How to cite this article: *Francisco Martín, Enrique Soriano, José M. Cañas, Quantitative analysis of security in distributed robotic frameworks, Robotics and Autonomous Systems, Volume 100, 2018, Pages 95-107, ISSN 0921-8890*

https://doi.org/10.1016/j.robot.2017.11.002

Quantitative analysis of security in distributed robotic frameworks

Francisco Martín, Enrique Soriano-Salvador, José M. Cañas
Universidad Rey Juan Carlos

Abstract

Robotic software frameworks simplify the development of robotic applications. The more powerful ones help to build such applications as a distributed collection of interoperating software nodes. The communications inside those robotic systems are amenable of being attacked and vulnerable to the security threats present on any networked system. With the robots increasingly entering in people's daily lives, like autonomous cars, drones, etc. security on them is a central issue gaining attention. This paper studies several well known communication middleware inside robotic frameworks running on robots with regular computers, and their support for cybersecurity. It analyzes their performance when transmitting regular robotic data of different sizes, with or without security features, and on several network settings. The experiments show that security, when available, does not significantly decrease the quality of the robotic data communication in terms of latency and packet loss rate.

1 Introduction

Robots have been traditionally used in industrial scenarios, but in the last years they have matured so much that their use is expanding to homes and streets, with people increasingly trusting them in some part of their life. For instance, robotic vacuum cleaners are now robust appliances in many homes. Commercial drones are now being used for event recording and their use in logistics is being explored. Autoparking capability and autonomous autopilots are included in some car models, driverless cars are already legal for research in several places and autonomous transportation systems are being tested with general public.

On their hardware, robots integrate sensors, actuators and computers. Their intelligence, algorithms and behaviors mainly rely on their software. Nowadays software is more than half of the cost of a robotic system. The robot software, as any other software, is amenable to suffer of security attacks. Many robots have external connectivity, for instance to let the user interact with them (like interacting with the vacuum cleaner through the smartphone). In addition, robots are complex systems and their software rely on robotic frameworks that simplify the development of applications. Most successful robotic frameworks are distributed. Such useful connectivity and distribution open the door to vulnerabilities and robot hacking.

The security attacks in robots have physical impact, not only information loss. The consequential damage of a hacked robot is directly commensurate with the amount of trust put into the system and is fully dependent on the capabilities of the robot. Simply hacking a robot to operate slightly out of a specified configuration mode can lead to everything from minor damage to death. Many attack examples have been reported, like the capture of a US Predator military drone (UAV) by the Iranian forces in 2011 [1]. Another relevant example is the remote hacking of a Jeep Cherokee car in 2014 [2].

Typically the robotics community has lived in a “happy naivety” [3]. For instance, the most popular robotic framework does not include any security mechanism. Researchers have discovered multiple common security flaws in mainstream robotic technologies from leading vendors, leaving them wide open to attack. Recently, cybersecurity in robotics is gaining attention [4, 5]. There is an increasing number of presentations about robotic systems in cybersecurity conferences (like DEF CON or RSA) and journals (like ACM Transactions on Cyber-Physical Systems (TCPS)), including many domains like rescue robotics [6], teleoperated robots [7, 8] or industrial robots [9].

Providing mechanisms to avoid undesired attacks and exploits in robot communication software is becoming increasingly required. Most promising solutions include security mechanisms inside the robot frameworks. Not all the security solutions are suitable for robotic environments. For instance, they have to comply with the general requirement of real-time operation, their computational cost can not be huge.

The goal of this paper is to quantitatively study the security solutions for communication explored inside distributed robotic frameworks. The performance of different communication middleware like Ice, Fast-RTPS (a DDS implementation) and the ROS transport system are compared when transmitting typical robotic data of different sizes (laser readings, images and point clouds). The experiments have been carried out in several network scenarios: local inside a single machine, an ethernet network and a WiFi network. Data transmission with and without security have been done in the bench tests, and the cost of the secure communications has been measured in terms of latency and data loss rate.

Section 2 presents the utility of the robotic frameworks and some illustrative examples. The security problems in robot software are introduced, classified and commented in section 3, including three communication choices. Section 4 describes the methodology followed, the experiments performed and the results. Finally some conclusions end the paper.

2 Robotic frameworks

In the last years, several robotic frameworks (SDKs) have appeared that simplify and speed up the development of robot applications [10, 11] . They favor the portability of applications between different robots, and ease the code reusability and integration. Modern robotic frameworks are based on software engineering criteria more than in cognitive issues. Their major achievements are: (i) the hardware abstraction, hiding the complexity of accessing heterogeneous hardware (sensors and actuators) under standard interfaces; (ii) the distribution capabilities, that allow to run complex systems spread over a network of computers; (iii) the multiplatform and multilanguage capabilities, that enable the developer to program and run the software in several computer types, robots and programming languages; and (iv) the existence of big communities around them, that share code, tools and algorithms.

There are several communication mechanisms and choices for providing the distribution capabilities. For instance, the Publish-Subscribe paradigm, the Remote Procedure Call (RPC) paradigm, distributed object-oriented models (like Common Object Request Broker Architecture, CORBA), the use of name servers, the use of central servers vs peer-to-peer communications, the use of specific well known communication middleware like Ice or DDS implementations vs the development of ad-hoc messaging software, etc.

Ice (Internet Communications Engine [12]) is an efficient, open source and object oriented RPC framework that provides SDKs for C++, Python, Java and other languages, and can run on various operating systems, including Linux, Windows, OS X and Android. It implements a proprietary communications protocol, called the Ice protocol, that can run over TCP, TLS, UDP, and WebSocket.

DDS (Data Distribution Service [13]) is a machine-to-machine standard that aims to enable scalable, real-time, dependable, high-performance and interoperable data exchanges using a Publish-Subscribe pattern. Both commercial and open-source software implementations of DDS are available. These include application programming interfaces (APIs) and libraries in Ada, C, C++, Java and other languages.

2.1 Advantages

Frameworks offer a more abstract access to sensors and actuators than the operating systems of simple robots do. The SDK Hardware Abstraction Layer (HAL) deals with low level details accessing to sensors and actuators, releasing the application programmer from that complexity. In addition, it provides high-level, easy-to-use interfaces.

Frameworks also provide a particular software architecture for robot applications, an specific way to organize the programs and deal with code complexity when the robot functionality increases. There are many options here: calling to library functions, reading shared variables, invoking object methods, sending messages via the network to servers, etc. Depending on the programming model the robot application can be considered an object collection, a set of modules talking through the network, an iterative process calling to functions, etc..

In addition, robotic frameworks usually include simple libraries, tools and common functionality blocks, such as robust techniques for perception or control, localization, safe local navigation, global navigation, social abilities, map building, etc. Libraries shorten the development time and reduce the programming effort needed to code a robotic application as long as the programmers can build it by reusing the common functionality included in the SDK, keeping themselves focused in the specific aspects of their application.

2.2 ROS

The Robot Operating System (ROS) [14, 15] is one of the biggest frameworks nowadays. It was founded by Willow Garage as an open source initiative and it is now maintained by Open Source Robotics Foundation. It has a growing user and developer community and its site hosts a great collection of hardware drivers, algorithms and other tools. ROS is a set of software libraries and tools that help to build robot applications (it includes from drivers to state-of-the-art algorithms, and with powerful developer tools simplifies the development of robotics projects). It is multiplatform and multilanguage.

The main idea behind ROS is an easy to use middleware that allows connecting several components (nodes) implementing the robotic behavior in a distributed fashion, over a network of computers using hybrid architecture. ROS is developed under hybrid architecture by message passing, mainly in Publish-Subscribe fashion (*topics*). Message passing of typed messages allows components to share information in a decoupled way. Therefore, the developer does not require to know which component sends a message, and vice versa, the developer does not know which component or components will receive the published messages.

Nodes send and receive messages on *topics*. A *topic* is a data transport system based on a Publish-Subscribe system. One or more nodes are able to publish data to a topic, and one or more nodes can read data on that topic. A topic is typed, the type of data published (the message) is always structured in the same way. A message is a compound data structure. It comprises a combination of primitive types (character strings, Booleans, integers, floating point real numbers, etc.) and messages (a message is a recursive structure). RPC mechanisms (like *services*) are available as well. Resources can be reached through a well defined naming policy and a ROS master.

