Implementation of a Control Architecture for Networked Vehicle Systems ⋆

José Pinto, Pedro Calado, José Braga, Paulo Dias, Ricardo Martins, Eduardo Marques, J.B. Sousa

Department of Electrical and Computer Engineering, University of Porto, Portugal – 4200-465.
Email:
zepinto,pdcalado,jose.braga,pdias,rasm,edrdo,jtasso@fe.up.pt

Abstract: This paper describes the layered control architecture and its software implementation developed and used at the Underwater Systems and Technology Laboratory. The architecture is implemented as a toolchain which consists on three main entities: DUNE onboard software, Neptus command and control software and a common IMC message-based communication protocol. The LSTS software toolchain has been tested throughout various field deployments where it was used to control heterogeneous autonomous vehicles like AUVs, ASVs, UAVs and ROVs in both single and multi-vehicle operations.

Keywords: Software Toolchain, Autonomous vehicles, Multi vehicle control, Marine systems, Communication Protocol, Inter-Module Communication

1. INTRODUCTION

The Underwater Systems and Technology Laboratory (LSTS) aims the creation of networked vehicles systems constituted by human operators, heterogeneous autonomous vehicles and other sensing devices. The networks composed by these systems are dynamic, in the sense that both vehicles and operators come and go. Vehicles have limited communication range and, as they move, communication and control links are created and destroyed at runtime. By moving, vehicles can function as mobile sensing and communication devices, eventually working as mules of information and retrieving data from remote locations of the world as seen in the example scenario of figure 1.

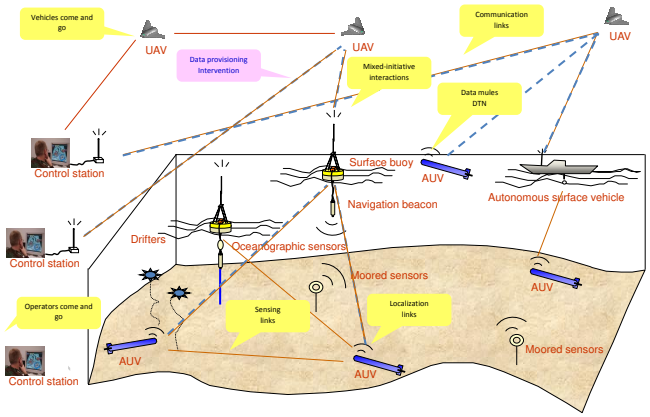

Fig. 1. LSTS networked vehicle systems concept.

⋆ The research leading to these results has received funding from the European Commission FP7-ICT Cognitive Systems, Interaction, and Robotics under the contract #270180 (NOPTILUS), and FCT COMPETE under the contract #PTDC/EEA-CRO/104901/2008 - (PERSIST).

LSTS has already built autonomous vehicles of different types, namely Autonomous Underwater Vehicles (Madureira et al. (2009)), Autonomous Surface Vehicles (Ferreira et al. (2007)), Unmanned Air Vehicles (Gonçalves et al. (2009)), and Remotely Operated Vehicles (Gomes et al. (2005)). Some of these vehicles can be seen in figure 2.

In order to control these vehicle networks, it is necessary to create software abstractions and protocols that can be reused across the different devices for single and cooperative operations. Its implementation must account for the following needs:

(1) manage the on-board vehicle sensors and actuators, and their use for autonomous navigation and control
(2) standardized communication among vehicles, communication gateways and operator consoles
(3) graphical interfaces for operator-vehicle interactions, including mission planning and analysis.
(4) flexibility to adapt existing solutions and controllers to new, still unforeseen robotic devices

To fulfill these requirements, we have developed different tools that comprise the LSTS software toolchain: DUNE onboard software, Neptus command and control software and the IMC communications protocol.

DUNE: Unified Navigational Environment is the runtime environment for vehicle on-board software. It is used to write generic embedded software at the heart of the vehicle, e.g. code for control, navigation, communication, sensor and actuator access, etc. It provides an operating-system and architecture independent platform abstraction layer, written in C++, enhancing portability among different CPU architectures and operating systems.

