Jorge Miguel Carvalho Claro

Licenciado em Ciências da Engenharia Eletrotécnica
e Computadores

AGNI: an API for the Control of Automomous Service Robots

Dissertação para obtenção do Grau de Mestre em
Engenharia Eletrotécnica e de Computadores

Orientador: João Paulo Pimentão, Professor Auxiliar, Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa

Co-orientador: Pedro Sousa, Professor Auxiliar, Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa

Júri:

Presidente: Doutor Ricardo Luís Rosa Jardim Gonçalves, Professor Associado com Agregação da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Vogais: Doutor José António Barata de Oliveira, Professor Auxiliar da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Doutor João Paulo Branquinho Pimentão, Professor Auxiliar da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2014

AGNI: an API for the Control of Autonomous Service Robots

Á minha família…

Acknowledgements

It was a hard journey since 2008, with its ups and downs. The work presented in this dissertation is the result of all the support I got during the last years, studying electrical Engineering.

I would like to thank my mentors and teachers João Pimentão, Pedro Sousa, Tiago Cabral Ferreira and Sérgio Onófre for all the support during the development this research. For many months I worked together with Bruno Dias, and Bruno Rodrigues who shared their ideas and made possible to accomplish the objectives that I intended to. Nuno Zuzarte naturally spread his good humor and João Lisboa, pooled great discussions that enriched my experience through Holos.

All my friends, especially Fábio Miranda, who guided, shared, listened and supported me for many, many years, also deserve my deepest thanks.

Finally I thank my loving family, who ever supported me, and without them I would never be who I am today.

Resumo

Com o crescimento do número de dispositivos ligados á internet, a escalabilidade dos protocolos utilizados para os interligar depara-se com um conjunto de novos desafios. Na robótica estes protocolos de comunicação são um elemento essencial e devem ser capazes de os superar.

Num contexto de uma plataforma de multi-agentes, os principais tipos de protocolos de comunicação utilizados na robótica são revistos, desde o planeamento de missões, até á alocação de tarefas. A forma de representação das mensagens, o seu transporte e todos os passos envolvidos neste processo num tradicional sistema distribuído também são tratados.

Uma abordagem à plataforma ROS está também presente, onde a possibilidade de integrar um dos protocolos de comunicação já existentes no ServRobot, um robot autónomo, e o DVA, um sistema de vigilância autónomo, é também estudada. A possibilidade de atribuir missões de segurança ao ServRobot é tratada como objectivo principal.

Palavras-chave: Robô; informação; protocolo; arquitetura; comunicação; distribuído; serviço; missão

Abstract

With the continuum growth of Internet connected devices, the scalability of the protocols used for communication between them is facing a new set of challenges. In robotics these communications protocols are an essential element, and must be able to accomplish with the desired communication.

In a context of a multi-agent platform, the main types of Internet communication protocols used in robotics, mission planning and task allocation problems will be revised. It will be defined how to represent a message and how to cope with their transport between devices in a distributed environment, reviewing all the layers of the messaging process.

A review of the ROS platform is also presented with the intent of integrating the already existing communication protocols with the ServRobot, a mobile autonomous robot, and the DVA, a distributed autonomous surveillance system. This is done with the objective of assigning missions to ServRobot in a security context.

Keywords: Robot; information; protocol; architecture; communication; distributed; service; mission

Contents

List of Figures

List of Tables

Glossary

AGNI	The API for the control of autonomous service robots presented in this dissertation.
AMQP	The Advanced Messaging Queuing Protocol is an open standard application layer protocol for message-oriented middleware.
API	Application Programming Interface is a set of programming instructions and standards for accessing a Web-based software application or Web tool. A software company releases its API to the public so that other software developers can design products that are powered by its service.
ARM	ARM is a family of instruction set architectures for computer processors based on a reduced instruction set computing (RISC) architecture developed by British company ARM Holdings.
CoAP	Constrained Application Protocol is a software protocol designed to be used in very simple electronic devices, allowing them to communicate over the Internet.

CORBA	Common Object Request Broker Architecture is a standard for interoperability in heterogeneous computing environments. It enables applications to overlay different technologies and programming languages. It specifies how client applications can invoke operations on server objects.
CPU	A central processing unit (CPU) is the hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system.
DDS	The Data Distribution Service for Real-Time Systems is an Object Management Group (OMG) M2M middleware standard that directly addresses publish-subscribe communications for "real-time" and embedded systems.
DPWS	Device Profile for Web Services is a set of constraints that resource constrained devices should implement in order to enable secure and seamless Web service messaging.
DVA	Is a distributed autonomous surveillance system able to detect risk situations using different types of sensors and their geo-localization.
GPS	The Global Positioning System is a space-based satellite navigation system that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites.

GUI	In computing, a graphical user interface is a type of interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation.
IP	An Internet Protocol address (IP address) is a numerical label assigned to each device (e.g., computer, printer) participating in a computer network that uses the Internet Protocol for communication.
IT	Information technology is the collection of technologies that store, retrieve, transmit and manipulate data, often in the context of a business or other enterprise.
JMS	Java Message Service API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients
M2M	Machine-to-Machine refers to technologies that allow both wireless and wired systems to communicate with other devices.
MAC	A media access control (MAC) address is a unique identifier assigned to network interfaces for communications on the physical network segment. MAC addresses are used as a network address for most IEEE 802 network technologies, including Ethernet.
MOM	Message Oriented Middleware is software or hardware infrastructure supporting sending and

	receiving messages between distributed systems.
MQTT	Message Queuing Telemetry Transport is a message-centric wire protocol designed for telemetry-style data, along high latency or constrained networks, to a server or small message broker, typically used in M2M communications.
OWL	The Web Ontology Language (OWL) is designed for use by applications that need to process the content of information instead of just presenting information to humans.
OWL-S	Web Ontology Language for Services (OWL-S) is the Semantic Web Services description language. OWL-S builds on the Ontology Web Language (OWL).
P2P	Peer-to-peer computing or networking is a distributed application architecture that partitions tasks or work loads between peers. Peers are equally privileged, equipotent participants in the application.
PID	A Proportional-Integral-Derivative (PID) controller is a control loop feedback mechanism (controller) widely used in industrial control systems.
PLAYER	Robot framework, client/server based, where the client is an external program that interacts with the player server using classic TCP/IP sockets.
QoS	Quality of Service refers to several related aspects of computer networks that allow the transport of traffic with special requirements. For example, Asynchronous Transfer Mode (ATM) networks specify modes of service that ensure optimum

	performance for traffic.
Qt	Qt is a cross-platform application framework that is widely used for developing application software with a graphical user interface (GUI).
REST	Representational state transfer is a style of software architecture for distributed systems such as the World Wide Web-
ROS	The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.
ROSJAVA	Rosjava provides both a client library for ROS communications in java as well as growing list of core tools (e.g. tf, geometry) and drivers.
Rqt	Rqt is a library for calling R functions within C++/Qt4 applications. The argument interface uses QVariant for flexible exchange of data to and from R. This allows R calls as if they were part of the local Qt environment.
RS-232	In telecommunications, RS-232 is a standard for serial communication transmission of data.
rviz	ROS 3D visualization tool.
ServRobot	Is an autonomous service robot designed specially to be integrated in surveillance systems.
SI	The International System of Units (SI) is the modern form of the metric system and is the world's most widely used system of measurement, used in both

	everyday commerce and science.
SOA	The concept of Service-Oriented Architecture (SOA) defines a way to organize and utilize distributed capabilities that may be controlled by different organizations or different owners.
SOAP	Simple Object Access Protocol is an Exchanging XML-based messaging protocol used as a component of various middleware platforms including CORBA, JMS, and other proprietary platforms.
URI	Uniform resource identifier is a string of characters used to identify a name of a web resource.
Wire Protocol	The term "wire protocol" is commonly used to describe how the information is represented and transferred at the application layer from point-to-point in the network.
WS-NOTIFICATION	WS-Notification is a family of related specifications that define a standard Web services approach to notification using a topic-based publish/subscribe pattern.
WSN	Is a wireless sensor network of spatially distributed intelligent sensors to monitor physical or environmental conditions, and cooperatively pass their data through the network to a central location.
XML	Extensible Markup Language (XML) is a subset of Standard Generalized Markup Language (SGML) that is optimized for delivery over the Web; XML provides a uniform method for describing and exchanging structured data that is independent of

	applications or vendors. In other words XML is the Web's language for data interchange.
ZMQ	Zero Message Queue (ZMQ), it's a lightweight message-driven middleware library, specially designed for high throughput and low latency scenarios, such as AMQP that can be found in financial systems.

1. Introduction

With the continuum growth of Internet connected devices, the scalability of the protocols used for communication between them is facing a new set of challenges. In robotics these communications protocols are an essential element, and must be able to accomplish with the desired communication.

In a context of a multi-agent platform, this dissertation refers to the main types of internet communication protocols used in robotics, mission planning and task allocation problems. How to represent a message and how to handle with their transport between devices in a distributed environment, reviewing all the layers of the messaging process, is the key objective of this dissertation.

A small review of the Player and ROS platform is also presented where the possibility of using one of the already existing communication protocols within the ServRobot, a mobile autonomous robot, capable of obstacle avoidance, follow people, among other functionalities and the DVA, a distributed autonomous surveillance system able to detect risk situations using different types of sensors and their geo-localization, is also considered. The objective consists of assigning missions to ServRobot in a security context.

1.1 Multicore and Cloud based Computing

Until a few years ago, multi-core CPUs were expensive and rare, and limited to higher-end servers. To achieve higher performance, the only solution was to increase more and more the clock cycles out of one single core CPUs, which lead to severe heat dissipation problems among other things. Today the multi-core CPUs have become very common and inexpensive, even in small devices that everybody uses in a day-to-day basis, like smartphones. While the clock speeds trends to become stable, as have been seen for the last years, the number of cores per processor is doubling every two years or less. Moore's Law still applies [1], [7].

There are several motives that support this change of approach when dealing with processing higher volumes of data. Manufacturers have found multi-core to be the best way to scale their architectures and offer more competitive products, and the spreading of multitasking operating systems, which can translate that power into performance, justify this even more.

Supercomputing, in the other end, faces similar problems. The cost of a single high-end computer can be much higher and more difficult to maintain when compared with a cluster of more common and cheaper computers networked together to achieve a common goal. These networked computers can be even faster, more reliable, more flexible and fault tolerant.

Cloud Computing, considered as the long-held dream of computing, has the potential to solve the large part of these problems. It refers to the ability to develop applications, without concerning about overprovisioning for a service whose popularity does not meet the expected predictions, thus wasting costly resources, or under provisioning for one that becomes wildly popular, thus missing potential customers and revenue. It refers also to the hardware and systems software in the data centers that provide those services. This flexibility that allows companies to scale their products, without paying for premium is unprecedented in the history of IT [2].

1.2 Distributed Computing

There are several definitions for distributed systems such as: “a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing” [3] or “A collection of independent computers that appear to the users of the system as a single computer” [4].

In distributed systems, different ways of organizing multiple processors have been proposed. The tightly-coupled systems which consist in several processors connected together by the same bus, and sharing the same memory. And the loosely-coupled systems consisting of independent devices, each with their own separate bus and memory, sharing data over a network to achieve a predefined goal.

The term distributed can also be used in a wider sense. There are several processing methods, besides of how the processors are organized. The systems can be programmed to exchange messages inside the same process (in-process), between processes (inter-process), or between different systems [5], [6].

Some properties of distributed systems are listed below [7], [8].

- **Collaboration and connectivity** - One of the main motivations of distributed systems is their ability to connect a high quantity of geographically distributed information and services.
- **Distributability and decentralization** - The possibility of distributing different tasks to the most adequate devices - capable of executing them the best way possible - even when the ideal one is not available, it's an effective way to assign tasks among the devices. Using combined computing, disseminating work among the available devices in the networks and eliminating bottlenecks or centralized elements, is a good way to increase the overall performance of the system easily and provides fault tolerance and resiliency. Another way of achieving decentralization is making roles transient or transferable between devices.

- **Performance and scalability** - Over time, software is required to serve more users and require more performance to be able to scale up in order to handle the increased load and data transmission requirements.
- **Reliability and topology** - Making a distributed system reliable is very important. The failure of a distributed system can result in anything from easily repairable errors to catastrophic meltdowns. A reliable distributed system is designed to be as fault tolerant as possible, reducing the dependability on the components. The topology of the components can be classified in two classes.
  - **Physical topology** - Where are all physical devices and network devices are interconnected in the real world.
  - **Virtual topology** - Where all virtual devices, responsible for sending commands, store data, route information among the desired devices, and apply security rules in the network, are represented.
- **Redundancy and fault tolerance** - If there are no additional mechanisms providing redundancy, it may make the system more vulnerable, since the failure of any element might impair the proper working of the whole system. The correct operation of all elements when facing partial failures is also a desired property, even if it is part of the network itself that fails.
- **Flexibility and responsiveness** - In order to accomplish different service missions in the same environment, one of the most convenient solutions is to use heterogeneous or flexible robots. When changes are introduced in the working environment or mission conditions, task added or removed, low latency until achieving good results is a good way to grant a good responsiveness of the whole system.

To write software is easy, but to write the right software is hard. Even with all the advantages described above, there are also some problems when implementing and programming this kind of distributed systems. There are some extra sets of rules that developers should be aware of, like these [9]:

- **"Forgotten Synchronization"** - When developing programs using multiple threads accessing shared data, freezes, bizarre loops and data corruption may occur. To avoid this kind of problems, developers must use protective locks and semaphores on critical parts of the code. Only this way shared data is safely read or written by one thread at a time.
- **"Incorrect Granularity"** - Splitting code into parts to protect it, does not grant that the system is going to work properly. Developers can easily make some sort of mistake, failing to consider all possible behaviors of the system. Those parts of dangerous code can be too large and they cause other threads to run slowly, or they can be too small, failing to protect the shared data properly.
- **"Lock-free reordering"** - Even taking care of the number of locks in code, the compiler and the CPU are free to reorder instructions to make things run faster, causing inconsistency in the code. This reordering problem can cause some randomly breaks. To solve this, the solution is to add some kind of "memory barriers" to protect the code.
- **"Lock convoys"** - When too many threads ask for the same lock at the same time, the entire application may freeze. There is no real solution to this problem except to try to reduce lock times and reorganize the code to reduce the probability of this problem.

There are some other rules besides these ones, but they cover more specific problems not much relevant for the main themes in this dissertation.

1.3 Applications

Several examples of applications of distributed systems include telecommunication networks, telephone and cellular networks, Air-Traffic Control Systems, GPS System, or computer networks like Internet. Note that the most powerful machines in the world are nearly all collections of computers sometimes numbering as many as several hundred, where each component participates in making services available to users or making complex calculations. The financial services industry spends billions on new IT initiatives every year, and there are lots of research and development projects over the world [5].

1.4 Network Abstraction

There are several ways available to communicate in a networked distributed environment, where the wide range of protocols can be very heterogeneous. When configuring, each component should be independent from the network interface it uses and from the protocol used to define the rules how the messages are transported. To handle the necessity to transmit information consistently, many developers end up doing some kind of messaging. There are some message queuing products that developers can use, but most of the time, they end up using simple TCP and UDP sockets.

In general TCP and UDP protocols are not hard, but when implementing large systems where the amount of data being transmitted is much higher, there are a lot of problems that need to be solved. Any messaging layer must take care of all or most of these [10]:

- When designing a messaging layer, the way the I/Os are handled must be defined. The application that creates architectures that not scale well must be blocked or opt to run the I/O procedures in the background, but that can be very hard.
- Split the components into "client/servers" and hope that servers do not disappear, or define an interval to try to reconnect every few seconds if the connection is lost, are both valid ways to handle the dynamics of our system.
- The message format on the network is also an important factor to consider. The message must be small, easy to read and write, safe from buffer overflows and easy to route between devices. Besides it must be able to be adequate to transmit large files granting their consistency.
- If the messages could not be delivered immediately, the decision to wait for that component to come back on-line, discard the messages, store them into a database, or into a memory queue, must also be taken.

- Distinct platforms such as: Windows, Unix, Solaris, between many others, represent the data in different formats, therefor there are many compatibly problems even at the operating systems layer.

1.5 Dissertation Outline

In the next chapter of this dissertation, there is an introduction to the definition of middleware; to the way the different types of middleware platforms can be organized in different groups, and a comparison between the broker based and peer-to-peer messaging concepts. In the end of the chapter, is presented a table with a detailed comparison of the most important specifications of the different messaging layer frameworks, including the ROS (Robot Operative System), which will be addressed later.

The third chapter is focused in the implementation. It starts with the proposed architecture, including the reasons taken to the choice of the previously boarded messaging layer frameworks, the message language and format used over it. The ROS platform and the basic modules needed for the developed of this API are explained, starting from here, with the concepts and resources available, through the ServRobot hardware, implemented ROS nodes and finally, the performance benchmarks.

