MDPI

Article

TalkRoBots: A Middleware for Robotic Systems in Industry 4.0 †

Marwane Ayaida [1,*], Nadhir Messai [1], Frederic Valentin [1] and Dimitri Marcheras [2]

[1] CReSTIC EA 3804, Université de Reims Champagne Ardenne, 51097 Reims, France; nadhir.messai@univ-reims.fr (N.M.); frederic.valentin@univ-reims.fr (F.V.)
[2] SATT Nord, 25 Avenue Charles Saint-Venant, 59800 Lille, France; dimitri.marcheras@univ-reims.fr
* Correspondence: marwane.ayaida@univ-reims.fr; Tel.: +33-698-959-387

† This paper is an extended version of "A new middleware for managing heterogeneous robot in ubiquitous environments" published in the Proceedings of 2020 8th International Conference on Wireless Networks and Mobile Communications (WINCOM), Reims, France, 27–29 October 2020.

Abstract: This paper proposes a middleware called TalkRoBots that handles interoperability issues, which could be encountered in Industry 4.0. The latter proposes a unified communication approach facilitating the collaboration between heterogeneous equipment without needing to change neither the already used software nor the existing hardware. It allows heterogeneous robots, using both open and proprietary robotic frameworks (i.e., ROS, ABB, Universal Robots, etc.), to communicate and to share information in a transparent manner. It allows robots and Industrial Internet of Things (IIoT) devices to communicate together. Furthermore, a resilience mechanism based on an Artificial Intelligence (AI) approach was designed in order to allow automatically replacing a defective robot with an optimal alternatively available robot. Finally, a remote interface, which could be run through the Cloud, allows users to manipulate fleets of robots from anywhere and to obtain access to sensors' data. A practical scenario using five different robots has been realized to demonstrate the different possibilities. This demonstrates the cost effectiveness of our middleware in terms of its impacts on the communication network. Finally, a simulation study that evaluates the scalability of our middleware clearly shows that TalkRoBots can be used efficiently in industrial scenarios involving a huge number of heterogeneous robots and IIoT devices.

Keywords: middleware; communications; IIoT; AI; robots; resilience; ROS

check for updates

Citation: Ayaida, M.; Messai, N.; Valentin, F.; Marcheras, D. TalkRoBots: A Middleware for Robotic Systems in Industry 4.0. *Future Internet* **2022**, *14*, 109. https://doi.org/10.3390/fi14040109

Academic Editor: Vijayakumar Varadarajan

Received: 16 February 2022
Accepted: 20 March 2022
Published: 29 March 2022

1. Introduction

The concept of Industry 4.0 is currently changing both the manner in which manufacturing systems as well as production business models have been thought of [1]. Smart Factories, which are a key feature of Industry 4.0, can be built to manufacture "Smart Products" with more flexibility. They are composed of several inter-connected advanced devices, including networked sensors, conveyor, robots or Cloud servers. Flexible Manufacturing Systems (FMS) are increasingly studied due to the evolving nature of industrial requirements. Hence, there is an emerging interest to consider cooperation between several heterogeneous robots evolving in ubiquitous environments [2]. These robots are not only limited to classical industrial ones, but they are increasingly including co-bots, Autonomous Guided Vehicles (AGVs), drones, etc. In fact, mobile robots enable, by essence, the enlargement of the scope of reactive behaviors in the dynamic products' routing since they navigate freely in the production floor or in the warehouse [3]. Moreover, this free guidance brings a higher level of manufacturing flexibility. As an example, commercial AGVs such as Kuka LMR IIWA can receive task orders from real-time production systems and manufacture execution systems.

Moreover, the technological evolution in mechatronics, computer engineering and Information and Communication Technologies (ICT) allows the deployment of flexible manufacturing applications including a fleet of heterogeneous robots that share data between them, with Industrial Internet of Things (IIoT) devices and Cloud servers [4]. These heterogeneous robots are not de facto easily interoperable since they are built by different manufacturers supporting various programming frameworks, which are not standardized. A fleet of heterogeneous AGVs, such as as KMP omniMove, Ridgeback and AGV MIR can, for instance, cooperate together to transport big products. Note also that a new generation of mobile robots is currently developed for warehouses to reduce travel time, improve safety or increase productivity.

The implementation of cooperative tasks involving heterogeneous robots in FMS or warehouse needs, however, to consider several communication and interoperability aspects for a powerful and cost-efficient operation of plants. Due to the increase in system complexity and intelligence, there is a growing need for transparent communication and data sharing between multi-robot systems, IIoT devices and the human operators who supervise the production tasks, especially in the case of resilient flexible manufacturing scenarios. Therefore, the design of a new middleware allowing a transparent communication between heterogeneous devices and an easy dynamical tasks' scheduling becomes crucial in the context of agile manufacturing systems.

This work proposes a new middleware, denoted TalkRoBots, that has been recently patented [5]. It allows several industrial robots (ABB, Universal Robots, KUKA, etc.), AGV and drones that support both open and proprietary robotic frameworks (ROS, ABB, etc.) to communicate, share information and cooperate in a transparent manner. Our technology allows robots to acquire information about the environment (temperature, light, humidity, video, sound, etc.) from ubiquitous sensors. TalkRoBots includes a Cloud server that allows operators to schedule dynamically the manufacturing scenarios, supervise their status, obtain access to sensors' data and re-configure a scenario when a fault occurs, for example, when using an Artificial Intelligence (AI) mechanism based on an Analytic Hierarchy Process (AHP) [6]. Note here that once the scenario is launched, the robots will automatically handle their tasks in a fully distributed manner, without needing any exchange with the Cloud platform. The deployment of such technologies is easy and cost-efficient since it requires only the integration of an embedded PC into existing production systems.

This paper extends our first study presented in [7] and proposes the following main contributions:

- An architecture for the proposed Middleware is presented and compared with existing works.
- The supervision of a monitoring Cloud application is detailed.
- A resilience mechanism based on IA is integrated in Middleware.
- An analytical study related to the performances of the suggested Middleware is introduced.
- A simulator was developed in order to validate the results of the analytical study.
- Real experimental results are discussed.

The next section of this paper provides an overview about the works related to the existing robotic frameworks and middleware. Then, Section 3 describes the architecture of our middleware, namely TalkRoBots. This section provides also some details about the differently used exchanging modes and some aspects related to the security of the middleware. Section 4 presents Cloud services offered by TalkRoBots. After that, the performances of the TalkRoBots are mathematically analyzed and evaluated by simulations in Section 5. Experimental results are presented in Section 6. This paper finishes with a conclusion that summarizes our contributions and provides some hints about future works.

2. Related Works

The current section contains a review of the most well-known robotic frameworks and middleware. Although TalkRoBots is not really a framework, the lack of technology surrounding interoperability between robots makes it legit to compare our framework to existing frameworks and middleware that are close to what TalkRoBots achieves and the way it is related with robots. A robotic framework is a set of programming tools and libraries targeting, simplifying and accelerating the development of a complex robotic system. The Robotic middleware's definition is similar. The term "robotic middleware" is used to describe something that binds various modules of a robotic framework. Thus, the most essential task of a middleware is to provide an infrastructure to allow communication between several software parts within a robotic system, including robotic framework tools. Therefore, the robotic frameworks and middleware will be introduced together.

First, we introduce the most well-known robotic framework: Robot Operating System (ROS). The latter is completely distributed and considered as a meta-operating system [8], which provides several services that are close to those of classical operating systems, such as hardware abstraction and low-level device control, as well as some high level libraries and tools, such as rviz and rqt (for data visualization and experimentation).

Convincingly, ROS tends to be a standard in robotic development and this reality is upheld by several publications, such as in [9–12].

Player is one of the pioneer well-known robotic frameworks [13]. Its main advantage consists in a client/server architecture allowing the development of robotic tasks using any programming language supporting Transmission Control Protocol/Internet Protocol (TCP/IP) communication protocols. Moreover, robot control can be performed from any computer as long as a network connection is available. In addition to the fact that Player supports a wide variety of physical robots and accessories, it is usually used in simulation together with either the Stage 2D simulator [14] or Gazebo (for 3D simulations) [15,16]. However, the server, running on the robot, offers only a few pre-configured functionalities that allow collecting data from robots and control their actuators. Furthermore, Player is only compatible with some specific robots, since the server must be able to translate API commands into the actual robotic native language.

ARIA (Adaptive Robot-Mediated Intervention Architecture) is fundamentally a C++ library that provides different tools allowing the control of mobile robots (https://www.eecs.yorku.ca/course_archive/2010-11/W/4421/doc/pioneer/aria/main.html, accessed on 15 February 2022) [17]. In addition, with the integration of the software input/output (I/O) with custom equipment and the support of all MobileRobots/ActivMedia devices, ARIA provides a tool called ArNetworking, which implements simple remote system activities and user interfaces. Furthermore, ARIA is a client-side software for low and high level robotic control that can be run in single or multi-threaded processes. However, ARIA supports only a limited number of robots that are manufactured by MobileRobots and ActivMedia.

ASEBA is a robotic framework that is built using an event-based architecture for the distributed control of mobile robots [18]. It aims to assist user in the control of multi-processors robots or groups of single-processor units that are real or simulated. Its main advantage is to provide access to microcontrollers using high-level languages. Note that ASEBA has been integrated with D-Bus, making it a robotic middleware [19]. However, ASEBA is still suitable for programming educational robots, such as Thymio, which is manufactured by Young Generations.

Carmen [20] is an open-source set of software for mobile robot control made by the Carnegie Mellon College. It integrates all basic algorithms, including base and sensor control, logging, obstacle avoidance, localization and mapping, which are handled in a modular manner and use IPCs (InterProcess Communication) to communicate and to allow process monitoring. Moreover, it provides robot equipment support for various platforms, such as iRobot's ATRV and B21R, ActivMedia's Pioneer I and II, Nomadic Technology's Scout and XR4000. Aside from robotic devices, some robotic sensors, such as the Sick LMS laser range finder, the GPS devices using the NMEA protocol or sonars and Hokuyo's IR sensors (Hokuyo URG-04LX, www.hokuyo-aut.jp, accessed on 15 February 2022), are handled through different APIs. It is noted that Carmen, running under Linux operating system and programmed with C++ language, is not control or real-time focused.

