SIMULATION OF MOBILE HYDROACOUSTIC COMMUNICATIONS IN UNDERWATER ACOUSTIC SENSOR NETWORKS

by

Bita Hasannezhad

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
September, 2015

Table of Contents

List of Figures

Dedicated to my beloved spouse, my devoted parents, and my great supervisor

List of Acronyms

AMOUR Autonomous Modular Optical Underwater Robot

AUV Autonomous Underwater Vehicle

AWGN Additive White Gaussian Noise

BEM Basis Expansion Model

CFD Computational Fluid Dynamics

DART Deep-ocean Assessment and Reporting of Tsunamis

DOP Dilution of Precision

DSSS Direct-Sequence Spread Spectrum

FPGA Field-Programmable Gate Array

FSK Frequency-Shift Keyed

GNED Graphical NEtwork Description

GPS Global Positioning System

GSL GNU Scientific Library

IDE Integrated Development Environment

MAC Medium Access Control

MATLAB MATrix LABoratory

MCM Meandering Current Mobility

MCR MATLAB Compiler Runtime

MoBAN Mobility Model for Body Area Networks

NED NEtwork Description

NIC Network Interface Controller

OFDM Orthogonal Frequency Division Multiplexing

OSTBC Orthogonal Space-Time Block Coding

PPP Point-to-Point Protocol

PSK Phase-Shift Keyed

ROV Remotely Operated Vehicle

SBR-DLP Sector-based Routing with Destination Location Prediction

SLMP Scalable Localization Scheme with Mobility Prediction

TCP Transmission Control Protocol

TDMA Time Division Multiple Access

TWS Tsunami Warning System

UDP User Datagram Protocol

UPS Underwater Positioning Systems

UUV Unmanned Underwater Vehicles

UWASN UnderWater Acoustic Sensor Network

UWSN UnderWater Sensor Network

Abstract

Underwater networks are gaining more attention not only because more than 71% of the Earth's surface is covered with water, but also due to the growing needs of underwater communications. Research on underwater communication techniques plays a most important role for exploring oceans and other aquatic environments. Underwater acoustic sensor network is an area of such research. An UnderWater Acoustic Sensor Network (UWASN) consists of a variable number of self-organized sensors and autonomous vehicles, deployed to monitor and explore the aquatic environment. These autonomous sensors communicate acoustically. We study the software simulation of underwater acoustic communications with mobility in UWASNs. Sensors in the studied networks are mobile due to the marine environments, such as ocean currents. As a consequence, displacements of sensors affect the UWASNs from different aspects, such as data transmission and network lifetime. We study and simulate the mobility of sensors and their communications, moving according to the Meandering Current Mobility (MCM) model. OMNeT++ and MATLAB are used to model and integrate three protocol layers of UWASNs: network, link, and physical. The network layer, simulated using OMNeT++, comprises transmission and reception of packets. The link layer, also simulated using OMNeT++, includes coding and decoding of frames. The physical layer is simulated using MATLAB. It contains models of a modulator, a channel, and a demodulator of underwater acoustic digital data signals. Moreover, the effects of noise and attenuation on the underwater communications are considered in the physical layer model. Two major metrics are calculated in this work: Bit Error Rate (BER) as a function of distance and BER as a function of signal energy per bit over noise power spectral density (E_b/N_0). Finally, the performance of communicating sensors moving according to the MCM model is evaluated. The simulation results illustrate that the BER for digital data signals is an increasing function of the distance. An increase in a transmitter-receiver separation distance, affects the data transmission by increasing the BER. In addition, the BER for digital data signals is a decreasing function of the E_b/N_0 ratio. Also, the simulated BERs are higher than the theoretical BERs, that only take into account white noise.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Michel Barbeau for his guidance and endless support during the completion of this work. I am very grateful for his patience, motivation, and immense knowledge. He is the best professor and teacher who truly made a difference in my life. However, words are not enough to express my gratitude to him. Besides my supervisor, I would like to thank the rest of my thesis committee: Professor Evangelos Kranakis, Professor Stéphane Somé, Professor Jean-Pierre Corriveau, for their insightful comments. I express my thanks to Stéphane Blouin, Joaquin Garcia-Alfaro, and Gimer Cervera. Special thanks to Public Works and Government Services Canada (PWGSC contract $\#W7707-145688/001/HAL$), Natural Sciences and Engineering Research Council of Canada (NSERC) for financial support. Last not least, to my spouce, parents, siblings and friends for all their patience and support.

Chapter 1

Introduction

1.1 Underwater Communications

Underwater communications are about sending and receiving messages in an aquatic environment. There are different options for underwater communications in terms of transmission media such as radio, optical, and acoustic waves. Radio waves are extremely attenuated in water. They propagate underwater at very low frequencies in the range from 30 *Hertz* to 300 *Hertz*. As a result, they need high transmission power and large antennas. On the other hand, optical waves are affected by scattering, although they have less attenuation in comparison with radio waves. As a consequence, using acoustic waves is the best solution for underwater communications [1, 2].

Underwater acoustics, i.e., hydroacoustics, is the study and application of sound propagation in water. In hydroacoustics, propagation of sonar longitudinal waves in an aquatic continuum, and their interaction with boundaries are studied.

The basic hydroacoustic communication model consists of a transmitter with an acoustic modulator and a receiver with an acoustic demodulator (see Figure 1.1). An acoustic modulator is used to transmit data underwater. It converts digital data into underwater sound signals. The received digital data from the *EndUserPC* is converted into underwater sound signals by a *Vibrator*. The *Vibrator* generates acoustic waves at a frequency f (in *Hertz*) by mechanical vibrations. It converts the electric energy into acoustic energy. This conversion leads to creating acoustic pressure in water. The acoustic pressure ($re\,\mu Pa$[1]) refers to the strength of a pressure, generated by acoustic waves. The *Hydrophone* on the receiver side, detects the created acoustic

[1]The term re $\mu\,Pa$ stands for relative to a reference pressure 1 micro Pascal. The pascal is a unit of force per unit area, i.e., pressure. It is equal to one newton per square meter.

pressure. It converts the acoustic energy back into electric energy. The received underwater sound signals are then processed by an *Acoustic Demodulator*. It converts them back into digital data for underwater instrumentation [1].

The hydroacoustics frequencies are mainly in the range from *5 kHz* to *80 kHz*. For example, Teledyne Benthos modems operate in the three different bands: *9-14*, *16-21*, and *22-27 kHz*. EvoLogics' modems operate from 7 to 78 *kHz* [1].

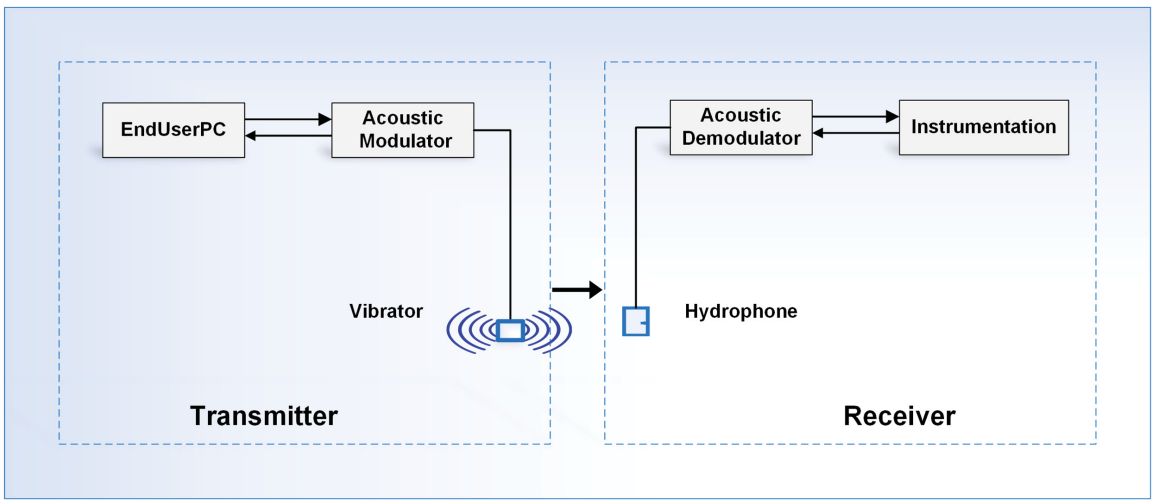

Figure 1.1: Basic hydroacoustic communication model.

Underwater communication channels can be significantly affected by aquatic environments and other phenomena. Harsh conditions make underwater communication channels complex and difficult to control. Underwater acoustic communications are influenced by path loss, noise, and multi-path effects as well as Doppler spread, and high and variable propagation delays. Path loss occurs due to attenuation and geometric spreading. The noise can be either man made or ambient. Furthermore, transmission loss and spreading of sound energy as a result of the expansion of the wave fronts are other factors that can affect underwater acoustic communications [3].

There are some constraints associated to the underwater acoustic signals such as limited bandwidth, high bit error rates, and severe attenuation as well as limited

power resources, and low speed of underwater sound. The speed of sound in seawater is about 1500 meters per second. It is approximately 2.10^5 times lower than the speed of light. Moreover, underwater sensors have high risk of failure due to corrosion of equipment, fouling, and temporary loss of connectivity in shadow zones [4].

1.2 Underwater Acoustic Sensor Networks

An UnderWater Acoustic Sensor Network (UWASN) consists of a variable number of self-organized sensors and autonomous vehicles that are deployed to monitor and explore the underwater environment. It uses acoustic waves for communication over a given underwater area. The *submarine communication system* is one of the first hydroacoustic systems. It has been developed in the United States of America during the Second World War [5, 2].

1.2.1 Applications

Advances in hydroacoustic communication techniques play a significant role in monitoring and exploring oceans and other aquatic environments. There are several advantages in using underwater communications such as providing information by oceanographic observation and data collection in order to find efficient solutions.

The need for underwater communications exists in a variety of applications. These applications are classified into different categories such as military, public safety, industry, and ecology. Military applications are found in national security and defense programs such as anti-submarine missions and tactical surveillance. There are applications in public safety and early warning such as disaster prevention, and hurricane and tsunami monitoring. Moreover, diverse applications of underwater networks are used in industry for offshore exploration, oil and gas industry, and discovery of new natural resources. Last but not least, applications related to ecology, includes pollution, sampling, water quality, and biological monitoring [6].

Underwater communications can be used with other communication media, such as satellite data links, in order to provide data in real-time. For example, the *Tsunami Warning System (TWS)* is used to predict and detect tsunamis with the goal of saving lives and mitigating damages. Tsunami is a series of ocean waves produced by underwater landslides or earthquakes. For this purpose, pressure sensors are used in the high risk zones of seafloor. Pressure data and other necessary information can be transmitted via hydroacoustic waves and underwater modems to the surface buoy. Finally, researchers in *Deep-ocean Assessment and Reporting of Tsunamis (DART)* can receive the collected data in real-time via satellite. This data can be analyzed for predicting tsunamis and early warnings to prevent potential disasters [7, 8].

In addition, underwater communications can be used in detecting seabed and undersea objects. For example, *Robot Crawler* is used for detecting undersea objects. It contains a camera, a modem, and a digital signal-processing unit. It crawls on the ocean floor to investigate objects. It sends images and information via acoustic signals to a shore-based station or a ship [7].

1.3 Mobility

Deployed underwater mobile sensors collect data and monitor the environment. They move around according to the aquatic environment. The displacements of sensor are influenced by water currents and other underwater activities, created by sea creatures and man-made vessels.

The mobility of sensors is an important issue in UWASNs owing to the fact that it affects the network lifetime in terms of connectivity and coverage [6]. In fact, the communications between sensors are influenced by their positions. In addition, the movements of sensors impact other parameters, such as bit error rate and attenuation. Any change in a distance between a source and a destination sensors affects attenuation. In other words, attenuation is sensitive to distance and mobility.

Obtaining the information related to the location of sensors is challenging, considering that the Global Positioning System (GPS) does not work underwater. Underwater GPS devices cannot receive electromagnetic signals from satellites since line-of-sight is required. Electromagnetic signals are immensely damped in water. They cannot penetrate water to any remarkable extent [9].

1.4 Statement of the Problem

UWASNs are expensive and prone to failures. They have some constraints due to limited communication ranges and transmission power while their members are mobile over large areas. The underwater environment makes them challenging to build and deploy. Despite many efforts that have been done in this area (see Section 3) due to constantly growing needs for underwater communications and evolving technologies, there are still several problems and gaps that need more research in order to find efficient solutions.

UWASNs should adjust themselves to the environmental conditions. The mobility makes their design hard due to the large scale and variable underwater environment. These constraints raise a need to simulate UWASNs with a mobility model that reflects underwater conditions in order to probe and improve underwater communications between sensors.

Specific problems addressed in this thesis are:

- How can the mobility of sensors, in an UWASN, be simulated?

- How can the performance of mobile sensors, in an UWASN, be evaluated?

1.5 Overview of Results

OMNeT++ and MATLAB are used to integrate three protocol layers of UWASNs: network, link, and physical. The network and link layers are simulated using OMNeT++. Also, the mobility of sensors in an UWASN is simulated using the Meandering Current Mobility (MCM) model in OMNeT++ (see Section 3.4). The physical layer is simulated using MATLAB functions, which are imported in OMNeT++ as a shared library. The physical layer model, consists of a PSK modulator, a channel, and a PSK demodulator of digital data signals. Noise and attenuation are considered in the channel. The distance between a sender and a receiver is calculated, while they are moving according to the MCM model. The relation between the BER and distance is analyzed. The BER is also examined as a function of the energy per bit to noise power spectral density (E_b/N_0) ratio. Finally, the simulation results are compared with the theoretical performance.

1.6 Organization of Thesis

Chapter 2 provides background on UWASNs. Chapter 3 reviews existing solutions and related works. Chapter 4 presents the underwater mobility simulation. Chapter 5 provides an evaluation of the work. Chapter 6 presents the conclusions. Appendix A reviews the simulation tools and software requirements for this work. Appendix B contains the most relevant sections of the source code of our simulation of mobility and communications in UWASNs. Appendix C contains definitions of concepts used in our work.

Chapter 2

Background

UWASNs use acoustic waves for underwater communications. They are influenced by environmental conditions. Underwater communications are easily impaired by geometrical spreading, noise, and attenuation. This chapter reviews concepts related to acoustic waves and underwater communications.

2.1 Underwater Acoustic Waves

Acoustics are the sound waves in any environment such as air and water. They are categorized as longitudinal or lateral (transverse) waves. However, longitudinal waves are the only ones that work in a fluid medium such as water. Longitudinal waves oscillate the particles of a medium in the direction of wave propagation, while the lateral waves oscillate the particles of a medium perpendicular to the direction of wave propagation (see Figure 2.1).

Depending on the shape of the source and boundary conditions of the medium that acoustic waves are propagated within, they can be further categorized into the followings:

- **Spherical waves** are formed in an unbounded medium with a point source.
- **Cylindrical waves** are formed in medium bounded by two infinite planar surfaces (as in the case of the sea that is bounded by the surface and seabed) with a point source.
- **Planar waves** are theoretically formed by an infinite planar source.

Both spherical and cylindrical wave fronts can be assumed to be planar when the wave from is significantly far from their source (see Figure 2.2).

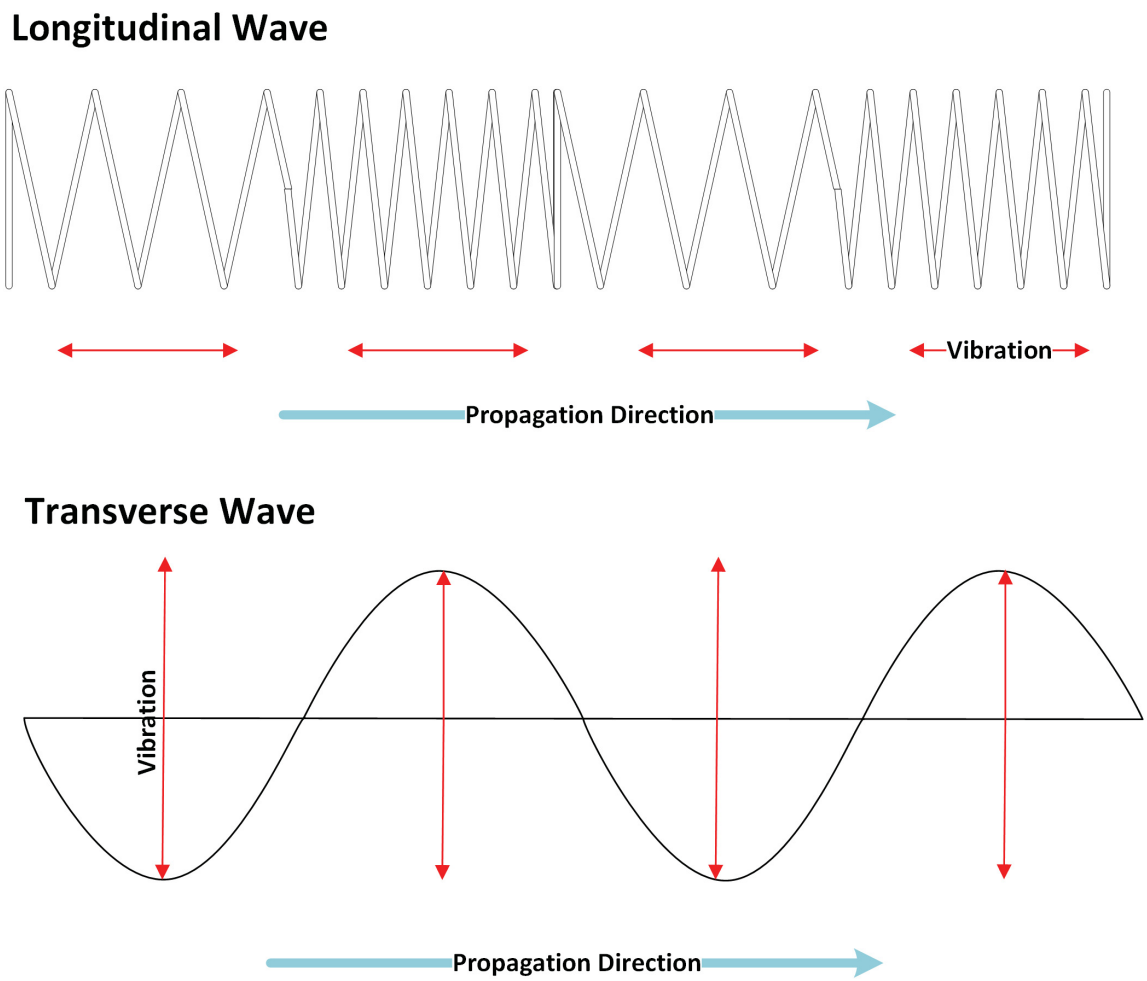

Figure 2.1: Longitudinal wave versus transverse (lateral) wave.

