Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Mário Macedo Ribeiro

Formalizing ROS2 security configuration with Alloy

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Manuel Alcino Pereira da Cunha
André Filipe Faria dos Santos

December 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

ACKNOWLEDGEMENTS

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Automation developments are becoming essential to industrial restructuring, as it brings more efficient and accurate processes with less associated cost. Consequently, robots are rapidly being deployed in a wide range of scenarios, especially where security is demanded. In such cases, it is critical to employ appropriate procedures to verify both the system's quality and its security.

Following the current growth of cyber-physical system, as well as their usage into various technology domains, the development of software applications is demanding due to the complexity behind the integration of needed services, beyond those provided by the operating system. Hereupon, middleware software to abstract systems hardware are constantly evolving, while offering services that support application development and delivery.

One of the most popular open-source software platforms for building robotic systems is the Robot Operating System (ROS) [1] middleware, where highly configurable robots are usually built by composing third-party modules. A major factor behind its popularity and widespread adoption is its flexibility and interoperability. One drawback of this flexibility, however, lies in the increased security risks that ROS applications face. Alongisde security risks, the arising of performance and scalability issues related to the ROS middleware specification, forced the creation of a new version of ROS.

Robot Operating System 2 (ROS2), which continues to provide a simple, uniform message passing interface to allow components to communicate with each other, now implemented using the Data Distribution Service (DDS) [2] communication protocol, where security guarantees are ensured by DDS-Security specification. Using DDS-Security, it is possible to configure ROS2 to run with security guarantees using the SROS2 toolset [3]. However, improper configuration can still lead security problems.

This thesis proposes a technique, based on the software verification perspective, to automatically verify system-wide properties related to the security configuration of ROS2-based applications. The intended purpose is to model the ROS architecture, as well as the network communication behaviour, in Alloy [18], a formal specification language and analysis tool supported by a model-finder over which, system-wide properties are subsequently model-checked.

KEYWORDS Robotics, ROS, ROS2, DDS, SROS2, Security, Software Verification, Alloy

RESUMO

A constante implementação da ideia de automização de processos tem motivado a reestruturação nos mais diversos setores industriais, com o obejtivo de aumentar a eficiência e precisão nos processos integrados, consequentemente, reduzindo os custos associados. Consequentemente, impulsionam a integração robótica nos mais amplos domínios tecnológicos, especialmente em domínios onde a segurança é exigida. Nestes cassos, é fundamental adotar técnicas apropriadas de forma a verificar tanto a qualidade do sistema, como a segurança do mesmo.

Como resultado do atual crescimento de sistemas sistemas ciber-físicos, nomeadamente sistemas robóticos, bem como sua utilização em vários domínios tecnológicos, o desenvolvimento de aplicações é exigente devido à complexidade da integração dos serviços necessários, tipicamente não fornecidos pelo sistema operativo. De forma a acompanhar o aumento na complexidade destes sistemas, middlewares que permitem abstrair hardware têm sido adoptados, oferecendo serviços que oferecem suporte ao desenvolvimento de aplicações robóticas.

Uma das plataformas considerada como *standard* no que toca ao desenvolvimento sistemas robóticos é o middleware Robot Operating System (ROS) [1], onde robôs altamente configuráveis são construídos atráves da composição modular de *software* externo, oferencedo características como flexibilidade e interoperabilidade aos sistemas integrados. No entanto, a flexibilidade resulta num aumento de vulnerabilidades de segurança, pondo em causa a integridade das aplicações. Além da falta de segurança apresentada, existem também problemas de desempenho e escalabilidade relacionados com a especificação do middleware. Assim, era necessário uma mudança na estruturação do ROS, resultando na criação do Robot Operating System 2 (ROS2).

A nova versão do ROS, o Robot Operating System 2 (ROS2) implementa um protocolo de comunicação de nome Data Distribution Service (DDS) [2], onde para além de garantir serviços de comunicação, fornece diversas especificações, sendo uma delas a especificação DDS-Security, que através de uma metodologia de *plugins*, oferece diferentes métodos de adoção de segurança. Através do uso desta especificação, juntamente com o uso do SROS2 *toolset* [3], é possível configurar o ROS2 de forma a adotar estas medidas de segurança.

Esta tese propõe uma técnica para a verificação automática de *system-wide properties* em aplicações ROS. Esta técnica apresentada baseia-se na formalização estrutural de arquiteturas ROS em Alloy [18], com o objetivo de modelar o comportamento associado à comunicação dentro do sistema, tendo em consideração configurações associadas às propriedades de segurança.

PALAVRAS-CHAVE Robótica, ROS, ROS2, DDS, SROS2, Segurança, Verificação de Software, Alloy

CONTENTS

LIST OF FIGURES

Part I

INTRODUCTORY MATERIAL

1

INTRODUCTION

The concept behind automation development is being incorporated into the industrial world, through the use of flexible tools to assist in the most various scnearios, as it brings efficiency and accuracy to the industry's processes. Robots have emerged into our world as they are used routinely to perform quests done traditionally by human beings. The decreasing cost of robots is one of the main reasons for their increased use. [12] Economics aside, the usage of robots brings increased productivity, safety and more manufacturing production work back to developed countries. [7] Robotics is already the key driver of competitiveness and flexibility in large scale manufacturing industries. [6] Robotics is significantly reliant on a variety of technologies, connected into the same robotic system. Due to the continuous growth of technology in these different domains, robots can be used in a wide range of applications. [27]

Despite the advances in technology, dealing with hardware-level applications becomes highly impractical as the complexity systems increases. Thereupon, developing and writing software code for robot applications is demanding, where multiple aspects must be considered properly [28], as robots became to be integrated into distributed systems through seperated components distributed through the network, connecting different hardware and software modules raises interoperability and communication issues. To solve this issue, modular architectures based on message-passing communication patterns are continually emerging as a middleware layer, offering services to the application layer, easing the development cost, while providing interoperability and communication facilitation. [27, 25]. The requirement for a middleware layer that meets different robot's specfication is a novel approach to enable the creation of robot applications over robotic systems, while supporting features such as robustness and modularity.

The Robot Operating System was created by a collaborative open-source community to contribute in the advancement of robots [17] that enables locomotion, manipulation, navigation, and recognition tasks, complemented by software libraries and tools with the aim of helping build robot applications easily, without the need of building it from scratch. ROS was designed to be flexible, since working with robots requires experimentation, given the variety of different hardware and software, enabling interpolation with potential added components. However, performance and scalability issues arised due to its middleware specification. [28] Additionally, real-time constraints such as fault-tolerance, deadlines, or process synchronization were not supported by ROS, making it unsuitable for safety-critical and real-time systems. [21]

Besides having no middleware support for distributed real-time systems, security features were also not featured by ROS, which started to be demanding for deployed systems. An increasing number of real-time applications, for instance robotic systems, requires security ensurance for protecting real-time sensive data, against unauthorized access. [22]

This lead to the creation of Robot Operating System 2, which continues to provide a simple, uniform message passing interface to allow components to communicate with each other, now implemented using the Data Distribution Service (DDS) [2] specification protocol as its middleware, leveraging for its messaging architecture. Issues concerning system integration and scalability are mitigated by DDS various implementations, due to the several transport configurations provided, making it suitable for real-time distributed systems. DDS also provides a security specification, called DDS-Security, used by multiple implementations, and ROS2 makes use of them to provide features to the deployed robotic systems. [8]

With the extended use of robotic systems, software verification, through the use of formal methods, are necessary to prevent potentially catastrophic consequences, mainly related to security matters, as safety guards are gradually implemented into the software domain. Within this context, Alloy [18, 23] framework enables the expressness the behaviour of systems with rich configurations, due to the combination of both relational and linear temporal logic (LTL) provided by its specification language, consequently supporting model-checking techniques. Model-checking techniques enable far better levels of coverage and, as a result, more reliability than traditional testing, where the system is abstracted as a conventional model, that is automatically checked over performing property verification on finite-state machines. [9]

The proposal of this dissertation is to develop a novel technique to automatically verify system-wide safety properties using Alloy framework, confining a ROS2 system into an abstract model, in order to obtain a prototype tool that can be used by developers to easily detect security configuration issues on their respective robotic application.

1.1 OBJECTIVES AND CONTRIBUTIONS

The first goal of this thesis rests in introducing concepts around the Robot Operating System, contextualizing the evolution behind its framework towards achieving security, where the former version of ROS lacked due to the focus on flexibility. Since ROS2 has been developed over the DDS framework, as its communication middleware, DDS must be properly understood before considering the security aspects. To do so, simple examples will first be studied and introduced, in order to provide ROS-based context to the reader. The domain of autonomous systems, where security is of extreme relevance, namely the Autoware [20] ROS2 platform for self-driving vehicles will also be introduced. To understand SROS2 we intend to configure and run a realistic case study related to Autoware with security guarantees.

The DDS Security standard functionality is evaluated, as well as how security is integrated into ROS2. Since security issues, concerning public networks, are recent to the robotics domain, ROS2 security network design should be analysed structurally.

Security configuration related to SROS2 toolset will be provided in this chapter, supported by an example that accounts multiple security features, those being authentication, encryption and, most importantly, access control, applying restriction constraints to the network and its participants, that by default are not controlled.