The *Inter-Module Communication* (IMC) protocol, a message-oriented protocol designed and implemented for

communication among heterogeneous vehicles, sensors and human operators. DUNE itself uses IMC for in-vehicle communication (Martins et al. (2009)).

Neptus is the command and control software used by human operators to interact with networked vehicle systems (Dias et al. (2006)). Neptus supports different phases of a mission's life cycle: planning, simulation, execution, revision and dissemination (Pinto et al. (2006)). Concurrent multi-vehicle operation is possible through specialized graphical interfaces, which evolved through the years according to the requirements provided by end-users.

Fig. 2. LSTS autonomous vehicles. From top left to bottom right: Swordfish ASV, Adamastor ROV, LAUV and Antex X03 UAV.

Similarities exist between this toolchain and the Robot Operating System (ROS), Quigley et al. (2009), in the sense that they both try to accomplish similar goals. However, some aspects can tell them apart:

(1) Neptus provides configurable interfaces that can be adapted for each type of autonomous vehicle, while ROS has a single interface for all types of agents (ROS visualization tools).
(2) Neptus has been tested in the field numerous times, adopting feedback from different end-users with academic, industrial and military backgrounds.
(3) DUNE runs on a very small footprint (16 MB) and was developed having embedded processors, with limited capabilities, in mind. Which also makes cross-compiling very straightforward. Cross-compiling ROS demands some added effort (using eros for full cross-compilation).
(4) DUNE can run on an operating system that has no processes, such as RTEMS or eCos.
(5) On the other hand, ROS is open source and has a contributing community helping to expand the toolchain.

2. CONTROL ARCHITECTURE

In our view of networked vehicle systems, these are composed by multiple components like vehicles, sensors, controllers, human operators, operator consoles, communication devices, etc. In order to cope with all these different network nodes, we use a layered approach in the control of these systems and establish common interfaces for communication and coordination between components, as seen in figure 3. Each layer encapsulates lower-level details and, at the same time, provides interfaces for retrieving state and accepting commands.

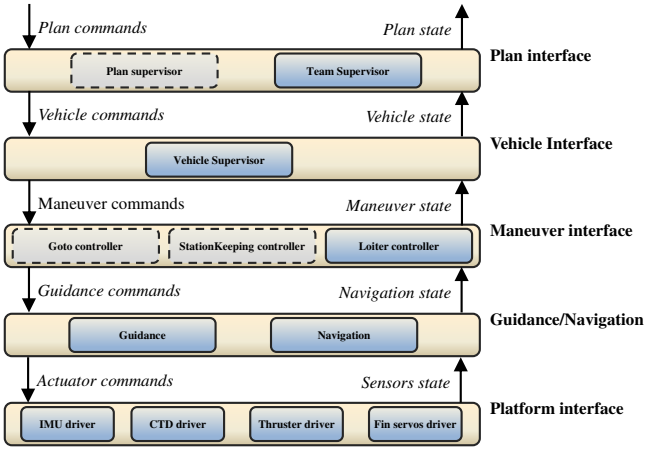

Fig. 3. Example architecture implementation and possible switching between active/inactive controllers.

All vehicles provide a *platform* with sensor and actuator low-level interfaces that are used by *guidance* and *navigation* software components. These components abstract specific hardware details by providing standardized sensor data together with a common command set for controlling the desired vehicle behavior.On top of guidance / actuation components, *maneuver* controllers receive current vehicle state (produced by the navigation) and generate intended behavior by producing guidance commands.

Maneuver controllers are instantiated and/or terminated by a *vehicle supervisor*. The vehicle supervisor continually verifies that the system is working properly and instantiates maneuver controllers according to requested maneuver specifications. Prior to instantiation, the vehicle supervisor may check if it is safe to execute a given maneuver according to current vehicle state (battery levels, hardware faults, etc) and may also terminate maneuver execution in the event of hardware failure or any safety violations.

