A Giovanni e Piero

Contents

Abstract

The goal of this thesis is to create a C++ module to interface the ns2/NS-Miracle network simulator [35] with the Micromodem developed by the Woods Hole Oceanographic Institution (WHOI Micromodem). This module is part of the DESERT Underwater (short for DEvelope, Simulate, Emulate and Realize Testbed for Underwater network protocols) framework, a suite of modules developed to support the design and testing of underwater network protocols. From the network simulator's point of view, the implemented module acts like a common physical layer, but, rather than to be connected to a simulated channel, it opens a serial connection between the machine running ns2 and the Micromodem. Then, the modem transmits acoustically the packet over the real underwater channel.

In collaboration with the Woods Hole Oceanographic Institution (WHOI), a leader in the field of underwater technology, we created a network testbench composed of seven easy deployable nodes. These nodes are intended to be used for a wide range of applications thanks to their multi purpose configuration: the embedded system is composed by a Gumstix board [14] running Emdebian (Debian distribution for embedded systems [11]) and it is possible to access each single node of the network through a WiFi SSH connection in order to schedule tests, launch programs and collect results.

The thesis is organized as follow: Part 1 introduces the work by means of a brief overview on underwater networks, the ns2 network simulator and its extension NS-Miracle. Part 2 details the code developed to interface NS-Miracle with the WHOI Micromodems; finally, Part 3 describes the field experiments we run to test the feasibility of our solution. The results of these tests show that the implemented interface allows us to command real hardware by reusing the same code written for simulation purposes.

Part I
Introduction

1 Underwater Acoustic Networks

Underwater sensor networks are mainly used to enable applications for oceanographic data collection, pollution monitoring, offshore exploration, disaster prevention, assisted navigation and surveillance applications. Some examples are:

- *Ocean sampling networks.* Networks of sensors and AUVs, such as the Odyssey-class AUVs [3], can perform synoptic, cooperative adaptive sampling of the 3D coastal ocean environment [26].
- *Robotics and ocean models.* Experiments such as the *Monterey Bay field experiment* [33] demonstrated the advantages of bringing together sophisticated new robotic vehicles with advanced ocean models to improve the ability to observe and predict the characteristics of the oceanic environment.
- *Environmental monitoring.* Underwater (UW) sensor networks can perform pollution monitoring (chemical, biological and nuclear). For example,

it may be possible to detail the chemical slurry of antibiotics, estrogen-type hormones and insecticides to monitor streams, rivers, lakes and ocean bays (water quality in situ analysis) [13]. Other types of monitoring include tracking of fishes or micro-organisms.

- *Environment predictions.* Improved weather forecasts detection of climate changes, enhancement of the awareness on the effect of human activities on marine ecosystems.

- *Undersea explorations.* Underwater sensor networks can help detecting underwater oilfields or reservoirs, determine routes for laying undersea cables, and assist in exploration for valuable minerals. [15]

- *Disaster prevention.* Sensor networks that measure seismic activity from remote locations can provide tsunami warnings to coastal areas [25], or study the effects of submarine earthquakes (seaquakes).

- *Assisted navigation.* Sensors can be used to identify hazards on the seabed, locate dangerous rocks or shoals in shallow waters, mooring positions, submerged wrecks, and to perform bathymetry profiling.

- *Distributed surveillance.* AUVs and fixed underwater sensors can collaboratively monitor areas for surveillance, reconnaissance, targeting and intrusion detection systems. For example, in [10], a 3D underwater sensor network is designed for a tactical surveillance system that is able to detect and classify submarines, small delivery vehicles (SDVs) and divers based on the sensed data from mechanical, radiation, magnetic and acoustic microsensors. With respect to traditional radar/sonar systems, underwater sensor networks can reach a higher accuracy, and enable detection and classification of low signature targets by also combining measures from different types of sensors.

- *Mine reconnaissance.* The simultaneous operation of multiple AUVs with acoustic and optical sensors can be used to perform rapid environmental assessment and mine-countermeasure applications.

As mentioned, multiple unmanned or autonomous underwater vehicles (UUVs, AUVs), equipped with underwater sensors, can also be employed for the exploration of natural undersea resources and gathering of scientific data in collaborative monitoring missions. The recent underwater sensor nodes and vehicles could be equipped with self-configuration capabilities, i.e., they may be able to coordinate their operation by exchanging configuration, location and movement information to fulfil a given task, e.g., the relaying of monitored data to one or multiple onshore stations.

To obtain the desired level of collaboration and self adaptation, sensors and AUVs must be able to reliably communicate through the wireless underwater channel, therefore, the acoustic communications are the key-enabling technology for this kind of applications. On one hand, radio waves propagate at long distances through conductive sea water only at extra low frequencies (30 - 300 Hz), which require large antennas and high transmission power and they are affected mostly by multipath and Doppler effects rather than SNR. On the other, optical waves do not suffer from high attenuation but are affected by scattering. Moreover, transmission of optical signals requires short range ($\leq 100m$) in very clear water [23]. Thus, links in underwater networks are based on acoustic wireless communications [34].

Generally speaking, there is a need to deploy underwater networks that will enable real-time monitoring of selected ocean areas, remote configuration and interaction with onshore human operators. This can be obtained by connecting underwater instruments by means of wireless links based on acoustic communication. Many researchers are currently engaged in developing networking solutions for terrestrial wireless ad-hoc and sensor networks. Although there exist many recently developed network protocols for wireless sensor networks, the unique characteristics of the underwater acoustic communication channel require very efficient and reliable new data communication protocols. [18] Major challenges in the design of underwater acoustic networks are:

- The available bandwidth is severely limited due to absorption losses;
- The underwater channel is time-varying, especially due to multi-path and fading;
- Propagation delay in underwater is five orders of magnitude higher than in radio frequency (RF) terrestrial channels, and extremely variable from link to link;
- The variability in time and space often leads to high bit error rates and temporary losses of connectivity (shadow zones);
- Battery power is limited, but solar energy can be exploited (for buoys and surface nodes);
- Underwater sensors are prone to failures because of fouling and corrosion.

2 The ISO/OSI stack

The Open Systems Interconnection model (OSI model) shown in Table 1, is a product of the Open Systems Interconnection effort at the International Organization for Standardization [38]. It is a prescription of characterizing and standardizing the functions of a communications system in terms of abstraction layers. Similar communication functions are grouped into logical layers and a layer serves the one above it and is served by the layer below it. The OSI standards documents are available from the ITU-T as the X.200-series of recommendations, and according to them there are seven layers, numbered from 1 to 7 starting from the bottom.

In the following, we investigate deeper the main functionalities of these layers. According to the interests of this thesis, more details are provided for the layers exploited in the underwater scenario we considered and for those hardwares that we adopted.

- *Physical layer*: this layer is responsible for transmitting bits through the channel, it could group these bits into words and it codes them into a physical signal. Due to the challenging characteristics of the underwater channel, the WHOI Micromodem development was initially based on

Table 1: ISO/OSI stack

OSI Model			
	Data Unit	Layer	Function
Host Layer	Data	7. Application	User level, generation and usage of data.
		6. Presentation	Data representation, encryption and decryption, convert machine dependent data to machine independent data.
		5. Session	Interhost communication, managing sessions between applications.
	Segments	4. Transport	End-to-end connections, reliability and flow control.
Media layers	Packet/Datagram	3. Network	Static or dynamic path determination, logical addressing.
	Frame	2. Data link	Medium Access Control (MAC), physical addressing.
	Bit	1. Physical	Physical binary transmission of data.

non-coherent frequency shift keying (FSK) modulation (see Section 7) , that relies on energy detection to reveal the presence of a packet in the channel. The multi-path effects are suppressed by inserting time guards between successive pulses to ensure that the reverberation, caused by the rough ocean surface and bottom, vanishes before each subsequent pulse is received. Although non-coherent modulation schemes are characterized by a high power efficiency, their low bandwidth efficiency makes them unsuitable for high data rate networks. Hence, coherent modulation techniques such as phase shift keying (PSK), made available by WHOI Micromodem (see Section 4.1.2) and quadrature amplitude modulation (QAM) [31] have been developed for high-throughput systems.

Other techniques that could be used, which are not of primary interest to this thesis, are:

  - channel equalization techniques to leverage the effect of the inter-symbol interference (ISI), instead of trying to avoid or suppress it;
  - differential phase shift keying (DPSK) [9] serves as an intermediate solution between incoherent and fully coherent systems in terms of bandwidth efficiency. DPSK encodes information relative to the previous symbol rather than to an arbitrary fixed reference in the signal phase and may be referred to as a partially coherent modulation. While this strategy substantially alleviates carrier phase-tracking requirements, the penalty is an increased error probability over PSK at an equivalent data rate;
  - orthogonal frequency division multiplexing (OFDM) [27] spread spectrum technique, which is particularly efficient when noise is spread over a large portion of the available bandwidth.

Many of the techniques discussed above require underwater channel estimation, which can be achieved by means of probe signals or training sequences to be sent before the actual packet transmission.

- *Data link layer.* The data link layer is the protocol layer that transfer data between adjacent nodes. Moreover, it might provide solutions to detect and possibly correct errors at the physical layer. Channel access control in UW networks faces additional challenges regard to the communications over the air, due to the intrinsic characteristic of the acoustic underwater channel, which is impaired by several factors such as high and variable delays that typically vary from link to link. At the core of the data link layer there is the Medium Access Control (MAC) mechanism, which aims at improving network performance, such as throughput, latency and energy efficiency. In the following we list some common MAC protocols:

  - ALOHA is a random access scheme [2] that allows to send the packets without preceding channel reservation and does not implement channel sensing or retransmission's techniques. Later enhancements include both an acknowledgement system and carrier sense mechanism, but this MAC is not suitable for environments affected by high delays;

  - Carrier sense multiple access (CSMA) [6] is a reasonable choice and prevents collision with the ongoing transmission at the transmitter side. To avoid collisions at the receiver side, it is necessary to introduce a guard time between transmissions dimensioned according to the maximum propagation delay in the network - this lowers the throughput. Due to propagation delay, it is not recommended to use RTS/CTS mechanisms [16]. Therefore is susceptible to hidden-terminal problem in multi-hop networks. CDMA is quite robust to frequency selective fading caused by underwater multi-paths, since it distinguishes simultaneous signals transmitted by multiple devices by means of pseudo-noise codes that are used for spreading the user signal over the entire available band. This allows exploiting the time diversity in the UW acoustic channel by leveraging Rake filters at the receiver. These filters are designed to match the pulse spreading, the pulse shape and the channel impulse response, so as to compensate for the effect of multi-path;
  - Distance-Aware Collision Avoidance Protocol (DACAP) [4], uses a very short warning packet in the RTS-CTS (short for Request To Send/Clear To Send) mechanism. It has been developed for high delay networks, and provide an optional acknowledge system. It comes with some problems: 1) it must be configured a priori, 2)it does not adapt easily to network topology changes, 3) trade-off between longer handshakes/idle times and collision probability;
  - Tone-Lohi (T-Lohi) [1], uses tones during contention rounds to reserve the channel. T-Lohi listens to the channel during a contention round to count the number of contenders in the same round. Thanks to the high propagation delays present in underwater networks, counting the number of contenders in the same round is easy because other requests from other nodes mostly arrive separately. The backoff time is adapted to the number of contenders. Generally, T-Lohi is not suitable for long range multi-hop transmissions because of the dependency between the contention round duration and the maximum propagation delay.

- *Network layer.* Broadly speaking, the network layer is in charge of determining the path between source and destination (e.g. from the sensor that samples a physical phenomenon to the surface station). While many impairments of the underwater acoustic channel are adequately addressed at the physical and data link layers, some other characteristics, such as the extremely long propagation delays, are better addressed at the network layer. In the last few years there has been an intensive study in routing protocols for ad hoc wireless networks [21] and sensor networks [19]; nowadays the results of these studies might be applied or adapted for underwater scenarios. The existing routing protocols are usually divided into three categories, namely proactive and reactive routing protocols:

  - *Proactive protocols* (e.g., Destination-Sequenced Distance Vector (DSDV) [30], Optimized Link State Routing Protocol (OLSR) [7]). These protocols attempt to minimize the message latency induced by route discovery. To this aim, they maintain up-to-date routing information at all times from each node to every other node in the network. This is obtained by periodically broadcasting control packets that contain routing table information (e.g., distance vectors). The main drawbacks of this protocols is a large signaling overhead to establish routes, and it can be very high especially in mobile networks where route tables need to be updated more often;

- *Reactive protocols* (e.g., Ad-hoc On-Demand Distance Vector (AODV) [29], Dynamic Source Routing (DSR) [8]). According to this approach, a node initiates a route discovery process only when a route to a destination is required (e.g., when it has a packet to transmit or it is selected as relay). Once a route has been established, it is maintained by a route maintenance procedure until it is no longer desired or considered obsolete (e.g. after a timeout expiration). These protocols are more suitable for dynamic environments but they are also characterized by high latencies due to the path discovery procedure, which may lead to even higher delays underwater because of the slow propagation of acoustic signals. Furthermore, links are likely to be asymmetrical, due to bottom characteristics and variability of the sound speed profile;
- *Geographical routing protocols.* These protocols establish the routing path by analyzing nodes position. These techniques are really promising, but it is hard to obtain and maintain updated localization informations underwater.