The second goal is to extend a previously proposed [10] formalization of ROS applications in Alloy/Electrum [18, 23] to also take into consideration the security configuration defined with SROS2. Using this extension, we intend to explore the viability of verifying simple information-flow security properties. For instance, to ensure that no commands to the vehicle motor can be sent via the infotainment system.

The final goal is to automate the extraction of such formal Alloy models from the configuration files of a ROS2 application, in order to obtain a prototype tool that can be used by roboticists to easily detect security configuration issues.

1.2 DOCUMENT STRUCTURE

The current dissertation structure is divided into four different chapters. Chapter (2) introduces all the concepts related to Robot Operating System, and its evolution as robotic development framework towards achieving system security. The current provided work regarding ROS and security in ROS is also presented in this chapter. Chapter (3) introduces the Alloy framework, as its specification language supported by a concrete example case. Chapter (4) presents the approach developed during this work, which allows the automatic verification of system-wide safety properties in ROS2 applications. Chapter (5) presents the thesis conclusions, alongisde possible future aproaches complementing the current work.

2

ROBOT OPERATING SYSTEM

As aforementioned, this thesis aims to explore the security configuration within Robot Operating System 2 (ROS2) and its security toolset Secure Robot Operating System 2 (SROS2), enabling the specification and model checking using Alloy.

This chapter presents a detailed introduction to the concepts around Robot Operating System (ROS), as well as the evolution approach that ROS faced towards providing security to its deployed systems. Regarding this goal, Data Distribution Service (DDS) and its integration on Robot Operating System 2 (ROS2) must be contextualized beforehand. Thereafter, it will be presented a section dedicated to related work regarding the study of security in ROS based systems.

2.1 BACKGROUND

Robotic systems have emerged into several scenarios, where its usage ranges between basic processes automation, up to full performance over critical tasks, consequently causing the complexity increase in these domains. Concerning the complexity behind writting software code, due to the widely variety presented in the robot's hardware as in fields of action, Robotic Operation System (ROS) presents itself as a middleware system, created to facilitate robotic systems development.

In ROS, software flexibility was valued above all else, meaning that values like security were not considered as a priority, so ROS-based applications tend to face increased security risks, compromising the whole robotic network. As ROS became a standard for many robotic systems, and due to the scale and scope of the robotics grows as they start to be integrated into the real world, security matters must be addressed as priority while developing systems involving robots. [17, 21]

The upgraded version of ROS, Robot Operating System 2 (ROS), presents itself as a framework for developing robotic systems, supported by a standard, the Data Distribution Service (DDS), where multiple middleware implementations are built over this standard, providing ROS applications multiple DDS-based specfications, as well as valued Quality of Service (QoS) settings.

The DDS-Security specification [3], embedded by every DDS implementation supported by ROS, supplies multiple plugins regarding the security domain. As result, ROS2 yields a wider command tool compared to the former version of ROS, as they bring forth to a toolset, the Secure Robot Operating System 2 (ROS2) toolset, concerning the security functionality that DDS-Security plugins offer.

This section introduces necessary background information over the major concepts on which this thesis rests.

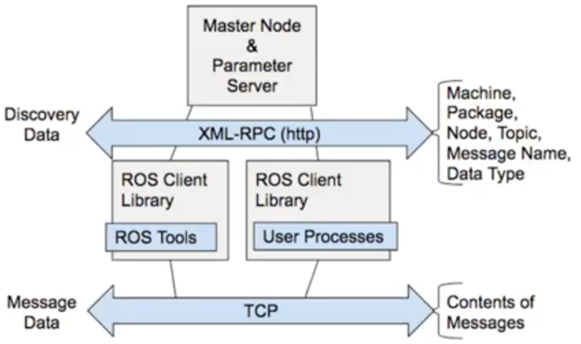
2.1.1 *Robot Operating System (ROS)*

The Robot Operating System was created by a collaborative open-source community, that has undergone rapid development, to contribute in the advancement of cyber physical systems, mainly robotic systems, serving as developer enhancer for the world of robotic applications [17], complemented by software libraries and tools with the aim of helping build robot applications easily, without the need of building it from scratch.

Although Robot Operating System furnishes services, often perceived as operating system's services, such as hardware abstraction, low-level device control and control over message-passing between processes, ROS can not be perceived as a proper operating system, in the sense of process management and scheduling. However, it has a significant impact on the deployed application's performance, with highly complex effects on timing leading to affecting the application's runtime behaviour. In result, the impact of the underlying operating system scheduling is over exceed by ROS. [28, 11]

Fundamentally, ROS is a middleware, as it provides a custom serialization format, a custom transport protocol as well as a custom central discovery mechanism, presenting itself as a distributed layer between the top application layer, the operating systems layer and its communication capabilities.

ROS was designed to provide as much as modularity and composability to the application layer [11] as possible, allowing ROS applications to be built over several software modules, as independent computing processes called *nodes*, that compose together to fulfill the deployment characteristics of the corresponding robot. [25] The communication approach that ROS implements to perform data exchange between nodes is based on a publish-subscribe model, based on TCPROS [4] and UDPROS [5], that makes use of TCP and UDP sockets, respectively. [25]



The former communication architecture supported by ROS focused on a centralization perspective, as it had a implementation of a *Master node*, that controlled every aspect of the communication establishment, meaning that every information exchange between nodes had to go through this master. A ROS Master node must be initialized before any node, as these nodes must also be able to address the ROS Master's location. The ROS Master was also in charge of the parameters, which offer important runtime information about the network nodes.

Due to the sheer wide capabilities controlled by the master, this centralization approach fits the purposes of a research tool, as it is simpler to monitor and analyse the system behaviour. This communication architecture, however, does not scale well since it is heavily reliant on the master node's availability, making it unsuitable for safety-critical and real-time applications. If the master fails, the entire system fails, representing a single point of failure and a huge performance bottleneck.

Many research communities tried to fix these real-time issues by proposing potential solutions, while supporting the same architecture design, that did not fully accomplished the needs of real-time applications. So, it became clear to the ROS community that the framework had architectural limitations that could not be rearranged using the same design approach. [25]

Robot Operating System 2 comes as a complete refactoring of ROS, with the aim of increase the framework's real-time capabilities, by allowing the development of time-critical control over ROS, as it moves away from the former architectural design towards the implementation of an external middleware that can support the production needs of the outgrowing robotic systems. [21, 11]

2.1.2 *Robot Operating System 2 (ROS)*

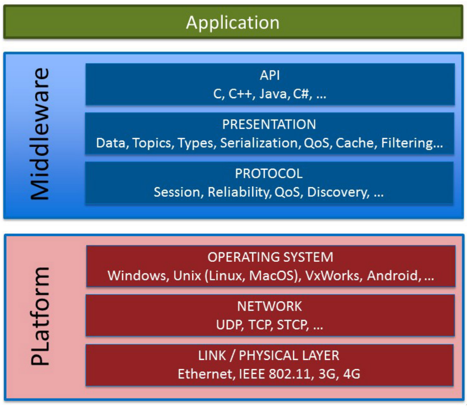
The Robot Operating System 2 (ROS2) was developed with the goal of easing the former ROS architecture. Although ROS2 continues to provide a simple, uniform message passing interface to allow components to communicate with each other, instead of implementing their own middleware specification, Data Distribution Service (DDS) communication protocol is implemented as an abstract middleware interface, through which serialization, transport and discovery is being provided. This implies that adding and integrating a new component into an existing system remains reasonably simple for a developer.

2.1.3 *A Secured Approach: Secure ROS2 (SROS2)*

2.1.4 *Data Distribution Service*

Data Distributed System, known as DDS, is a middleware Object Management Group (OMG) standard, resulted from the need of better interoperability between different vendors middleware frameworks, directly addressing data communication between nodes that belong to a *publish-subscribe* communication architecture, for real-time and embedded systems, that relies on a data-centric approach.

A middleware, such as DDS, is the software layer that comes between the **Platform layer**, where configuration about operating systems and network transport is covered, and the **Application layer**. By that, developers can mainly focus on their applications purposes, instead of concerning about information passing between layers.


Application
Middleware
API
C, C++, Java, C#, ...
PRESENTATION
Data, Topics, Types, Serialization, QoS, Cache, Filtering...
PROTOCOL
Session, Reliability, QoS, Discovery, ...
PLatform
OPERATING SYSTEM
Windows, Unix (Linux, MacOS), VxWorks, Android, ...
NETWORK
UDP, TCP, STCP, ...
LINK / PHYSICAL LAYER
Ethernet, IEEE 802.11, 3G, 4G

Usually, middlewares are implemented by distributed systems, that rely on timing-measures and efficiency when it comes to data-delivering between nodes. DDS aims to ease the complexity of building their own middleware. Complex network configuration, communication establishment, data sharing and low-level details are handled by DDS.