Wavelength refers to the distance between two successive points of the same phase. The following equation represents the relationship between the sound speed c (in *meters/sec*), frequency f (in *Hertz*), and wavelength λ (in *meter*) [10]:

$$c = \lambda \cdot f \quad \text{meters/sec} \tag{2.1}$$

An acoustic wave propagates with a speed between 1450 to 1500 meters per second [1]. In water, the speed of propagation of acoustic waves (c) is a function of ambient

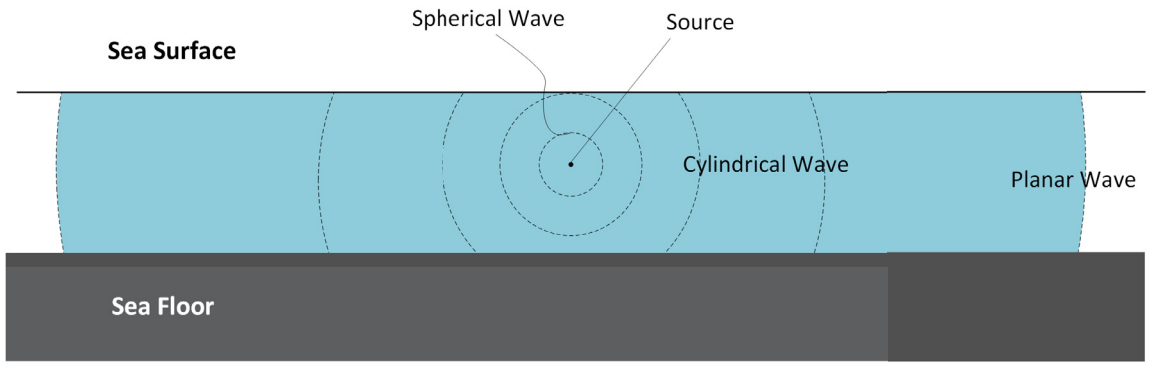

Figure 2.2: Spherical, cylindrical, and planar waves.

temperature (T), pressure (p), and water salinity(S) as shown in the following equation [10]:

$$c = function(T,\, p,\, S) \tag{2.2}$$

In other words, the sound speed varies with changes in the value of these three parameters. Pressure is one of the key parameters in acoustic waves since it is a result of applying a restoring force by water molecules which are moved with the propagation of acoustic waves. Pascal (Pa) is the unit of pressure. One Pa is equivalent to one Newton per square meter (N/m^2). The speed of sound in sea water is determined by the following equation [10]:

$$c(T,S,D) = A_1 + A_2T + A_3T^2 + A_4T^3 + A_5(S-35) + A_6D + A_7D^2 + A_8T(S-35) + A_9TD^3 \tag{2.3}$$

with constant values:

$$\begin{array}{lll} A_1 = 1448.96 & A_2 = 4.591 & A_3 = -5.304 \times 10^{-2} \\ A_4 = 2.374 \times 10^{-4} & A_5 = 1.340 & A_6 = 1.630 \times 10^{-2} \\ A_7 = 1.675 \times 10^{-7} & A_8 = -1.025 \times 10^{-2} & A_9 = -7.139 \times 10^{-13} \end{array}$$

T indicates the temperature in degree Celsius. S represents the salinity in parts per thousand. In this equation, the sound speed c is also expressed as a function of depth in meters (D).

2.2 Underwater Communications

2.2.1 Noise

Noise is a common impairment that distorts the signals during transmission. It is one of the main constraints on attaining efficient communication with high data rate and low error rate within the available bandwidth.

Underwater acoustic communications are influenced by two types of noise: ambient noise and site-specific noise [1]. Ambient noise refers to the residual noise related to the environment. It can be generated by different sources such as water surface waves, rainfall, thermal agitation, turbulence, and shipping. Several studies of the oceans, such as the Northeast Pacific, report a rise in ambient noise in comparison with a few decades ago, on the grounds that ocean traffic and shipping have been increased [10]. Site-specific noise is related to the geographical conditions of the area. Iceberg cracking and aquatic creatures are two sources of site-specific noise.

Water surface waves are one of the main sources of ambient noise. Wave noise (wn) in the $dB\, re\, \mu Pa\, per\, Hz$ form can be obtained using the following equation [1]:

$$wn(f)_{dB\, re\, \mu\, Pa\, per\, Hz} = 50 + 7.5\sqrt{w} + 20 log f - 40\, log(f + 0.4) \qquad (2.4)$$

$re\ \mu Pa$ is reference pressure one micropascal. f is the frequency in kiloHertz (kHz). w represents the wind speed in meters per second ($meters/sec$) [1].

Thermal noise or white noise is an electronic noise resulting from thermal agitation of electrons in transmission media and electronic devices. It is a function of temperature [11]. Thermal noise (thn), as one of the main sources of ambient noise, is modeled according to the following equation in the $dB\ re\ \mu Pa\ per\ Hz$ form [1]:

$$thn(f)_{dB\,re\,\mu\,Pa\,per\,Hz} = -15 + 20\,log f \tag{2.5}$$

Turbulence noise is another source of ambient noise. The following equation models turbulence noise (tn) in the $dB\ re\ \mu Pa\ per\ Hz$ form [1]:

$$tn(f)_{dB\,re\,\mu\,Pa\,per\,Hz} = 17 - 30\,log f \tag{2.6}$$

Finally, shipping noise (sn) in the $dB\ re\ \mu Pa\ per\ Hz$ form is represented by the following equation [1]:

$$sn(f)_{dB\,re\,\mu\,Pa\,per\,Hz} = 40 + 20(s - 0.5) + 26log f - 60log(f + 0.03) \tag{2.7}$$

s is the level of shipping activity in the range from zero to one. In summary, ambient noise N is modeled in this equation as a sum of its four main sources [1]:

$$N(f)_{dB\,re\,\mu\,Pa\,per\,Hz} = 10\,log[10^{tn(f)/10} + 10^{sn(f)/10} + 10^{wn(f)/10} + 10^{thn(f)/10}] \tag{2.8}$$

Also, ambient noise N is modeled in the following equation [1]:

$$N(f)_{dB\,re\,\mu\,Pa\,per\,Hz} = 50 - 18\,log f \tag{2.9}$$

2.2.2 Nyquist's Formula

The following equation represents the Nyquist's formula [11]:

$$C = 2B\,log_2 M \quad \text{Bits/sec} \tag{2.10}$$

It represents the relationship between the channel capacity C (in bits per second (*Bits/sec*)), bandwidth B (in *Hertz*), and number of discrete signal elements M. This formula considers that the channel is noise-free. It shows that doubling the bandwidth results to doubling the data rate.

2.2.3 Shanon's Formula

The Shanon's formula considers noise. It represents the relationship between signal power, noise power, bandwidth, and data rate [11]:

$$C = B\,log_2(1 + SNR) \quad \text{Bits/sec} \tag{2.11}$$

C is the channel capacity (in *Bits/sec*). B refers to the bandwidth in *Hertz*. *SNR* (Signal-to-Noise-Ratio) is the ratio of the received signal power (S) to the noise power (N), in Watts [11]:

$$SNR = \frac{S}{N} \tag{2.12}$$

Higher signal quality corresponds to higher *SNR*. The channel capacity increases by hiking the signal power, increasing the bandwidth, or decreasing the noise power. The Shanon's formula shows the maximum capacity that can be attained theoretically since it only considers thermal noise. It does not consider impulse noise, delay distortion, and attenuation distortion [11].

2.2.4 Signal Energy per Bit over Noise Power Spectral Density (E_b/N_0)

The E_b/N_0 ratio (aka *SNR per bit*) is the signal energy per bit over noise power spectral density [12, 11]:

$$\frac{E_b}{N_0} = \frac{\frac{S}{R}}{N_0} = \frac{S}{kTR} \tag{2.13}$$

This metric is used as a standard quality scale to specify the performance of digital system communication. N_0 represents the noise power density in Watts per one Hertz of bandwidth. S represents the signal power. k is the Boltzmann's constant which equals 1.38×10^{-23} Joules per Kelvin. T represents the temperature in Kelvins. R is the data rate in Bits per second. The following equation represents the E_b/N_0 ratio in decibel form [11]:

$$\left(\frac{E_b}{N_0}\right)_{dB} = S_{dBW} \; - \; 10logR \; + \; 228.6\,dBW \; - \; 10logT \tag{2.14}$$

The bit error rate of a digital data signal is a decreasing function of the E_b/N_0 ratio. In other words, by increasing the E_b/N_0 ratio, the bit error rate reduces. The relationship between the quantities E_b/N_0 and SNR is shown in the following equation [11]:

$$\frac{E_b}{N_0} = \frac{S}{N_0 R} \tag{2.15}$$

Let B_T denote the bandwidth, the noise in a signal is:

$$N = N_0 B_T \tag{2.16}$$

Using the last two equations, the signal energy per bit over noise power spectral density ratio (E_b/N_0) is [11]:

$$\frac{E_b}{N_0} = \frac{SB_T}{NR} \tag{2.17}$$

2.2.5 Attenuation

Acoustic waves are converted into heat during their propagation in water. This conversion is a cause of attenuation. The scale of the conversion is related to the signal frequency. Geometrical spreading is another cause of attenuation. Attenuation is dependent on the distance. As acoustic waves travel away from their source, they cover larger areas. Therefore, their acoustic energy intensity decreases. Attenuation and its relationship with the mentioned parameters are represented in the Thorp model [1]:

$$A(d, f)_{dB} = k \cdot 10 \cdot log\, d \; + \; d \cdot a(f) \tag{2.18}$$

The Thorp model is used for the calculation of underwater attenuation. Attenuation (A) is calculated as a function of distance and frequency. f indicates the frequency in kiloHertz (kHz). The distance (d) in kilometer is calculated based on the position of sender and receiver. k indicates the geometrical spreading coefficient which can have the value 1 for cylindrical, 1.5 for practical, and 2 for spherical. a represents the absorption coefficient. $a(f)$ is a frequency-dependent absorption coefficient(in dB/km) [13, 14]:

$$a(f)_{dB/km} \; = \; \frac{0.11 \cdot f^2}{1 + f^2} \; + \; \frac{44 \cdot f^2}{4100 + f^2} \; + \; 2.75 \cdot 10^{-4} \cdot f^2 \; + \; 0.003 \tag{2.19}$$

The acoustic channel contains two main loss mechanisms: viscous absorption for high frequency acoustic signals and molecular absorption at low frequencies. Attenuation in both is dependent on the aquatic conditions such as the type of dissolved compounds in water as well as pressure, temperature, and water salinity [15].

Acoustic signals can propagate long distances at low frequencies, owing the fact that acoustic attenuation in sea water is lower at low frequencies. It may result in multi-path propagation which is a propagation of acoustic signals from the source in several ways due to refraction in water and reflection at sea surface or seabed [15].

2.2.6 Signal-To-Noise-Ratio

Signal-to-Noise-Ratio (SNR) for an underwater acoustic signal in UWASNs is modeled in the following equation [1]:

$$SNR_{dB} = P - [A(d, f) + N(f)] \tag{2.20}$$

Based on this model, the attenuation according to the Thorp model (see Equation 2.18) and total noise density (see Equation 2.9) are subtracted from the transmission power P, which is in the $dB\,re\,\mu Pa\,per\,Hz$ form. Since noise is frequency dependent, it is considered to be a function of f. Attenuation $A(d,\ f)$ is a function of frequency f and d. Symbol d represents the distance from the source in kilometer. In other words, attenuation is dependent on the frequency and propagation distance.

Chapter 3

Related Work

3.1 Introduction

The characteristics of UnderWater Acoustic Sensor Networks (UWASNs) make them remarkably different from other networks such as terrestrial networks. Packet error, packet loss, limited bandwidth, and high variable propagation delay are common shortcomings of UWASNs. This kind of network is mainly characterized by its sensor mobility. Most of the existing mobility models are not adapted to the conditions of the underwater environment. This chapter reviews the related works from three different aspects: underwater communications, underwater sensor networks, and underwater mobility models.

3.2 Underwater Communications

Kumar et al. [16] studied the advancement trend of wireless communications from the farthermost points from the earth, in space, to the deepest points, underwater. They listed the main challenges in underwater communications such as high ambient noise, transmission loss, and propagation delay. They also reviewed recent improvements in underwater acoustic communications.

Sendra et al. [17] reviewed recent advancements in underwater acoustic modems and their specifications. Existing underwater systems, in commerce and industry, have been compared to the ones used in academia. The results demonstrated that commercial systems have more energy consumption and high ability to reach further distances.

Singer et al. [18] reviewed recent progressions and improvements in underwater acoustic communications. They also surveyed different signal processing techniques. The role of several key factors on the performance enhancements of signal processing systems have been studied, includes *single carrier*, *iterative decoding*, *spatial multiplexing methods*, and *emerging multi-carrier*.

Bui and Yeoh [19] examined the propagation of electromagnetic waves in a marine environment. They proposed a model for underwater transmissions via electromagnetic waves. The horizontal sea surface propagation of electromagnetic waves has been compared to the vertical propagation. The results illustrate that horizontal wave propagation is more efficient in terms of traveling farther distances and having less attenuation. In contrast, the vertical wave propagation faces high attenuation. Moreover, higher performance of transmission in terms of data rate was observed for frequencies below *five* megaHertz. Finally, the efficiency of a magnetic antenna and an electric antenna has been compared. The former outperforms the latter in an aquatic environment.

Collecting data related to the location and time is a challenge in UnderWater Sensor Networks (UWSNs). The Global Positioning System (GPS) does not work underwater. To address this issue, Taraldsen et al. [9] examined *Dilution Of Precision (DOP)* in both GPS and *Underwater Positioning System (UPS)*. *DOP* refers to the precision and validity of received location data. *UPS* is used in different underwater systems and aquatic vehicles. Several related statistical models and estimators have been represented and compared in their work. An acoustic underwater positioning system has been developed by Ren et al. [20]. It is mainly based on *Field-Programmable Gate Array (FPGA)* technology. Their system results in acquiring the location-based data of mobile destinations using acoustic signals.

Due to the limitations of underwater acoustic communications such as restricted bandwidth and double selectivity (frequency-selectivity and time-selectivity), Qu and

Yang [21] presented and evaluated four schemes, characterized by data rate and reliability. The first scheme is called *Basis Expansion Model (BEM) single-carrier*. It has high performance and data rate. The second one is *differential Orthogonal Space-Time Block Coding (OSTBC) on plain Orthogonal Frequency Division Multiplexing (OFDM)* scheme. It has a high data rate and low reliability. This scheme has less complexity in comparison to BEM. The third scheme, *BEM-based differential OSTBC*, shows poor results in terms of data rate. Last but not least, the *coherent Direct-Sequence Spread Spectrum (DSSS)* scheme has low complexity, low data rate, and high reliability.

Hydroacoustic communications have been simulated by Borowski [13]. He has simulated a time-variant channel and a physical layer model. The model consists of a modulator, a channel simulator, and a demodulator. Either binary Phase-Shift Keyed (PSK) or Frequency-Shift Keyed (FSK) waveforms can be generated by the modulator in his work. PSK is a digital modulation scheme that changes the phase of the carrier wave. In contrast, FSK changes the frequency of the carrier wave. Then, the received modulated signals are demodulated by the PSK or FSK demodulator. He has also developed an acoustic binary modem for the Linux environment. It runs network applications that communicate acoustically. This modem is highly adaptable to the different environments. In our work, the PSK modulator and demodulator are borrowed from this work (see Sections B.2.5.4 and B.2.5.6). In addition, the model provides an option to set several parameters, such as baud rate, carrier frequency, and sampling frequency (see Section B.2.5.3). Developed communication system is highly adaptive to any aquatic environments. At the end, the developed adaptive model shows better results in comparison to non-adaptive ones in terms of efficiency.

Burdinskiy et al. [22] have proposed an algorithm for underwater acoustic systems. Their algorithm processes PSK pseudo noise signals. It also considers the Doppler effect which influences underwater acoustic communications. The Doppler effect refers to the changes in wavelength and frequency of waves due to relative displacement between a source and a destination. Their algorithm has high efficiency

and low computational complexity.

3.3 Underwater Sensor Networks

Bhambri and Swaroop [23] surveyed the main applications and challenges of UWSNs. Two-dimensional and three-dimensional architectures of UWSNs have been studied. They also investigated UWSN features such as ease of deployment, autonomy multi-hop, dynamic topologies, data gathering, and transmission signals. UWSNs have been compared to terrestrial networks from different aspects. The UWSN challenges have been listed as: cost, limited storage capacity, low battery life, attenuation, corrosion of equipment, and no real time monitoring.

Heidemann et al. [5] discussed main difficulties and progressions in design, analysis, and implementation of UWSNs. They probed the impacts of the acoustic technology on this kind of network. Different underwater vehicle technologies have been reviewed such as *Remotely Operated Vehicles (ROVs)*, *Autonomous Underwater Vehicles (AUVs)*, and *ROV/AUV tandems*. They also reviewed the related networking technologies and applications. Finally, they discussed different testbeds, hardware platforms, simulators, and models.

Peach et al. [24] also reviewed UWSNs. They explored the role of UWSNs in different applications such as military, oceanography, geological, and environmental monitoring. The two-dimensional and three-dimensional architectures of UWSNs have been overviewed in details. In addition, they discussed *AUVs*, *Unmanned Underwater Vehicles (UUVs)*, and some of their test deployments. Finally, concepts related to modulation, propagation, and Medium Access Control (MAC) in UWSNs have been discussed.

Sendra et al. [17] probed different software and network simulators related to UWSNs. NS2 (Network Simulator 2) showed better results in comparison with other simulators in terms of accuracy and resemblance of results to the real experiments.

The main challenges in developing marine applications in scalable mobile UWSNs have been introduced by Cui et al. [25]. They investigated the mobile UWSNs for two network architectures of marine applications: *long-term non-time-critical aquatic monitoring* and *short-term time-critical aquatic exploration*. They also reviewed the challenges in the design and implementation of UWSNs, organized from the topmost layer of the protocol stack to the lowest layer.

Designing and improving efficient routing protocols is one of the main challenges in UWSNs due to their particular characteristics. Souiki et al. [26] reviewed UWSN routing protocols. They mainly focused on geographic routing protocols rather than non-geographic ones. They surveyed them according to three criteria: *greedy*, *restricted directional flooding*, and *hierarchical*. They evaluated the performance of these protocols from different aspects such as design goals and forwarding strategies. Since nodes have displacement underwater, finding the position of a source and a destination is a major challenge for routing protocols. They studied *Sector-Based Routing with Destination Location Prediction* (*SBR-DLP*). It is a restricted directional flooding routing protocol. There is no need for the localization of the destination. It uses a pre-arranged scenario of movement. However, it does not consider the displacement of destination due to the aquatic environment. It raises a need for having a corresponding mobility model to solve the problems.

3.4 Underwater Mobility Models

UWASNs are characterized by several properties such as having non-permanent links, dynamic and mobile nodes. Nodes have changeable positions that are influenced by the aquatic environment. As a consequence, having a mobility model, adapted to underwater conditions, is essential for UWSNs.

A brief survey on the mobile and float sensors and their differences in UWSNs has been carried out by Cui et al. [25]. A mobile UWSN has been compared to a

small-scale UWSN with fixed sensors in terms of scalability, self-organization, and localization.

Detweiller et al. [27] developed an UWSN. It communicates either optically for point-to-point data transmission or acoustically, for broadcast communications. Beside that, they developed deployable hardware. It is suitable for shallow waters using an acrylic enclosure. It can be used in deep waters, at depths more than 100 meters using a titanium or a glass enclosure. The platform includes static sensors and mobile robots, called *AMOURs (Autonomous Modular Optical Underwater Robots)*. An *AMOUR* relocates the sensors and collects their information optically. In order to find and follow the sensors and *AMOURs*, a *passive localization and tracking algorithm* has been developed. The UWSN hardware also include acoustic and optical modems that have been made specifically for this work. Sensors communicate with a *self-synchronizing Time Division Multiple Access* (*TDMA*) protocol. It enhances the UWSN by providing an option for dynamic sampling.

