A Simulation Framework for Performance Analysis of Molecular Nano Communication Networks

Behzad Amirzadeh Shams

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
September 2014
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Elvan Yilmaz
Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Hadi Işik Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Dogu Arifler
Supervisor

Examining Committee

1. Assoc. Prof. Dr. Dogu Arifler

2. Assoc. Prof. Dr. Muhammed Salamah

3. Asst. Prof. Dr. Gürcü Öz

ABSTRACT

Nanonetworks have attracted a lot of attention over the past decade due to advances in nanotechnology and their wide range of biomedical, industrial, environmental, and military applications. Nanodevices are made up of nanoscale components and are capable of performing only simple computation, sensing, and actuation tasks. A nanonetwork is formed when nanodevices are interconnected. In a nanonetwork, nanodevices can cooperate and share information to achieve more complex tasks. Nanomachines can employ diffusion-based molecular communication as a possible, biocompatible information transport method. Unlike traditional communication techniques in which electromagnetic waves are employed as information carriers, molecules are considered as carrier signals to convey the information. For instance, a transmitter nanomachine encodes the message symbols into the molecular signals and then sends them into a fluidic propagation channel. These information molecules propagate in the medium according to Fick's laws of diffusion, and then are received by the receiver nanomachine at which the information is decoded. In this work, a simulation framework for molecular nano communication networks will be developed. In particular, transmission, propagation channel, and the reception processes of molecules will be analyzed. The simulator will then be used to analyze specific performance metrics in a diffusion-based molecular communication network. The simulator will also provide a three-dimensional visualization of the network.

Keywords: Nanonetworks, Molecular Communication, Diffusion, Simulation

ÖZ

Nano-ağlar son yıllarda, nanoteknolojideki ilerlemeler ve biyomedikal, endüstriyel, çevresel ve askeri alanlardaki geniş uygulamalarından dolayı dikkat çekmiştir. Nano-ölçekte bileşenleri olan nano-aygıtlar, kendi başlarına sadece basit işlemler, algılamalar ve eyleyim yapabilirler. Nano-ağlar, nano-aygıtların birbiriyle bağlanmasıyla oluşur. Nano-ağlarda, nano-aygıtlar daha karmaşık işlemler için işbirliği ve bilgi paylaşımı yapabilirler. Nanomakineler arasında iletişim için kullanılan metodlardan biri biyo-uyumlu moleküllerin difüzyonudur. Bu metod, elektromagnetik dalgaların kullanıldığı alışılmış iletişim tekniklerinin aksine, bilgi taşıyıcı olarak molekülleri kullanır. Gönderici bir nanomakine, bilgiyi moleküllere kodlayıp sıvı bir ortama bırakır. Moleküller Fick kanununa göre yayılıp bir alıcı nanomakine tarafından alınır. Bu çalışmada, nano iletişim ağları için bir benzetim çerçevesi geliştirilmiştir. Özellikle, moleküllerin iletim, yayılım ve alış süreçleri analiz edilmiştir. Benzetici ile difüzyona dayalı moleküler iletişimin performans ölçütleri değerlendirilmiştir. Benzetici ayrıca, nano-ağın üç boyutlu görselleştirmesinde de kullanılabilmektedir.

Anahtar kelimeler: Nano-ağlar, Moleküler iletişim, Difüzyon, Benzetim

TO MY FAMILY,
for their unconditional love

ACKNOWLEDGMENT

First, I have to thank God who has always paid attention to my needs and never left me alone throughout my life.

I would like also to take particular note of the people who made this research possible with their great help and support.

I would like to extend my gratitude to my supervisor, Assoc. Prof. Dr. Dogu Arifler, for showing me the right directions and helping me with all my questions.

Finally, I would like to dedicate this thesis to my family who encouraged me to broaden the horizons of my imagination and supported me through my education. I hope that dedicating this thesis to them, even though it is not much, could return a bit of their love and kindness.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Nanotechnology

The term nanotechnology was first introduced by Taniguchi [1] as follows: "Nanotechnology mainly consists of the processing of, separation, consolidation, and deformation of materials by one atom or by one molecule." Later, this description of nanotechnology is generalized by the US National Nanotechnology Initiative program, which describes nanotechnology as the manipulation of matter with dimensions on the nanoscale, when at least one of their dimensions is scaled below 100 nanometer.