To conclude, there is a fourth chapter with the analysis of the developed work and a view of possible future improvements.

2. Middleware Platforms

2.1 Defining Middleware

Most middleware messaging frameworks try to solve distributed architectures problems in modern distributed systems.

Creating a layer that insulates the application developer from the worries about implementation details, like different operative systems and different network interfaces. Middleware frameworks also allow the programmer to integrate applications developed for different executions contexts and in different times [7].

2.2 Service-Oriented Architectures and Web Services

Services represent intangible products such as accounting, banking, cleaning, consultancy, education, insurance expertise, medical treatment or transportation. Most of the times services combine more than one of these, and cannot be transferred of possession or ownership, cannot be stored or transported and come into existence when they are bought and consumed.

In business, a service represents the part of the code wrapped with a formal and documented interface that doesn't depend on other services or the way they operate. The concept of Service-Oriented Architecture (SOA) defines a way to organize and utilize distributed capabilities that may be controlled by different organizations or different owners. It designates anything contributing to an enterprise platform based on service-oriented principles [11].

SOA provides the adequate means to offer, discover and interact with independent or loosely-coupled systems and inter-operable or tightly-coupled systems to support the exigent requirements of the business software and applications users [7].

With the emerging technologies, Cloud-based services and SOAs, as referred in section "1.1 Multicore and Cloud based Computing", are booming, serving every client, ranging from casual Internet users to IT giants, and opening many possibilities of research advances by the scientific community. This includes more computational power, storing, networking and new infrastructure innovations, allowing significant progresses in understanding and solving complex real-world challenges [7][12]. Such challenges normally require a new approach when modeling a complex system at different levels of abstraction. It helps addressing separate system requirements and concerns, and integrates diverse sources of knowledge on the system's components and their interactions [12].

Software as a Service (SaaS), virtualization and peer-to-peer are the key to cloud computing, providing formal ways to provision computational resources, improve deployment flexibility and increase scalability, as well the dependability of cloud computing, reducing the possible points of failure [12].

The SOAP allowed that a new variant of SOAs called Web Services to be spawned, allowing developers to package application logic into services whose interfaces are described with the Web Service Description Language (WSDL). WSDL-based services are often accessed using standard higher-level Internet protocols like Enterprise Service Bus (ESB), which is a distributed computing architecture that simplifies inter-working between disparate systems. It binds the protocols and transports required at runtime across devices, and allow the reuse of different components, independently from their implementations technologies [7]. These are specified by the DPWS (Device Profile for Web Services), a set of constraints that resource constrained devices should implement in order to enable secure and seamless Web service messaging [13].

2.3 Mission Planning and Task Assignment

Web Services technologies are already mature and support many middleware products and tools. It is important that robots deployed over large distributed systems use open standards and can communicate independently from the lower levels in the protocol stack.

With the progress of those communication protocols, many researchers have been working in Internet based remote control, monitoring, mission planning and task allocation platforms, with the objective of controlling mobile robots, unnamed vehicles or simple sensors in a networked environment.

Heterogeneous robots deal with different capabilities, disparity of tasks that can be performed, secondary problems like localization and navigation, are factors that affect the outcome of a mission. Besides those factors, distributed robot systems allow users from all over the world to visit museums, automatize distribution and storage centers, manufacturing systems or networks of embedded devices. They have great potential for industry, education, entertainment and security by making valuable robotic hardware accessible to a broad audience [8].

The main goal of mission planning and task allocation in a multi robot team is to optimize available resources, and integrate them into ubiquitous computing environments using a service-oriented approach. One good example of a platform capable of this, is SURF (Service-oriented Ubiquitous Robotic Framework) [14].

SURF implements a mission-planning platform based in semantic web services technology using AI-based algorithms to provide interoperation between devices. SURF platform defines 3 main entities:

- **EKR (Environmental Knowledge Repository)** - It stores KB (Knowledge Bases) with knowledge about the Web Services in OWL-S (Web Ontology Language for Services) the Semantic Web Services description language.
- **SA (SURF agent)** - It can automatically discover required knowledge using KB to communicate with specific device and build a feasible service plan for according with the mission plan and the current environment.
- **DWS (Device Web Service)** - Each DWS can have control objects for one device or multiple devices that may work cooperatively, for instance air-conditioning devices and temperature sensors. It uses SOAP to transmit and receive the messages between them.

This structure allows SA to adapt their plans, when it is placed in unknown environments or when a new sensor is added to the actual one.

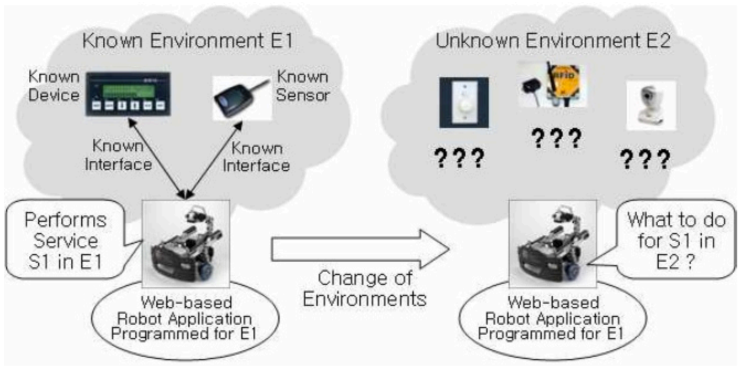

Known Environment E1
Known Device
Known Sensor
Known Interface
Known Interface
Unknown Environment E2
???
???
???
Performs Service S1 in E1
Change of Environments
What to do for S1 in E2 ?
Web-based Robot Application Programmed for E1
Web-based Robot Application Programmed for E1

Figure 2.1 - SURF - Traditional networked robotic system for specific environment E1 [14]

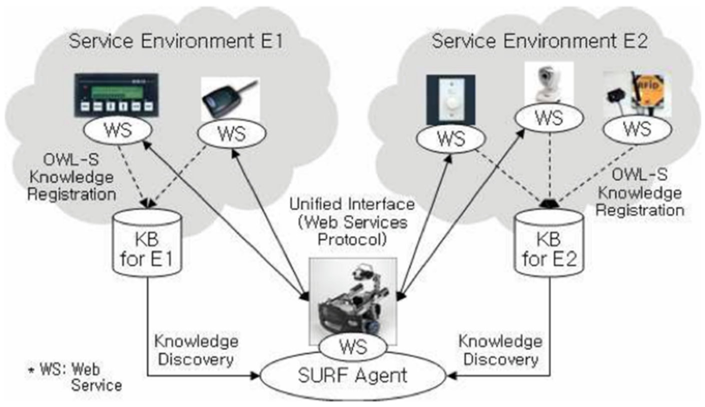


Figure 2.2 - SURF - Robots integrated into current service environments [14]

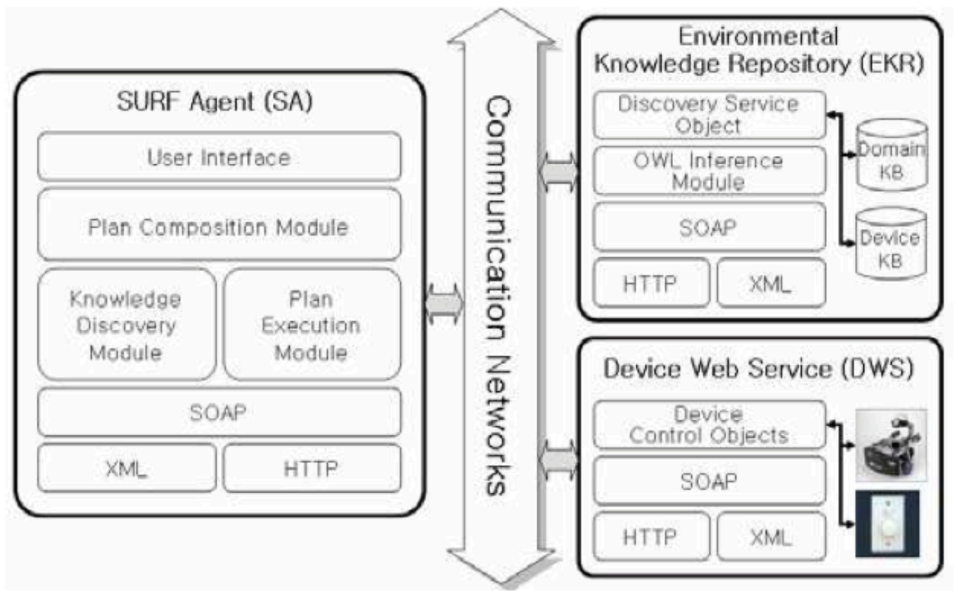


Figure 2.3 - SURF - Detailed architecture [14]

RoboLink protocol provided by the Robolink Consortium, which includes different manufacturer companies and vendors, is another platform based on Web Services.

When compared with SURF is was not implemented for mission planning capabilities, only for scattered communication among loosely coupled robots, promoting the standardization of the robot architecture and connecting robots to the network [15], [16].

The RoboLink Protocol defines two main blocks:

- **RoboLink Common Protocol** - It provides common functions to connect to a network and to communicate (session management, conversation, security).
- **Profiles** - It organizes different types of functions into different profiles. The basic interface includes generic profiles for all kinds of robots like toys, home robots and service robots. The Extended interface includes the vendor specific profiles implemented by the manufacturer with higher-level functions specific of a given robot.

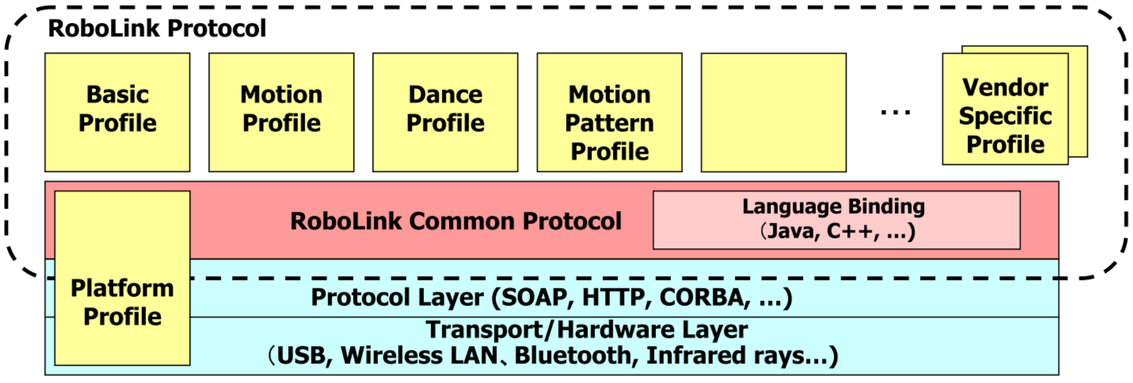

RoboLink Protocol
Basic Profile
Motion Profile
Dance Profile
Motion Pattern Profile
…
Vendor Specific Profile
RoboLink Common Protocol
Language Binding (Java, C++, ...)
Platform Profile
Protocol Layer (SOAP, HTTP, CORBA, ...)
Transport/Hardware Layer (USB, Wireless LAN、Bluetooth, Infrared rays...)

Figure 2.4 - RoboLink protocol architecture [15]

The Basic Profile includes all essential functions transversal for all kinds of robots, Motion Profile includes the basic low level motion functions, Dance profile gives control to independent components and Motion Pattern Profile is used to instruct a robot with the specification of the predefined movement pattern.

One last protocol important for the theme of this dissertation is SANCTA: An Ada 2005 General-Purpose Architecture for Mobile Robotics Research. As the name says, Sancta is a flexible architecture for controlling multi-robot teams mainly written in Ada 2005 programming language.

Accordingly with A. Mosteo:

> "The SANCTA architecture receives its name from Simulated Annealing Constrained Task Allocation, since these are the first novel features that it implemented in the field of multi-robot task allocation. Simulated annealing is a probabilistic tool useful for optimization problems with large solution spaces, able to escape local minima and, with enough running time, find good solutions or even the optimal one" [17].

In SANCTA protocol each node is defined in an agent element, and can have different configuration depending on its capabilities, and can be synced and updated in real time using XML file formats as SURF [14] or RoboLink [15] platforms. It implements a predefined abstract network interface independent from the lower levels and also a local database used to store configuration files.

The SANCTA can be integrated with different robot platforms like Player, as it was tested with, or even ROS (Robot Operative System), since it provides an integration module.

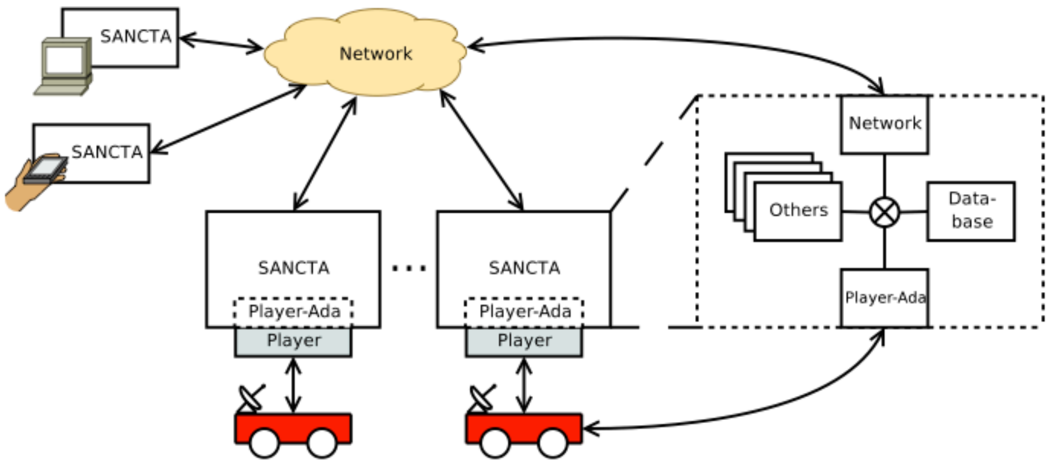


Figure 2.5 - SANCTA platform architecture [17]

Different executions modes can be used when assigning tasks, since Periodic, Event-driven or Asynchronous.

- **Periodic**: After an initial call, a determined interval can be defined for each subsequent run slice.
- **Event-driven (or synchronous)**: This mode is based in a Publish/Subscribe middleware, where a specific component subscribes a topic of interest from a database and using an observer pattern it will invoke a specific procedure.
- **Asynchronous**: Is the main task assignment mode when dealing with real-time functions. The predefined components listed in the next table are safe for use with this approach, in a typical client/server mode.

Component	Inputs	Outputs	Explanation
Global database	Network	Database	Globally accessible database for data sharing among nodes with built-in replication.
Local database	None	Database	Local data storage and sharing among components.
Annealer	Pose, Database	Task allocation	Computes a best effort task allocation for a multi-robot team using simulated annealing techniques [Mosteo 06b].
Bidder	Pose, Network	Task allocation	Bids on auctioned tasks using market-based techniques [Dias 05].
Map	Pose, Laser scan	Map	Builds an environment grid map.
Network	None	Network	Provides messaging between nodes.
Transformer	Pose	Pose	Transforms a pose in robot coordinates to world coordinates.
Scan matching	Pose, Laser scan	Pose	Uses MBICP [Minguez 05] to improve odometry using laser readings.
Aligner	Pose, Laser scan	Pose	Corrects the pose angle when the robot is in an environment with known principal orientations (i.e. orthogonal walls).
GPS	Pose	Pose	Combines an odometry pose with GPS readings to produce global localization.
Executor	Task list	Robot commands	Determines the robot actions needed to perform a task.
Go to goal	Pose, Goal	None	Issues Player [Gerkey 03b] movement calls.
GUI relay	Robot state, Network	None	Relays information to remote GUIs.
Logger	Any	None	Logs some input to disk.
Watchdog	Any	None	Aborts execution if input remains unchanged for some time.
Player_Ada	Robot commands	Robot sensors	Proxy to robot hardware [Mosteo 06a].

Table 2.1 - SANCTA Predefined component list [17]

2.4 Publish/Subscribe and Message-Oriented Middleware

RPC (Remote Procedure Calls) is a powerful abstraction technique, based on a request/response communication model. Using the network, two systems can communicate and call procedures to each other, even if they do not exist in the same address space. Caller waits for a response to be returned from the remote procedure and the calling arguments are passed to remote procedure when an RPC is made in functional call. Until either a reply is received, or the call times out, the thread is blocked from processing [5]. This block behavior can cause some troubles for some types of distributed applications, particularly those that react to external stimuli and events, such as control systems and online stock trading systems [7].

The main aspects that prevent these systems to work properly in a request/response model can be summarized in synchronous communications restrictions between client and server. These restrictions can derail parallelisms necessary to the well function of these. The client must know the identity of the server, and bound a partnership between them, forming a point-to-point communication, where no other component can interact with them [7].