OpenRDK is another framework made to accelerate the building of robotic systems. It is built in a modular manner with a main object called agent. Inside this agent, several modules, meaning threads, can be launched dynamically. Since OpenRDK relies on a distributed architecture, modules can be launched on different machines and communicate through what is called a repository. It could be described as publish/subscribe system, but for more details, an extensive description is presented in [21,22].

Orca is an open-source C++ framework supporting Linux, Windows and QNX Neutrino operating systems. It is meant to allow great reusability by the normalization of interfaces and providing high-level API to make repositories of components. The idea around the components is detailed in [23,24] while [25] describes the lightweight characteristics of Orca.

Orocos is a free software project written in C++. Once again, it is mainly component based and it supports a few different vendors. Focusing on the real-time control of robots, it relies on three parts: a kinematics and dynamics library, a Bayesian filtering library and Orocos Toolchain. The last part allows interactive scripting, the use of distributed processes and code generation. In [26], a more detailed description of Orocos is available and [27] describes how to use it for indoor navigation. It can also be integrated relative to ROS with Orocos RTT since 2009.

Urbi is divided into two parts. Firstly, it has a component library developed in C++ and is called UObject. It specifies a standard method for using motors, sensors and algorithms. To handle the interaction between the different modules of the system, the Urbi script language is used. It is a high-level language that shares similarities with Python and LUA and supports event-based programming to be used in asynchronous systems. Similarly to almost all frameworks, Urbi wants to help seamlessly integrate different robotic software modules. The architecture software design for an exploration robot is presented in [28]. Although it can be interfaced with ROS for an extended toolset, only some robots are compatible.

There are also few researchers that focused on the cooperation between robots frameworks and IoT. For example, in [29], authors proposed a semantic-driven framework for coordination between robots and IoT devices. They proposed a test-bed experiment. However, they did not consider any interoperability issues between robots and IoT devices and even between the robots themselves.

There are also some attempts to propose a Middleware for Intelligent Automation (MIA) that acts as a gateway between field devices and Decision Support Systems (DSSs), supporting integration software such as industrial communication standards, fog computing, and Big Data warehousing such as in [30]. This work targets interoperability and Cloud functionality. However, it neglects fleet management and resilience capability.

As summarized in Table 1, contrary to several of these middleware and frameworks, TalkRoBots' goal is not to offer an abstraction level for robot programming or an easier method to control and monitor robots through APIs. TalkRoBots is only here to provide a simple message format with which robots can communicate between each other and become aware of their environment with the integration of external devices. TalkRoBots introduces a real decentralized horizontal communication between robots without typical client/server architecture. Moreover, using TalkRoBots, the robotician's work will not change since he will still program his robots in their native language. Thus, there is no need for a specific driver for each robot's model in order to use TalkRoBots. Secondly, TalkRoBots offers a web application that we call a robotic Cloud, which can be used for monitoring and to control some aspects of fleet management.

Table 1. Frameworks/middleware comparison (× not implemented/designed for; ✓fulfilled).

Robotic Frameworks	Programming Language	OpenSource	Distributed Architecture	Fleet Management	Robotic Cloud	Resilience	Dynamic Scheduling	Simulation	Robot/Robot Communication
ROS	Python, C++	✓	✓	✓	×	×	×	✓	×
Player	C++, Tcl, Java, Python	✓	×	✓	×	×	✓	×	×
ARIA	C++, Python, Java	✓	×	✓	×	×	×	×	×
ASEBA	Aseba	✓	✓	✓	×	×	×	×	×
Carmen	C++	✓	✓	✓	×	×	×	✓	×
OpenRDK	C++	✓	✓	×	×	×	×	×	×
Orca	C++	✓	✓	×	×	×	×	×	×
Orocos	C++	✓	✓	✓	×	×	×	×	×
Urbi	C++ like	✓	×	✓	×	×	×	×	×
TalkRoBots	Python	✓	✓	✓	✓	✓	✓	✓	✓

3. TalkRoBots Middleware Presentation

This section presents the main contributions of our patented Middleware denoted by TalkRoBots. It provides a description of TalkRoBots' architecture, as well as the messages' exchange procedure and some considered security aspects.

3.1. Architecture

The proposed *TalkRoBots Middleware* architecture is organized in several layers that interact between them, as depicted in Figure 1. They include the Robotic Applicative layer and the Robotic Framework layer, such as ROS, ABB, ARIA, ASEBA, etc. The former interacts with remote monitoring and supervision applications. The latter controls directly the hardware of the heterogeneous robots. Moreover, since TalkRoBots is designed to allow an easy integration of heterogeneous robots in the same fleet, a package, named "TalkRoBots Package", is integrated in the Robotic Framework in order to allow communication between the native robot's programming language and the first level of the Middleware TalkRoBots by defining how to receive, send and handle messages. The other layers are detailed below:

- TalkRoBots Layer 1: This layer is designed as a gateway that handles the messages exchanged between the "TalkRoBots Package" and either ConnectToBots or the TalkRoBots Layer 2. On the one hand, it allows transparent and homogeneous communication between the heterogeneous robots and the external devices of the fleet. On the other hand, it handles the exchange between the Robotic Framework by using the "TalkRoBots Package" and the second level of Middleware TalkRoBots through ConnectToBots.
- ConnectToBots Layer: This layer acts as a communication bus that monitors the different connection ports of the embedded PC (USB, Serial port, etc.), which is connected to the robot. Then, it allows the identification and recognition of the plugged devices, including external sensors and actuators (temperature sensor, camera, RFID tag, etc.). Therefore, it allows adding new capacities to the concerned robots related to the plugged devices and makes them available to all those in the fleet.
- TalkRoBots Layer 2: It includes all services that are developed by the operator, such as the "Collaboration" and "Election" services. It is responsible for sharing data between the Cloud and the robotic fleet.

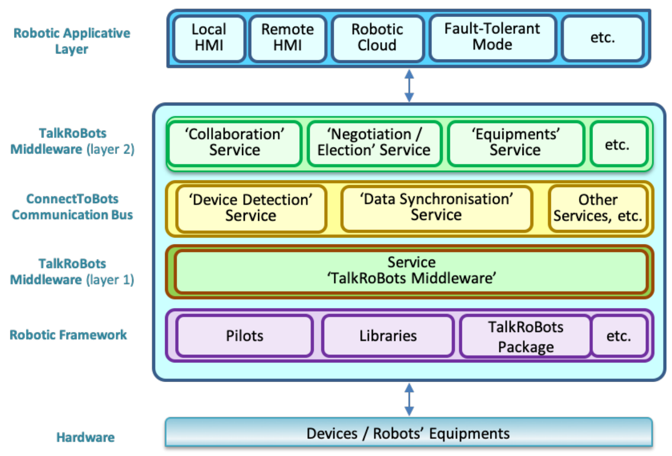

Robotic Applicative Layer
Local HMI
Remote HMI
Robotic Cloud
Fault-Tolerant Mode
etc.
TalkRoBots Middleware (layer 2)
'Collaboration' Service
'Negotiation / Election' Service
'Equipments' Service
etc.
ConnectToBots Communication Bus
'Device Detection' Service
'Data Synchronisation' Service
Other Services, etc.
TalkRoBots Middleware (layer 1)
Service 'TalkRoBots Middleware'
Robotic Framework
Pilots
Libraries
TalkRoBots Package
etc.
Hardware
Devices / Robots' Equipments

Figure 1. TalkRoBots Middleware.

For more details about the architecture of TalkRoBots Middleware, the reader is refereed to Patent [5].

3.2. Communication

In this section, we detail the mechanism used for the exchanges between the robots and IIoT devices. It involves the robots' detection and the configuration procedures allowing the identification of how to handle messages related to the robotic framework and IIoT devices.

First, we will present how a given robot is handled by the middleware. In fact, each robot will typically receive a task to be launched. Thus, the middleware must be able to know the method of transmitting this task to the robot. To do that, three configuration files were specified, as described in Appendix A.

3.2.1. Routing

The routing allows the communication with all other robots of the fleet. It is ensured thanks to routing tables stored at the first level of the middleware associated with each robot. Within this table, a route is defined by the following:

- Transmission Control Protocol (TCP) sockets used to communicate with the other robots and devices;
- The ID of the robot or the device associated with each socket;
- The list of sensors plugged on the remote robot.

To build this table, each robot/device has to register, according to the procedure presented in Figure 2a, the details of the integration of a new robot/device in a given fleet. Moreover, an example that illustrates the messages exchanged when a Pioneer robot joins a fleet is provided in Figure 2b. In fact, once a robot is connected to layer 1, middleware broadcasts a message including its listening interface (IP address), its ID and the list of its plugged sensors. Once this message is received, the other robots establish encrypted (cf. Section 3.3) TCP connections with the newly integrated robot. Then, they exchange their IDs and their list of available sensors. In this manner, the linked middleware has a routing table including all the other robots and their available sensors. From now on, each robot is able to communicate with the others or any sensor connected to the middleware using this routing table.