The vehicle supervisor receives maneuver specification commands from upper layers. These can be either a *team controller* that commands maneuver execution in multiple vehicles (by sending commands through network links), or it can also be an on-board *plan supervisor* that, according to a plan specification, triggers the execution of maneuvers in the vehicle. Plan supervisors can use imperative plan specifications or they can also be deliberative planners that, from a set of specified high-level objectives, generate maneuvers that must be executed in order to fulfill the plan objectives. Plan specifications / high-level objectives can be created by human operators through operator consoles and then sent for execution.

Except for the hardware-specific platform layer, any other layer follows common interfaces and this fact allows us to have multiple instances of upper layer controllers. This provides flexibility to have possibly interchangeable and even migrating controllers at run-time.

In the following sections we describe the implementation of this architecture by the LSTS software toolchain. This software toolchain is composed by on-board software (DUNE), shore-side control software (Neptus) and a communication protocol which is shared by all components, the IMC inter-module communication protocol (section 5).

3. DUNE: UNIFIED NAVIGATIONAL ENVIRONMENT

DUNE is the on-board software running on the vehicle, which is responsible not only for every interaction with sensors, payload and actuators, but also for communications, navigation, control, maneuvering, plan execution and vehicle supervision.

It is CPU architecture independent (Intel x86 or compatible, Sun SPARC, ARM, PowerPC and MIPS) as well as operating system independent (Linux, Solaris, Apple Mac OS X, FreeBSD, NetBSD, OpenBSD, eCos, RTEM, Microsoft Windows 2000 or above and QNX Neutrino).

Thanks to its modularity and versatility, DUNE does not only run in our ASVs, ROVs, AUVs and UAVs, but also in our communication gateways, LSTS (2011).

3.1 Modularity

DUNE functions as a message passing mechanism where independent tasks run in a separate thread of execution. All these tasks are connected to a bus to which they can publish and subscribe to messages, that can be consumed or published by other tasks. An example of a task is, for instance, a sensor driver, that can publish a message containing information about the sensor being read. This information may later on be consumed by a task whose purpose is to get the vehicle navigating in three dimensional space (see figure 4). The same idea is valid for a task that works as a motor controller, or power consumption manager, and so on.

If a new sensor is installed, or a new controller is going to be tested, all that is necessary is to enable and disable some tasks. This high modularity makes life easier not only for the everyday developer, but also for a newcomer or temporary developer that will work on a certain module of the software. That person can be abstracted from the complexity of the remaining entities on the framework.

It is important to point out that the messages passed to the bus are specified in the LSTS communication protocol, the IMC (see section 5).

3.2 Profiles and Configuration

A task in DUNE may be common to more than one vehicle. The same task may be able to run either in an underwater vehicle or in an aerial vehicle, but configured in a different manner. The set of parameters that tune a task to function in a certain way are determined by the configuration scheme. These configurations can be changed easily with no need for software recompilation. It also allows the enabling and disabling of tasks.

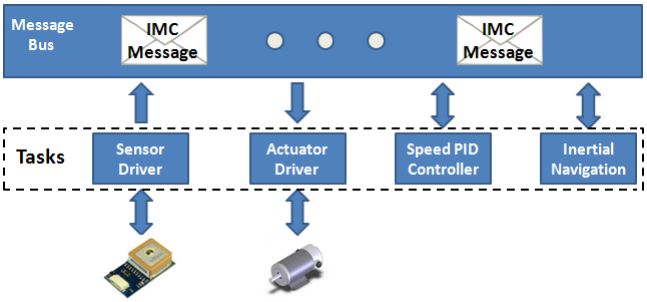

Fig. 4. Message passing concept behind DUNE tasks implementation.

Moreover, DUNE can run with different profiles, that by taking advantage of the configuration mechanism, enable and disable sets of tasks for that profile purpose. For instance, DUNE can run in Simulation mode, which disables all the sensor and actuator drivers and replaces them with simulating tasks. It may also run in Hardware-in-the-loop mode, which allows for some sensor or actuator drivers to be enabled, together with simulating tasks. These features are very important from a developing perspective, since they allow for "offline" task/feature testing and validation.