In computer networking, a wire protocol refers to the way of how data is passed from point to point, where multiple applications, that could have different DDS implementations implemented, have to interoperate with each other. DDS wire specification protocol, called **DDSI-RTPS** (*The Real-time Publish-Subscribe Wire (RTPS) Protocol DDS Interoperability (DDSI) Wire Protocol Specification*), offers flexibility and reliability for real-time systems. Since DDS is built over UDP transport protocol, which does not feature reliable delivery of data, so reliability is ensured by the specification of **Quality of Service** (QoS) parameters. Quality of Service parameters offers flexibility over communication, because of the versatility of each QoS parameter. Versatility allows the developer to specify whatever QoS satisfies its system communication needs. A typical case is when the developer wants to ensure reliability over the communication, by ensuring the correctness of packet delivery, which UDP by default does not provide. By changing the QoS parameters, it is possible to change the communication's behaviour into something similar to the TCP protocol.

When it comes to easing the communication establishment, DDS features a **Dynamic Discovery** within nodes, by providing automatic discovery between endpoints and useful information about them. Network configuration is also guaranteed by this feature, where DDS participants are accessible across the network, as the application uses the same DDS API for communications. **Security** features are also considered, where DDS sets a Plugin interface, that combine different plugins that interact with DDS API. These security matters will be later discussed.

An End-to-End Middleware

DDS disposes a software API supported by a rich documentation about its exact behaviour. By furnishing this specification, it enables third parties applications, such as ROS, to implement this middleware, while auditing and understanding matters are covered by its documentation.

The drawback of using DDS as a middleware is that, by being considered an **End-to-End** middleware, the applications that implement it, must follow its design approach. Although, DDS is still considered flexible when it comes to how distinct systems can make use of it as its middleware, it moulds them around their publish-subscribe communication architecture. As far as DDS is concerned, this is should not be viewed as a problem to the developers.

Data-Centric Approach

Initially, systems used to prioritized the message as the establishment of the interaction, meaning that the system was mainly concerned if the message was correctly passed between nodes. This way of structuring is called Message-Centric approach, where the key is the message itself, and not its content.

DDS architecture approach is based on a concept named **Data-Centric**, that instead of focusing on the reliable transmission of the message, the distributed infrastructure takes more responsibility, where the message-data itself is the focal point. Here, the infrastructure formally defines the data and imposes rules over it (**modular**), with the continuous awareness of the contents in the data space, where known structures exchange these contents. Therefore, the data is independent of any particular functionally of any application that uses it, because the data is described by its state, making it easily **scalable**.

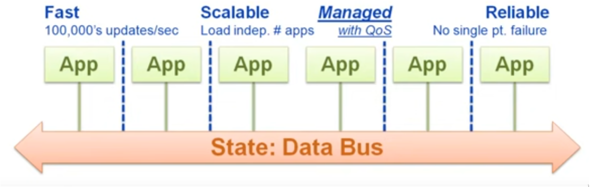
The essence of data centricity is that DDS must know the data schemas, treating the data values as known objects and controls how the data is retrieved to the applications. Thus, DDS is responsible for providing the definition of 'truth' regarding the message management. Whereas, in the traditional message-centric approach, each application must store the message state and treated them as its own, meaning that the 'truth' of the message state is subjective. Even though this offers some flexibility to the application developers, state managing often leads to inconsistency. Rather than forcing each application to deal with this complexity of defining the data space, DDS directly implements and provides controlled, managed, secure data sharing.

DDS Architecture

The Data-Centric approach does not define the full DDS architecture, but only how messaging is treated as a technique. Other technologies, such as Databases supports data-centricity, but usually these are not scalable when it comes to accessing data by multiple applications, updates are slow and highly centralized, meaning that reliability of data could be potentially compromised. Since DDS uses the concept of a transport-independent virtualized **Data Bus**, scalability is not a issue, since network resources are efficiently used, and inside local networks DDS uses multicast, so that the load on the network is independent of the number of applications. There is no centralization whatsoever, maintaining the awareness on the information flow.

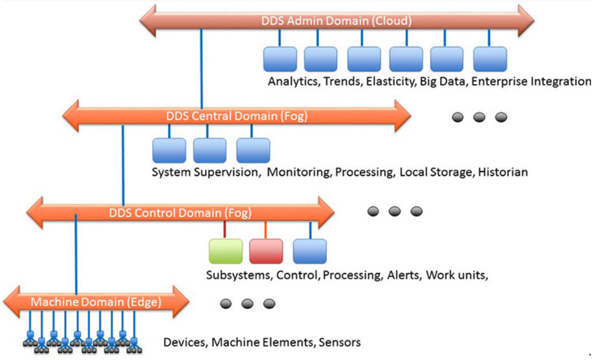
Reliability is ensured by implementing a buffer mechanism, where both publisher and subscriber has a corresponding buffer where the samples of data are processed. The subscriber receives these data samples on the receive buffer, and acknowledges them back to the publisher. The sending of data is multicast, because there might be multiple subscribers to a specific topic, whereas acknowledgments and the re-sending process, that happens if a sample drops, are unicast.

DDS is really about distributing stateful data, acessing it and querying it in motion, in a analogous way to how its done in a database or another data-centric technology, by improving these architectures and making them available for a much wider class of applications and problems.



As previously mentioned, DDS uses network resources very efficiently, meaning that hardware footprint can be reduced, or even with the same hardware, DDS offers significantly greater performance and functionality. Since DDS supports a distributed peer-to-peer topology, with no single points of failure, these capabilities are provided in a more deterministic and reliable way.

DDS communications are also governed with explicit Quality of Service, that allows the system to manage and monitor which applications are able to communicate in which ways with the Data Bus and with each other.


DDS Admin Domain (Cloud)
Analytics, Trends, Elasticity, Big Data, Enterprise Integration
DDS Central Domain (Fog)
System Supervision, Monitoring, Processing, Local Storage, Historian
DDS Control Domain (Fog)
Subsystems, Control, Processing, Alerts, Work units,
Machine Domain (Edge)
Devices, Machine Elements, Sensors

DDS provides flexibility and scalibilty to every sort of systems, from small devices, located at the "Edge" where DDS can be used for high-speed machine to machine communications, up to large systems, where DDS provides scalable access and distribution of information over the whole cloud.

2.1.5 *Robot Operating System 2 - DDS Integration*

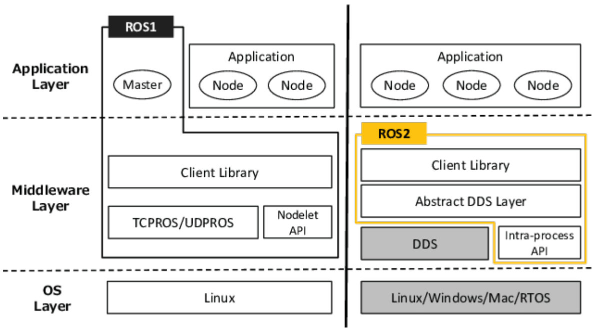
As stated in the previous section, ROS2 was developed with the goal of easing the former ROS architecture, by implementing an abstract middleware interface, through which serialization, transport and discovery is being provided. This means that it continues to be relatively straightforward for a developer to add and integrate a new component into an existing system.

ROS2 Middleware Architecture

ROS2 offers an interface on top of DDS, called **ROS client library** (RCL), which hides much of the complexity of DDS specification API, offering a more friendly approach for the ROS users. However, it still provides access to the underlying DDS implementation for users that might want to integrate different DDS implementations for their use cases.

ROS2 is built over the DDS framework, since it must work with that existing DDS design. So ROS2 applications are actually considered DDS applications, which leads to this provided flexibility and compatibility with other DDS applications.

Native DDS applications can freely interoperate with ROS2 DDS applications, but some restrictions, that module the communication flow and establishment (such as the selected QoS settings), have to be respected.



ROS2 aims to support multiple DDS implementations, even though each of these implementations API specification differ from each other. In order to abstract from the specifics of these APIs, an abstract interface is being introduced, which can be implemented for different DDS implementations, depicted above as **Abstract DDS Layer**. So, the main idea is to support **portability** among DDS vendors, enabling ROS developers to interpolate DDS implementations depending on their requirements at time.

This layer could be removed, if there was no need to use more than one DDS implementation. But this is not the wanted scenario, since it is way more practical to switch the DDS implementation depending on the application needs. This layer was already customized and implemented in ROS1, defined by the Client Library which allows the communication between the Application Layer, where the Master is implemented, and the communication protocols. The main reason behind implementing this middleware layer is so that ROS2 developers can focus on their application, instead of also focusing on all these aspects of communication configuration.

Note that, these implementations offered by DDS are low-level of abstraction, specified by its API, with predefined functions defined by vendors. DDS only defines basic procedures in a more higher level of abstraction.

Information flow through the Middleware

As stated, ROS client library hides the complexity of the DDS specification API from the developer. Although the ROS structures are yet to be introduced, RCL must work under these ROS structures, to abstract the developer from the DDS structures.

In order to keep the passing of information through the middleware layer, a structure mapping between ROS and DDS must occur. The data transformation accounts the DDS implementation that is being considered at runtime, implemented at the `Abstract DDS Layer`. Then, the converted data is handled by the DDS wire protocol, RTPS, where the DDS publish-subscribe architecture and their respective structures comes to practice. Then, the reverse process must be ensured as well, to fully complete the information flow, where the data must be converted back to ROS structures before being returned to the ROS client library.

The way that ROS handles communication is by introducing the **message** definition, which characterizes every data structure concerning the information exchange between ROS participants. Former ROS developers relied on this message approach, since ROS code was driven by the structure of these messages. So, keeping the messages' idea was necessary in ROS2 kept the message structure because of the value Since that provides a simple format.