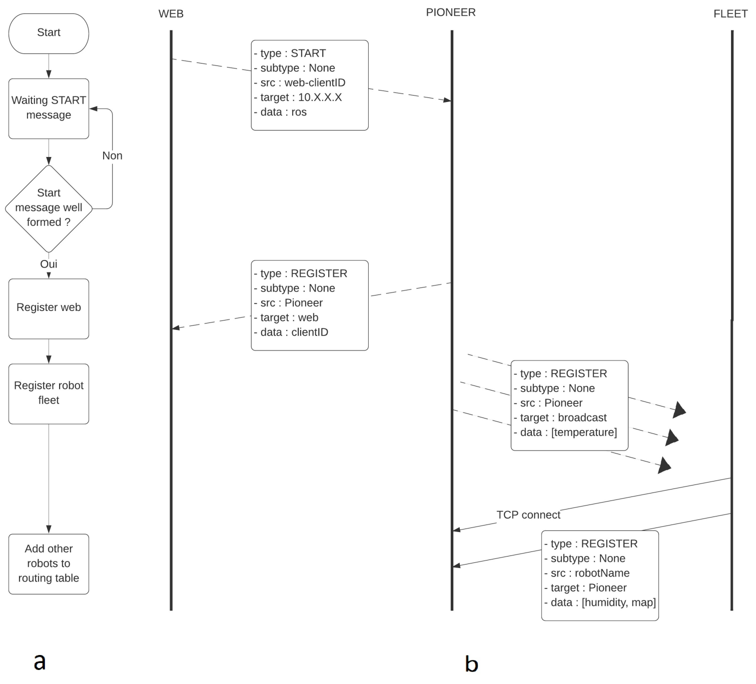


Figure 2. Procedure to register a new robot into the fleet.

3.2.2. Sensors Integration

This feature, which allows the automatic detection and recognition of the devices' connection, is based on *D-Bus* associated with *udev* software. In fact, *D-Bus*, a software bus, is an InterProcess Communication (*IPC*) and Remote Procedure Call (*RPC*) mechanism that allows communication between multiple computer programs concurrently running on the same machine. On the other hand, *udev* is a device manager for the Linux kernel. By using it, we can handle new sensors in a generic manner without limitation and with almost no effort when, usually, only a restricted set of devices is compatible with other middleware. Each time *udev* detects a new sensor, a signal is sent on the bus; thus, each actor listening in is noticed. TalkRoBots is one of these actor we are concerned about. Once the signal is received and the sensor is identified, our middleware broadcasts a message to register it on the Cloud and within all other middlewares in its routing table. Therefore, the sensor is available for all actors. Finally, at each time, a robot needs information from a sensor, and it sends a request through TalkRoBots layer 1 to the ConnectToBots layer, which handles this feature. Upon receiving the request, the ConnectToBots layer sends the required information.

Within each embedded computer, a sensor publishes its data on D-Bus using the same signal naming 'trbots.*sensorname*'. When receiving a request message for sensor data, ConnectToBots subscribes to the appropriate D-Bus signal to retrieve this information. Once it is completed, it unsubscribes to avoid receiving a continuous stream of unnecessary data. However, one exception for this behavior remains. When a modal window is opened on the Web interface to obtain access to the sensor's detailed information, in real-time, the request sent to ConnectToBots is launched in a dedicated thread to handle a continuous stream of data. Then, it is sent, every second, to the Web interface. Then, it unsubscribes and kills the thread when the modal window is closed. More details about the formatted TalkRoBots messages are provided in the following section.

3.2.3. Message Format

The format of the exchanged packet is depicted in the datagram of Figure 3. It contains the following six fields:

- Message *Type*:
  We defined three main different message types: *Command*, *Request* and *Information*. *Command* and *Request* types are handled differently according to the modes defined in Appendix A. In “direct” and “ros” modes, messages are sent directly to the robot without using the TalkRoBots package. Otherwise, in the same manner as for *Information* messages, they are sent field-by-field to the TalkRoBots package to be able to communicate with robots that cannot parse a received string message.
- Message *Subtype*:
  The *Subtype* field allows user to specify exactly what kind of *Information*, *Command* or *Request* it is sending. Although some *Subtype* fields are already predefined, such as “position” or “temperature”, for example, it may take whatever value. However, when using “direct” or “ros” mode, the *Subtype* needs to be defined accordingly in the configuration file presented previously in Figure A3. Otherwise, it is received as any other field by the TalkRoBots package.
- *Source*: It indicates the ID of the robot that sends the message.
- *Target*: It corresponds to the ID of the robot for which the message is addressed to.
- *Size*: It indicates the size of useful information.
- *Data*: It contains the data to be used.

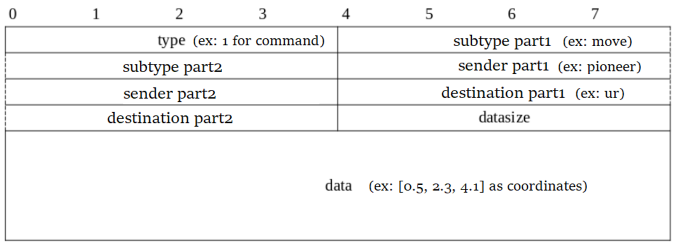


Figure 3. Message structure.

3.2.4. Message Transmission Modes

As presented previously, TalkRoBots uses three different modes, which will be detailed in this section.

1. **Interpreter Mode:** In the case where a robot is not able to receive a command directly, messages will be received field-by-field in order to prevent compatibility problems if a robotic language cannot manipulate strings. In this case, two functions have to be implemented on the robot, in the robotic language, to define how to handle receiving and sending messages, when communicating with the TalkRoBots middleware. Figure 4 presents an example of an exchange between two robots in the *Interpreter Mode*. In this figure, Robot1 needs to share its position with Robot3. To do so, it sends a message to its TalkRoBots middleware. The latter prepares an object *Message* with these data. Then, the message is serialized in JavaScript Object Notation (JSON) format and encrypted using Transport Layer Security (TLS). The security mechanisms will be described in the next Section 3.3. When the message is received on the middleware of Robot3, it is verified and deserialized. Finally, it is sent field-by-field to Robot3.

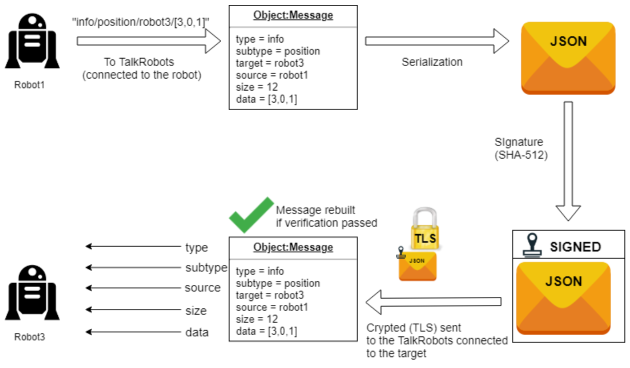


Figure 4. Information transmission (field by field).

2. **Direct Mode:** If a message targeting a robot is a *Command* or a *Request*, the configuration file defined in Figure A3 is used to translate the message into the appropriate command using robotic language. This command is transmitted to the robot, and it is executed immediately. However, if the command is not referenced in the configuration file or if the file itself does not exist, the message is transmitted using the default mode (i.e., the *Interpreter* mode). Figure 5 shows an example of a *Command* message sent by a user to a robot. The first steps are the same as those described for the previous Figure 4. Once the message reached the middleware of the Robot3, we differentiate two cases. The first one corresponds to the case where the *Subtype* is properly defined in the configuration file so it can be transmitted using the *Direct* mode. The second one, used by default, relies on the *Interpreter* mode.

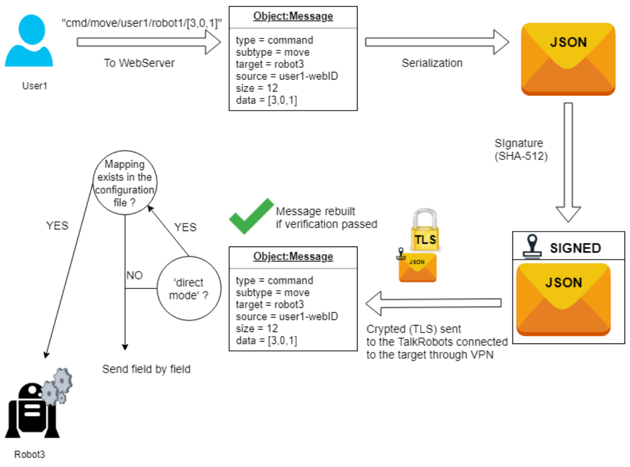


Figure 5. Command direct transmission.

3. **ROS Mode:** In the same manner as for the "direct mode", a command/request message is associated through the configuration file to a ROS topic. A Topic message type is retrieved automatically and the TalkRoBots message is transformed into a corresponding ROS message and published in the right topic.

3.3. Security

Security is one of the most important aspects that should be tackled when considering industrial systems, particularly in the context of Industry 4.0. TalkRoBots has been designed by considering some security mechanisms that are presented in this section.

The first security aspect is related to communication between the middleware and both the robot and devices. This issue is handled by using exclusively a directly wired Ethernet link between the robot and the computer on which TalkRoBots is installed. This choice is still coherent with security practices in the industry and prevents the remote control of TalkRoBots by a non-authorized person, such as in the case of wireless communications.

The second level of the security concerns *TCP* communications between different embedded computers, which uses an encrypted communication channel. The latter relies on *TLS* encryption and peer authentication facilities for network sockets using the *OpenSSL* library. As for the cipher suite, we use *ADH-AES256-SHA*. Note here that *ADH* stands for Anonymous Diffie Hellman key exchange, while *AES* (Advanced Encryption Standard) and *SHA* (Secure Hash Algorithm) are about stream cipher and message authentication. Unfortunately, even if this allows securing the *TCP* socket, it is still insufficient for covering the broadcast of *UDP* (User Datagram Protocol) messages, which are used to notify the integration of a new robot/device in the fleet. Indeed, the TLS version of *UDP*, called *Datagram Transport Layer Security* (DTLS) (TLS Python documentation, available at, https://tools.ietf.org/html/rfc6347, accessed on 15 February 2022), cannot be used in broadcast communication since it only applies on end-to-end communication, and broadcast is unidirectional. Thus, for this part, we use an *AES* cipher with a *SHA512* key to encrypt *UDP* messages. A summary of this secured exchange methodology is depicted on the flowchart of Figure 6.