An underwater vehicle simulator, called *PLATYPUS*, has been developed for coastal exploration by Suzuki at al. [28]. The vehicle comprises of mechanical flapping pectoral fins. The motion of the vehicle with *simple fin motion* has been simulated using a *Computational Fluid Dynamics (CFD) Motion* simulator. The hydrodynamic forces are produced by flapping fins. For simulating the motion, these forces have been determined using statistical and mathematical methods. They enhanced the simulator, using experimental data. Moreover, the motion control has been simulated using a fuzzy control algorithm.

An underwater glider, called *ALEX*, has been developed by Arima et al. [29]. It consists of an on-board microcomputer system and tractable wings, which increase the transportability and adjustability of *ALEX*. The hydrodynamic forces on *ALEX* are computed using a *CFD* method. The performance of *ALEX* has been compared to other aquatic gliders with immovable wings. The results demonstrate the superior performance of *ALEX* with respect to motion control and motion capacity.

To find a solution for the localization problems of large-scale UWSNs, Zhou et al. [30] introduced *Scalable Localization Scheme with Mobility Prediction (SLMP)*. The localization process has been carried out hierarchically for both typical and anchor sensors. A typical sensor moves following a mobility pattern, complemented with prior position data. An anchor sensor, as a controller of localization, has a specified location. The results indicate that the performance has been improved in terms of communication cost, localization coverage and accuracy.

Bower [31] worked on determined patterns of fluid parcel trajectories. A fluid parcel refers to a little amount of fluid that proceeds with a fluid flow [32]. A *two-dimensional kinematic model of a meandering jet* has been elaborated based on the acquired patterns [31]. This model represents the fluid parcel routes beside the streamfunction patterns. The streamfunction in fluid dynamics, refers to two-dimensional divergence-less flows to shows the pathway of particles in the flow [33]. Another study on this model based on findings of Gulf Stream research has been done by Samelson [34]. He probed the trajectory of fluid parcels of a meandering jet according to the Melnikov's technique. He examined the effects of high and low frequencies of meander amplitudes on the performance of cross-jet trajectories.

Caruso et al. [6] proposed a mobility model, called *Meandering Current Mobility (MCM)*. This model is based on the underwater environments, such as ocean currents. It addresses several issues of UWASNs such as networking protocols and localization services. The model consists of autonomous sensors with networking capability. These sensors communicate acoustically. The mobility of sensors expands the dynamic coverage. Each sensor moves according to ocean meandering currents. The trajectory of underwater sensors interchange between the jet downstream motion and the rotating vortices.

There are different ways for modeling of ocean currents such as *ocean forecasts* and *Lagrangian transport*. The former costs a lot because it requires continuous and

detailed observation of different areas. Lagrangian transport has been used for ocean currents modeling in the MCM model. It only considers normal conditions of an ocean environment, while ignoring unusual water circulation and vertical movements.

The divergence-free velocity field $\mathbf{u}$ ($\mathbf{u} \equiv (u, v)$) consists of two components: u and v. These components are calculated according to the following equation [6]:

$$u = -\frac{\partial \psi}{\partial y}, \qquad v = \frac{\partial \psi}{\partial x} \tag{3.1}$$

u is an eastward component and v is a northward component of the velocity field. ψ represents a streamfunction. A streamfunction defines divergence-less flows in two dimensions (x and y). It is commonly used in the field of fluid dynamics to model a medium. The components of the velocity vector are then found by [6]:

$$\dot{x} = -\partial_y \psi(x,\, y,\, t), \qquad \dot{y} = \partial_x \psi(x,\, y,\, t) \tag{3.2}$$

These equations are the vector form of Hamilton's equations. The following equation represents a non-dimensional form of the meandering jet model [6]:

$$\psi(x,\, y,\, t) = -\tanh\left[\frac{y - B(t)\sin(k(x - ct))}{\sqrt{1 + k^2 B^2(t) cos^2(k(x - ct))}}\right] \tag{3.3}$$

c represents the phase speed. The number of meanders in the unit length is shown by k. The width of the meanders is modulated by B as below [6]:

$$B(t) = A + \epsilon \cos(\omega t) \tag{3.4}$$

The function B is dependent on time t. A is the average meander width. The amplitude of the modulation is shown by ϵ. Its frequency is shown by ω. This streamfunction shows an interchange between the jet downstream motion and rotating vortices.

It has been shown in the MCM model results that an appropriate waiting time in multi-round simulations increases the connectivity and lifetime of networks.

3.5 Summary

Caruso et al. [6] proposed the MCM model for UWASNs. They studied the effects of the MCM model on localization errors, network connectivity, and coverage. In our work, OMNeT++ is used to simulate mobility in UWSNs using the MCM model. In addition, underwater acoustic communications are simulated using MATLAB modems by Borowski [13].

Chapter 4

Underwater Mobility Simulation

This chapter provides information regarding the simulation of physical, link, and network protocol layers. The network and link protocol layers are simulated in OMNeT++. In addition, the Meandering Current Mobility (MCM) model is simulated in OMNeT++. MATLAB functions are integrated into OMNeT++ as a physical layer. After the integration of all three protocol layers, several parameters are calculated and analyzed such as Bit Error Rate (BER) and distance.

4.1 Simulation Model Architecture

The network simulation consists of three protocol layers of UWASNs: network, link, and physical. Two software tools are used in this work: OMNeT++ and MATLAB. Appendix A reviews the simulation tools and software requirements. The physical layer is simulated using MATLAB functions. The link and network layers are simulated using OMNeT++ (see Figure 4.1) [14].

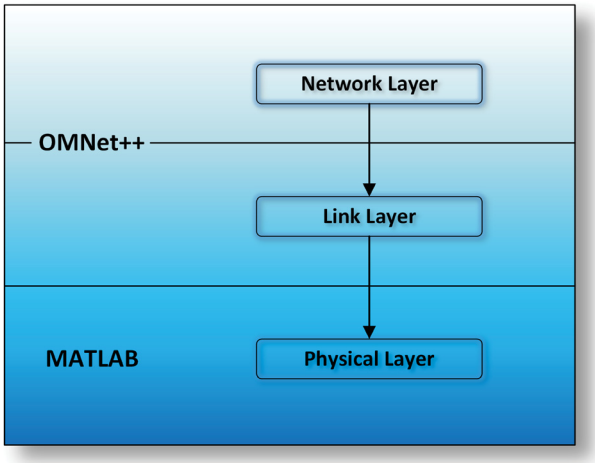

Figure 4.1: Integration of three protocol layers.

The simulation model architecture is shown in Figure 4.2. It consists of three protocol layers. The physical layer simulated using MATLAB, contains models of a modulator, a channel, and a demodulator of hydroacoustic digital data signals. In addition, the effects of noise and attenuation on the underwater communications are considered in the physical layer model. The link layer, simulated using OMNeT++, includes coding and decoding of frames. The network layer, also simulated using OMNeT++, comprises transmission and reception of packets.

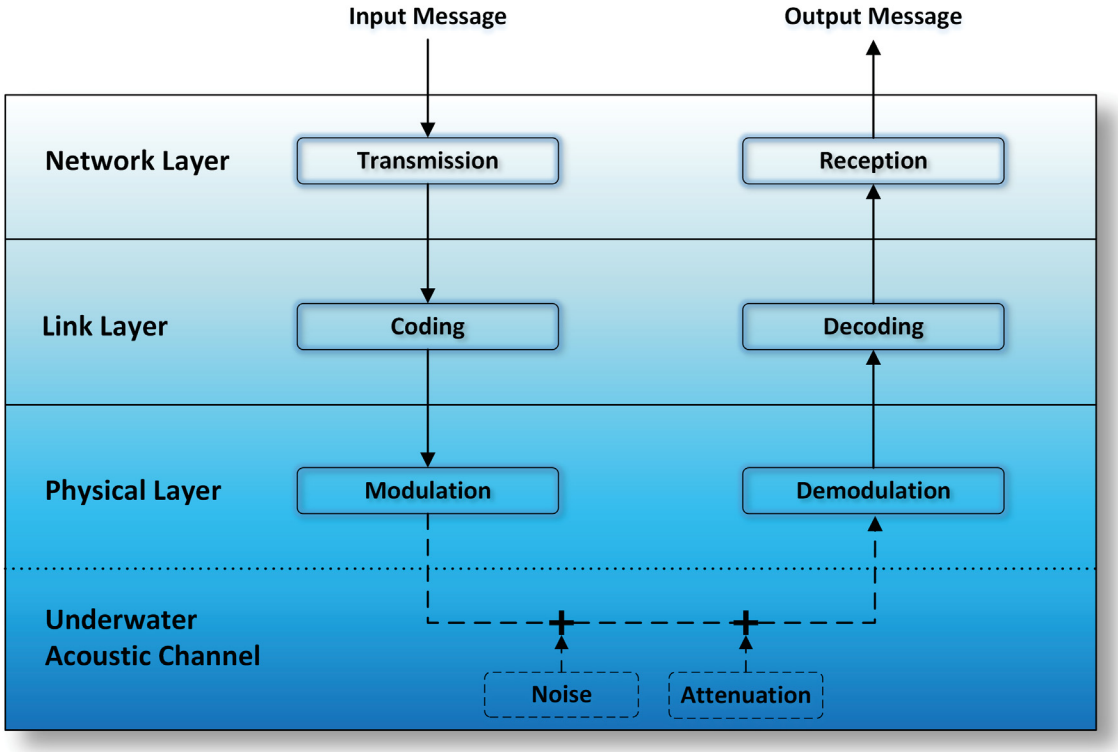

Figure 4.2: Simulation model architecture.

4.2 Simulation Steps

Two software tools are used in this work: OMNeT++ and MATLAB. The simulation in OMNeT++ consists of two general procedures. Firstly, the simulation setup, which defines parameters and metrics. Secondly, the implementation, which specifies

the module behaviors, such as event generation, and message formats [35].

To begin with, we need to define four core components of OMNeT++: simple modules, compound modules, gates, and channels. These components determine the structure of the simulation model. The algorithmic part of the simulation model is elaborated in the simple and compound modules. The connections between modules are then determined in gates and channels (see Section B.2.1). Next, messages and their format are declared to add events to the simulation model (see Section B.2.2). Several parameters and network configurations are provided according to simulation requirements (see Section B.2.3). Considering the existing options for mobility modeling, the MCM model is simulated in OMNeT++ (see Section B.2.4). After that, the simulated model in OMNeT++ is integrated with MATLAB (see Section B.2.5). Statistics are collected during and at the end of the simulation to analyze the results (see Section B.3). The final steps are building and running of the simulation project (see Section B.4).

4.3 The Integration Process of OMNeT++ and MATLAB

The physical layer is simulated using the MATLAB functions. They are compiled into a dynamic shared library (see Section B.2.5.1). Then, the shared library is loaded in the OMNeT++ environment to integrate the three protocol layers. To load the MATLAB shared library in OMNeT++, the initialization functions are called (see Section B.2.5.2). Next, the required variables and parameters are defined (see Section B.2.5.3). The packet transmission process has three major steps: the modulation, channel simulation, and demodulation of digital data signals (see Sections B.2.5.4 to B.2.5.6). Several metrics are calculated according to the simulation output (see Section B.2.5.7). Finally, the occupied memory is deallocated (see Section B.2.5.8).

Chapter 5

Evaluation

In this chapter, the performance of communicating sensors in UnderWater Acoustic Sensor Networks (UWASNs) moving according to the Meandering Current Mobility (MCM) model is evaluated.

5.1 Simulation Environment

Mobile underwater sensors and vehicles are categorized as drifters and floats. A drifter is either floating on the water surface, or hanged from a buoyant object. A float on the other hand, is a sensor that is operating much deeper. Ocean water density is a function of temperature, salinity and depth. Therefore, the density of floats can be adjusted to operate along an isopycnal surface (a surface with constant density). Isopycnal surfaces in the oceans are usually almost horizontal [13].

In certain circumstances, vertical movement do exist in oceans. For instance, due to wind driven waves, or due to thermohaline circulation caused by density gradients and freshwater fluxes, or rarely due to body waves caused by massive earthquakes. The drifters experience more vertical movement compared to the floats. In this work, vertical movements are assumed to be negligible for underwater free-floats, ignoring unusual water circulation and circumstances.

Mobility of UWASNs can be modeled using random walks or any other random methods. However, this is not very realistic. Underwater sensors tend to be carried away along with water velocity fields. In other words, underwater sensors are advected by the same streamfunction due to the fluid nature of the medium. In this work, mobility of underwater floats is simulated by assuming the water as a layered

continuum. This enables us to apply kinematic models of two dimensional flows, when neglecting the vertical movements according to the MCM model [13].

5.2 Simulation Application

The British Petroleum (BP) oil spill in the Gulf of Mexico happened in 2010 [36]. It was one of the largest disasters of marine oil spill. More than four million barrels of oil flowed in water for 87 days. Several attempts had been made to decrease the pollution using various skimmer vehicles, controlled burns, and floating booms. However, the extend and speed of oil spreading on water were uncontrollable and unpredictable. Oil covered almost 180,000 square kilometers of water surface. As a consequence, the catastrophic effects on the environment and animal lives were significant.

Having well-equipped and mobile sensors with networking capability seems to be helpful and essential in such condition, where human activities are hard to conduct. UWASNs are enabling technology in ecology observations for monitoring and measuring the water quality and pollutant concentration. The simulated mobility of UWASNs in this work could have been employed to a situation such as the oil spill in the Gulf of Mexico.

In the context of this thesis, let us assume an aquatic environment, polluted by oil spreading on the water surface. The main purpose is monitoring the effects of the oil spill on wild life beneath the water. A group of 100 free-float sensors are released over a limited underwater area near the water surface. These sensors are equipped to measure light and carbon dioxide of water in the located depth. The extent of pollution is measured, comparing the obtained results to the related data in normal conditions. These sensors move according to the ocean currents and vortices. The pattern of their movements is predicted by the MCM model. During their movement, they perform a collaborative task to measure the water pollution and the scope of the polluted area. They also monitor the progression of oil spreading in the sea.

5.3 Network Configuration

An UWASN is modeled as a time-dependent graph: G=($S(t)$, $L(t)$). $S(t)$ represents a set of sensors at time t [6]. $L(t)$ indicates the links between sensors at time t: $L(t) \subseteq S(t) \times S(t)$. Data is collected when any pair of nodes, a sender and a receiver, are communicating with each other via messages.

Initially, 100 sensors are distributed, randomly and uniformly, over a limited area. After deployment according to the MCM model, they distribute on a wider extent. Figure 5.1 shows the evolution of 100 sensors in an OMNeT++ simulation. It has been recorded at five different simulation time steps, T_1 to T_5. The x and y axes show relative distance in meters.

For our evaluation, we consider the communications between two specific nodes, a sender and a receiver. These nodes move according to the MCM model. They communicate with each other via messages. The messages have randomly generated data. The messages are transmitted to the receiver. They pass through the physical layer model, that consists of a PSK modulator, a channel, and a PSK demodulator of digital data signals. Noise and attenuation are considered in the channel. At the end, the received message is obtained at the output. Then, the Bit Error Rate (BER) is calculated, comparing input data and output data. The sender-receiver separation distance is calculated at the time of data transmission. In addition, the E_b/N_0 ratio is calculated according to Equation 5.3. Finally, the obtained results are analyzed.

5.4 Metrics

This section reviews collecting distances and two metrics used in this work: BER and signal energy per bit over noise power spectral density (E_b/N_0).

5.4.1 Collecting Distances

After releasing the sensors, they start to move according to the mobility model over a wider area. Their positions and movements are recorded during the simulation.

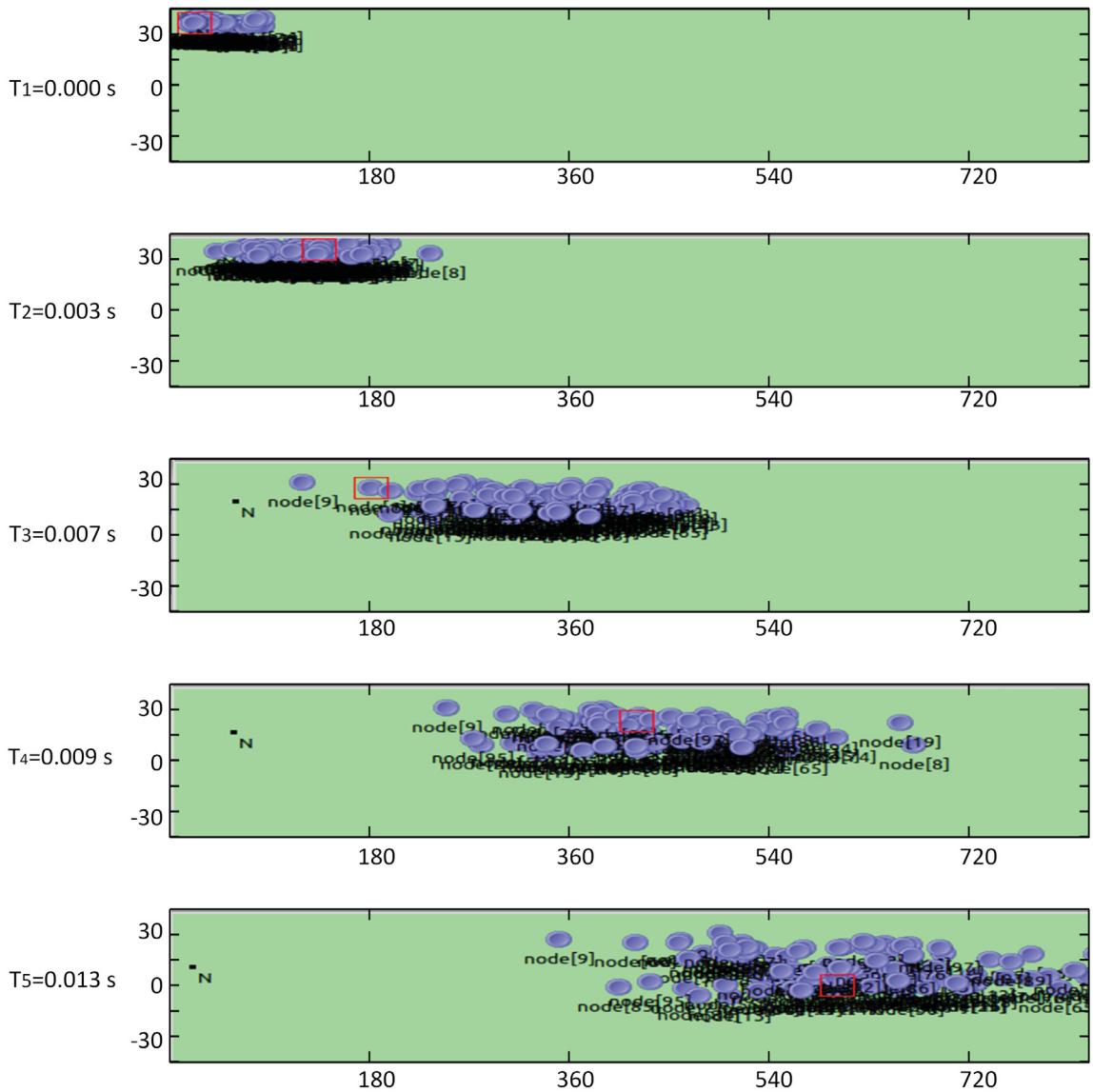

Figure 5.1: The position of sensors at different time steps: T_1, T_2, T_3, T_4 and T_5.

