



## **UNEXMIN DELIVERABLE D3.1**

# **UX-1 ROBOT SOFTWARE ARCHITECTURE REPORT**

### **Summary:**

This document reports the software design process and UX-1 software architecture.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 690008.

## Authors

Claudio Rossi, UPM

Sergio Dominguez, UPM

Pascual Campoy, UPM

André Dias, INESC TEC

Alfredo Martins, INESC TEC

Carlos Almeida, INESC TEC

José Almeida, INESC TEC



<b>Lead beneficiary:</b>		<b>Universidad Politécnica de Madrid (UPM)</b>	
<b>Other beneficiaries:</b>		INESC TEC, TUT	
<b>Due date:</b>		M16	
<b>Nature:</b>		Report	
<b>Diffusion</b>		Public	
<b>Revision history</b>	<b>Author</b>	<b>Delivery date</b>	<b>Summary of changes</b>
<b>Version 1.0</b>	C. Rossi	31.03.2017	--
<b>Version 2.0</b>			
<b>Version 3.0</b>			
<b>Version 4.0</b>			
<b>Version 4.1</b>			

<b>Approval status</b>			
<b>Function</b>	<b>Name</b>	<b>Date</b>	<b>Signature</b>
<b>Deliverable responsible</b>	Claudio Rossi	30.05.2017	
<b>WP leader</b>	Claudio Rossi	30.05.2017	
<b>Reviewer</b>	Alfredo Martins	30.05.2017	
<b>Reviewer</b>	Stephen Henley	30.05.2017	
<b>Project leader</b>	Norbert Zajzon	31.05.2017	



**Disclaimer:** This report reflects only the author's view. The European Commission is not responsible for any use that may be made of the information it contains.



## Table of contents

<b>AUTHORS</b> .....	<b>2</b>
<b>TABLE OF CONTENTS</b> .....	<b>5</b>
<b>LIST OF FIGURES</b> .....	<b>6</b>
<b>LIST OF TABLES</b> .....	<b>6</b>
<b>1 INTRODUCTION</b> .....	<b>7</b>
1.1 MIDDLEWARE .....	7
1.2 INTERFACES .....	7
1.3 STANDARDIZATION .....	8
<b>2 UML FOR ROS IN UNEXMIN</b> .....	<b>10</b>
2.1 NODES .....	10
2.2 TOPICS (EXTENDED VERSION) .....	10
2.3 SERVICES.....	10
2.4 SUBSCRIPTION/PUBLICATION (COMPACT FORM) .....	11
2.5 SUBSCRIPTION/PUBLICATION (EXTENDED VERSION) .....	11
2.6 PARAMETERS SERVER .....	12
<b>3 MAIN SOFTWARE ORGANIZATION</b> .....	<b>13</b>
<b>4 DETAILED DIAGRAMS</b> .....	<b>14</b>
4.1 COMPONENTS RELATIONSHIPS .....	14
4.2 SENSOR FUSION .....	15
4.3 LOCALISATION AND MAPPING .....	16
4.4 GNC .....	17
4.5 LOW-LEVEL CONTROL .....	18
<b>5 COMPONENT INTERFACES</b> .....	<b>19</b>
5.1 SLAM INTERFACE .....	20
5.2 POSE INTERFACE .....	20
5.3 LL COMMANDS INTERFACE .....	21
5.4 LL FEEDBACK INTERFACE.....	21
5.5 PERCEPTION INTERFACE .....	22
5.6 MANEUVERS INTERFACE .....	22
<b>6 PARAMETERS SERVER (GLOBAL SYSTEM VARIABLES)</b> .....	<b>23</b>
<b>7 OTHER SYSTEM'S ELEMENTS</b> .....	<b>24</b>
<b>8 HARDWARE</b> .....	<b>25</b>
8.1 COMPUTATIONAL HARDWARE .....	25
8.2 UX-1 HARDWARE ARCHITECTURE .....	26
<b>9 REFERENCE SYSTEM</b> .....	<b>27</b>
<b>10 SOFTWARE VERSIONS</b> .....	<b>28</b>
<b>11 REFERENCES</b> .....	<b>29</b>
<b>12 APPENDIX: UML QUICK REFERENCE</b> .....	<b>30</b>
12.1 CLASS DIAGRAMS .....	30
12.2 COMPONENTS DIAGRAMS .....	31
12.3 DEPLOYMENT DIAGRAMS AND NODES.....	32
12.4 MODELLING SYSTEM'S BEHAVIOUR .....	32
12.5 COMMON MECHANISMS .....	33



## List of Figures

<i>Figure 1. General middleware concept.....</i>	<i>7</i>
<i>Figure 2. Basic ROS architecture .....</i>	<i>8</i>
<i>Figure 3. Components view of the software system .....</i>	<i>13</i>
<i>Figure 4. Components and their relationships.....</i>	<i>14</i>
<i>Figure 5. Sensor Fusion .....</i>	<i>15</i>
<i>Figure 6. Localisation and mapping module .....</i>	<i>16</i>
<i>Figure 7. GNC component: main modules.....</i>	<i>17</i>
<i>Figure 8. Components' interfaces .....</i>	<i>19</i>
<i>Figure 9 - UX-1 Hardware Architecture .....</i>	<i>26</i>
<i>Figure 10. Reference system and thrusters numbering .....</i>	<i>27</i>

## List of Tables

<No tables>



## 1 Introduction

This document reports the general UX-1 software architecture as well as details of its current implementation status. The Middleware, interfaces and standardization process are also described.

### 1.1 Middleware

The role of the middleware is to provide an abstraction layer between the hardware and the software, as well as providing communication means between software modules and between software and hardware components.

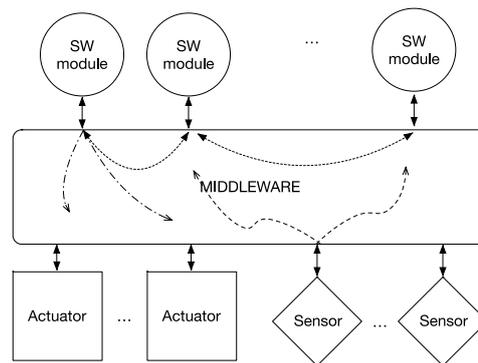


Figure 1. General middleware concept

After a brief analysis of the alternatives, it was decided to adopt the originally proposed middleware, **ROS (Robot Operating System)**. The basic reason of this choice is that ROS is imposing itself as a *de facto* standard in robotics. Additionally, the research groups involved in the software development have a long standing experience in its use, so its adoption allows a quick start of the SW development task, as well a common understanding. In brief, the advantages of ROS are:

- It is a modular system.
- Excellent hardware support (drivers).
- Many algorithms already implemented and available as open source software, which helps early prototyping of own modules.
- Easy to use tools for development
- Big community of users/developers
- It provides abstraction of communications.
- Supports point-to-point and broadcast communications.
- It is Open Source.

In brief, “ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license” [1].

### 1.2 Interfaces

Interfaces between software and hardware modules are provided by the ROS messaging system. The main communication means foreseen is the publisher/subscriber model, a broadcasting model that is agnostic to where the data is coming from and how. Consumers



(subscribers) know there will be a specific type of data (published by some other module) they are listening to. This allows decoupling software modules, at the benefit of distributed software development, software modularity and maintenance.