A general observation is that protocols that rely on symmetrical links, such as most of the reactive protocols, are unsuited for many underwater environments. Moreover, the topology of UW networks is unlikely to vary dynamically on a short time scale. We briefly conclude this point with some open research issues related to efficient routing:

- For delay-tolerant applications, there is a need to develop mechanisms to handle loss of connectivity without provoking immediate retransmissions. Strict integration with transport and data link layer mechanisms may be advantageous to this end;
- It is necessary to devise routing algorithms that are robust with respect to the intermittent connectivity of acoustic channels. The quality of acoustic links is highly unpredictable, since it mainly depends on fading and multi-path, which are phenomena hard to model;
- Algorithms and protocols need to detect and deal with disconnections due to failures, unforeseen mobility of nodes or battery depletion. These solutions should be local so as to avoid communication with the surface station and global reconfiguration of the network, and should minimize the signalling overhead;
- Mechanisms are needed to integrate AUVs in underwater networks and to enable communication between sensors and AUVs. In particular, all the information available to sophisticated AUV devices (trajectory, localization) could be exploited to minimize the signaling needed for reconfigurations.

- *Transport layer.* In the contest of Under Water Acoustic Sensor Networks (UWASN), transport layer protocols are needed especially to achieve reliable collective transport of event features, more in general, however, to perform flow and congestion control. Congestion control is needed to prevent the network from being congested by excessive data with respect to the network capacity, while flow control is generally performed to avoid that network devices with limited memory might be overwhelmed by data transmissions. As a matter of fact, the primary objective is to save scarce sensor resources (e.g. energy) and increase the network efficiency. A reliable transport protocol should guarantee that the applications be able to correctly identify event features estimated by the sensor network.

  However, most existing Transmission Control Protocol (TCP) implementations are unsuited for the wireless communication in general, as for the underwater environment in particular, since the flow control functionality is based on the assumption that the losses over the channel are caused by congestion in the network which turns out in a window-based mechanism that relies on an accurate estimate of the Round Trip Time (RTT).

- *Application layer.* Many application areas for underwater sensor networks can be outlined, stemming from the overview presented in Section 1. A deeper understanding of these application areas and of the communication problems in underwater sensor networks is crucial to determine useful design principles on how to extend or reshape existing application layer protocols for terrestrial sensor networks. For this kind of studies we refer the reader, e.g., to [17].

Usually, to investigate and design new network protocols, researchers test their performances using network simulators and rely on the accuracy of the channel model in use. With this work we go beyond the simulation: not only it is possible to bypass the simulation of the channel (slow and not precise in harsh environments), but we allow to create and test real network prototypes. Implementing research solutions on actual devices, in fact, is of key importance to realize a communication and networking architecture that allows heterogeneous nodes to communicate reliably in the underwater environment.

The main research efforts are related to the layers above the physical one, however, since every new protocol requires a testing process and the results are strictly dependent from the channel's properties, it would be appropriate to use something as close as possible to a real acoustic channel. The models in use are reliable in deep water scenario without complex obstacles, but there is no accepted shallow water model or even a tool to simulate, for example, an harbour.

With our work we let developers use a testbed that, through the modems, lets the developer deal with the real channel.

3 The network simulator

3.1 ns2

[24] The network simulator ns2 is a discrete event simulator targeted at networking research. It provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

Historically, ns began as a variant of the REAL network simulator in 1989 and has evolved substantially over the past few years. In 1995 ns development was supported by the Defense Advance Research Projects Agency (DARPA) through the Virtual InterNetwork Testbed (VINT) project at Lawrence Berkeley National Laboratory (LBNL), Xerox PARC, University Of California (UCB), and University of Southern California Information Science Institute (USC/ISI). The second version, ns2, is supported through DARPA with Simulation Augmented by Measurement and Analysis for Networks (SAMAN) and through National Science Foundation (NSF) with Cooperative Online Serials (CONSER), both in collaboration with other researchers including ATT Center for Internet Research (ACIRI). Ns has always included substantial contributions from other researchers, including wireless code from the UCB Daedelus and Central Michigan University (CMU) Monarch projects and Sun Microsystems.

While the developers have considerable confidence in ns2, it is not a polished and finished product, but the result of an on-going effort of research and development. In particular, bugs in the software are still being discovered and corrected.

We choose to use NS-Miracle, a set of libraries for ns2, that exploits the possibility of loading the libraries dynamically: especially in an embedded system, with the fact that lot of modules were already developed, this was the optimal starting point for our work. NS-Miracle introduces the possibility of multiple modules at the same layer, as well as cross-layer messages. Last but not least, there is a lot of code already developed for UW networks.

3.1.1 Fundamental mechanisms of ns2: Shadowing, binding and command

The ns2 simulator uses two different programming languages, OTcl and C++. On one hand, OTcl is compiled at run time and allows the developer to quickly setup network parameters and configurations. On the other hand, C++ gives the developer the power to handle efficiently algorithms, packet's headers, data structures and large data sets. It is used to program the node's modules that are compiled and ready to be used by the simulator. When we want to load a C++ object as an OTcl object, we must link these two classes together, technically speaking we must create in the OTcl domain a *shadow* C++ object.

ns2 manages also the variable *binding*, such that both the OTcl member variable and the C++ member variable access the same data. This mechanism allows to set any C++ variables by changing the value of its corresponding OTcl variable.

Finally, from the OTcl domain it is possible to call some methods of a C++ object. As well as for object names and variables, in fact some OTcl *commands* could be linked to the corresponding C++ function.

3.2 NS-Miracle

For our project we use Miracle, an extension of the ns2 simulator developed by the Department of Information Engineering at the University of Padova [22]. The acronym stands for Multi InteRfAce Cross Layer Extension and describes the two main capabilities introduced by the modularity of this extension. In fact, using Miracle we can manage nodes in a cross layer fashion and additionally provide multi technology support (within the same node as illustrated by Figure 1) [35].

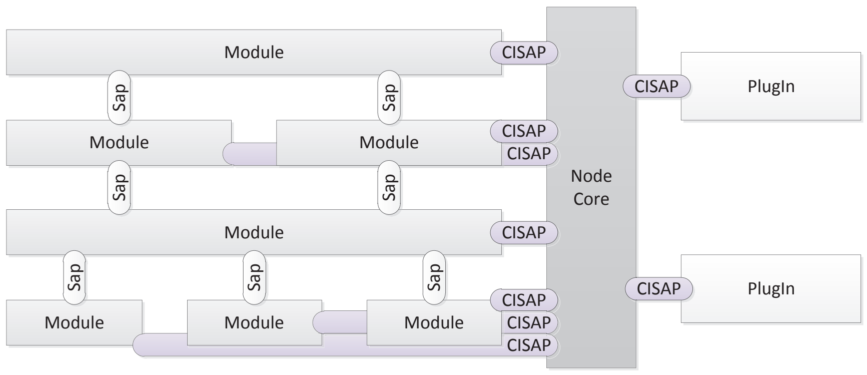


Figure 1: Miracle node

A motivation for creating NS-Miracle is to fill the gap between ns2 and the simulation of multi technology environments and provide the possibility of performing cross layer messaging, using a set of additional dynamic libraries. The architecture is highly modular as it allows the interconnection of multiple down and upstream modules at every layer of the protocol stack. In addition, a dedicated communications facility provides the protocol stack of each node with cross-layer interaction capabilities. In Miracle every modification can constitute a stand alone library and can be loaded when necessary with the *load* command. In this case modifications can be rapidly done, loading the appropriate library. Furthermore a modification of a library requires only to recompile the code associated to that library and it usually requires only a few seconds.

In detail, Miracle resolves some issues that are not addressed to a great extent by the current network simulator ns2 [22]. The channel and PHY layer modeling have big importance in the result's accuracy, hence Miracle offers a more realistic representation of signal propagation and reception process, giving at the same time a framework to handle a complete system, in particular the cross layer design. As mentioned above, as more and more devices nowadays are equipped with multiple interfaces using different communication technologies, network simulators should provide support for proper modeling of these scenarios, by means of a flexible and modular protocol stack architecture together with proper support for the development of the control modules which are needed to manage such a complex architecture. The heterogeneity also concern the network devices and simulating this kind of scenarios today is very challenging, in particular due to the fact that the routing layer of state-of-the-art simulators is mainly designed for homogeneous networks. As a consequence, there is a need for supporting this type of heterogeneous network composition at all the layers of the protocol stack.

3.2.1 The NS-Miracle node

The NS-Miracle node is based on generic entities connected by each other (see figure 1). The protocol stack is implemented through entities called Modules. The novel idea consists in allowing presence of more than one Module per layer. Every Module is connected to another entity called Node Core. It goes off the layer ordering scheme and has the task of coordinating message exchange between modules. It also stores the node actual position in the simulation field in order to allow position-dependent calculations such those related to propagation and interference models.

There are also another kind of entities connected to the Node Core, called PlugIns. As Node Core, they don't depend on layer classification. Due to their independence from the OSI stack, the PlugIn may be exploited for node coordination functionalities (e.g., cognitive engines and multi interfaces manager) and cross layer intelligence.

Communication among different layers is provided by Service Access Points (SAP), accordingly to OSI structure. As in ns2, connections between Modules, NodeCore and PlugIns are made by Connectors, but the latter are completely reprojected to trace packets passing through them according to rules established by the user.

3.3 DESERT Underwater

We have already discussed about how hard is to project and test an underwater network, especially when it involves multiple devices and the scenario differs from the simplest ones. When pursuing the latter goal, the research community demands a simulating framework to test and tune their networks before the on field applications. A flexible, reliable tool for performance evaluation is of fundamental importance to test and improve the design of network protocols: DESERT Underwater aims to solve this lack, allowing scientists to DEvelope, Simulate, Emulate and Realize Testbed for Underwater network protocols. It is a complete set of public C/C++ libraries that extends the NS-Miracle simulation software and provides several protocol stacks for underwater networks, as well as the support routines required for the development of new protocols.

Moreover, DESERT Underwater will make it possible to evolve from pure simulation towards the realization of actual prototypes by framing the hardware of real acoustic modems into NS-Miracle itself. According to what done in recent papers such as [5], the idea is to wrap all the commands required to communicate with the modem hardware within an NS-Miracle module. In this perspective, the developer can rely on two supported experimental settings: i) a (small-scale) emulation setting, where multiple acoustic modems are connected with a single device (e.g., PC, laptop); ii) a testbed setting, where each acoustic modem is connected with its corresponding unique device. Moreover, the use of embedded platforms such as the Gumstix [14], or the PandaBoard [28], that can replace actual PCs, would allow us to build more portable, autonomous and realistic testbeds that go in the direction of realizing actual prototypes.

Part II
Development

The core of this work is to create an interface to allow the network simulator NS-Miracle to command the WHOI Micromodems for sending packets over the acoustic channel. The overall goal is to bridge the gap between the studies based on the simulation environments and those exploiting the real UW networks. The above need arises from the observation that even if the UW channel is well modeled in certain situation(e.g. deep open sea), there is no software or algorithm able to reproduce UW channel behaviour in complex scenarios both reliably and in a reasonable time (think about the reflections, shadowing and multipath in a harbour). With our interface we want to provide the community with a tool to realize testbeds and perform their network protocols using the real acoustic channel.

This task implies several challenges: for instance, according to the modem in use, the informations to be sent may need to be shrunk in order to reduce the payload size and to overcome some problems like the limited bandwidth and low transmission rates.

Section 4 introduces the ns2/NS-Miracle packet and the acoustic packets in use; Section 5 describes the two different scenarios we considered, emulation and testbed; from Section 6 to 9 we describe the details of our work.

4 Structure of the ns2/NS-Miracle packet

In ns2, the packet is a structure containing all the possible fields that can be used within a simulation, and this is passed from one layer to another and it arrives to destination with a probability dictated by the simulated channel. So, there is no actual data that moves inside the simulation: all delays, probabilities of error and packet's manipulations are handled by ns2 and its modules.

For instance, there is an application layer's header that includes the sequence number of the packet, another one belonging to the IP layer that stores the source and destination IP address, header of the MAC module that memorizes the correct MAC addresses and so on. Even if a simulation does not use some modules that require a given header, the memory allocated for a single packet includes them all. Therefore, the memory allocated during the simulation for a given packet is usually bigger than the simulated packet size: only a portion of it carries critical data for the correct functioning of a given test.