The data structure mapping between ROS and DDS must account the preservation of the messages' structure. The RCL works under the ROS messages' `.msg` files, and explained, these messages must be converted into messages' `.idl` files, to be used by the DDS transport, The conversion functions, that create DDS in-memory instances from ROS files, are being generated for each DDS implementation. Depending on the DDS implementation that is being used, the `.idl` can be directly used, since the conversion were already provided by the conversion functions.

For instance, lets consider a simple example of a publisher node, that publishes a message that is later subscribed by a subscriber node. So, the publisher creates the message, and by publishing it, the message is passed on to the RCL layer. Then the `.msg` is converted into its corresponding DDS `.idl` file. The process of transporting the converted message is treated by DDS, ensuring that the message reach its destination, which is the DDS structure that maps to the ROS subscriber node. Finally, the reversed process is considered, where the message is converted back to a `.msg` file, and RCL handles the message delivery to the subscriber node.

DDS as a Dependency

The modularity approach that ROS makes use of, where multiple modules should be applied when needed, promoting the idea of code reuse, while reducing the number of dependencies attached, was considered a priority goal upon ROS development, since it offer great portability.

By relying on an outside middleware, these goals highly depend on the middleware implementation that is being used. As far as dependencies are concerned, DDS implementations have light sized dependencies, often related to language implementation libraries, easing the complexity behind installing and running dependencies for ROS developers.

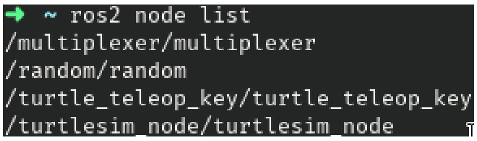
2.1.6 *Robot Operating System 2 Data Model*

As previous stated, ROS provides a way of data distribution between nodes base on published and subscribed data. Before considering about how the data is passed and treated by publishing and subscribing, nodes must be studiedly covered.

Nodes can be perceived as processes that will likely perform computation over the network. The network is comprised of many nodes, running simultaneously and exchanging data between them, with each node having its corresponding module purpose. Generally speaking, nodes are programs, built over a package, where their software code treats data that has been exchanged to each of them.

In ROS2, there is a modular convention about how to program nodes, where each class inherits from the Node object, easing the life of the programmer. For instance, lets considered a well-known example called

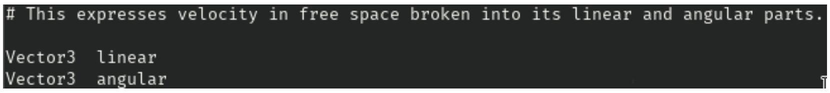
TurtleSim, which is a simulator typically used for learning ROS, mainly composed by 2 nodes, that perform together towards moving a turtle. Additional nodes were implemented so security matters could be supported. After launching each node using the predefined ROS2 run command, ros2 run <package_name> <node_name>, the running nodes can be easily checked using the command ros2 node list. Note that, each node has its corresponding package, and nodes can be connected within a single package or between multiple packages.



```
➜ ~ ros2 node list
/multiplexer/multiplexer
/random/random
/turtle_teleop_key/turtle_teleop_key
/turtlesim_node/turtlesim_node
```

The communication between nodes is done by message-passing, where the nodes involved exchange information through **messages**. A message is defined by its type, also known as its **interface**, which can be primitive (integer, string, boolean), or can be defined by a complex data structure, where multiple data types are assigned to their corresponding variables.

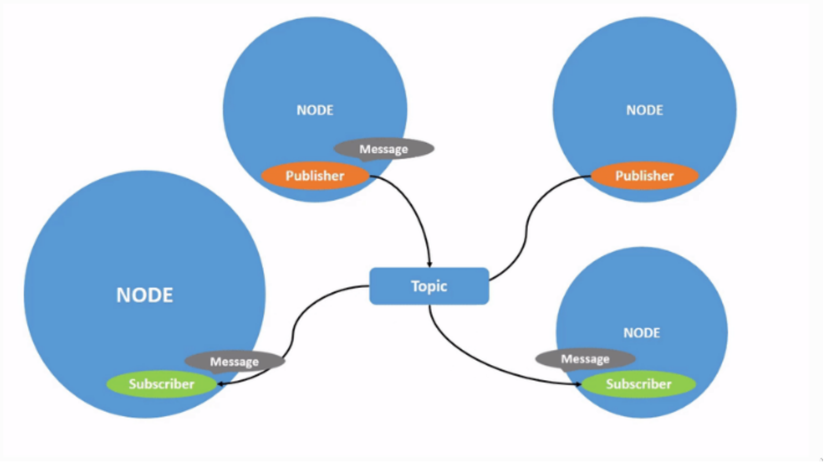
The message that triggers the turtle movement, in the Turtlesim example, is composed by multiple data movement variables regarding the linear and circular movement of the turtle. The interface is called Twist and it is a ROS predefined geometry message.



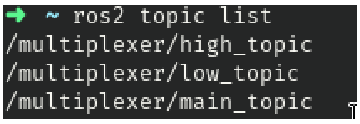
```
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
Vector3  angular
```

Messages are passed through via the pattern of publishing and subscribing over a **topic**. A node publishes the message to any number of topics, that are then subscribed by nodes that want to get access to that message. Topics provide a multicast routing scheme, where publish data can be casted into multiple subscribers.



The topic is created upon specifying in the node either a publisher or a subscriber instance, providing the topic name as an argument. After building and executing the node, their associated topics are created. The available topics can be listed by running the predefined ROS command ros2 topic list.

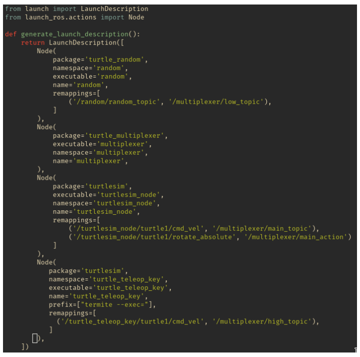


```
➜  ~ ros2 topic list
/multiplexer/high_topic
/multiplexer/low_topic
/multiplexer/main_topic
```

If the nodes that want to exchange data through a topic are implemented in the same package, the respective subscriber and publisher nodes must have the same topic name instantiated. The configuration is pretty simple, whereas if the nodes are specified in different packages, a technique of **remapping** must be used.

The topic list related to the `Turtlesim` is depicted above. The 3 listed topics are vital for the communication process, and they all correspond to the `multiplexer` node. This happens because remapping between topic names is used to establish communication between packages, as explained above.

Both launching and remapping process are done by running a ROS **launch file**, that simultaneously launches each node, providing a simpler way of monitoring the system nodes.

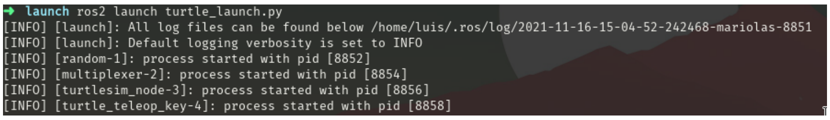


```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtle_random',
            namespace='random',
            executable='random',
            name='random',
            remappings=[
                ('/random/random_topic', '/multiplexer/low_topic'),
            ]
        ),
        Node(
            package='turtle_multiplexer',
            executable='multiplexer',
            namespace='multiplexer',
            name='multiplexer',
        ),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            namespace='turtlesim_node',
            name='turtlesim_node',
            remappings=[
                ('/turtlesim_node/turtle1/cmd_vel', '/multiplexer/main_topic'),
                ('/turtlesim_node/turtle1/rotate_absolute', '/multiplexer/main_action')
            ]
        ),
        Node(
           package='turtlesim',
           namespace='turtle_teleop_key',
           executable='turtle_teleop_key',
           name='turtle_teleop_key',
           prefix=["termite --exec="],
           remappings=[
             ('/turtle_teleop_key/turtle1/cmd_vel', '/multiplexer/high_topic'),
           ]
       ),
    ])
```

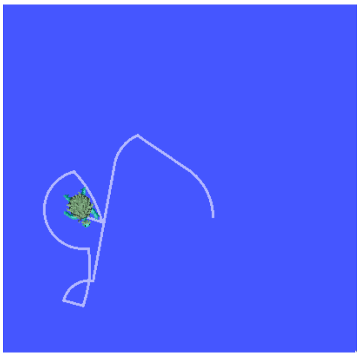
As it can be seen, every node is instantiated in this file, and have each topic remapping defined in `remappings` section. Also, distinctive namespaces allow the system to start the nodes, without any name nor topic name conflicts. However this technique has some flaws attached, since it does not furnish a way of launching nodes in a separated terminal, needed for user interaction purposes, like input reading.

Running the following ROS command `ros2 launch <file_name>`, will automatically compile and run the launch file.