2.3 Other frameworks

Another important example is ORCA [16], an opensource framework for developing component-based robotic systems. It provides the means for defining and developing the building-blocks which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks. It uses the Ice communication middleware from ZeroC and its explicit interface definition to exchange messages among the components. It was discontinued in 2009, but it was very influential.

Other relevant component-based framework is RoboComp [17] by Universidad de Extremadura. It is open source and also uses the Ice communication middleware as glue between its components. It includes some tools based on Domain Specific Languages to simplify the whole development cycle of the components. Most component code is automatically generated from simple and abstract descriptions over a component template. In addition, RoboComp includes a robot simulation tool that provides perfect integration with RoboComp and better control over experiments than current existing simulators.

Another framework that mainly uses Ice as communication middleware to connect its components is JdeRobot [18]. It includes a set of applications for teaching robotics and computer vision, and tools for visual programming of hierarchical Finite State Machines. It can also connect to ROS nodes using ROS messages.

Yarp [19] is a robotic middleware that has been used in humanoid robots for more than a decade and puts emphasis in interoperability with other environments like ROS nodes, web apps and others. It consists of a number of modules potentially on several different hosts, connected in a peer-to-peer topology. It includes an optional authentication mechanism which adds a key exchange to the initial handshaking on the TCP connection in order to authenticate any connection request.

There are many other open frameworks like OROCOS [20], Urbi [21] or NAOqi [22] (for Nao and Pepper robots).

3 Cybersecurity in Distributed Robotic Frameworks

As a distributed system, the basic security goals for a robotics system based on distributed components are:

- Confidentiality: The transmitted data must be available only for the authorized principals.
- Integrity: The transmitted data can not be modified by an unauthorized principal.
- Authentication: The transmitted data must be generated only by authorized principals.
- Availability: The components must be running and providing the corresponding service.

Normally, the threat model for communication protocols assumes an adversary that is able to:

- Act like any other node in the network, that is, it can send and receive messages.

- Access to the messages of other nodes. The adversary can read, modify and remove messages sent from other nodes. Some of the attacks require to take over the network infrastructure (e.g. routers or access points) or attack the physical layers (e.g. injecting noise in channels). Passive attacks only require observation. On the other hand, active attacks require interference.

The STRIDE [23, Chapter 3] threat model, defined by Microsoft, includes six different threat categories [24]. For a Publish-Subscribe model of communication, the threats are [25]:

- Spoofing: Unauthorized publish, message modification, unauthorized subscription and publish message rejection.
- Tampering: Replay attack, message modification and stored message tampering.
- Repudiation: Publish disclaimer and message receipt repudiation.
- Information disclosure: Confidentiality violation, metadata disclosure and stored message tampering.
- Denial of service: Publish message rejection and system overload.
- Elevation of privilege: Unauthorized publish, unauthorized subscription and confidentiality violation.

These are some examples in our context:

- Confidentiality violation: The images transmitted from the camera component to the controller component are intercepted by the attacker. Later, these images are used any illegitimate purpose (e.g. privacy violations).
- Message modification: The attacker changes the distance measures transmitted by a laser sensor to cause a crash of the robot and physically damage the robot or the environment.
- Unauthorized publish: The attacker forges false localization messages and injects them in the system to move the robot to another location.
- Overload: The attacker performs a *delay attack* and delays in the delivery of the localization data while the robot is moving. The controller component make decisions based on deprecated localization data.

- Replay attack: The attacker injects old petitions sent by a genuine component in order to repeat an actuator action and physically damage the robot or the environment.
- Publish message rejection and unauthorized publish: The adversary performs a relay attack, so she removes the original messages from the publisher and retransmits them in its behalf in order to replace a legitimate component (i.e. identity theft).
- Denial of Service: The attacker stops the component that controls the effectors (wheels, legs, etc.) in order to immobilize the robot.

Other threats related to host operating system vulnerabilities, application level vulnerabilities (e.g. command injection), malware or physical security of the robot are out of the scope.

In order to countermeasure the secure network protocols are based on cryptographic algorithms and tools like asymmetric/public-key encryption (e.g. RSA), symmetric encryption (e.g. AES), key agreement algorithms (e.g. DH), secure hashes (e.g. SHA2), message authentication codes (e.g. SHA2-HMAC), secure pseudo random generators, and so on.

Next, we briefly describe the security capabilities of the communication layers inside the robotic platforms selected for the present study.

3.1 Ice middleware

Ice permits to select between three transport layers: UDP, TCP and SSL[1]. SSL [26] is a widely used, standard protocol (e.g. HTTPS). The SSL transport layer is over the TCP layer. It provides an encrypted bi-directional connection. For the application, the SSL connection is similar to a TCP connection.

SSL is based on Public Key Infrastructure (PKI), that is, digital certificates. It follows an hybrid scheme that combines public key encryption and symmetric encryption. Basically, public key encryption is only used to start the communication and negotiate a shared session key (handshake). This session key is used to encrypt the channel by using symmetric encryption (which is faster than public key encryption) and is discarded when the session ends. In addition, it uses Message Authentication Codes (MACs) to provide message integrity and authentication. Client and server can support different sets of cryptographic algorithms (cipher suites). The cipher suite is negotiated in the SSL handshake.

[1]TLS (Transport Layer Security) is the successor of SSL. These acronyms are often used indistinctly.

Usually, client and server authenticate each other in the SSL handshake by interchanging some messages, encrypted with their RSA private keys. They follow the PKI scheme to obtain and validate the public keys, checking the chain of trust from the Certificate Authority root certificate (which must be locally installed). Alternatively, SSL/TLS can use Diffie-Hellman (DH), a key exchange protocol, to negotiate a shared secret and derivate the session key. There are three modes: anonymous DH, fixed DH and ephemeral DH (DHE). The recommended mode is DHE, which combines PKI and DH to provide authentication and perfect forward secrecy (PFS) [27].

SSL is often considered too heavy for constrained devices and embedded systems [28], because it depends on costly public key cryptography. Nevertheless, there are different lightweight implementations targeted for embedded systems [29, 30, 31].

IceSSL [32] is the Ice plugin that implements the SSL transport layer, available for C++, Java and .NET. It only requires to create the PKI infrastructure (key pairs and certificates for the certification authority, publishers, subscribers and infrastructure nodes), configure the plugin and update the properties of the application. In general, it is not necessary to modify the source code of the application. Therefore, the burden of adding security to the application is relatively low. The C++ IceSSL implementation uses the OpenSSL library [33]. Thus, it can use all the supported cipher suites.

In this study we evaluate IceSSL 3.5. All the components of the application (publisher, subscriber, IceBox service and the IceStorm service) use SSL end-points. They use the default cipher suite for this version, `DHE-RSA-AES256 -GCM-SHA384`: Diffie-Hellman is used to negotiate the shared secret and RSA is used for authentication, AES (with 256-bit keys) is used for symmetric encryption, GCM (Galois Counter Mode) is the mode of operation for AES and SHA2 is used to create secure hashes (384-bit digests). Note that GCM is an authenticated mode of operation, that is, the ciphertext is not *malleable*: any modification of the cipher text is detected and the decryption fails in the receiver.

3.2 The ROS transport system

ROS includes its own communication middleware: TCPROS/UDPROS. It does not provide any security features [34]. The only secure approach is to isolate the ROS nodes in private networks. If the nodes need to be connected to a public network, perimeter security is the only solution: the network has to be protected by configuring firewalls and data diodes, and the routers must use network address translation (NAT). In addition, virtual private networks (VPN) are used to communicate robots in different private networks. Unfortunately, the security issue is usually overlooked in standard ROS applications.

Some research efforts have been made in order to add security features to ROS [35]. For instance, in [36] the use of web tokens for achieving secure authentication for remote, non-native clients in ROS is discussed. Only authentication of messages is addressed.

Dieber et al. [37] proposed a security architecture intended for use on top of ROS on the application level for industrial scenarios. They use a dedicated authorization server to ensure that only valid nodes are part of the application. Cryptographic methods ensure data confidentiality and integrity.

SROS is a proposed addition to the ROS API. It provides an ecosystem to support modern cryptography and security measures [38, 39]. SROS is intended to secure ROS across three main fronts: (a) Transport Encryption (for instance verifying the identity of nodes, including TLS support and PKI certificates) ; (b) Access Control (for instance restricting the node's scope of access within the ROS graph to only what is necessary) ; and (c) Process Profiles (for instance hardening node processes on using Linux Security Modules in kernel)

A huge increase in security support and communication performance inside ROS is expected with the jump from ROS1 to ROS2 and the use on it of DDS communication middleware [40, 41].