An alternative to convey its information to all interested recipients, distributed systems must use message-oriented middleware to handle the messages transactions. The main advantages when comparing with request/response systems include its support for asynchronous communications, where the senders don't need to lock threads until they receive a reply. Many message-oriented middleware platforms provide a set of properties, where messages are reliable queued and/or persisted until needed by the receiver.

Publish-subscribe is a sibling of message queue paradigm and defines a pattern where publishers and subscribers are loosely coupled and thus do not know about each other existence. The main elements of a publish/subscribe middleware can be defined as:

- **Publishers**, the source of information. They classify and update information in topics that can later be read by subscribers interested on it.
- **Subscribers**, the information sinks. Every subscriber can request data from different topics and only receive messages from topics subscribed by it. Multiple subscribers can receive messages from the same publisher, and they don't have knowledge of what or how many publishers have written in a specific topic.

- **Topics**, the components in the system that create a channel, and propagate information from the publishers to subscribers. These channels propagate information across distribution domains to remote subscribers and can perform various services, such as filtering and routing, QoS enforcement, and fault management.

To represent the information passed from publishers to subscribers, there are various options available, ranging from, simple text messages to even richly-typed data structures like XML. This flexibility allows the interfaces to be generic, such as send and receive methods that exchange arbitrary dynamically typed XML messages in WS-NOTIFICATION, or specialized formats, such as a data writer and data readers that exchange statically typed event data in DDS [7].

2.5 Queuing and Messaging Layer Frameworks

When the necessity of planning missions or any framework dedicated to their assignment to the robots could not be discarded, messages syntax, identification, routing, transportation, and error checking among other problems, still need to be solved. This is where messaging frameworks take in action.

Messaging platforms abstract some of the these low-level details or socket types, allowing to hide much of the complexity, that developers are forced to repeat in their applications, every time they try to exchange messages in a consistent way.

2.5.1 Message Broker versus Peer-to-Peer (P2P)

There are two main approaches to control the way the messages are exchanged between nodes, broker-based or peer-to-peer.

In broker-based implementations, data does not flow directly from the publishers to the subscribers. Instead the data streams of all publishers are concentrated in a single trusted node. This node is responsible for the routing and delivery service. Subscribers only connect with the broker agent, and do not have to keep track of the status of the publishers, considering that it also performs message filtering, and prioritize a queue before routing [18].

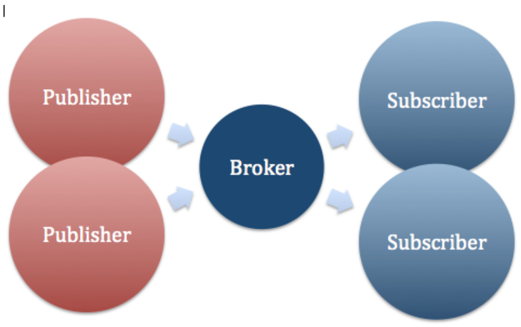


Figure 2.6 - Message Broker Architecture

In peer-to-peer implementations, subscribing node directly contacts every publisher, which is publishing a specific topic and maintains a separate subscription to each of them. This method has the advantage of independent connections between the nodes, leading to a more stable and more robust system. If the bandwidth of the underlying network is large enough, it also provides a low latency between the publishers and the subscribers [18].

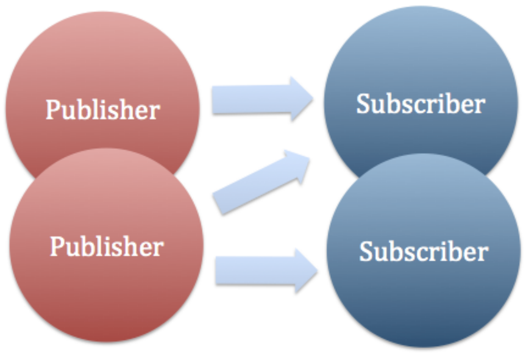


Figure 2.7 - Peer-to-Peer Architecture

2.5.2 Messaging Frameworks

This section presents an overview of the most important messaging frameworks used in networked systems, an their possible relevance in the scope of this dissertation.

2.5.2.1 AMQP

AMQP (Advanced Messaging Queuing Protocol) is a message-centric protocol born proposed by the financial sector, aimed to free users from proprietary and non-interoperable messaging systems. As well as JMS (Java Message Service), it was designed to address applications requiring fast and reliable business transactions, but unlike it, AMQP assures that implementations from different vendors are truly interoperable. JMS merely defines an API (Application Programming Interface) and AMQP is a true wire protocol. The term "wire protocol" is commonly used to describe how the information is represented and transferred at the application layer from point-to-point in the network.

AMQP uses a binary encoding format and provides flow control with message-delivery guaranties such at-most-once (where each message is delivered once or never), at-least-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL (Simple Allocation and Security Layer) and/or TLS (Transport Security Layer), assuming an underlying reliable transport layer protocol [19].

2.5.2.2 JMS

Java Message Service (JMS) is one of the most widely used publish-subscribe messaging technologies. JMS is a message centric API for sending messages between two or more clients. JMS is part of Java Enterprise Edition (Java EE) and assures the communications of distributed and loosely-coupled applications based on Java EE. JMS is asynchronous and supports both point-to-point and publish-subscribe style routing. The main limitations of JMS is that it is only a Java API standard, and does not define a wire protocol [19].

2.5.2.3 MQTT

Message Queue Telemetry Transport (MQTT) is a message-centric wire protocol designed for telemetry-style data, along high latency or constrained networks, to a server or small message broker, typically used on M2M (Machine-to-Machine) communications.

Is an extremely simple protocol, supports publish-subscribe style and devices may range from sensors, actuators, smartphones, embedded systems on vehicles, or full-scale computers.

MQTT uses a binary encoding, and supports partial interoperability between different MQTT implementations. The Message body must be agreed between peers, otherwise the message cannot be interpreted [19].

2.5.2.4 REST

Representational State Transfer (REST) started as the predominant Web API design model in the context of HTTP (Hypertext Transfer Protocol), but not limited to that. RESTful style architectures are based on conventionally request-response messaging style. It defines resources as any coherent and meaningful concept that may be addressed, based or not, in already existing applications layer protocols if they provide a rich and uniform vocabulary capable to represent a state [19].

2.5.2.5 CoAP

Constrained Application Protocol (CoAP) is a document transfer protocol designed to allow very simple electronic devices to communicate over the Internet. CoAP is lightweight, runs over UDP with support for multicast addressing, and is often used in WSNs (Wireless Sensor Networks). CoAP is compatible with client-server model based on RESTful architectures in which resources are server controlled abstractions identified by Universal Resource Identifiers (URIs). This new standard enables the use of IPv6 in Low-Power and Lossy-Networks (LLNs) such as those based on IEEE 802.15.4 and is currently being standardizing by the IETF (Internet Engineering Task Force) [19].

2.5.2.6 DDS

Data Distribution Service (DDS) was designed to support large scale, real-time data sharing between devices in a network. It is used in many mission critical systems with large device-to-device data exchanges requiring efficient, predictable, low latency and reliable data sharing.

Unlike other platforms such as AMQP, MQTT or JMS, DDS provides support for dynamic discovery. This means that DDS doesn´t need to implement a broker agent to exchange messages between peers.

Communication between publishers and subscribers are all based on direct P2P links, between nodes (inter-process) or even on a single node (in-process) as referred in section *"1.2 Distributed Computing"* of this dissertation.

By design DDS´s connectionless architecture scales better than the other protocols when the number of applications on the node producing and consuming data increases [19].

2.5.2.7 ZeroMQ

ZeroMQ (ØMQ/ZMQ) resembles the standard Berkeley sockets. It's a lightweight message-driven middleware library, specially designed for high throughput and low latency scenarios, such as AMQP that can be found in financial systems.

It provides a new type of sockets that carry whole messages across different types of transport, and able us to connect N-to-N sockets with patterns like Publish-Subscribe, Parallel-Pipeline, Fair-Queuing, or Request-Response. Those concepts, made ZeroMQ initially called a 'messaging middleware' later 'TCP on steroids' and right now a 'new layer on the networking stack' [20]. It is transport agnostic, supports in-process, inter-process, and multicast communication, all together. To achieve the best possible performance it uses different protocols, depending on the peers location [20]. Users have full control over communication policies and QoS (synchronous or asynchronous communication, timeouts). As an asynchronous processing model, the messages can be dispatched, delivered and queued (sender or receiver side) in parallel without need to block the main application process [21].

ZeroMQ is routing and network topology aware, since isn't needed to explicitly manage the peer-to-peer connection state. A single ZeroMQ socket is able to bind two or more distinct ports to listen for inbound requests, at the same time without any conflict, or the same in reverse using a single API call to send data to distinct sockets [1], [10].

ZeroMQ has no type specification and does not know anything about the data a user sends. For this reason it has to be used with an external serializer. It's considered as one of the major candidates to replace CORBA (Common Object Request Broker Architecture), as a standard on distributed systems, facilitating the collaboration between different operating systems, programming languages, and computing hardware. [6].

2.5.2.8 ROS

Robot Operating System (ROS) platform arose from the need to integrate common solutions employed in the robotic area and make the development easier. ROS is not a common operating system but rather a mixture of different tools. This high level and service oriented communication concept can be defined as a middleware, whereas the core libraries execute framework functionalities [22].

ROS developers adopted to use peer-to peer (P2P) communication instead of a centralized node (brokered) to handle the messages. Considering ROS a robot development framework where multiple nodes share information collected by sensors to processing nodes, the P2P model offers better scalability and performance.

There is only a small part of the system is centralized - the naming service. It is responsible for registering new services, inform which ones are available and which nodes are responsible for them. After this stage, all communication is independent from the naming service [22].

2.5.3 Overview

Table 2.2 - Comparison of Middleware Platforms

	DDS	MQTT	AMQP	JMS
Abstraction	Pub-Sub	Pub-Sub	Pub-Sub	Pub-Sub
Architecture Style	Global Data Space	Brokered	P2P / Brokered	Brokered
Interoperability	Yes	Limited	Yes	No
Performance	10s of 1000s of messages per second. Massive fan-out performance	Typically 100s to 1000+ messages per second per broker.	Typically 100s to 1000+ messages per second per broker.	Typically 100s to 1000+ messages per second per broker.
Real-time Processing	Yes	No	No	No
Transport Layer	UDP by default TCP can also be used	TCP	TCP	Not Specified Typically TCP
Subscription Control	Partitions, Topics; Message filtering	Topics with hierarchical matching	Exchanges, Queues and bindings	Topics and Queues; Message filtering
Data Serialization (Wire Protocol)	CDR	Undefined	AMQP type system or user defined	Undefined
Encoding	Binary	Binary	Binary	Binary
Licensing Model	Open Source; Commercial support	Open Source; Commercial support	Open Source; Commercial support	Open Source; Commercial support
Service Discovery Layer	Yes	No	No	No
Mobile Devices (Android, iOS)	Yes, on commercial applications	Yes	Yes	Dependent on the OS JAVA capabilities
6LoWPAN	Yes	Yes	Implementation Specific	Implementation Specific
Security	Vendor Specific; SSL; TLS; Proprietary access control	Simple Username-Password; SSL	SASL; TLS	Vendor specific; SSL; TLS

	REST/HTTP	CoAP	ZeroMQ	ROS
Abstraction	Request-Response	Request-Response	Pub-Sub Parallel-Pipeline Fair-Queuing Request-Response	Pub-Sub Request-Response
Architecture Style	P2P	P2P	P2P	P2P
Interoperability	Yes	Yes	Limited	Limited
Performance	Typically 100s of requests per second.	Typically 100s of requests per second	Typically 100s to 1000+ of messages per second. Faster than ROS in every case.	Typically 100s to 1000+ of messages per second.
Real-time Processing	No	No	No	No
Transport Layer	TCP	TCP	TCP	TCP UDP
Subscription Control	N/A	Multicast Addressing	Queues Multicast Addressing	Topics
Data Serialization (Wire Protocol)	No	Configurable	ZMTP	TCPROS
Encoding	Plain Text	Binary	Binary	Binary
Licensing Model	HTTP available for free on most platforms	Open Source with commercial support	Open Source with commercial support	Open Source
Service Discovery Layer	No	Yes	Yes	Yes
Mobile Devices (Android, iOS)	Yes	Via HTTP proxy	Yes	Yes
6LoWPAN	Yes	Yes	Yes	Yes
Security	SSL; TLS	DTLS	Plain-text Username – Password; Curve25519	No

3. Implementation

3.1 Introduction

Surveillance systems are part of the current mechanisms of the society for its protection against events that attempt against people health or goods. These systems have evolved, being less depended of humans and using more sensors' information to detect events.

In the DVA project (partially sponsored by the European Regional Development Fund and the Portuguese Government), a surveillance system based in geographic position of events and humans agents was developed which improves human-machine cooperation. This system reduces the dependence on humans in the detection of events and benefits from the location of the events and of human agents to improve and accelerate the response to events. It is composed by: Sensor Agents, Processor Agents, Inference Agent, Action Agents, Mobile Agents, Interface Agent, Backup Agent and Monitor Agent.

Nevertheless, there are some tasks performed by humans that could be delegated to machines, such as: confirmation of events; access to areas dangerous to human health; mobile sensors' information; reconnaissance of areas. To respond to these gaps, a partnership was established with the ServRobot project (also partially sponsored by the European Regional Development Fund and the Portuguese Government) in order to integrate the autonomous robot, developed in this later project, as an agent of the DVA system. ServRobot is an autonomous service robot designed specially for surveillance activities, and it is composed by many types of sensors, gathering information about its environment. The use of this robot as an agent, enables the execution of tasks (hitherto performed only by humans), by the robot and minimizes human intervention in various hostile situations. The ServRobot should adapt it self to different types of usage and environmental conditions, providing different residing capabilities such as: following people, lines, teleoperation, execute a predefined path based on reference points, avoid possible obstacles, in autonomous navigation and cargo transportation.

Given this partnership, AGNI, an API for the control of autonomous service robots, is presented in this dissertation. Agni is a Hindu deity, and the sacrifices made to Agni go to the deities because Agni is a messenger from and to other gods [23]. AGNI will enable the integration of the ServRobot as a DVA mobile agent.

3.2 Proposed Architecture

To use the autonomous robot, developed in ServRobot, as a new agent in the DVA's surveillance system, it is necessary to define an integration architecture and a communication protocol. The integration of the robot in DVA's system could provide new capabilities to this system, such as: send the robot to execute a mission; teleoperate the robot; or get robot's sensory information.

The architecture developed, depicted in Figure 3.1, has three primary nodes, DVA as the core of the architecture, ServRobot representing the robot and the Client representing others devices that can interact with the robot.

The DVA node refers to the DVA's surveillance system. As the core of the architecture, it handles requests regarding registration, sensor and device list queries. Authentication and permission level requests are also managed by DVA.

The ServRobot node represents the robot as an operable device. The robot must first request a registration to DVA. ServRobot's sensors can be subscribed by DVA. Subscribing a sensor allows receiving its output's values updated with a specific frequency. It's also possible for DVA to request execution of missions or remote control of the ServRobot.

The communication between ServRobot and DVA can be synchronous or asynchronous depending on what is being requested. For example in a teleoperation scenario it's clearly a synchronous communication, the orders sent to the robot must have a "real time" acknowledgement. If the DVA wants to subscribe to sensors it doesn't need to reply every time it receives an update, in this case the communication would be asynchronous [24].

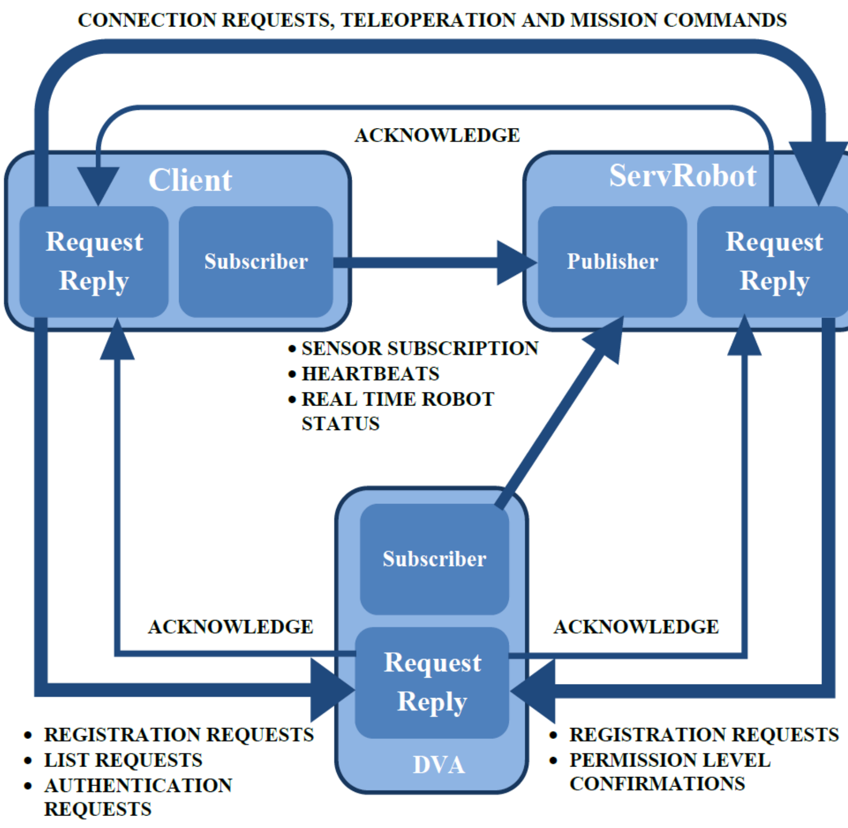


Figure 3.1 - Projected Architecture [24]

This architecture was projected to be scalable, allowing the integration with other systems. Client node could represent for example a mobile device that interacts directly with a ServRobot by teleoperating it as a remote control.