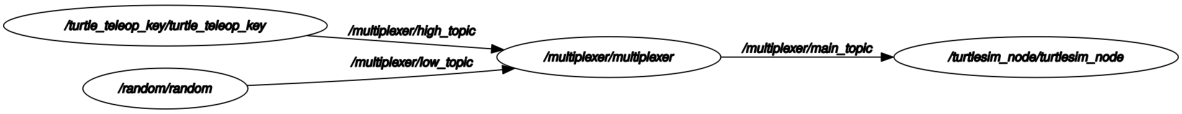
```
➜ launch ros2 launch turtle_launch.py
[INFO] [launch]: All log files can be found below /home/luis/.ros/log/2021-11-16-15-04-52-242468-mariolas-8851
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [random-1]: process started with pid [8852]
[INFO] [multiplexer-2]: process started with pid [8854]
[INFO] [turtlesim_node-3]: process started with pid [8856]
[INFO] [turtle_teleop_key-4]: process started with pid [8858]
```

Every network node is now running simultaneously in the background. The only foreground running node is the turtlesim one, because it has an associated interface showing the turtle movement.



For understanding reasons, the reader may want to see how does the network architecture is organized. ROS2 provides a GUI tool called **rqt**, that helps the programmer in manipulating the network elements, in a more user-friendly way. This tool has a graph visualizer plugin that allows the programmer to view the network graph and how the elements are connected.

While the nodes are running in the background, the reader may want to run the following command `rqt_graph`, in a separated terminal. The network graph related to the `TurtleSim` is represented bellow.



The `multiplexer` node, being the main node in this particular network, subscribes to two different topics from two different nodes.

— The `turtle_teleop_key` node corresponds to a keyboard controller, that sends movement commands through the `high_topic` to the `multiplexer`.

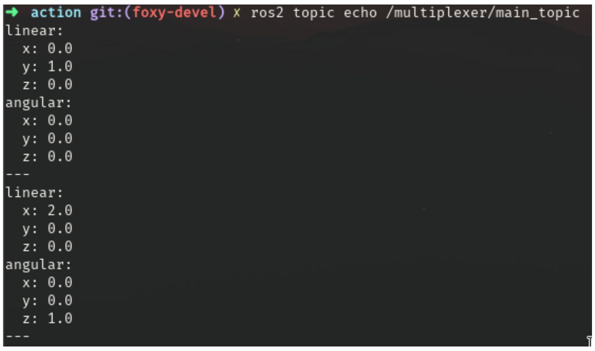
— The `random` node sends random movement commands through the `low_topic` to the multiplexer.

Since the `multiplexer` has two different active subscriptions, managing two different turtle movement values, it must keep them synchronized with each other. This is achieved by setting different priorities to each subscription, where the keyboard node has priority over the random controller, meaning that when the keyboard publishes movement data through its corresponding topic, the `multiplexer` must process that received data. **Timers** are also used, since they provide a useful way of managing these topics, by time-assigning, alternately changing the priority after the timer runs out, through its corresponding callback function.

The `multiplexer` also has a publisher instance defined, since it has an active connection through the `main_topic` connection to the `turtlesim_node` node, that is responsible for controlling and projecting the turtle movement. The `main_topic` will be in charge of the values passed to the turtlesim node, instructing the turtle to move.

Given the priority set at each moment, the `multiplexer` has to forward the movement command, either from the keyboard or the random controller, through the `main_topic`, enabling the turtle movement. Every aspect related to the publisher-subscriber pattern, in this ROS2 system, is treated over remapping of topic names.

The data passed through the `main_topic` can be checked using the command `ros2 topic echo main_topic`. As expected, the data has both linear and angular vectors defined because of interface used.



```
➜ action git:(foxy-devel) ✗ ros2 topic echo /multiplexer/main_topic
linear:
  x: 0.0
  y: 1.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
---
```

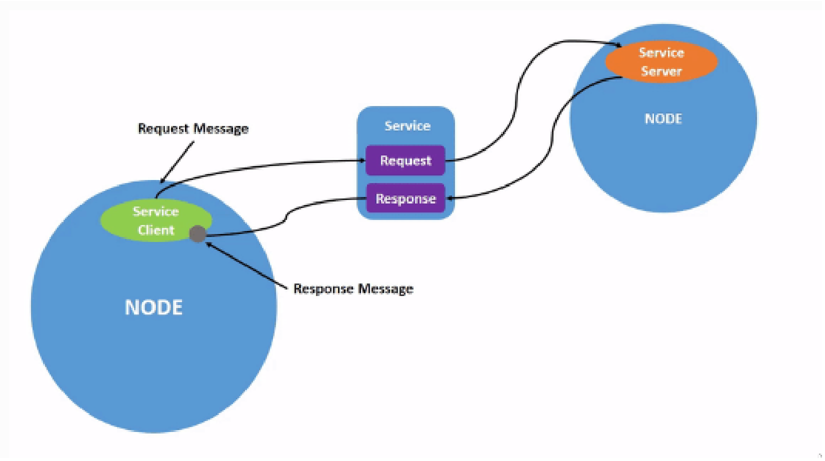
Other ROS concepts

The `TurtleSim` application given as example above, so the reader could get familiar with the basics concepts about ROS, does not address all the existing ways of data-exchanging between nodes.

Even though topics are the most conventional way of communication, because of its multicast scheme, subscribers can not be perceived by the publishers, so logging and matching becomes rather difficult. Node synchronization within the network is also not featured by topics.

SERVICES **Services** allows a client, that can also be seen as a topic subscriber, to request data from a server, that likewise a topic publisher, furnish data through a service. Services works under a synchronous policy, since the data is only provided when the client node makes a request. Each service is always linked to just one server node, and does not maintain active connections.

In ROS1, services used to work under a synchronous policy, where the service client would stall waiting for the response of the service server. While, in ROS2, services are **asynchronous**, since it is possible to specify a callback function that is triggered when the service server responds back to the client.

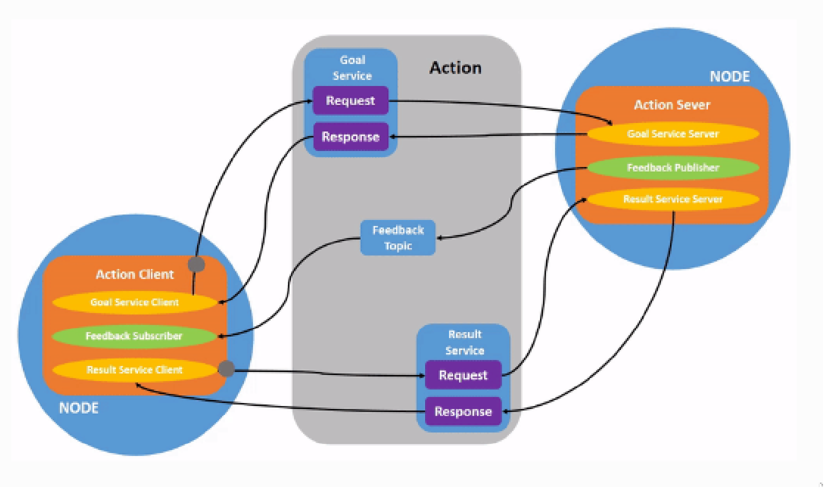


The interface used on services messages differ from the topics interface. That is because a service is composed by a request and response, so their data structures may differ.

Suppose, considering the `TurtleSim` example, that a client wants to make a request to move the turtle by providing coordinates (valued in integers) and a *theta* angle. A conventional way of the node server sending a response back to the client is by passing a boolean value, ensuring that the request has been processed, and a string value regarding the name of the turtle. By that, it must be considered two different data structures. Therefore, this considered service interface should be similar to the following:

```
float32 x
float32 y
float32 theta
---
bool check
string name
```

ACTIONS Another way of exchanging data is by setting goals through **Actions**. Actions, likewise services, also uses a client-server model, but they were design for other purposes rather than only processing a request and sending back a response. Actions are intended to process long-running tasks, where the client sends a goal request to the server node, that confirms the receiving of this goal. Before returning a response back to the client, the server can send feedback back to the client. Unlike services, actions can be cancelled, so the return response could not be acknowledge.



The interface structure used in actions is also different from the others, since it provides an additional data structure to the services interface, corresponding to the feedback feature that actions provide.

Considering the same example of the `TurtleSim`, a client may want to request a goal providing a *theta* angle, to trigger the turtle rotation. To keep the client updated, one way of the server feedbacks the client is by sending the angle rotation that is left to be executed. After the execution, the server may want to acknowledge the client by confirming the angle passed as a request, confirming that the rotation has been executed properly. By that, it must be considered three different data structures. Therefore, this considered service interface should be similar to the following:

```
float32 theta
---
float32 delta
---
float32 remaining
```

This given example is commonly used in robotic systems for navigation, where the robot travels to a position, previously requested by a client. While its traveling, it can send information about the transition state. When the robot reaches the predefined position, the server acknowledges the client by sending a result message.

Note that, in both interfaces introduced, the `--` divides the each structure that are used in each communication. In services, the request structure must be separated from the response structure. In actions, there is also the feedback structure which is the last structure defined.

PARAMETERS Another relevant concept behind ROS is the existence of nodes **Parameters**, that allows individual configuration of the network nodes. In the former version of ROS, the node parameters were controlled by a shared **Parameter Server**, managed by each ROS Master node responsible for the its corresponding network. As stated before, in ROS2 each node declares and manages its own parameters. Usually parameters are saved using files related to the nodes, but parameters can also be updated and checked using `get` and `set`, respectively. Additionally, using a parameter function callback, the node's parameters can easily be edited.