3.3 DDS Implementation: Fast-RTPS

The DDS Security specification is defined by the Object Management Group (OMG) [42].

In this study, we evaluate the Fast-RTPS implementation [43]. By default, the communication is not secure. Nevertheless, it can be configured to provide authentication and encryption.

Authentication is performed to discover remote nodes. The authentication plugin (`Auth:PKI-DH`) is based on PKI (X.509 certificates) and the ECDSA signature algorithm. The authentication process finishes with the creation of a shared secret that can be used later for encryption and message authentication. The negotiation of the shared secret is based on a variant of Diffie-Hellman based on elliptic curve cryptography (ECDH). Each node must have the X.509 certificate of the Certification Authority, its own X.509 certificate (signed by the authority) and its own private key.

Encryption is provided by the plugin `Crypto:AES-GCM-GMAC`. It uses symmetric encryption, AES, in GCM mode. Each published message is encrypted with a shared key. All subscribers know this shared key used for encryption. In addition, each subscriber shares another key with the publisher. This key is used to generate MACs (Message Authentication Codes). Every published message includes multiple MACs, one for each receiver. When a subscriber receives a message, it decrypts the message and checks the corresponding MAC. This way, a subscriber is not able to publish messages, even knowing the shared key used for encryption: it does not know the keys to generate the MACs for the other subscribers.

There are three levels of encryption:

- Encrypt the whole RTPS message.
- Only encrypt RTPS submessages of an entity.
- Only encrypt the payload of a particular *writer*.

In this study we opted for the first option.

4 Experiments

In this section we will describe the experiments performed to analyze the impact of security on distributed robotics frameworks.

4.1 Experimental design

Robotics applications are typically distributed into several software components and communication middlewares allow their interconnection. The goal of the experiments is to measure the impact of the security mechanisms present in the middlewares described in previous sections.

The working hypothesis is that the security mechanisms implemented in the communication middlewares included in this study produce an acceptable overload for typical distributed robotic applications running on common robotic hardware.

The independent variable of the experiments is the activation of the security mechanisms in the underlaying communication middleware.

The two dependent variables to be measured are:

- Message loss rate. The messages with robotic data can be lost, some of them do not arrive on time to the subscriber or are discarded. The number of lost messages is accounted for and stored as a measurement of communication quality.

- Latency. Latency is measured as the time to complete these steps: (a) the serialization of the data in the publisher; (b) the transmission of the message through the communication software; (c) the reception of the message; and (d) the deserialization of the message on the subscriber.

The experiments consist of two main nodes, a producer and a consumer, sending robot sensor data over different networks using different communication middlewares and settings. These nodes use the Publish-Subscribe mechanisms of the underlaying middleware to communicate. Both the producer and consumer are ROS nodes (similar to UNIX processes) using the different communication middlewares we want to compare. The standard ROS serialization/deserialization mechanisms are used in all the experiments (natively for ROS and using ROS libraries for the other communication solutions). The communication configurations that have been tested in the experiments are:

- **ROS**. This is the baseline framework: we use the result of this configuration as a reference to evaluate the results of the others. Although TCP and UDP are supported in ROS, we will use TCP. This is the default option and it is not usually changed.
- **Ice**. We use the common IceStorm configuration, which is based on TCP.
- **IceSec**. We also use the common IceStorm configuration, but using a SSL connection, which only supports TCP.
- **Fast-RTPS Best-effort (FRTPS)**. This is the baseline configuration for Fast-RTPS. It uses multicast addresses over UDP, which is the only supported protocol.
- **Fast-RTPS Reliable (FRTPS_rel)**. This is a variation of the baseline configuration, but trying to warranty the message delivery.
- **Fast-RTPS Best-effort with authentication (FRTPSAuth)**. The subscriber and the producers are authenticated.
- **Fast-RTPS Reliable with authentication (FRTPSAuth_rel)**. Similar to previous variation, adding reliability.
- **Fast-RTPS Best-effort encrypted (FRTPSCrypt)**. The entire message is encrypted.

- **Fast-RTPS Reliable encrypted (FRTPSCrypt_rel)**. Besides of the encryption, the delivery is reliable.

We propose a set of scenarios that we consider complete and representative, including:

- **Several networks**. Even when a robotic framework is distributed, most of the application components usually run inside the robot. The main reason to distribute components in other hosts is for debugging, or for computations which require more resources than those available onboard the main robot processor. In addition, several computers may be inside the robot system, in the same wired or wireless network. To cover the most illustrative scenarios, the experiments are performed on three different networks: *localhost*, an *ethernet* connection, and a *WiFi* connection.

- **Several data types**. Mobile robots are usually equipped with sensors to perceive its environment. Most common sensors nowadays are lasers, cameras and 3D cameras. For this reason we carry out the experiments using the information provided by these sensors: LaserScan (9Kb per reading), Image (900Kb) and PointCloud (9Mb). The last one is very challenging for the communication solutions which are not usually designed to manage such large amount of data in one message.

4.2 Execution

The experiment has been executed in the real robot of Figure 1. This system reproduces the real scenario used during regular robot operation. This configuration, with standard hardware, is common in other robots, such as those shown in Figure 2.

We aim to perform high precision measurements. In order to do so, we use a global clock to measure the latency. Therefore, all the nodes (publisher, subscriber and the additional services required by the middleware) run in the same machine.

We have taken extra precautions to measure the latency of the messages with high accuracy. We read the time stamp counter register (TSC) of the CPUs directly from the user space process to avoid OS interference (e.g. system calls, etc.). Every message has an ID. The publisher reads the TSC before serializing the data and publishing it. Later, the ID of the message, together with the TSC value, is stored in memory. When the subscriber receives and deserializes a message, it reads the TSC and stores it in memory.

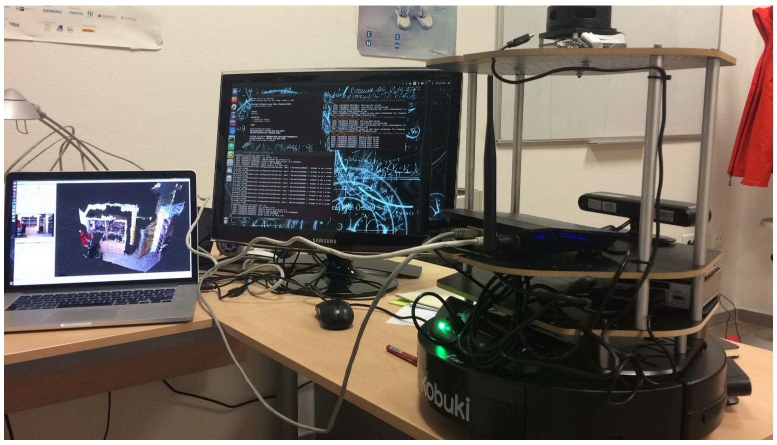

Figure 1: Robot used in the experiments.

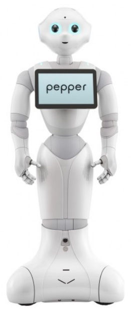

Figure 2: Another robots with standard computing hardware.

When all rounds are finalized, the publisher and the subscriber print the TSC values for each message ID. This way, we avoid interferences caused by I/O while the rounds are executed. When the experiment finishes, we calculate the CPU cycles for each message ID and transform it to microseconds.

Measuring with the TSC is not straightforward [44]. First of all, we must pay attention to the guarantees provided by the CPU. The CPU is an Intel Core i5-5250U CPU at 1.60GHz with four cores. Each core has its own TSC register, which counts the number of cycles since reset. This Intel CPU keeps the TSC of the four cores synchronized. Thus, the code that is being measured can be moved to another core by the operating system scheduler and the measurement is not affected at all. In addition, in some CPUs, de TSC rate may not be invariant (e.g. changes the frequency for power saving). That is not the case, the CPU used for the experiment has the `nonstop_tsc` and `constant_tsc` flags set[2]. Thus, it guarantees *invariant*

[2]This can be checked by reading `/proc/cpuinfo` in a GNU/Linux system.

TSC: The CPU increments the TSC at a constant rate even when the CPU frequency is changed. Moreover, the TSC rate is constant in all ACPI P-, C-, and T-states [44].