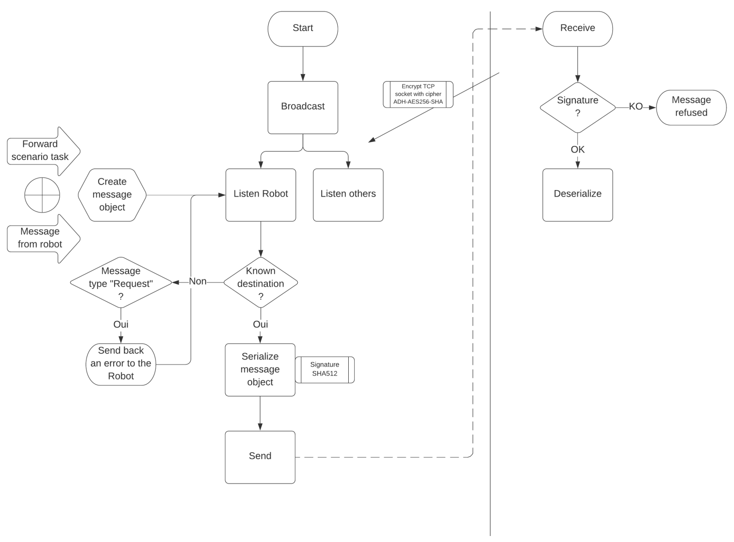


Figure 6. Exchange between two embedded computers.

Finally, all communications between the embedded computers and the Cloud server are passed through *OpenVPN*, which uses a *TLS* protocol with the *AES-256-CBC* cipher suite to secure information exchange. In this context, each authenticated user is affiliated to a private Virtual Private Network (VPN). By performing this, he is only able to communicate with the embedded computers that are preconfigured with the corect certificates. This allows the association of a private *VPN* with each fleet and/or company using TalkRoBots.

4. Supervision and Monitoring Cloud Application

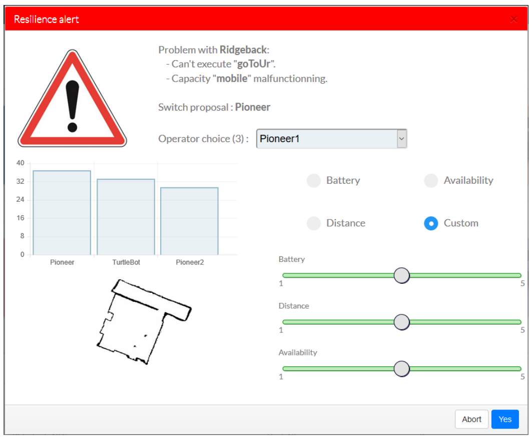
The integration of robotic systems in real world applications generally needs interaction with operators that program and supervise the evolution of the tasks and the possible faults. TalkRoBots proposes a Cloud application implementing several functionalities including monitoring and controlling. In addition to this functionality available in other Robotic Cloud applications [31], TalkRoBots can schedule and affect tasks and offers a resilience mechanism to keep performing operations without interruption when a robot is failing. The most significant implemented functionalities are described below:

- **Monitoring**: This functionality permits the operator to monitor the status of each robot and to obtain information about its embedded sensors and/or other data shared with its environment.
- **Controlling**: Users can also control their robot, for example, moving them, through the Cloud interface, by only using a keyboard or a mouse.
- **Tasks scheduling**: One of the most valuable functionality is the task scheduler. The latter is based on the association of capacities (e.g., mobile, fly and grab) and skills (moving forward, take off and grab left) to each robot. By skills, we mean basics or complex actions already individually implemented on the robot. Furthermore, once a sensor is plugged into the embedded PC's middleware, the robot connected to this middleware will acquire a new capacity that is offered by this sensor. For example, if we connect a temperature sensor, the robot will acquire the capacity of measuring the temperature. By performing this, this robot or another, could request the former and use this new available information to modify its behavior. In our experimental scenario, we pick up a box with an UR3 robot on the right or on the left side depending on the temperature's value. Using the task scheduler, we can define a scenario and organize these skills in whatever order we want. Once a scenario is defined, it will be sent to the first robot(s), which will participate in the scenario; then, it will be completely and autonomously shared between the different actors at each step.
- **Resilience**: Another valuable functionality is the capability of handling a failure that occurs on one or several robots during a scenario. In fact, when a failure is detected, a faulty message is automatically sent from the embedded computer to the Cloud in order to alert the operator. Once this message is received, an algorithm is executed in order to compute the best alternative solution, guaranteeing the execution of the scenario. In the Industry 4.0 context, it is important that everything is under the operator's control; thus, the calculated solution can be set so that it either requires its validation or is executed automatically. This algorithm is executed in different steps. Firstly, it requests the database to identify potential candidates. For this, we select the robots that have the same ability or abilities needed to perform the task that cannot be performed by the broken robot. Once the potential candidates are obtained, a ranking algorithm is executed according to several criteria that can be defined by the operator in order to choose the optimal alternative, as depicted in Figure 7.
  For this purpose, an Artificial Intelligence (AI) algorithm, denoted Analytic Hierarchy Process (AHP) [6], was adapted and implemented in order to suit our needs. The AHP algorithm can be summarized in the following steps:
  - Model the problem as a hierarchy containing the decision goal, the alternatives for reaching it and the criteria for evaluating the alternatives;
  - Establish priorities among the elements of the hierarchy by making a series of judgments based on pairwise comparisons of the elements;
  - Synthesize these judgments to yield a set of overall priorities for the hierarchy;
  - Check the consistency of the judgments;
  - Come to a final decision based on the results of the previous steps.

To guarantee the resilience of the system, TalkRoBots should reconfigure the scenario and the control tasks using only the available equipment when a fault occurs. To do so, the goal is to find a robot with the capacities that allow it to take over the faulty one. To define these alternatives, meaning the robots that are able to perform the tasks that were assigned to the faulty one, a database query is used to identify the list of all the robots with the needed capacities. Then, the possible alternatives are ranked by optimizing a cost function related to some predefined criteria. The example shown in Figure 8 describes an example in our scenario presented in Section 6; three criteria were selected in the case of faults that occur on mobile robots (i.e., state of the battery, position of each alternative robot and its availability) (an open source implementation code of AHP could be found here: https://github.com/avataru/PHP-AHP, accessed on 15 February 2022). Thus, each time we receive a message from a robot announcing a problem, we first request alternatives, then we define priorities between criteria according to the parameters defined in the Graphical User Interface (GUI), as shown in Figure 7. The outcome is redefined every time the user changes the parameters by automatically sending a request from the Cloud to the concerned robots. In this scenario, we have two criteria that need to be obtained by requesting robots: the battery level and the position. The third one, which is the availability, is defined as a coefficient by being default setted to 100, which is lowered if the alternative robot performs a task later in the scenario (or in another one). Finally, the GUI displays different information for the user:

- The list of alternatives, ranked by score according to the AHP algorithm;
- A graphical representation of ranking scores;
- The parameter values currently set, which can be modified.

Taking all this into consideration, the operator can choose a robot among all alternatives to replace the faulty one or he can simply abort the scenario.


Resilience alert

Problem with **Ridgeback**:
- Can't execute "**goToUr**".
- Capacity "**mobile**" malfunctionning.

Switch proposal : **Pioneer**

Operator choice (3) : Pioneer1

Battery Availability Distance [✓ Custom]

Battery: 1 – 5
Distance: 1 – 5
Availability: 1 – 5

Figure 7. Resilience Graphical User Interface (GUI).

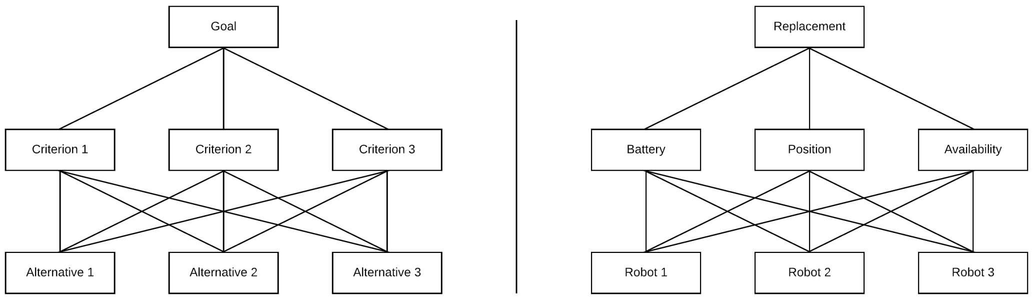


Figure 8. Analytic Hierarchy Process.

5. Performance Analysis

This section aims to study the impact of the different variations that could occur when performing different scenarios, i.e how the messages are transmitted to the robot, the impact of the number of tasks and actors on the bandwith comsumption and the latency induced by using encrypted channels.

5.1. Analytical Results

According to TalkRoBots' architecture, it is obvious that the "interpreted mode" (cf. Section 1) takes a longer period of time than the direct mode when transmitting messages. This is due to the idle time added between each sent field to overcome the incapacity of robots to manipulate human-readable data. Thus, since each sent message using the "interpreter mode" is composed of five fields, this induces a delay about five times longer than the other modes that have a size that is lower than the TCP's payload length. We recall here also that both the "direct mode" and "ros mode" (cf. Section 3) involve configuration file access, described in Figure A3, in order to obtain proper syntax. However, this step is still less time consuming.

To facilitate the reading of equations, Table 2 lists various symbols used in the next equations.

Table 2. List of symbols.