Distances are calculated while messages are passed between sensors. Let us consider two nodes, a sender and a receiver with the positions (X_0, Y_0) and (X_1, Y_1). The distance between them is calculated using the following equations:

$$dx = X_0 - X_1; \tag{5.1}$$

$$dy = Y_0 - Y_1;$$

$$Distance = \sqrt{dx^2 + dy^2};$$

5.4.2 Bit Error Rate

The BER is calculated by dividing the number of bit errors by the total number of transmitted bits during the communication.

$$BER = \frac{Number\,of\,Errors}{Total\,Number\,of\,Sent\,Bits} \tag{5.2}$$

The calculation of BER during simulation is discussed in details in Appendix B (see Section B.2.5.7).

5.4.3 Signal Energy per Bit over Noise Power Spectral Density (E_b/N_0)

The E_b/N_0 ratio (aka *SNR per bit*) is the energy per bit over noise power spectral density. It is discussed in detail in Section 2.2.4. In this work, the E_b/N_0 ratio (unit less) is calculated using the following equation [11, 12]:

$$\frac{E_b}{N_0} = \frac{SB_T}{NR} \tag{5.3}$$

S represents the signal acoustic pressure (in μPa). B_T is the channel bandwidth (in *Hertz*). The noise acoustic pressure is shown by N (in μPa). R is the bit rate (in *Bits per second*). The E_b/N_0 ratio is important because the BER for digital data signals is a decreasing function of this ratio. Moreover, it is widely used normalized performance metric.

5.5 Curve Fitting

MATLAB has several polynomial and linear curve fitting tools for regression, such as *Trendline*. The nonlinear curve fitting capabilities, such as logarithmic and exponential, have been optimized in the *curve fitting* toolbox. This toolbox is used to fit curves to the existing datasets [37]. It is available under the name *cftool*.

There are different curve fitting models, including *Polynomial*, *Gaussian*, *Exponential*, *Fourier*, *Weibull Distribution*, *Power Series*, and custom models. In this work, the following general *Gaussian* model is used [38, 37]:

$$f(x) = a_1 \cdot e^{-((x-b_1)/c_1)^2} + a_2 \cdot e^{-((x-b_2)/c_2)^2} \tag{5.4}$$

The terms a_1, b_1, c_1, a_2, b_2, and c_2 are coefficients. They are generated automatically.

Confidence Intervals and Prediction Bounds: Confidence intervals are calculated for the fitted coefficients, using the *curve fitting* toolbox in MATLAB [39]. The prediction bounds are represented either graphically or numerically. The default value for the confidence level is 95%. After creating a fitted curve according to the *Gaussian* function, prediction bounds are estimated for corresponding confidence intervals.

5.6 BER as a Function of Distance

Figure 5.2 plots the BER as a function of distance, obtained with our simulations. In this work, attenuation is modeled using the Thorp model (see Section 2.2.5). In addition, colored noise is added according to the Equation 2.9. A PSK modulator and a demodulator are used, developed by Borrowski [13] (see Section B.2.5). The model does not consider the propagation delay. The bit rate is equal to 1000 *Bits per second*. The *x-axis* shows the distance in kilometer. The *y-axis* represents the BER. The points are observed data items. The solid line curve represents a fit according to the *Gaussian* statistic. The dotted lines show the prediction bounds (the lower and

upper bounds) with 95% confidence. As is shown in the plot, the BER is low when the sensors are close to each other. The BER rises with an increase in the distance. When sensors are up to 75 km apart from each other, most of the observed data points are within the prediction bounds. However, as sensors get farther from each other, the dispersion increases. An increase in a distance, between a sender and a receiver, affects the data transmission by increasing the BER.

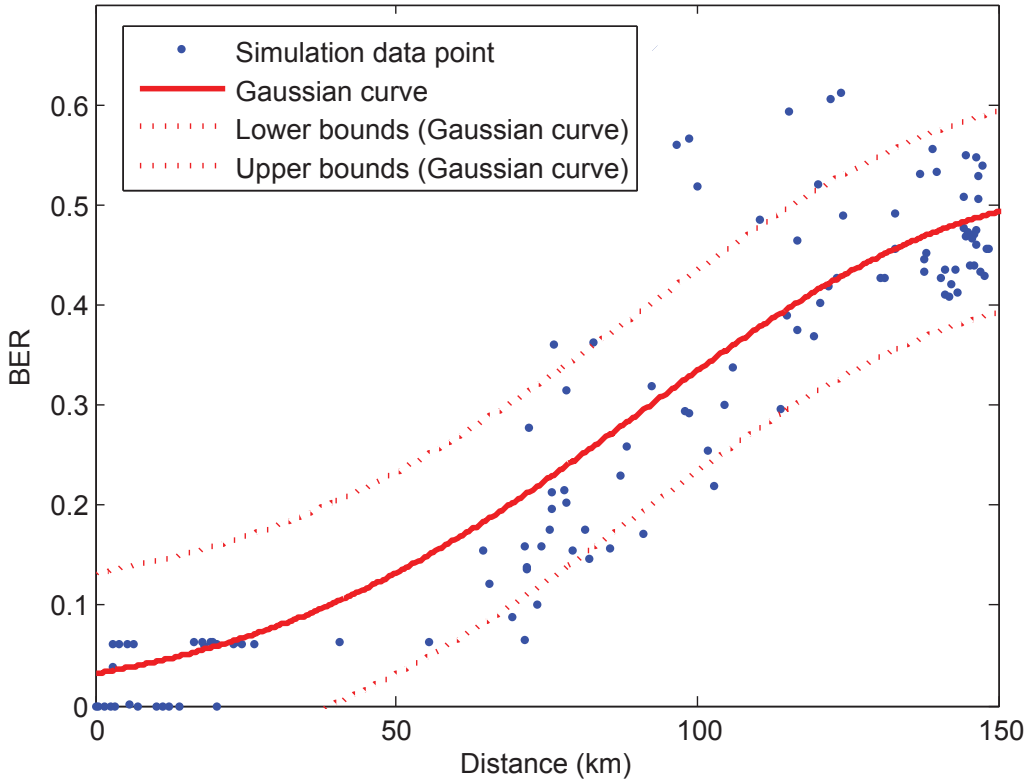

Figure 5.2: BER as a function of Distance.

5.7 BER as a Function of Signal Energy per Bit over Noise Power Spectral Density

Figure 5.3 plots the BER as a function of the E_b/N_0 ratio, obtained with our simulation. The simulation conditions are the same as Section 5.6. The E_b/N_0 ratio is calculated according to the Equation 5.3 (see Section 5.4.3).

The *x-axis* shows the E_b/N_0 ratio. The *y-axis* represents the BER. The points show observed data items. The dashed line curve corresponds to the theoretical BER curve. As the E_b/N_0 ratio increases, the BER drops. The theoretical BER is calculated according to the following equation [40, 41]:

$$P_e = \frac{1}{2} erfc\left(\sqrt{\frac{E_b}{N_0}}\right) \tag{5.5}$$

P_e is the probability of error based on the standard error probability function. $erfc$ is the complementary error function. It is calculated in MATLAB using function *berawgn* for *Additive White Gaussian Noise (AWGN)* channels [39].

The solid line curve shows a fit according to the *Gaussian* statistics. Two dotted lines determine the prediction bounds (the lower and upper bounds) with 95% confidence. The simulation results of the BER are higher than the theoretical BER curve. The environmental characteristics are not considered in the theoretical model. This model is developed for wireless networks. In these networks, communications are through electromagnetic waves that are subjected to white noise. In other words, the theoretical BER model only considers white noise. Whereas in UWASNs, communicational acoustic waves are subjected to colored noise. The theoretical model is included for reference purposes, i.e., the simulation should not outperform the theory. For the E_b/N_0 ratio at zero, the theoretical curve gives a BER of 14%. The *Gaussian* curve starts at 50% for the E_b/N_0 ratio of zero. As the E_b/N_0 ratio increases, the BER decreases. For large E_b/N_0 values, the *Gaussian* curve merges toward the theoretical curve.

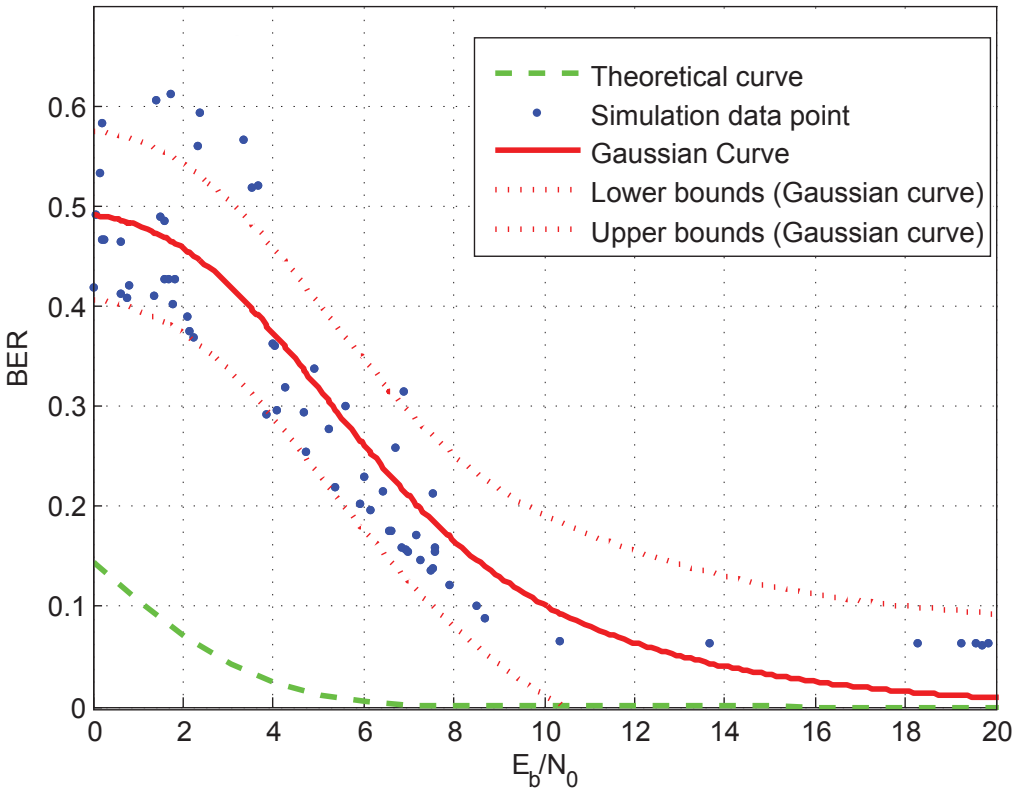

Figure 5.3: BER as a function of the E_b/N_0 ratio.

Chapter 6

Conclusion

Underwater communication is about sending and receiving messages in a sea environment. It has become an important data transmission technology, used in different areas such as defense, public safety, industry, and ecology. Since electromagnetic waves do not work underwater, using acoustics waves is the best solution for underwater communications. Hydroacoustics is the science of sound propagation in water. A hydroacoustic channel is characterized by a high Bit Error Rate (BER) at long distances and low BER at short distances. As a result, distance is relevant in underwater acoustic communications.

An UnderWater Acoustic Sensor Network (UWASN) consists of a variable number of self-organized underwater sensors that collect data and monitor the environment. These sensors move around, according to a marine environment. So, the displacements of sensors are influenced by water currents and other underwater activities, created by sea creatures and man-made vessels. The mobility of sensors is an important issue in UWASNs. It affects the network from different aspects, such as data transmission, connectivity, and coverage. As a consequence, simulation with a mobility model, adapted to underwater conditions, is essential for UWASNs. In this work, the mobility of sensors is simulated, according to the Meandering Current Mobillity (MCM) model [6].

The ultimate objective of this work is to simulate underwater acoustic communications with mobility in UWASNs. OMNeT++ and MATLAB are the main tools used in this work. Our model simulates three protocol layers of UWASNs: network, link, and physical. The link and network layer protocols are simulated using OMNeT++. The physical layer is simulated using MATLAB functions. It contains models of a

modulator, a channel, and a demodulator of underwater acoustic digital data signals. The physical layer model considers attenuation and noise. The MATLAB functions are compiled into a dynamic shared library. It is loaded in the OMNeT++ environment to integrate the three protocol layers.

The communications between sensors are influenced by their position and separation distance. In addition, the movements of sensors affect other parameters, such as the BER and attenuation. Any change in a transmitter-receiver separation distance affects attenuation. In other words, attenuation is sensitive to distance and mobility. Attenuation is modeled as a function of distance and frequency, using the *Thorp* model.

Two major metrics are calculated and used in this work: BER as a function of distance and BER as a function of signal energy per bit over noise power spectral density (E_b/N_0). The transmitter-receiver separation distance is calculated while they are moving according to the MCM model. The BER is calculated by dividing the number of bit errors by the total number of transmitted bits during the communication. Then, performance of communicating sensors moving according to the MCM model is evaluated. The BER as a function of distance is illustrated in a plot. The obtained results are demonstrated to fit a *Gaussian* statistics curve with 95% confidence bounds. An increase in a distance between a sender and a receiver, affects the data transmission by increasing the BER. In addition, the BER as a function of the E_b/N_0 ratio is represented in a plot. The obtained results are also demonstrated to fit a *Gaussian* statistics curve with 95% confidence bounds. The BER for digital data signals is a decreasing function of the E_b/N_0 ratio. A plot compares the obtained BER with the theoretical BER. In the simulation results, the BERs are higher than the theoretical BERs, that only consider white noise.

The work presented in this thesis has been partially published in a conference paper [14].

6.1 Future Works

In this work, mobile hydroacoustic communications between a sender and a receiver sensors are studied. We will consider multi-hop communications and routing protocols for future works. Moreover, the effect of propagation delay on underwater communications will be studied. Hydroacoustic waves suffer high propagation delay. The transmitter-receiver separation distance and propagation delay are relevant.

This work only considers ambient noise generated by four main sources: turbulence, water surface wave, thermal, and shipping noise. Site-specific noise will be considered in our future works. This kind of noise is geographically dependent, created by sea creatures or breaking icebergs. Due to the specific characteristics of the aquatic environments, the effects of site-specific noise need to be considered.

In future works, comparison of the simulation results with real world data will be done. At this stage, the simulation has a predictive role due to the lack of availability of experimental data. In addition, vertical movements and motion during transmission of packets will be considered in future works.

Appendix A

Simulation Tools and Software Requirements

This chapter reviews the simulation tools and software requirements for this work.

A.1 OMNeT++

OMNeT++ is a discrete event network simulation framework. It has been made available by *Andras Varga et al.* at *Technical University of Budapest* [35]. It is a component-based framework. OMNeT++ is used to model network protocols, queuing networks, distributed hardware systems, and wired and wireless networks. The components for network simulations are provided by other related frameworks, such as INET (see Section A.1.1) and MiXiM (see Section A.1.2). It can be installed on different operating systems, such as Windows, Mac OS, and Linux. It is free for non-profit use. OMNEST is a commercial version of OMNeT++. It is available for non-academic simulations. More details about OMNeT++ are provided in Appendix B.

A.1.1 INET

INET has been developed for OMNeT++. It is an open-source package to simulate communication networks [42]. Moreover, this framework implements the *User Datagram Protocol (UDP)*, *Transmission Control Protocol (TCP)*, and *IPv4/6*. It also supports link layer modeling, such as *802.11*, *Ethernet*, and *Point-to-Point Protocol (PPP)*. INET is a merger of mobility frameworks. It supports wireless communications and mobility.

A.1.2 MiXiM

MiXiM is a simulation framework to model mobile, fixed, and sensor networks. It uses the OMNeT++ simulation engine. It includes several frameworks: *Mac Simulator (in Delft University of Technology)*, *ChSim (in the University of Paderborn)*, *Positif (in Delft University of Technology)*, and *Mobility framework (by Technical University of Berlin and Telecommunication Networks Group)* [43]. Several other models have been integrated into MiXiM, such as *CSMA*, *IEEE 802.15.4*, and *Mobility Model for Body Area Networks (MoBAN)*.

There are two ways to create a MiXiM project in the OMNeT++ IDE. The first one is *Analogue Model Test Network*. It makes a simple network with an analogue model. The second one is *Basic MiXiM Network*. It creates a basic network with predefined parameters, such as *Network Interface Controller (NIC) protocols*, *application layers*, and *playground topology*. Both of these project types have their own settings and configurations. For example, the *playground* can be either two-dimensional or three-dimensional [43].

A.1.3 MiXiM-INET-Bundle

There are several problems and contradictions in using both the MiXiM and INET frameworks. Adding both of them to a project simultaneously returns error. Mixnet has been developed to solve these compatibility issues. It is a MiXiM-INET-bundle package. For example, both INET and MiXiM include wireless modules. To prevent potential errors, these modules have been omitted from INET code in Mixnet.

A.1.4 Existing Mobility Models in INET and MiXiM

To implement a new mobility model in OMNeT++, a mobility framework is needed. INET and MiXiM include several predefined mobility models. They are classified

from different aspects. One of these classifications is: scenario-based movement versus movement without any specified scenario. Another classification is based on mobility speed, which can be either stationary or constant. Figure A.1 represents the mobility models chart of INET. In this chart, they are classified into two main groups: models with mobility (*MovingMobilityBase*) versus models with fixed positions (*StationaryMobility*). Each one of them is structured into other models. More details are provided in Section B.2.4.1.

A desired mobility model should be selected from a list of existing mobility models. To use one of the existing mobility models in a project, *IMobility* should be imported. *IMobility* is a module interface for mobility models. It is an abstract base class which defines a public interface. It must be provided by all mobility modules. *MobilityBase* is an abstract base module for mobility models. Its inheritance relationship with *IMobility* is shown in Figure A.2 [42, 44].

A.2 MATLAB

MATrix LABoratory (MATLAB) is a programming language, with a focus on numerical computing, developed by MathWorks [39]. It integrates visualization and computation. It supports object-oriented programming and complex data structures. It contains built-in editing commands, math functions, and debugging tools. MATLAB facilitates mathematical calculations, performing numerical methods, and generating plots. It also supports implementing algorithms, models, and applications. MATLAB is used in a variety of areas, such as control systems, signal processing, and computational biology. It solves numerical problems in different engineering streams, such as math, computer, and physics.

On the grounds that MATLAB is a commercial product, *Octave* is a free alternative for most computing platforms. *Octave* is an open source and high-level interpreted language. Since it is similar to MATLAB, it runs most of MATLAB scripts and functions. Apart from *Octave*, there are other alternatives for MATLAB,

such as *Scilab*, *FreeMat*, and *Spyder*. In this work, MATLAB is used because of its shared libraries, signal processing, and curve fitting tools.

A.2.1 Using MATLAB Shared Libraries

To integrate MATLAB with OMNeT++, C++ code needs to be extracted from MATLAB functions. The code should be compiled as objects, which are used by OMNeT++ modules. It results a shared library. To call and use the MATLAB shared library, the *MATLAB Compiler Runtime (MCR)* toolbox is used. MCR bundles some libraries and components that help to deploy applications, built with MATLAB. It removes dependency to the whole MATLAB package. Finally, this shared library and all related files are compiled and built together in OMNeT++.