Interfaces between modules are described in detail later in this document.

### 1.3 Standardization

Provided at least three independent partners will be developing software (INESC: sensors registration, data fusion, SLAM; UPM: Guidance, Navigation and Control, TUT: low-level hardware controllers) and in each institution several researchers will be developing software modules, it was crucial to agree on a common language. Surprisingly, there is no standard (visual) modelling language for ROS. Therefore, we opted for adopting the widely know UML software modelling language. This required to “translate” the ROS jargon to the UML conceptual framework. UML provides modelling tools both for the static and dynamic views of the software (that is, its *structure* and its *behaviour*), and a wide range of additional elements to precisely define module interfaces, communications and interactions.

### ROS in a nutshell

We review here the ROS basics extremely briefly. For further detail we refer to [1].

In ROS, software modules are called **NODES**. “A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics and RPC services” [1]. A robot control system is composed of many nodes.

Typically, there will be a node the controls each sensor (and publish its data), nodes that control robot's actuators (subscribing to, i.e., receiving, some reference command), nodes that performs localization, mapping, planning etc. An important feature of ROS is its *distributed architecture*, that, is ROS nodes can –and typically do– run on different physical computational resources (computers, embedded PCs, etc.). The ROS system make this feature totally transparent to the nodes.

The first inter-node communication means is via **TOPICS**. Topics are intended for unidirectional, one-to-many communication. “Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic2. There can be multiple publishers and subscribers to a topic” [1].

If nodes need to perform Remote Procedure Calls (RPC), i.e. receive a response to a request, ROS provide the concept of **SERVICES**. RPC request/reply interaction is done via a Service, which “is defined by a pair of messages: one for the request and one for the reply. A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply. Client libraries usually present this interaction to the programmer as if it were a remote procedure call” [1].

Finally, global configuration parameters can be shared via a **PARAMETER SERVER**. “A parameter server is a shared, multi-variate dictionary that is accessible via network APIs.

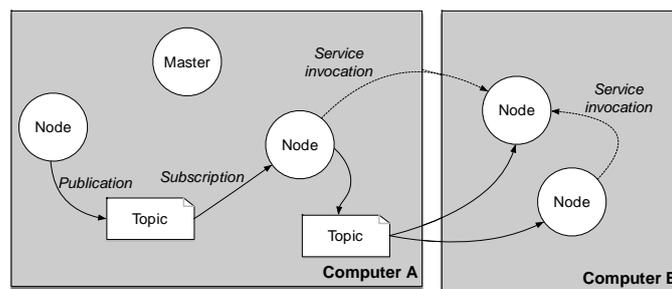


Figure 2. Basic ROS architecture



*Nodes use this server to store and retrieve parameters at runtime. (...) It is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary” [1].*

The whole system is governed by a **MASTER** node. This “*provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. The Master also provides the Parameter Server” [1].*

In the following sections, we assume the reader has basic knowledge of the UML modelling language. The Appendix provides a basic guide to the main UML concept used. For a more complete UML reference we refer the reader to [2].

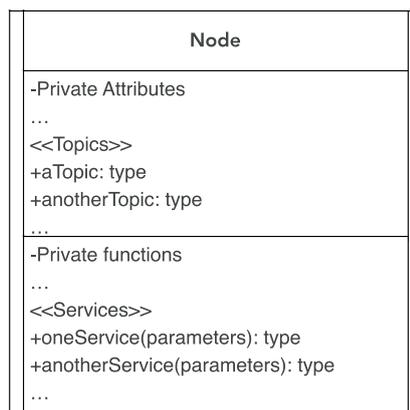


## 2 UML for ROS in UNEXMIN

This section formulates a proposal for the use of the UML style to model ROS systems. Such proposal constitutes the formal common “language” that the partners will use to talk to each other. It is intended for a smooth common understanding at a high level, as well as for a smooth integration of software module developed by different development teams. Its use is also proposed (but not enforced) for internal/informal communications. UML diagrams should have sufficient information associated to nodes, interfaces and communications to allow these to be translated in a coherent, complete and unambiguous way into software modules (mainly in C/C++ programming languages).

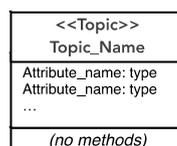
### 2.1 Nodes

ROS nodes are modelled as UML Classes. Since nodes are processes, they are modelled specifically as “*active classes*”. Each Node has a unique name, a set of attributes (internal variables) and a set of methods, that implement internal functions and public services. Published Topics are specified in a compact form as public attributes, grouped under the stereotype <<topics>>. Services are specified as public methods, grouped under the stereotype <<services>>.



### 2.2 Topics (extended version)

In case of need of more details, topics can be expressed as “stereotyped” classes (<<Topic>>) associated to the publish/subscribe association between nodes. Topic attributes model the data published. Additional information can be added as “properties” (labels between braces, e.g.,: -Image{640x480}, +size:int {positive only} ).

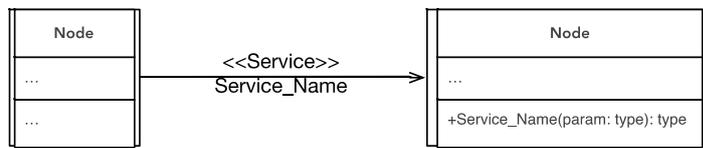


### 2.3 Services

Service requests are modelled as methods invocations, via a stereotyped association (<<Service>>). Note that the direction of the association goes from the caller to the provider, since it is a method call. The result of the request is given as return value of the service method.



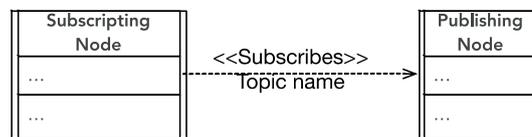
In other words, the requesting node sends a request to the service provider, and this sends back a result as a return value. A service request can have one or more parameters. Return type and parameters are specified in the method declaration in the corresponding class section.



## 2.4 Subscription/publication (compact form)

Topics publication is simply implied in the definition of the Node (as public attribute, under the stereotype <<Topics>>).

Subscription is expressed in a compact form as a stereotyped dependency (<<Subscribes>>) of the subscribing node on the publisher Node, as in the figure below:

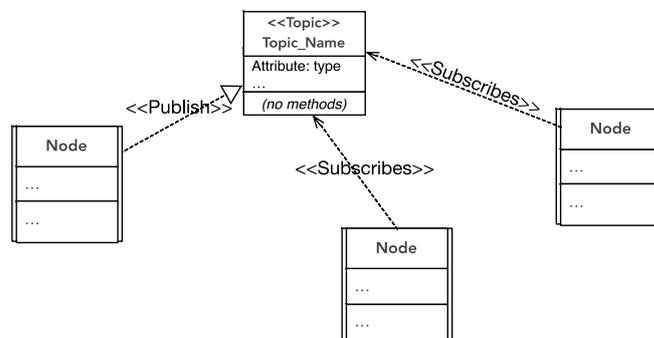


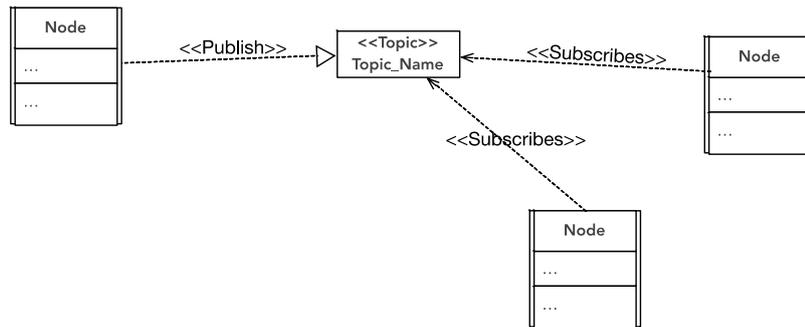
Note the direction of the arrow: from the subscribing node to the publishing node (the former depends on the latter).