Symbols	Significations
d_{boot}	the number of the exchanged messages for initializing the middleware
n	the number of robots
$d_{identification}$	the number of the exchanged messages during the fleet creation
$d_{devices}$	the number of the messages related to the plugged devices
d_{start}	the number of the messages needed for the initialization step
s	the number of IIoT devices
x_s	the number of messages sent per second by each device
Δ_t	the number of elapsed seconds in the experimentation
N_{task}	the total number of tasks in the scenario
N_d	the number of tasks transmitted using direct or ROS mode
N_i	the number of tasks transmitted using interpreted mode
$d_{scenario}$	the number of exchanged messages to execute a scenario
d_{task}	the number of messages sent to handle the tasks
$d_{forward}$	the number of messages needed to forward the tasks between the actors
d_{sensor}	the number of messages sent to handle a request from a sensor
R_l	the number of locally sent requests to a sensor
R_e	the number of remotely sent requests to a sensor
d_{fail}	the number of the exchanged messages to handle a faulty behavior
N_f	the number of faulty robots

In the remainder of this paper, we will provide an estimation of the number of exchanged messages during a predefined scenario by splitting it into different phases. At the beginning, we have to take into account the number of messages sent to launch different robots, which will also send messages in their turn to each others in order to form a fleet. As we can see in Figure 2, there is a first message sent by the robotic Cloud to start the middleware. Then, the middleware answers to notify that it has already started. The robot should also send a message to let the middleware know that it is still waiting for a new instruction. An extra message is counted to send the scenario to the first actor. Thus, the number of messages needed for the initialization of TalkRoBbots is as follows:

$$d_{boot} = 3n + 1 \tag{1}$$

where d_{boot} is the number of the exchanged messages for initializing middleware, and n is the number of robots.

Then, when a robot joins the fleet, it broadcasts a message to all other connected members. Then, each robot sends an acknowledgement with its identity. This results in the following:

$$d_{identification} = n + \frac{(n-1)n}{2} = \frac{n(n+1)}{2} \tag{2}$$

where $d_{identification}$ is the number of the exchanged messages during the fleet creation.

We also consider the exchanged messages when a new external IIoT device is plugged. Each time a new IIoT device is added, a message is sent to the robotic Cloud in order to create a new monitoring line on the GUI that will be regularly updated. A message is also sent to inform all other connected robots. Thus, we can establish a first estimation of the overhead related to the sensors:

$$d_{devices} = 2s + \Delta_t \sum_{i=1}^{s} x_i \tag{3}$$

where $d_{devices}$ is the number of the messages related to the plugged devices, s is the number of IIoT devices, x_s is the number of messages sent per second by each device (by default, it is set to 1 for each device) and Δ_t is the number of elapsed seconds.

To sum up the number of messages needed for the initialization process is described as follows.

$$\begin{aligned} d_{start} &= d_{boot} + d_{identification} + d_{devices} \\ &= \frac{1}{2}\left[n(n+7) + 4s + 2 + 2\Delta_t \sum_{i=1}^{s} x_i\right] \end{aligned} \tag{4}$$

Now, let us deal with the number of messages related to the tasks that should be executed during a scenario. These tasks could be executed using the direct/ROS mode or the interpreted one, and it can be calculated as follows:

$$N_{task} = N_d + N_i \tag{5}$$

where N_{task} is the total number of tasks in the scenario, N_d the number of tasks transmitted using direct or ROS mode and N_i the ones using the interpreted mode.

Moreover, the number of exchanged messages to execute a scenario ($d_{scenario}$) can be separated into two parts. The first one, called d_{task}, is related to the execution of the tasks themselves. The second one, named $d_{forward}$ is related to the tasks' forwarding from a robot to its successor. Thus, one can obtain the following:

$$d_{scenario} = d_{task} + d_{forward} \tag{6}$$

where d_{task} is the number of messages sent to handle the tasks and $d_{forward}$ is the number of messages needed to forward the tasks between the actors.

For the accurate estimation of $d_{scenario}$, the remainder of this section supposes that the fleet mixes robots using the direct mode and others using ROS and interpreted ones. In fact, at each time, a task is performed by a robot, and several messages are exchanged. The first message is sent from the robot to the middleware to notify that it is ready to handle a task. Then, the information related to the task that should be executed is sent from the middleware to the robot. More particularly, middleware sends five messages when the robot uses the interpreted mode. Otherwise, it sends only one message. Then, when the task is achieved, an acknowledgement message is sent from the robot to the middleware. Finally, the latter forwards an acknowledgment message to the GUI in order to update HMI. This scenario applies also when several successive tasks should be performed by the same robot. Hence, the number of exchanged messages to execute all tasks is as follows.

$$d_{task} = 4N_d + 8N_i \tag{7}$$

Finally, one more message is necessary when the next task in the scenario is not performed by the same actor than the current one. To illustrate this, let us consider a scenario including twelve tasks.

As shown in Figure 9, counting the forward messages involves three variables: (1) the total number of tasks in the scenario, already defined as N_{task}; (2) the number of blocks k, a block being defined as the set of tasks performed by the same actor; and (3) the number of tasks included in each block j, m_j. In our example, we have the following: $N_{task} = 12$,

$k = 3$ and $\sum_{j=1}^{k} m_j = 7$. From this scenario and without a loss of generality, one can obtain the following:

$$\begin{aligned} d_{forward} &= N_{task} - \sum_{j=1}^{k} m_j + k - 1 \\ &= N_d + N_i - \sum_{j=1}^{k} m_j + k - 1 \end{aligned} \tag{8}$$

where k is the cardinality of task blocks that have to be executed by the same actor successively, and m_j ($j \in \{1, 2, \ldots, k\}$) is the number of tasks in the set of task blocks.

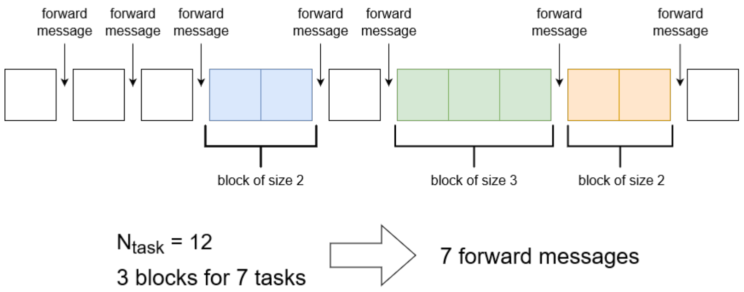


Figure 9. Illustration of the counting of the forward messages.

Applying this on our example, we have, as expected, a result of seven forward messages.

Finally, by using Equations (6)–(8), the number of exchanged messages to execute a scenario is as follows.

$$\begin{aligned} d_{scenario} &= 4N_d + 8N_i + N_d + N_i - \sum_{j=1}^{k} m_j + k - 1 \\ &= 5N_d + 9N_i - \sum_{j=1}^{k} m_j + k - 1 \end{aligned} \tag{9}$$

Another type of messages that should be taken into account concerns data exchange with the IIoT devices. In fact, two situations are possible. The first one is related to the case where the robot requesting the data is associated with the device. In this case, only two messages are needed (i.e., the request from the robot and the response from the device). However, the second one occurs when IIoT devices are plugged with a remote robot. In this case, two additional messages are needed (i.e., forwarded request and forwarded response from the remote robot). This leads to the following.

$$d_{sensor} = 2R_l + 4R_e \tag{10}$$

where R_l includes locally sent requests, and R_e includes externally sent requests.

Then, we have to take into consideration the fault tolerance mechanism. When a robot becomes faulty, a message is sent to the Cloud. Thus, the operator can accept an alternative solution among those proposed. The fixed scenario is then sent back to the alternative actors. This results in the following:

$$d_{fail} = 2N_f \tag{11}$$

where d_{fail} is the number of the exchanged messages to handle a faulty behavior, and N_f is the number of faulty robots ($N_f \leq \mathrm{n} - 1$).

Finally, we obtain the following equation.

$$\begin{aligned} d &= d_{start} + d_{scenario} + d_{sensor} + d_{fail} \\ &= \frac{n(n+7)}{2} + k + 2(s + R_l + 2R_e + N_f) \\ &+ 5N_d + 9N_i + \Delta_t \sum_{i=1}^{s} x_i - \sum_{j=1}^{k} m_j \end{aligned} \tag{12}$$

5.2. *Simulations*

This section presents the results of the simulations that were conducted in order to validate the analytical study and to evaluate the scalability of our TalkRoBots middleware.

5.2.1. Simulation Settings

In order to validate the theoretical analysis previously stated in Section 5.1, a simulator, for which the HMI is depicted in Figure 10, has been developed.

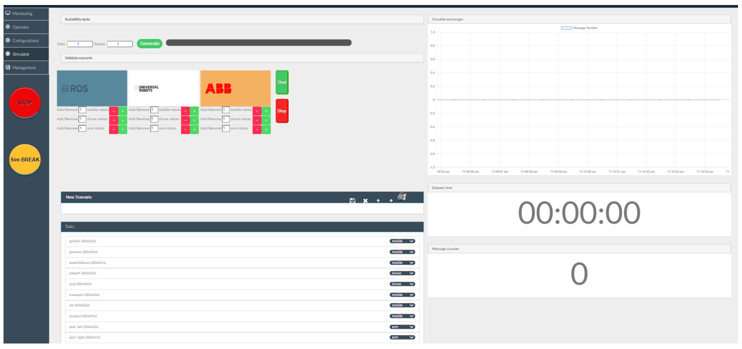


Figure 10. Simulator interface.

The user can select the number of tasks and the number of robots involved in a fleet. Then, a virtual fleet and a virtual scenario are created. The latter assigns the tasks cyclically to the robots, when their number is less than the number of tasks. Otherwise, it assigns the tasks in sequential order. Finally, all tasks are sequentially executed in the virtual scenario, the requests to external devices are not handled and each task is supposed to have an arbitrarily chosen duration of three seconds.

Several tests have been carried out for different values of number of tasks and robots to measure the impact of each variable on the number of messages and the bandwidth's consumption.

5.2.2. Simulation Results

As we can observe in Figure 11, the number of messages is more impacted by the number of robots than by the number of tasks. It is coherent with our theoretical analysis since the most costly part in terms of the number of messages is d_{start} given by Equation (4).

Figure 12 shows that the exchanged data size increases with the number of tasks. This can be explained by the fact that every time a robot has to forward the remaining part of the scenario to the next actor, it has to send all the remaining tasks. Thus, if the number of tasks is important, the size of exchanged data is proportionally bigger. Logically, we can also observe that if there is only one robot, the size of exchanged data stays low since there will not be any tasks that have to be forwarded. Finally, we can observe that even for a scenario with fifty robots and one hundred tasks, only 1.4 MB has been exchanged.