This leads to the conclusion that is desirable not to send the whole memory allocated for a given packet because of the corresponding huge overhead: we need to shrink it and keep only the informations necessary for the correct functioning of the protocol to test. Alternatively we can exploit other techniques as explained in Section 7.2.

4.1 The acoustic modem payload

The WHOI Micromodem exchange messages using the NMEA 0183 standard, a combined electrical and data specification for communication between marine electronic devices such as echo sounder, sonars, anemometer, gyrocompass, autopilot, GPS receivers and many other types of instruments. It has been defined by, and is controlled by, the U.S.-based National Marine Electronics Association [36]. NMEA messages are ASCII strings that are sent through a serial port and they obey to a strict defined format, they could include fields like source, destination, modulation type, channel statistics and error messages.

As with many communications systems, all data transmitted by the Micromodem is broken up into frames. Depending upon the data rate, there may be more than one frame of data within a packet. The current version of the Micromodem software supports multiple transmit and receive rates, the higher rates contain multiple frames and the integrity of each frame is protected with a cyclic redundancy check (CRC). The FSK modem can send both Minipackets that carry 13 bits of data or binary data packet with a 32 bytes payload. The PSK modem can send both Minipackets as well as binary data with variable rate. Our developed solution to interface NS-Miracle with real hardware is compatible both with the FSK version of the modem as well as the PSK version (U.S. only).

4.1.1 FSK, Minipacket

Frequency-shift keying (FSK) is a frequency modulation scheme in which digital information is transmitted through discrete frequency changes of a carrier wave. [12] The simplest FSK is binary FSK (BFSK). BFSK uses a pair of discrete frequencies to transmit binary information. With this scheme, the "1" is called the mark frequency and the "0" is called the space frequency. The time domain of an FSK modulated carrier is illustrated in figure 2.

Currently, the WHOI trades the FSK Micromodem only outside United States, therefore we concentrate some efforts to realize a version of our module that is compatible with the limitations of this hardware and maximizes the available resources. The FSK modem allow to send up to 32 bytes of data, but the transmission rate is really low, 80 bps, hence our choice has been to exploit only the communication via Minipacket despite the constraints implied by the 13 bits payload. The advantage of this choice is the 1:1 correspondence between the ns2's packet and the acoustic one (this is not possible with bynary data because it needs a Cycle Init as the first packet in a transmission). In this case the 13 bits of data payload are encoded in the NMEA sentence as two 8 bit hex values (4 characters). The format of a Minipacket message is as follows:

$CCMUC,SRC,DST,BHHH*CS

SRC	Source (4 bits)
DST	Dest (4 bits)
BHHH	ASCII coded hex data (2 hex values). Value in the range of 0 to 1FFF are legal (13 bits)
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

The minipacket is relatively short, less than 1 second: precisely, the modem transmits at 160 symbols per second, that is 80 bps because of the 1/2 code rate. The overall size is 32 bits (29 bits listed above + 3 mode bits that identify the type of minipacket) that are coded on 80 symbols. Furthermore, the packet is preceded by a 10 ms probe, a 200 ms time guard (between the probe

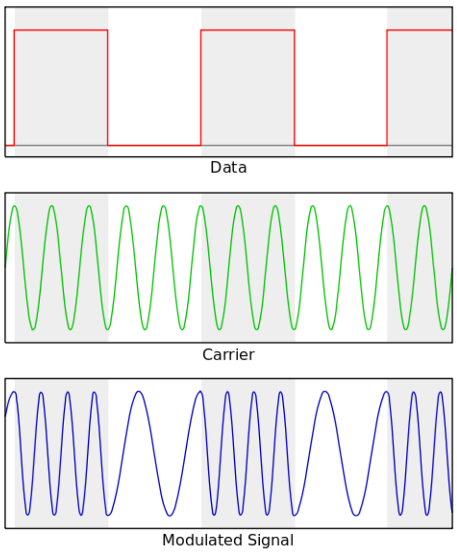


Figure 2: FSK modulation

and the acoustic packet) and 21 sync symbols. Therefore, the sending time is: $10ms + 200ms + (21/160)ms + (80/160)ms \simeq 710ms$.

The sequence of messages exchanges between two modems that send and receive a Minipacket is reported in Table 2 (remember that CCxxx indicates a communication messages from host to modem whilst CAxxx messages from modem to host). For clarity we indicate only the NMEA command name, not all the corresponding fields.

Table 2: Minipacket NMEA messages exchange

SOURCE	DESTINATION	Description
CCMUC		request from host to modem to send a minipacket
CAMUC		modem to host, command correctly received
CATXP		begin of transmission
	CADQF	receiver detect incoming packet
CATXF		end of transmission
CAXST		statistics of the sent packet
	CAMUA	receiver gets the minipacket
	CACST	statistics of the received packet

4.1.2 PSK, Binary Data

Phase-shift keying (PSK) is a digital modulation scheme that conveys data by changing, or modulating, the phase of a reference signal (the carrier wave).

Any digital modulation scheme uses a finite number of distinct signals to represent digital data. PSK uses a finite number of phases, each assigned to a unique pattern of binary digits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data. This requires the receiver to be able to compare the phase of the received signal to a reference signal (such a system is called coherent and referred to as CPSK).

The PSK version of WHOI's Micromodem allows to send the binary data packets at variable rates, the messages are generated by the host and the payload (which size is defined by the transmission rate as shown in Table 4) contain hex encoded binary data. The format of a binary data message is as follows:

\$CCTXD,SRC,DEST,ACK,HH...H*CS

SRC	Source (4bits)
DST	Dest (4bits)
ACK	Packet Acknowledgement
HH...H	Hex coded data bytes, 2 characters each, e.g. 00-FF.
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

According to the WHOI Micromodem specifications, a binary data message must be preceded by a Cycle Init (CCCYC message):

\$CCCYC,CMD,SRC,DST,PacketType,ACK,NumFrames*CS

that alerts the receiver and the other nodes of the network about the incoming transmissions and specifies the total number of frames that form the packet.

The messages involved in a binary data packet transmission are shown in Table 3. For a complete description, we show the transmission of a packet splitted in two frames.

Unlike the User Minipacket, using the Binary Data packet we need a Cycle Init message and possibly to split a single ns2 packet into multiple frames, loosing the 1:1 correspondence with the ns2's packet. The physical delivery of a packet is seen as an atomic event by the simulator, while there are at least two packets sent over the real underwater acoustic channel (Cycle Init plus one or more frame). Oppositely, the system gains higher transmission rates and the payload can be 20 to 1260 times larger. In all the tests reported in Part 3, we use the DATA packet because of the overall amount of informations that nodes need to exchange and cannot be fitted in the Minipacket payload.

5 Emulation, Testbed

The module interface we have developed can be used in two different settings, *emulation* and *testbed*.

Table 3: Binary data packet NMEA messages exchange

SOURCE	DESTINATION	Description
CCCYC		host to modem, Cycle Init
CACYC		modem to host, Cycle Init command correctly received
	CARXP	detected incoming transmission
	CACYC	Cycle Init packet received
CADRQ		modem to host data request (first frame)
CCTXD		transmission of first frame
CATXP		modem to host, start transmission of first frame
	CARXD	reception of the first frame
CADRQ		modem to host data request (second frame)
CCTXD		transmission of second frame
CATXP		modem to host, start transmission of second frame
	CARXD	reception of the second frame
	CACST	statistic of the received packet (both frames received)
CATXF		end of packet transmission
CAXST		statistics of the transmission

Table 4: Packet types and corresponding payloads

Packet Type	Max num. of frames	Bytes/Frame	Bytes/Packet
0	1	32	32
1	3	64	192
2	3	64	192
3	2	256	512
4	2	256	512
5	8	256	2048
6	6	32	192

5.0.3 Emulation

It is used when a single host controls all the modems, hence there is only one instance of ns2 running. This is a very simple and convenient case in terms of simplicity and reliability because there is only one simulator running, it means that the nodes involved in the test share the same memory. Remind that in NS-Miracle a packet is a pointer to a struct, therefore, the only information that needs to be sent is that pointer. The receiver's physical layer is able to send that pointer to upper layers without any segmentation fault because the receiver node has access to the simulator's allocated memory.

Similarly, in the emulation setting, we exchange among nodes a pointer to some memory location of the machine running the simulator; this pointer points to the data struct containing the ns2 packet to transmit. A pointer size is machine dependent and, generally, bigger then the Minipacket payload (13 bits). To overcome this problem and realize the emulation setting also for the FSK modem we use disk files to store the pointers of the packets to be sent over the UW channel, then we send acoustically the line number of the file where to find the desired pointer. We have therefore $2^{13} = 8192$ different line/payloads.

The receiver gets through the modem the line where to retrieve the right pointer. There are some drawbacks whit this approach: the emulation setting is suitable for small testing area because the modems must be physically connected to the same host.

5.0.4 Testbed

The testbed setting involves one pc/host for every modem. In this case there are multiple simulator's instances that run at the same time without sharing any memory, hence the challenge is to send a NS-Miracle packet over the UW channel. To this end, the sender must convert the NS-Miracle packet into the payload of an acoustic message that contains all the necessary informations in order to allow the receiver to recreate the original simulator's packet.

When we use the Minipacket, we need to compress the necessary informations in 13 bits and this leads to limitations in the doable testbeds (e.g. max number of nodes).

When using the PSK version, instead, we adopt a different technique that is the serialization of the memory area containing the headers of the packet which provide for the correct functioning of the simulator (see Section 8.1 for further details). Our approach is to create a payload that mirrors the memory area where the essential packet informations are stored. The payload size with the packet rate 4 (see Table 4) is big enough for our purpose, and the adoptes solution has the advantage of handling informations coming from the upper layers by treating them as "black boxes" (e.g., without knowing their actual implementation or the packet fields they need). A future improvement could use compression techniques to reduce the its size (even if the transmission duration is affected more by time guards that belongs to the MAC module implemented in the WHOI Micromodem firmware rather than the actual packet's sending time) to save some space for the application layer (e.g., to transmit sensors' data).

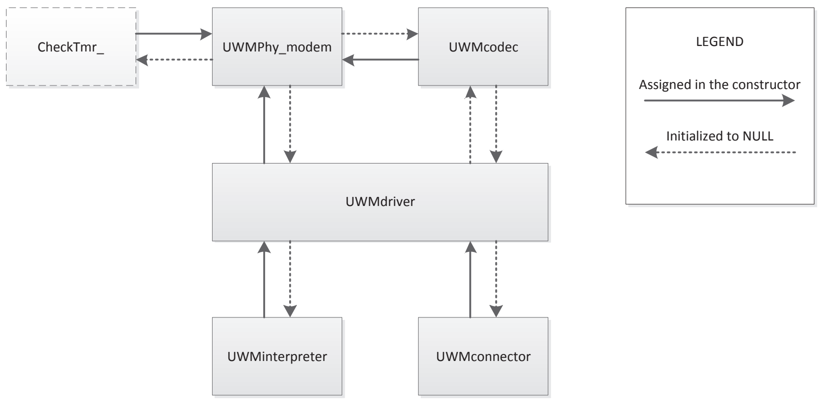
6 Design of the interface between NS-Miracle and general modem hardware

The NS-Miracle module we developed to interface the network simulator with the acoustic modem is intended to be as general as possible, modular and easy adaptable to future development (e.g. extension to support different modem hardware).

The general structure of this module, called UWMPhy_modem, is shown in Figure 3, where every block is a class that groups some logical duties, as described hereunder. UWMPhy_modem is developed to work in this way:

- UWMPhy_modem receives NS-Miracle packet from the above layer, it opens and closes the serial port at the beginning and at the end of the experiment;
- it interacts both with UWMcodec and UWMdriver: UWMcodec takes care of mapping the received NS-Miracle packet into a legal NMEA payload (that will be included in the acoustic packet) or demapping the received acoustic payload into a legal NS-Miracle packet, while UWMdriver handles

Figure 3: Sketch of the NS-Miracle modem hardware interface's design



the sequentiality of the messages exchanged with the modem (it works like a state machine);

- UWMinterpreter is called by UWMdriver to create the strings to be sent via serial port to the modem. These strings are hardware dependent and in our case they follow the NMEA standard. On the receiver side, the interpreter extracts the payload from the acoustic packet and it passes it to UWMcodec through the UWMdriver;
- UWMconnector interact with UWMdriver as it is the class that manage the serial connection, namely, the transmission and reception of NMEA string to/from the modem.

Notice that in Figure 3 some blocks have a pointer assigned by the constructor to other classes, meanwhile some other pointers are NULL initialized. Since the necessary dependencies can only be determined upon the knowledge of the hardware, we fix some links between the classes that must be implemented, leaving the necessary degrees of freedom to the developer. Any derived classes of UWMPhy_modem must specify the right linkage for the hardware in use.