## 2.5 Subscription/publication (extended version)

Subscription and publication of Topics by Nodes is modelled in details as a “realization” relationship between the publishing Node and the Topic, and as a dependency of the subscribing Note from the Topic. Such relationships will be stereotyped respectively as <<Publish>> and <<Subscribes>>. Note, again, the direction of the arrow: form the subscribing node to the Topic (the former depends on the latter). This system is similar to the one used for the specification of interfaces.

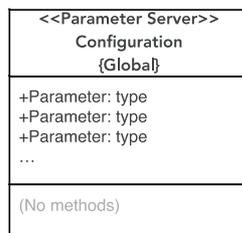
Relationships stereotypes can be omitted in case the relationships kind is evident from the context. Also, as common in UML, class details (e.g., topics attributes) can be omitted for simplicity.





## 2.6 Parameters Server

This is modelled as a class, with obvious stereotype <<Parameter Server>>. Its parameters are modelled as attributes of the class. Note the “{Global}” label, indicating that it is a global variable of the system (all nodes have access to it).



### 3 Main software organization

The diagram below describes the main organization of the software, in terms of software components (collection of SW modules). Components are color-coded according to the responsibility for their development: UPM (CAR), TUT, INESC TEC, RCI.

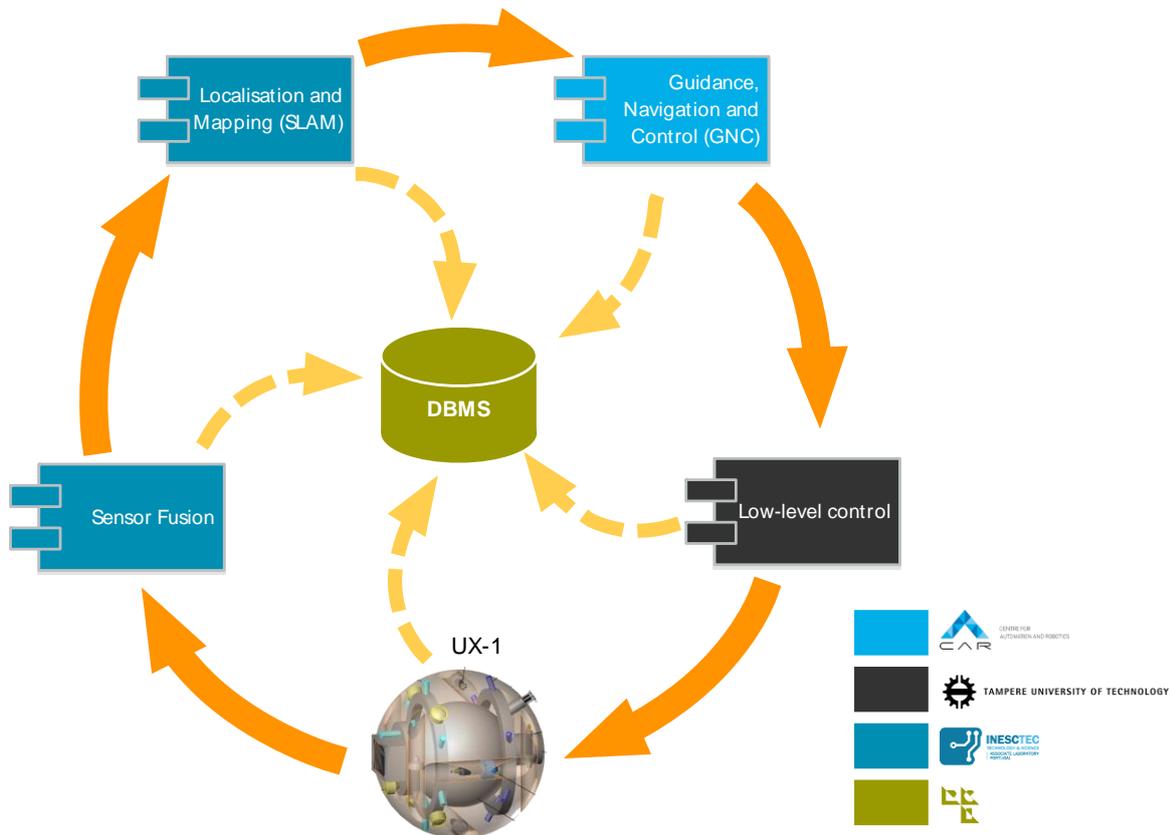


Figure 3. Components view of the software system

In brief, sensors data is processed by the “Sensor Fusion” and SLAM software, which provide data for the navigation and control software. This is in charge of generating control commands, which are sent to the low-level control software to the actuators. The dashed arrows represent a data logging component which receives data from all the other components, converts it to a standard format and transfers it to a database management system for offline post-processing after completion of each mission. This will be explained in detail in D6.1-3.

A more detailed and more formal architecture of the robot control system is shown in the following sections. Note that this document is focussed on the robot autonomous navigation systems. For details about scientific instrumentation and data management see D2.2 and D6.1-3.



## 4 Detailed diagrams

### 4.1 Components relationships

The following diagram depicts more in detail the interconnection between the for main software components, in terms of their *interfaces*. In the next subsections, detailed diagrams for each of the modules are presented. Their interfaces are described in Section 5.

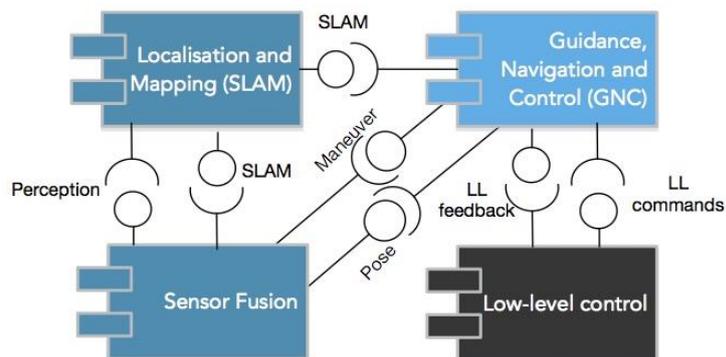


Figure 4. Components and their relationships



## 4.2 Sensor Fusion

This module will be responsible for providing pose and perception, through components to the localization and mapping module. Internally the sensor fusion is composed by three modules:

- Pose Estimation
- Sensor Perception
- Data Acquisition and Registration: responsible for sensor acquisition of the following sensors
  - o Multibeam M3
  - o SLS
  - o Scanning Sonar
  - o DVL
  - o INS
  - o Instrumental Sensors (PH, Water Sampling, Magnetic Field, EC and Subbottom Profile)
  - o Multispectral Camera

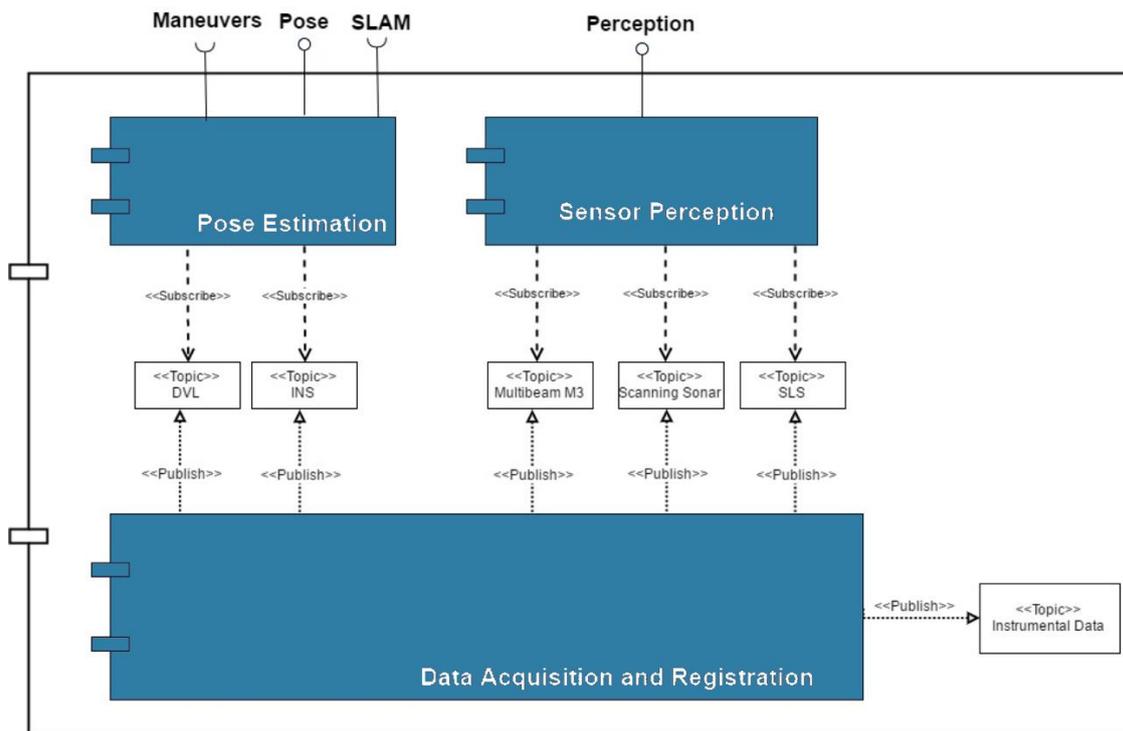


Figure 5. Sensor Fusion



### 4.3 Localisation and mapping

This module is responsible for receiving the Pose and Perception information (pointCloud, features and localization data) provided by the Sensor Fusion module. Internally the localization and mapping is composed by two modules that share the information through Map Data interface:

- SLAM: responsible for providing corrected map data and global localization;
- Mapping: responsible for managing the maps structures (local, topologic and semantic);

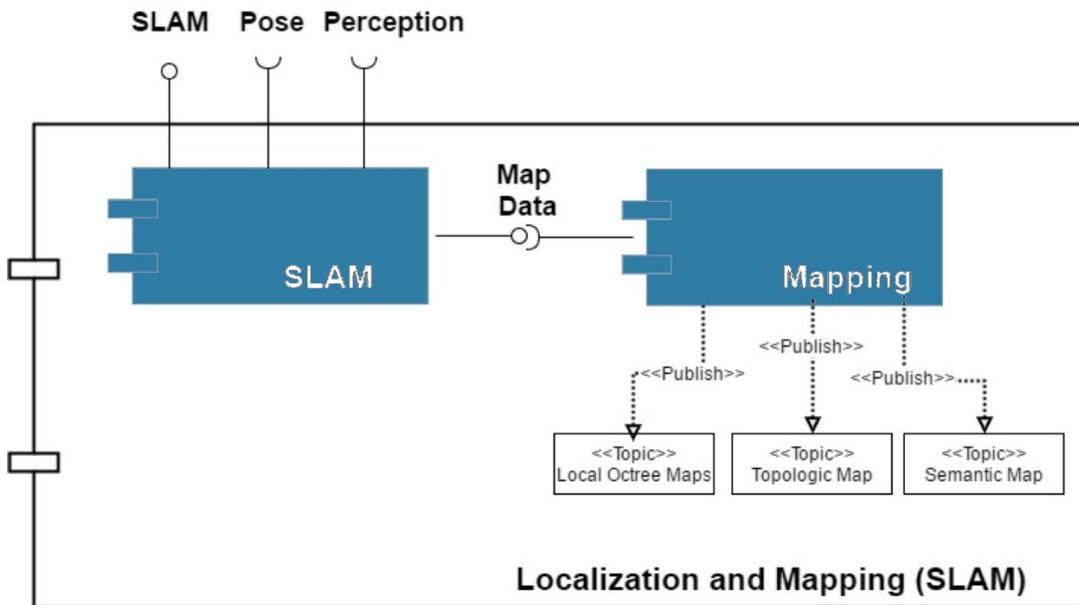


Figure 6. Localisation and mapping module



### 4.4 GNC

The GNC module has three main components: Planner, Guidance and Navigation. The first is the highest level module in charge of deciding the overall movement strategy, and send actions (e.g., turn, stop, go forward, ...) to the Guidance module. This computes trajectories in form of waypoints. Waypoints are fed to the Navigation module which generates velocity profiles that are finally transformed into control commands. The Control module publishes the control commands which constitute the “LL commands” interface. Note the dependencies of the Guidance module from the localisation and mapping information (SLAM and Pose interfaces). These three modules depend on the topological map which is created/updated during the mission. Also, the Planner may take into account a previously generated map and/or other pre-defined mission requirements.

Finally, the Data logging and conversion module is responsible of writing all the relevant data to the log database.