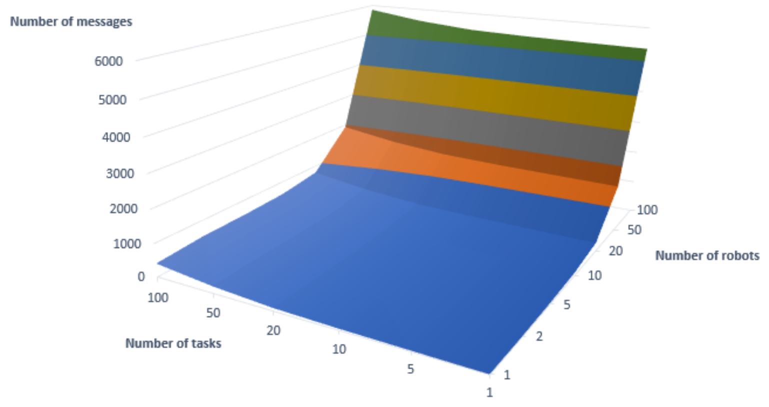


Figure 11. Number of messages according to the number of robots and the number of tasks.

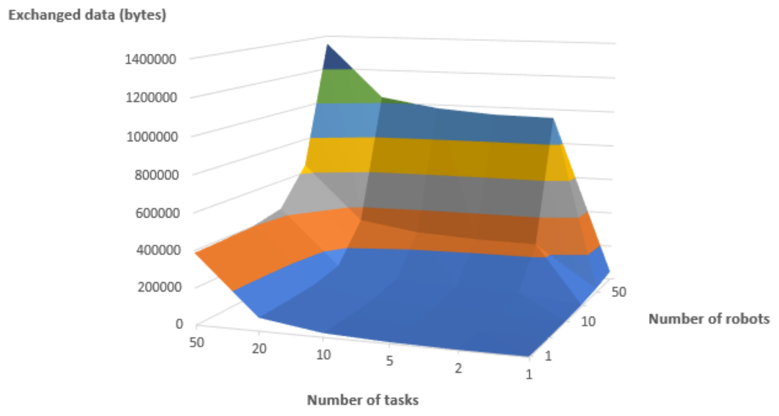


Figure 12. Exchanged data amount according to the number of robots and the number of tasks.

Moreover, Figure 13 shows the distribution of the exchanged messages with a different number of robots. As expected, the largest part of the messages is exchanged during the creation of the fleet. Then, there are some intermittent exchanges when the scenario is forwarded. Therefore, only the number of messages during fleet creation will be impacted by the number of robots.

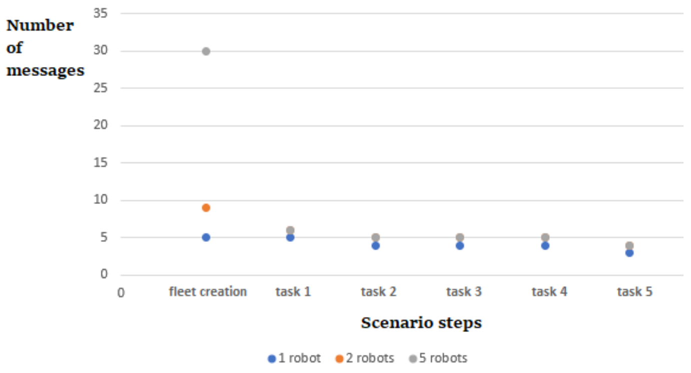


Figure 13. Message distribution overtime.

6. Experimentation

This section discusses an implementation of a real manufacturing scenario involving the following: a Ridgeback Automated Ground Vehicle (AGV) coupled to a Sawyear robot (both of them are under ROS), a Universal Robot UR3, a fleet of mobile Pioneer robots, a Parrot drone, a conveyor associated to a Programmable Logic Controller (PLC) and an external IIoT device (temperature sensor). A graphic illustration of the equipment is shown in Figure 14.

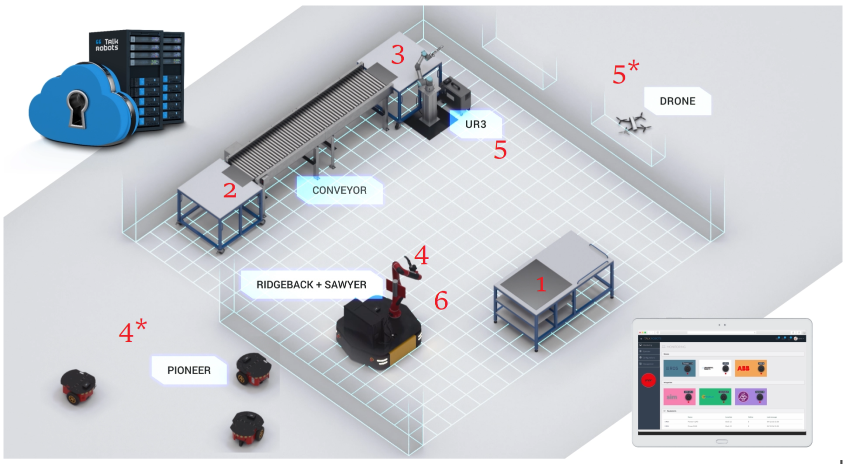


Figure 14. Prototype.

6.1. Scenario Presentation

To demonstrate the efficiency of our middleware and to test its functionalities, we designed a prototype integrating the previously presented equipment. The middleware was installed in several embedded computers, namely Raspberrys Pi Model 3 B+, which run Ubuntu Mate. Each robot is linked through Ethernet with its Raspberry and all robots and embedded computers (middleware) are connected to the same network.

The operator uses the Cloud application to send the scenario depicted in Figure 15 to the first involved actor.

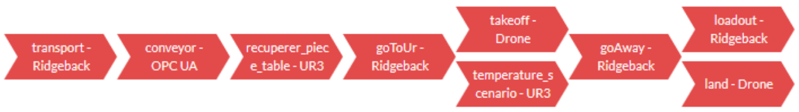


Figure 15. Demonstration scenario on GUI.

The details of the different scenario's tasks are as follows:

1. The first task involves the robot, which receives the scenario from the Cloud. This task consists in moving the Ridgeback to the first Table (1) to catch up a box and to bring it to the conveyor's entry (2). Once the task is completed, the robot sends an acknowledgment to the middleware, which will send the following tasks to the next actor(s). In our case, it is the conveyor. It will also forward the acknowledgment to the Cloud. Therefore, the operator can have feedback on the progress of the scenario.
2. The conveyor is launched and moves the box from Table (2) to Table (3); then, it sends the next task to UR3.
3. UR3 grasps the box from Table (3) and places it on the ground near itself. Afterwards, it sends the next task to Ridgeback.
4. Ridgeback proceeds towards UR3. However, a simulated fault occurs on Ridgeback, which stops. An alert is sent to the operator through GUI, explaining that a problem has been observed with the ability of mobility for Ridgeback. The developed resilience mechanism suggests what other robots could replace it. Different choices are offered to the operator on HMI. These choices are ordered by priority according to different parameters: battery, position and availability (in this scenario, one of the Pioneer robots will be selected as the best alternative).
5. Once this alternative is validated, the Pioneer moves to UR3. Then, it sends the next tasks to UR3 and the Drone. The drone and the arm receive the remaining tasks of the

scenario and execute the first ones they are concerned with. The drone takes off and the arm, through the middleware, requests an external temperature sensor and picks a box on the left or the right side according to the response (above or below 20 °C). Once they are completed, they send an acknowledgement. Then, the remaining tasks are sent once again to the next actor (Pioneer).

6. The Pioneer moves back to the Ridgeback and the latter can take the box and place it over its platform before sending the last tasks (landing) to the Drone and the Ridgeback (loading out).

For more details, the reader is referred to a video (SATTNord TalkRobots https://www.youtube.com/watch?v=lneJ5o0sODY, accessed on 15 February 2022), which is available online.

6.2. Experimentation Results

In our demonstration, the scenario includes five robots, one sensor and nine tasks. The results in Figure 16 show that 284 messages have been exchanged during the experimentation. Moreover, considering the theoretical analysis in Equation (12) and the considered scenario, one can estimate an upper and lower bound of the number messages according to the possible faulty behavior and the involved actors (robots, sensors, etc.). In fact, in the best case (251 messages from Equation (12)), there is no faulty behavior, the sensor request is locale and all tasks are performed by the same actor and sent using the ROS or direct modes. In the worst case (297 messages from Equation (12)), the sensor request is external: One robot will have faulty behavior (since we only have one spare here) and all tasks are sequentially performed by different actors and sent field by field. In every case, we consider that the sensor sends an update every seconds and that the total duration of the scenario is about three minutes. Therefore, the results are still coherent with the analytical study.

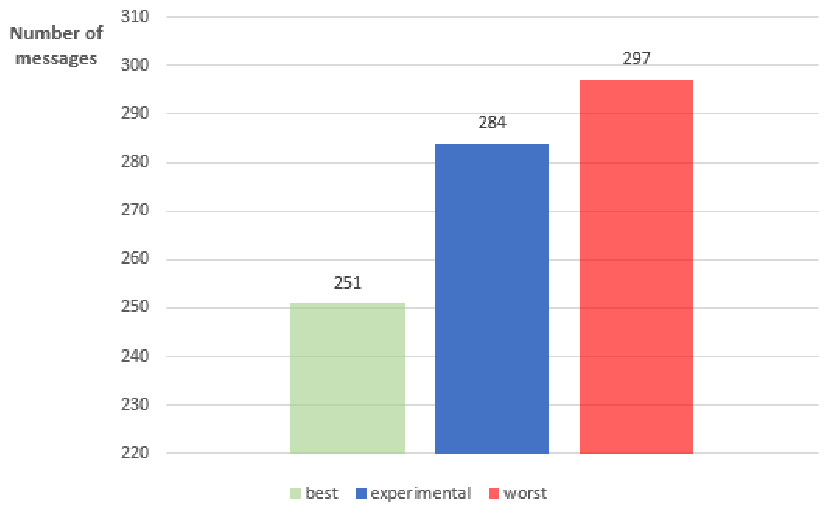


Figure 16. Number of messages exchanged during a scenario.