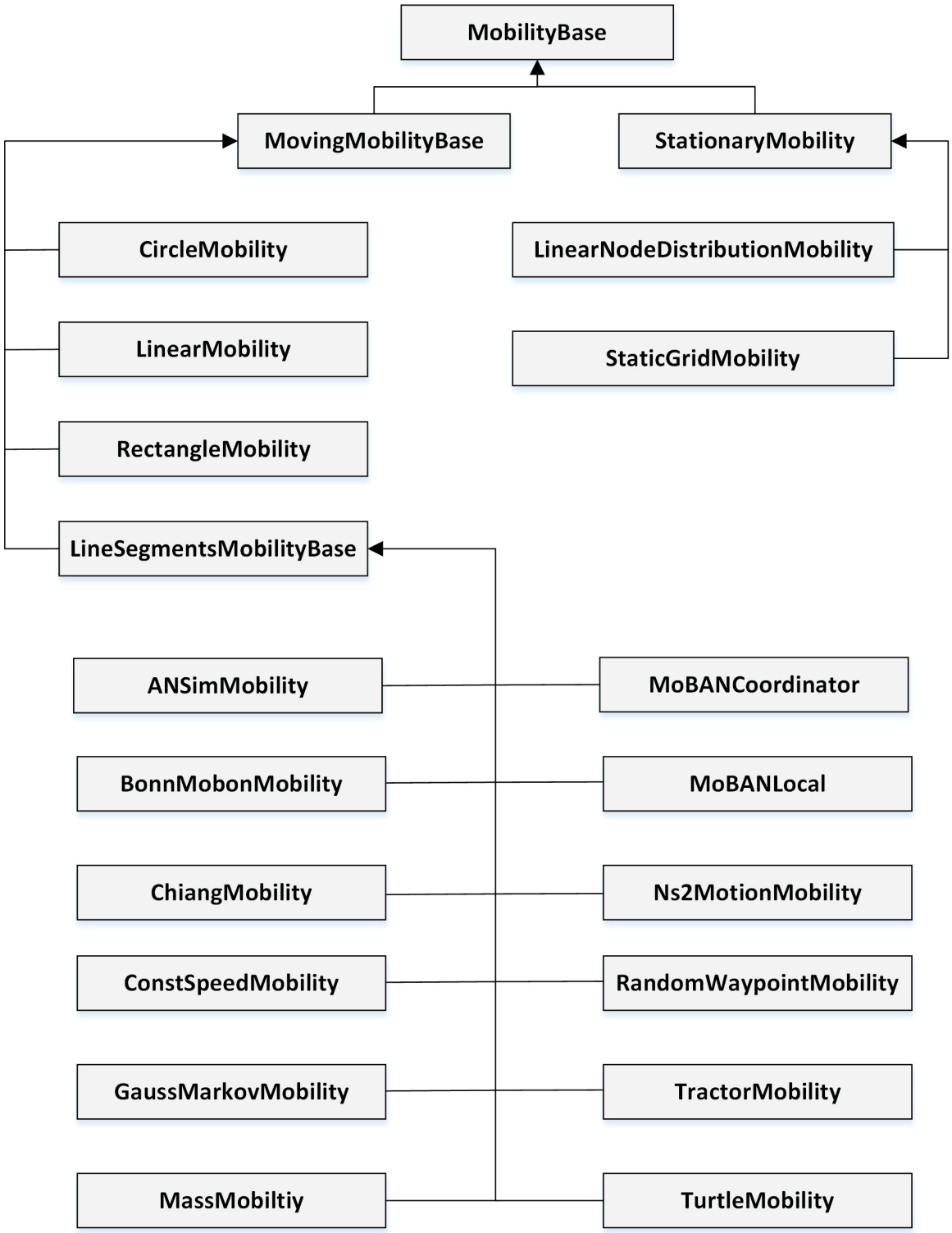

Figure A.1: Mobility Models chart in INET.

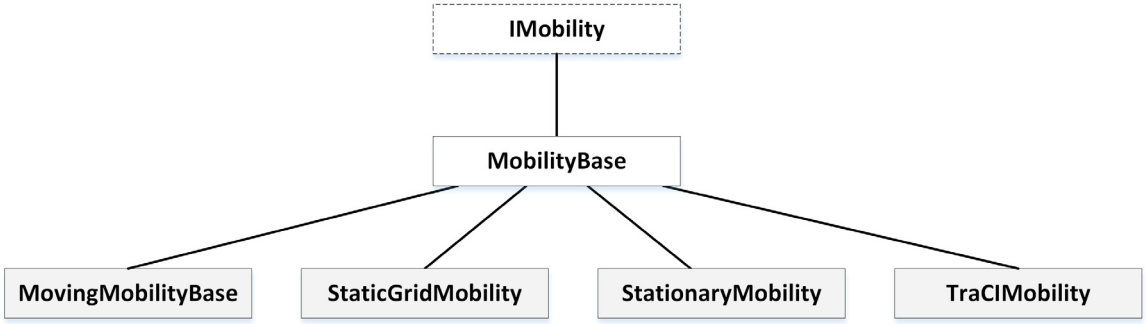

Figure A.2: The inheritance diagram of *MobilityBase*.

Appendix B

Simulation Implementation

In this work, the Meandering Current Mobility (MCM) model of Caruso et al. [6] is simulated in OMNeT++. This model is used to simulate mobility in UnderWater Acoustic Sensor Networks (UWASNs) (see Section 3.4, for details). In this model, each node plays a sensor role in an UWASN. The position of nodes, x and y coordinates, are calculated according to this model. The mobility and movement of nodes are updated according to the specified time steps.

To integrate the mobility model and physical layer model, MATLAB functions are compiled and called, as a dynamic shared library, in OMNeT++. By this means, the physical layer and two protocol layers above (the link and network layers), are combined together. The model in MATLAB includes a modulator, a channel simulator, and a demodulator.

This chapter reviews the simulation implementation in details. To start, the OMNeT++ Integrated Development Environment (IDE) and project files are introduced in Section B.1. The implementation aspects are discussed in details in Section B.2. The core components, related parameters, and configurations are explained in Sections B.2.1 to B.2.3. The existing mobility models and simulated model are discussed in Section B.2.4. The integration process of MATLAB and OMNeT++ is provided, step-by-step, in Section B.2.5. Finally, collecting statistics, building, and running the project are explained in Sections B.3 and B.4.

B.1 OMNeT++ IDE

OMNeT++ is supported by the Eclipse IDE. It supports parallel distributed simulation, which is a key feature for scalability. A simulation in OMNeT++ has two different interfaces. The first one is *Cmdenv*, which is a command-line user interface for batch execution. The second interface is *Tkenv*, which is a TCL/Tk-based graphical and windowing user interface. A desired runtime environment should be selected in the simulation executable [35].

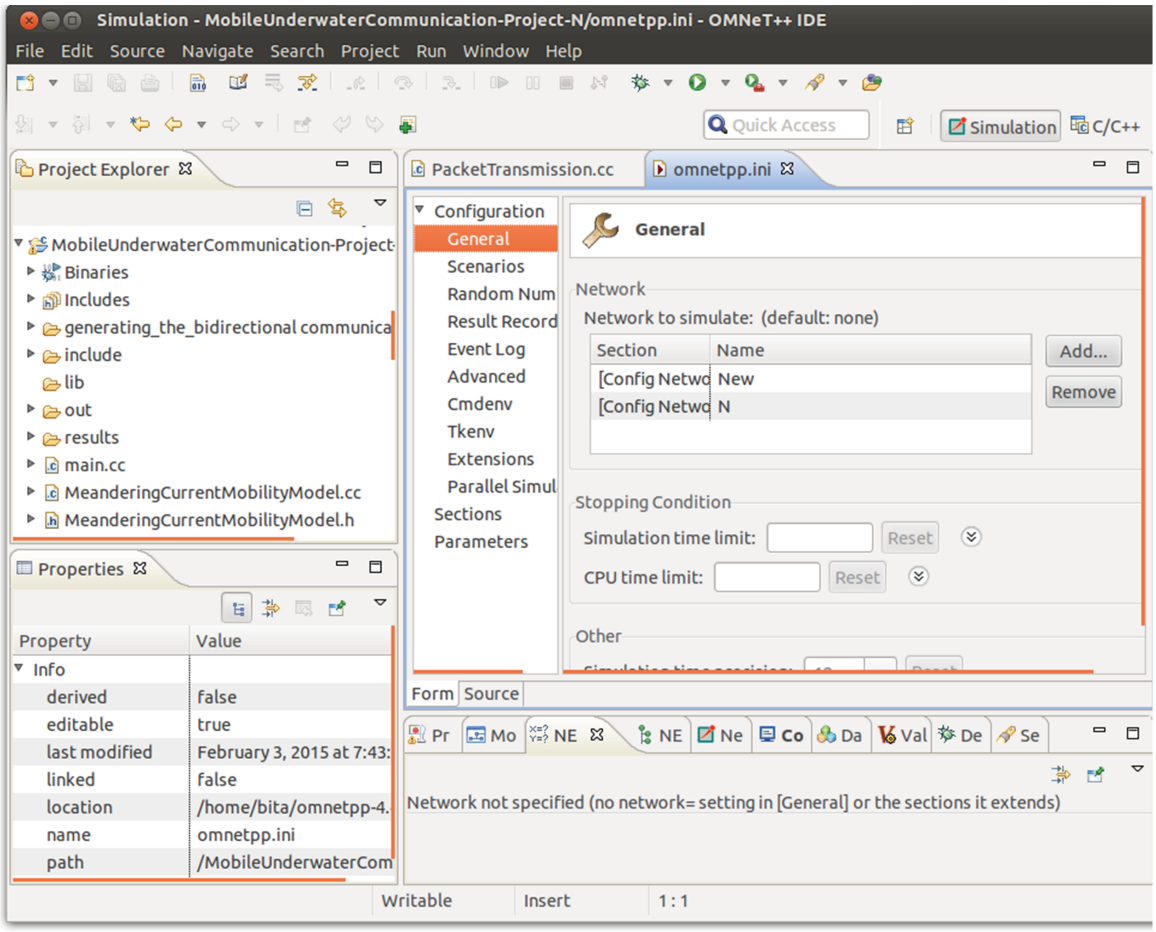

Figure B.1: OMNeT++ IDE.

Figure B.1 represents the OMNeT++ IDE of the project [35]. The top-left window

is the *project explorer*. Under the project name (*MobileUnderwaterCommunication-Project*), the list of project files are listed (see Section B.1.1). The bottom-left window is the *Properties*. It edits and customizes the objects properties. The main right window is the workspace. All project files are opened in the workspace. For example, the *INI file editor* (*form-based*, in this case) is opened here. The upper bars are the *tool-bar* and *menu-bar*. The bottom-right window shows other views, such as *Problems*, *Tasks*, and *Module Hierarchy*.

The core concepts in OMNeT++ are simple module, compound module, gate, channel, and message (see Figure B.2). A simple module is the smallest structural unit. It is at the lowest level of the module hierarchy. It represents the algorithmic part of the model. Simple modules are integrated to make compound modules. In other words, a compound module is a group of one or several simple modules. Unlike a simple module, a compound module does not have any active behavior by itself. It leads to a model consisting of hierarchically nested modules, which communicate with each other via messages [35, 44].

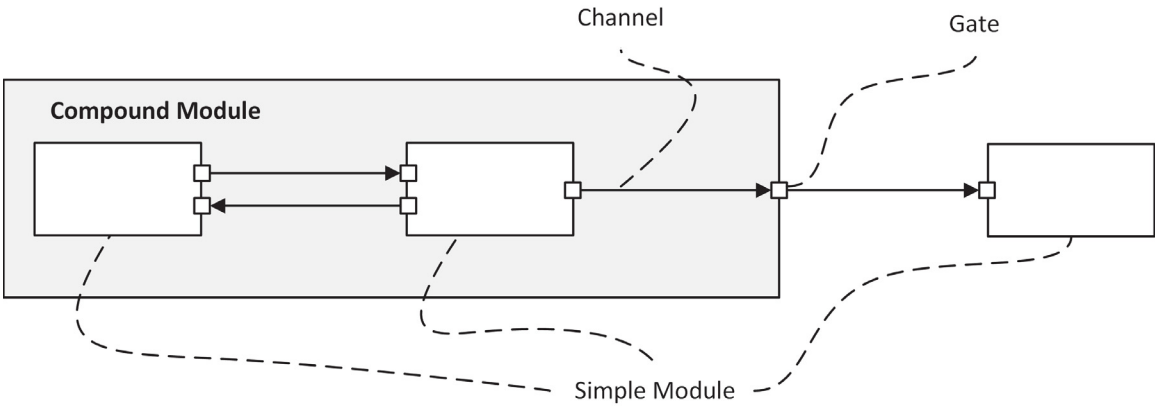

Figure B.2: OMNeT++ core components: simple modules, compound module, gates, and channels.

Simple and compound modules have input and output interfaces. These connection points are called gates. Messages are transferred through gates. Channels

simulate connections using different parameters, such as bit error rate, propagation delay, and data rate. Messages are sent either directly or through channels. These parameters are used to customize the module behavior and parameterize the model topology [35].

OMNeT++ mainly consists of three types of files: NED file (with *.ned* extension), C++ file (with *.cc* and *.h* extensions), and INI file (with *.ini* extension).

The model structure is defined by a high level topology description language. It is called *NEtwork Description (NED)*. An NED model is represented in two ways: with a text editor and with a graphical editor, called *Graphical NEtwork Description (GNED)*. NED includes the initial settings and configurations of the simulation. Several submodule and module parameters should be instantiated in NED files. Different network topologies are possible with by NED, such as mesh, ring, hypercube, chain, and user-defined topologies [35].

OMNeT++ uses C++ as a primary programming language for making modules. All required simple modules should be programmed into a C++ file. C++ is used to implement the functionality of simulation components, such as modules and channels.

An INI file is used to configure the simulation models to run. Parameters and variables are modified in an INI file. In addition, the network configurations are done in this file. It has a form-based editor (see Figure B.3) and a source editor (see Figure B.4). As it can be seen, the form-based editor presents different configurations (in the left window) and their related parameters (in the right window) graphically. In contrast, parameters and their assigned values should be programmed into an INI source editor [35].

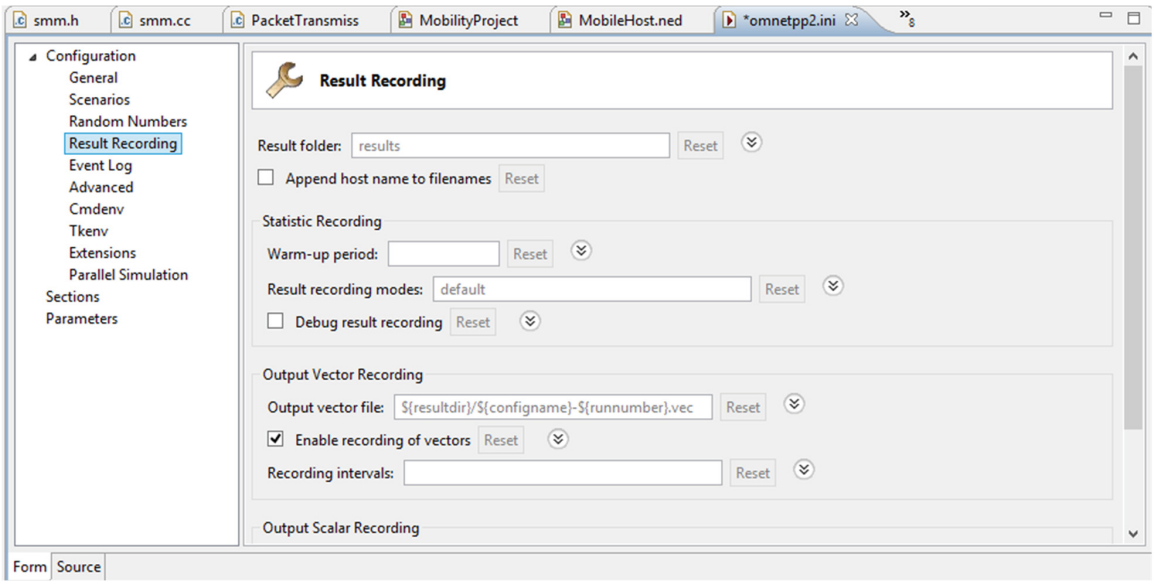

Figure B.3: The INI file form-based editor.

B.1.1 The Project Files

This section introduces the main project files. The defined network in the project is named *UWASN*. It consists of two NED files:

- *MobileHost.ned*
- *MobilityProject-NED.ned*

These files work as hosts for demonstrating the mobility model. The configurations related to the simple module (i.e., *PacketTransmission*), compound module (i.e., *MobileHost*), gates, and channels are done in these two NED files. More details are provided in Section B.2.1. There are five files related to the mobility model in the project:

- *MeanderingCurrentMobilityModel.cc*
- *MeanderingCurrentMobilityModel.h*
- *MeanderingCurrentMobilityModel.ned*

```
smm.h   smm.cc   PacketTransmiss   MobilityProject   MobileHost.ned   *omnetpp2.ini

[General]
 tkenv-plugin-path = ../../../etc/plugins
 sim-time-limit =  1 day
 debug-on-errors = true
 debugger-attach-on-error = true

###########################################################

[Config Network-UWASN]
network  = UWASN

New.playgroundSizeX = 0m
New.playgroundSizeY = 0m
New.playgroundSizeZ = 0m

**.node*.mobilityType = "MeanderingCurrentMobilityModel"
**.node*.mobility.MyMobilitySmmTimeWait = 0.001s
**.SmmTimeStep =0.01
**.numHosts = 2  # number of hosts
**.debug = true
**.updateInterval = 0.1s # lets getCurrentPosition update the display string from a test module
 updateInterval in mobility models: the simulation time interval used to regularly signal mobility state changes and update t


###########################################################

Form   Source
```

Figure B.4: The INI file source editor.

- *smm.cc*
- *smm.h*

The first three files contain the core logic of the mobility in OMNeT++ (see Section B.2.4). The last two files include the C++ code for the calculations and statistics of the MCM model [6]. There are three files related to the messages:

- *MUCmessage.msg*
- *MUCmessage_m.cc*
- *MUCmessage_m.h*

MUCmessage.msg defines the message format. All details regarding to the messages, their format and functionalities are discussed in Section B.2.2.

B.2 Implementation Aspects

This section discusses the implementation aspects of our UWASN simulation. To begin with, the OMNeT++ core components are explained in Section B.2.1. This section includes: simple and compound modules, gates, and channels. The message generation and message format are discussed in Section B.2.2. Next, more details regarding to the parameters and network configurations are provided in Section B.2.3. It also includes details for finding and updating the position of nodes. The existing mobility models and the simulated model are discussed in Section B.2.4. Finally, the integration process of OMNeT++ and MATLAB is explained in Section B.2.5.

B.2.1 Core Components

This section reviews four core components in OMNeT++: simple modules, compound modules, gates, and channels [35].