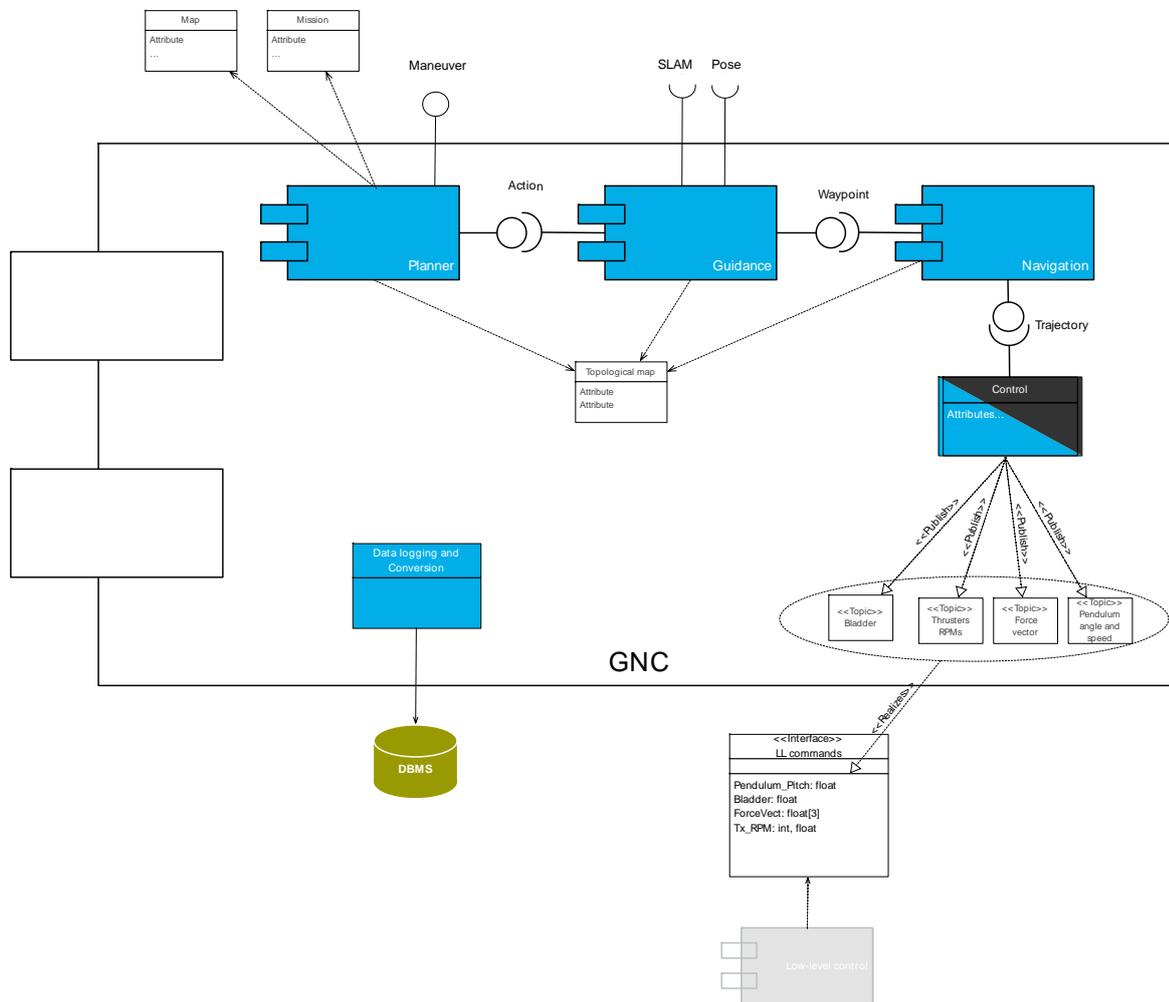
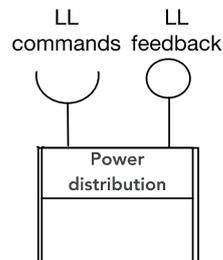


Figure 7. GNC component: main modules



## 4.5 Low-level control

From a Software Engineering viewpoint, the low-level software organization is relatively simple, since it will consist in two monolithic programs running, one in each microcontroller (see also Sec. 8, Computational Hardware). The following diagram illustrate the low-level software.



According to the commands received (see **LL commands** interface), the microcontrollers will distribute the power requirements to 8 CAN data frames which includes the NOD ID number of each thruster- Additionally, they will send commands to the pendulum's stepper motor and to the ballast motor.

All feedback from the thrusters (EPRM, duty cycle, force and current) will be sent through the microcontroller to the main CPU (see **LL feedback** interface). The microcontroller also monitors the power used by each motor in order to detect any malfunction.



## 5 Component interfaces

Components' interfacing is implemented via the topic publishing/subscription mechanism of ROS or via RPC (service invocation). In the following, the details of such interfaces is reported.

***It is important to highlight that ROS provides a set of “standard” topics (e.g., geometry messages, sensor messages, diagnostic messages,...). We will adopt such standard topics format whenever possible.***

***It is also important to highlight that all interface data will be time-stamped and fed to the “Data logging” subsystem.***

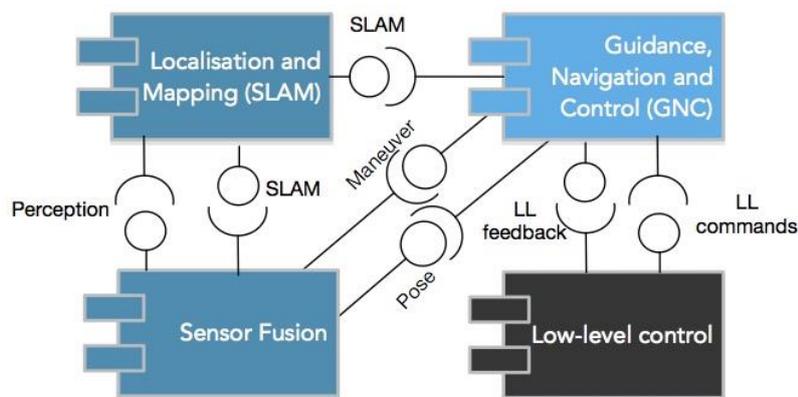


Figure 8. Components' interfaces

### Interfaces

1. **SLAM** – localization and mapping information for GNC
2. **Pose** – pose information (position and orientation) as computed by sensor fusion algorithms
3. **Low-level commands** – control commands for low-level controllers
4. **Low-level feedback** – feedback from the actuators
5. **Perception** – fused localization information, point cloud data and extracted features information
6. **Maneuvers** – maneuvers requests for perception purposes



## 5.1 SLAM interface

Name	Input parameters	Units/ comments	Output parameters	Units/ comments	Kind
Octree_map	None	--	theOctree	Featured map (ROS Octomap format)	Topic
PointCloud	--	Raw data to be processed by GNC in case of need	--	--	Topic
absLoc	None		theLoc: float[3]	Estimated X,Y,Z w.r.t deployment point	Topic (geometry_msgs/ PointStamped)

## 5.2 Pose interface

Name	Input parameters	Units/ comment s	Output parameter s	Units/ comments	Kind
Pose	None	Meters, degrees	thePose: float[7]	7-dimensional array of floating point numbers (x,y,z+quaternion for orientation (see Section “Reference System“ below)	Topic (geometry_msgs/ PoseStamped)
dotPose	None	Meters, degree /sec	dotPose: float[6]	6-D velocity	Topic (geometry_msgs/ TwistStamped)
dotdotPose	Note	Meters, degrees /sec <sup>2</sup>	dotdotPose: float[6]	6-D acceleration	Topic (geometry_msgs/ TwistStamped)
Instrument s status	--	--	--	Nav. instruments status	Topic (diagnostic_msgs / Diagnostic Status.msg)