Nanotechnology can be used to create new material and devices by engineering matters at the nanometer scales. At this scale, we can consider a nanomachine as the most basic integrated functional device that can perform simple tasks like computing, communicating and sensing at the nano-level.

1.2 Nanomachines

Generally, nanomachines can be defined as devices which are made of nano-scale components and are able to perform specific tasks such as computing, communicating, data storing, sensing and actuation, at nano-level [2].

Nanomachines whether artificially built or those found in biological systems, can only perform very simple and restricted tasks due to their tiny size and low

complexity. Nevertheless, these devices can be used as building blocks of more complex systems such as nano-processors, nano-memory and nano-robots.

1.2.1 Manufacturing Nanomachines

The application range and capabilities of nanomachines tightly depend on the method in which they are manufactured. As depicted in Figure 1.1, different approaches can be used for the development of nanomachines, ranging from the reuse of existing biological entities in nature to the use of artificially made components. These approaches can be grouped into three main classes, namely, bottom-up, top-down and bio-hybrid [3]:

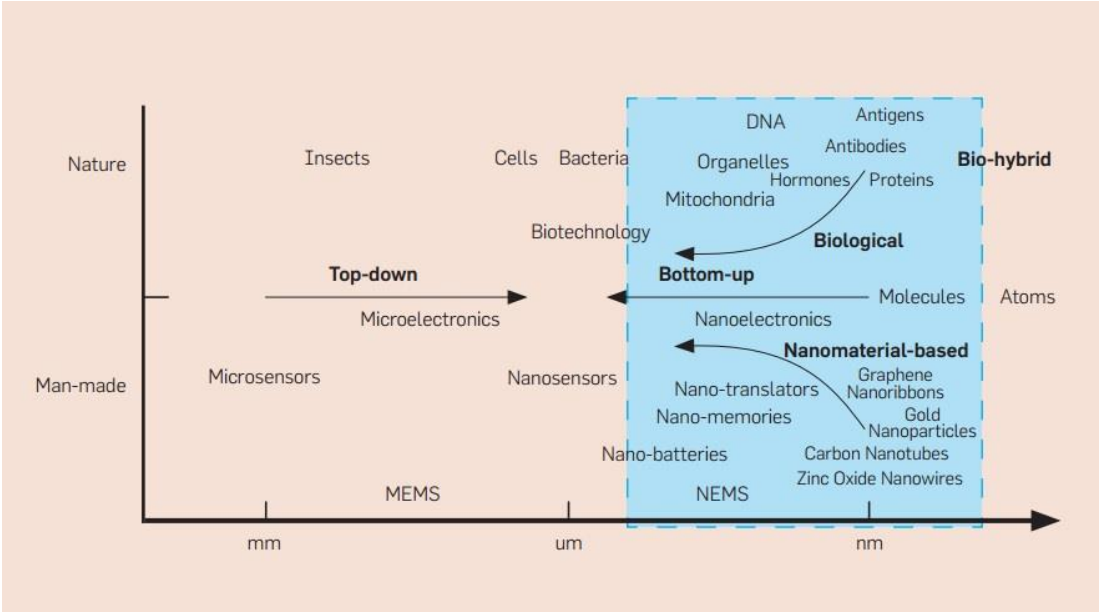

Figure 1.1: Different approaches for the manufacturing of nanomachines (Reproduced from [4])

- **Top-down**. In this approach, nanomachines are created by downscaling current device components which are in the micro-level to achieve nano-level objects.
- **Bottom-up**. This approach which is also called molecular manufacturing, uses molecules as the building blocks to assemble nanomachines precisely

[5]. However, the required technologies to build nano-structures, molecule by molecule, still does not exists.