B.2.1.1 Simple Modules

A simple module is at the lowest level of the module hierarchy in OMNeT++ [42, 35]. It is defined to expose the algorithm of the work. A simple module is defined using class *cSimpleModule*:

```
class PacketTransmission : public cSimpleModule
{ ... }
```

PacketTransmission is the name of the defined simple module. It is the key module in the project. It is a subclass of class *cSimpleModule*. *cSimpleModule* is a base class for all simple module classes in OMNet++. Since it cannot do anything by itself, there is a need to subclass from it. Its virtual member functions need to be redefined in order to make them work according to the simulation requirements. The inheritance

diagram of *cSimpleModule* is shown in Figure B.5. *cSimpleModule* improves the functionality of the *cModule* class, which represents all the modules in the simulation. It provides the information related to gates and parameters. [45]

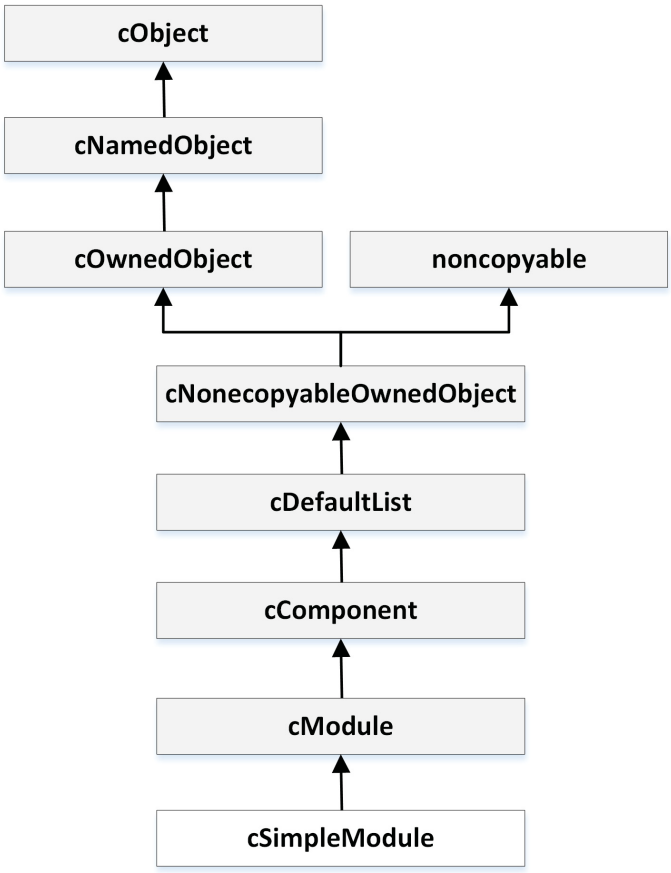

Figure B.5: The inheritance diagram of *cSimpleModule*.

Four virtual member functions of *cSimpleModule* are listed:

- void **initialize**()
- void **handleMessage**(cMessage *msg)
- void **activity**()

- void **finish**()

These four major functions should be redefined to work according to the simulation purpose. Once the module is created, the *initialize* function is called. This function defines and initializes different events, parameters, and variables such as state variables. After receiving a message, the *handleMessage* function is called by the simulation kernel to run the module internal logic. Furthermore, the *handleMessage* function is used to deliver sent messages to the modules (see Section B.2.2). *activity* is a function with an infinite loop. Use of this function is optional. Finally, the *finish* function is called when the simulation is finished, either by the user or simulation timer. It is mainly used for statistics collection (see Section B.3). In other words, most of the required data and statistics are collected at this stage. The following code shows the *cSimpleModule* and its functions in the project [35]:

```
# include  <OMNeT++.h>
...
class PacketTransmission : public cSimpleModule
{
  private:
  ...
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
    virtual MUCmessage *generateMessage();
    virtual void forwardMessage(MUCmessage *msg);
    virtual double CalculateDistance();
    ...
};

Define_Module(PacketTransmission); %register the module class
```

Three functions, including *initialize*, *handleMessage*, and *finish* are redefined. Once a simple module is defined from the C++ class (i.e., *PacketTransmission*), the name of the module needs to be registered using the *Define_Module* statement. The *CalculateDistance* function calculates the transmitter-receiver separation distance, a key parameter in this work. The *generateMessage*, *handlemessage*, and *forwardMessage* functions are used for generating and sending messages. More details are provided in Section B.2.2.

Initialize: The *initialize* function boots the execution and schedules an initial self-message. An arbitrary node should be chosen to send the first message. The following code shows the *initialize* function [35]:

```
void PacketTransmission::initialize()
{
        ...
        % Choose the node 0 to send the first message
        if (getIndex()==0)
        {
            MUCmessage *msg = generateMessage();
            scheduleAt(0.0, msg);
        }
        ...
}
```

As it can be seen, the node *zero* has been chosen to send the first message. *ScheduleAt* sends a message with a delay. The first input parameter of the *ScheduleAt* function indicates the desired delay time. In the above code, sending a message is done without delay since the first parameter equals zero. More details are found in Section B.2.2.

B.2.1.2 Compound Modules

An OMNeT++ project consists of compound modules in addition to simple modules. A compound module is a unit of grouped simple modules. The following NED code represents these two modules, omitting the bodies [35]:

```
 % Define a simple module:
 simple   PacketTransmission
{
    parameters:
        ...
    gates:
        ...
}

% Define a compound module:
 module MobileHost
 {
    parameters:
          ....
    gates:
         ....
    submodules:
           host[2]: PacketTransmission;
           ...
    connections:
           ...
 }
```

MobileHost is the name of the compound module, used in the project. It consists of a simple module, called *PacketTransmission*. Both simple and compound modules have parameters and gates. The submodule vector, called *host*, refers to instances of

PacketTransmission. [35].

B.2.1.3 Gates

Gates are used to connect the simple and compound modules as in the following NED code [35]:

```
module MobileHost
{
    parameters:
          ....

    gates:
        % Declare gates with two way connections:
        inout gate[];


    submodules:
          host[2]: PacketTransmission;
          ...

    connections:
          % Declare two way connections between gates
          host[0].gate++ <--> host[1].gate++;
}
```

inout gate[] defines a new gate vector with an unspecified size. *inout* refers to two way connections, both as an input and an output. In the *submodules* section, a vector *host*, with a size of two, is made from simple module *PacketTransmission*. In the *connections* section, two hosts are connected to each other, through their gates. The double arrow ($< -- >$) indicates a bidirectional connection. It consists of two

uni-directional connections (−− >, < −−). It is used to connect two *inout* gates [35].

B.2.1.4 Channels

Connections are simulated using channels in OMNeT++. Messages are sent through channels. They customize the module behavior, using a variety of parameters such as propagation delay, bit error rate, and data rate. A channel is defined as in the following NED code [35]:

```
network UWASN {
  parameters:
        ....
  types:
          channel UWChannel extends ned.DatarateChannel
          { ... }

  submodules:
      node[numHosts]: MobileHost {
            parameters:
                @display("i=,,#00ff00");
            }

  connections:
          node[0].gate++ <--> UWChannel <--> node[1].gate++;
}
```

In the *types* section, the *UWChannel* channel is defined. *ned.DatarateChannel* is a predefined channel type. *@display* adds properties to the components in the graphical environment, such as color, size, width, and icon (i.e., "i"). *node* is defined as a vector of *MobileHost*. A channel is assigned to the module in the *connections* section. Finally, two nodes are connected to each other using the *UWChannel* channel, through

their gates [35].

B.2.2 Defining Messages

Modules in OMNeT++ communicate with each other by exchanging messages [35]. It results in creating events in the simulation. That is to say, an event is a reception of a message by a receiver. Not only events are produced by sending messages from one module to other modules, but they are also generated by sending messages from a module to itself. A message is a C++ class, either a *cMessage* class or a class derived from that. The process of sending a message and generating an event consists of the three major steps: *send*, *scheduleAt*, and *cancelEvent*. They are implemented by the following methods [35, 45]:

- send (cMessage* msg, int gateid)
- scheduleAt (simtime_t t, cMessage* msg)
- cancelEvent (cMessage* msg)

The *send* method sends a message through a specified gate. The *ScheduleAt* method schedules the message sending process. It is used in case of a need to delay sending a message for a specific time period. A scheduled event can be canceled with the *CancelEvent* method.

Message format definition: A new messages is defined in a *.msg* file. The message fields are defined based on the simulation requirements. The *MUCmessage* is defined as below [35]:

```
Message MUCmessage
{
        fields:
        int source,
```

```
        int destination;
        string Data;
}
```

As it can be seen, *MUCmessage* contains three fields: source, destination, and data. The data field gets randomly generated data, which is used in the communications as input data.

Based on the defined *.mgs* file (*MUCmessage.msg*), two other files with the *.cc* and *.h* extentions (*MessageName_m.cc* and *MessageName_m.h*) are created using the tool *opp_msgc*. All getter and setter methods are automatically generated. Then, the related header file is included in the module code using: *#include "MUCmessage_m.h"*. The following code automatically generates a class from the *.msg* file. It results in creating two new files, called *MUCmessage_m.cc* and *MUCmessage_m.h*.

```
opp_msgc -Xnc -Xns MUCmessage.msg
```

The initialization process: The following C++ code represents an initialization process of a message [35]:

```
        % node 0 sends the first message
        if (getIndex() == 0)
        {
            % Boot the process scheduling the initial message
                as a self-message.
            MUCmessage *msg = generateMessage();
            scheduleAt(0.0, msg);
        }
```

The process starts with scheduling an initial self-message. An arbitrary node (i.e., node 0) sends the first message. Then, a message (i.e., *msg*) from class *MUCmessage*

is created. The *generateMessage()* function is called to generate a new message.

generateMessage: This function is represented as below [35]:

```
MUCmessage *PacketTransmission::generateMessage() {
            ...
            % source and destination addresses
            int src = getIndex();
            int n = size();
            int dest = intuniform(0,n-2);
            if (dest >= src) dest++;
            ...

            % Create a message object
            MUCmessage *msg = new MUCmessage();

            % Set source, destination, and data field
            msg->setSource(src);
            msg->setDestination(dest);
            msg->setData((const char *)TempRndStr.c_str());

            return msg;
}
```

First of all, the source and destination addresses are set in the first segment of code. The *getIndex()* function returns the index of the module. The *size()* function returns the size of the gate vector. *intuniform(0,n-1)* returns a random integer with uniform distribution in the range [0,n-1]. Then, a new message from *MUCmessage* is created. The new message fields are set: source, destination, and data field. Data is generated randomly. Finally, the function returns the new generated message *msg* [35].

handleMessage: The *handleMessage* function is shown below [35]:

```
void PacketTransmission::handleMessage(cMessage *msg)
 {
    ...
    MUCmessage *ttmsg = check_and_cast<MUCmessage *>(msg);

    % Check the destination:
    if (ttmsg->getDestination() == getIndex())
      {
              % Delete the message:
              delete ttmsg;

              % Generate a new message:
              MUCmessage *newmsg = generateMessage();

              % Forward the new message:
              forwardMessage(newmsg);
      }
    else
     {
              % Forward the message:
              forwardMessage(ttmsg);
     }
 }
```

In the *if statement*, the *handleMessage* function checks the reception of the message by the destination node. It deletes the received messages. Then, it generates a new message, calling the *generateMessage* function. At the end, the message should be forwarded, calling the *forwardMessage* function. In the *else statement*, the function forwards messages that have not reached the destination yet [35].

forwardMessage: Messages forwarding is done using the *forwardMessage* function. It is invoked from the *handleMessage()* function when a message arrives at the node. The *forwardMessage* function is shown as below [35]:

```
void PacketTransmission::forwardMessage(MUCmessage *msg) {
    ...
    int n = gateSize("gate");
    int k = intuniform(0,n-1);
    sendDelayed(msg, 0.1, "gate$o", k);
    ...
}
```

The *gateSize* function gets a gate name. It returns the size of the gate vector. The *intuniform(0,n-1)* function returns a random integer with uniform distribution in the range [0,n-1]. *k* is a random gate number. Messages are sent to other modules either by the *sendDelayed* or *send* methods. *sendDelayed* sends a message after holding it for a specified time interval. Its input parameters are: *message*(i.e., *msg*), delay time(i.e., 0.1), *gate*, and gate index. Two last parameters specify a gate that should be used to send a message out [35].

B.2.3 Parameters and Configurations

This section provides details about several parameters, network configuration, and *DisplayString*. It also provides information how to find and update the position of nodes in the simulation.

B.2.3.1 Parameters

Parameters in NED files: A global variable, common to a NED file and C++ code, must be declared in the *parameters* section of a NED file:

```
Simple PacketTransmission
{
    parameters:
        double SmmTimeStep @unit(s) = default(0.01s);
        ...
}
```

As it can be seen, the *SmmTimeStep* parameter is defined in the *parameters* section of the *PacketTransmission* simple module (see Section B.2.1.1). We define its data type (i.e., *double*) and a default value, using the keyword *default()*. The *SmmTimeStep* parameter is important in this simulation. Every change in its value, affects the simulated network according to the MCM model. In this model, the default value of this parameter is set to 0.01 *second*.

Units: A unit (i.e., *second* or *meter*) for a parameter is defined in a NED file, using the *@unit()* statement. In the above code, the seconds (s) is used for the *SmmTimeStep* parameter. In this work, the unit for the separation distances between senders and receivers is set to kilometer (km). Moreover, all timing parameters are in seconds (s).

Number of nodes: The number of nodes, initialized in the simulation, is defined in a NED file [35]:

```
network UWASN {
    parameters:
       % number of nodes with the default value 2 :
        int numHosts= default(2);
        ...
     submodules:
        node["numHosts"]: MobileHost
```

```
        ...
}
```

The *numHosts* parameter, with the default value *two*, is defined in the *UWASN* network. Next, the node vector (two nodes, in this case) is instantiated from the *MobileHost* module.

Parameters in the C++ code: To access and modify variables in the C++ code, they must be imported using the *par()* statement:

```
int Counter = par("SmmTimeStep");
```

As it can be seen, a new integer type variable is defined, named *Counter*. It is equivalent to the value of the *SmmTimeStep* parameter.

Parameters in INI files: The values of the parameters are modified in an INI source editor:

```
% Setting the value of SmmTimeStep:
**.SmmTimeStep = 0.3;
```

The value of the *SmmTimeStep* parameter is changed from its default value to 0.3 *second*.

UpdateInterval: It is the simulation time interval. OMNeT++ frequently monitors and signals position state changes, using the *UpdateInterval* parameter. Then, it updates the display accordingly [35, 42]. This parameter has the default value 0.1 *second*.

```
% Modify the simulation time interval:
```

```
**.updateInterval = 0.2s
```

The above INI code modifies the default value *UpdateInterval* to 0.2 *second*.

ConstraintArea: It refers to a three-dimensional area, used to control the movement of objects in a domain (in *meters*). *ConstraintAreaMinX* is a double type parameter. It is the minimum value of the x coordinate in the constraint area, with the default value -1 *meter*. *ConstraintAreaMaxX* is a double type parameter. It is the maximum value of the x coordinate in the constraint area, with the default value 1 *meter*. The same parameters are available for y and z coordinates, such as *ConstraintAreaMinY* and *ConstraintAreaMaxZ* [42].

Network Configurations in INI files: A project in OMNeT++ has one or several networks. In this work, *UWASN* is the name of the defined network in a NED file. The network configurations are provided for the whole project in the *General* setting section of an INI file [35]:

```
% General settings for the whole project:
[General]
tkenv-plugin-path = ../../../etc/plugins
sim-time-limit =  1day
...
```

Besides, settings specific to a particular network (i.e., UWASN) are defined:

```
% Settings specific to the UWASN network:
[Config Network-UWASN]
network = UWASN
**.debug = true
...
```

The *tkenv-plugin-path* parameter specifies the search path for *Tkenv plugins*. The *debug* parameter activates the debugging option for the simulation.

Simulation time: The execution time of the simulation is detemined by *sim-time-limit* (upper bound on the simulation time) and *cpu-time-limit* (upper bound on the wall-clock-time). It comes in different units, such as *minute*, *second*, *hour*, and *day* [35].

```
% Setting the value of sim-time-limit (in minutes)
sim-time-limit= 2 min

% Setting the value of cpu-time-limit (in seconds)
cpu-time-limit= 60 s
```

In the first segment of the code, the *sim-time-limit* is set to 2 *minutes*. Then, the *cpu-time-limit* is set to 60 *seconds*.

B.2.3.2 DisplayString

DisplayString is used for the automatic arrangement of module vectors to control the graphical display [35]. It contains different properties and parameters related to each module:

```
@display("bgb=$playgroundSizeX, $playgroundSizeY, white; bgp
    =10,10");
```

Statement *@display* declares the *DisplayString*, a parameter in double quotes. As it can be seen, the *DisplayString* parameter has several properties and tags to characterize the module, such as background information (i.e., *bgb*), playground size (i.e.,

$playgroundSizeY), playground color (i.e., *white*), and background coordinate offset (i.e., *bgp*).

To update the *DisplayString* parameter during the simulation, the *initFromDisplayString* parameter needs to be set to *true* in an INI file. When this parameter has value *true*, it specifies that nodes get their properties at run-time from the *DisplayString* parameter. The node properties are set manually with pre-defined values where the *initFromDisplayString* parameter is set to *false*, the default. In summary, *initFromDisplayString* controls whether the node properties come from *DisplayString* or from the pre-defined values. In this work, the properties are updated at run-time with *DisplayString* [35].

To obtain the *DisplayString* string (i.e., *dispStr*) of the current module, the *getDisplayString()* function is called [35]:

```
cDisplayString dispStr=getParentModule()->getDisplayString();
```

The function *getParentModule* returns the related module. To obtain the particular properties of *DisplayString*, *setTagArg* is used. In the following code, the *getDisplayString* function is called with several tags (i.e., *"ls", 0, "yellow"*), through the gate [35]:

```
Gate("gate")->getDisplayString().setTagArg("ls", 0, "yellow")
    ;
```

Besides, the *DisplayString* parameter is updated, using the *setDisplayString* function [35]:

```
Void setDisplayString(const char* dispstr);
```

This kind of setters updates the *DisplayString* parameter at run-time.

B.2.3.3 Finding the Position of Nodes

In the *DisplayString*, module and submodule parameters are specified using "$" sign as a name prefix: *"$ name"*. To obtain a position from *DisplayString* at run-time, the *getTagArg* function is used [35]:

```
getTagArg("p", 0)  % the last position of x
getTagArg("p", 1)  % the last position of y
```

The tag "p" shows the position of the current module. The value *zero* indicates the x position. The value *one* represents the y position. The required information is extracted from the module string, using several string-related functions, such as *find* and *substr*. The following C++ code implements finding the x and y positions from the module string [35]:

```
cDisplayString dispStr = getParentModule()->
    getDisplayString();

std::string str=(std::string)dispStr;

% Find the position of "p=" tag
std::size_t posP = str.find("p=");

% Find the position of ";"
std::size_t posEnd = str.find(";");
```

```
% From the position "p=" to end
std::string strPositions = str.substr (posP+2, posEnd-2);

% Find the position of ","
std::size_t posComma = strPositions.find(",");

% Find the first substring
std::string FirstStr = strPositions.substr(0, posComma);

% Find the second substring
std::string SecondStr = strPositions.substr(posComma+1);
```

First of all, the string *dispStr* of the current module is obtained, using *getDisplayString*. Then, the position of the tag "p" and tag ";" are determined, using the *find* function. Tag ";" indicates the end of a string. Different properties in a module string are separated by ",". As a result, the x and y positions are found from two comma-separated substring.

The *getParentModule* function returns the parent module of the current module. It returns a *NULL* value for a system module. An option for direct calls between modules is also provided: *parentModule()− >submodule("name")* [35]. To find the name and ID of the module, the *getindex()* function is used. It is defined in the *cModule* class. The *getindex()* function returns the module index in the vector [35]:

```
double MyModuleID = getParentModule()->getIndex();
```

Alternatively, the following C++ code updates the position of a node during the simulation:

```
Coord VecPos = Coord::ZERO;
```

```
cModule *module = getParentModule();

% Finding the related node:
cModule *nodeP = findContainingNode(module);

VecPos = MobilityAccess().get(getParentModule())->
    getCurrentPosition();
```

The *getParentModule* function finds the target module. The node related to the module is found, using the *findContainingNode* function. The position of node (i.e., *VecPos*) is obtained, using the *getCurrentPosition* function. *MobilityAccess* gives access to the *IMobility* submodule [35, 42].

Updating the positions: Besides the calculation of the x and y positions in the simulated mobility model, the last position of nodes needs to be updated, using the *setTargetPosition()* function. The positions are updated regularly to reflect the visual changes in the simulation graphical window.

B.2.4 Mobility Modeling

This section reviews existing mobility models and the simulated mobility model.