In our scenario, the data field is almost always empty except in the forward messages and the acknowledgments sent to the Cloud. Therefore, we only have seven messages with a significant size superior to 32 bytes (an acknowledgment is about 52 bytes). Actually, the first forward message has an initial size of 769 bytes since it must transmit the eight tasks left to the second actor. The experimental results show that 15.1 kbytes has been exchanged in approximately 3 min, producing a bandwidth of 83 B.s^{-1}, which is very low in the industrial context. This shows that the overhead induced by TalkRoBots is very limited.

Another parameter that has been evaluated concerns the impact of the security mechanism on the transmission's delay. Before presenting the results for this indicator, let us show that the security involves two processes: a signature and a send/receive TLS encrypted tunnel. Contrary to the signature, TLS encryption is only performed once on the communication channel when it is created. Thus, this operation is not performed again every time a message has to be sent. Thus, to evaluate the impact of security, we measure the transmission delay between sending and receiving a message when we have to sign it on an encrypted canal and when it is not signed.

As shown in Figure 17, adding a signature and using encrypted communications only slightly increases the delay between the time a message is sent and when it is received. To produce these results, we start measuring time before signing a serialized message; then, we send it. We stopped measuring only when the message has been received and validated (signature matched). All exchanged messages had a size of 32 or 48 bytes (depending on data type). The analysis of this figure shows that processing a non=signed message takes at least 0.8 ms and a signed one takes no more than 1.4 ms. Thus, the maximum observed delay is 0.6 ms. This could be largely accepted compared to all benefits of such security mechanisms. Note here that this delay is obtained by using cheap embedded computers and, thus, can be significantly reduced by using more powerful ones.

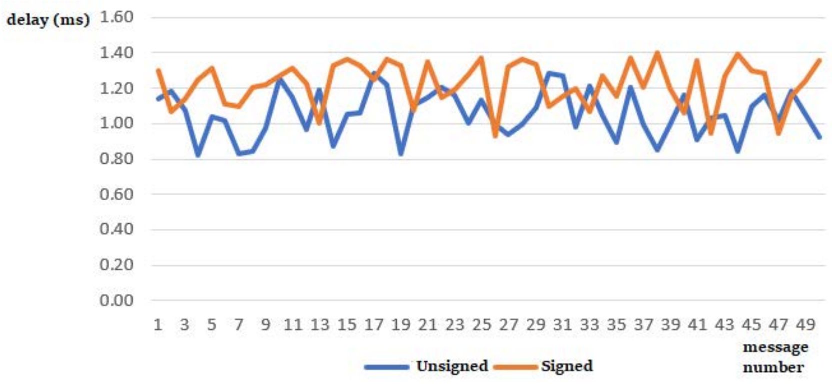


Figure 17. Transmission delay with security enabled and disabled.

Finally, some experimentation has been confirmed in order to study the impact of the transmission mode on transmission delay, while including the security mechanism. The delay between receiving an external message by the middleware and forwarding it to the robot may change according to the message transmission mode. As we can see in Figure 18, this delay is more important when we transmit field by field in the interpreter mode, as previously stated. Indeed, in order to send multiple fields using TCP and to prevent some fields being concatenated into the same buffer, we add a slight delay between each sent field. As expected, it takes around 10 ms to send a message using the interpreter mode, which is five times more than in the direct mode, which takes only about 1.8 ms.

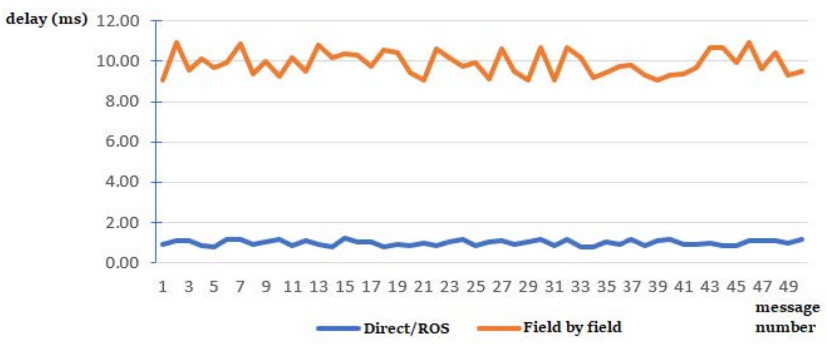


Figure 18. Transmission delays according to the transmission mode.

6.3. Results Discussions

As presented by the previous results, one can observed that all results are coherent with the analytical study. This shows that TalkRoBots does not overload the network with many messages. In addition, these messages are not heavy, which resulted in low bandwidth consumption (about 83 $B.s^{-1}$). The second remark to mention is that the secured communications using encrypted exchanges with certificates, which are very important in an industrial context, do not have a big impact in the exchanging process. On average, it adds 0.6 ms of delay when using certificates, which is largely acceptable. Finally, the used mode has an important impact in communication delay. Since it takes five time more to send a message field by field in the Interpreter mode than by sending the message directly in ROS or Direct modes. Since the mode is directly linked to the robotic operating system installed on the robot, we have very limited control on this parameter.

7. Conclusions

In this paper, we proposed a middleware, which aims to assist an operator to manage heterogeneous fleets of robots. Associated with Cloud services, it allows the following: (1) execute different scenarios with several robots without needing to reprogram them and (2) monitor the achievement of tasks and the robots' status. Moreover, the middleware, implemented in Python, is not intrusive since the robots are still programmed in their own robotic language. The nature of the proposed middleware architecture makes the integration of new robotic resources and IIoT devices very simple since it only requires their compatibility with the TCP/IP protocol. Some available configuration files could be edited for more flexibility on the network parameters and for more functionalities, enabling direct communication between the middleware and the robot's controller. Communication is performed both on the local network, between robots, using Ethernet or Wi-Fi and over the Internet to communicate with the Cloud application and the remote sites. In both cases, communication is secured using TLS encryption on SHA512 signed-messages. When communicating over the Internet, all communications use a private and secured VPN. Contrary to most robotic middleware, TalkRoBots do not offer libraries but do a different method for different robots, with different languages, to communicate using their own language. Our middleware is used for communication and fleet management. In this context, we designed a specific message format to transmit the different information between the different middleware and the robotic Cloud. Another important aspect of fleet management is resilience. Using decision-making algorithms, TalkRoBots is able to propose alternatives when a robot is faulty. By using a prototype, we have been able to confront theoretical performances with experimental ones and show the low impact of the encryption protocols on message transmission delay and network bandwidth consumption. However, even if TalkRoBots aims to facilitate the integration of new robots, there is still a need for human interaction. In fact, there is a configuration step that needs to be handled manually. Moreover, the definition of a scenario can, until now, only be performed by an operator, who must select the tasks and the robots that will execute them. This limits its application to a statically defined scenarios and it is not possible to use it for dynamic ones such as in crisis management. These limitations will be confronted with in future works.

Author Contributions: Conceptualization, M.A., N.M. and F.V.; Software, D.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by SATT Nord grant number TalkRoBots Project.

Data Availability Statement: Not Applicable, the study does not report any data.

Conflicts of Interest: The authors declare no conflicts of interest.

Acronyms

The following acronyms are used in this manuscript:

ADH	Anonymous Diffie Hellman
AES	Advanced Encryption Standard
AGVs	Autonomous Guided Vehicles
AHP	Analytic Hierarchy Process
AI	Artificial Intelligence
ARIA	Adaptive Robot-Mediated Intervention Architecture
DSSs	Decision Support Systems
DTLS	Datagram Transport Layer Security
FMS	Flexible Manufacturing Systems
GUI	Graphical User Interface
ICT	Information and Communication Technologies
IIoT	Industrial Internet of Things
IP	Internet Protocol
IPC	InterProcess Communication
JSON	JavaScript Object Notation
MIA	Middleware for Intelligent Automation
PLC	Programmable Logic Controller
RPC	Remote Procedure Call
ROS	Robot Operating System
SHA	Secure Hash Algorithm
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtual Private Network

Appendix A. Configuration Files

These configurations files are intended to be minimalist and relies only on three JavaScript Object Notation (JSON) files. The first one depicted in Figure A1 defines the profile related to the used robot before its integration in the fleet. More particularly, TalkRoBots specifies three modes allowing the integration of most commercialized robots without any additional effort. The first mode ("ros") corresponds to the robots using ROS, and it is based on the integration of some functions that exploit the publish/subscribe concept usually used in ROS (refer to Section 3.2.4). The second mode ("direct") is used for robots, such as Universal Robots, which are able to parse the commands sent by the middleware using their native robotic language to a specific port directly on the robot's controller and then execute them. Finally, the last mode ("interpreter") is used for all other robots that cannot satisfy the previous conditions.

```
{
    "ros": "ros",
    "ur": "direct"
    "abb": "interpreter",
}
```

Figure A1. Profile configuration file.

The second configuration file defines all network parameters. In performing this, TalkRoBots can automatically obtain the IP address to configure its communication sockets without needing to modify it manually, when it dynamically changes, as long as the network interface used is the same. An example of this configuration file is presented in Figure A2. In this example, we first define on which port the external messages are received when sent by the web application. In our case, it is set on port 50,000, but it can be changed by the user as long as it matches with the one set on the web application's server. The middleware handles the messages from two other sources in addition to the web application: (1) the other robots' middleware and (2) the robot, which it is connected to. Thus, we define two other ports. The "TCP_PORT_EXT" value is the one on which the middleware expects to receive messages from the other robots, while the "TCP_PORT" value defines the port number on which the middleware expects to receive messages from its connected robot. The last two values, "HOST_INT" and "VPN_INT", inform the middleware which network interfaces must be used to communicate, respectively, on the local network (essentially used to communicate inside the fleet) and on the VPN (used to communicate with the web application or fleets that are not on the same network).

```
{
"_comment" : "Listening port for external messages (web or GUI)",
"UDP_PORT" : 50000,
"_comment" : "Listening port for external messages (robots)",
"TCP_PORT_EXT" : 45000,
"_comment" : "Listening port for internal messages (from handled robot)",
"TCP_PORT" : 40000,
"_comment" : {
"Interface used for local network communication": "",
"(when VPN is shutdown)": "",
"On Windows : usually 'Ethernet' or 'Wi-Fi'": "",
"On Linux : usually 'eth0', 'enp3s0' (ethernet) or 'wlan0' (Wifi)": ""
},
"HOST_INT" : "Wi-Fi",
"_comment" : {
"Interface used to communicate through VPN" :"",
"(with web server for example)":"",
"On Windows (OpenVPN) usually 'Ethernet 2'":"",
"On Linux : usually 'tun0'":""
},
"VPN_INT" : "tun0"
}
```

Figure A2. Network configuration file.