A permission level was defined to control accesses between system' nodes. This permission level allows the definition of authorization to subscribe not all the sensors of the robot, but only a few. Depending on permission level, one client may only be allowed to do teleoperation, missions, or both. There are different classes of permission levels; they limit the client's freedom, regarding the actions that they can perform [24].

3.2.1 Choice of a Message Framework

Taking into account all the requirements of the architecture described in the previous section, and all the middleware platforms reviewed in the section "2.5.2 Messaging Frameworks" of this dissertation, it was decided to use ZMQ (ZeroMQ), as its communication middleware.

The choice of the platform is justified clearly, considering the main requirements of the proposed architecture. Is has to be as much distributed as possible, must be capable of having several subscribers requiring information from the robot simultaneously, but also it must be able to provide synchronous communication patterns. Other important factors were the fact that it is one of the fastest and lightest communications frameworks available, crucial for a battery-based system [25].

DDS standard was also an option, considering all the layers that it offers, since its distributed architecture, performance, scalability, and interoperability. Even being available as open-source, it is mainly focused in commercial applications.

The community open-source distribution of DDS lacks of the more advanced features, such as mobile applications, and has considerable performance constraints.

3.2.2 Message Patterns in Use

This architecture uses two ZMQ message patterns. Request-Reply (REQ/REP) and Publish-Subscriber (PUB/SUB). Essentially, REQ/REP is used when an acknowledge is expected, for example on registration messages, direct orders or one time sensor output requests. The concept of PUB/SUB is used when one or more devices need to have periodically updates from a sensor, also to send heartbeats to the connected clients as "I'm alive" messages.

3.2.3 Message Language

The language used in the exchanged messages, is XML. XML was adopted taking into account its advantages to: modulate the concepts of the scenario in study (instead a byte codification); make changes in the message protocol by modeling new objects and types of data [26]; develop in different platforms, debug problems and validate the messages' composition [27], [28].

Also with XML an important issue to this architecture is guaranteed: communication interface does not contain limitations, so in future, new functionalities can be added easily with scalability, for new sensor/modules in the autonomous robot or new devices in the system [29].

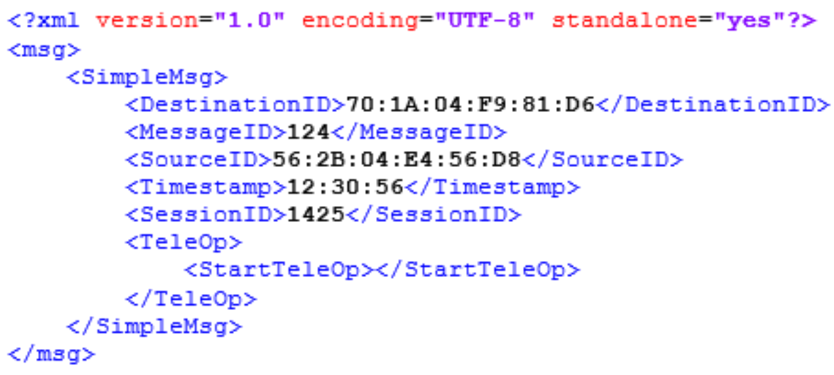
3.2.4 Message Format

The message format was defined using XML tags. Messages are initiated by the tag <msg> and followed by the *MessageType* tag as a chilld element, which defines the type of the message.

In most cases this tag contains the receiver's identification (*DestinationID*), session used to send the message (*SessionID*), message identification (*MessageID*), sender's identification (*SourceID*) and when this message was sent (*Timestamp*). After these, the *DataType* tag must follow. It identifies the specific type of data, and contains all the necessary child elements that form the message to achieve a specific purpose.

The Message's structure could be different depending on the message type. The types of messages implemented are [24]:

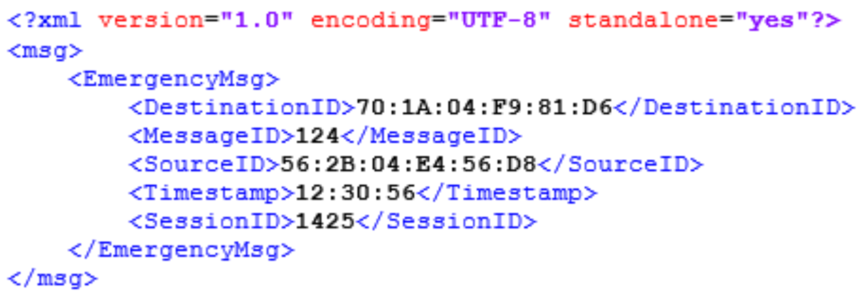
- **Simple Message** (*SimpleMsg*) – Message with the regular structure explained before.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<msg>
    <SimpleMsg>
        <DestinationID>70:1A:04:F9:81:D6</DestinationID>
        <MessageID>124</MessageID>
        <SourceID>56:2B:04:E4:56:D8</SourceID>
        <Timestamp>12:30:56</Timestamp>
        <SessionID>1425</SessionID>
        <TeleOp>
            <StartTeleOp></StartTeleOp>
        </TeleOp>
    </SimpleMsg>
</msg>
```

Figure 3.2 - Example of a "start teleoperation" message request

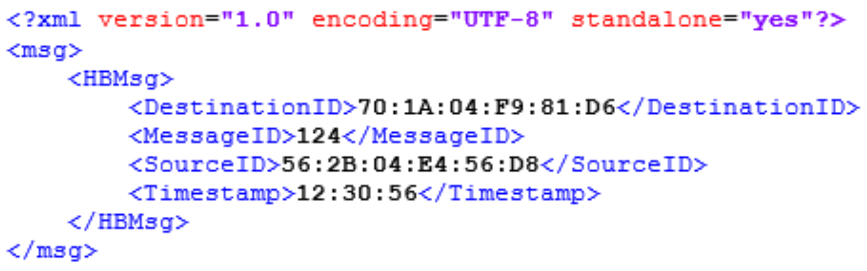
- **Emergency Command** (*EmergencyMsg*) – Message with the regular structure explained before, but without any additional elements.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<msg>
    <EmergencyMsg>
        <DestinationID>70:1A:04:F9:81:D6</DestinationID>
        <MessageID>124</MessageID>
        <SourceID>56:2B:04:E4:56:D8</SourceID>
        <Timestamp>12:30:56</Timestamp>
        <SessionID>1425</SessionID>
    </EmergencyMsg>
</msg>
```

Figure 3.3 - Example of a Emergency Message

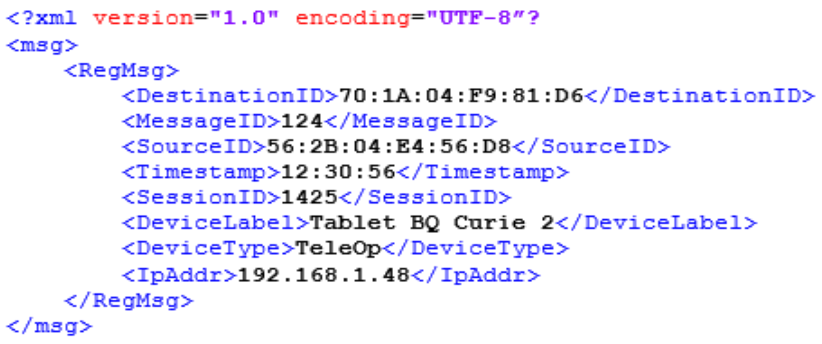
- **Heartbeat Message** (*HBMsg*) – Similar to the Emergency Command but also without the *SessionID.*



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<msg>
    <HBMsg>
        <DestinationID>70:1A:04:F9:81:D6</DestinationID>
        <MessageID>124</MessageID>
        <SourceID>56:2B:04:E4:56:D8</SourceID>
        <Timestamp>12:30:56</Timestamp>
    </HBMsg>
</msg>
```

Figure 3.4 - Example of a Heartbeat Message

- **Request Registration Message** (*ReqRegMsg*) – This message has two additional tags: *Device* and *IpAddr*. In the message, *Device* is a tag to identify the type of device (robot, teleoperation device, etc) that intends to register. *IpAddr* is a tag with the IP address of the device on the network.