INTERFACES Although it was already mentioned, since it was needed for explaining each method of communication for nodes in ROS, **Interfaces** should be duly studied. As explained before, each type of communication has its different corresponding interface, and they must be specified in different folders with unique namespaces. The folder namespaces used are the following:

— Topics messages: `msg/`

— Services messages: `srv/`

— Actions messages: `action/`

Recall the `TurtleSim` example and its corresponding package `turtlesim`. Suppose a case where, it could be possible to add boxes into the turtle GUI, blocking the turtle movement though those boxes. To add boxes, the topic message considered should have two different values regarding the x and y axis value related to the turtle map. This interface could be named as `Boxes`, with the `.msg` extension (`Boxes.msg`).

```
    float32 x
    float32 y
```

In your node's code, the following import must be specified:

```
turtlesim/msg/Boxes
```

LIFECYCLED NODE ROS2 introduces a new definition of node, called **lifecycled node**, as addition to the traditional one. A lifecycled node is defined by its current state. Its state can either be unconfigured, inactive, active or finalized. This can be useful, since it provides a structural way of setting up the node and separating all its state functionalities. The transition through states are done by running a callback function. Nodes that need initial configuration before exchanging data within the network can be perceived as lifecycled nodes, like sensors, where its detection must be ensured before the data reading/publishing process.

NODE COMPOSITION A concept that both ROS and ROS2 provides is the idea of composing nodes into a single **process** for improved **performance**. Usually a node is attached to a single process, but it is possible to combine multiple nodes into a single porcess, structurally abstracting some network parts.

There is a slight difference about how ROS and ROS2 approaches the node composition. In the former version of ROS, node composition was done over the combination of `nodelets`. In ROS2, instead of calling it a `nodelet`, it is called a `Component`.

Formerly, `nodelets` were designed to ease the cost of overusing TCP for message-passing between nodes, that if the message consisted huge sized data could negatively affect the performance on the network communications. By grouping performance critical nodes, nodelets provided a local way of message-passing between the considered nodes.

Supported by the former idea of `nodelets`, `components` are compiled into shared libraries. A component itself does not furnish node composition, since a component is a node with special features. Although, each component can be loaded into a **component container** process at runtime in the network. By loading components into these containers, the node composition is ensured.

Note that, nodes do not provide this feature since they are compiled into executables that need to be previously compiled. Components can be also be configured to be loaded at compile-time, where a single executable runs multiple components. Components can be also configured in a ROS launch file, using containers as well.

Node composition could also be applied for security matters. Suppose a scenario where multiple nodes respect the same security policies. By combining them into a single process, the mapping into this set of rules would be direct, easing the usage of security enclaves. This is will be later explained in the `security` section.

2.1.7 *The need for Security*

A major factor behind its popularity and widespread adoption is its flexibility and interoperability. In ROS, this flexibility was valued above all else, meaning that values like security were not considered as a priority, so ROS applications tend to face increased security risks.

Also, ROS works over a publish-subscribe pattern, which is very useful, since it brings various forms of decoupling and transparency. However, because of this transparency, publishers within ROS are unable to control the consumption of their data and subscribers, most of the times, cannot identify the integrity of their received data. This means a malicious actor could potentially inject code or vulnerabilities into a library, which could then be reused by another unsuspecting developer.

These concerns about security take on particular importance, since robots are becoming more frequently used and can directly affect the physical world. **Cyber-physical** systems, commonly related to the arising of the automatizing concept of robots, feature unique vulnerabilities that exploit both cyber and physical nature of these devices, mainly by sensing and actuating. Sensing becomes relevant when it comes to the potential leak of information about the surroundings of its robot. Whereas, actuating is related to the functionality of the robot, where physical safety is the main concern.[14, 26]

ROS faces known vulnerabilities due to its architecture and model nature. ROS internal communication is built around TCP ports, allowing robots to be built using several modules, that can be distributed. However, this modularity results in the exposure of TCP ports, that usually leads to attacks that compromise the data itself. There is also a concern about how ROS manages the communication between nodes. In ROS, the node-to-node communications are settled in plain text. Despite the fact that this way of handling communications bring certain benefits, especially better performance and data recovery, it is evident that data content is vulnerable to unauthorized usage.

As discussed previously, this is mainly due to the fact that ROS had its own middleware, which didn't provide any security features. The 'centralization' behind ROS architecture, where everything was controlled and connected to the ROS Master node, was also a major fact to the need of a new implementation where the middleware could not also facilitate the establishment of communications, but also could provide security features. ROS2 adoption of middleware interface for data exchange offers a flexible way to build appropriate defenses for a given application.

DDS ensures not also communication features, but also guarantees security properties, where ROS's flexibility is preserved despite this increase on security. The **DDS-Security** is a specification that serves as an add-on to the DDS specification, defined by a set of plugins combined in a Service Plugin Interface (SPI) architecture. The main plugins provided by DDS-Security are the following ones:

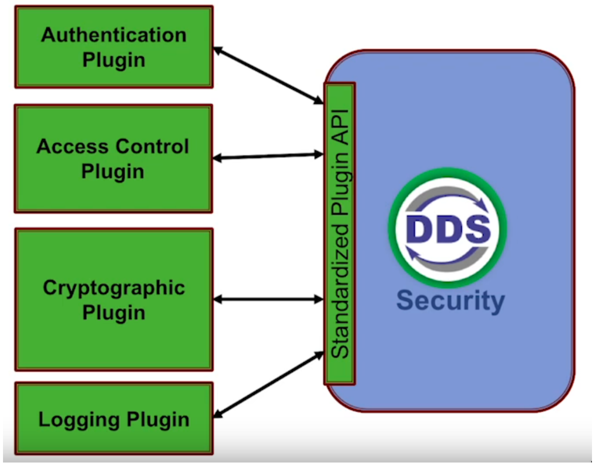
— **Authentication**: Verification of each identity of a given Domain Participant, by assigning a private key and a public key to each one, as well as an certificate that relates the participant's public key to its corresponding specific name.

— **Access control**: Provides control over DDS-related operations with the support of the Authentication plugin. It requires two files per domain participant signed by a Certificated Authority, that specifies how the domain should be secured and their permissions.

— **Cryptographic**: Operations related to cryptographic purposes, such as encryption are handled by this plugin.

— **Logging**: Provides logging over events.

— **Data tagging**: Add tags to data samples.



These plugins can be customized since different applications require different needs when it comes to security. That is why, as depicted above, there must be a **Standardized Plugin API** that acts as an interface between modules and DDS security protocols, while including all aspects of secure communications. ROS2 is deployed without security mechanisms by default, but DDS can provide security guarantees such as authentication and access control with a variant called DDS-Security. Using DDS-Security it is possible to configure ROS2 to run with security guarantees using the **SROS2** toolset. Most of these tools supported by SROS2 is contained within the ROS Client Library (RCL), and includes the following features:

— Support for security files for each domain participant, where in ROS2 are seen as a process, and each process must be associated with a set of these files.

— Support for both permissive and strict enforcement of security, meaning that can be considered two approaches when it comes to dealing with security files.

— Support for a master "on/off" switch, that enables or disables every security feature.

The SROS2 has been designed to work with any secure middleware. However, not all middleware is considered open-source and highly depends on the ROS distribution in use.

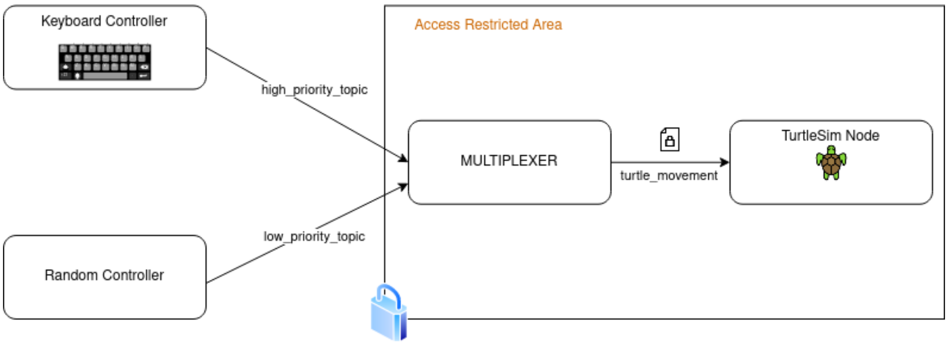
Since this security process involves a lot of new technology, the user must be aware of improper configuration, that can still lead to security problems. However, SROS2 CLI include a tool, named `ros2 security`, that aims to help those who don't want to set it all up by themselves.

TurtleSim with SROS2

Now that the concept behind SROS2 is introduced, the `TurtleSim` application given as an example in the previous section, can now be treated using security features related to the nodes involved.

The `multiplexer` node handled the commands responsible for the turtle movement. Note that, the turtle is implemented in the `turtlesim` node, so both `multiplexer` and `turtlesim` nodes are directly connected through a specified topic. `Multiplexer` acts as topic selector between two different subscribed topics, where each of them was respectively associated with a priority. Based on the priority set at each moment, the `multiplexer` node forwards the command related to the selected topic into the `turtlesim` node, by topic remapping, which triggers the turtle movement. Even though the `multiplexer` handles the two specified subscribed topics that receive movement data from both `random controller` and `keyboard` nodes, its not exclusive to the `turtlesim` node, as it is still possible to directly publish commands to the topic that handles the turtle movement, since the security properties have not been implemented yet.