UWMPhy_modem inherits from MPhy, the base class of NS-Miracle for physical layers that is meant to define the functionalities shared among channel models and wireless technology implementations. In particular, our physical module defines the self explanatory methods `startTx()`, `endTx()`, `startRx()` and `endRx()`, that are called upon the reception of specific control messages received from the modem. In NS-Miracle the physical layer has a strict interaction with the MAC layer (which modules inherit from the MMAC base class): the communication between these two layers exploits the cross-layer functionality introduced in Section 3.2. In fact, MPhy's derived classes can trigger the transmission/reception start/end events on the MAC layer (logically, this happen inside the functions listed above), using the MMAC's methods `PhyToMacEndTx()`, `PhyToMacStartRx()` and `PhyToMacEndRx()` (that corresponds to the `sendUp` function). Consequently, we designed UWMPhy_modem obey to the natively mechanisms of NS-Miracle that regulate the interaction between PHY and MAC layer. In detail, we programmed an internal state machine that handles the cross-layer messaging and the standard operations of any modem. Figure 4 reports the state machine used by `UWMdriver`, the block of the interface commanding the modem hardware.

We give a brief description of the states to better understand how our module interacts with the MAC layer. The methods `start()`, `modemTx()` and `resetStatus()` are called by UWMPhy_modem, while the other transitions are handled by UWMdriver. From the IDLE state, we change to the CFG state to configure, e.g., the modem ID; then there could be two events: transmission (TX) or reception (RX) of a packet. After a transmission the modem could go back to IDLE state or, if the transmission request happened right before an incoming packet detection:

1. the transmission is paused (TX_PAUSED) and the reception physically ends,
2. the paused packet is sent (TX_RX),
3. the reception logically ends (IDLE_RX).

Note that in a single unique device, the intermission of a transmission caused by a concurring reception cannot generally happen because the device is already busy (transmitting). In our case, instead, we have two separated entities (the network simulator and the acoustic modem) that work together by means of an interface actions as a patch with the MAC layer: in fact, when a MAC sends down a packet to the physical layer, it goes in the transmission's state and any notification but `PhyToMacEndTx()` causes an erroneous logic transition. Since `UWMPhy_module` is placed between the hardware (whose firmware could implement a basic MAC protocol that allows to receive a packet after the notification that a packet needs to be sent and a NS-Miracle MAC module (assuming that such event cannot occur), it must act as a patch to avoid logical errors of both sides. Therefore, in case of a reception during transmission procedure, `UWMPhy_modem` buffers the incoming packet until the end of the transmission, hence, 1) it notifies the MAC module about the correct transmission with `PhyToMacEndTx()`, 2) the MAC module goes back to its IDLE state, 3)`UWMPhy_modem` calls `PhyToMacStartRx()` and `sendUp`. In this way, we logically split the transmission and reception procedures.

Let us now explain deeper the functionality of the blocks that compose UWMPhy_modem.

6.1 UWMPhy_modem

UWMPhy_modem is the class used to implement the interface between NS-Miracle and real acoustic modems. Since UWMPhy_modem (as the other blocks) is meant to be a base class, it only defines virtual functions and it contains all the binded variables and commands accessible by OTcl scripts. Binded variables are:

`ID_`: node ID [range in $\{0, 1, \ldots, 255\}$, the use of 0 is deprecated] (mandatory, default value = 0)

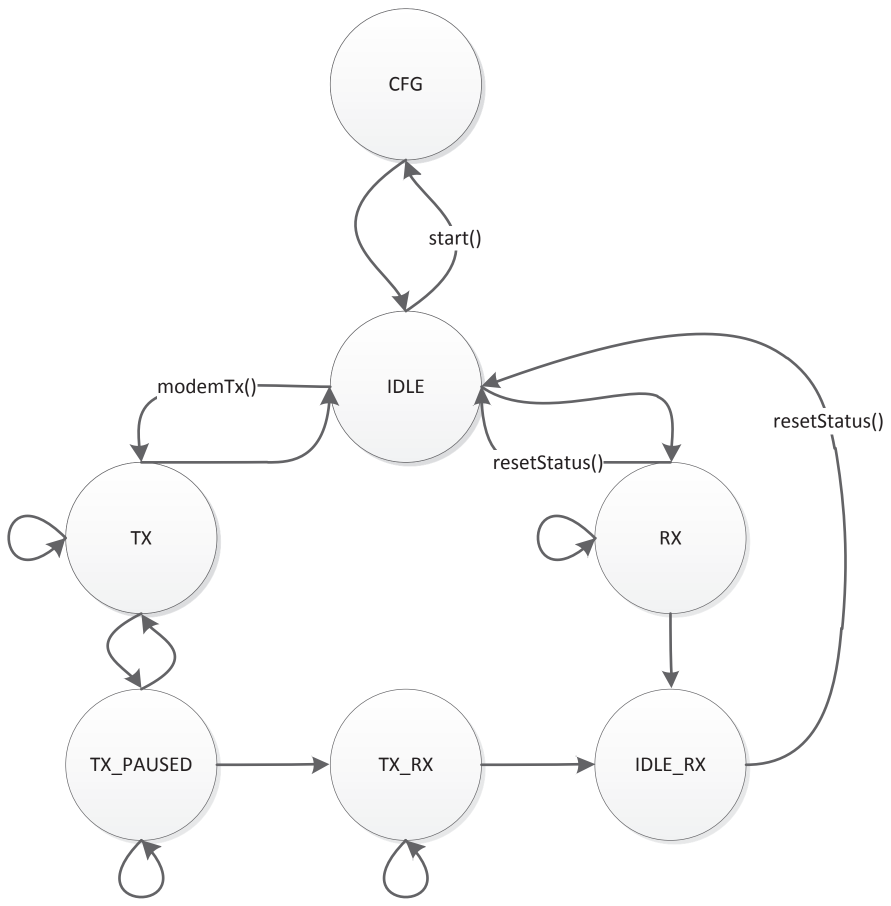


Figure 4: UWMdriver's state machine

`period_`: time interval [in sec] between two successive checks on the modem status (i.e., to control if packet reception is occurred) [range in $(0, \infty)$] (optional, default value = 0.001)

`setting_`: flag to switch between the TESTBED (flag equals to 1) and the EMULATION (flag equals to 0) setting [range in {0,1}] (optional, default value = 0)

`stack_`: flag to notify about the use of different protocol stacks [currently, range in {0,. . . ,2}] (optional, default value = 0)

`debug_`: flag to disable or enable debug messages [range in {0,1}] (optional, default value = 0)

`show_`: flag to disable or enable messages for presentation purposes (e.g., user-friendly messages that notify when a packet is going to be sent or received) [range in {0,1}] (optional, default value = 0)

For a better comprehension of these variables see Section 6.2, 6.3 and 6.5.

We defined also two commands: `start` and `stop`. The commands explain themselves, `start` activates the modem, that is, to open the serial port and to set certain modem's parameters like node ID, while `stop` closes and restores the serial port. Even if we cannot power on/off the modem form the TCL script, these two commands could be meant as if they worked for this purpose.

`UWMPhy_modem` includes a ns2 timer to schedule tasks: e.g., the serial port is checked periodically in case of an incoming packet.

Furthermore, this class coordinates the interactions between `UWMdriver` and the block in charge of transform the ns2 packets into legal acoustic payloads (`UWMcodec`). This base class does not include critical code like the shadowing: in fact this code belongs to the derived classes, as every inherited class must be designed for a specific hardware and creates its own object with corresponding unique name.

`UWMPhy_modem` declares also some virtual functions (listed in Table 5) that act like guide lines for the developing of any specific interface. As mentioned earlier, the functions `endTx` and `startRx` are crucial because they notify the upper MAC layer by calling the corresponding cross-layer messages about ending transmissions and starting receptions.

Table 5: Functions of UWMPhy_modem

public	
virtual void **recv** (Packet*)	This function handles packet reception from the upper layer of the network simulator

virtual int **command** (int argc, const char*const* argv)	command() function. This is used to map c++ methods to TCL commands: • *argv[3] is the TCL command name • *argv[4, 5, ...] are the parameters for that command.
protected	
void **setConnections** (CheckTimer*, UWMdriver*, UWMcodec*)	this function must be used by any derived class to specify the uninitialized links of the various object included in UWMPhy_modem (see figure 3)
virtual void **start** ()	This function starts the connection with the modem. It performs all the needed operations to open an host-modem connection (e.g., set up of the connection port's parameter, start of the "check-modem" process)
virtual void **stop** ()	This method should be used before stopping the simulator as it closes and, if needed, resets all the opened files and ports.
virtual int **check_modem** ()	This method is at the core of the "check-modem" process. It is called periodically to verify if something has been received or is going to be received from the channel. This function returns a flag on the status of the modem.
virtual void **startTx** (Packet* p)	Function to start packet transmission
virtual void **endTx** (Packet* p)	Function to call after the end of the packet transmission to notify the MAC layer about the end of a given transmission
virtual void **startRx** (Packet* p)	Start reception of a packet. It also sends to the above layers (MAC) the notification of such event
virtual void **endRx** (Packet* p)	End packet reception and send it to the MAC layer.

Table 6: Main functions of UWMcodec

public	
void **setConnections** (UWMdriver*)	this function must be used by any derived class to link the pointer to `UWMdriver` of UWcodec to the corresponding derived objects contained in the object pointed by the pointer to `UWMphy_modem`
virtual void **map_HtoM** (Packet* p)	function to code legal NS-Miracle packets into legal modem packets.
virtual void **demap_MtoH** ()	function to decode received acoustic packets into NS-Miracle packets.

6.2 UWMcodec

`UWMcodec` is in charge of coding the NS-Miracle packet into a legal modem payload and it does the reverse operation. This base class contains a pointer to the main class `UWMPhy_modem` to access the ns2 packet to send, it is also connected with an `UWMdriver` object that is in charge of notify the occurrence of a new incoming packet.

The actual procedure of mapping and de-mapping NS-Miracle packets is left to the developer, as it is hardware dependent. The approach we adopted for our field test is explained in Section 5. The main methods of `UWMcodec` are listed in Table 6.

6.3 UWMdriver

This class is needed by `UWMPhy_modem` to handle the different transmissions cases and corresponding protocol messages to be generated according to the tcl-user choices and modem firmware, respectively. This class define the state machine of Figure 4 and every derived class must respect these transitions.

The main methods of `UWMdriver` are listed in Table 7.

Table 7: Main functions of UWMdriver

public	
virtual void **modemTx** ()	Function to notify to the driver that there is a packet to be sent via modem. When this function is called, the status must be set to TX_

virtual int **updateStatus** ()	Function to update modem status. This function has to update the modem status according to the messages received from the modem/channel (e.g., after the check on the output of `UWMconnector`). It may return after an arbitrary period if nothing has happened, but it must return immediately after a status change
protected	
void **setConnections** (UWMinterpreter*, UWMconnector*)	Link connector. This function must be used by a derived class D to link the pointer to `UWMinterpreter` and the pointer to `UWMconnector` of `UWMdriver` to the corresponding derived objects contained in D
virtual void **modemTxManager** ()	Function to manage modem to host and host to modem communications. This function has to handle the different transmissions cases and corresponding protocol messages to be generated according to the tcl-user choices and modem firmware, respectively

6.4 UWMinterpeter

`UWMinterpreter` is the link between the `UWMdriver` and the acoustic modem since it creates/parses strings for/from the modem.

This class does not declare any virtual function, in fact the communication between host and modems is strictly dependent on the modem in use. It is free to be filled with the device-dependent algorithms to parse/create messages from/to the modem.

6.5 UWMconnector

The class needed by *UWMPhy_modem* to handle the physical connection between NS-Miracle and a real acoustic modem device. Its main methods are listed in Table 8.

Table 8: Main functions of UWMconnector

public	
virtual int **openConnection** ()	Method to open the connection with the modem. It uses the `pathToDevice` variable that is set from OTcl scripts.
int **writeToModem** (std::string)	Method for writing to the modem. It return the number of transmitted bytes.

std::string ()	**readFromModem**	Method to check the reading buffer.

7 Interface's specialization: the FSK Micromodem case

This module works with the FSK version of Micromodem, the classes described below inherit from the classes introduced in the previous section. The logic division of tasks leads us to revisit the module composition shown in Figure 3 as in Figure 5 (with a detail in Figure 6) and we explain below how the blocks are specialized and connected.

7.1 mFSK_WHOI_MM

`mFSK_WHOI_MM` inherits from `UWMPhy_modem` and it is the main class to implement the interface between NS-Miracle and the FSK WHOI Micromodem as it contains the shadowing code to load the module in the OTcl domanin. This class also defines all the missing linkage in Figure 3 in accordance with the specific modules that must be used (see below). To include the module in your OTcl scripts write:

```
set phy [new Module/UW/MPhy_modem/FSK_WHOI_MM serialPath]
```

where `serialPath` is the path to the serial port in use (e.g. `/dev/ttyUSB0`).