```
<?xml version="1.0" encoding="UTF-8"?>
<msg>
    <RegMsg>
        <DestinationID>70:1A:04:F9:81:D6</DestinationID>
        <MessageID>124</MessageID>
        <SourceID>56:2B:04:E4:56:D8</SourceID>
        <Timestamp>12:30:56</Timestamp>
        <SessionID>1425</SessionID>
        <DeviceLabel>Tablet BQ Curie 2</DeviceLabel>
        <DeviceType>TeleOp</DeviceType>
        <IpAddr>192.168.1.48</IpAddr>
    </RegMsg>
</msg>
```

Figure 3.5 - Example of a Registration Message

3.2.5 Message Types

Each message type has a specific purpose, described in this section.

- **Emergency Command** - Used to abort an activity that is being undertaken. Independently of the current status of the robot, when it receives an emergency command, it should stop immediately.

- **Heartbeat Message** - The heartbeat message is used as an "I'm here." type of message. It is sent using a Publish/Subscribe messaging pattern and is present in all devices in the architecture.

- **Request Registration Message** - To make part of the system, the devices and clients need to send a registration request to the DVA. If a client wants to operate an available robot in the system, it must be authenticated before. To obtain a permission level, the client must ask to DVA for permissions. Figure 3.5 shows an example of a Request Registration Message that is sent by the devices to the DVA system.

- **Simple Message** - *Simple Messages* are a regular messages defined in this protocol with the structure presented specifies the type of data that is contained on the message to send and can take values as:

- **Robot Status** - Commands available in this *DataType: GetRobotStatus*. These messages can request the general status (*GetRobotStatus*) to the robot device and reply to it, with a *RobotStatus*. *RobotStatus* provides general information about the robot status: Operation Mode, Speed, Turn Rate, Speed Left Wheel, Speed Right Wheel, Direction Angle, Pitch, Roll, Yaw, coordinate X, latitude, coordinate Y, longitude, Terrain type, Reliability hours and Battery capacity.

- **Mission** - Commands available in this *DataType*: *DoStoredMission, DoNewMission, GetMissionsList, GetMissionInfo, GetMissionStatus, StoreMission, RemoveMission, GetRefPtList, StoreRefPt, RemoveRefPt*. Mission messages are used to request execution of missions to the robot (*DoStoredMission*, *DoNewMission*) and get feedback from the result of an executed mission (*GetMissionStatus*, receiving a *MissionStatus*). It is also possible get a missions list available at the robot memory (*GetMissionsList*, receiving a *MissionList*) or get a more detailed data about one specific mission (*GetMissionInfo*, receiving a *MissionInfo*). It is possible to upload and remove missions on the robot's memory (*StoreMission*, *RemoveMission*). The robot also deals with the concept of Reference Point. A Reference Point is a known location (by the Robot and the DVA) that is associated with a label. This way it is possible to execute a mission at 'Room01' instead using its coordinates. There are messages that manipulate and get these Reference Points (*GetRefPtList* receiving a *RefPtList*, *StoreRefPt*, *RemoveRefPt*).

- **Teleoperation** - Commands available in this *DataType: StartTeleOp, StopTeleOp, SetTeleOp. Teleoperation* messages are sent by a Teleoperation Device and are used to control a target device. It is possible to toggle on or off the Teleoperation Mode of the robot (*StartTeleOp*, *StopTeleOp*), and to control his speed and steering angle (*SetTeleOp*).

- **Sensor** - Commands available in this *DataType: SubSensor, UnsubSensor, SetSensorUpInt, GetSensorUpInt, SetSensorConfig, GetSensorList, GetParameterValue, GetSensorConfig, GetSensorInfo, GetMetaParameterValues, GetSensorParametersValue.* All the devices connected to a robot/sensorial device can get information about its sensors, for example: get the list of all the sensors (*GetSensorList* receiving a *SensorList)* or get the detailed information of a specific sensor (*GetSensorInfo* receiving a *SensorInfo)*. With these messages it is also possible to subscribe (*SubSensor*) or unsubscribe (*UnsubSensor*) to specific sensor values that are published by the device. Those values are published with a certain Update Interval requested by the device when it subscribes it. This Update Interval can be updated with *SetSensorUpInt*, *GetSensorUpInt* (receiving a *SensorUpInt)* messages. There are other methods to get a values from sensors, such as: *GetParameterValue* (receiving a *ParameterValue)*, *GetSensorParametersValue* (receiving all *ParameterValue* of an sensor), *GetMetaParameterValues* (receiving all *ParameterValue* of an specific Metaparameter type). The *MetaParameter* represent the type of measurement that is being sent, such as: length, temperature, pressure, acceleration, etc.

- **Device Subscription** - Commands available in this *DataType: GetDeviceList, GetSessionID.* These messages make accessible a list of devices online and connected to the DVA system (*GetDeviceList*, receiving a *DeviceList*). All the devices that are part of this list had to be registered in the system. It's also possible to use these messages to initiate a new session on a device (*GetSessionID*, receiving a *SessionID*) with a certain Permission Level. On the first interaction between a client device and ServRobot, the last one requests DVA to verify what Permission Level this session has.
  - **Permission Levels**
    - Robot Status Only
    - Sensors Only
    - Robot Status + Missions
    - Robot Status + Teleoperation
    - Robot Status + Sensors
    - Robot Status + Missions + Sensors
    - Robot Status + Teleoperation + Sensors
    - Robot Status + Missions + Teleoperation
    - Full Permissions
- **Reply** - This *DataType* of message is used as "answer" to all the messages that require some result. The reply message uses *MessageID* to identify to which request is answering. The types of *Data* used, are: *RobotStatus, MissionInfo, MissionStatus, MissionList, RefPtList, SensorList, SensorInfo, ParameterValue, DeviceList, SessionID and Acknowledge.* In case of request failure, instead of the previous types, one *ErrorCode* is returned. Figure 8 shows an example of *Reply* message with *ParameterValue*.

The diagram below shows the content and the chronological order of the messages exchanged between the DVA, the Client, and the ServRobot.

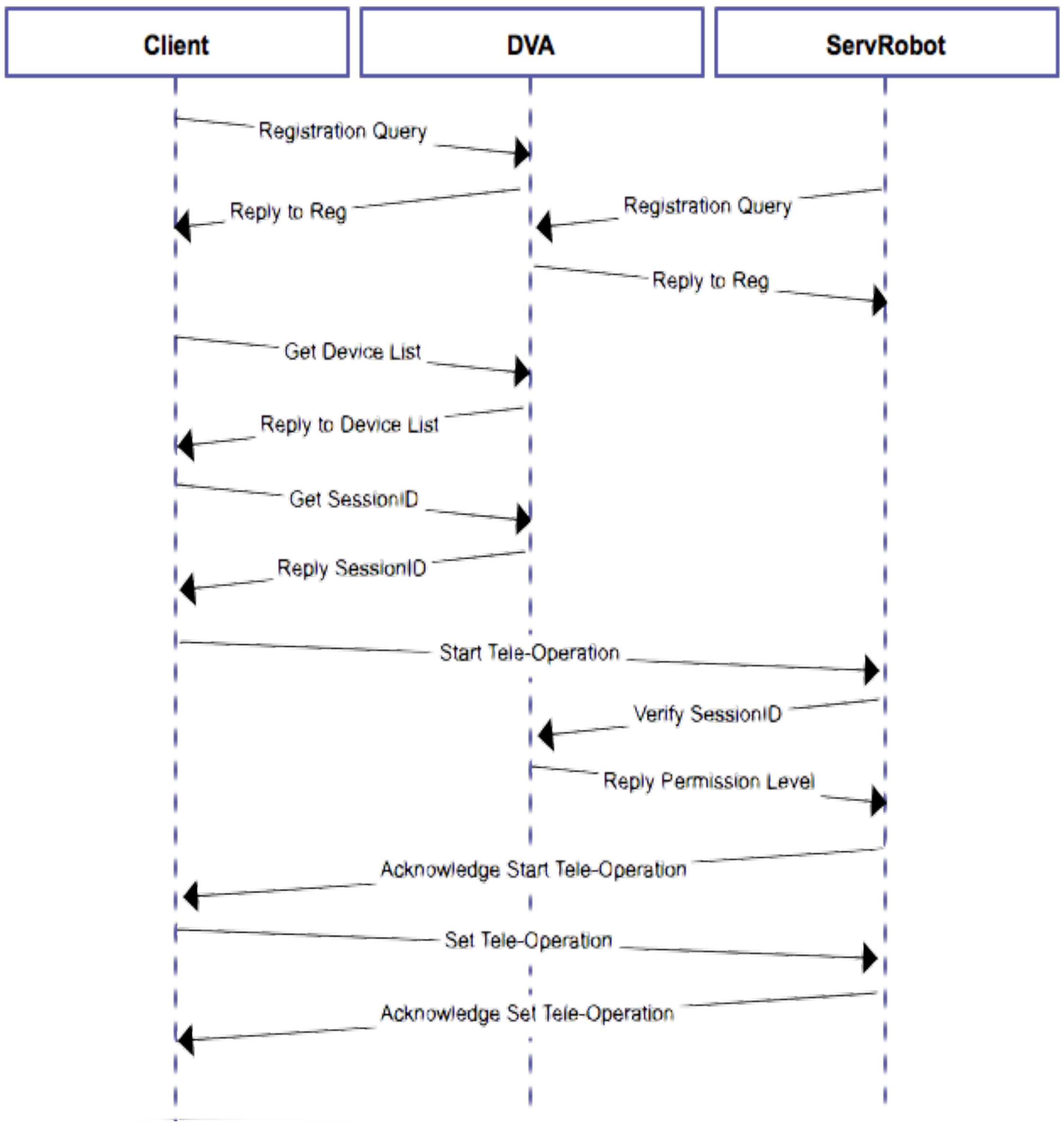


Figure 3.6 - Messages Exchange UML

After the inital registration requests and respective replies, the client requests the actual device list from DVA. From then on the client can request a new sessionID from DVA to gain access to a specific ServRobot, which will be verified later, after the robot receives a new command from the client.

Teleoperation requests or sensor information can now be exchanged between the client and the ServRobot, according with the negotiated permission level.

3.3 ROS

3.3.1 Introduction to ROS

In robotics, the widely varying types of hardware, makes software development for robot control one of the most challenging tasks in robot creation.

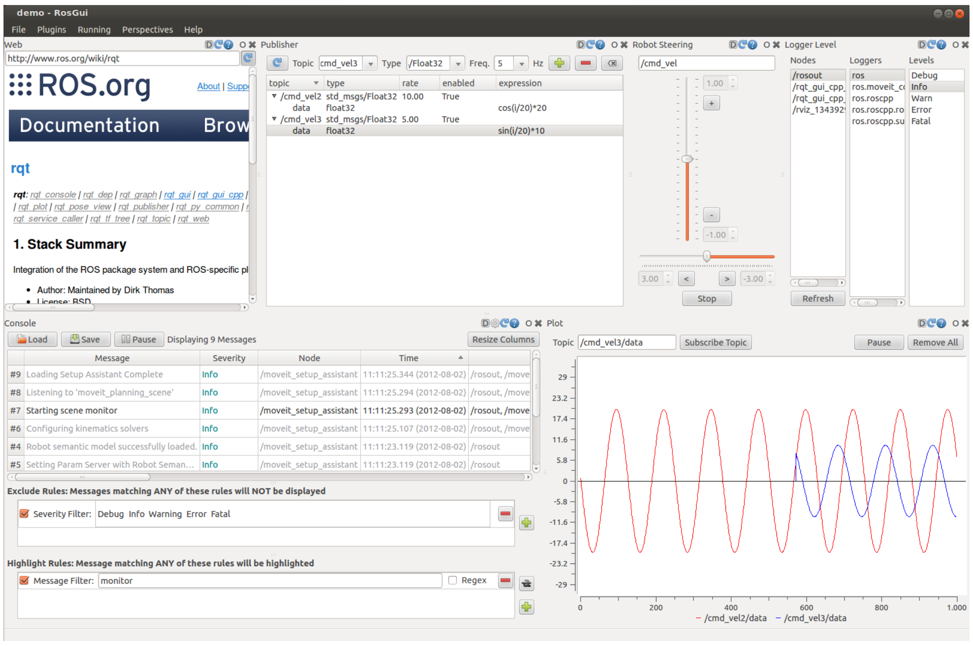
In general, robotic systems are controlled by Robotic Software Frameworks. These frameworks are focused on providing scalability, reusability, and deployment, helping to debug the software developed in the system easier. There are many Open-Source Frameworks available for the development of Robotic Systems such as: Player, OROCOS, YARP, OpenRave, OpenRTM, ROS, and others.

ROS is a product of tradeoffs and prioritizations during its development cycle. As already referenced in section "2.5.2.8 ROS" of this dissertation, the underlying goals of ROS can be summarized as: Peer-to-Peer; Tools'-based; Multi-lingual; Thin; Free and Open-Source.

It is designed to minimize the difficulty of debugging, as its modular structure allows nodes undergoing active development alongside preexisting, well-debugged nodes.

This "infrastructure" graph can be started and left running during an entire experimental session. Only the node(s) undergoing source code modification need to be periodically restarted, at which ROS silently handles the interactions between them.

It provides rqt as a Qt framework of ROS that implements the various GUI tools in form of plugins. In one single window, all ROS GUI tools are dockable within rqt, even rviz, a ROS package that visualizes robots, point clouds, etc [30].



http://www.ros.org/wiki/rqt

ROS.org About | Supp

Documentation Brow

rqt

rqt: rqt_console | rqt_dep | rqt_graph | rqt_gui | rqt_gui_cpp | | rqt_plot | rqt_pose_view | rqt_publisher | rqt_py_common | r rqt_service_caller | rqt_tf_tree | rqt_topic | rqt_web

1. Stack Summary

Integration of the ROS package system and ROS-specific pl

- Author: Maintained by Dirk Thomas
- License: BSD

Topic cmd_vel3 Type /Float32 Freq. 5 Hz

topic	type	rate	enabled	expression
▾ /cmd_vel2	std_msgs/Float32	10.00	True	
data	float32			cos(i/20)*20
▾ /cmd_vel3	std_msgs/Float32	5.00	True	
data	float32			sin(i/20)*10

/cmd_vel 1.00 + - -1.00 3.00 < > -3.00

Nodes	Loggers	Levels
/rosout	ros	Debug
/rqt_gui_cpp_	ros.moveit_c	Info
/rqt_gui_cpp_	ros.roscpp	Warn
/rviz_134392	ros.roscpp.ro	Error
	ros.roscpp.su	Fatal

Displaying 9 Messages

	Message	Severity	Node	Time	
#9	Loading Setup Assistant Complete	Info	/moveit_setup_assistant	11:11:25.344 (2012-08-02)	/rosout, /move
#8	Listening to 'moveit_planning_scene'	Info	/moveit_setup_assistant	11:11:25.294 (2012-08-02)	/rosout, /move
#7	Starting scene monitor	Info	/moveit_setup_assistant	11:11:25.293 (2012-08-02)	/rosout, /move
#6	Configuring kinematics solvers	Info	/moveit_setup_assistant	11:11:25.107 (2012-08-02)	/rosout, /move
#4	Robot semantic model successfully loaded.	Info	/moveit_setup_assistant	11:11:23.119 (2012-08-02)	/rosout
#5	Setting Param Server with Robot Seman...	Info	/moveit_setup_assistant	11:11:23.119 (2012-08-02)	/rosout

Exclude Rules: Messages matching ANY of these rules will NOT be displayed

☑ Severity Filter: Debug Info Warning Error Fatal

Highlight Rules: Message matching ANY of these rules will be highlighted

☑ Message Filter: monitor ☐ Regex

Topic /cmd_vel3/data

Figure 3.7 - RQT ROS GUI framework [30]

Logging values in ROS is also possible using the "rosbag" tool, which subscribes to one or more ROS topics. The data collected is stored in a file as it is received, leaving the need of implement logging software in each new ROS node. It allows ROS to playback the retrieved data to the same topics they were recorded from, or even to remapped new topics [31].

3.3.2 ROS Concepts and Resources

As is summarized below and in later sections, ROS structure has three levels of concepts [32]:

- File System Level – Representation of the main ROS resources
  - Packages – Runtime (nodes), libraries, datasets
  - Manifests – Flags; configuration files; licenses
  - Stacks – Collection of Packages
  - MsgTypes – Message descriptions
  - SrvTypes – Service descriptions
- Communication graph level – P2P network of ROS processes
  - Nodes – Wheel controllers
  - Master – Control communication between nodes
  - Parameter Server – Makes configuration values available

    - Messages – Communication “data structures” (int, char, ...)
    - Topics – Conjuncts of messages of a given Type
    - Services – Used to reply messages to a specific node.
    - Bags – Storing data
- Community level – Main resources that enable software and knowledge exchange.
    - Distributions – Collections of versioned stacks that can be easily installed. Similar to Linux distributions.
    - Repositories – Network of code repositories, where different institutions develop and release their own robot software components.
    - Wiki – ROS community forum and documenting platform for all the information about ROS. Anyone can contribute.

The software can be organized in several nodes, and the information can be addressed between them using the concept of topics. These topics contain information that is shared across all nodes and can be updated by any one that publishes new information on them.

The choice of the ROS as the main framework for the ServRobot is justified by several factors. The previous versions of the software developed for the ServRobot are one of the most important ones, considering the fact that ROS is developed using knowledge already acquired by the Player framework. Its high level of software integration and service oriented communication concept [22] is another important factor as already referred in the section *“2.5.2.8 ROS”* of this dissertation.

3.3.3 ROS versus Player

Player by the other hand, (the previous implementation of the software in the ServRobot), is client/server based, where the client is an external program that interacts with the player server using classic TCP/IP sockets.

It can simulate and control the behavior of the robot using all of its sensors and actuators. Is interface based, as it uses a pre-defined set of messages and data types to interact with a specific device or algorithm.

Provides several tools like Stage and Gazebo, 2D and 3D multi-robot simulators, for indoor and outdoor applications and it is responsible of all driver abstraction, hardware communication protocols, used in every sensor, actuator, or even algorithmic features as path-planning, vision, etc…

3.3.4 ServRobot Hardware

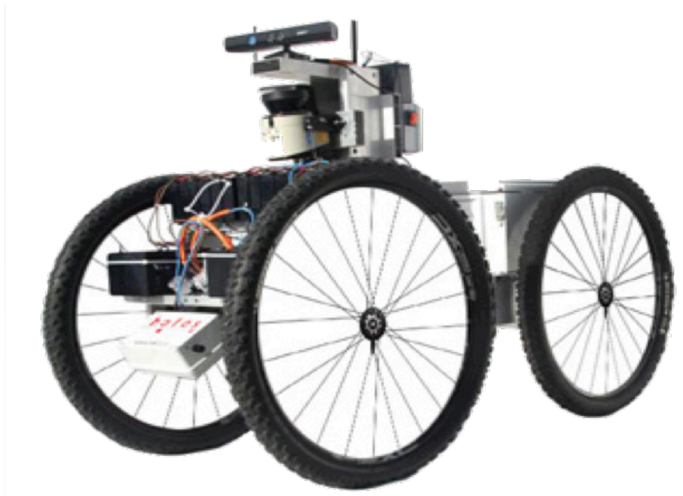


Figure 3.8 - ServRobot [33]

- ITX Computer – Used to run ROS over Ubuntu 12.04 LTS Linux operating system.
- Roboteq ax3500 – The motor controller used to control the speed of the two front wheels with a PID controller.
- Diamond Systems Hercules II - Data acquisition device to gather information from several sensors, as the direction angle encoder, electric currents and voltages of the different components, like the battery information. It also runs a Linux distribution – Knoppix.
- xsens MTI – Inertial Measurement Unit
- Sick 111 – Laser radar guidance system
- Arduino platform – Used to acquire information from the weather shield or other external sensors, and run a ROS node to publish that information.
- Weather shield – Shield with several weather sensors, including temperature, humidity, light, altitude and GPS position if available.
- USB Camera – Used to obtain images of the floor for later processing in a line follower algorithm.
- Microsoft Kineck – Human-machine interface for gesture recognition.

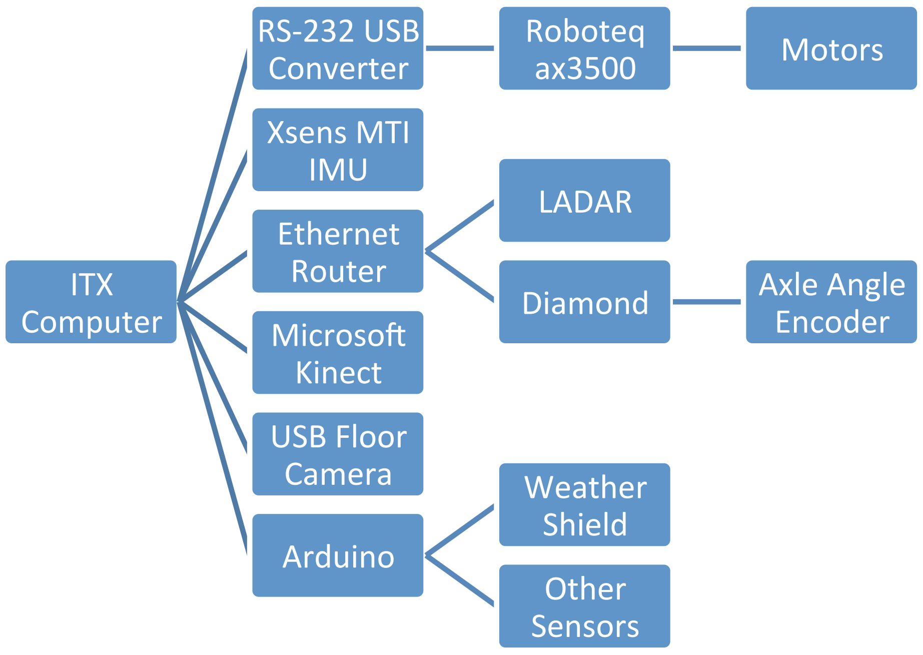


Figure 3.9 - ServRobot internal diagram

3.3.5 Motor Controller Solution

For the integration of the ServRobot with the DVA using the proposed messaging protocol, the software running on the ServRobot hardware, was rewritten to run on the new ROS framework. Some nodes that were created needed to control the basic movements of the robot, as described next in more detail.

3.3.5.1 Implemented Nodes

- **/hconfig** – Configuration node responsible to publish the configuration information of the several ServRobot ROS nodes. It verifies which configuration topics are being subscribed and updates them.
- **/hio** – Data acquisition node responsible to acquire information of the steer angle of the axis, battery power consumption information, or other new sensor values for future developments.
- **/hsteerpid** – PID correction factor calculator node responsible to receive the input speed and turn-rate command and send the corrected one to the **/hroboteq** node. This correction factor, alongside the steer angle, is used to replace the original Roboteq ax3500 differential algorithm, using

the independent wheel encoders. This able the ServRobot to execute a more accurate path.

- **/hroboteq** – Roboteq motor controller node responsible to process the received command, and interact directly with the hardware via a RS-232 serial link.
- **/serial_node** – Source of the speed and turn-rate input commands (joystick or other input method). It publishes them into **/hsteerpid_input_vel** topic.

3.3.5.2 Implemented Topics

- **/hio_config** – Data acquisition node configuration message
- **/hroboteq_config** – Motor controller node configuration message
- **/hsteerpid_config** – PID node configuration message
- **/hio_steer_angle** – Message with the steer angle of the axis in real time
- **/hsteerpid_input_vel** - Message with the value of the input speed and turn rate pretended to be executed by the ServRobot. This message can be provided by a joystick or interpreted for example as a teleoperation message from the proposed messaging protocol.
- **/hroboteq_cmd_vel** – Message with the value of the speed and turn rate to the motor controller including the PID correction factor, from the /hsteerpid node.
- **/hroboteq_raw_vel** – Message with the actual speed of the left and right wheel being executed by the /hroboteq node.
- **/hroboteq_estimated_pos** – Message with the estimated odometry data from the motor encoders.

3.3.5.3 Hroboteq node

The development process started with the communication with the Roboteq ax3500 motor controller **/hroboteq** by a serial RS-232 link. All the low-level serial communication functions were implemented, and a few ROS messages were created to start sending input commands to the motors.

There was some problems at first, related with the ROS publish / subscribe messaging structure, where the callback functions, started to conflict with some ported static routines from the previous Player drivers.

It subscribes **/hroboteq_cmd_vel** topic with the value of the speed and turn rate to the motor controller including the PID correction factor, from the **/hsteerpid** node, essential for the teleoperation function, but it also publishes the

/hroboteq_raw_vel topic with the actual speed of the left a right wheel being executed by the **/hroboteq** node and **/hroboteq_estimated_pos** topic with the estimated odometry data from the motor encoders.

3.3.4.4 Hconfig node

A configuration node was created as soon as the motor controller problems were solved. All the code written after this point was prepared to be configured at any time, using dedicated configuration messages.

All the nodes start subscribing the correspondent configuration topic. The configuration node **/hconfig** verify which configuration topics are being subscribed and update them. After receiving the first configuration, the several nodes start their initialization process. The configuration node can update the configuration topics at any time, giving the possibility to reconfigure a specific node in real-time without the need to restart all the other nodes.

3.3.5.5 Hio node

The data acquisition node **/hio** was the next one to be implemented with the purpose to get the steer angle from the axis encoder in real-time and to calculate the PID correction factor to the **/hroboteq**, by the **/hsteerpid** nodes.

The **/hio** node gets the information from the data acquisition board Hercules II from Diamond Systems, using a TCP/IP direct link. The data acquisition board should be initiated before using a SSH link. This link is started automatically using a ROS *.launch* file as described in the section "3.3.6 Parameter Server and the Roslaunch Tools" of this dissertation.

The data structure sent by the board is a simple C structure with information from the axis absolute encoder, electrical currents from the several components and correspondent electrical voltages.

3.3.5.6 Hsteerpid node

The **/hsteerpid** algorithm was ported almost without any major changes from the Player code, and interacts with the **/hio** and **/hroboteq** nodes to calculate the turn-rate correction factor. It subscribes **/hsteerpid_input_vel**, to receive the commands and publishes to the **/Hroboteq_cmd_vel**.

3.3.5.7 Basic Teleoperation ROS Messages Exchange

As represented in the next figure, after all these nodes are implemented, the ServRobot can be operated remotely using a generic ROS node that publishes the desired values of the speed and turn-rate. These values can be gathered from a simple joystick, for example.

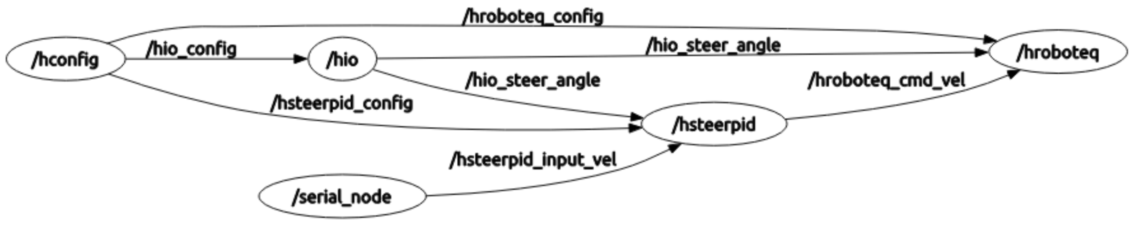


Figure 3.10 - ROS Nodes and Topics Graph for the basic robot control

3.3.6 Parameter Server and the Roslaunch Tools

After some tests, it was clear, that the ROS platform wasn´t being taken to all its potential. The use of custom messages, was working well but, for the configuration of the several nodes, the parameter server was a better solution.

Parameter Server has all the tools to publish all the parameters of all ROS nodes in one single place. Roslaunch is an easy way to fill the parameter server and order to launch new nodes to the platform, including the roscore, the base of the ROS infrastructure. Every node can access and modify if needed, all the information available in the server at any time.

In the Figure 3.11, there is a *launch* file with all the parameters needed to configure and start the **/hio** node, and run the Batch script needed to start the SSH link.

