

  # direct bridge
  _run_cmd(“ip link add direct_br%d type bridge”%i)

  # direct veth pair
  _run_cmd(“ip link add direct_veth%d type veth peer name direct_vethn%d”%(
                                                                      i,i))

  # veth to bridge
  _run_cmd(“ip link set direct_veth%d up”%i)
  _run_cmd(“ip link set direct_veth%d master direct_br%d”%(i,i))

  # bridge IP
  _run_cmd(“ip address add %s dev direct_br%d “%(direct_br,i))

  # veth to nns
  _run_cmd(“ip link set direct_vethn%d netns nns%d”%(i,i))
  _run_cmd(“ip netns exec nns%d ip addr add %s “
        “dev direct_vethn%d”%(i,direct_vethn,i))
  _run_cmd(“ip netns exec nns%d ip link set direct_vethn%d up”%(i,i))
  _run_cmd(“ip netns exec nns%d ip route add default via %s”%(i,direct_default))

  # bridge up
  _run_cmd(“ip link set direct_br%d up”%i)


def do_teardown_wifi(i):
  # wifi
  _run_cmd(“ip link set wifi_br%d down”%i)
  _run_cmd(“ip link set wifi_tap%d down”%i)
  _run_cmd(“ip link set wifi_vethb%d down”%i)
  _run_cmd(“ip link delete wifi_vethb%d”%i)
  _run_cmd(“ip link delete wifi_tap%d”%i)
  _run_cmd(“ip link delete wifi_br%d type bridge”%i)

def do_teardown_direct(i):
  # direct
  _run_cmd(“ip link set direct_br%d down”%i)
  _run_cmd(“ip link delete direct_br%d”%i)
  _run_cmd(“ip link set direct_veth%d down”%i)
  _run_cmd(“ip link delete direct_veth%d”%i)

def do_teardown_nns(i):
  # network namespace
  _run_cmd("ip netns del nns%d"%i)


if __name__=="__main__":
  DEFAULT_COUNT=5
  parser = ArgumentParser(prog='lxc_setup.py',
                        description="Manage containers used with ns-3.")
  parser.add_argument("command," type=str, help="The command to execute.",
                    choices=["setup," "teardown"])
  parser.add_argument("-c," "--count," type=int, default=DEFAULT_COUNT,
                    help="The number of network namespaces to set up for, "
                      "default %d."%DEFAULT_COUNT)
  parser.add_argument("-d," "--include_direct," action="store_true",
                    help="Suppress direct connection setup")
  args = parser.parse_args()

  print("Providing '%s' services for %d network namespaces..."%(
                                    args.command, args.count))

  if args.command == "setup":
    for i in range(1,args.count+1):
      do_setup_nns(i)
      do_setup_wifi(i)
      if args.include_direct:
        do_setup_direct(i)
  elif args.command == "teardown":
    for i in range(1,args.count+1):
      do_teardown_wifi(i)
      if args.include_direct:
        do_teardown_direct(i)
      do_teardown_nns(i)
  else:
    print("Invalid command: %s"%args.command)
    sys.exit(1)

  print("Done providing '%s' services for %d network namespaces."%(
                                    args.command, args.count))
```

APPENDIX B. ROS 2 PUBLISHER SCRIPT

The following script is used to generate the ROS 2 Publisher node. The script accepts arguments that allow you to specify the QoS policies to be used.

```
#include <chrono>
#include <cstdio>
#include <memory>
#include <string>
#include “rcl/time.h”
#include “rclcpp/rclcpp.hpp”
#include “rcutils/cmdline_parser.h”
#include “std_msgs/msg/string.hpp”

using namespace std::chrono_literals;

class Talker: public rclcpp::Node {
public: explicit Talker(const std::string & topic_name,
  const std::string & zlvarmsg,
    const std::int16_t & qos1,
      const std::size_t & qos2,
        const std::int16_t & qos3,
          const std::int16_t & qos4,
            const std::int16_t & zlcount): Node(“talker”) {
  msg_ = std::make_shared < std_msgs::msg::String > ();

  // Create a function for when messages are to be sent.
  auto publish_message = [ & ]() - > void {

    auto now = std::chrono::high_resolution_clock::now();
    auto now_ns = std::chrono::time_point_cast < std::chrono::nanoseconds > (now);
    auto value = now_ns.time_since_epoch();
  long duration = value.count();

  msg_ - > data = “Hi!” + zlvarmsg + “ : “ + std::to_string(count_++) + “ % “ +
    std::to_string(duration);