### 5.3 LL commands interface

Name	Input parameters	Units/ comments	Output parameters	Units/ comments	Kind
Pendulum_Pitch	--	--	Angle: float	Degrees	Topic
Bladder	--	--	B: float	Force (newtons, positive or negative buoyancy)	Topic
Force vector	--	--	float[3]		Topic
Tx_RPM	--	--	T=1..8  RotSpeed: int32	Thruster No. (see Section “Reference System“ below)  RPMs (<0 means backwards)	Topic

### 5.4 LL feedback interface

Name	Input parameters	Units/ comments	Output parameters	Units/ comments	Kind
Pendulum	--	--	pitch_angle: float	Degrees Actual pendulum pitch angle	Topic
Bladder_stat				Actual %of filling	Topic
Tx_RPM	--	--	T=1..8  ERPM: float	Thruster No. (see Section “Refence System“ below) RPMs (actual RPMs)	Topic
Thruster status			int	Error code to be defined	Topic
Battery status	--	--		Battery level and status	Topic



## 5.5 Perception interface

Name	Input parameters	Units/ comments	Output parameters	Units/ comments	Kind
fPose	None	Corrected Pose after sensor fusion	7-dimensional array of floating point numbers.	X,Y,Z position + quaternion for orientation	Topic
Features	none	Features extracted from RAW sensor (Multibeam, SLS and Scanning Sonar)	type: int feature_parameters: array(float)	type, features_parameters	Topic
SPointCloud	--	RAW PointCloud SLS	--	--	Topic (sensor_msg/PointCloud)
MPointCloud	none	RAW PointCloud Multibeam			Topic (sensor_msg/PointCloud)

## 5.6 Maneuvers interface

Name	Input parameters	Units/ comments	Output parameters	Units/ comments	Kind
Pitch	Angle: float	Degrees	Actual pitch angle: float		Action
Yaw	Angle: float	Degrees	Actual Yaw angle: float		Action
Thrust	Speed: float	Meters/sec	Actual speed: float		Action
Forward	Amount: float	Meters	--		Action
Heave	Amount: float	Meters	--		Action
Sway	Amount: float	Meters	--		Action



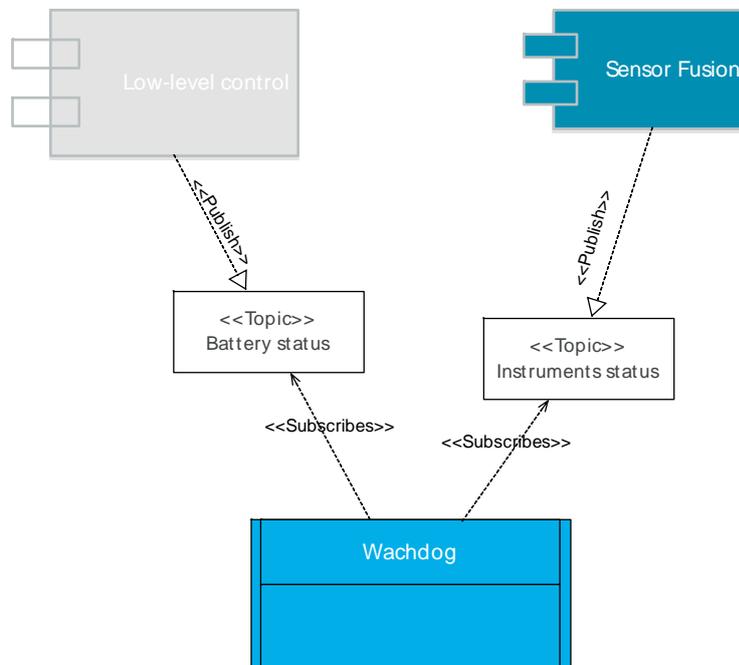
## 6 Parameters server (global system variables)

No global parameters have been defined yet.



## 7 Other system's elements

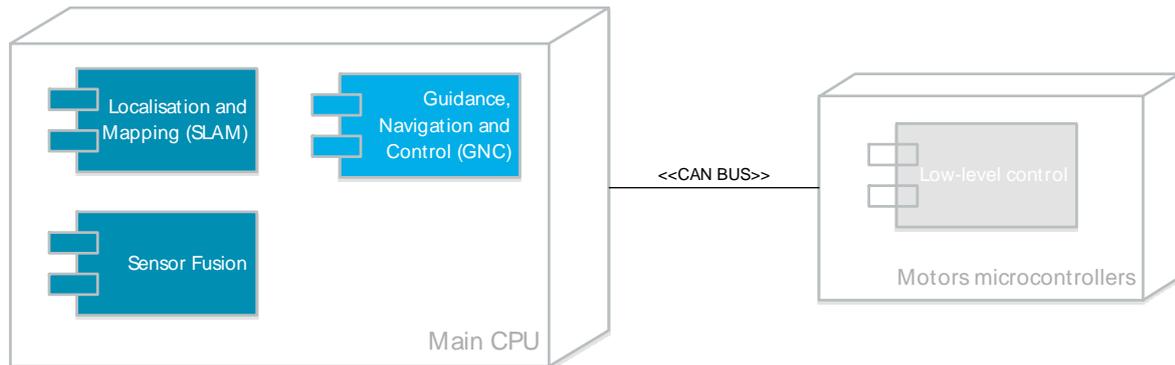
The watchdog, or Supervisor, is a SW module of the GNC component. Its purpose is to monitor both hardware and software subsystems, in order to monitor their proper functioning and take appropriate measures (e.g., restart software modules that has crashed, calibrate control actions according to the available actuators and their performance, establish reckon manoeuvres and even decide if/how to continue with the mission).