7.2 mcodecFSK_WHOI_MM

`mcodecFSK_WHOI_MM` inherits form `UWMcodec` and receives from `mFSK_WHOI_MM` the ns2 packet to be sent and it codes it into a legal NMEA payload. Depending on the setting in use (emulation or testbed), this class returns a different string has to be used as payload for the NMEA message to send to the modem.

7.3 mdriverFSK_WHOI_MM

This class extends `UWMdriver` (see section 6.3) and defines methods and state machines necessary for the right control of the messages exchanged with the modem. The driver respects the state machines shown in Figure 4 and it uses two other state machines to follow the particular sending/receiving protocol related to the modem in use /e.g., NMEA standard). In figure 7 and 8 those two state machines are shown: the links' labels report to the NMEA messages that trigger that particular transition and the states' labels are explained in table 9

7.4 minterpreterNMEA

The interpreter inherits from `UWMinterpreter` and contains methods (called by `mdriverFSK_WHOI_MM`) in charge of composing and parsing the NMEA sentences exchanged between host and modem, according to the NMEA 0183 standard

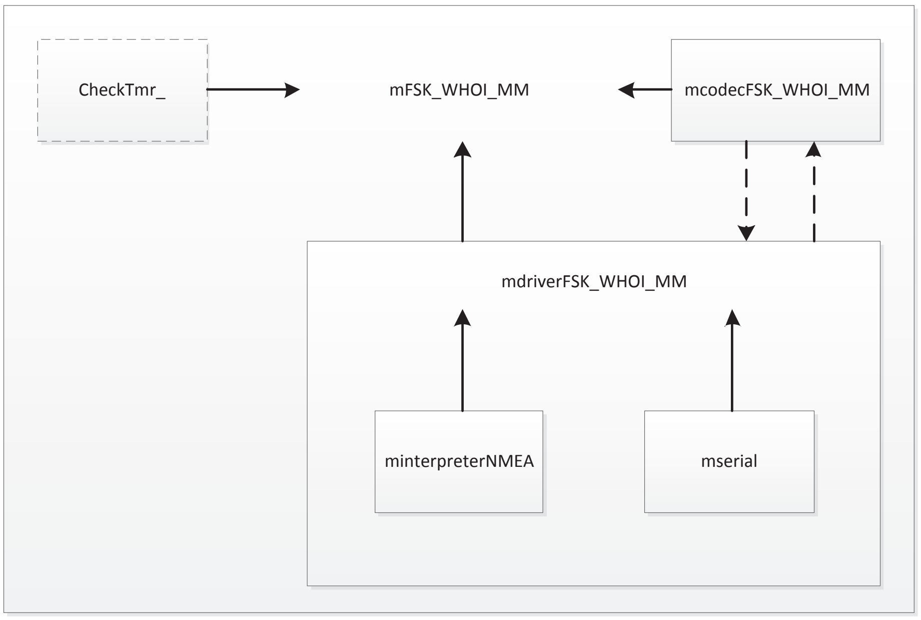


Figure 5: mFSK_WHOI_MM

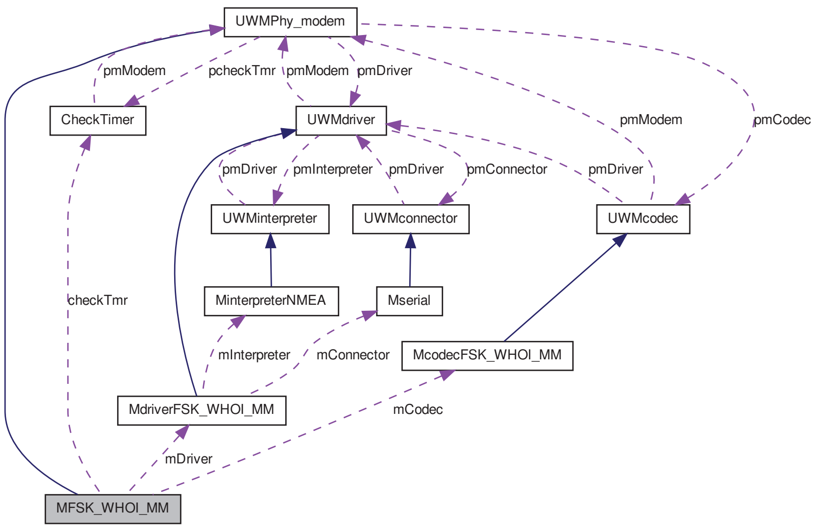


Figure 6: mFSK_WHOI_MM class hierarchy and pointers

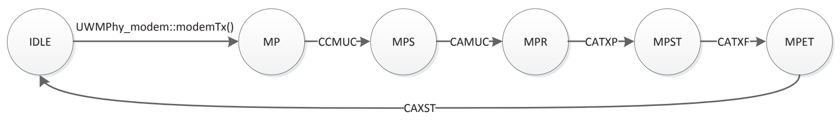


Figure 7: Transmission state machine implemented in mdriverFSK

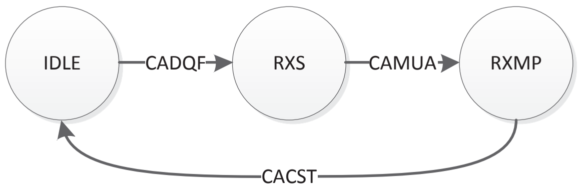


Figure 8: Reception state machine implemented in mdriverFSK

(see section 4.1). We remind a simple rule to understand the direction of the messages:

$CC*	host to modem
$CA*	modem to host

For example, the messages listed below occur in the transmission/reception of a packet and they need to be created/parsed:

- To configure WHOI Micromodem's parameters :
  $CCCFG,NNN,vv$*$CS

NNN	Name of parameter to set
vv	Value of parameter
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

- Cycle Init message, it is sent before the binary data packet:
  $CCCYC,CMD,ADR1,ADR2,Packet Type,ACK,Npkt$*$CS

Table 9: State machine labels

Transmission	
MP	Send minipacket
MPS	Minipacket sent to modem
MPR	Minipacket received by the modem
MPST	Start minipacket transmission
MPET	End minipacket transmission
Reception	
RXS	Incoming packet notified
RXMP	Reception of a minipacket

CMD	Name of parameter to set
ADR1	Source
ADR2	Destination
Packet Type	Packet type: – 0 80bps (FH-FSK) – 1 250 bps 1/31 spreading – 2 500 bps 1/15 spreading – 3 1200 bps 1/7 spreading – 4 1300 bps 1/6 rate block code – 5 5300 bps 9/14 rate block code
ACK	Deprecated. Use either 0 or 1.
Npkt	Number of frames to send in packet
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

- Send a binary data packet:
  $CCTXD,SRC,DEST,ACK,HH...H*CS

SRC	Source (4bits)
DST	Dest (4bits)
ACK	Packet Acknowledgement
HH...H	Hex coded data bytes, 2 characters each, e.g. 00-FF.
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

- Send a Minipacket :
  $CCMUC,SRC,DST,BHHH*CS

SRC	Source (4bits)
DST	Dest (4bits)
BHHH	ASCII coded hex data (2 hex values). Value in the range of 0 to 1FFF are legal
CS	Hex coded checksum (8 bit XOR of sentence). Optional.

- Modem reports Cycle Init reception:
  $CACYC,CMD,ADR1,ADR2,Packet Type,ACK,Npkt*CS
  (see **CCCYC**)

- Minipacket acoustically received:
  $CAMUA,SRC,DST,HHHH∗CS
  (see **CCMUC**)

- Modem reports received data in binary format:
  $CARXD,SRC,DST,ACK,F#,HH...H∗CS

SRC	Source
DST	Destination
ACK	ACK bit set by transmitter (0 or 1)
F#	Frame number
BHHH	Hex coded data bytes (2 hex value each), e.g. 00-FF to represent nummbers from 0 to 255.
CS	Hex coded checksum (8 bit XOR of sentence).

7.5 mserial

This class inherits from *UWMconnector*, and it allows the host to communicate with a corresponding connected modem via serial connection. According to the WHOI Micromodem's specifications, the parameters of the serial connection are:

Bits per second	19200bps
Data bits	8
Parity	none
Stop bits	1
Flow control	none

Actually, the bit rate could be increased up to 115200bps, but this would require to change the corresponding modem's setting.

`mserial` starts a connection by opening the serial port file and by properly setting the termios parameters. At the same time, a parallel reading thread is started using *pthreads*: this thread is in charge to read new strings from modem to host, it uses continuously the `read` method and it saves the incoming strings into a dedicated file (which acts as a buffer). That buffer is checked periodically (ns2 handles the whole scheduling) by the `UWMPhy_modem` base class.

To write to the modem is even simpler, as `mserial` uses the `write` termios method. The string to send trough the serial port is the output of the interpreter (NMEA legal sentence) to which are added the terminators $'\backslash r','\backslash n'$ and $'\backslash 0'$

Once the connection is closed, `mserial` handles the closure of the RS-232 communication resetting termios parameters.

8 Interface specialization: the PSK Micromodem case

`mPSK_WHOI_MM` inherits from `UWMPhy_modem` and it is meant to be used with the PSK version of WHOI Micromodem. The PSK module is slightly different from the FSK one, therefore we created a different NS-Miracle object, that is loadable in Otcl scripts with the following command:

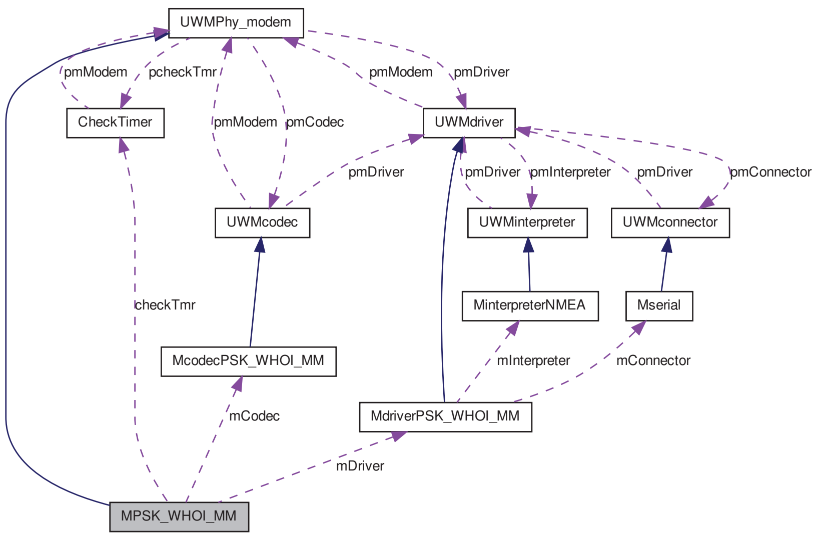


Figure 9: mPSK_WHOI_MM class hierarchy and pointers

```
set phy [new Module/UW/MPhy_modem/PSK_WHOI_MM serialPath]
```

where `serialPath` has the same role as in the FSK module.

8.1 mcodecPSK_WHOI_MM

This class inherits from `UWMcodec` and it works differently than `mcodecFSK_WHOI_MM`. First, we choose to use only the binary data packet to exploit the larger payload, second, the payload is composed serializing all the headers of the modules needed by the tested protocol. The routine implemented for such serialization is as follow:

```
$char * x;$
$std :: stringstream\ str;$
$hdr\ h = header;$
$x = (char*)h;$
for $i = 0 \to sizeof(h) - 1$ do
    for $k = 7 \to 0$ do
        if $*(x + i)\&(1 \ll k)$ then
            $str \ll 1$
        else
            $str \ll 0$
        end if
    end for
end for
```

The algorithm works in this way:

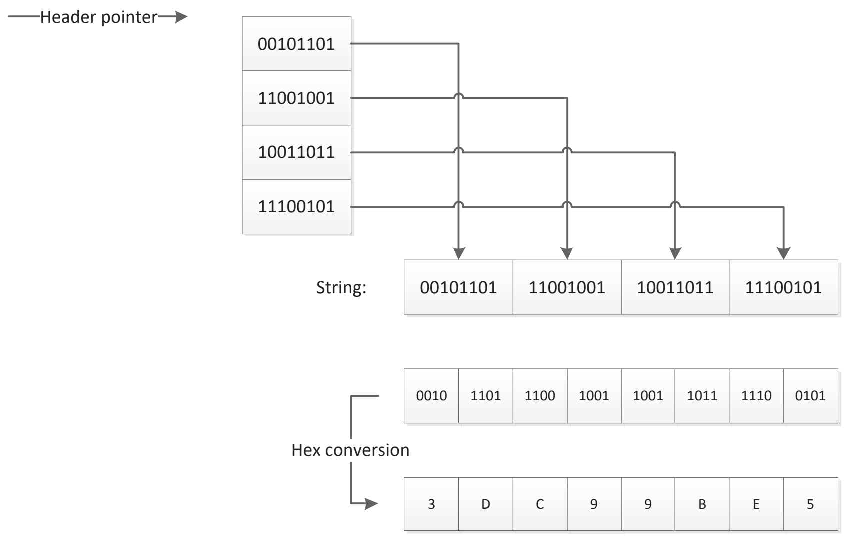


Figure 10: Header serialization procedure

- h is a pointer to a header (hdr), e.g., the header we want to include in the payload;

- the pointer h is casted to a char pointer;

- the bitwise operation inside the inner `for` cycle is responsible for the serialization process: it reads; the char pointed by $x + i$ from the most significant bit to the last significant one and it appends these bits to the stringstream `str`.