```
<launch>
        <node pkg="hio" type="start.sh" args="192.168.1.221 root diamond1"
name="start_diamond" output="screen" launch-prefix="xterm -e">
        </node>

        <node pkg="hio" type="hio" name="hio" output="screen">
          <param name="IP" value="192.168.1.221" />
          <param name="Port" value="51717" />
          <param name="Publish_CPUTemp" value="false" />
          <param name="CPUTemp_path" value="/sys/bus/platform/devices/coretemp.0/
temp2_input" />
```

Figure 3.11 - Example of a .launch file

```
#!/usr/bin/expect

#Usage sshsudologin.expect <host> <ssh user> <ssh password>

set timeout 60

spawn ssh [lindex $argv 1]@[lindex $argv 0]

expect "yes/no" {
   send "yes\r"
   expect "*?assword" { send "[lindex $argv 2]\r" }
   } "*?assword" { send "[lindex $argv 2]\r" }

expect "# " { send "./DSCServRobot\n" }
interact
```

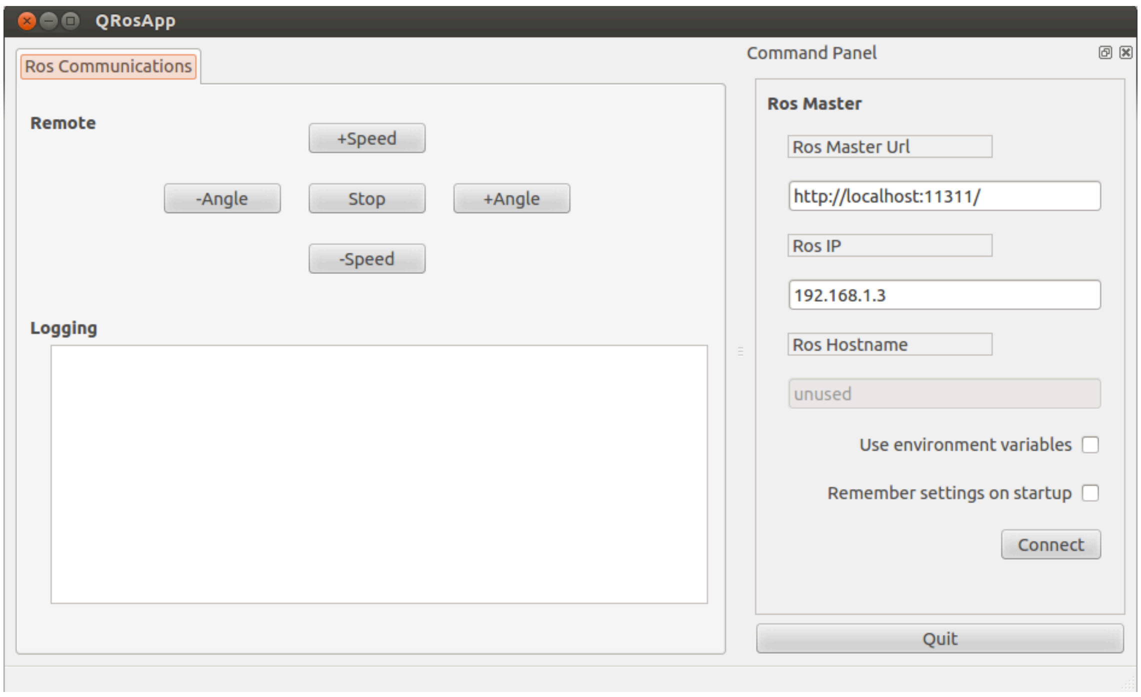
Figure 3.12 - SSH Diamond Systems Hercules II Batch Start Script

3.3.7 ROS ServRobot Remote Client

To give control of the robot to the final user, it was created a GUI using the Qt framework, the most used in ROS projects and already present in the main ROS tools and features.

The application allows the user to control the basic movements of the ServRobot using five push buttons, to increase and decrease the actual speed and turn-rate.

It's another available way to send commands to the ServRobot, publishing values to the **/hsteerpid_input_vel** topic, as already refereed in section "3.3.5.6 Hsteerpid node".



Ros Master Url: http://localhost:11311/
Ros IP: 192.168.1.3
Ros Hostname: unused
Use environment variables ☐
Remember settings on startup ☐

Figure 3.13 - ROS ServRobot Remote Client

3.3.8 Arduino, Weather Shield and Converter node

The Arduino is an open source project intended to make the application of interactive objects and environments more accessible. It's a physical computing platform based on a simple microcontroller board based on an 8-bit Atmel AVR microcontroller or a 32-bit Atmel ARM. There is also available an IDE that can be downloaded from the official Arduino website. The programing language used, is an implementation Wiring, a similar computing platform, which is based on the Processing multimedia environment [34].

The Sparkfun weather shield is an easy way to access simple weather information, like the barometric pressure, relative humidity, luminosity and temperature. It provides also connections to optional sensors such as wind speed, direction, rain gauge and GPS, if present.

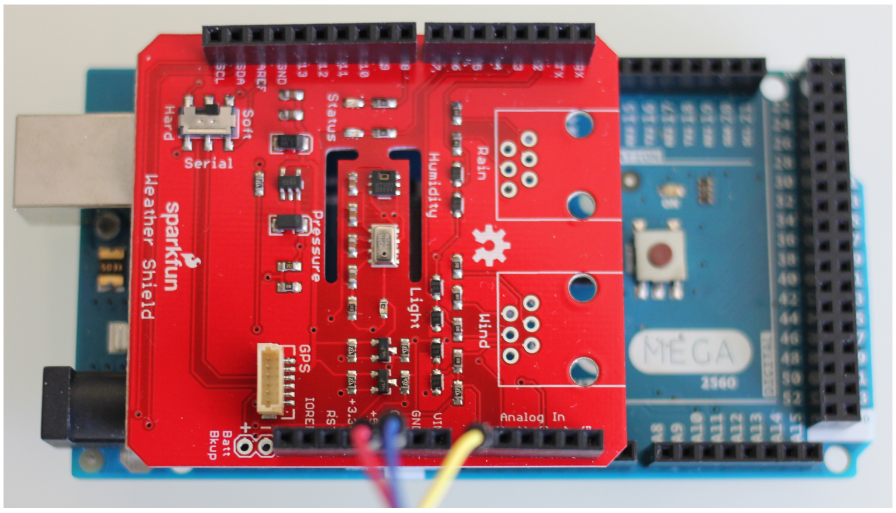


Figure 3.14 - Arduino Mega 2560 with Sparkfun weather shield

The Arduino model used is a Mega 2560, to accomplish the task of running a native ROS node internally. The Arduino will connect it self to the **/roscore** and will be represented as an **/serial_node.**

The **/serial_node** represents the sensorial information gathered from the weather shield and other external sensors, processed by the Arduino platform, and published using sensor_msgs ROS topics.

Given the lack of ROSJAVA compatibility with sensor_msgs in the current versions, an additional converter node is needed.

3.3.9 Integration of the Proposed Protocol in the ServRobot

Considering that DVA is based on a JADE platform and the mobile clients, Android devices, it was decided to implement the proposed protocol using the Java programing language. ZeroMQ and ROS are based on C++, but as already said, both are Multi-language capable.

All the middleware messaging patterns were implemented using a ZeroMQ Java code and all the XML validation was done using the JAXB platform.

A .JAR was created using all these required tools and all the new code to generate and parse the exchanged messages, to create ZMQ REQ/REP servers and clients along with ZMQ publishers and subscribers.

3.3.9.1 Implemented Nodes

- **/roszmqdriver** – ROSJAVA node that runs all the ServRobot – DVA proposed protocol.
- **/converter** – Converter node responsible to convert the sensor_msgs topics published by the Arduino to std_msgs topics, already available in the ROSJAVA platform.

3.3.9.2 ROSZMQDriver

/roszmqdriver is a ROSJAVA node with the .JAR integration allowing ROS to interact with the DVA and teleoperation clients using the proposed message protocol referred in the section "3.2 Proposed Architecture" of this dissertation.

Roslaunch is configurable, *(*3.3.6 Parameter Server and the Roslaunch Tools*)*, allowing for example, to change ServRobot registration labels, ID's or update intervals of the sensorial information. **/roszmqdriver** node is structured to use the ROS parameter server to handle these values, discarding the use of local variables.

The sensorial information available in the robot is structured using two hash-tables, with all the sensor and parameters information. The first one includes the sensor ID, sensor name and a list of meta-parameters linked with a specified sensor. The second one includes the parameter ID, meta-parameter, parameter SI base units, and values. The process of adding a new sensor to the hash-tables is represented in the Figure 3.15.

```
SensorInfo Batteries = create_sensor("Batteries");
SensorParameter Total_i = create_parameter(MsgStrings.MetaParameters.OTHER,
MsgStrings.Units.A);
SensorParameter Batt_u = create_parameter(MsgStrings.MetaParameters.OTHER,
MsgStrings.Units.V);
link_parameter(Batteries, Total_i);
link_parameter(Batteries, Batt_u);
```

Figure 3.15 - Add new sensor function

The ROS callback functions that subscribes the sensorial information, verifies if the sensor exists and update the hash-tables when new values are available. The callback functions are always the first ones to be started.

```
Subscriber<std_msgs.Float64> himu_temp_sub = rosNode.newSubscriber("/himu_raw_temp",
std_msgs.Float64._TYPE);
himu_temp_sub.addMessageListener(new MessageListener<std_msgs.Float64>() {

      @Override
      public void onNewMessage(std_msgs.Float64 imu_temp) {
          set_parameter_value(get_sensor_parameter_list(get_sensor_id("IMUtemp")),
MsgStrings.MetaParameters.TEMP, MsgStrings.Units.C, imu_temp.getData());
      }

 });
```

Figure 3.16 - ROS topic subscriber updating the hash-tables

After creating the hash-tables, the **/roszmqdriver** node gathers the current IP and MAC addresses from the Linux operative system, and verifies in the ROS parameter server if it should send a registration request to DVA.

The MAC address is normally used as the ServRobot ID tag, and the IP address used to inform the DVA, where it can find the ServRobot in the network.

```
void regist_Robot(){
        String regmsg = gen.newRegMsg(MsgStrings.DVA_IDTAG, myID, sessionID, myLabel,
MsgStrings.DeviceTypes.ROBOT, myIP);
        ZMQregClient.sendMsg(regmsg);
        String regreply;
        regreply=ZMQregClient.requester.recvStr();
        ROS_INFO(regreply);
        parser.parse(regreply);

if(parser.getMsgObject().getSimpleMsg().getReply().getCode().equals(MsgStrings.ErrorC
odes.OK)){
           //Registered OK
           ROS_INFO("Robot registered on DVA successfully");
           robot_dva_registered=true;
        }
        else{
           //Error registing ServRobot on DVA
           ROS_INFO("Robot failed to regist on DVA");
           ROS_INFO("Error:
".concat(parser.getMsgObject().getSimpleMsg().getReply().getCode()));
       }
    }
```

Figure 3.17 - /roszmqdriver registration function

If it is registered successfully, the ROSJAVA cancelable loops responsible for the several types of messages exchanged are started.

ROSJAVA cancelable loops are used to create independent ZMQ publishing cycles, for heartbeats, sensorial information or ZMQ REQ/REP servers. Each cancelable loop handles a specific ZMQ port, making the information routing more differentiable and more efficient, and has an independent update interval for each ZMQ Pub/Sub messaging pattern.

As represented in Figure 3.18, the temperature publisher ROSJAVA cancelable loop, there is a setup() function that retrieves the ZMQ port number from the ROS parameter server, and a loop() function. In every publisher, the loop() function verifies if ZMQdebug parameter is active, if is authorized to publish values, if there is any sensor installed in the ServRobot with the specified MetaPatrameter, that is added to a temporary list of parameters.

The list is published and the thread sleeps for a specified time interval.

```
CancellableLoop pub_temp_cl = new CancellableLoop() {
      ZMQPublisher temp_publisher = new ZMQPublisher();

      @Override
      protected void setup(){
     temp_publisher.connect(String.valueOf(ROSparams.getInteger("/"+getDefaultNodeNa
me()+"/ZMQ_Temp_Port")));
        ROS_INFO("Temp publisher server started");
      }
      @Override
      protected void loop() {

temp_publisher.debug=ROSparams.getBoolean("/"+getDefaultNodeName()+"/ZMQdebug");
        if(publish_values==true){
            if(temp_cl_upd_interval!=-1){
                List pValues = new ArrayList();
                Enumeration<Integer> enumKey = sensor_pValues_tab.keys();
                while(enumKey.hasMoreElements()) {
                   Integer key = enumKey.nextElement();
                   PValue pv = (PValue) sensor_pValues_tab.get(key);
                   if(pv.getMetaParameter().equals(MsgStrings.MetaParameters.TEMP)){
                     pValues.add(pv);
                   }
                }
                String tempmsg = gen.newParameterValuesMsg(MsgStrings.DVA_IDTAG,
myID, sessionID, reqID_PUB, pValues);
                ROS_INFO(tempmsg);
                temp_publisher.pusblishMsg(tempmsg);
                thread_sleep(temp_cl_upd_interval);
            }
        }
     }
};
```

Figure 3.18 - Example of a Publisher ROSJAVA Cancelable Loop

A ParameterListener is always updating the several update intervals for each type of message. It uses a parameter server callback function to receive the new values.