We read the TSC following the guidelines published by Intel to benchmark code execution [45]. In order to measure just the time to complete the actions described before, we embedded the required assembly code to read the TSC from user space in the publisher and subscriber C++ source code (i.e. we measure no extra function/method/system calls at all). Moreover, we take into account that modern CPUs implement *out-of-order executions* dynamic execution. Thus, we serialize the previous instructions with a barrier instruction before reading the TSC. The barrier instruction forces the CPU to finish the execution of every pending instruction before reading the TSC.

In order to reproduce the network scenarios described before (localhost, Gigabit ethernet and WiFi networks), we take advantage from Linux NetEm [46]. This module allows the simulation of network delays. Moreover, we are able to simulate bandwidth and packet loss.

We measured the real ethernet and wireless networks of our laboratory to characterize the scenarios[3]. The parameters for each one are:

- **Localhost**. Loopback device with no limits.
- **Gigabit Ethernet**. Delay: normal distribution, mean=0.3ms, sd=0.36ms. Bandwidth limit: 1000 Mbit/s. Packer loss rate: 0%.
- **WiFi**. Delay: normal distribution, mean=13ms, sd=22ms. Bandwidth limit: 54 Mbit/s. Packer loss rate: 1%.

For all the experiments, the data to be transmitted are the same: a collection of laser readings, images and 3D point clouds stored in log files of the ROS framework, *ROSbag* files. *ROSbag* is a critical resource for our experiments. Instead of reading directly from sensors, the producer reads the data from a file called bag file. This file is built from an offline execution of the sensor driver, storing there the sensor data and their corresponding timestamps. This let us to reproduce this file ensuring exactly the same dataset for all the experiments. It also avoids any communication between the driver node and the producer, which could affect the measurements. Each bag file stores 1 minute of real data acquisition. The `LaserScan` bag file contains 2398 samples, the `Image` bag file contains 1910 samples and the `PointCloud` bag file contains 1083 samples.

[3]There is only one subnet in the ethernet and WiFi scenarios.

For the execution of each experiment, the configuration[4] includes setting the communication mechanism for both the publisher and the subscriber (one of the nine aforementioned), setting the network conditions and the type of data to be sent. The producer reads all the bag file, serializes the messages and publishes the data at the same frequency as they were originally produced. The consumer receives the messages and deserializes them.

To compute the final latency values and loss percentage, the first rounds (10%) of each execution are discarded in order to avoid warming effects (cache, buffers, queues, etc.). Both scores reflect the quality of the selected communication mechanism under different conditions.

4.2.1 Discussion

As we stated before, we need a global clock to measure the path from the sender (publisher) to the receiver (subscriber). Thus, these nodes have to run in the same machine and the network has to be emulated.

Alternatively, this path could be measured in a real scenario (i.e. the nodes running on different machines connected through a real network). In this case, there are two options: (i) to synchronize the clocks of the machines or (ii) to measure round-trips by using the clock of the sender and estimate the time for the path as the half of the round-trip time. Note that the second option does not fit the publisher/subscriber model well. In addition, both options are far less precise than the followed approach.

In an experimental study, there is a trade-off between realism and control. In this case, we sacrifice the realism of using a real network in order to have more control in the measurement. We prefer to measure the latency in a very accurately way on realistic emulated networks rather than loosely measuring it on real networks. With Linux NetEm, we are able to create a synthetic scenario close the real behavior of the network. The network behavior for each message is just a random sample extracted from the modeled network. Therefore, the results of the experiments represent the expected case for our real networks (and the number of executions).

4.3 Analysis

4.3.1 Analysis of loss rate

In the following experiments we have measured the loss rate, from 0% to 100%, when transmitting different robotic data using several communication

[4]The source code of the developed bench test and the used log *ROSbag* files are publicly available at `https://gitlab.aulas.gsyc.urjc.es/fmartin/JdeRobotROS`,

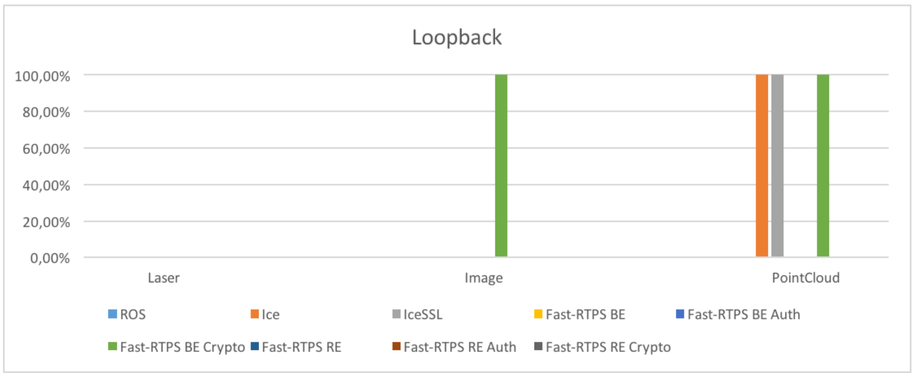

Figure 3: Loss rate using the loopback interface.

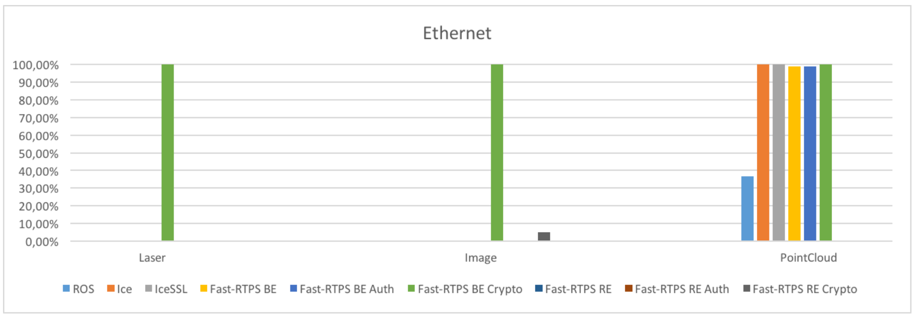

Figure 4: Loss rate using Gigabit Ethernet.

solutions.

First, Figure 3 shows the loss rate using the loopback interface. This scenario is very illustrative, as in many settings all the components of the robotic application are executed on a single computer on board the robot. On those settings the WiFi or the ethernet are used mainly for debugging.

Second, Figure 4 shows the results using a Gigabit Ethernet network. And third, Figure 5 shows the loss rate when using a wireless network.

4.3.2 Analysis of latency

The presented graphs are Tukey's boxplots [47]. The bottom and top of the boxes are the first and third quartiles. The line in the box is the median and the point is the mean. The upper whisker extends from the hinge to the highest value that is within $1.5 * IQR$ of the hinge (IQR, interquartile range, is the distance between the first and the third quartile). The lower whisker extends from the hinge to the lowest value within $1.5 * IQR$ of the hinge. Data beyond the end of the whiskers are outliers and plotted as points.

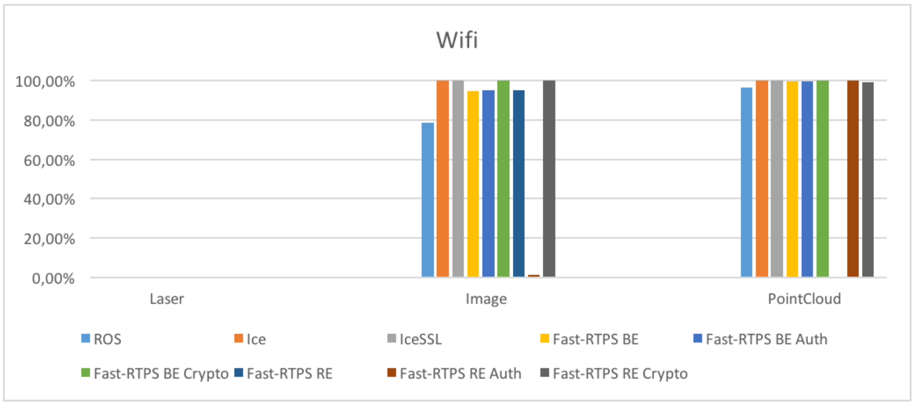

Figure 5: Loss rate using a Wireless network.

In the following figures, the Y axis plots the latency in a *log10* scale in most cases. Certain frequencies, like 10Hz (100000us) or 33Hz (33000us) have been marked as references for real-time operation.

Figure 6 shows the latency when the publisher and the subscriber use the loopback network interface and different communication solutions to send robotic data. The combinations with high loss rate are not shown.