3.3 Implementation of the Control Architecture

The layers *vehicle*, *maneuver*, *guidance*, *navigation* and *platform* Interface pointed out in section 2, have been implemented using the DUNE framework. Therefore, all the interactions present in figure 3, such as Maneuver State, were implemented as messages, by using the message passing scheme described earlier. The interfaces themselves are represented by one or more tasks. For instance, the Maneuver Interface consists of a *Maneuver Supervisor* task, plus one task per type of maneuver (*Maneuver Controllers*). The *Maneuver Supervisor* task is always enabled during vehicle operation, but only one of the Maneuver Controllers is on, while the remaining are disabled.

4. NEPTUS COMMAND AND CONTROL INFRASTRUCTURE

Neptus software is used by operators to visually plan, simulate, monitor and review missions executed by autonomous vehicles. Neptus provides user interfaces to control vehicles of different types like AUVs, UAVs, ASVs and heterogeneous teams of the former simultaneously.

4.1 Mission Planning

In Neptus, a mission is specified as a set of maneuvers (each with a specific type and parameters) and transitions between those maneuvers, forming a graph. A maneuver is thus a unit of work that can be accomplished by a specific vehicle or a class of vehicles by instantiating a controller that potentially changes the physical state of the vehicle. A transition condition is a boolean expression that can be evaluated against the vehicle state or triggered by asynchronous events.

Since it is very common to create plans that are similar to others used in the past, Neptus also provides a templating mechanism which generates mission plans (for one or more

vehicles) based on parameters introduced by the user. This was implemented by creating a plan generation API and by creating javascript bindings for this API. New templates can be created by adding respective javascript plugins.

Mission planning in Neptus can also be done visually (figure 5). The graphical editor provides a map view of the mission site and maneuvers can be added to this map (setting possible location parameters) and edited. Moreover, connections between maneuvers and any specific maneuver parameters can also be edited in this graphical interface. To verify a mission plan, usually the operator previews its execution by using a simulator.

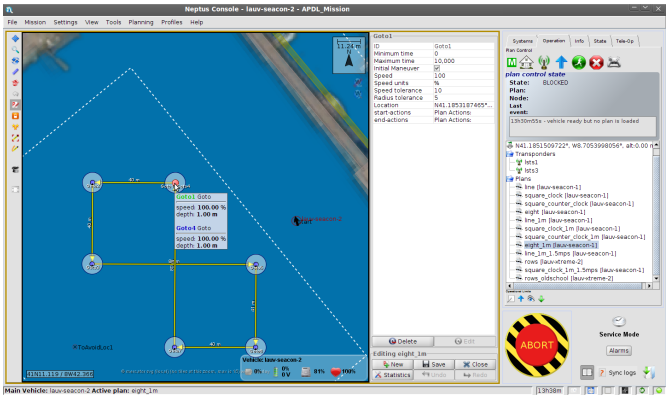

Fig. 5. Neptus console showing plan creation on top of the mission site map.

4.2 Mission Simulation

Neptus provides three different levels of simulation accuracy: behavior prediction, software simulation and hardware-in-the-loop (HIL) simulation. Software simulation is done by connecting to one or more simulated vehicles running DUNE in simulation mode (sensor values and actuations are simulated). Moreover, the simulators can also be running inside actual vehicles and real sensors / actuators may be used together with simulated ones for HIL simulation.

Software simulation and HIL simulation are usually employed for testing mission specifications and also to train personnel prior to real-world deployments. HIL simulation can also be used to test hardware in dry-run tests.

While a mission is being executed Neptus also provides rough behavior simulations whenever the vehicles are disconnected from the base. This is used to predict the state of the vehicles while they are at remote locations and thus aid the operator in managing the complex behavior of vehicle fleets.

4.3 Mission Execution

Mission execution is supported by Neptus through the use of operational consoles. From these consoles, operators can monitor vehicle execution, quickly change or create new plan specifications, send plans for execution, teleoperate vehicles, etc.