- **Bio-hybrid**. Existing biological nanomachines such as nano-actuators and nano-biosensors, found in living organisms, can be used as a model or combined with manufactured nano-structures to create new nanomachines.

1.2.2 Nanomachine Architecture

A nanomachine, proportional to its level of complexity, consists of one or more components. Typically, architecture of a complex nanomachine will consist of the following components [3]:

1) **Control unit**. This unit executes instructions given by the intended tasks by controlling other parts of the nanomachine.
2) **Communication unit**. It comprises a transceiver, with which the information (e.g., molecules) could be transmitted and received at nanoscale.
3) **Reproduction unit**. This unit is responsible for replicating the nanomachine by fabricating all the components of it, using external sources, and then reassembling them.
4) **Power unit**. The function of this component is to power all other components of the nanomachine. Furthermore, it can absorb energy from external sources like temperature and light and store it for future consumption.
5) **Sensor and Actuators**. This unit can be considered as an interface between the nanomachine and its surrounding environment. Depending on its functionalities, nanomachines can have several sensors and actuators (e.g., chemical sensors, temperature sensors or locomotion mechanisms).

With the current technology, building such a complex nanomachines is not feasible. However, similar architectures, such as living cells exists in nature that can be used to model and develop new bio-inspired nanomachines. Figure 1.2 shows a component mapping between biological components of a living cell and a typical architecture of a nanomachine.