In order to guarantee this exclusivity where the `multiplexer` is the only node that directly publishes data to the topic that enables the turtle movement, is by specifying **access control** files. The remaining nodes should be considered as untrustworthy, denying any potential undesired turtle movement originated by these nodes that tried to publish data through the topic. The idea is to encapsulate both `multiplexer` and `turtlesim` nodes, as the `multiplexer` monitors and manages all the data intended to manipulate the turtle movement, similar to a reversed proxy server.



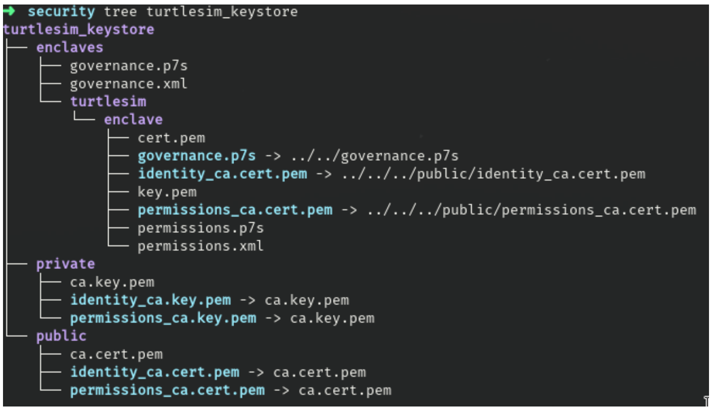
Note that, as it is now, the only nodes that are supervised by the `multiplexer` are the `random controller` node and the `keyboard` node. However, it can be implemented new nodes with ease, as it is only required to remap topics in the launch file and create as many subscriptions in the `multiplexer` as many implemented nodes.

So, in technically terms, a **SROS2 keystore** must be initiated. ROS2 provides a command that permits the creation of **keystores** by using the `security` keyword. This command enables features regarding security matters, that has access to the SROS2 tools related to the DDS-security artifacts.

The keystore is a created directory where DDS security features are stored. By creating a keystore directory, then it can be later sourced to be used by RCL to provide security to the associated ROS2 network.

```
ros2 security create_keystore turtlesim_keystore
```

The created keystore directory structure is displayed bellow:



```
➜  security tree turtlesim_keystore
turtlesim_keystore
├── enclaves
│   ├── governance.p7s
│   ├── governance.xml
│   └── turtlesim
│       └── enclave
│           ├── cert.pem
│           ├── governance.p7s -> ../../governance.p7s
│           ├── identity_ca.cert.pem -> ../../../public/identity_ca.cert.pem
│           ├── key.pem
│           ├── permissions_ca.cert.pem -> ../../../public/permissions_ca.cert.pem
│           ├── permissions.p7s
│           └── permissions.xml
├── private
│   ├── ca.key.pem
│   ├── identity_ca.key.pem -> ca.key.pem
│   └── permissions_ca.key.pem -> ca.key.pem
└── public
    ├── ca.cert.pem
    ├── identity_ca.cert.pem -> ca.cert.pem
    └── permissions_ca.cert.pem -> ca.cert.pem
```

By creating the keystore, three different sub-directories will be generated.

- The `enclaves` directory contains the security tools related to each enclave created. An **enclave** is a group of ROS nodes, controlled by the same set of security rules, defined in its corresponding enclave directory. Each enclave includes files needed to enable security, such as CA certificates and their own private key (`key.pem`). Besides containing each enclave created (for instance, in the figure above, a `/turtlesim/enclave` enclave is created), this directory also has a governance policy document `governance.xml`, as well as a signed copy file related to the CA permissions, `governance.p7s`.

- The `public` directory contains material that is permissible as public. A Certificate Authority certificate, `ca.cert.pem` is stored in this directory. This file stores the CA **public key** and it is used to validate the identity and permissions of each ROS network node by the CA. Note the existence of `identity` and `permissions` folders with their respective certificate/key file. This is due to the fact that DDS supports the separation of identity and permission chains, however ROS usually uses the former `ca.cert.perm` file, meaning that only a CA is used for both these processes.

- The `private` directory contains material that is considered private. A Certificate Authority certificate, `ca.cert.pem` is stored in this directory. This file stores the CA **private key**. It is used to modify the network policies, such as access permissions, and to add new participants. Similar to the `public` directory, the CA key corresponding to its identity and permissions can be stored in their corresponding individual directories.

The reader may want to create as many keystores as needed, but RCL can only use a keystore's artifacts at once. The directory is defined by the SROS2 environment variable `ROS_SECURITY_KEYSTORE`, that needs to be previously sourced. Environment variables sourcing will be explained further on.

Once the keystore has been created, the respective enclaves can be implemented. By using the tag `create_key`, with the keystore namespace provided (in this case is `turtlesim_keystore`), each enclave can be created.

Although the idea of enclave has already been introduced, the full context behind it is yet to be explained. As mentioned, an enclave is a group of ROS nodes that follow the same security policy. Since an enclave is a

DDS domain term, typically a node is an abstraction of a DDS `Participant` (previously mentioned as a Domain Participant). In ROS2, the usage of an enclave is done upon running a node, which causes the creation of a running process. This means that, enclaves are used by processes, providing them security based on their artifacts and specification.

Since the mapping is between processes and enclaves, the reader may wonder, if node composition into a single process is a reliable way for matching multiple nodes simultaneously to the same enclave. Well, this is actually a good approach but nodes can not be perceived as domain participants, while considering node composition, due to the fact that composing participants results in non-negligible overhead.

There is also not convenient to compose nodes as individual participants, as far as security is concerned. That is because each participant has its own security identity related to its node, and by composing participants, they all share the same OS memory space, consequently access data is shared between them, and permission distribution becomes rather difficult to handle.

So, each participant must be matched to a node shared context, instead of being directly related to a specific node.

Each enclave security artifacts are used by each participant, meaning that each node mapped into the same context is controlled by the same enclave. Thereby, the initial given definition of an enclave is not totally correct, since a participant can either be perceived as single node or as multiple node shared context.

Usually the node **namespace** are quite relevant upon creating enclaves, since the default way of implementing enclaves creates a namespace hierarchy path depending on the node namespace, so node remapping could affect its corresponding enclave path. With the introduction of node contexts, the enclave path is now disassociated from node namespaces, having its own resource identifier. This approach is also appropriate when composing nodes from different packages (that consequently have different namespaces) scenario is considered, where the path of the shared enclave must be common to each node.

```
ros2 security create_key turtlesim_keystore /security_enclave/
```

In order to run these nodes concerning the created keys, all the packages related to the network must be re-built using the **Fast DDS** additional flag.

```
colcon build --symlink-install --cmake-args -DSECURITY=ON
```

The following exports need to be sourced to force SROS2 security features. The reader may want to put these exports into a shell script, and then add this to the `PATH` environment variable.

```
export ROS_SECURITY_KEYSTORE=/path/to/the/created/keystore
export ROS_SECURITY_ENABLE=true
export ROS_SECURITY_STRATEGY=Enforce
```

These exports are related to relevant SROS2 environment variables, that the reader must be aware when considering the SROS2 tools.

- **ROS_SECURITY_KEYSTORE**: This variable sources the root of the created keystore. By sourcing this location, ROS2 knows where the security artifacts are stored.
- **ROS_SECURITY_ENABLE**: This variable is the SROS2 enabler.
- **ROS_SECURITY_STRATEGY**: As stated before, SROS2 provides two different approaches when it comes to dealing with security files. This variable handles that property, and it can be set either to `Enforce` or `Permissive`.

Now that all ROS2 security prerequisites have been successfully configured, the network can be launched.

Since SROS2 has no support for launch files, each node needs to be launched separately on its own terminal, with regard of their respective keys. Later, this process is simplified by launching a shell script.

```
ros2 run turtle_multiplexer multiplexer --ros-args --enclave /multiplexer

ros2 run turtlesim turtlesim_node --ros-args --enclave /turtlesim_node/tu
```

As pointed previously, DDS has several implementations that can be considered when the ROS2 network is being configured. The ROS2 distribution usually dictates what is the default middleware implementation of DDS being used. Most of these ROS2 distributions uses the same default middleware implementation, which has the name of **Fast DDS** (formerly named `Fast RTPS`). The features that stand-out the most in this implementation are the promotion of an interface generator (`Fast DDS-Gen`) and the usage of `RTPS` communication protocol, which guarantees performance, real-time behaviour, efficiency, scalability and, most importantly, security. As explained earlier, the Fast DDS security flag, `-DSECURITY=ON`, must be considered upon building the ROS2 packages. Although, the reader may want to set a different DDS implementation. In order to do this, the reader should install the DDS middleware software, following the implementation specification and dependencies, and then rebuild the workspace forcing the DDS implementation to change. ROS2 has a environment variable, `RMW_IMPLEMENTATION`, that allows the replacement of the middleware that is being used.