Figure 7 shows the latency when the publisher and the subscriber use the ethernet network interface and different communication solutions to send robotic data.

Figure 8 shows the latency when the publisher and the subscriber use the WiFi network interface and different communication solutions to send robotic data.

		Laser	
	Local	Ethernet	Wifi
ROS	$41609 \pm 13231\mu s$	$33905 \pm 12925\mu s$	$60235 \pm 34614\mu s$
ICE	$124 \pm 25\mu s$	$851 \pm 444\mu s$	N/A
ICE SSL	$186 \pm 845\mu s$	$968 \pm 1245\mu s$	$48674 \pm 32206\mu s$
Fast-RTPS Best Effort	$89 \pm 16\mu s$	$418 \pm 296\mu s$	$17343 \pm 17107\mu s$
Fast-RTPS Best Effort Auth	$89 \pm 9\mu s$	$438 \pm 315\mu s$	$17252 \pm 16160\mu s$
Fast-RTPS Best Effort Crypto	$576 \pm 18\mu s$	$450 \pm 304\mu s$	$17590 \pm 17013\mu s$
Fast-RTPS Reliable	$96 \pm 9\mu s$	$449 \pm 302\mu s$	$20914 \pm 20576\mu s$
Fast-RTPS Reliable Auth	$96 \pm 7\mu s$	$441 \pm 299\mu s$	$20453 \pm 19601\mu s$
Fast-RTPS Reliable Crypto	$617 \pm 49\mu s$	$965 \pm 322\mu s$	$21839 \pm 19631\mu s$

Table 1: Latency results for Laser Data

4.4 Interpretation

Next, we will interpret the results of the experiments presented in last sections. For clarity, we will follow the same structure for this interpretation.

	Image		
	Local	Ethernet	Wifi
ROS	$31809 \pm 22330\mu s$	$41357 \pm 20529\mu s$	$688181 \pm 115580\mu s$
ICE	$5013 \pm 415\mu s$	$27104 \pm 6914\mu s$	N/A
ICE SSL	$6224 \pm 1011\mu s$	$27875 \pm 7260\mu s$	N/A
Fast-RTPS Best Effort	$1834 \pm 121\mu s$	$8533 \pm 452\mu s$	$4610726 \pm 2605580\mu s$
Fast-RTPS Best Effort Auth	$1845 \pm 185\mu s$	$8543 \pm 442\mu s$	$4558665 \pm 2501261\mu s$
Fast-RTPS Best Effort Crypto	$2341 \pm 663\mu s$	N/A	N/A
Fast-RTPS Reliable	$2541 \pm 152\mu s$	$8679 \pm 369\mu s$	$4536934 \pm 2446427\mu s$
Fast-RTPS Reliable Auth	$2317 \pm 192\mu s$	$8528 \pm 374\mu s$	$4526905 \pm 2506736\mu s$
Fast-RTPS Reliable Crypto	$28249 \pm 771\mu s$	$28458 \pm 1107\mu s$	N/A

Table 2: Latency results for Image Data

	Point Cloud		
	Local	Ethernet	Wifi
ROS	$67496 \pm 13508\mu s$	$197445 \pm 28664\mu s$	$2116548 \pm 222370\mu s$
ICE	N/A	N/A	N/A
ICE SSL	N/A	N/A	N/A
Fast-RTPS Best Effort	$54185 \pm 2741\mu s$	$329298 \pm 144492\mu s$	$6760720 \pm 1394592\mu s$
Fast-RTPS Best Effort Auth	$54259 \pm 2823\mu s$	$324007 \pm 133125\mu s$	$5184121 \pm 2342175\mu s$
Fast-RTPS Best Effort Crypto	N/A	N/A	N/A
Fast-RTPS Reliable	$58185 \pm 14807\mu s$	$319437 \pm 172991\mu s$	$10435526 \pm 3784635\mu s$
Fast-RTPS Reliable Auth	$58568 \pm 17114\mu s$	$323849 \pm 172553\mu s$	N/A
Fast-RTPS Reliable Crypto	$439160 \pm 59368\mu s$	$433916 \pm 45754\mu s$	$6083977 \pm 3064795\mu s$

Table 3: Latency results for Point Cloud Data

	Laser			Image			Point Cloud		
	Lo	Eth	Wifi	Lo	Eth	Wifi	Lo	Eth	Wifi
ROS	0.13%	0,04%	0,04%	0,00%	0,00%	78,42%	0,00%	36,78%	96,46%
ICE	0,00%	0,00%	100%	0,00%	0,00%	100%	100%	100%	100%
ICE SSL	0,00%	0,00%	0,00%	0,00%	0,00%	100%	100%	100%	100%
FR BE	0,00%	0,00%	0,00%	0,00%	0,00%	94,89%	0,00%	98,68%	99,62%
FR BE Auth	0,00%	0,00%	0,17%	0,00%	0,00%	94,95%	0,09%	98,68%	99,62%
FR BE Crypto	0,00%	100%	0,00%	0,00%	100%	100%	100%	100%	100%
FR RE	0,00%	0,00%	0,00%	0,00%	0,00%	95,23%	0,00%	0,00%	0,00%
FR RE Auth	0,00%	0,00%	0,00%	0,00%	0,00%	1,44%	0,00%	0,00%	100%
FR RE Crypto	0,00%	0,00%	0,00%	0,00%	5,11%	100%	0,00%	0,00%	99,34%

Table 4: Loss rate.

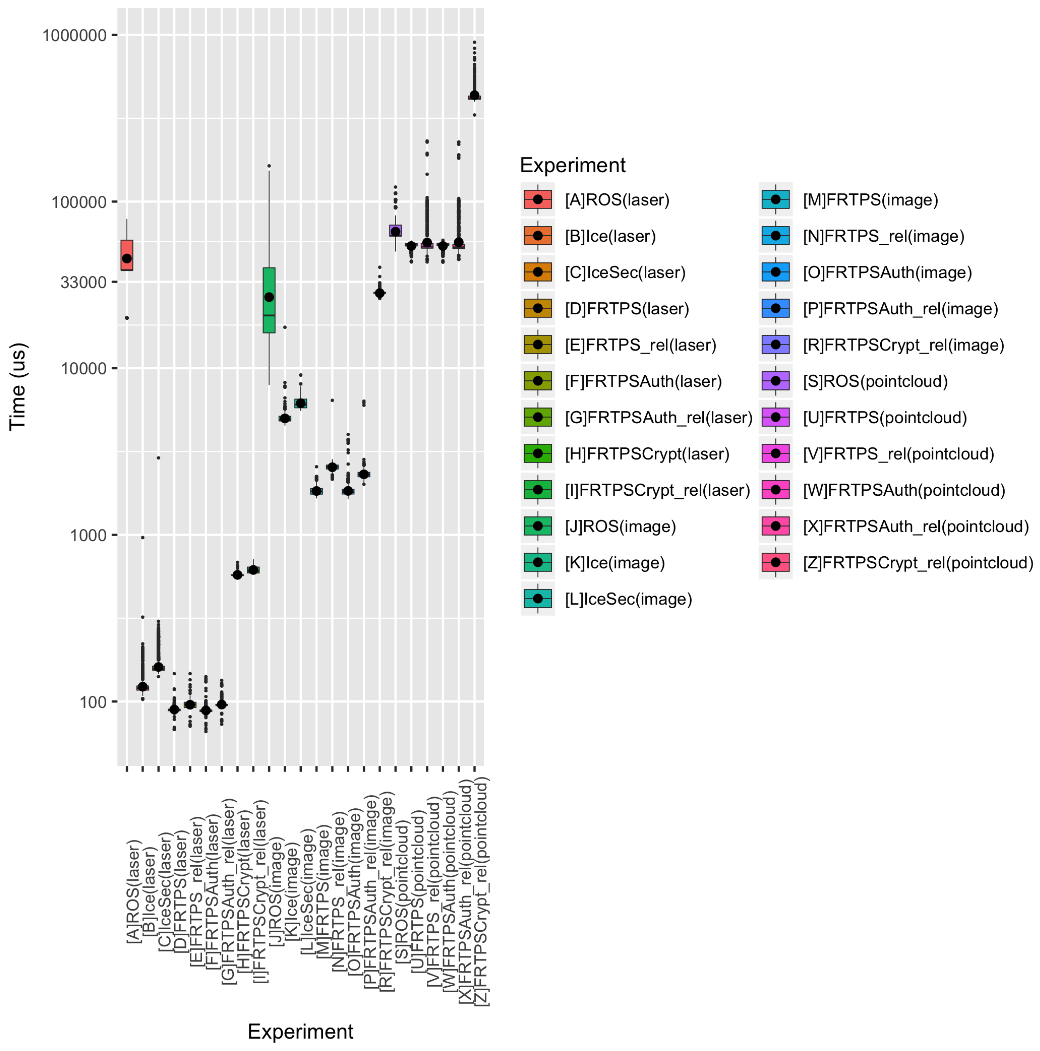

Figure 6: Latency using the loopback interface, with different data and communication solutions