  RCLCPP_INFO(this - > get_logger(), “Publishing: ‘%s’,” msg_ - > data.c_str());

  pub_ - > publish(msg_);

  if (count_ == zlcount) {
    rclcpp::shutdown();
  }

};

auto var1 = RMW_QOS_POLICY_HISTORY_KEEP_LAST;;
auto var3 = RMW_QOS_POLICY_RELIABILITY_RELIABLE;
auto var4 = RMW_QOS_POLICY_DURABILITY_VOLATILE;

if (qos1 == 2) {
  var1 = RMW_QOS_POLICY_HISTORY_KEEP_ALL;
}
if (qos3 == 2) {
  var3 = RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT;
}
if (qos4 == 2) {
  var4 = RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL;
}

rmw_qos_profile_t custom_qos_profile = {
  var1,
    qos2,
    var3,
    var4,
    false
  };

  pub_ = this - > create_publisher < std_msgs::msg::String > (topic_name,
    custom_qos_profile);

  // Use a timer to schedule periodic message publishing.
  timer_ = this - > create_wall_timer(500 ms, publish_message);
}

private: size_t count_ = 1;
std::shared_ptr < std_msgs::msg::String > msg_;
rclcpp::Publisher < std_msgs::msg::String > ::SharedPtr pub_;
rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char * argv []) {
setvbuf(stdout, NULL, _IONBF, BUFSIZ);

if (rcutils_cli_option_exist(argv, argv + argc, “-h”)) {
  print_usage();
  return 0;
}

rclcpp::init(argc, argv);

// Parse the command line options.
auto topic = std::string(“chatter”);
```

```
char * cli_option = rcutils_cli_get_option(argv, argv + argc, “-t”);
if (nullptr != cli_option) {
  topic = std::string(cli_option);
}

auto msg_string = std::string(“ Hello World! “);
char * zl_string = rcutils_cli_get_option(argv, argv + argc, “-s”);
if (nullptr != zl_string) {
  msg_string = std::string(zl_string);
}

std::int16_t zqos1 = 1;
char * qos_choice1 = rcutils_cli_get_option(argv, argv + argc, “-q1”);
if (nullptr != qos_choice1) {
  zqos1 = std::atoi(qos_choice1);
}

std::size_t zqos2 = 10;
char * qos_choice2 = rcutils_cli_get_option(argv, argv + argc, “-q2”);
if (nullptr != qos_choice2) {
  zqos2 = std::atoi(qos_choice2);
}

std::int16_t zqos3 = 1;
char * qos_choice3 = rcutils_cli_get_option(argv, argv + argc, “-q3”);
if (nullptr != qos_choice3) {
  zqos3 = std::atoi(qos_choice3);
}

std::int16_t zqos4 = 1;
char * qos_choice4 = rcutils_cli_get_option(argv, argv + argc, “-q4”);
```

```
if (nullptr != qos_choice4) {
  zqos4 = std::atoi(qos_choice4);
}

std::int16_t zzlcount = 15;
char * cou = rcutils_cli_get_option(argv, argv + argc, “-c”);
if (nullptr != cou) {
  zzlcount = std::atoi(cou);
}

auto node = std::make_shared < Talker > (topic, msg_string, zqos1, zqos2, zqos3,
  zqos4, zzlcount);

rclcpp::spin(node);

rclcpp::shutdown();
return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. ROS 2 SUBSCRIBER SCRIPT