```
ParameterListener heartbeat_upd_int_pl = new ParameterListener() {
        @Override
        public void onNewValue(Object value) {
            heartbeat_cl_upd_interval=Integer.parseInt(value.toString());
            ROS_INFO("New Heartbeat update interval");
        }
};
```

Figure 3.19 - Parameter Listener for heartbeat update interval

3.3.10 Protocol Benchmarks and Performance

In terms of performance, the following information is based on a simple Wifi network, and two computers. The server runs on a 3.4Ghz Intel Core i5 4670K and the client, runs on a 2.53Ghz Intel Core 2 Duo, both with 8 GB of ram.

The tests executed to the protocol, were programmed to discover where is used the majority of the time during all the several steps. Batches 5000 messages were sent between the two computers, using the XML message generating process, the serialization using the ZMQ Req/Rep messaging pattern, and on the receiving computer, the XML schema validation and respective message parsing process.

Sets of small (acknowledge), regular (with information about 20 generic sensors), and large messages (with information about 200 generic sensors) were sent. The tests executed were based on request, request-reply, message generation, XML validation, and ZMQ serialization times.

The time lost during the request is based on the difference between the original message timestamp and the time when the respective content is ready to be processed after being validated and parsed by the protocol, on the destination.

The request test results are shown in Table 3.1.

Table 3.1 - Test bench with one-way simple message request delays

Delay (ms)	Small messages (Acknowledge)	**Regular messages (20 Sensors)**	Large messages (200 Sensors)
200-205	0	**0**	0
205-210	65	**16**	1
210-215	1774	**1325**	338
215-220	1953	**2483**	2052
220-225	765	**679**	1609
225-230	170	**178**	460
230-235	96	**117**	285
235-240	46	**71**	112
240-245	27	**34**	36
245-250	17	**18**	14
>250	87	**79**	93
Total	5000	**5000**	5000

The values obtained from the Table 3.1 show that the request delays were approximately around 210 and 230 milliseconds. The Figure 3.20 represents the values of the regular messages delays, from the Table 3.1.

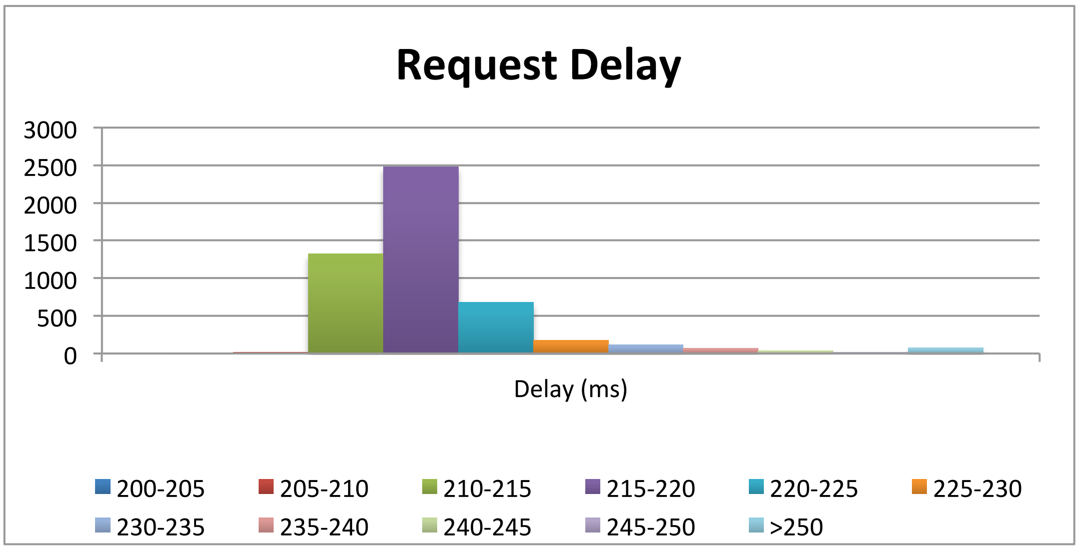


Figure 3.20 - One-way simple message request delay

The request-reply delays were measured accounting with all the steps, since the generation of the messages on the source computer, until the respective reply is received from the server and validated again on the client. The request-reply test results are shown in Table 3.2.

Table 3.2 - Test bench with simple message request-reply delays

Delay (ms)	Small messages (Acknowledge)	**Regular messages (20 Sensors)**	Large messages (200 Sensors)
200-210	0	**0**	0
210-220	22	**0**	0
220-230	2081	**1073**	621
230-240	2100	**2770**	2758
240-250	454	**636**	1021
250-260	156	**265**	328
260-270	62	**90**	69
270-280	26	**52**	48
280-290	19	**25**	30
290-300	15	**12**	26
>300	65	**77**	99
Total	5000	**5000**	5000

From the **Table 3.2**, the request-reply delays were approximately around 220 and 260 milliseconds, which are plausible, considering that the server is faster than the client and the reply message is a small acknowledge message. The delays are more spread when compared with the request delays. The Figure 3.21 represents the values of the regular messages delays, from the Table 3.2.

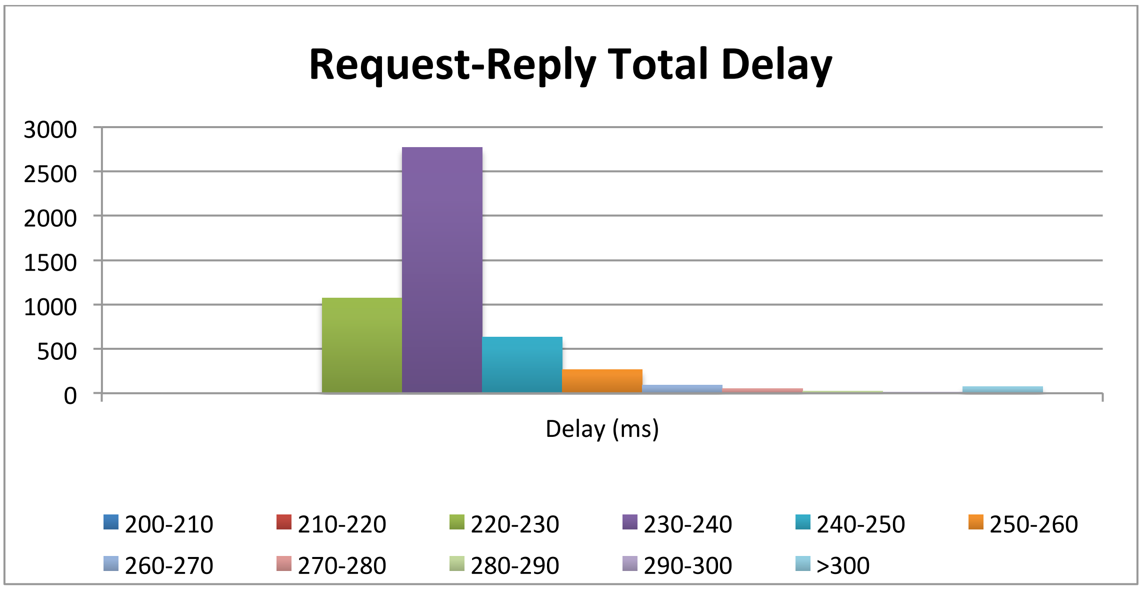


Figure 3.21 - Simple message request and response delay

The generation process delays were measured on the client computer and are shown in the Table 3.3.

Table 3.3 – Test bench to generate message process

Delay (ms)	Small messages (Acknowledge)	**Regular messages (20 Sensors)**	Large messages (200 Sensors)
50-60	0	**0**	0
60-70	0	**0**	0
70-80	2070	**1545**	2088
80-90	2562	**3041**	2384
90-100	223	**254**	365
100-110	65	**81**	69
110-120	24	**35**	28
120-130	9	**9**	15
130-140	10	**8**	5
140-150	5	**2**	10
>150	32	**25**	36
Total	5000	**5000**	5000

The values obtained from the Table 3.3, show that the generation process delays were approximately around 70 and 100 milliseconds. The Figure 3.22 represents the values of the regular message generation delays, from the Table 3.3.

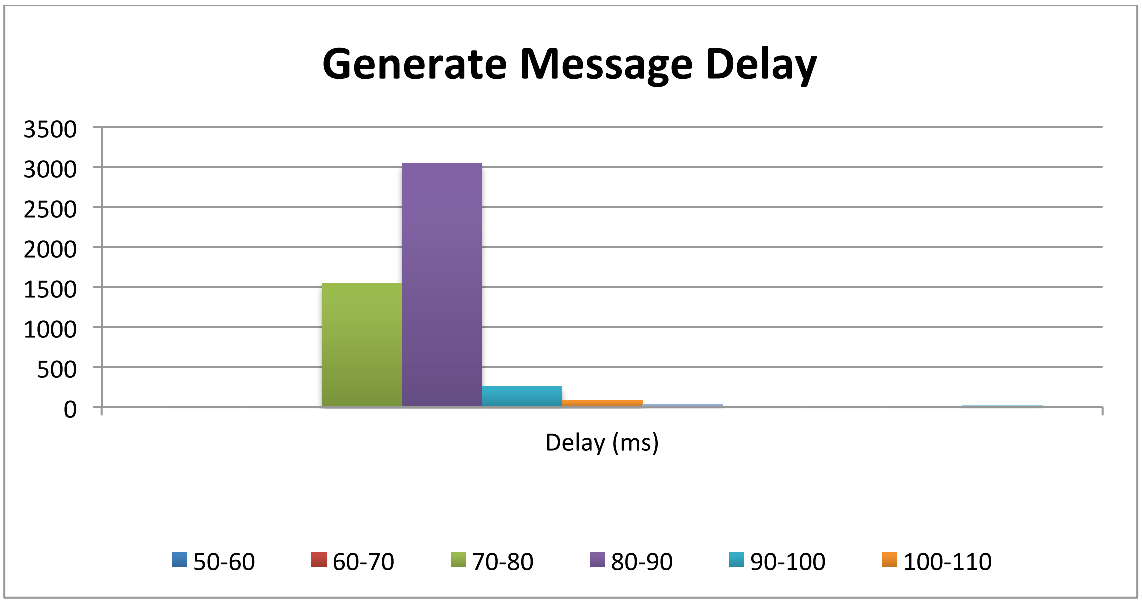


Figure 3.22 - Delay of the XML message generate process

The XML validation and parsing process delays were measured on the client computer and are shown in the Table **3.4**.

Table 3.4 - Test bench to XML validation and Parsing Delay

Delay (ms)	Small messages (Acknowledge)	**Regular messages (20 Sensors)**	Large messages (200 Sensors)
50-60	0	**0**	0
60-70	0	**0**	0
70-80	2	**0**	21
80-90	2178	**2155**	1788
90-100	2348	**2256**	842
100-110	252	**339**	495
110-120	77	**90**	946
120-130	35	**38**	318
130-140	24	**39**	418
140-150	6	**16**	84
>150	78	**67**	88
Total	5000	**5000**	5000

The values obtained from the Table **3.4**, show that the XML validation and parsing process delays were approximately around 80 and 100 milliseconds.

When compared with the generation process, the validation is slightly slower in average and more spread with larger messages.

The Figure 3.23 represents the values of the regular message XML validation and parsing delays, from the Table 3.4.

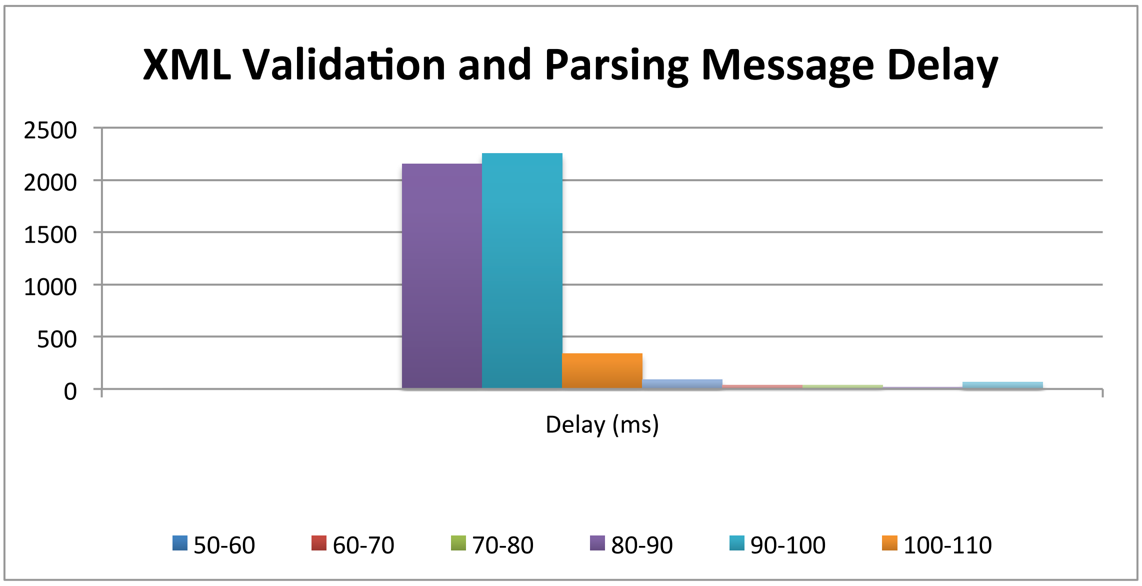


Figure 3.23 - Delay of the XML message schema verification and parsing

The ZMQ serialization delays were measured on the client computer, when it was sending request messages to the server, and are shown in the Table 3.5.

Table 3.5 - Test bench with ZMQ serialization delays

Delay (ms)	Small messages (Acknowledge)	**Regular messages (20 Sensors)**	Large messages (200 Sensors)
0-2	4578	**4494**	4391
2-4	247	**230**	356
4-6	145	**212**	194
6-8	10	**38**	21
8-10	5	**4**	9
>10	15	**22**	29
Total	5000	**5000**	5000

The Figure 3.24 represents the values of the regular messages ZMQ serialization delays.

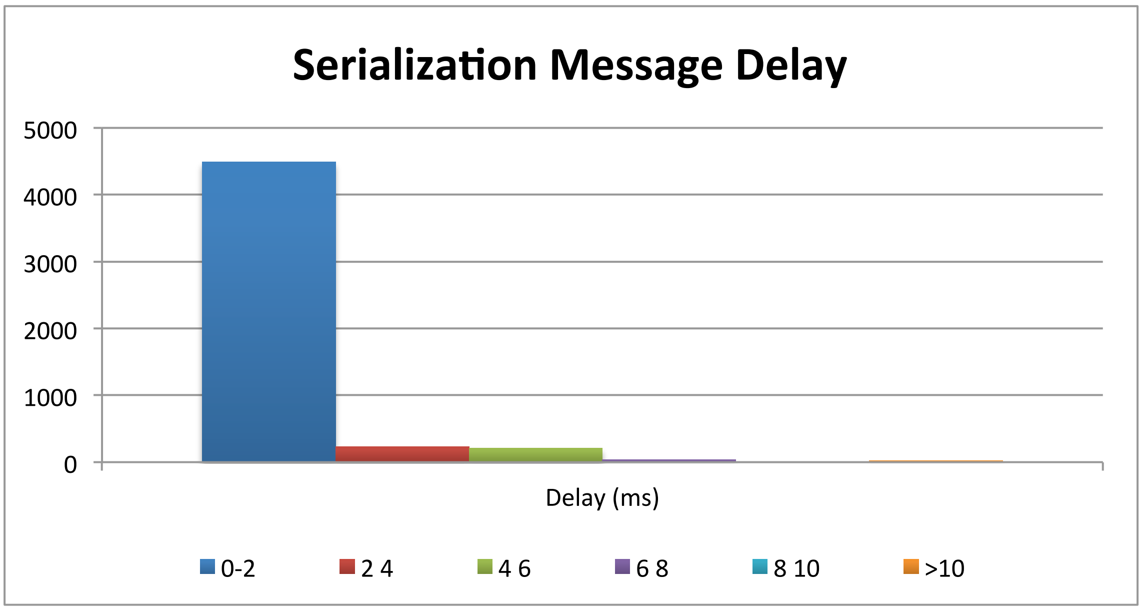


Figure 3.24 - Delay of the ZMQ message serialization

3.3.11 Himu node

The **/himu** node wraps the official Xsens C++ library, and is responsible to connect to the MTI sensor. It configures the sensor and publishes the RAW and calibrated values of the accelerometer, gyroscope, and magnetometer. Also publishes the RAW temperature and Euler angles, needed for later developments.

Figure 3.25 - Xsens inertial measurement unit

3.3.12 Lms1xx node

The **/lms1xx** allows future ports of the previous Player software. It connects with the Sick111 laser radar, configures it, and publishes the **/scan** topic with all the measured points.

Figure 3.26 - Sick LMS111 LIDAR

4. Conclusions and Future Work

At this moment the ROS nodes and topics working in the ServRobot can be represented in the graph present in Figure 4.1. The **/hsteerpid_input_vel** receives speed and turn-rate commands from the **/remote** node, representing the GUI, or the **/roszmqdriver** node, representing the teleoperation client, with negligible delay. All other nodes related with the basis ServRobot control are working as already referenced in the previous section of this dissertation, "3.3.5 Motor Controller Solution".

For future work, ServRobot ROS nodes for autonomous driving, obstacle avoidance and mapping should be ported from the previous Player implementation.

It is expected the drop of the **/converter** node in the future, as soon as ROSJAVA became compatible with a greater variety of ROS message types, including sensor_msgs. The **/hgps** is being used only to simulate the GPS position of the ServRobot. It is also expected to be dropped in the future.

The messaging protocol should be optimized, using its capabilities to customize the ServRobot's behavior, execute autonomous missions and handle session IDs, increasing the general security.

The possibility of setting the publishing update intervals for the several ROSJAVA cancelable loops in the ROS parameter server remotely, through the DVA for example, is one of the future objectives.

Error messages should also be interpreted accordingly with the desired behavior for each situation. For now, only the ID tag and command hierarchy are being verified.

The teleoperation client using the tablet and ROS remote GUI, are able to access some of the sensorial information published by the ServRobot using the same channels as DVA or ROS framework directly.

A few adjustments were made to optimize the general behavior of the messaging protocol. The ZMQ buffer was limited to a specific number of messages, considering that the ServRobot is publishing at a higher rate than DVA is subscribing and parsing the messages. The different types of messages started by being processed altogether in a single ZMQ port. It was verified that some messages should have greater priority than others and it was decided to separate them in different ROSJAVA cancelable loops, using different ZMQ ports. This way, the overhead in the network was considerable reduced and they could already be handled separately and executed only when necessary.