The result of this operation is a binary string containing a copy of the memory area where a given header is saved, then the string is hex encoded in order to create a legal payload for the *$CCTXD* NMEA string (see Figure 10). The serialization is performed for every header that must be sent to the receiver for the overall success of the transmission.

On the receiver side, the operation is quite the opposite. After the conversion from hex to binary data, the node allocates a new packet with uninitialized headers, then the following algorithm is called:

```
$char* \; x;$
$hdr \; h = header;$
$x = (*char)h;$
**for** $i = 0 \to sizeof(h) - 1$ **do**
    $j = 0$
    **for** $k = 0 \to 7$ **do**
        $h = received\_string[8 * i + k] == 1?1 : 0$
        $j+ = h \ll (7 - k)$
    **end for**
    $*(x + i) = j$
**end for**
```

The algorithm's target is to fill a header with the informations received in the packet's payload. It works in as follow:

- it reads from the *received_string* groups of eight characters, where each one of them is interpreted as a bit;
- the first bit of the group is the most significant one and accordign to this order we recover the corresponding `char`;
- these reconstructed `char`s are written in the memory starting from the first memory address of the header, till the end of it;
- this operation will overwrite the header's memory area with the acoustically received bits.

8.2 mdriverPSK_WHOI_MM

`mdriverPSK_WHOI_MM` class inherits from `UWMdriver` and it respects the state machine shown in Figure 4. Moreover, to manage the transmission/reception of WHOI Micromodem binary data packet, we implemented two other state machines shown in Figure 11 and 12. The main difference with `mdriverFSK_WHOI_MM` is that these two state machines have a loop: in fact, the informations sent with a binary data packet could be split into frames and those loops regard the sending/receiving of multiple frames. The cycle $BIN \to BINS \to BINR \to BIN \dots$ (whose labels are explained in Table 10) persists until `mPSK_WHOI_MM` runs out of frames to send, meanwhile the cycle $RXBIN \to RXBIN \dots$ stops when it gets a number of frames equal to the one declared by the antecedent Cicle Init (supposing that no errors occur).

8.3 minterpreterNMEA

This class is the same for the FSK and PSK WHOI Micromodem, since both hardwares use the same kind of messages (see Section 7.4).

8.4 mserial

This class is the same for the FSK and PSK WHOI Micromodem, since both hardwares are connected to host via serial connection (see Section 7.5).

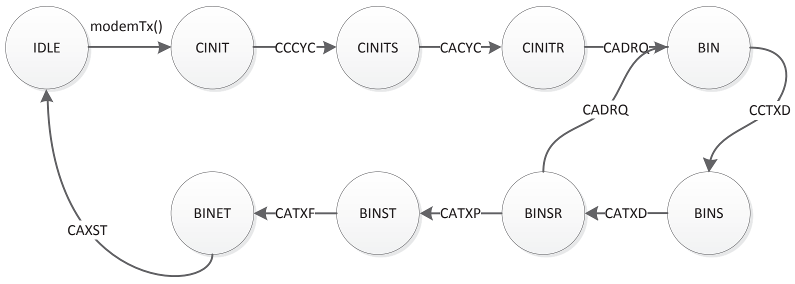


Figure 11: State machine implemented for the transmission of binary data

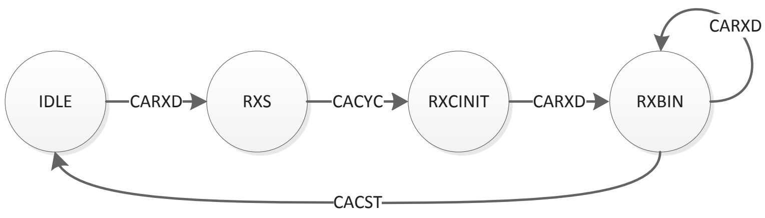


Figure 12: State machine implemented for the reception of binary data

Transmission	
CINIT	Send cycle init
CINITS	Cycle init request sent to modem
CINITR	Cycle init request received by the modem
BIN	Free to send binary data
BINS	Binary data sent to modem
BINSR	Binary data received by the modem
BINST	Start binary data transmission
BINET	End binary data transmission
Reception	
RXS	acoustic signal reception detected
RXCINIT	reception of cycle init message
RXBIN	reception of binary data message

Table 10: State machine labels

9 Goby Software

The Goby Underwater Autonomy Project is a software developed by Toby Schneider from MIT [32] and it aims at creating a unified framework for multiple scientific autonomous marine vehicle collaboration, seamlessly incorporating acoustic, ethernet, wifi, and serial communications. Presently the main thrust of the project is developing a set of robust acoustic networking libraries, including the Dynamic Compact Control Language (DCCL) and relies on simple extensions to Google Protocol Buffers. The advantage of using this framework to interface the modem with the simulator is that the code to communicate with the modem is already available and simplifies considerably our module. A first temptation of interfacing NS-Miracle with the WHOI's hardware using Goby has been done without inheriting from `UWMPhy_modem` (since this latter was not already fully defined), therefore a first beta version of our interface including Goby exists, it is called `GobyModule` and is composed of a single class (it is still a beta version). It was used mainly with the develop box (a small box developed by WHOI for testing purpose containing two modems [37]) to get familiar with the interaction between the network simulator and the PSK WHOI Micromodem. Future work on this module include the code reformatting of this beta version to follow the logic structure of *UWMPhy_modem* and testing it in a testbed setting

The user should prefer Goby if the overall system already uses it (that means it is already installed) or if future improvements require the physical module to interact with other devices that use Goby's framework.

To load the module, write in the OTcl script:

```
set phy [new Module/UW/GobyModule serialPath]
```

where `serialPath` is the path of the serial port in use (e.g. `/dev/ttyUSB0`).

In transmission, the module receives a packet from above (e.g. MAC layer), formats it into an NMEA string, sends the string through the serial port. In reception, this module keeps checking the serial for an incoming packet, once received it parses the payload and allocate a new packet filled with the received informations and it sends it up to the MAC module.

Goby manages the composition of the NMEA message and uses its own state machine to interact with the modem, that is going through the sequence of messages listed in Table 2 and 3, both in transmission and reception.

The rest of the code regards the mapping and demapping of the packet as well as the splitting in multiple frames, that is the same as we described for the previous modules.

Part III
Tests and conclusions

In order to test the implemented interface and its good functioning, in collaboration with WHOI, we have conducted a series of field tests detailed in this Part.

Table 11: Stacks protocols used during the sea-tests

Stack 1	Stack 2
CBR	CBR
UDP	UDP
	StaticRouting
IP	IP
MLL	MLL
CSMA-ALOHA	CSMA-ALOHA
mPSK_WHOI_MM	mPSK_WHOI_MM

10 Network protocols and metrics

Let us define 2 network stacks (see Table 11), each useful to check different aspects of the correctness of our work.

Stack 1 is composed of an application layer (CBR) that generates packets with a constant bit rate, an UDP transport layer, an IP interface, a media link layer protocol (MLL), a CSMA MAC layer and the interface module we developed for the PSK Micromodem. Stack 2 adds the static routing module, useful for multi-hop network topologies. The first stack has been chosen to:

- check the hardware and the operating system in use;
- to test the interface between NS2 and Micromodem (PSK) in a point-to-point communication;
- to verify the correctness (and right tuning) of MAC protocols in point-to-point communications;
- to evaluate the benefits of carrier sensing when both the senders want to transmit.

Stack 2, instead, has been adopted to test the behaviour of multi-hop underwater networks. In this case the routing is static such as we can force certain routes inside the network (see Section 13).

In both cases the considered metrics at the application level are:

- *Packet Loss [%]*:

$$\frac{\Sigma packet_{lost}}{\Sigma packet_{recv} + \Sigma packet_{lost}}$$

- *Delivery Delay [s]*: end-to-end delay, it is the difference between the arrival time and the sending time (it considers the HW/SW delays too)

$$Arrivaltime - Sendingtime$$

- *Throughput [bps]* (normalized by hop count): Correctly received bits, normalized to real packets transmission time (seconds).

$$\frac{\Sigma bytes_{ok}}{\Sigma delay_i}$$

- *Mean number of retransmissions* for every packet sent (since CSMA-ALOHA has been running with an automatic retransmission request mechanism).

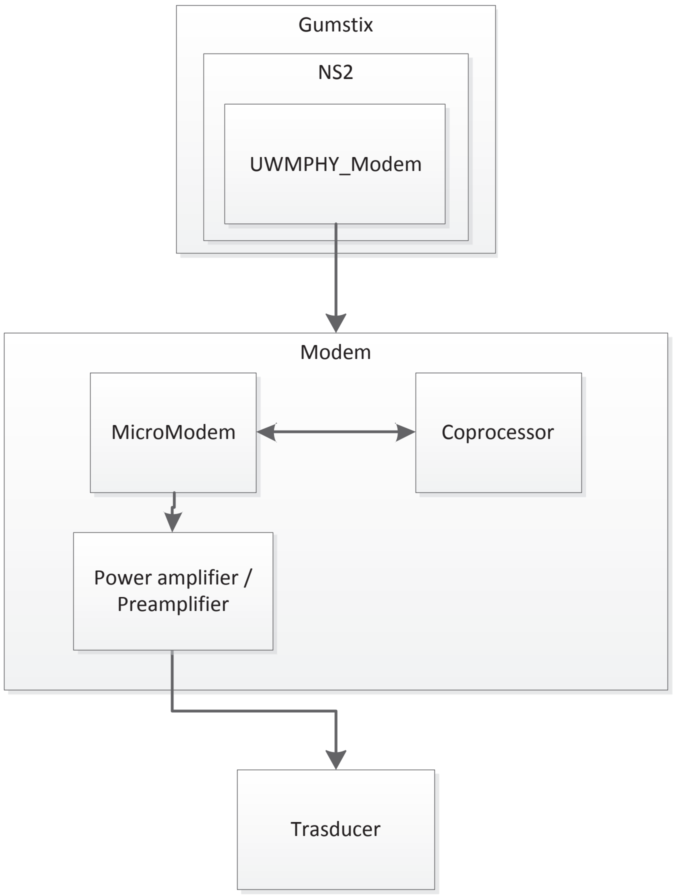


Figure 13: Software and Hardware scheme of the physical node

11 Hardware and Software

We have built 7 nodes which are very small and easy to deploy by hand, for example by suspending them from a dock or other existing infrastructure. Each node includes a WHOI Micromodem [20] with its floating-point coprocessor board to allow PSK packet reception, as well as an optional multi-channel receive array to improve PSK receive performance. The transducer center frequency is nominally $25kHz$ with $5kHz$ bandwidth. Burst data rates range from $80bps$ to $5400bps$. The power amplifier is a low-power ($150dBre : 1\mu Pa@1m$) linear power amplifier (maximum link ranges on the order of $500 - 1000m$) to allow multi-hop networks to be deployed in a relatively small physical area and to increase deployment duration with the limited batteries in a small pressure housing. The pressure housings are rated to $100m$. The deployment duration is on the order of 100-200 hours, depending on the network's offered load and node settings.

In each testbed node, the Micromodem is controlled by a Gumstix

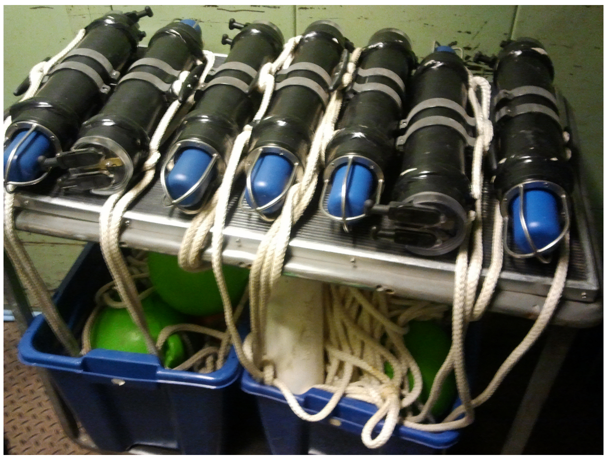


Figure 14: Picture of the 7 nodes

(http://gumstix.com) embedded computer running Emdebian (embedded version of the operating system Debian). The Gumstix has been chosen due to the low power consumption (1.4-2.5 Watts), the well known programming of the expansion board (widely used at WHOI) and the network capabilities (ethernet and 802.11). On the other hand, Emdebian offers the ease of deb packages installation, but it lacks of online support and we experienced some network problems. We set up a wifi connection with the nodes when out of water, such that the embedded systems are accessible through a SSH connection. The hardware runs flawlessly NS-Miracle simulator, and, we execute the tests listed below with the usage of `cron` (a time-based job scheduler in Unix-like computers . At the end of the tests, once the nodes are recovered, the collected tracefiles (the output of NS-Miracle) could be copied to a PC both using wifi or removing the SD cards from the Gumstix.