Operational consoles can easily be adapted for the operation of specific vehicles or to a specific mission scenario. In order to edit an operational console, users can chose from a collection of plugin components (widgets, daemons) and drag them onto the console frame. The components can furthermore be grouped into layout containers for preserving screen real estate. Optionally, users can create more than one presentation layout and store them as visualization profiles. Different profiles can be activated by the operator according to mission execution phases like deployment, operation, debug, recovery, etc.

Neptus operator consoles support multi-vehicle operations by displaying received data from all vehicles and allowing the user to switch between controlled vehicle. Since most of the time these vehicles operate autonomously, this allows the user to plan future missions while other missions are being executed.

For safe operation, Neptus provides an alarm framework composed by multiple daemons that continually monitor the data being received by the vehicles. The user is notified in events of interest like mission start, mission completion and failures through different sensory cues. Figure 6 shows the execution of an operational console used to control UAVs.

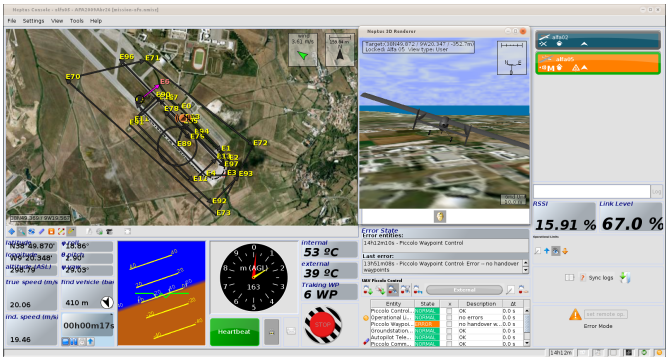

Fig. 6. Neptus operational console for UAVs.

4.4 Mission Review and Analysis

LSTS vehicles store mission data as serialized streams of generated and received messages. In order to be possible to inspect and analyse these data, Neptus provides a specialized application (Neptus MRA) that decompresses data into text files (that can be later imported to programs like Excel^{TM}or Matlab^{TM}) and provides several visualizations and utilities to process and export data.

According to available mission data, different plots will be rendered like vehicle estimated position, acoustic ranges, Euler angles, conductivity, salinity, etc. Moreover, all data is presented to the user in the form of table and any combinations of scalar fields can be plotted against time. Neptus MRA provides also specialized visualization plugins for viewing of side-scan data, log revision and colormaps (bathymetric, temperature, salinity, ...).

Since all messages are time-tagged by the generating system, MRA also allows the replay of mission data, allowing the visualization of the vehicle's execution of past missions. Replay data can also be fed back into Neptus operational consoles so that the entire mission can be visualized using the console visualization widgets.

Using a plugin framework, Neptus MRA can also be used to export mission data into different formats like comma-separated values or PDF reports. We are currently working on export plugins for NetCDF and KML file formats.

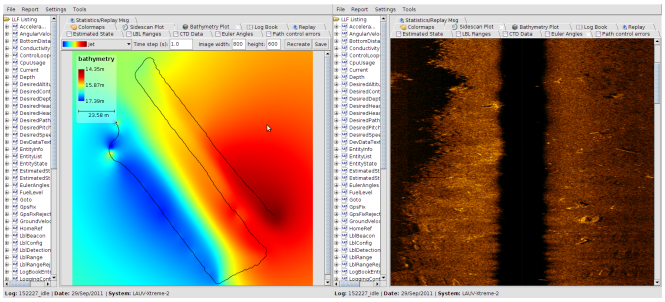

Fig. 7. Different Neptus MRA visualizations. On the left bathymetry colormap and, on the right a side-scan data plotter.

5. INTER-MODULE COMMUNICATION PROTOCOL

The IMC protocol (Martins et al. (2009)) is a message-oriented protocol targeting networked vehicle and sensor operations. IMC defines a common message set that is understood by all systems and used for communication between network nodes, DUNE tasks and Neptus plugins. IMC is fully defined and documented in a single XML file which can be translated into different language bindings using XSLT. Figure 3 depicts a typical communication flow among several layers of the control architecture inside vehicles.