4.4.1 Interpretation of results of loss rate

The interpretation of the results of this experiment (Figure 3) using the loopback interface shows how IceStorm is not able to send PointCloud messages, since it does not allow sending so large data. Most of the communication solutions efficiently transmits laser readings and images, as the loss rate is zero. But for the bigger readings (i.e., PointClouds) Ice variants stall and lose all messages. This figure also shows that the Fast-RTPS Best Effort variant with encrypted messages is only capable of sending data from the

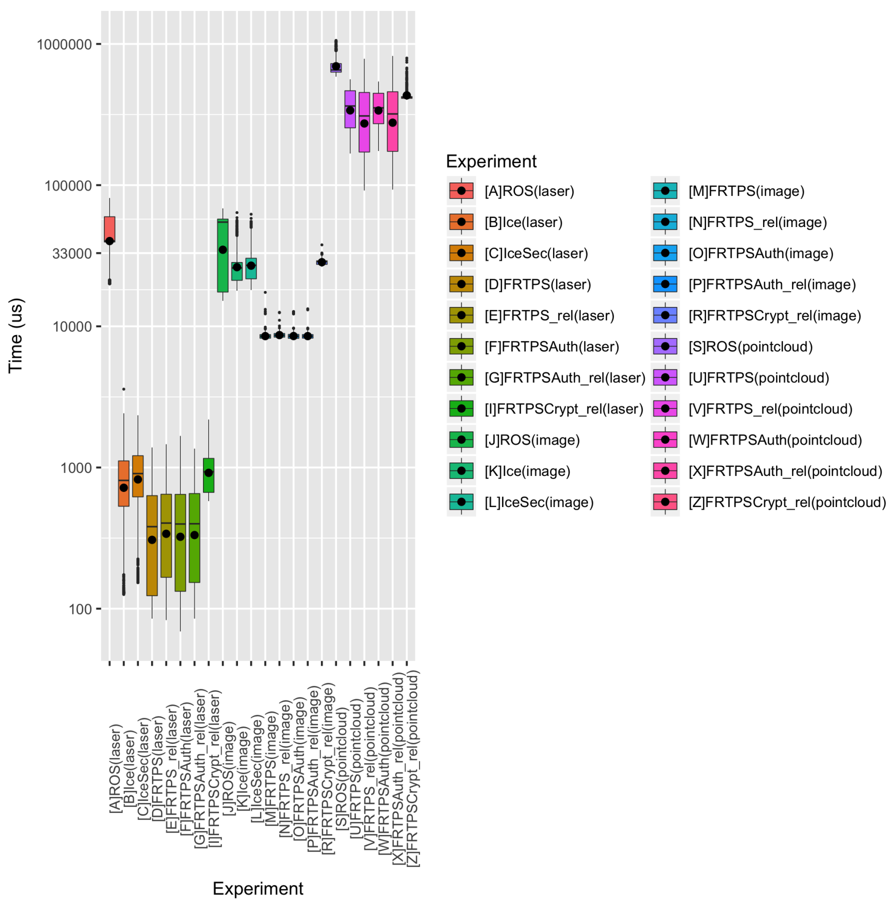

Figure 7: Latency using the ethernet interface, with different data and communication solutions.

laser.

Regarding to the results using a Gigabit Ethernet network (Figure 4), Fast-RTPS Best Effort with the encrypted messages again has problems for the communication of the data, this time even with laser. Regarding larger data, it is interesting to see how the loss rate when transmitting PointCloud with ROS remains less than 40% while most of other communication solutions lose all of PointCloud messages. In addition, the Reliable variant of fast-RTPS is the only one that succeeds in sending all data, either encrypted, authenticated or clear.

In Figure 5, which corresponds to the loss rate when using a wireless network, only some reliable variant of Fast-RTPS is able to send images and pointcloud with a loss rate lower than ROS. In this case, the experiment verifies that all communication solutions, with their variants, are capable of

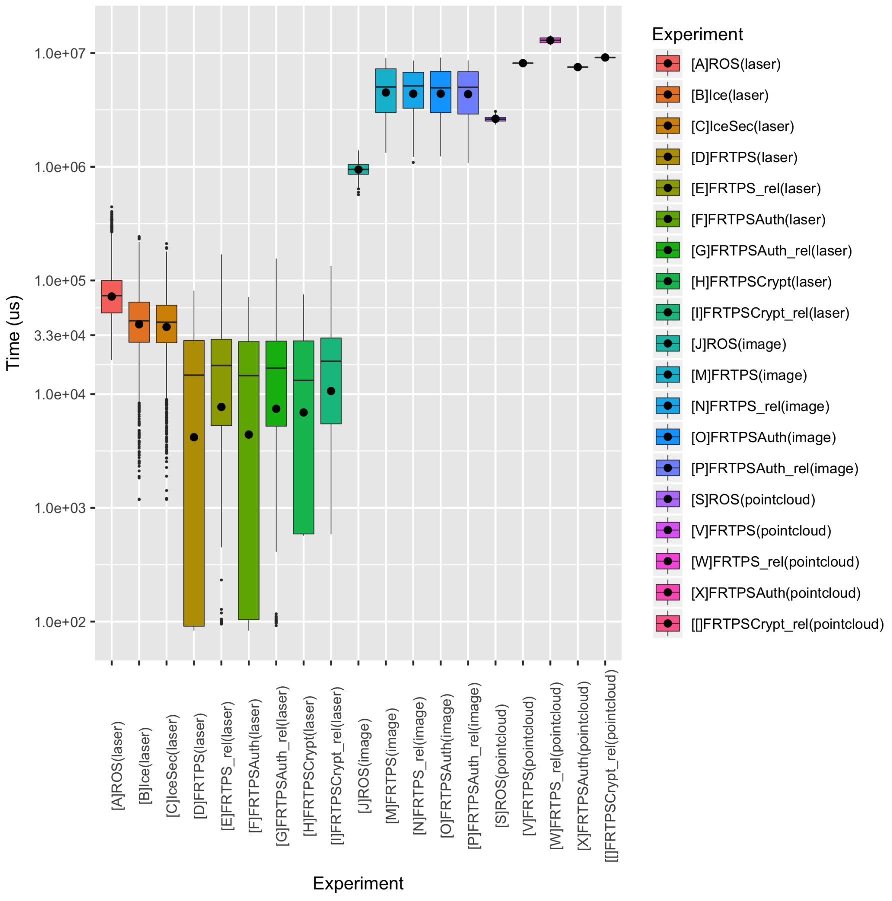

Figure 8: Latency using the WiFi interface, with different data and communication solutions.

sending small data, but not larger sensor readings.

As expected, loopback is the best network scenario, then ethernet and then WiFi. The wireless scenario has the smaller throughput and is more prone to transmission errors. All tested communication solutions are able to efficiently transmit laser sensor readings, but problems appear (non zero loss rates) with bigger sensor readings as the network conditions become hard. As the network throughput decreases, also does the size of the data that the communication solutions are able to efficiently transmit without losses.

In addition, in loss rate terms, there are no big difference between the secure and no secure communication solutions.