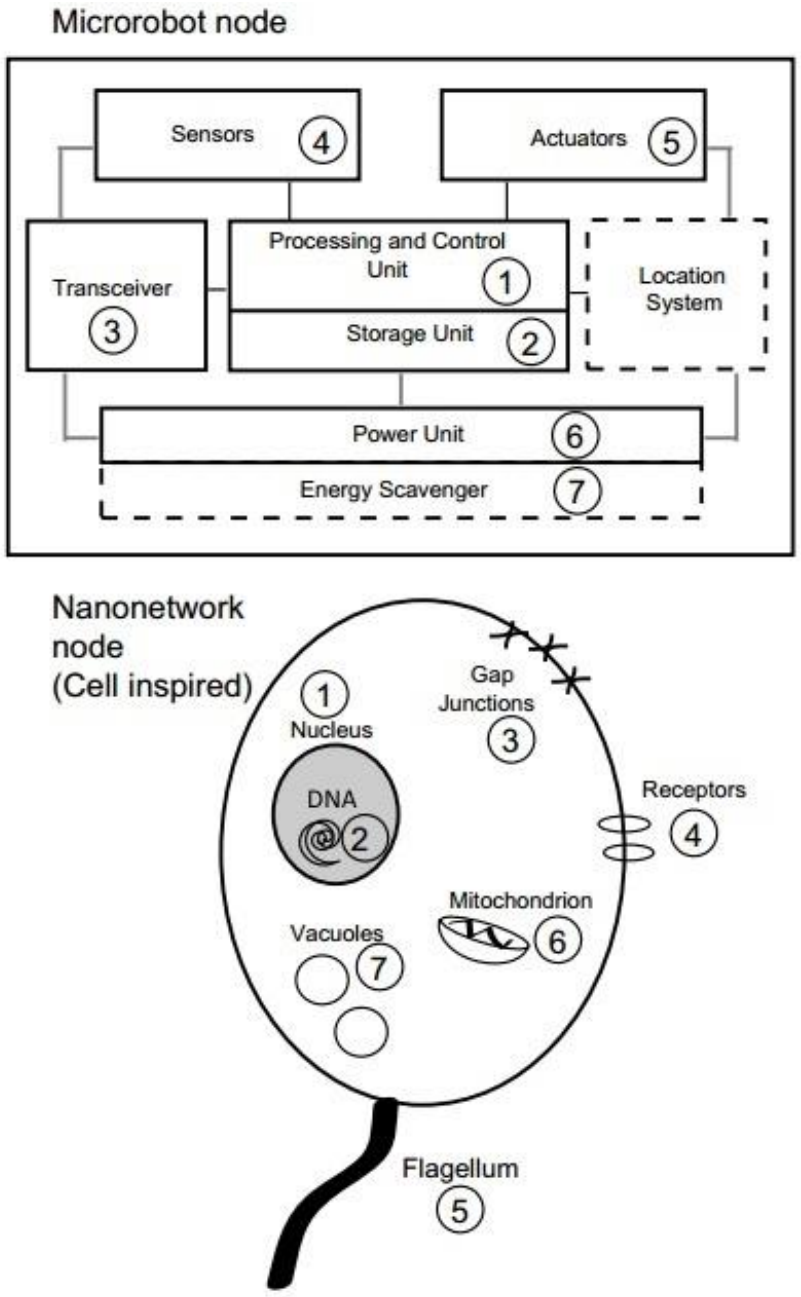

Figure 1.2: Mapping between biological components of a living cell and a typical architecture of a nanomachine (Reproduced from [3])

The following components in the cell are akin to the architecture of a nanomachine [3]:

1) **Control unit**. In the cell, the nucleus can be regarded as the control unit and it holds all the instructions necessary to understand the cell functionality.
2) **Communication unit**. The receptors and gap junctions placed on the plasma membrane, can be seen as molecular transceivers responsible for sending and receiving the molecules.
3) **Reproduction unit**. Biological entities like molecular motors and centrosome participate in the duplicating process of the cell. Each duplicated cell will have the same DNA sequence as the original cell.
4) **Power unit**. Cells can have different means for power generation, e.g. the mitochondrion that produces big portion of the chemical substances which are used as energy source in a lot of cellular processes.
5) **Sensors and actuators**. A particular cell may contain many sensors and actuators such as the flagellum which primarily is responsible for locomotion.

1.3 Nanonetworks

Nanomachines can only perform very simple and restricted tasks due to their tiny size and low complexity. However, by interconnecting nanomachines by means of nanonetworks, it is possible to considerably expand their capabilities and range of operation. The communication method among nanomachines, strongly depends on the way they are built and their characteristics. Moreover, choice of the particular method of communication is constrained by the specific application for which the nanonetwork will be implemented. Based on the limitations of the nano-level communications, mainly two communication paradigms have been proposed:

- **Terahertz band.** Considering the size range of a nanomachines, terahertz band (0.1 THz – 10 THz) with the focus on the graphene-based nano-antennas, has recently attracted the interest of scientists [6]. This frequency band can support very high transmission rates for distances up to few meters. Nevertheless, it is still not clear how it is possible for the nanomachines with their limited capabilities to exploit the features of this band.
- **Molecular communication.** This is a completely new paradigm inspired from the biological structures like cells [3]. In the molecular communication, molecules are used to encode, transmit and receive the information.

1.3.1 Applications

There are a great number of potential applications for nanonetworks which can be presented in three main categories as follows [7]:

- **Biomedical field**. Nanonetworks can be deployed over or inside the human body to monitor cholesterol and glucose levels or to identify specific types of infections or cancer. Furthermore, by leveraging the sensing capabilities of nanomachines and their ability to release the specific drugs inside the body at specified locations, implementing smart drug delivery systems would be possible.
- **Industrial field**. Nanonetworks can be used for the development of new materials and can help with quality control mechanisms. For example, nano-sensors can make detection of small bacteria and toxic components possible where is not possible by using traditional sensing technologies.
- **Surveillance**. Nano-sensors can detect aggressive and unauthorized biological and chemical particles and send the information through the nodes

in the network to the sink. They also can be used for air monitoring in a similar way as quality control applications.