## 8 Hardware

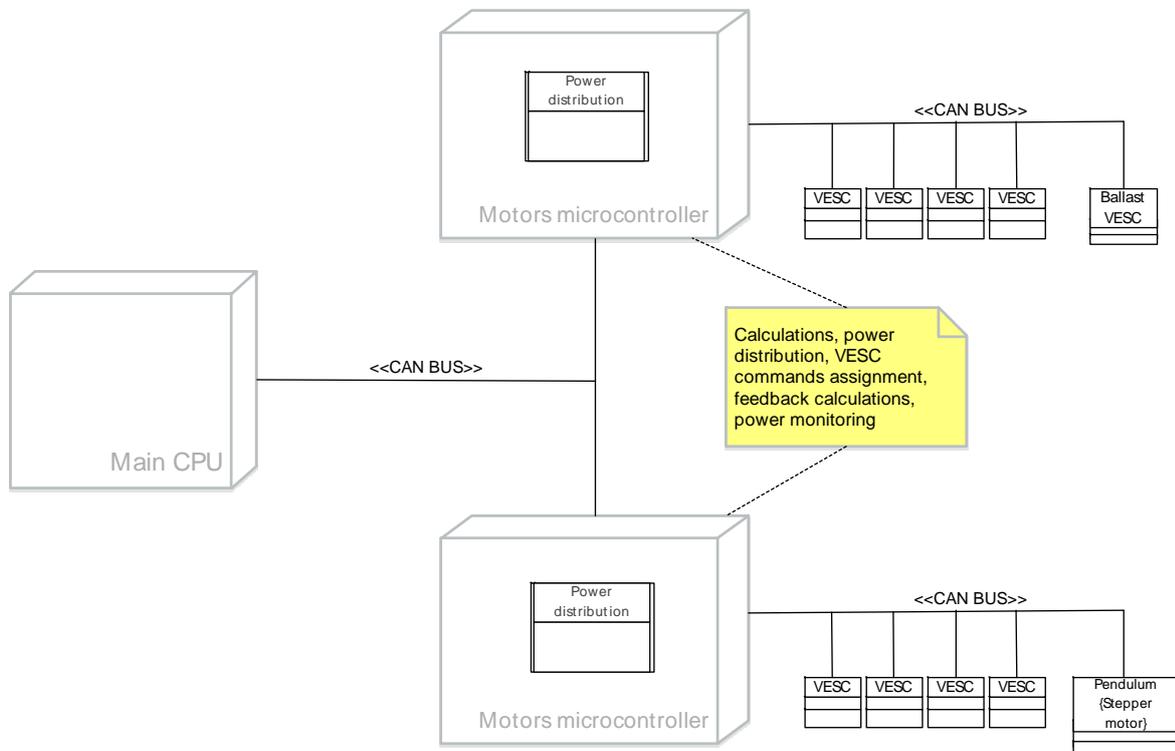
### 8.1 Computational Hardware

The following UML deployment diagram depicts the computation devices that will be adopted, and specifies where the various software modules will be actually executed.



As shown in the diagram, the UX-1 is equipped with a CPU board that runs all the perception, navigation and control software. The main CPU is connected via a CAN bus to the microcontrollers that directly control the actuators of the robot.

More in details, the hardware is organized as shown in the following diagram.



Two microcontrollers will be employed for the low level control system, which receives data (e.g., a desired force vector) from the main CPU. Each microcontroller is responsible to communicate with the speed controllers (VESC).

### 8.2 UX-1 Hardware Architecture

In order to support the software architecture that will detail in this report, the proposed UX-1 hardware and communications architecture is presented in the following diagram. Notice the Master Clock used to sync the devices (red arrows).

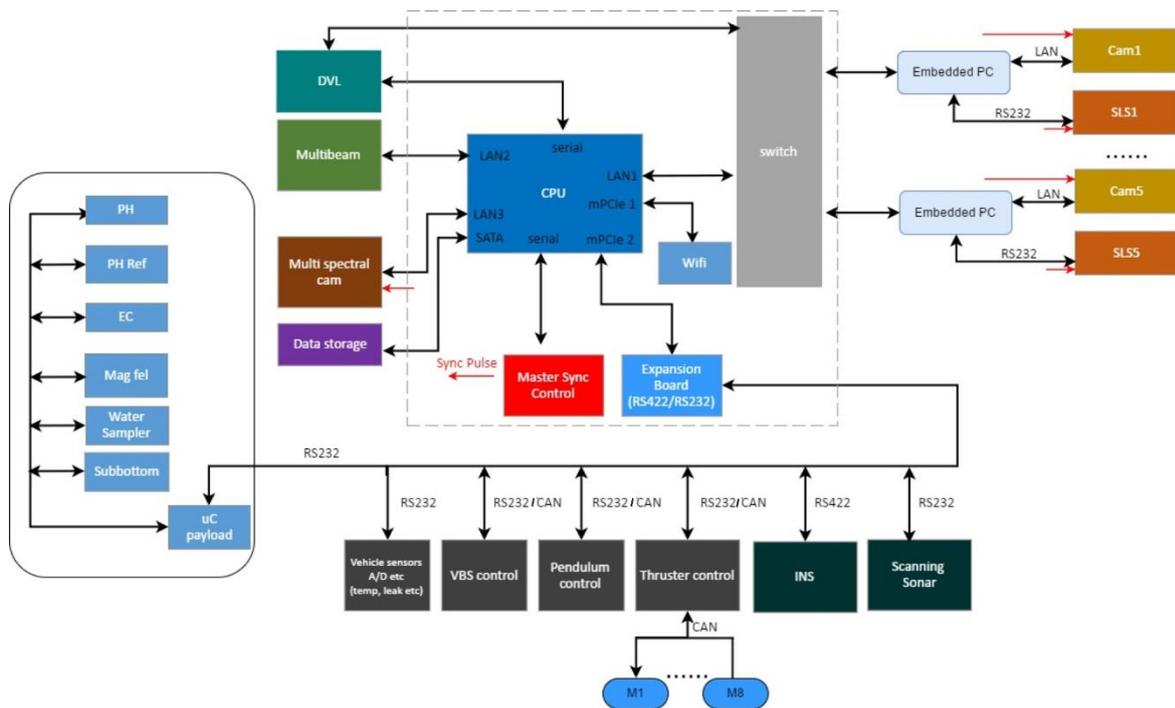


Figure 9 - UX-1 Hardware Architecture



## 9 Reference system

The following image (from D1.3) depicts the thrusters numbering as well as the reference frame of the robot.

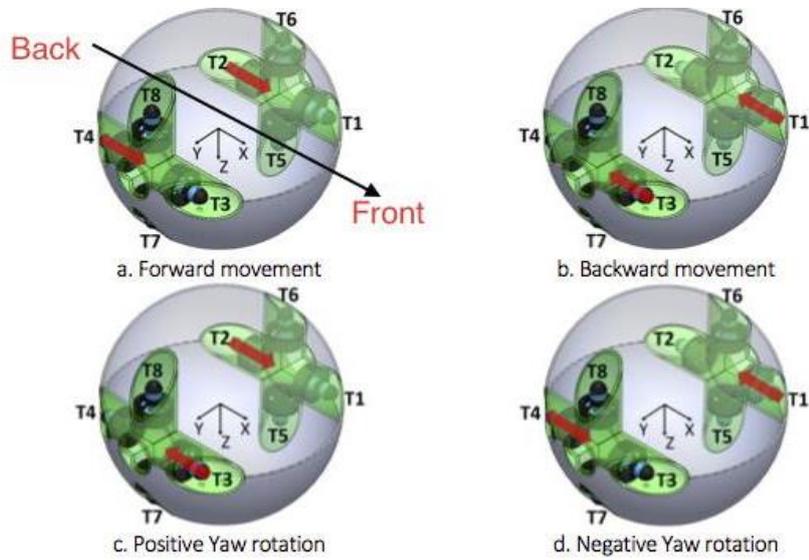


Figure 10. Reference system and thrusters numbering



## 10 Software versions

We will use the following versions of software:

- **Operating system:** Linux UBUNTU distribution 16.04
- **Middleware:** ROS version Kinetic Kame
- **Simulation environment:** Gazebo version 7.0.0
- **GITLab repository, version control and collaborative software development** (the GitLab is provided by INESC TEC in the following link address: <http://www.lsa.isep.ipp.pt:8000>)



## 11 References

[1] [wiki.ROS.org](http://wiki.ros.org)

[2] Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide (2nd Edition), Addison Wesley, 2005.

[3] <http://www.digilife.be/quickreferences/qrc/uml%20quick%20reference%20card.pdf>

[4] <https://www.holub.com/goodies/uml/>



## 12 APPENDIX: UML quick reference

In the following, we briefly describe the elements used and their main features. Quick UML reference and cheat sheets can be found, e.g. in [3][4]. The UML “classical” –and recommended– reference book is [2].