12 Location

The testbed has been deployed mainly around the dock behind WHOI's Smith Laboratory (41°31'27.5"N, 70°40'15.5"W), for security reasons. In Figure 16 the positions of the nodes are marked.

Even if the area is small, the shallow water, the concrete dock and piles and occasional obstacle like docked ships, make the communication challenging.

As mentioned above, the nodes are equipped with a WiFi antenna, that makes the access to the Gumstix immediate with the constrain that the hardware must be out of water. The nodes' plastic housing is almost irrelevant in terms of attenuation, so that the range is high enough to allow the user to SSH them without moving around the dock.

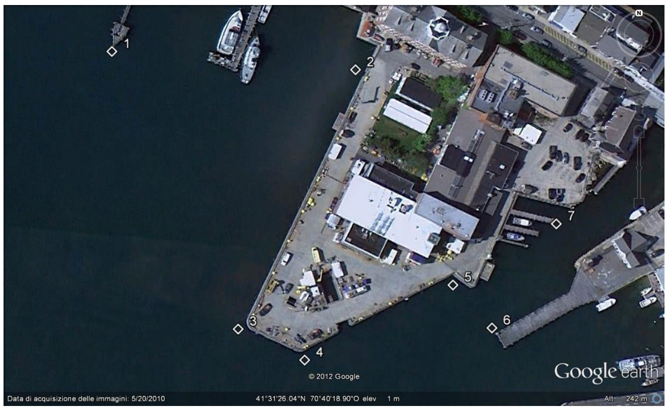

1
2
3
4
5
6
7
© 2012 Google
Google earth
Data di acquisizione delle immagini: 5/20/2010
41°31'26.04"N 70°40'18.90"O elev 1 m
Alt 242 m

Figure 15: Position of the nodes in the testbed deployed at WHOI

Figure 16: Picture of the dock where we deployed the nodes

13 Network topologies and experiments description

NOTE: see legend in Figure 17.

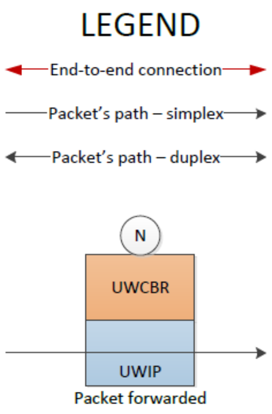


Figure 17: Network topologies legend

Test 1 The topology in Figure 18 uses stack 1. It is useful to discover any software bugs and it is really fast to implement and run it.

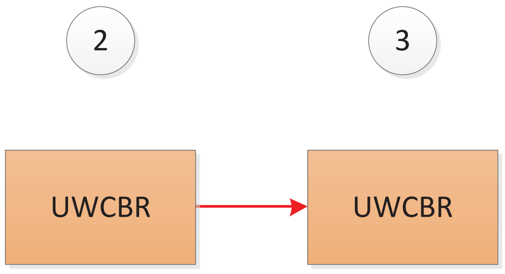


Figure 18: Test 1: single hop

Test 2 Figure 19 represents a classic scenario where the MAC layer is stressed: nodes 3 and 5 start sending packets at the same time in order to congest the channel, while the CSMA-ALOHA is exploit to reduce the packet losses. Stack 1 is used also in this test.

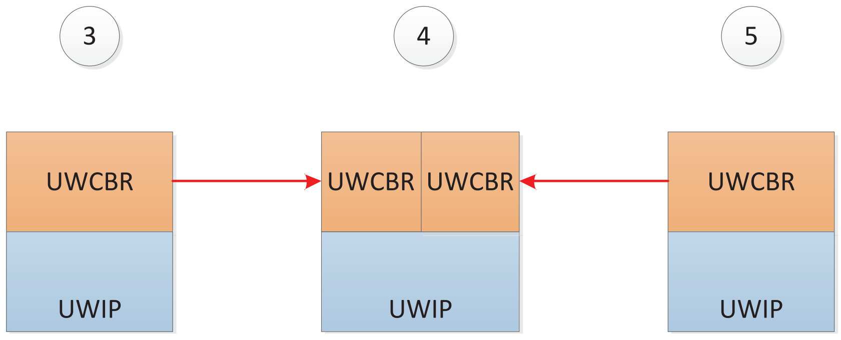


Figure 19: Test 2: simple MAC test

Test 3 To stress even more the MAC module, we test it in a network deployed as in Figure 20, where multiple nodes are connected in a single hop fashion and transmit simultaneously. Stack 1 is used.

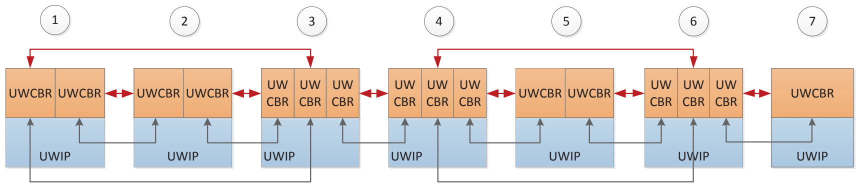


Figure 20: Test 3: MAC stress test

Test 4 The *linear topology* shown in figure 22 is meant to send packet back and forth between the ends of the network (node 1 and 7 exploit multi-hop transmissions): this kind of test reflects one of the most common scenarios in UW networks, where the nodes are placed along a line (e.g. a pipe) to reach further point of interest, out of the transducer range. This test uses stack 2 and is preceded by a check on every link (depicted in Figure 21).

Test 5 We also split the network in two clusters of nodes (stack 2), both sending at the same destination (figure 23). Likely the previous test, this is a very common network as the sink can be one only and there may be multiple areas monitor, each one with its own devices.

Test 6 With two clusters and two sinks, we also tested a scenario with an overlapping area that involves some collisions and packet losses. Nodes use stack 2. See figure 24.

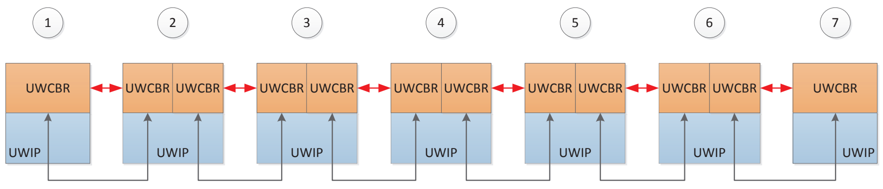


Figure 21: Test 4a: sanity chech

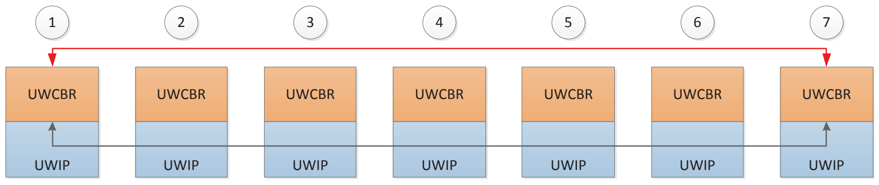


Figure 22: Test 4b: linear topology

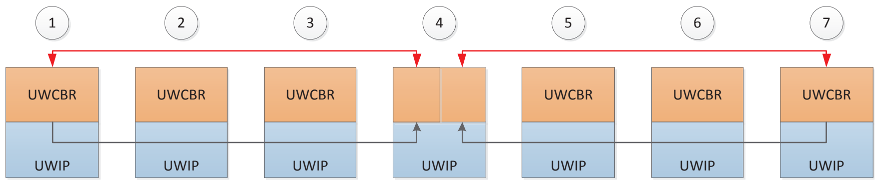


Figure 23: Test 5: two clusters, one sink

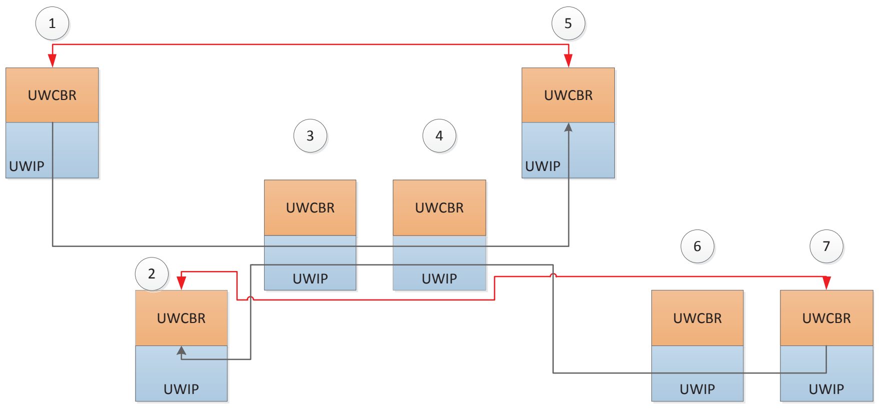


Figure 24: Test 6: two clusters, two sinks. With overlapping area.

14 Results

Before proceeding further, we list some considerations about the tests we performed:

Table 12: Module/UW/MMac/CSMA parameters

listen_time_	3
ACK_timeout_	10
max_tx_tries_	5
wait_costant_	7

Table 13: Test 1 results

PER (%)	0
Mean delay (s)	14,1
Mean throughput (bps)	327.35
Mean number of rtx	2

- we needed several tries before getting satisfying results. MAC module's parameters largely affect the results and system performances, as it is the only one that introduces delays comparable with the transmission time;
- the main problem we faced is the channel access, because of the channel proprieties introduced in Section 1. Even if the network is not extended over a big area, the transmission time has a big overhead (time wise speaking), and this makes necessary to introduce long time guards to avoid collisions;
- the binary data packet type is suitable for large payloads, theoretically it has a high baud rate, but the transmission needs a long waiting time to listen to the channel and MAC module expects an acknowledgement (ACK) (that is sent without any channel sensing);
- one packet transmission time (send plus ACK) with the CSMA parameter in 12 is about 12 seconds with packet rate 4 (1300bps) and 8 seconds with packet rate 5 (5300bps);
- we decided to keep the acknowledgement on because of the potential errors and interferences, so the packets that run across the network duplicate.

In the next section we present the numerical results obtained in our sea-tests.

14.1 Packet loss, delay, throughput, retransmissions

14.1.1 Test 1

For the first test we reduce the CSMA waiting time to 4 seconds (the same value is used in test 2), because there are just few packets to be sent. Table 13 summarize values recorded for the considered metrics.

We noticed that, since the receiver is deeper than the transmitter, the transmission works better "downhill" (from shallow to deep water) than "uphill", and for this reason the network looses more ACKs than packets.

Table 14: Test 2 results

Sender (nodeID)	3	5
PER (%)	0	0.16
Mean delay (s)	41.8	39
Mean throughput (bps)	223	374
Mean number of rtx	1.43	2.66

14.1.2 Test 2

This test tries to repeat the hidden terminal scenario. The CSMA we use does not implement the RTS/CTS protocol, so we rely to the backoff times in case of packet loss. The results are reported in Table 14.

As with the first test, we notice that the acknowledgements suffer more than the packets.

14.1.3 Test 3

This critical test shows that it is possible to arrange a quite successful transmission paying in terms of low throughput and high delays.

We report the results in Table 15, 16, 17 and 18.

FROM/TO	1	2	3	4	5	6	7
1		0.05	0.25				
2	0.07		0.15				
3	0.08	0		0.27			
4			0.27		0.3	1	
5				0.66		0.16	
6				1	0		0
7						0.26	

Table 15: Test 3: Packet error rate (%)

FROM/TO	1	2	3	4	5	6	7
1		14.17	40.86				
2	8.84		27.12				
3	22.27	12.45		27.87			
4			85.62		47.14	∞	
5				15.67		36.10	
6				∞	22.78		19.06
7						23.76	

Table 16: Test 3: Mean delivery delay (s)

FROM/TO	1	2	3	4	5	6	7
1		494	358				
2	302		257				
3	242	415		270			
4			139		108	0	
5				116		237	
6				0	326		363
7						229	

Table 17: Test 3: Mean throughput (bps)

FROM/TO	1	2	3	4	5	6	7
1		2.66	2.4				
2	2.57		2.5				
3	2.41	2.45		2.73			
4			2.82		2.7	3	
5				2		2.58	
6				3	2.11		1.66
7						2.17	

Table 18: Test 3: Mean number of transmissions

14.1.4 Test 4

We found that the main issue with linear topology is to have all the links working, in fact nodes were not able to transfer a packet successfully between node 1 and 7. The scenario, even if the distances are really small, is challenging, and the link between two nodes is not always good as expected. The SNR is high enough to make us believe that the problem relies mainly on shallow water and shadows areas.