1.4 Problem statement

Diffusion-based molecular communication has proved to be a promising communication paradigm for the nanonetworks due to its biocompatibility and energy efficiency. Several studies have focused on this area and various aspects of diffusion-based molecular communication have been modeled analytically by different authors. However, in order to validate these analytical models, either simulations or experimental studies are needed. Experimental setup of molecular communication at the nanoscale, despite the recent advances in nanotechnology and synthetic biology, is generally not possible. This leaves the simulation as the most reliable choice currently available. Hence, simulators can be used for validation of theoretical models and simplifying the development of new communication techniques and more accurate models. Furthermore, Monte Carlo simulations can be used when there are no close form solutions for theoretical models or in the more complex cases where analytical solutions are not feasible. In this thesis, a simulation framework for molecular nano communication networks will be developed. In particular, transmission, propagation channel, and the reception processes of molecules will be analyzed. The unique features of this simulator are its ability to model a reception process which is inspired from the ligand-receptor binding kinetics in living cells; a graphical user interface and visualization capabilities that can display simulation elements by 3-D graphics and its composite design that lets new modules easily developed and added to the simulator, based on needs of users and simulation model.

The rest of this thesis is structured as follows: in Chapter 2, molecular communication will be described. Next, Chapter 3 describes the models which are used in this work and summarizes the related work. In Chapter 4, we illustrate the simulator design and its software architecture. Simulation results and evaluation of the simulator are discussed in Chapter 5. Finally, we will conclude this thesis and preview the future work in Chapter 6.

Chapter 2

MOLECULAR COMMUNICATION

Molecular communication is a completely new paradigm inspired from the biological structures like cells and can be considered as a biocompatible alternative for the telecommunication technologies at the nano-level [3].

In this communication paradigm, molecules are considered as carrier signals to convey the information. For instance, a transmitter nanomachine encodes the message symbols into the molecular signals and then sends them into a fluidic propagation channel. These molecules then move toward the compatible receivers through the designated transport methods. In this thesis, the words “particles” and “molecules” will be used interchangeably.

2.1 Molecular Communication vs. Telecommunication

Traditional telecommunication paradigms make use of the electromagnetic waves and electrical or optical signals to achieve very fast and reliable data transfers over relatively long distances. On the other hand, molecules as the information carriers in the molecular communication move slowly and stochastically, which consequently makes this kind of communication slow and unreliable. However, molecular communication can benefit from its ability to transmit complex data, like DNA sequences, while in traditional methods this is not possible. Moreover, unlike conventional communication devices, nanomachines are responsible for producing their required energy themselves instead of relying on an outside energy source [3];

and this makes the molecular communication very energy efficient compared to the traditional telecommunication. Due to energy efficiency and biocompatibility, molecular communication can be very effective in specific applications like the biomedical field, where other communication types might not be feasible.

The existing noise in the environment can affect transmitted signal in both molecular communication and telecommunication. In the traditional networks, noise can be an undesired signal which overlaps with the information signal at the receiver. In the molecular communication, noise can be the presence of the messenger molecules from the previously transmitted signal, at the receiver sensing area. Also, when two different sources transmit identical molecules which then interfere at the receiver. Table 2.1 depicts the comparison between to communication paradigms:

Table 2.1: Comparison between Molecular Communication and Telecommunication (reproduced from [8])

	Telecommunication	Molecular communication
Information carrier	Electromagnetic waves, electrical/optical signals	Chemical signals
Media	Space, cables	Aqueous
Speed	Speed of light (3×10^8 m/s)	Extremely slow (nm$\sim$µm/s)
Range	Long distance ($\sim$km)	Short distance (nm$\sim$m)
Information	Texts, audio, videos	Chemical reactions, states
Other features	Reliable, high energy consumption	Unreliable, biocompatible energy efficient

2.2 Properties of Molecular Communication

The unique characteristics of the molecular communication stems from using biological components and mechanisms for communication in a fluidic environment. The following summarizes these characteristics [9]:

- **Use of chemical signals for encoding and decoding**. In molecular communication, physical properties as well as characteristics of information molecules, such as structure of information molecules (e.g., protein) and their concentration and type (e.g., calcium concentration), can be used to encode the information. Moreover, unlike the traditional communication methods where signals are used to encode binary data, in here, complex data can be encoded into structure of messenger molecules.
- **Limited range, high loss rate and slow speed**. Propagation of carrier molecules is very slow and happens in short ranges. It also varies based on the environment and mechanism used. For example, in the case of neural signaling, electrochemical signals with the speed of 100 m/s are used to achieve fast communications over the rage of several meters. On the other hand, in the case of the free diffusion of molecules, communication occurs in the micrometer range. In addition, due to the stochastic movement of messenger molecules, these molecules may arrive at a receiver nanomachine late, or may not even arrive, which leads to a high loss rate in the communication.
- **Biocompatibility**. Since molecular communication utilizes similar communication mechanisms as biological systems, man-made nano machines might be able to directly communicate with natural entities inside a biological system. This biocompatibility then can be exploited in the specific applications like medical field, where using a biologically friendly nanomachine is mandatory.
- **Energy efficiency**. Since molecular communication follows mechanisms used in biological systems, it is expected to be very energy efficient. For

example, molecular motors (e.g., myosin) are known to be very efficient in converting chemical energy to mechanical work. The needed energy (e.g., glucose) is expected to be found in the surrounding environment where nanomachines are deployed.

2.3 Categorization Based on Propagation Model

Molecule-based communication is a very common communication method among biological organisms. These biological nanomachines use different mechanisms for inter/intra cell molecular communication which can be categorized into passive-transport and active-transport molecular communication, based on how signal molecules propagate within the medium [8].

2.3.1 Passive-Transport Molecular Communication

In passive transport, messenger molecules propagate randomly in all possible directions by means of diffusion and without consuming chemical energy. This type of transport is particularly appropriate for dynamic and unpredictable environments. Furthermore, it is suitable for situations in which having an infrastructure for the communication is not feasible. However, because of the unpredictable movement of particles in the passive transport, in order to increase the chance for messenger molecules to arrive at a distant destinations, large number of them are required to be emitted from the source. Passive transport is also relatively slow due to the fact that squared displacement of particles is proportional to the elapsed diffusion time and this displacement can happen in all directions, randomly.

In the following, different cases of passive transport molecular communication found in living organisms are described:

- **Free diffusion-based molecular communication**. In this mechanism, signal molecules (e.g., proteins) are released by source cells into the environment in which they propagate randomly in all available directions based on laws of diffusion. These molecules then can be captured by receiver cells through their protein receptors which results in activation of a ligand-receptor chemical reaction. One example of this mode of molecular communication is quorum sensing, a prevalent communication mechanism used by bacteria to coordinate specific behaviors such as bioluminescence generation and biofilm formation. Quorum sensing makes sure that the concentration of the molecular signals in the surrounding medium passes a threshold before cells respond to them.
- **Gap junction mediated diffusion-based molecular communication**. In this mode, signals diffuse from one cell to an adjacent cell through physical channels called gap junction as shown in Figure 2.2.a. Using gap junction channels for communication enables coordinated actions between connected adjacent cells.
- **Reaction diffusion-based molecular communication**. In this mode of communication, diffusion of first messenger molecules can involve in biochemical reactions at the receiver which result in sudden increase and decrease in the concentration of secondary messenger molecules inside the receiver cell which consequently leads to an impulse of the signal molecules that propagates in the medium. For instance, the endoplasmic reticulum (ER) in a cell acts as a reservoir for the calcium ions (Ca^{2+}) and when a cell is stimulated, it releases the calcium ions stored in the ER and the sudden

increase in the calcium ions concentration forms an impulse that diffuses to adjacent cells as shown in Figure 2.2.b.