The following script is used to generate the ROS 2 Subscriber node. The script accepts arguments that allow you to specify the QoS policies to be used.

```
#include <cstdio>
#include <memory>
#include <string>
#include <iostream>
#include <fstream>
#include “rclcpp/rclcpp.hpp”
#include “rcutils/cmdline_parser.h”
#include “std_msgs/msg/string.hpp”

class ListenerBestEffort : public rclcpp::Node {
public:
  ListenerBestEffort(const std::int16_t& qos1, const std::size_t& qos2, const std::int16_t& qos3, const std::int16_t& qos4, const std::int16_t& distance, const std::int16_t& duration_time)
      : Node(“listener”)
  {

      auto callback =

         [&](const typename std_msgs::msg::String::SharedPtr msg) -> void {

         int totalpacket = 30; //number of initial packet to ignore
         long time;

         counter++;

         if (counter == 1) {
             auto initial = std::chrono::high_resolution_clock::now();
```

```
    }

    //get time
    auto now = std::chrono::high_resolution_clock::now();
    auto now_ns = std::chrono::time_point_cast<std::chrono::nanoseconds>(now);
    auto value = now_ns.time_since_epoch();
    long duration = value.count();

    if (counter > 1) {
        std::chrono::duration<double, std::micro> Elapsed = now - initial;
                auto zlabc=((duration_time)*1000000)-1000000;

        if (Elapsed.count() >= zlabc) {


            using namespace std;
   if(lostpacket<0) {                    rclcpp::shutdown();
        }
            ofstream myfile("result.txt," std::ios_base::app);
            if (myfile.is_open()) {
                myfile << qos1 << ",";
                myfile << qos2 << ",";
                myfile << qos3 << ",";
                myfile << qos4 << ",";
                myfile << distance << ",";
                myfile << lostpacket << ",";
                myfile << receivedpacket << ",";
                myfile << totallatency / receivedpacket << ",";
                myfile << packetnumber << ",";
                myfile << "\n";
                myfile.close();
            }
```

```
          else {
              cout << “Unable to open file”; }

          rclcpp::shutdown();
      }
  }

  std::string zlstr = msg->data; //copies the data in C++ format

  packetnumber = std::stoi(zlstr.substr(zlstr.find(“:”) + 2));
  time = std::stol(zlstr.substr(zlstr.find(“%”) + 2));



  if (packetnumber >= totalpacket) {
              if (lastpacket == 0) {lastpacket=packetnumber-1;}
      totallatency = totallatency + (duration - time);
      RCLCPP_INFO(this->get_logger(), “I heard: [%s],” msg->data.c_str());

      if (packetnumber - lastpacket != 1) {
          lostpacket = lostpacket + (packetnumber - lastpacket - 1);
      }
      lastpacket = packetnumber;

      RCLCPP_INFO(this->get_logger(), “[lost / received] : [%d/%d] packets,”
lostpacket, receivedpacket);



      receivedpacket++;
  }
  else {
      lostpacket = 0;
  }
```

```
      if(lostpacket<0) {                    rclcpp::shutdown();
          }

    };

    auto var1 = RMW_QOS_POLICY_HISTORY_KEEP_LAST;
    ;
    //auto var2=10;
    auto var3 = RMW_QOS_POLICY_RELIABILITY_RELIABLE;
    auto var4 = RMW_QOS_POLICY_DURABILITY_VOLATILE;

    if (qos1 == 2) {
      var1 = RMW_QOS_POLICY_HISTORY_KEEP_ALL;
    }
    if (qos3 == 2) {
      var3 = RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT;
    }
    if (qos4 == 2) {
      var4 = RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL;
    }

    rmw_qos_profile_t custom_qos_profile = {
      var1, qos2, var3, var4,
      false
    };

    sub_ = create_subscription<std_msgs::msg::String>(
      “chatter,” callback, custom_qos_profile);
  }
```

```
private:
  size_t counter = 0;
  long totallatency = 0;
  int receivedpacket = 1;
  int lostpacket = 0;
  int packetnumber = 1;
  int lastpacket = 0;
  std::chrono::time_point<std::chrono::system_clock>                    initial                    =
std::chrono::high_resolution_clock::now();
  rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub_;
};

int main(int argc, char* argv [])
{
  // Force flush of the stdout buffer.
  setvbuf(stdout, NULL, _IONBF, BUFSIZ);

  rclcpp::init(argc, argv);

  std::int16_t zqos1 = 1;
  char* qos_choice1 = rcutils_cli_get_option(argv, argv + argc, “-q1”);
  if (nullptr != qos_choice1) {
      zqos1 = std::atoi(qos_choice1);
  }

  std::size_t zqos2 = 10;
  char* qos_choice2 = rcutils_cli_get_option(argv, argv + argc, “-q2”);
  if (nullptr != qos_choice2) {
      zqos2 = std::atoi(qos_choice2);
  }

  std::int16_t zqos3 = 1;
  char* qos_choice3 = rcutils_cli_get_option(argv, argv + argc, “-q3”);
  if (nullptr != qos_choice3) {
     zqos3 = std::atoi(qos_choice3);
  }

  std::int16_t zqos4 = 1;
  char* qos_choice4 = rcutils_cli_get_option(argv, argv + argc, “-q4”);
  if (nullptr != qos_choice4) {
     zqos4 = std::atoi(qos_choice4);
  }

  std::int16_t zzlcount = 15;
  char* cou = rcutils_cli_get_option(argv, argv + argc, “-c”);
  if (nullptr != cou) {
     zzlcount = std::atoi(cou);
  }

  std::int16_t distance = 1;
  char* dis = rcutils_cli_get_option(argv, argv + argc, “-d”);
  if (nullptr != dis) {
     distance = std::atoi(dis);
  }

  std::int16_t time = 30;
  char* ztime = rcutils_cli_get_option(argv, argv + argc, “-t”);
  if (nullptr != ztime) {
     time = std::atoi(ztime);
  }
  auto Start = std::chrono::high_resolution_clock::now();
  auto node = std::make_shared<ListenerBestEffort>(zqos1, zqos2, zqos3, zqos4,
distance, time);
```

```
 while (1) {
    auto End = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> Elapsed = End - Start;

    rclcpp::spin_some(node);

    if (Elapsed.count() >= ((time) * 1000) + 10000) {
       break;
    }
 }
 return 0;
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. NS-3 SIMULATOR SCRIPT

The following script is used to generate the simulated Wi-Fi network within NS-3.