14.1.5 Test 5

The numerical results are shown in Table 26, 27 and 28. This test reported the same problems as Test 4, in particular the link 5 $\rightarrow$ 4 was broken.

FROM/TO	1	2	3	4	5	6	7
1		0					
2	0		0.4				
3		0		0.16			
4			0.08		1		
5				1		0.36	
6					0		0
7						0	

Table 19: Test 4a: Packet error rate (%)

FROM/TO	1	2	3	4	5	6	7
1		9					
2	17.42		13.66				
3		8.15		11.66			
4			9.91		∞		
5				∞		16.78	
6					6.64		6.5
7						12.76	

Table 20: Test 4a: Mean delivery delay (s)

FROM/TO	1	2	3	4	5	6	7
1		574					
2	259		202				
3		429		323			
4			388		0		
5				0		228	
6					436		446
7						370	

Table 21: Test 4a: Mean throughput (bps)

FROM/TO	1	2	3	4	5	6	7
1		1.21					
2	1.28		1.93				
3		1.47		1.61			
4			1.33		2		
5				2		1.42	
6					2		1.1
7						1.35	

Table 22: Test 4a: Mean number of transmissions

FROM/TO	1	2	3	4	5	6	7
1		0.02	0.73	0.73	0.87	1	1
2							
3							
4							
5							
6							
7	1	1	1	1	0	0	

Table 23: Test 4b: Packet error rate (%)

FROM/TO	1	2	3	4	5	6	7
1		25	109	117	128	∞	∞
2							
3							
4							
5							
6							
7	∞	∞	∞	∞	15	8	

Table 24: Test 4b: Mean delivery delay (s)

FROM/TO	1	2	3	4	5	6	7
1		113	34	34	34	0	0
2							
3							
4							
5							
6							
7	0	0	0	0	360	360	

Table 25: Test 4b: Mean throughput (bps)

FROM/TO	1	2	3	4	5	6	7
1		0.38	0.67	0.87			
2							
3							
4							
5							
6							
7				1	0.67	0.42	

Table 26: Test 5: Packet error rate (%)

FROM/TO	1	2	3	4	5	6	7
1		17	72	165			
2							
3							
4							
5							
6							
7				∞	60.7	24.1	

Table 27: Test 5: Mean delivery delay (s)

FROM/TO	1	2	3	4	5	6	7
1		351	186	186			
2							
3							
4							
5							
6							
7				0	227	271	

Table 28: Test 5: Mean throughput (bps)

14.1.6 Test 6

This test highlights, as expected, that the packet loss is really high in correspondence of the area where the two clusters overlay (see Tables 29 to 31.

FROM/TO	1	2	3	4	5	6	7
1		0.6	0.94	0.95			
2							
3							
4							
5							
6							
7		0.99	0.99	0.97		0.14	

Table 29: Test 6: Packet error rate (%)

FROM/TO	1	2	3	4	5	6	7
1		39	151	175			
2							
3							
4							
5							
6							
7		168	79	72		24	

Table 30: Test 6: Mean delivery delay (s)

15 Final considerations

The interface we developed showed a high usability and ease of implementation of new modules, as well as correct operation through field experiments.

The nodes in use has proven to have some software problems related to EmDebian:

FROM/TO	1	2	3	4	5	6	7
1		80	80	80			
2							
3							
4							
5							
6							
7		154	261	261		261	

Table 31: Test 6: Mean throughput (bps)

- because of the lack of drivers (e.g. ethernet and real time clock), some random crashes totally compromised several testing sessions. In fact, there is no real time clock (RTC) on the board, therefore they get the time from the network on every boot through wifi connection. Of course, when deployed there is no wifi and in case of crash the nodes set the date randomly. Even backing up RTC with a battery, the kernel does not load the driver correctly, and the only clock in use is the linux's one;
- the linux's clock has a huge drift. Initially the longest tests were affected by this problem (collecting the data from multiple nodes, we notices some timestamps of received packets to precede the sending time of the same packet), so we had to run tests for a shorter period of time and re-sync the clock before new ones;

The testbed setup is relatively quick and, thanks to the wireless connection, it is possible to program the nodes without opening the housing (this speeds up a lot the preparation of the hardware). Furthermore, the energy consumption of the Gumstixs is really acceptable, giving the possibility to run tests for 72 hours (with a sending rate of 4 packets per minute). As expected, the main problem relies on the position and depth of the nodes, because it is easy to erroneously deploy a node in a shadow area.

The future work on our module regards the improvement of the payload creation (shrinking its size) and an improved coding on 13 bits for Mini-packet's payload. Moreover, the hardware requires a better clock management and a better understanding of the causes of the random crashes we experienced.

The performed sea-tests allow us to to assess the feasibility of the implemented interface for network prototyping. These tests allow us to successfully perform single-hop as well as multi-hop transmissions using the same code implemented in NS-Miracle for simulation purposes. We believe that this work along with its future development, represent a fundamental step for the study of effective underwater network protocols, moving from simulations to the real world.

References

[1] J. Heidemann A. A. Syed W. Ye. *T-Lohi: A new class of mac protocols for underwater acoustic sensor networks.* Technical report ISI-TR-638. 2007.

[2] *ALOHA MAC.* Last time accessed: March 2012. URL: http://en.wikipedia.org/wiki/ALOHAnet#The_ALOHA_protocol.

[3] *AUV Laboratory at MIT Sea Grant, Available from.* Last time accessed: March 2012. URL: http://auvlab.mit.edu/.

[4] M. Stojanovic B. Peleato. "Distance aware collision avoidance protocol for ad-hoc underwater acoustic sensor networks". In: *IEEE Communication Letters* 11.12 (2007), pp. 1025–1027.

[5] J. Shusta C. Petrioli R. Petroccia and L. Freitag. "From underwater simulation to at-sea testing using the ns-2 network simulator". In: *OES/IEEE OCEAN 2011* (2011).

[6] *Carrier sense multiple access MAC.* Last time accessed: March 2012. URL: http://en.wikipedia.org/wiki/Carrier_sense_multiple_access.

[7] T. Clausen and P. Jacquet. *Optimized Link State Routing Protocol (OLSR).* RFC Editor, 2003.

[8] J. Broch D.B. Johnson D.A. Maltz. *DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks.* Addison-Wesley, 2001.

[9] *DPSK modulation.* Last time accessed: March 2012. URL: http://en.wikipedia.org/wiki/Phase-shift_keying.

[10] Y. Dogan V. Coskun E. Cayirci H. Tezcan. *Wireless sensor networks for underwater surveillance systems, Ad Hoc Networks.* Elsevier, 2006.

[11] *Emdebian website.* URL: http://www.emdebian.org/.

[12] B. Davis G. Kennedy. *Electronic Communication Systems.* 4th ed. McGraw-Hill International, 1992.

[13] J.H. Gibson G. Xie. "A network layer protocol for UANs to address propagation delay induced performance limitations". In: *IEEE OCEANS'01.* Vol. 4. Honolulu, HI, 2001, pp. 2087–2094.

[14] *Gumstix.* Last time accessed: December 2011. URL: http://www.gumstix.com/.

[15] John Heidemann et al. "Research Challenges and Applications for Underwater Sensor Networking". In: *Proceedings of the IEEE Wireless Communications and Networking Conference.* Las Vegas, Nevada, USA, 2006.

[16] *IEEE 802.11 RTS/CTS.* Last time accessed: March 2012. URL: http://en.wikipedia.org/wiki/IEEE_802.11_RTS/CTS.

[17] Y. Sankarasubramaniam E. Cayirci I.F. Akyildiz W. Su. "Wireless sensor networks: A survey". In: *Computer Networks* 4.38 (2002), pp. 393–422.

[18] M. Zorzi J. Heidemann M. Stojanovic. *Underwater Sensor Networks: Applications, Advances, and Challenges.* Last time accessed: March 2012.

[19] M. Younis K. Akkaya. “A survey on routing protocols for wireless sensor networks”. In: *Ad Hoc Networks* 3.3 (2005), pp. 325–349.

[20] J. Partan E. Gallimore S. Singh P. Koski L. Freitag K. Ball. *Underwater Acoustic Network Testbed. Wuwnet 2011 demo extended abstract.* 2011.

[21] E. Dutkiewicz M. Abolhasan T. Wysocki. “A review of routing protocols for mobile ad hoc networks”. In: *Ad Hoc Networks* 1.2 (2004), pp. 1–22.

[22] F. Guerra M. Rossi M. Zorzi N. Baldo M. Miozzo. “Miracle: the multi-interface cross-layer extension of ns2”. In: *EURASIP J. Wirel. Commun. Netw.* 26 (2010), pp. 1–2.

[23] J. Ware C. Pontbriand M. Tivey N. Farr A. Bowen. “An integrated, underwater optical/acoustic communications system”. In: *OCEANS 2010 IEEE* (2010), pp. 1–6.

[24] *Network Simulator 2 website.* Last time accessed: March 2012. URL: `http://www.isi.edu/nsnam/ns/`.

[25] S.M. Holt N.N. Soreide C.E. Woody. “Overview of ocean based buoys and drifters: Present applications and future needs”. In: *16th International Conference on Interactive Information and Processing Systems (IIPS) for Meteorology, Oceanography, and Hydrology.*

[26] *Ocean Engineering at Florida Atlantic University.* Last time accessed: March 2012. URL: `http://www.oe.fau.edu/research/ams.html`.

[27] *OFDM modulation.* Last time accessed: March 2012. URL: `http://en.wikipedia.org/wiki/Orthogonal_frequency-division_multiplexing`.

[28] *Pandaboard.* Last time accessed: December 2011. URL: `http://www.pandaboard.org/`.

[29] C. Perkins, E. Belding-Royer, and S. Das. *Ad hoc On-Demand Distance Vector (AODV) Routing.* United States: RFC Editor, 2003.

[30] Charles E. Perkins and Pravin Bhagwat. “Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers”. In: *SIGCOMM Comput. Commun. Rev.* 24.4 (1994), pp. 234–244.

[31] *QAM modulation.* Last time accessed: March 2012. URL: `http://en.wikipedia.org/wiki/Quadrature_amplitude_modulation`.

[32] Toby Schneider. *Gobysoft web page.* Last time accessed: March 2012. URL: `http://gobysoft.org/`.

[33] *Second field test for the AOSN program, Monterey Bay August 2003, Available from.* Last time accessed: March 2012. URL: `http://www.mbari.org/aosn/MontereyBay2003/MontereyBay2003Default.htm`.

[34] M. Stojanovic. *Acoustic (underwater) communications: Encyclopedia of Telecommunications*. Wiley, 2003.

[35] *The Network Simulator - NS-Miracle*. Last time accessed: March 2012. URL: `http://dgt.dei.unipd.it/download`.

[36] *The NMEA FAQ*. Last time accessed: March 2012. URL: `http://www.kh-gps.de/nmea.faq`.

[37] WHOI. *Micromodem develop box*. Last time accessed: March 2012. URL: `acomms.whoi.edu/umodem/documentation.html`.

[38] *Wikipedia web page*. Last time accessed: March 2012. URL: `http://en.wikipedia.org/wiki/OSI_model`.

THANKS

Thank you Jim Partan for being a fundamental guide during the whole thesis, in particular for being patient, mindful, amazingly kind and for all the support you gave me. Thanks also to Lee, Eric, Sandipa, Andrew, Peter and Keenan, great workers, respectable and respectful people.

Thank you Riccardo Masiero, helpful project colleague, firm correlator and pleasant friend.

Thanks to Michele Zorzi and Paolo Casari for making a big dream come true and for the trust given to me. Living in the US made me a little more mature and made me understand some fundamental aspects of life.

Thank you Mattia for the time we spent together in Padua, for being always a close friend and forbearing house mate. It is exciting to share part of my life with a trusted friend.

Thanks to Fabio, Riccardo, Paolo, Marco, Nicola, Eleonora and Chiara, amazing class mates and new friends. Thank you for the time we spent together, the lunches and dinners and the laughter.

Thank you family, Romana, uncles, for being always close and for supporting my job and dreams. Thank you for having raised me as I am and to urge to follow my wishes.

Thanks to Topper, for the never ending chats about life and feelings, for the regattas and the time we spent anchored with the Strega. Downwind to you, always.

Last but not least, thank you infinitely Laura, enviable girlfriend, for being always part of my life even from the other side of the world, for believing in me and for your s smiles during these past months and the ones yet to come.