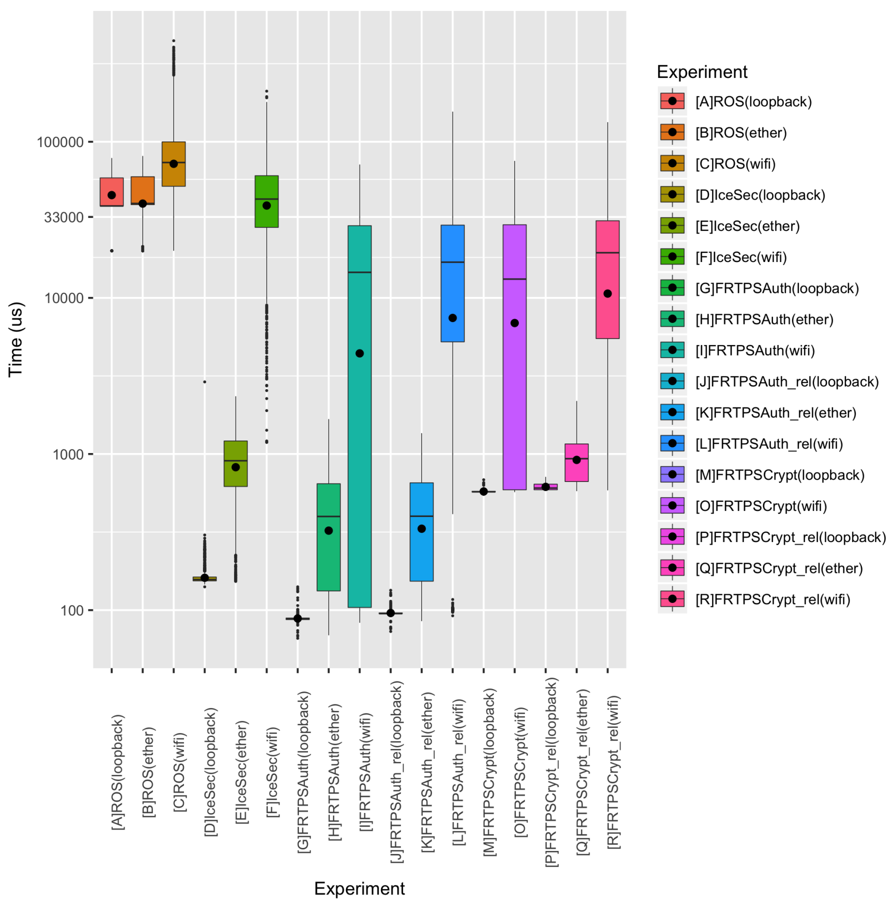

Figure 9: Latency transmitting laser readings with different communication solutions and networks.

4.4.2 Interpretation of result of latency

When transmitting laser readings and images (Figure 6) in the loopback interface, the ROS based transport system performs in real time (close to 33ms), but Ice and Fast-RTPS solutions perform better with significantly smaller latency. Fast-RTPS performs slightly better than Ice. For heavy sensor readings like PointClouds, the ROS transport system stays stable keeping real-time operation (close to 60ms), the Fast-RTPS latency is similar but Ice solutions do not work well.

Another relevant conclusion is that the increase in latency due to inclusion of reliability in the communications is very small. For instance, comparing the latencies for Ice and IceSec when transmitting laser readings or images, they are quite similar. The same when comparing Fast-RTPS and Fast-RTPS

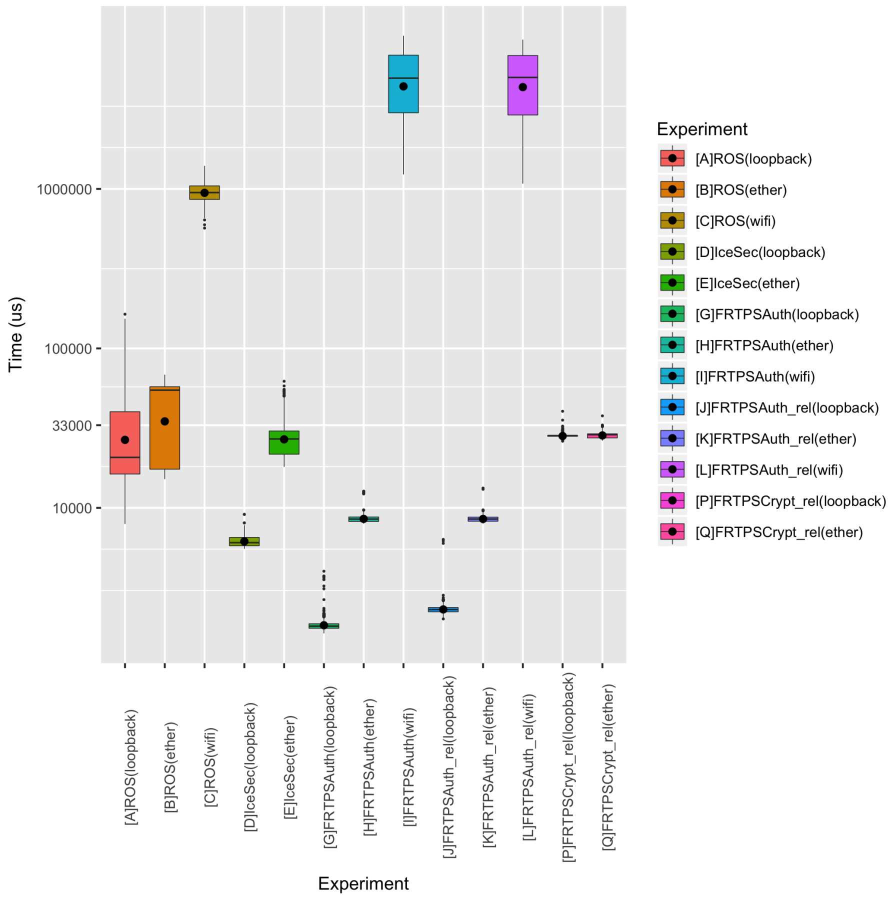

Figure 10: Latency transmitting images with different communication solutions and networks.

Reliable.

Looking at latencies of Fast-RTPS encrypted when transmitting laser data, images or PointClouds, the encryption the messages implies a non negligible increase in latency, but not so big to lose real-time operation.

The same main conclusions drawn from Figure 6 hold also in Figure 7, but with bigger latencies and bigger typical deviations due to a worse network. The latency for laser data when using Ice of Fast-RTPS jumps from 100us when using loopback to 900us for Ice and from 100us to 150us for Fast-RTPS when using the ethernet interface. Using Ice, the average latency for images jumps from 8000us in loopback to 30000us in ethernet. Using Fast-RTPS, average latency jumps from 3000us to 10000us. Compared to the loopback network the difference in performance for laser and images between ROS and other communication solutions shortens.

Again the increase in latency due to reliability in Ice and Fast-RTPS is negligible. The increase due to encryption is a litte bit more relevant, but keeps the real-time operation.

In the WiFi network scenario (corresponding to Figure 8) the average latency increases and the loss rate too. The tested communication solutions only work in an acceptable way for short sensor readings like laser. And the difference in performance between them reduces, keeping that Fast-RTPS solutions deliver shorter latencies than Ice and much shorter than ROS transport system. Again, the degradation of performance due to adding security is small.

Comparing the numbers for robotic data of different sizes, all the tested communication solutions successfully cope with short data like laser readings (Figure 9). ROS transport system has proven to be very robust to different network conditions and data sizes, being relatively slow when the network conditions are good. Ice-based solutions suffer with large sensor readings like PointClouds, performs well with images in ethernet and loopback networks (Figure 10). Fast-RTP solutions also seem very robust to different network conditions and are fast in good or medium network scenarios. The performance of all communication solutions logically degrades as the network conditions become harder or as the data size increases.

5 Conclusions

This work presents a study on the communication quality provided by different communication solutions used in robotics, when security capabilities are enabled. The robotic applications that are targeted in this study are those which use a distributed middleware (like ROS) and that are typically composed of several concurrent software components, running on regular computers inside robots. The study is based on tailored bench tests to measure the latency and loss rate of a ROS application over different distribution solutions (Ice based or DDS based), when sending popular sensory data (laser, image and point cloud sensor readings) over different kinds of networks (loopback, ethernet and wifi).

In short, the main contributions of the paper are: (i) a quantitative analysis of the impact of the security capabilities for two popular middlewares used in robotics, Ice and Fast-RTPS; (ii) a comparative of these middleware (with and without security) and the standard ROS transport system, which is presented as the baseline/reference; (iii) a description of a meticulous methodology to perform this kind of experiments using a real robot; and (vi) an overall description of the risks and the threat model in this context, and the security solutions available in these systems.

The main conclusion of the study is that, in general, the security capabilities of the analyzed communication solutions have an acceptable impact in latency and loss terms. Therefore, they should be enabled in common cases. As expected, the results heavily depend on the size of transmitted data and network type. The experiments show that neither Ice nor Fast-RTPS behave well with large data types: the loss rate is unacceptable for the biggest type (pointcloud 3D images), with and without security. On the other hand, for small and medium data sizes (laser scan and 2D images), the overhead of security mechanisms is relatively low in local and wired networks, for both solutions.