The ZMQ ports used in the **/roszmqdriver** node, were initially hard coded in the shared Java code. For easier configurations, these were also ported for the ROS parameter server. It should be possible to unregister, change the ZMQ port numbers through any other node, and register again the ServRobot, without the need of restarting the **/roszmqdriver**.

The request message and device IDs, should be verified in every reply received to grant a minimum level of security. This method can also be implemented in the teleoperation context if it is proved that doesn´t affect the delay considerable, when compared with the native ROS motor controller nodes.

In the section "3.2.4 Message Format" of the proposed protocol in this dissertation, is already presented a generic solution for this problem, the device subscription command.

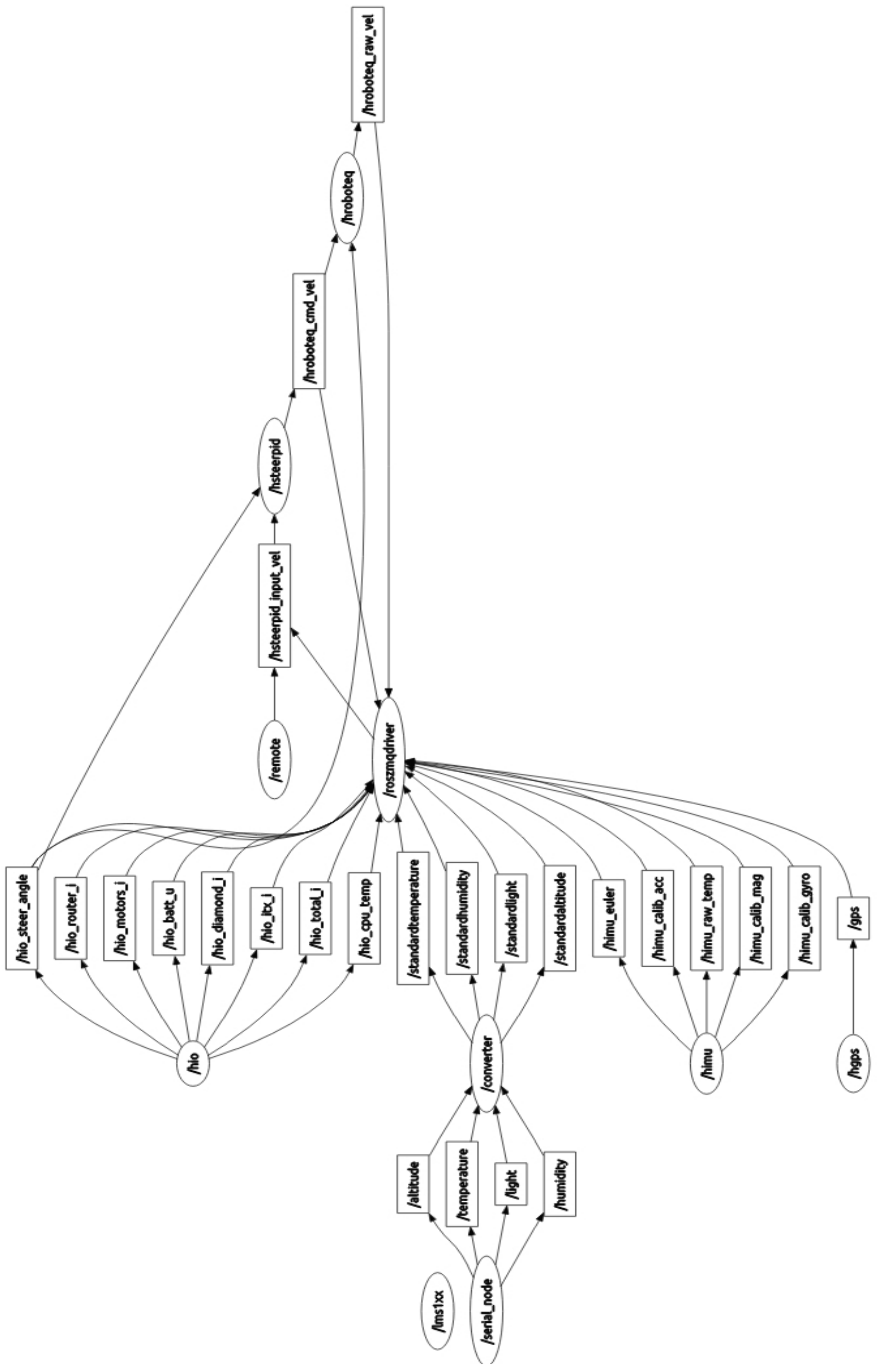
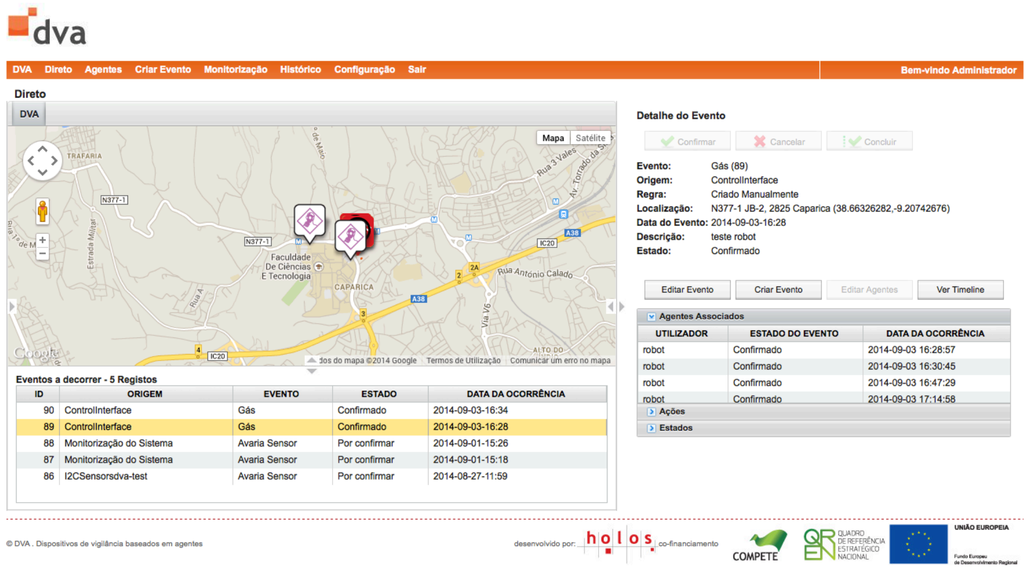


Figure 4.1 - ROS Graph with all nodes and topics

The proposed protocol, along with ZMQ and ROS, were successfully implemented in the ServRobot, and run as expected. The sensorial information collected by the robot is available for consultation through DVA or any other device using the same protocol.



Eventos a decorrer - 5 Registos

ID	ORIGEM	EVENTO	ESTADO	DATA DA OCORRÊNCIA
90	ControlInterface	Gás	Confirmado	2014-09-03-16:34
89	ControlInterface	Gás	Confirmado	2014-09-03-16:28
88	Monitorização do Sistema	Avaria Sensor	Por confirmar	2014-09-01-15:26
87	Monitorização do Sistema	Avaria Sensor	Por confirmar	2014-09-01-15:18
86	I2CSensorsdva-test	Avaria Sensor	Por confirmar	2014-08-27-11:59

Detalhe do Evento

Evento:	Gás (89)
Origem:	ControlInterface
Regra:	Criado Manualmente
Localização:	N377-1 JB-2, 2825 Caparica (38.66326282,-9.20742676)
Data do Evento:	2014-09-03-16:28
Descrição:	teste robot
Estado:	Confirmado

Agentes Associados

UTILIZADOR	ESTADO DO EVENTO	DATA DA OCORRÊNCIA
robot	Confirmado	2014-09-03 16:28:57
robot	Confirmado	2014-09-03 16:30:45
robot	Confirmado	2014-09-03 16:47:29
robot	Confirmado	2014-09-03 17:14:58

© DVA . Dispositivos de vigilância baseados em agentes

Figure 4.2 - DVA Website [35]

It is highly customizable and easily ported to other scenarios. The message tags defined in the protocol are only a limited set of possible applications. The maximum message size is not limited in any of the layers, and the smallest one is already defined as a heartbeat message, with only the header. The application of the protocol is mainly focused in sensorial data acquisition, configuration messages and mission assigning. It proved that it can be used close to real time applications, but is not the best choice when dealing with critical information in short time intervals. The most, time expensive, step in the proposed protocol is the XML validation against the XSD model. The choice of the ZMQ framework, proved to be resilient, including the possibility of a high quantity of data to be transferred in the network with almost no delay at all.

As already said, the protocol was implemented in Java, but since it uses XML language, it can be implemented in any other language that is compatible with the ZMQ framework.

For any future developments on the ServRobot, all information gathered at the moment is available to any new ROS node, running internally or externally in the same network.

Scientific Contributions

During the development of this dissertation, two papers were accepted and published in two different conferences. The first one more focused in the choice of the messaging framework and the second one in the proposed architecture and integration on the DVA.

J. Claro, B. Dias, B. Rodrigues, J. Paulo, P. Sousa, and S. Onofre, "Autonomous robot integration in Surveillance System - Architecture and communication protocol for systems cooperation," in *16th International Power Electronics and Motion Control Conference and Exposition (PEMC 2014)*, 2014, pp. 714–720.

B. Dias, B. Rodrigues, J. Claro, J. P. Pimentão, P. Sousa and S. Onofre, "Architecture and Message Protocol Proposal for Robot ' s Integration in Multi-Agent Surveillance System," in *Rough Sets and Current Trends in Soft Computing*, 2014, pp. 366–373.

References

[1] P. Hintjens, "Multithreading Magic." [Online]. Available: http://zeromq.org/blog:multithreading-magic. [Accessed: 20-Nov-2013].

[2] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, p. 50, Apr. 2010.

[3] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed systems: concepts and design*. 2005, p. 1067.

[4] M. Van Steen, *Distributed Systems Principles and Paradigms*. 2004, p. 686.

[5] G. Murali and A. Shirisha, "Remote procedure calls implementing using distributed algorithm," vol. 2, no. 6, pp. 1742–1746, 2011.

[6] A. Dworak, M. Sobczak, F. Ehm, W. Sliwinski, and P. Charrue, "Middleware trends and market leaders 2011," in *13th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2011.

[7] B. Frank, H. Kevlin, and C. Douglas, "Pattern Oriented Software Architecture 'A pattern Language for Distributed Computing,'" *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*, 2007.

[8] A. R. Mosteo, “Multi-Robot Task Allocation for Service Robotics : from Unlimited to Limited Communication Range.”

[9] J. Duffy, “Solving 11 Likely Problems In Your Multithreaded Code.” [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc817398.aspx. [Accessed: 10-Dec-2013].

[10] P. Hintjens, “ZeroMQ: The Guide.” [Online]. Available: http://zguide.zeromq.org/page:all. [Accessed: 20-Nov-2013].

[11] H. Luthria and F. Rabhi, “Service-Oriented Architectures: Myth or Reality?,” *IEEE software*, pp. 46–52, 2012.

[12] L. Liu and J. Xu, “Clouds and service-oriented architectures,” *Future Generation Computer Systems*, vol. 29, no. 1, pp. 271–272, Jan. 2013.

[13] F. Campos and J. Pereira, “Improving the Scalability of DPWS-Based Networked Infrastructures,” p. 28, Jul. 2014.

[14] Y. Ha, J. Sohn, and Y. Cho, “Service-oriented integration of networked robots with ubiquitous sensors and devices using the semantic Web services technology,” *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3947–3952, 2005.

[15] M. Narita and F. Limited, “A Robot Collaboration Protocol based on Web Services - RoboLink Protocol Connect Robots to the Network,” 2004.

[16] M. Narita, M. Shimamura, and M. Oya, “Reliable Robot Communication on Web Services,” vol. 18, no. 1, pp. 36–37, 2006.

[17] A. Mosteo and L. Montano, “SANCTA: an Ada 2005 general-purpose architecture for mobile robotics research,” *Reliable Software Technologies–Ada Europe …*, 2007.

[18] V. Netze, T. Schneider, and P. M. Kranz, “Distributed Networks Using ROS – Cross- Network Middleware Communication using,” 2012.

[19] A. Foster, “Messaging Technologies for the Industrial Internet and the Internet of Things,” no. January, pp. 1–22, 2014.

[20] N. Piël, “ZeroMQ an introduction.” [Online]. Available: http://nichol.as/zeromq-an-introduction. [Accessed: 20-Oct-2013].

[21] P. Hintjens, *Code Connected Volume 1*, vol. 1. 2013.

[22] "About ROS." [Online]. Available: http://wiki.ros.org/About ros.org. [Accessed: 06-May-2014].

[23] "Agni." [Online]. Available: https://en.wikipedia.org/wiki/Agni. [Accessed: 06-Oct-2014].

[24] B. Dias, B. Rodrigues, J. Claro, and J. P. Pimentão, "Architecture and Message Protocol Proposal for Robot ' s Integration in Multi-Agent Surveillance System," in *Rough Sets and Current Trends in Soft Computing*, 2014, pp. 366–373.

[25] J. Claro, B. Dias, B. Rodrigues, J. Paulo, P. Sousa, and S. Onofre, "Autonomous robot integration in Surveillance System - Architecture and communication protocol for systems cooperation," in *16th International Power Electronics and Motion Control Conference and Exposition (PEMC 2014)*, 2014, pp. 714–720.

[26] T. Bray and J. Paoli, "Extensible markup language (XML)," *W3C*, vol. 1, no. August, 2006.

[27] M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi, "XML Screamer : An Integrated Approach to High Performance XML Parsing , Validation and Deserialization," *International World Wide Web Conference Committee*, pp. 93–102, 2006.

[28] F. Wang, J. Li, and H. Homayounfar, "A space efficient XML DOM parser," *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 185–207, Jan. 2007.

[29] P. Iñigo-Blasco, F. Diaz-del-Rio, M. C. Romero-Ternero, D. Cagigas-Muñiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, Jun. 2012.

[30] "RQT." [Online]. Available: http://wiki.ros.org/rqt. [Accessed: 10-Jun-2014].

[31] "ROSBAGS." [Online]. Available: http://wiki.ros.org/rosbags. [Accessed: 10-Jun-2014].

[32] “ROS Concepts.” [Online]. Available: http://wiki.ros.org/ROS/Concepts. [Accessed: 10-Jun-2014].

[33] “ServRobot.” [Online]. Available: http://servrobot.holos.pt. [Accessed: 10-Jun-2014].

[34] “Arduino.” [Online]. Available: http://arduino.cc. [Accessed: 10-Aug-2014].

[35] “DVA.” [Online]. Available: http://dva.holos.pt. [Accessed: 10-Jun-2014].