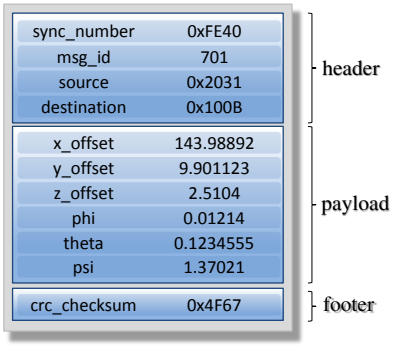

Fig. 8. Example IMC message structure.

This layered control and sensing infrastructure is in line with typical control infrastructures for autonomous vehicles (Gerkey et al. (2003), Quigley et al. (2009)) which enable modular development of robotic applications. Using IMC, software components can run in logical isolation, interfacing with other modules only through the exchange of IMC messages. Moreover, a common message set strictly defines the interfaces of the different types of components.

Networking of vehicles and consoles, is enabled through traditional IP-based communication-mechanisms, like raw UDP or TCP sockets, or by other means, such as the Real-Time Publish-Subscribe protocol (Marques et al. (2006)), or underwater acoustic modems (Marques et al. (2007)). IMC also has established serialization standards for JSON and XML which allows its use by any web-enabled devices and frameworks.

All IMC messages are divided in header, payload and footer. The header contains among other fields, a synchronization number which allows us to detect different byte order serializations and/or protocol versions; a message identifier, a source and a destination. The message payload varies according to the message identifier (as described by the IMC protocol specification) and can also include other (inline) messages recursively. To see a message example consult figure 8.

6. TOOLCHAIN DEPLOYMENTS AND DEMONSTRATIONS

The LSTS software toolchain has been the backbone of all our operations since, as previously stated, this toolchain guarantees communication among all our platforms and is used to define the behavior of the autonomous vehicles. During 2011, LSTS has successfully performed different sea-trials with multiple vehicles working cooperatively, amounting to a total of more than 100 hours of autonomous operation. This section describes two demonstrations of coordinated behavior achieved by the LSTS toolchain.

6.1 Multi-vehicle operations

From 12th July 2011, a two-week underwater experiment was carried out jointly with the Portuguese Navy near Sezimbra coast (Portugal). The experiment, designated Rapid Environmental Picture 2011 (REP 2011), aimed at demonstrating the use of multiple LSTS vehicles to acquire sidescan imagery simultaneously, while sharing information among them.

Two underwater vehicles (LAUV-Seacon and LAUV-Xtreme), the Swordfish ASV and unmanned aerial vehicles (Cularis UAV) were operated simultaneously from the *Bacamarte* navy ship.

Most notably, one Cularis UAV was used as a data mule to gather information from a floating LAUV-Seacon, afterwards delivering these data back to the control room aboard the ship.

6.2 Cooperative maneuvers

On 7th July 2011, LSTS has carried out a series of tests demonstrating cooperation between autonomous vehicles using solely acoustic communication links (Teledyne Benthos acoustic underwater modems).

In one of the experiments, nAUV was programmed with a trajectory following plan while Swordfish ASV was programmed with a plan that consists in following nAUV: whenever a position estimate is received via the acoustic modem, go to that position. As a result, Swordfish ASV roughly followed the plan execution of nAUV and, in the future, this behavior can be used to preserve a network link from air to underwater (ASV working as communication relay).

The second experiment consisted in having both vehicles pre-loaded with formation following maneuvers. These maneuvers are parameterized by a trajectory to follow and list of formation participants (with respective offsets to the

trajectory). Periodically all participants (Swordfish ASV and nAUV) share their state of completion with other vehicle(s) and, as a result, remaining participants can decrease/increase speed accordingly. This experiment successfully demonstrated two autonomous vehicles sharing information, such as their estimated states and the desired paths, to perform cooperative maneuvers (see figure 9).

Fig. 9. LSTS nAUV and ASV Swordfish performing cooperative maneuvers.

7. CONCLUSIONS AND FUTURE WORK

The proposed control architecture has been successfully implemented as the LSTS software toolchain and used across multiple devices for single and multi-vehicle deployments. Reusing tools like DUNE and Neptus, common to all vehicles and operator consoles, not only prevented duplication of effort but also allowed a faster development of new vehicles and controllers.