```
#include <iostream>
#include <fstream>

#include “ns3/core-module.h”
#include “ns3/network-module.h”
#include “ns3/mobility-module.h”
#include “ns3/wifi-module.h”
#include “ns3/tap-bridge-module.h”

// network devices, do not exceed COUNT in nns_setup.py
static const int COUNT=5;

using namespace ns3;

NS_LOG_COMPONENT_DEFINE (“LxcNs3Wifi”);

int
main (int argc, char *argv [])
{
double distance = 1.0;
int zltime = 60;

CommandLine cmd;
 cmd.AddValue (“distance,” “Distance apart to place nodes (in meters).”,
             distance);
 cmd.AddValue (“time,” “time to run experiment (in seconds).”,
             zltime);
cmd.Parse (argc, argv);
distance=distance/10;
std::cout << “Providing ns-3 Wifi network emulation for “ << COUNT << “ devices\
n”;
std::cout << “Distance: “ << distance << “ \n”;
std::cout << “Time: “ << zltime << “ \n”;

GlobalValue::Bind (“SimulatorImplementationType,” StringValue
(“ns3::RealtimeSimulatorImpl”));
GlobalValue::Bind (“ChecksumEnabled,” BooleanValue (true));

NodeContainer nodes;
nodes.Create (COUNT);
WifiHelper wifi;
wifi.SetStandard (WIFI_PHY_STANDARD_80211a);
wifi.SetRemoteStationManager (“ns3::ConstantRateWifiManager,” “DataMode,”
StringValue (“OfdmRate54Mbps”));

WifiMacHelper wifiMac;
wifiMac.SetType (“ns3::AdhocWifiMac”);

YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
wifiPhy.SetChannel (wifiChannel.Create ());

NetDeviceContainer devices = wifi.Install (wifiPhy, wifiMac, nodes);

MobilityHelper mobility;
Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator> ();
for (int i=0; i<COUNT; i++) {
    positionAlloc->Add (Vector (0.0, 0.0, 0.0));
positionAlloc->Add (Vector (distance, 0.0, 0.0));
positionAlloc->Add (Vector (-distance, 0.0, 0.0));
positionAlloc->Add (Vector (0.0, distance, 0.0));
positionAlloc->Add (Vector (0.0, -distance, 0.0));
}
mobility.SetPositionAllocator (positionAlloc);
mobility.SetMobilityModel (“ns3::ConstantPositionMobilityModel”);
mobility.Install (nodes);

TapBridgeHelper tapBridge;
tapBridge.SetAttribute (“Mode,” StringValue (“UseLocal”));
char buffer [10];
for (int i=0; i<COUNT; i++) {
  sprintf(buffer, “wifi_tap%d,” i+1);
  tapBridge.SetAttribute (“DeviceName,” StringValue(buffer));
  tapBridge.Install (nodes.Get(i), devices.Get(i));
}