The creation of these keys only ensures that the communication between the `multiplexer` and the `turtlesim` is authenticated, and the exchanged data is encrypted, so Access Control is yet to be implemented.

By using tools like `tcpdump`, the data packets can be displayed and this encryption can be easily verified. If no SROS2 security is regarded (no export made), the data is passed in plain-text, as expected. After sourcing the security features, using the exports specified above, the packet displayed is encrypted, and its larger since additional information is considered to increase safety.

SROS2 - Automatic key generation

Before considering applying access control to the `TurtleSim` example, let this SROS2 feature be introduced. SROS2 supplies tools to automate the process of generating enclaves and their respective permissions files.

First a `policy.xml` needs to be created, while the ROS2 network is running with SROS2 security enabled, so that the file stores information about the network.

```
ros2 security generate_policy policy.xml
```

Now, the created policy file can be used as a reference to automate the process of key generation. To do this, both keystore and policy must be given as arguments to the following SROS2 command.

```
ros2 security generate_artifacts -k turtlesim_keystore -p policy.xml
```

Access Control

In order to provide access control, the permission files, of the enclave that is being considered, need to be modified. This is ensured by adding security permissions to these files, with the mandatory signature of the CA.

However, a suitable way of editing the permission file, `permissions.xml`, that dictates how the enclave manages the permissions within the network, is by creating a policy file and running the following command.

```
ros2 security create_permission turtlesim_keystore /enclave policy.xml
```

Besides changing the permissions file, the CA signature is also generated, meaning that the permissions modification have been successfully implemented in the considered enclave.

This `policy.xml` passed as an input for easing the alteration of the enclave permissions, follows a restrict SROS2 **policy XML schema**. This schema makes use of a few concepts, that are relevant to understand how the schema design was conceived.

- **Mandatory Access Control**: Permissions to allow the access to an object are pre-established by a set of authorization rules, and can not be modified by any subject.
- **Principle of Least Privilege**: This principle is to guarantee that only the minimum access privileges are used to the intended purpose.
- **Privilege Separation**: The access privileges that a subject owns are separated, decreasing the possibility of a security vulnerability.
- **Separation of Concerns**: Distribution of the system security concerns into each corresponding section.

2.2 RELATED WORK

This section aims to present previous works that attended security matters related to the deployment of robotic systems using Robot Operating System as its application enhancer. When developing real-time systems, safety matters are often referred as critical, because of the overall integration with the real world. Security however, it is measured by evaluating different security issues by properly analysing the system model. However, due to the high nonlinearity and complexity of real-time systems, implementing such a thorough analysis method in near real-time remains a significant difficulty.[15]

The deployment of real-time systems results in the importance of concerning about safety in a performance point of view, resulting of the demanding time-critical scenarios. Many studies were made regarding the performance in both ROS and ROS2 (also regarding DDS Quality of Service policies [25]), by analysing using performance measurement approaches, providing a guided and supported review on how performance can directly affect time critical situations, where safety is envolved.[25, 11]

System security concerning network exposure, often explored by unauthorized acess and data leaking, can be treacherous and it is considered a complex subject, due to the abundance of different network security technologies that do not cover every security aspect, since absolute security does not exist, as new vulnerabilites arise from the tecnology evolution.[19] The creation and deployment of security countermeasures are essential upon configurating the network towards achieving security. Within this vast topic, several different avenues of endeavor come to mind, each deserving of a substantial study. Network security means exploring the network beforehand by computer intrusion detection, traffic analysis, network monitoring, alongside many other practical networking security aspects.[24]

The literature concerning the network security enhancment that Robot Operating System 2 furnishes, by offering the SROS2 toolset, is quite limited. Most of the existing work is on the exploration of the former version of ROS in terms of port exposure, contextualized in the approach considered to protect the system network.

Many researches were made regarding this issue that ROS faces, one in particular that explored the IPv4 address space of the Internet for instances of ROS, named *Scanning the Internet for ROS: A View of Security in Robotics Research*[14], with the goal of identifying ROS vulnerable hosts, mostly master nodes since they provide information about their related topics and node's parameters, mainly by port scanning, so that developers could be aware of the possibility of exposure of their robots. The performed scans furnished information about hosts that could either be a sensor, an actuator or even a simulator. Topics were also identified since they provide evidence of what is likely to be available to an attacker. This study is rather relevant because of how easily can attackers gather information about potential robots, and control them further on, through the public Internet, making it unavoidable to develop mechanisms concerning security.

A different approach was previously presented[16] when it comes to protecting the ROS network, which primarily focused on applying security measures on the application layer, treating ROS architecture as a black-box, by mainly running an Authentication Server, storing certificates and files related to trusted domain participants, while controlling and providing session keys related to the communication process. Even though the message data can not be perceived, unauthorized nodes can still collect information about the flow of the data, since ROS architecture is responsible for the publish-subscribe interaction, meaning that security issues still persist which cannot be handled on the application level alone.

Secure Robot Operating System (SROS)[29] was initially developed as an experimental tool (later evolved to SROS2 as a supporting tool for ROS2), which supports TLS for all socket transport, node restrictions and chains of trust, guaranteeing publishers authorization when it comes to publish to a specific topic. Another worth-mentioned tool is Rosbridge[13], which provides a WebSocket interface to ROS and corresponding server to allow interaction between applications and ROS nodes, by using TLS as support and also access control over topics and API calls.

3

ALLOY SPECIFICATION FRAMEWORK

As aforementioned, the main problems that this thesis aims to tackle are security vulnerabilities and data processing over big workloads of streaming data. In this chapter, it is intended to explore technologies that are relevant to overcome the above-mentioned challenges (??) as well as previous developed work that has the same or similar goals as this thesis (??).

3.1 BACKGROUND

Enterprises, particularly startups and small and medium businesses, are gradually shifting to the Cloud to outsource data and processing. [?] As advantageous as this may be, the sharing of content with third-party systems, poses challenges about privacy and security. As previously stated (??), traditional strategies for achieving confidentiality, such as a user encrypting their data with their own key or even using techniques such as homomorphic encryption, make some computations unfeasible, hence alternative methods must be investigated.

3.2 RELATED WORK

4

CURRENT WORK

5

CONCLUSION

BIBLIOGRAPHY

[1] https://www.ros.org.

[2] Object management group. data distribution service (dds). https://www.omg.org/omg-dds-portal/.

[3] Ros 2 dds-security integration. https://design.ros2.org/articles/ros2_dds_security.html.

[4] Tcpros. http://wiki.ros.org/ROS/TCPROS.

[5] Udpros. http://wiki.ros.org/ROS/UDPROS.

[6] What robots are used today? https://sciencing.com/what-robots-are-used-today-12745877.html.

[7] Why is robotics important? https://www.onlynaturalenergy.com/why-is-robotics-important/.

[8] Kai Beckman and Jonas Reininger. Adaptation of the dds security standard for resource-constrained sensor networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–4, 2018.

[9] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pages 99–114. Springer, 2017.

[10] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. Verification of system-wide safety properties of ros applications. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7249–7254, 2020.

[11] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn B Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[12] John J Craig. Introduction to robotics. 2005.

[13] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. Rosbridge: Ros for non-ros users. In *Robotics Research*, pages 493–504. Springer, 2017.

[14] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. Scanning the internet for ros: A view of security in robotics research. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8514–8521, 2019.

[15] Ruisheng Diao, Vijay Vittal, and Naim Logic. Design of a real-time security assessment tool for situational awareness enhancement in modern power systems. *IEEE Transactions on Power systems*, 25(2):957–965, 2009.

[16] Bernhard Dieber, Severin Kacianka, Stefan Rass, and Peter Schartner. Application-level security for ros-based applications. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4477–4482, 2016.

[17] Vincenzo DiLuoffo, William R Michalson, and Berk Sunar. Robot operating system 2: The need for a holistic security approach to robotic architectures. *International Journal of Advanced Robotic Systems*, 15(3):1729881418770011, 2018.

[18] Daniel Jackson. Alloy: a language and tool for exploring software designs. 62:66–76, 2019.

[19] Merike Kaeo. *Designing network security*. Cisco Press, 2004.

[20] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296, 2018.

[21] Jongkil Kim, Jonathon M Smereka, Calvin Cheung, Surya Nepal, and Marthie Grobler. Security and performance considerations in ros 2: A balancing act. *arXiv preprint arXiv:1809.09566*, 2018.

[22] Man Lin, Li Xu, Laurence T Yang, Xiao Qin, Nenggan Zheng, Zhaohui Wu, and Meikang Qiu. Static security optimization for real-time systems. *IEEE Transactions on Industrial Informatics*, 5(1):22–37, 2009.

[23] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. page 373–383, 2016.

[24] Gerald A Marin. Network security basics. *IEEE security & privacy*, 3(6):68–72, 2005.

[25] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pages 1–10, 2016.

[26] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascarenas. A preliminary cyber-physical security assessment of the robot operating system (ros). In *Unmanned Systems Technology XV*, volume 8741, page 874110. International Society for Optics and Photonics, 2013.

[27] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *2008 IEEE Conference on Robotics, Automation and Mechatronics*, pages 736–742. Ieee, 2008.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[29] Ruffin White, Dr Christensen, I Henrik, Dr Quigley, et al. Sros: Securing ros over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060*, 2016.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.