We hope this paper may contribute to raise awareness of the importance of secure communications on distributed robotic systems, encourage middleware and framework developers to add and support security mechanisms and encourage robot programmers to use secure communications by default.

Acknowledgements

The research leading to these results has received funding from the RoboCity-2030-III-CM project (Robótica aplicada a la mejora de la calidad de vida de los ciudadanos. fase III; S2013/MIT-2748), funded by Programas de Actividades I+D en la Comunidad de Madrid and cofunded by Structural Funds of the EU. It has also received funding from the RETOGAR project (TIN2016-76515-R) from the Spanish Ministerio de Economía y Competitividad.

References

[1] "Iran-US RQ-170 incident," 2011, https://en.wikipedia.org/wiki/Iran%E2%80%93U.S._RQ-170_incident.

[2] A. Greenberg, "Hackers remotely kill a jeep on the highway with me in it," 2015, https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway.

[3] S. Morante, J. G. Victores, and C. Balaguer, "Cryptobotics: Why robots need cyber safety," *Frontiers in Robotics and AI*, vol. 2, p. 23, 2015. [Online]. Available: http://journal.frontiersin.org/article/10.3389/frobt.2015.00023

[4] M. Finnicum and S. T. King, "Building secure robot applications," in *Proceedings of the 6th USENIX Conference*

on Hot Topics in Security, ser. HotSec'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2028040.2028041

[5] C. Cerrudo and L. Apa, "Hacking robots before skynet," Tech. Rep., 2017.

[6] T. Bonaci and H. J. Chizeck, "On potential security threats against rescue robotic systems," in *Proceedings of the 10th IEEE International Symposium on Safety, Security, and Rescue Robotics*, november 2012.

[7] T. Bonaci, J. Yan, J. Herron, T. Kohno, and H. J. Chizeck, "Experimental analysis of denial-of-service attacks on teleoperated robotic systems," in *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ser. ICCPS '15. New York, NY, USA: ACM, 2015, pp. 11–20. [Online]. Available: http://doi.acm.org/10.1145/2735960.2735980

[8] H. Chizeck, T. Bonaci, and T. Lendvay, "Enhanced security and safety in telerobotic systems," Mar. 6 2014, uS Patent App. 13/935,436. [Online]. Available: https://www.google.com/patents/US20140068770

[9] F. Cianfrocca and B. Barnes, "A case study for building cybersecurity policies for industrial robots," 2015.

[10] A. Elkady and T. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *Journal of Robotics*, vol. 2012, 2012.

[11] G. Magyar, P. Sinčák, and Z. Krizsán, "Comparison study of robotic middleware for robotic applications," in *Emergent Trends in Robotics and Intelligent Systems: Where is the Role of Intelligent Technologies in the Next Generation of Robots?*, P. Sinčák, P. Hartono, M. Virčíková, J. Vaščák, and R. Jakša, Eds. Cham: Springer International Publishing, 2015, pp. 121–128.

[12] "What is ICE?" https://zeroc.com/products/ice.

[13] "Documents associated with data distribution service™, v1.4," http://www.omg.org/spec/DDS/1.4.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and N. A.Y., "Ros: an open-source robot operating system," in *Proceedings of ICRA Workshop on Open Source Software 2009*, 2009.

[15] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS*. O'Reilly Media, 12 2015.

[16] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Orca: a component model and repository," in *Software Engineering for Experimental Robotics, Springer Tracts in Advanced Robotics,30*, D. Brugali, Ed. Springer, 2007, pp. 231–251.

[17] L. J. Manso Argüelles, P. Bachiller Burgos, P. Bustos García, P. Núñez Trujillo, R. Cintas Peña, and L. V. Calderita Estévez, "RoboComp: A tool-based robotics framework," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6472 LNAI, pp. 251–262, 2010.

[18] J. M. C. nas, M. González, A. Hernández, and F. Rivas, "Recent advances in the jderobot framework for robot programming," in *Proceedings of RoboCity2030 12th Workshop, Robotica Cognitiva.* UNED, Madrid, July, 2013, pp. 1–21.

[19] P. Fitzpatrick, E. Ceseracciu, D. Domenichelli, A. Paikan, G. Metta, and N. L., "A middle way for robotics middleware," *Journal of Software Engineering for Robotics*, vol. 5, no. 2, 2014.

[20] H. Bruyninckx, "Open robot control software: the orocos project," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, 2001, pp. 2523–2528 vol.3.

[21] J. C. Baillie, "Urbi: towards a universal robotic low-level programming language," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug 2005, pp. 820–825.

[22] "Softbank robotics: Naoqi," https://www.ald.softbankrobotics.com/en/robots/tools.

[23] A. Shostack, *Threat Modeling: Designing for Security.* Redmond, WA, USA: John Wiley and Sons, 2014.

[24] "The stride threat model," https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx.

[25] "Publish/subscribe threat modeling," https://blog.securitycompass.com/publish-subscribe-threat-modeling-11add54f1d07.

[26] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[27] W. Diffie, P. C. Van Oorschot, and M. J. Wiener, "Authentication and authenticated key exchanges," *Des. Codes Cryptography*, vol. 2, no. 2, pp. 107–125, Jun. 1992. [Online]. Available: http://dx.doi.org/10.1007/BF00124891

[28] M. Koschuch, M. Hudler, and M. Krüger, "Performance evaluation of the tls handshake in the context of embedded devices," in *2010 International Conference on Data Communication Networking (DCNET)*, July 2010, pp. 1–10.

[29] Q. Ge and F. Chen, "Strategies for implementing ssl on embedded system," in *2008 International Seminar on Future BioMedical Information Engineering*, Dec 2008, pp. 457–460.

[30] "Cyclonessl," https://www.oryx-embedded.com/cyclone_ssl.html.

[31] "Woldssl," https://www.wolfssl.com/wolfSSL/Home.html.

[32] "The ice manual: Icessl," https://doc.zeroc.com/display/Ice35/IceSSL.

[33] The OpenSSL Project, "OpenSSL: The open source toolkit for SSL/TLS," April 2003, www.openssl.org.

[34] J. McClean, C. Stull, C. Farrar, and D. Mascareas, "A preliminary cyber-physical security assessment of the robot operating system (ros)," in *Proceedings of SPIE*, vol. 8741, 2013, pp. 874 110–874 110–8. [Online]. Available: http://dx.doi.org/10.1117/12.2016189

[35] B. Breiling, B. Dieber, and P. Schartner, "Secure communication for the robot operating system," in *Proceedings of 2017 IEEE Syscon*, 2017.

[36] R. Toris, C. Shue, and S. Chernova, "Message authentication codes for secure remote non-native client connections to ros enabled robots," in *2014 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, April 2014, pp. 1–6.

[37] B. Dieber, S. Kacianka, S. Rass, and P. Schartner, "Application-level security for ros-based applications," in *Proceedings of the 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2016)*, 2016.

[38] R. White, M. Quigley, and H. Christensen, “SROS: Securing ROS over the wire, in the graph, and through the kernel,” in *Humanoids Workshop: Towards Humanoid Robots OS.* Cancun, Mexico, 2016.

[39] “SROS,” http://wiki.ros.org/SROS.

[40] “Why ROS 2.0?” http://design.ros2.org/articles/why_ros2.html.

[41] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ros2,” in *Proceedings of the 13th International Conference on Embedded Software*, ser. EMSOFT ’16. New York, NY, USA: ACM, 2016, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/2968478.2968502

[42] “Dds security,” http://www.omg.org/cgi-bin/doc?formal/16-08-01.

[43] “eprosima fast rtps documentation,” https://eprosima-fast-rtps.readthedocs.io/en/latest/index.html.

[44] “Pitfalls of TSC usage,” http://oliveryang.net/2015/09/pitfalls-of-TSC-usage/.

[45] G. Paoloni, *White paper: How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*, Intel, 2010. [Online]. Available: http://www.intel.es/content/www/es/es/embedded/training/ia-32-ia-64-benchmark-code-execution-paper.html

[46] S. Hemminger, “Network Emulation with NetEm,” in *Linux Conf Au*, Apr. 2005. [Online]. Available: http://developer.osdl.org/shemminger/netem/LCA2005_paper.pdf

[47] “Ggplot2: Box and whiskers plot.” http://docs.ggplot2.org/0.9.3.1/geom_boxplot.html.