Here, we we are mainly concerned with SW architecture, so will mainly use 3 kinds of UML *structural* diagrams:

- Class diagrams
- Components diagrams
- Deployment diagrams

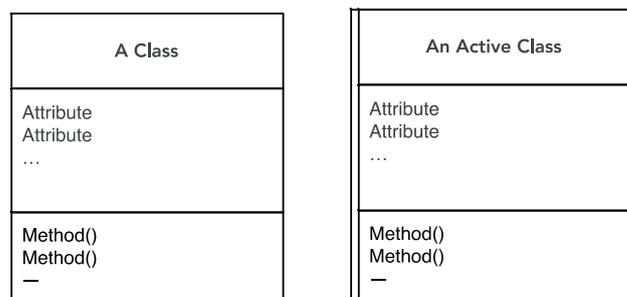
For the description of the behaviour of the system (its dynamic aspect, i.e., its evolution in time) UML provides:

- Sequence/communication diagrams
- Use cases,
- State Machine diagrams

These will be used in a latter stage of the development.

### 12.1 Class diagrams

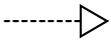
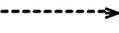
Class diagrams are used to model the **static** (or **structural**) part of the system. A **class** is an element used model any kind of entities of the problem/solution. We will use classes to model Nodes, Topics and other elements. A class is specified by its name, a set of **attributes** (internal variables and parameters) and a set of **methods** (functions that the class can perform). In particular, **active classes** are used to model “flows of control, i.e. **processes** and **threads**. These will be used for the ROS nodes.



Classes have **relationships** with each other. These can be generalization, association, dependency, realization and aggregation.

Symbol	Relationship	Use
	Aggregation/Specialization	Models a “KIND OF” relationship. Used to define new classes starting from existing ones by means of adding specific details. For instance, a LASER is a kind of SENSOR.
	Association	Associated classes communicate to each other. Typically, associations are used to send messages/invoke methods.



	Aggregation	Models a “PART OF” relationship. For instance, the PLANNER is a part of the NAVIGATION class.
	Realization	Specifies that an element realizes (implements) a certain function.
	Dependency	Expresses a functional dependency (e.g. use).

Note that classes are *abstract* structural elements. Their concrete implementation, or *instances*, are named **objects**. Objects are used to specify the dynamics of the system and of its components (see sequence diagrams below).

Finally, classes are also used to model physical element of a system, like electronic components (except for computational resource, which are modelled as “nodes, see below).

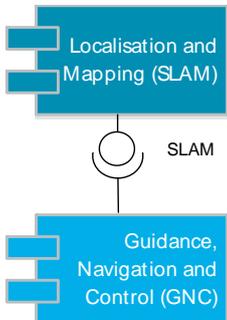
## 12.2 Components diagrams

Simplifying, a **component** is a part of a system that groups a set of functionalities, and are used for modularity. We will use components to group software modules in terms of the high-level function they perform (see, e.g., *Figure 3*) and/or the partner responsible for them. An example is shown below.

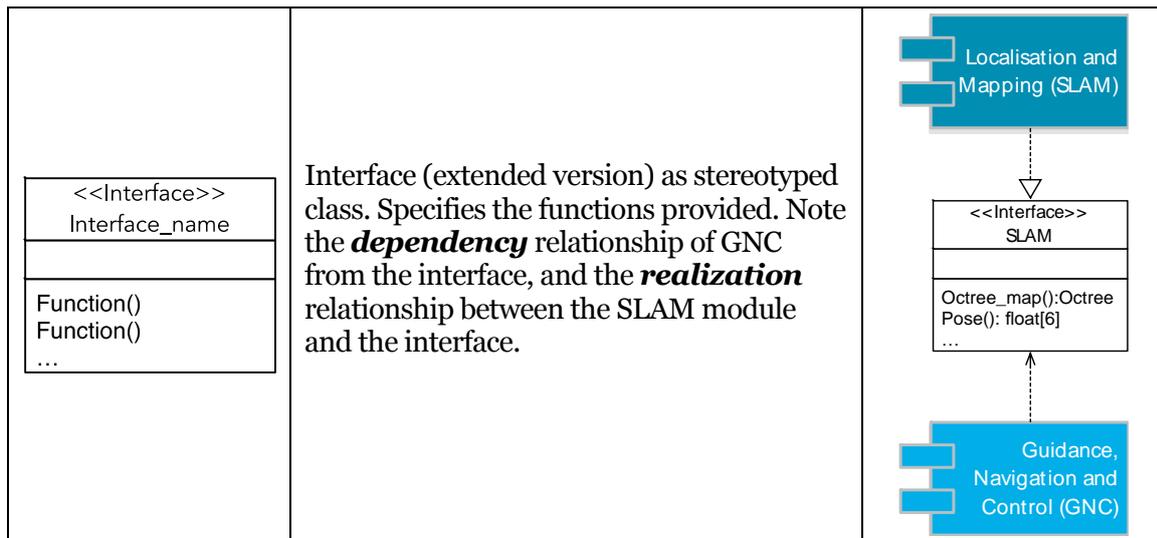


Components are characterised by the functionalities they provide, which are described in terms of **interfaces**.

An **interface** is a collection of operations that are used to specify a service implemented by (an element of) a component. Interfaces define the separation between system’s modules, and provide a clear separation between the component’s inside (how the service is implemented) and outside (what the service does) [2].

Symbol	Meaning	Use
 Name	Output interface (compact version)	
 Name	Input interface (compact version)	





### 12.3 Deployment diagrams and nodes

Deployment diagrams specify where the software modules are executed. Such diagrams contains **nodes** (not to be confused with ROS nodes) and their relationships. A UML node is physical element that represents a computational resource (e.g. a computer, a CPU, a microcontroller). Communication between nodes is represented as an association relationship. The communication protocol or other features can be expressed with a stereotyped association (e.g.: <<Ethernet>>).

Symbol	Meaning	Use
	Node	A computational resource

### 12.4 Modelling system’s behaviour

- **Sequence Sequence/communication diagrams**  
 Sequence diagrams are one of the diagrams that are used to model the *dynamic* behaviour of the system. They consist of a class of **objects** (class instances, i.e. concrete system elements) and specify, in a time-ordered way, the messages these exchanges.
- **Use cases**  
 Use cases specifies a high-level functionality of the system (or sub-system). It involves an interaction with an “*actor*“, the user of the functionality, either a human or another (sub-) system.
- **State diagrams**  
 State charts are one of the diagrams that are used to model the *dynamic* behaviour of a component or subsystem. They are used mainly to model an element’s life.



## 12.5 Common mechanisms

Additional semantic can be added/specified for the elements of a diagram using **stereotypes**, **properties** and **notes**.

- <<Stereotype>> A stereotype is a label that is added to UML elements to create new, system-specific elements whose semantic is “similar” to the original element. We will use stereotypes to create the Topic class (expanded mode) and to specify Topics and services of the Nodes.
- {Properties}. A property is a free-text label that specifies some property of the element it is associated with.
- Finally, a Note is a visual element that contains any comment and remark that one may need to do on a specific element, group or diagram.

Symbol	Example	Use
<<Stereotype>>		Creates a new element extending an existing one's semantic
{Property}		Specifies some characteristic of the element
		Comments, remarks, clarifications, ...