B.2.4.1 Existing Options for Mobility Modeling in OMNeT++

The movement of nodes in a mobile network is determined by mobility models. A mobility model can be either random or deterministic. A deterministic mobility model is based on mobility traces and scenarios. A mobility scenario is the recorded motions of mobile nodes during a simulation. To use a pre-defined scenario for the movement of nodes, the name of a scenario file is specified. Besides, the file needs to be added to the project folder. The following INI code specifies the name of a mobility trace file (i.e., *UWMovementSenario.movements*):

```
**.host*.mobility.tracefile = ("UWMovementSenario.movements")
```

host is a vector of instantiated nodes, which are defined earlier in a NED file (see Section B.2.1). The star sign suffix of *host* indicates that this setting is applied to all hosts. OMNeT++ also provides an option to use mobility patterns. They simulate the mobility, according to a mobility scenario file (i.e., a text file). In the following INI code, the scenario file name is set for the *mobilityPatternFile* parameter:

```
**.host*.mobilityPatternFile="myMobilityPatternFileName.txt";
```

Before setting the *mobilityPatternFile* parameter in an INI file, this functionality needs to be enabled, using the *useMobilityPattern* parameter:

```
**.host*.useMobilityPattern = true
```

This parameter has the default value of *false*. So, it should be set to true in order to simulate the mobility according to a scenario file (i.e., *myMobilityPatternFileName.txt*).

The scenario files can have different data formats. *BonnMotion* is one of them, used in the project as an extra option. It is a mobility scenario generator and analysis tool. The data format in this model is: (t, x, y, [z]). x, y, and z indicate a node position in a three-dimensional space. However, z is optional. It can be ignored in case of a two-dimensional space. t shows the time related to the position. In other words, a given node gets position (x_k, y_k) at time t_k. *BonnMotion* generates data with this format:

$$t_1\, x_1\, y_1\, [z_1] \;\; t_2 x_2 y_2\, [z_2] \;\; t_3\, x_3\, y_3\, [z_3] \;\; ...\; t_{k-1}\, x_{k-1}\, y_{k-1}\, [z_{k-1}] \;\; t_k\, x_k\, y_k\, [z_k]$$

To sum up, there are several ways to simulate mobility according to pre-defined scenarios. The mobility scenarios are made based on previously collected position-based data. This option is also made available in this work, in the *BonnMotion* format. Positions of nodes are classified based on node name. However, this work takes advantage of run time mobility simulation.

B.2.4.2 The Simulated Mobility Model

The MCM model [6] programmed a C++ file, named *smm.cc* [46]. It is the main library for the mobility model (see Section 3.4). Various streamfunctions are implemented in this model, such as *SMM-BLNKWAW (blinking_wavenumbers)*, *SMM-MNDRJET (meandering jet)*, *SMM-ALTCHAN (alternating_channels)*, *SMM-PULSVOR (pulsating_vortex)*, *SMM-STDYVOR (steady_vortex)*, and *SMM-TAYLJET (taylor_dispersion_jet)*. However, the *SMM-MNDRJET* is the main one, used in this model. In other words, the MCM model advects underwater sensors according to the *meandering jet* streamfunction.

Prior to start the deployment of sensors, a desired type of streamfunction is selected. It is set to *SMM_MNDRJET* in this work. In addition, the values of several key parameters (i.e., *SmmTimeStep*) need to be set. The initial deployment of sensors is done in a domain, considering the streamfunction. Initially, sensors are uniformly released over a limited area. Then, they disperse over a wider area [6].

To begin with, the UWASN is deployed with a specified number of sensors. Then, this number can be decreased or increased. The function, related to the movement of sensors, gets the initial position, as input parameters. It determines their new position based on the velocity fields, defined by their streamfunction. This process moves forward according to the value of *SmmTimeStep*. The previous positions are overwritten with new positions at *current_time*, which is incremented by *SmmTimeStep* [46].

The simulated mobility model in OMNET++ consists of three major files:

MeanderingCurrentMobilityModel.cc
MeanderingCurrentMobilityModel.h
MeanderingCurrentMobilityModel.ned

These files mainly include the C++ and NED code to simulate the MCM model. To use the functionality of existing mobility models in INET, *MobilityBase* is imported. *MobilityBase* is an abstract base module for mobility models [44, 42]. It has four subclasses: *MovingMobilityBase*, *StaticGridMobility*, *StationaryMobility*, and *TraCIMobility* (see Figure B.6). *MeanderingCurrentMobilityModel.ned* defines a simple module (*MeanderingCurrentMobilityModel*) which extends *MovingMobilityBase* [35]:

```
import inet.mobility.models.MobilityBase;

simple MeanderingCurrentMobilityModel extends
    MovingMobilityBase
{
    parameters:
        % The simulation time interval
        double UpdateInterval @unit(s) = default(0.1s);
    ...
}
```

UpdateInterval is the simulation time interval. OMNeT++ frequently monitors and signals the position state changes, using this parameter. Then, it updates the display accordingly. The value *zero* turns off the signal. Its default value is set to 0.1 second [35].

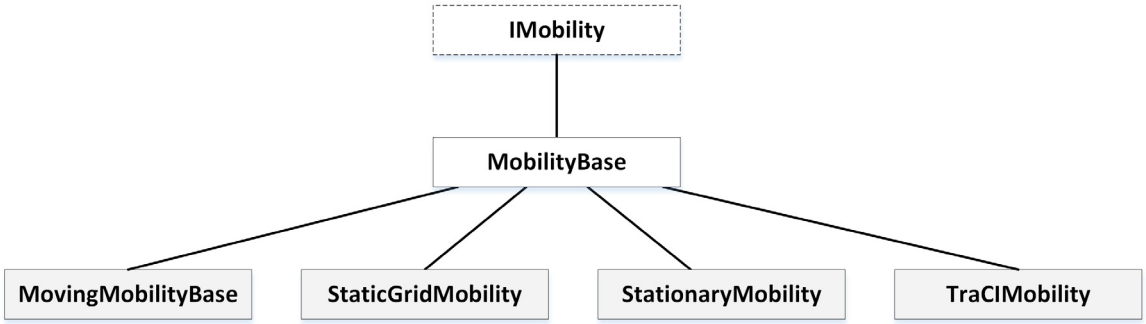

Figure B.6: The inheritance diagram of *MobilityBase*.

MovingMobilityBase is a base class for mobility modules [35, 42]. It has four subclasses: *CircleMobility*, *LinearMobility*, *LineSegmentsMobilityBase*, and *RectangleMobillity* (see Figure B.7). *LineSegmentsMobilityBase* is a base class for mobility models with linear movements in INET. The subclasses redefine the function *setTargetPosition()* to update the positions. In other words, this function is called after reaching the target position. Then, a new target position is set. The function *Move()* leads to the movement of a node, according to the current simulation time. The subclasses must override and update several attributes, according to the mobility model [44].

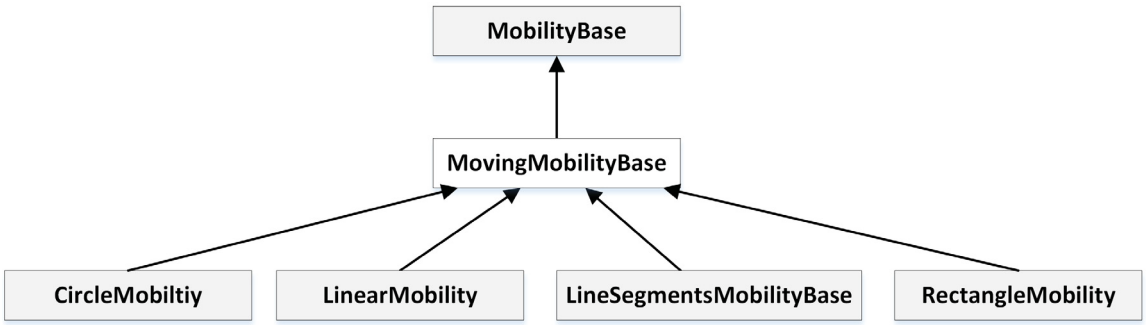

Figure B.7: The inheritance diagram of *MovingMobilityBase* and *IineSegmentsMobilityBase*.

To put it briefly, the nodes get their initial x and y positions after the first time

deployment. Then, the nodes with their initial position are displayed in a *TCL/TK* window. Next, they start to move according to the MCM model. The simulated mobility model can be used, similar to other mobility models in the INET and MiXiM frameworks. Then, the name of the mobility model (*MeanderingCurrentMobilityModel*) is assigned to the *mobilityType* parameter in an INI file:

```
**.node*.mobilityType = "MeanderingCurrentMobilityModel"
```

B.2.5 The Integration Process of OMNeT++ and MATLAB

This section reviews the integration process of OMNeT++ and MATLAB. To begin with, a MATLAB shared library is built (see Section B.2.5.1). Next, the initialization functions are called in OMNeT++ to access the shared library code (see Section B.2.5.2). Then, the required variables and parameters are defined (see Section B.2.5.3). The modulation, channel simulation, and demodulation are done to simulate the physical layer (see Sections B.2.5.4 to B.2.5.6). Several metrics are calculated according to the simulation output (see Section B.2.5.7). At the end, memory deallocation is addressed (see Sections B.2.5.8).

B.2.5.1 Building a MATLAB Shared Library

A MATLAB shared library is built to make MATLAB functions available in OMNeT++. The following bash shell script builds a MATLAB shared librariy, which is used in the simulation [13, 35]:

```
# !/bin/sh
rm -f *$~
$mcc -B csharedlib:libuwpsk -v A.m absorption.m chirp.m
      demodulatePSK.m modulatePSK.m mylowpassfilter.m
    simulateChannel.m spower.mm
cp libuwpsk.so ../lib/
```

libuwpsk.so is the name of the shared library, created by the MATLAB compiler *mcc*. All related MATLAB files and functions with *.m* extension (i.e., *modulatePSK.m*) are added to the script. The last line of the script, copies the shared library to the folder *lib* in order to put it in the library path.

B.2.5.2 Initialization

Prior to the use of a shared library in OMNeT++ [35], the initialization routine is executed. It includes calling the initialization functions: *mclInitializeApplication* and *LibraryNameInitialize (i.e., libuwpskInitialize, in this case)* [47, 13]:

```
% Initialization:
if(!mclInitializeApplication(NULL, 0)){
        fprintf(stderr, "Could not initialize the MATLAB
           application (MCL).");
        exit(-1);
}
printf(MCL initialized successfully.");

if(!libuwpskInitialize()){
        fprintf(stderr,"Could not initialize libuwpsk.so");
        exit(-1);
}
printf("libuwpsk.so initialized successfully.");

setupUserInterface(argc, argv);

% Termination:
libuwpskTerminate();
mclTerminateApplication();
```

```
printf("MCL and libuwpsk terminated successfully.");

printf("End of simulation");
```

The first two segments of code call the initialization functions. After a successful initialization, the *setupUserInterface* function is called. It starts the OMNeT++ execution. At the End, the process is terminated by calling the termination functions [47]: *libuwpskTerminate* and *mclTerminateApplication*.

B.2.5.3 Assignment of Values

There are several formal parameters defining the simulation of a physical layer [13]. *ByteStream* contains randomly generated data, used in the communications between a sender and a receiver. Parameter d represents the distance (in *kilometer* (km)) between a sender and a receiver nodes, moving according to the MCM mobility model. Parameter d is calculated while exchanging messages. Parameter *sps* specifies the symbol rate (in *baud*) with the default value 1000 *baud*. Parameter P (in μPa) is the transmission acoustic pressure in linear form. It is set to the value $10^{14}\ \mu\, Pa$. Parameter fc is the carrier frequency (in *Hertz*). It has the default value 20000 *Hertz*. The selected values are in the range of values supported by actual commercial products [1]. Parameter fs indicates the sampling frequency (in *samples per second* (*sps*)). It is set to the value 100000 *sps* in this work. This value meets the Nyquist's criterion, with oversampling.

Matrix Array: Each variable in MATLAB is a matrix. As a consequence, there is a need for a common data type in both C++ and MATLAB to exchange data between each other. It is called Matrix Array (*mxArray*) [48]. It is a basic opaque type. All input arguments and output values of functions are of *mxArray* type. Even a simple data type, such as *double*, is converted into an array of one element. The header file, containing this data type, is *matrix.h*. The *mxCreateDoubleScalar* function is

used to create a scalar double *mxArray*. Its signature is: *mxArray *mxCreateDoubleScalar(double value);*. This function uses *value* to initialize an array. It returns a pointer to the created array. The following code uses the *mxGetPr* function to convert data into an array of doubles [39]:

```
% Define a variable to point to the double:
double *data;

mxArray *in;

% Get a pointer to the double data in mxArray:
data = mxGetPr(in);
```

data is a variable that points to the double. *mxGetPr* gets a pointer to a *mxArray* of type double (*in*). It returns a pointer to a copy of the double value, stored in the MATLAB engine.

MLX and MLF Library Functions: Each *.m* file in MATLAB has two shared library functions [39]. The first one starts with *mlx* (*mlxFunctionName*). The second library function starts with *mlf (mlfFunctionName)*. They are different in a way that their arguments are passed. A *mlf* function is easier to use in comparison with a *mlx* function. A *mlx* function uses the *plhs[]* and *prhs[]* vectors. A *plhs* vector includes pointers to each output parameter of the function in a MATLAB *.m* file. Output parameters are located in the left side of the parameters list, prior to input parameters. Pointers to right side parameters, as inputs, are included in a *prhs* vector. An index inside brackets "[ ]" indicates a specific parameter. For example, *prhs[0]* points to the first right side parameter (the first input) and *plhs[1]* points to the second left side parameter (the second output). A *mlf* library function uses the name of parameters. Here is an example of *mlxModulatePSK* and *mlfModulatePSK*, created after building the shared library *libuwpsk* from the *ModulatePSK* function in MATLAB:

```
% modulatePSK.m in MATLAB:
```

```
function [txPSK numberOfSamples] = modulatePSK(byteStream, fs
    , fc, symbolsPerSecond)
{
  ...
}


% mlxModulatePSK in libuwpsk:
bool MW_CALL_CONV mlxModulatePSK(int nlhs, mxArray *plhs[],
    int nrhs, mxArray *prhs[])
{
      return mclFeval(_mcr_inst, "modulatePSK", nlhs, plhs,
          nrhs, prhs);
}


 % mlfModulatePSK in libuwpsk:
bool MW_CALL_CONV mlfModulatePSK(int nargout, mxArray** txPSK
    , mxArray** numberOfSamples, mxArray* byteStream, mxArray*
     fs, mxArray* fc, mxArray* symbolsPerSecond)
{
     return mclMlfFeval(_mcr_inst, "modulatePSK", nargout, 2,
         4, txPSK, numberOfSamples,byteStream, fs, fc,
         symbolsPerSecond);
}
```

The first segment of code shows the signature of the MATLAB function *modulatePSK* in the *modulatePSK.m* file. It has four input parameters: *byteStream, fs, fc,* and *symbolsPerSecond*. It has two output parameters: *txPSK* and *numberOfSamples*. The second segment of code shows the corresponding *mlx* library function (*mlxModulatePSK*) in the shared library *libuwpsk*. The last segment of code represents the

corresponding *mlf* library function (*mlfModulatePSK*) in the shared library *libuwpsk*.

B.2.5.4 ModulatePSK

The following code shows the signature of the *ModulatePSK* function in MATLAB [13, 14]:

```
function [txtPSK numOfSamples] = modulatePSK(byteStream, fs,
    fc, symbolsPerSecond)
```

This function has four input parameters. Firstly, *byteStream* is input data, which needs to be modulated. Other input parameters are: sampling frequency fs, carrier frequency fc, and symbol rate *sps* (see Section B.2.5.3). The function has two output parameters: *txtPSK* and *NumberOfSamples*. *txtPSK* contains a list of samples, a PSK modulated signal [13]. The corresponding *mlf* library function in OMNeT++ is:

```
mlfModulatePSK(NumberOfOutputArguments, &txPSK, &
    NumberOfSamples, byteStream, fs, fc, sps)
```

mlfModulatePSK is the *mlf* library function of the MATLAB function *ModulatePSK*. To call the *mlf* function in OMNeT++, the first parameter of the function should be specified. It is called *NumberOfOutputArguments*, which indicates the number of output parameters. It is equal to two. All output parameters, such as txPSK, should be initialized to *NULL* (i.e., *mxArray *txPSK = NULL;*). It prevents potential errors related to *mxArray* variables.

B.2.5.5 SimulateChannel

SimulateChannel is a channel simulator implemented in MATLAB. It considers noise and attenuation [14]. Attenuation is modeled as a function of distance and frequency.

It is simulated using the *Thorp* model (see Section 2.2.5). The following code shows the signature of the MATLAB function *SimulateChannel* [14]:

```
function[samples nr S N]=SimulateChannel(Signal, fs, sps, d,
    P)
```

The function has five input parameters: *Signal*, *fs*, *sps*, *d*, and *P*. *Signal* refers to samples of a modulated signal, e.g., obtained as an output of the *mlfModulatePSK* function. *d* represents a distance between a sender and a receiver (in *kilometer*). *P* is a transmission acoustic pressure in linear form. It has the default value 10^{14} $\mu\,Pa$ in the simulation. This function has four output parameters: *samples*, *nr*, *S*, and *N*. *samples* corresponds to the samples of a modulated signal affected by the channel. *nr* indicates the length of samples. The output parameter *S* is the received signal acoustic pressure in linear form (in μPa). *N* is the peak noise acoustic pressure in linear form (in μPa). The corresponding *mlf* function in OMNeT++ is:

```
mlfSimulateChannel(NumberOfArguments, &Samples, &nr, &S, &N,
    signal, fs, sps, d , P)
```

NumberOfArguments is set to the value four since the function has four output parameters. The output of the *SimulateChannel* function is an attenuated and a noisy signal.

B.2.5.6 DemodulatePSK

The following code shows the signature of the *DemodulatePSK* function in MATLAB [13, 14]:

```
function rxPSK = demodulatePSK(PacketWaveForm,
    numberOfBytes, fs, fc, symbolsPerSecond)
```

PacketWaveForm contains the output of the *simulateChannel* function. It represents the samples of a modulated signal. *numberOfBytes* is equivalent to the size of the input data: *sentPacket.length*. *rxPSK* is the output of the MATLAB *DemodulatePSK* function. It contains demodulated data, obtained from a demodulation process [13]. The output data is converted back into characters at the end. The corresponding *mlf* function in OMNeT++ is:

```
mlfDemodulatePSK(NumberOfArguments, &rxPSK, PacketWaveForm,
    numberOfBytes, fs, fc, sps)
```

mlfDemodulatePSK is the *mlf* library function, corresponding to the MATLAB *DemodulatePSK* function in the shared library *libuwpsk*. *NumberOfArguments* is equal to one, as there is only one output parameter.

B.2.5.7 Metrics and Parameters

In this section, we calculate distance and BER.

B.2.5.7.1 Distance

The distance between two nodes is calculated, while they are moving according to the mobility model. In this work, distances are collected when nodes are communicating with each other. More details are provided in Section 5.4.1.