The third configuration file is needed only for the translation of a message between robots using either the "ros" mode or the "direct" mode. Figure A3 shows an example of such a file. In this example, we define a specific port to use with a Universal Robot, which allows sending the commands written in robotic language. In addition, the specified value (i.e., 30002 here) allows the middleware to know on which port it has to send the messages to. Then, under "commands" and "requests" keys, we specify which topic or command to use according to the robot vendor and the message subtype (for more details about message subtype, see Section 3.2.3). In this case, if the middleware receives a command message with subtype "move", which should be transmitted to the connected robot, it should either publish the command on the "/cmd_vel" topic in cases where the robot is using ROS or send a "movej" command on port 30002 if it concerns a robot manufactured by the vendor Universal Robot.

```
{
        "ports":
        {
                "ur": 30002
        },
        "commands":
        {
                "move" :
                {
                        "ros" :
                        {
                                "/cmd_vel" : 0
                        },
                        "ur"  :
                        {
                                "movej(params)\n" : "[array, float, float]"
                        }
                },
                "takeoff" :
                {
                        "ros" :
                        {
                                "/bebop/takeoff" : 0
                        }
                },
        },
        "requests":
        {
                "modal" :
                {
                        "ros" :
                        {
                                "/pose" : 0
                        }
                },
                "position" :
                {
                        "ros" :
                        {
                                "/pose" : 0
                        }
                },
        }
}
```

Figure A3. Commands configuration file.

References

1. Jaloudi, S. Communication Protocols of an Industrial Internet of Things Environment: A Comparative Study. *Future Internet* **2019**, *11*, 66. [CrossRef]
2. Nakagawa, E.Y.; Antonino, P.O.; Schnicke, F.; Capilla, R.; Kuhn, T.; Liggesmeyer, P. Industry 4.0 reference architectures: State of the art and future trends. *Comput. Ind. Eng.* **2021**, *156*, 107241. [CrossRef]
3. Farkhana, M.; Abdul Hanan, A. Mobility in mobile ad-hoc network testbed using robot: Technical and critical review. *Robot. Auton. Syst.* **2018**, *108*, 153–178. [CrossRef]
4. Wang, J.; Lim, M.K.; Wang, C.; Tseng, M.L. The evolution of the Internet of Things (IoT) over the past 20 years. *Comput. Ind. Eng.* **2021**, *155*, 107174. [CrossRef]
5. Ayaida, M.; Messai, N.; Valentin, F.; Marcheras, D.; Afilal, L. Robot Interconnection Method. WO/2019/162595. 29 August 2019. Available online: https://patentscope2.wipo.int/search/en/detail.jsf?docId=WO2019162595 (accessed on 15 February 2022).
6. Forman, E.H.; Gass, S.I. The Analytic Hierarchy Process—An Exposition. *Oper. Res.* **2001**, *49*, 469–486. [CrossRef]
7. Marcheras, D.; Ayaida, M.; Messai, N.; Valentin, F. A new middleware for managing heterogeneous robot in ubiquitous environments. In Proceedings of the 2020 8th International Conference on Wireless Networks and Mobile Communications (WINCOM), Reims, France, 27–29 October 2020; pp. 1–5. [CrossRef]
8. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A. ROS: An Open-Source Robot Operating System. 2009, Volume 3. Available online: http://lars.mec.ua.pt/public/LAR%20Projects/BinPicking/2016_RodrigoSalgueiro/LIB/ROS/icraoss09-ROS.pdf (accessed on 15 February 2022).
9. Joyeux, S.; Albiez, J. Robot development: From components to systems. In Proceedings of the Control Architecture of Robots, Grenoble, France, 24–25 May 2011; p. 115.
10. Osentoski, S.; Jay, G.; Crick, C.; Pitzer, B.; DuHadway, C.; Jenkins, O. Robots as web services: Reproducible experimentation and application development using rosjs. In Proceedings of the 2011 IEEE International Conference on Robotics & Automation, Shanghai, China, 9–13 May 2011.
11. Roalter, L.; Kranz, M.; Moller, A. A middleware for intelligent environments and the internet of things. *Ubiquitous Intell. Comput.* **2010**, *6406*, 267–281.

12. Beetz, M.; Mosenlechner, L.; Tenorth, M. CRAM—A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, Taiwan, 18–22 October 2010; pp. 1012–1017.
13. Collett, T.; Macdonald, B.; Gerkey, B. Player 2.0: Toward a Practical Robot Programming Framework. In Proceedings of the 2005 Australasian Conference on Robotics and Automation, ACRA 2005, 2008. Available online: http://users.isr.ist.utl.pt/~jseq/ResearchAtelier/papers/collet.pdf (accessed on 15 February 2022).
14. Vaughan, R.T. Massively multi-robot simulations in Stage. *Swarm Intell.* **2008**, *2*, 189–208. [CrossRef]
15. Koenig, N.; Howard, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In Proceedings of the IEEE/RSj International Conference on Intelligent Robots and Systems, Sendai, Japan, 28 September–2 October 2004; pp. 2149–2154.
16. Agüero, C.E.; Koenig, N.; Chen, I.; Boyer, H.; Peters, S.; Hsu, J.; Gerkey, B.; Paepcke, S.; Rivero, J.L.; Manzo, J.; et al. Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response. *IEEE Trans. Autom. Sci. Eng.* **2015**, *12*, 494–506. [CrossRef]
17. Whitbrook, A. *Programming Mobile Robots with Aria and Player: A Guide to C++ Object-Oriented Control*; Springer: London, UK, 2010.
18. Magnenat, S.; Rétornaz, P.; Bonani, M.; Longchamp, V.; Mondada, F. ASEBA: A Modular Architecture for Event-Based Control of Complex Robots. *IEEE/ASME Trans. Mechatron.* **2011**, *16*, 321–329. [CrossRef]
19. Magnenat, S.; Mondada, F. ASEBA Meets D-Bus: From the Depths of a Low-Level Event-Based Architecture into the Middleware Realm. In Proceedings of the IEEE TC-Soft Workshop on Event-Based Systems in Robotics (EBS-RO), St. Louis, MO, USA, 15 October 2009.
20. Montemerlo, M.; Roy, N.; Thrun, S. Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) Toolkit. In Proceedings of the Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453), Las Vegas, NV, USA, 27–31 October 2003; Volume 3, pp. 2436–2441.
21. Calisi, D.; Censi, A.; Iocchi, L.; Nardi, D. OpenRDK: A framework for rapid and concurrent software prototyping. In Proceedings of the International Workshop on System and Concurrent Engineering for Space Applications (SECESA), Nice, France, 22–26 September 2008.
22. Calisi, D.; Censi, A.; Iocchi, L.; Nardi, D. OpenRDK: A modular framework for robotic software development. In Proceedings of the 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, Nice, France, 22–26 September 2008; pp. 1872–1877.
23. Makarenko, A.; Brooks, A. Orca: Components for robotics. In Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06), 2006. Available online: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.7562&rep=rep1&type=pdf (accessed on 15 February 2022).
24. Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S.; Orebäck, A. Orca: A Component Model and Repository. In *Software Engineering for Experimental Robotics*; Brugali, D., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 231–251.
25. Makarenko, A.; Brooks, A.; Kaupp, T. On the Benefits of Making Robotic Software Frameworks Thin. In Proceedings of the International Conference on Intelligent Robots and Systems, San Diego, CA, USA, 29 October–2 November 2007.
26. Bruyninckx, H. Open robot control software: The OROCOS project. In Proceedings of the 2001 ICRA IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164), Seoul, Korea, 21–26 May 2001; Volume 3, pp. 2523–2528.
27. Li, W.; Christensen, H.I.; Oreback, A.; Chen, D. An architecture for indoor navigation. In Proceedings of the IEEE International Conference on Robotics and Automation, ICRA '04, New Orleans, LA, USA, 26 April–1 May 2004; Volume 2, pp. 1783–1788.
28. Baillie, J.C.; Demaille, A.; Duceux, G.; Filliat, D.; Hocquet, Q.; Nottale, M. Software architecture for an exploration robot based on Urbi. In Proceedings of the 6th National Conference on Control Architectures of Robots, INRIA Grenoble Rhône-Alpes, Grenoble, France, 24–25 May 2011; p. 12.
29. Nejkovic, V.; Petrovic, N.; Tosic, M.; Milosevic, N. Semantic approach to RIoT autonomous robots mission coordination. *Robot. Auton. Syst.* **2020**, *126*, 103438. [CrossRef]
30. Coito, T.; Martins, M.S.; Viegas, J.L.; Firme, B.; Figueiredo, J.; Vieira, S.M.; Sousa, J.M. A Middleware Platform for Intelligent Automation: An Industrial Prototype Implementation. *Comput. Ind.* **2020**, *123*, 103329. [CrossRef]
31. Mouradian, C.; Errounda, F.Z.; Belqasmi, F.; Glitho, R. An infrastructure for robotic applications as cloud computing services. In Proceedings of the 2014 IEEE World Forum on Internet of Things (WF-IoT), Seoul, Korea, 6–8 March 2014; pp. 377–382.