The same behavior abstractions (maneuvers and plans) have been successfully used to define the behavior of AUVs, UAVs, ROVs and ASVs. By using cooperative maneuvers and transition conditions in plans, we are able to also define multi-vehicle coordinated behavior.

Currently, we are working towards the inclusion of delay-tolerant networking functionalities in our vehicles and consoles. This will allow using the vehicles to actively extend the network range by functioning as data mules. This way, vehicles can carry both sensor data and commands between remote network locations.

In the near future, we plan to add required security mechanisms into our networks for a safer operation workflow. This involves handling varying authority levels, distinction between vehicles and payloads, explicit control links and handover of these links between operator consoles.

In another line of work, we are developing new planning mechanisms for controlling these networks. In one hand we are integrating existing deliberative planning mechanisms (Py et al. (2010)) for decoupling objectives from lower-level execution and, on the other hand, we are building a new framework for the creation of IMC software agents that are able to migrate and redefine the behavior among the network.

In the future, and as a means of further testing the robustness of the toolchain, we will continue to test it in further single and multi-vehicle deployments and, as possible, test continuous (24/7) operations at sea using heterogeneous vehicles with limited endurance.

REFERENCES

Dias, P., Gonçalves, G., Gomes, R., Sousa, J., Pinto, J., and Pereira, F. (2006). Mission planning and specification in the neptus framework. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, 3220 –3225.

Ferreira, H., Martins, R., Marques, E., Pinto, J., Martins, A., Almeida, J., Sousa, J., and Silva, E. (2007). Swordfish: an autonomous surface vehicle for network centric operations. In *OCEANS 2007 - Europe*, 1 –6.

Gerkey, B., Vaughan, R., and Howard, A. (2003). The Player / Stage Project : Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the 11th international conference on advanced robotics*, 317–323.

Gomes, R., Martins, A., Sousa, A., Sousa, J., Fraga, S., and Pereira, F. (2005). A new rov design: issues on low drag and mechanical symmetry. In *Oceans 2005 - Europe*, volume 2, 957 – 962 Vol. 2.

Gonçalves, G.M., Pereira, E., de Sousa, J.B., Morgado, J., Bencatel, R., Correia, J., and Félix, L. (2009). Unmanned air vehicles for coastal and environmental research.

LSTS (2011). Manta user manual. `http://whale.fe.up.pt/manta/a300/Manta_A300_User_Manual_r1.pdf`, last accessed date: 7 March 2012.

Madureira, L., Sousa, A., Sousa, J.B., and Gonçalves, G.M. (2009). Low cost autonomous underwater vehicles for new concepts of coastal field studies. In *10th International Coastal Symposium (ICS 2009)*.

Marques, E., Gonçalves, G., and Sousa, J. (2006). Seaware: A publish/subscribe communications middleware for networked vehicle systems. In *Proc. IFAC Conference on Manoeuvring and Control of Marine Craft (MCMC)*. IFAC.

Marques, E., Pinto, J., Kragelund, S., Dias, P., Madureira, L., Sousa, A., Correia, M., Ferreira, H., Gonçalves, R., Martins, R., Horner, D., Healey, A., Gonçalves, G., and Sousa, J. (2007). AUV control and communication using underwater acoustic networks. In *Proc. IEEE Oceans Europe*. IEEE.

Martins, R., Dias, P., Marques, E., Pinto, J., Sousa, J., and Pereira, F. (2009). Imc: A communication protocol for networked vehicles and sensors. In *OCEANS 2009 - EUROPE*, 1 –6.

Pinto, J., Dias, P.S., Gonçalves, R., Marques, E., Gonçalves, G.M., Sousa, J.a.B., and Pereira, F.L. (2006). NEPTUS – A Framework to Support the Mission Life Cycle. In *7th IFAC Conferent on Manoeuvring and Control of Marine Craft*. Lisbon, Portugal.

Py, F., Rajan, K., and McGann, C. (2010). A systematic agent framework for situated autonomous systems. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 2 - Volume 2*, 583–590.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.