B.2.5.7.2 BER

The BER is calculated either in OMNeT++ or MATLAB, comparing input data ($txPSK$) and output data ($rxPSK$). The BER is defined by the Equation 5.2. More details are provided in Section 5.4.2. There are different ways to calculate the BER

in MATLAB. For example, using the *biterr* function in MATLAB:

```
[Number, Ratio] = biterr(x, y, k)
```

biterr computes the number of bit errors (*Number*) and bit error rate (*Ratio*). k is a number of bits, used to represent each element in x and y [39]. This function compares the binary representation of the elements in x and y with each other. It returns the number of bit differences in *Number*. *Ratio* is the ratio of *Number* to the total number of bits. The following function also calculates the BER in MATLAB:

```
// The function BERratio calculates the BER for input and
   Output data
function BERratio = ClaculateBER(InputByteStream,
   OutputByteStream)

// It computes a length of input data
LengthOfByteStream = length (InputByteStream);
NumOfBitErrors = 0;

for k=1: LengthOfByteStream
    // Compare input and output ByteStream bit by bit
    if InputByteStream (k) != InputByteStream (k)
        NumOfBitErrors = NumOfBitErrors +1;
    end
end

// It returns the BER
BERratio = NumOfBitErrors / LengthOfByteStream;
```

In this function, *InputByteStream* represents input data in bits, before transmission.

OutputByteStream shows the received data in bits, after transmission. At the End, *BERratio* is calculated as the ratio of error bits over the total number of transmitted bits. We can use both functions.

B.2.5.8 Deallocation of Memory

At the end of a simulation, the memory, occupied by *mxArrays*, should be deallocated, using the *mxDestroyArray* function [35, 39]:

```
void mxDestroyArray(mxArray *array_ptr);
```

It prevents the potential memory management issues. This function needs the *matrix.h* header file to be added to the code. The following code represents freeing the occupied memory:

```
% When finished using the array, deallocate its space.
% Free the memory created:
  mxDestroyArray(fs);
  mxDestroyArray(fc);
  mxDestroyArray(sps);
  mxDestroyArray(d);
  mxDestroyArray(P);
  mxDestroyArray(numberOfBytes);
  mxDestroyArray(mxMyByteStream);
  mxDestroyArray(txPSK);
  mxDestroyArray(numberOfSamples);
  mxDestroyArray(samples);
  mxDestroyArray(nr);
  mxDestroyArray(rxPSK);
```

As it can be seen, *mxDestroyArray* frees the memory, created by different *mxArrays*, such as *fs* and *d*. Freeing an invalid pointer usually causes a segmentation fault or a simulation fatal error.

B.3 Collecting Statistics

The simulation outputs are collected programmatically in the simple modules of the project [35]. The results are saved in the output scalar (with *.sca* extension) and output vector (with *.vec* extension) files. First of all, output vectors should be created and initialized in the *initialize* function. Next, the values of vectors are collected in the *handleMessage* function, per each new event. Scalars are collected in the *finish* function since there is no need to collect them during the simulation. They need to be collected just once at the end of the simulation to record the final results. The following C++ code records the results and statistics of the *PacketTransmission* module in OMNeT++ [35]:

```
void PacketTransmission::initialize()
{
        % Create the output vectors:
        cOutVector currentPosXVec;
        cOutVector currentPosYVec;

        % Set the name of output vector:
        currentPosXVec.setName("posx");
        currentPosYVec.setName("posy");
}


void PacketTransmission::forwardMessage(MUCmessage *msg)
{
```

```
        % Record output vectors as a .vec file
        % keep statistics for positions:
        currentPosYVec.record(targetPosition.x);
        currentPosYVec.record(targetPosition.y);
}


void PacketTransmission::finish()
{
        % Record scalar statistics in the finish() method as
           a .sca file
        recordScalar("x", targetPosition.x);
        recordScalar("y", targetPosition.y);
}
```

cOutVector collects the simulation results, as an output vector. It writes doubles to an output vector file. All results, with *.vec* and *.sca* extensions, are kept in the *results* sub-folder of the project. They are used to generate plots and histograms.

B.4 Building and Running the Simulation

B.4.1 Building the Project

Prior to building a project in OMNeT++, there is a need to set several properties related to debugging and running. It is done by right-clicking the name of the project in the *project explorer* window (see Section B.1). Then, the *Run/Debug setting* tab is selected. There are two options: editing existing configurations and adding a new configuration. It can be either *C/C++ application* or *OMNeT++ simulation*. In the *edit configuration* window (see Figure B.8), there are several options to choose: an *INI* file, a *config name*, and a *run number*. A user interface should be selected among a *command view* and *TCL/TK* graphical window. There is also an option to record

event logs during the simulation. An option to debug potential errors is also available in the configurations.

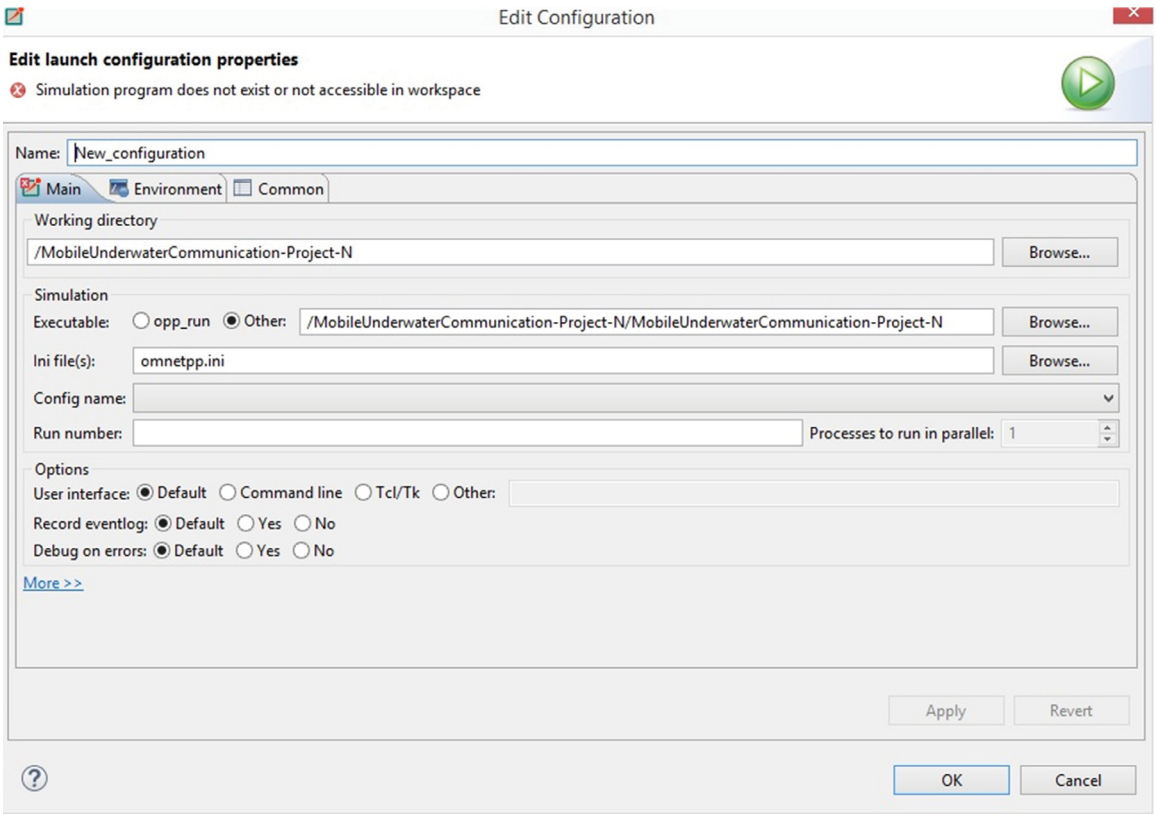

Figure B.8: The Edit Configuration Window.

***opp_makemake*:** It builds the projects in OMNeT++. *opp_makemake* generates a makefile based on the existing source files in the current directory of the project. Several sample projects are provided in the INET and MiXiM frameworks. Apart from that, new source files should be added to the source (*src*) sub-folder in case of creating a new project. *opp_makemake* needs several configurations and settings related to the libraries and compiler flags.

Libraries: In the case of various source files in different directories, the code should be linked to these directories. *opp_makemake* needs additional options: (*opp_makemake*

option1 option2 ...). To put it another way, all necessary libraries and library paths should be added, using appropriate switches. More information are found using the command: "opp_makemake -h". It provides several tips and helps about the switches of *opp_makemake*. Two switches, "-w lib" and "-w cpplib", should be added in case that any MATLAB compiler generated code are loaded in other applications. "-a" is an option which makes a *.so* file.

To use external libraries, located out of the source directory tree, the include path should be specified for the header files using: "-I IncludeDirectory". The system and OMNeT++ headers are exempted since they have been included automatically. An external library is linked with the project using: "L Directory". It links the directory of external library to the project library path. The name of the external dependency should be specified using: "-l LibraryName".

Problems of libraries are very common during the simulation, since the project is dependent on libraries and references. For example, the libraries related to GNU Scientific Library (GSL) should be added to the project, otherwise it leads to errors. They should be added in the *project properties* and *makemake* tab. It is done by choosing either the *executable* or *shared library* options. Also, the references should be set via the *project references* tab in the *project properties* window.

B.4.2 Running the Simulation

Calling the project executable in OMNeT++, starts the GUI (*Tkenv*) or command line (*Cmdenv*) window. The simulation with several runs (i.e., 10 times) is started by the following shell script [35]:

```
#! /bin/sh
for ((i=1; $i<=10; i++)); do
  ./omnetpp.ini -r $i      % $i = run number
Done
```

"-r" specifies the number of runs to be executed. "$i" indicates the run number which is in a range of 1 to 10, for 10 runs. *omnetpp.ini* indicates the name of *INI* file, which is *omnetpp* by default. When there are more than one INI file, the shell script is written as follow:

```
#! /bin/sh
foreach f (*.ini)   % Loop for all .ini files
nice +10 ./simulation -f $f >! $f:r.log
end
```

It starts the simulation for each *INI* file. "-f" specifies the *INI* file. There is a capability to assign different parameters to each simulation independently.

Appendix C

Decibel and Linear Form

The decibel scale (dB) is a logarithmic unit. It is an easy form to express big numbers and ratios. To convert a number X in the linear form into its corresponding value Y in the decibel form, the following equation is used [1]:

$$Y_{dB} = 10.log_{10}(X) \tag{C.1}$$

To convert a number Y in the decibel form into its corresponding value X in the linear form, the following equation is used [1]:

$$X = 10^{\frac{Y_{dB}}{10}} \tag{C.2}$$

In the calculations of this work, most of big numbers and values are expressed in the decibel form.

Bibliography

[1] M. Barbeau, "Wireless mobile communications, networks, and security." Draft, 2015.

[2] "Underwater Acoustic Sensor Networks (UW-ASN)." Available:`http://www.ece.gatech.edu/research/labs/bwn/UWASN/`, 2015. [Online; Last Access: 26-06-2015].

[3] Y. Xiao, *Underwater Acoustic Sensor Networks*. Auerbach Publications Boston, MA, USA, 2009.

[4] Y. M. Huang, H. C. Chao, D. J. Deng, and J. H. Park, *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*. Springer, 2013. Lecture notes in Electrical Engineering, HumanCom and EMC.

[5] J. Heidemann, W. Ye, J. Wills, A. Syed, and Y. Li, "Research challenges and applications for underwater sensor networking," in *IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 1, (Las Vegas, NV), pp. 228–235, IEEE, April 2006.

[6] A. Caruso, F. Paparella, L. F. Vieira, M. Erol, and M. Gerla, "The meandering current mobility model and its impact on underwater mobile sensor networks," in *The 27th IEEE Conference on Computer Communications (INFOCOM)*, (Phoenix, AZ), IEEE, April 2008.

[7] "Disovery of sound in the sea." Available:`http://www.dosits.org/people/communications/transmitdataunderwater/`, 2015. [Online; Last Access: 27-07-2015].

[8] "Tsunami warning system." Available:`https://en.wikipedia.org/wiki/Tsunami_warning_system`, 2015. [Online; Last Access: 28-07-2015].

[9] G. Taraldsen, T. A. Reinen, and T. Berg, "The underwater GPS problem," in *The IEEE Conference on OCEANS-Spain*, (Santander), pp. 1–8, IEEE, June 2011.

[10] R. P. Hodges, *Underwater acoustics: Analysis, design and performance of sonar*. John Wiley and Sons, July 2010.

[11] W. Stallings, *Wireless communications and networks*. Pearson Education Limited, 2 ed., 2009.

[12] "Energy per Bit to Noise Power Spectral Density Ratio (E_b/N_0)." Available:`https://en.wikipedia.org/wiki/Eb/N0`, 2015. [Online; Last Access: 05-08-2015].

[13] B. S. Borowski, *Application of channel estimation to underwater, acoustic communication*. PhD thesis, Stevens Institute of Technology, Castle Point on Hudson, Hoboken NJ, 2010.

[14] M. Barbeau, S. Blouin, G. Cervera, J. Garcia-Alfaro, B. Hasannezhad, and E. Kranakis, "Simulation underwater communications with colored noise approximation and mobility," in *The 28th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, (Halifax, NS), pp. 1532–1537, IEEE, May 2015.

[15] M. Lanzagorta, "Underwater communications," in *Synthesis Lectures on Communications*, vol. 5, pp. 1–129, Morgan and Claypool Publishers, 2012.

[16] S. Kumar and S. Babu, "Underwater communications," in *The IEEE Conference on Underwater Technology (UT)*, (Chennai), pp. 1–3, IEEE, Feburary 2015.

[17] S. Sendra, J. Lloret, J. Jimenez, and L. Parra, "Underwater acoustic modems," *The IEEE Sensors Journal*, vol. PP, p. 1, June 2015.

[18] A. C. Singer, J. K. Nelson, and S. S. Kozat, "Signal processing for underwater acoustic communications," *The IEEE Communications Magazine*, vol. 47, pp. 90–96, January 2009.

[19] V. P. Bui and W. S. Yeoh, "Propagation and channel characteristics in seawater environment," in *IEEE Antennas and Propagation Society International Symposium (APSURSI)*, (Memphis, TN), pp. 701–702, IEEE, July 2014.

[20] W. Ren, G. Shomalnasab, and L. Zhang, "Acoustic underwater positioning on FPGA," in *The IEEE Conference on Oceans-St. John's*, (St. John's, NL), pp. 1–5, IEEE, September 2014.

[21] F. Qu and L. Yang, "Rate and reliability oriented underwater acoustic communication schemes," in *The 13th IEEE Digital Signal Processing Workshop and The 5th IEEE Signal Processing Education Workshop (DSP/SPE)*, (Marco Island, FL), pp. 144–149, IEEE, January 2009.

[22] I. Burdinskiy, I. Karabanov, M. Linnik, and A. Mironov, "Processing of phase-shift keyed pseudo noise signals of underwater acoustic systems with the Doppler effect," in *2015 International Siberian Conference on Control and Communications (SIBCON)*, (Omsk), pp. 1–4, IEEE, May 2015.

[23] H. Bhambri and A. Swaroop, "Underwater sensor network: Architectures, challenges, and applications," in *The IEEE International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 915–920, IEEE, 2014.

[24] C. Peach and A. Yarali, "An overview of underwater sensor networks," in *The Ninth International Conference on Wireless and Mobile Communications*, pp. 31–36, 2013.

[25] J. H. Cui, J. Kong, M. Gerla, and S. Zhou, "The challenges of building mobile underwater wireless networks for aquatic applications," *The IEEE Network Journal*, vol. 20, no. 3, pp. 12–18, 2006.

[26] S. Souiki, M. Feham, M. Feham, and N. Labraoui, "Geographic routing protocols for underwater wireless sensor networks: a survey," *arXiv preprint:1403.3779*, March 2014.

[27] C. Detweiller, J. Vasilescu, and D. Rus, "An underwater sensor network with dual communications, sensing, and mobility," in *The IEEE Conference on OCEANS-Europe*, (Aberdeen), pp. 1–6, IEEE, June 2007.

[28] H. Suzuki, N. Kato, T. Katayama, and Y. Fukui, "Motion simulation of an underwater vehicle with mechanical pectoral fins using a CFD-based motion simulator," in *The IEEE Symposium on Underwater Technology and Workshop on Scientific Use of Submarine Cables and Related Technologies*, (Tokyo), pp. 384–390, IEEE, April 2007.

[29] M. Arima, N. Ichihashi, and Y. Miwa, "Modelling and motion simulation of an underwater glider with independently controllable main wings," in *The IEEE Conference on OCEANS-EUROPE*, (Bremen), pp. 1–6, IEEE, May 2009.

[30] Z. Zhou, Z. Peng, J. H. Cui, Z. Shi, and A. C. Bagtzoglou, "Scalable localization with mobility prediction for underwater sensor networks," *The IEEE Journal on Transactions on Mobile Computing*, vol. 10, pp. 335–348, September 2011.

[31] A. S. Bower, "A simple kinematic mechanism for mixing fluid parcels across a meandering jet," *Journal of Physical Oceanography*, vol. 21, no. 1, pp. 173–180, 1991.

[32] "Fluid parcel." Available:`https://en.wikipedia.org/wiki/Fluid_parcel`, 2015. [Online; Last Access: 21-07-2015].

[33] "Streamfunction." Available:`https://en.wikipedia.org/wiki/Stream_function`, 2015. [Online; Last Access: 21-07-2015].

[34] R. M. Samelson, "Fluid exchange across a meandering jet," *Journal of physical oceanography*, vol. 22, no. 4, pp. 431–444, 1992.

[35] A. Varga, "OMNeT++ - User Manual." Available:https://omnetpp.org/doc/omnetpp/manual/usman.html, 2011. [Online; Last Access: 28-06-2015].

[36] "British Petroleum deepwater horizon oil spill in the Gulf of Mexico." Available:https://en.wikipedia.org/wiki/Deepwater_Horizon_oil_spill, 2015. [Online; Last Access: 10-09-2015].

[37] "Curve fitting toolbox." Available:http://www.mathworks.com/help/curvefit/. [Online; Last Access: 13-07-2015].

[38] "Gaussian models." Available:http://www.mathworks.com/help/curvefit/gaussian.html. [Online; Last Access: 13-07-2015].

[39] I. MathWorks, *MATLAB: the language of technical computing. Desktop tools and development environment, version 7*, vol. 9. MathWorks, 2005.

[40] J. G. Proakis and M. Salehi, *Digital Communications*. McGraw-Hill, New York: McGraw-Hill Education, 5, illustrated ed., 2007.

[41] H. P. Stern and S. Mahmoud, *Communication systems: analysis and design*. Prentice-Hall, Inc., 2003.

[42] A. Varga, "The OMNeT++/INET framework - User Manual." Available:https://inet.omnetpp.org/, 2010. [Online; Last Access: 24-06-2015].

[43] "The OMNeT++/MiXiM framework - User Manual." Available:http://mixim.sourceforge.net/. [Online; Last Access: 13-02-2015].

[44] "INET framework for OMNeT++/OMNEST." Available:https://omnetpp.org/doc/inet/api-current/neddoc/index.html. [Online; Last Access: 13-03-2015].

[45] A. Varga, "OMNeT++ API reference." Available:http://www.omnetpp.org/doc/api/main.html, 2007. [Online; Last Access: 19-06-2015].

[46] A. Caruso, F. Paparella, L. F. Vieira, M. Erol, and M. Gerla, "Simple Jet Meandering Model (SMM)." Available:https://github.com/antoniocaruso/smm, 2008. [Online; Last Access: 28-05-2015].

[47] M. Barbeau, S. Blouin, G. Cervera, J. Garcia-Alfaro, and E. Kranakis, "Appendix to simulation of mobile underwater communications." Available:http://people.scs.carleton.ca/barbeau/Publications/2015/CCECE_2015_Appendix.pdf, September 2014.

[48] "mxArray." Available:http://www.mathworks.com/help/matlab/apiref/mxarray.html, 2015. [Online; Last Access: 21-07-2015].