 Simulator::Stop (Seconds (zltime));
 Simulator::Run ();
 Simulator::Destroy ();
}
```

LIST OF REFERENCES

[1] S. O'Donnell, "A short history of unmanned aerial vehicles," Media Centre Blog, Jun. 16, 2017. [Online]. Available: https://consortiq.com/media-centre/blog/short-history-unmanned-aerial-vehicles-uavs

[2] B. T. Clough, "UAV swarming? So what are those swarms, what are the implications, and how do we handle them?," Air Force Research Laboratory, Wright-Patterson Air Force Base, Ohio, AFRL-VA-WP-TP-2002-308, 2002

[3] S. Brimley, B. FitzGerald, K. Sayler, and P. W. Singer, "Game changers: Disruptive technology and U.S. Defense strategy," Center for a New American Security, Sep. 27, 2013. [Online]. Available: http://www.cnas.org/files/documents/publications/CNAS_Gamechangers_BrimleyFitzGeraldSayler_0.pdf

[4] Department of Defense, "Unmanned systems integrated roadmap: 2017–2042," Defense Daily. [Online]. Available: https://www.defensedaily.com/wp-content/uploads/post_attachment/206477.pdf

[5] ROS 2, "Why ROS 2?" Accessed Jul. 24, 2019. [Online]. Available: https://design.ros2.org/articles/why_ros2.html.

[6] N. DeMarinis, S. Tellex, V. Kemerlis, G. Konidaris, and R. Fonseca, "Scanning the internet for ROS: A view of security in robotics research," Jul. 23, 2018. [Online]. Available: arXiv:1808.03322 [cs]

[7] Y. Maruyama, S. Kato, T. Azumi, "Exploring the performance of ROS2," in *EMSOFT '16 Proc. of the 13th Int. Conf. on Embedded Softw.*, Article no. 5. Oct. 2016. [Online]. DOI: http://dx.doi.org/10.1145/2968478.2968502

[8] ROS, "ROS 2 and different DDS/RTPS vendors." Accessed Aug. 08, 2019. [Online]. Available: https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/

[9] C.S.V. Gutierrez, L.U. San Juan, I.Z. Ugarte, V.M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation for ROS 2.0 communications for real-time robotic applications," Sep. 7, 2019. [Online]. Available: arXiv:1809.02595v1 [cs.RO]

[10] Object Management Group, "DDS security version 1.1," Jul. 2018. [Online]. Available: https://www.omg.org/spec/DDS-SECURITY/1.1

[11] ROS, "Roadmap." Accessed Jul. 25, 2019 [Online]. Available: https://index.ros.org/doc/ros2/Roadmap

[12] G. Pardo, R. White, “Leveraging DDS security in ROS2,” presented at ROSCon, Madrid, Sep. 29, 2018. [Online]. Available: https://roscon.ros.org/2018 /presentations/ROSCon2018_DDS_Security_in_ROS2.pdf

[13] V. DiLuoffo, W. R. Michalson, B. Sunar, “Robot operating system 2: The need for a holistic security approach to robotic architectures,” *Int. J. Adv. Robot. Sys*., May 3, 2018. [Online]. DOI: 10.1177/1729881418770011.

[14] J. Kim, J.M. Smeraka, C. Cheung, S. Nepal, M. Grobler, “Security and performance considerations in ROS 2: A balancing act,” Sep. 24, 2018. [Online]. Available: arXiv:1809.09566v1 [cs.CR]

[15] Object Management Group, “The real-time publish-subscribe protocol (RTPS) DDS interoperability wire protocol specification Version 2.2,” Sep. 2014. [Online]. Available: https://www.omg.org/spec/DDSI-RTPS/2.2

[16] NSNAM, “About NSNAM.” Accessed Jul. 28, 2019. [Online]. Available: https://www.nsnam.org/about/

[17] NSNAM, “ns-3: src/tap-bridge/examples/tap-wifi-virtual-machine.cc Source File.” Accessed Jul. 28, 2019. [Online]. Available: https://www.nsnam.org/ doxygen/tap-wifi-virtual-machine_8cc_source.html

[18] Eprosima, “Security — Fast RTPS 1.9.0 documentation.” [Online]. Available: https://fast-rtps.docs.eprosima.com/en/latest/security.html

[19] ROS, “Sample policy.” [Online]. Available: https://github.com/ros2/sros2/blob/ master/sros2/test/policies/sample_policy.xml

[20] S. Sandoval, “Cyber security testing of the robot operating system in unmanned aerial systems,” M.S. thesis, Dept. of Elec. Eng., NPS, Monterey, CA, USA, 2018. [Online]. Available: http://hdl.handle.net/10945/60458

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California