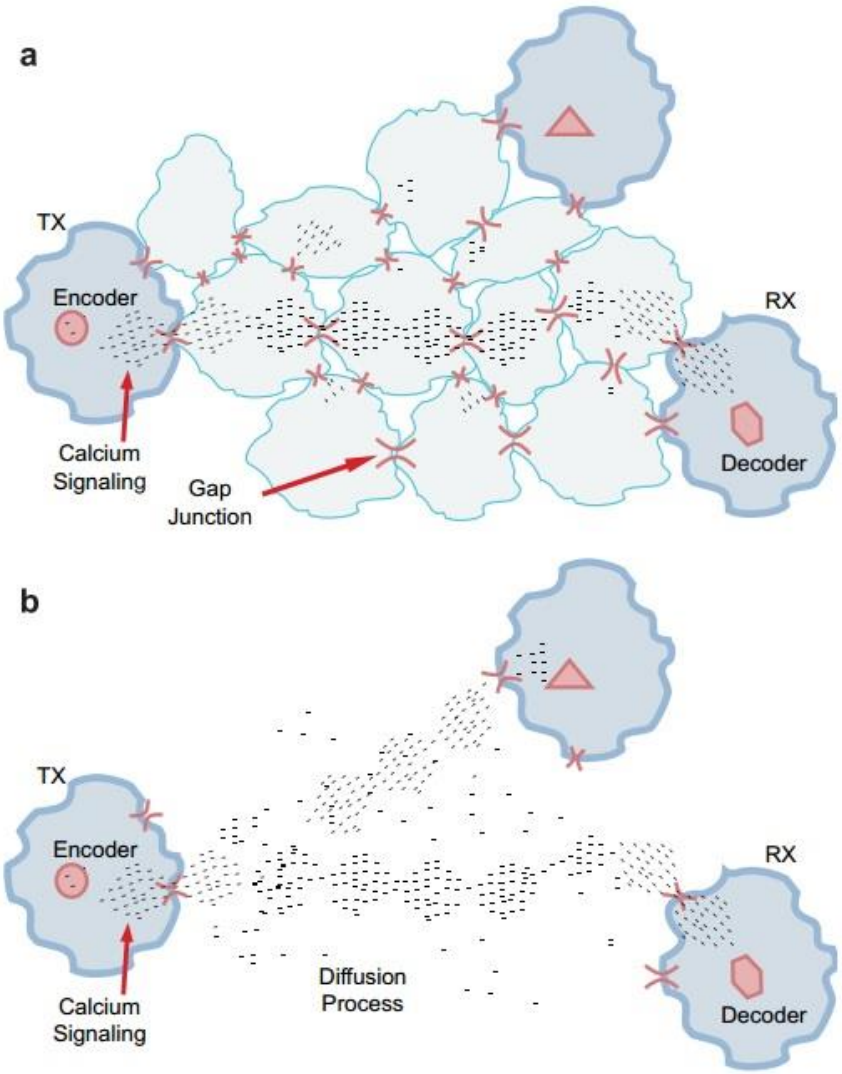

Figure 2.1: Signal propagation in diffusion-based molecular communication by (a) gap junctions signal forwarding and (b) by reaction diffusion-based molecular communication (Reproduced from [3]).

2.3.2 Active-Transport Molecular Communication

In this communication mechanism, signal molecules are navigated directly to the specific destination. Active transport uses chemical energy to generate adequate force in order to transport larger signal molecules and to longer distances compared to passive transport. Since signal molecules are directed toward the destination in an

active transport, molecules have higher chances reaching the intended destination compared to passive transport. Hence, active transport can be considered more reliable than passive transport given fewer signal molecules. However, in order to maintain the transport, active transport requires an appropriate infrastructure such as molecular motors, microtubules and vesicles. Moreover, active transport of signal molecules requires a sustained supply of energy to successfully deliver molecules to the destination. Two examples of active-transport molecular communication in biological systems are molecular motor-based and bacterial motor-based molecular communications.

2.4 Categorization Based on Communication Range

Based on the effective range of communication, molecular communication can be categorized into three groups [3, 10]:

- **Short-range communication**. This includes communication ranges between nanometer and micrometer. Generally, passive transport methods like calcium signaling or molecular motors for intracellular communication have been proposed for this range.
- **Medium-range communication**. This includes communication ranges between micrometer and millimeter. Communication techniques such as catalytic nano-motors and flagellated bacteria have been proposed for this range [10].
- **Long-range communication.** This includes communication ranges between millimeter and meter. Generally, active transport methods like neuron-based communication techniques and pheromones have been suggested for this range [11].

2.5 Diffusion-based Molecular communication

Diffusion-based molecular communication, because of its independency from infrastructure and energy source is a simple and flexible molecular communication mechanism. Due to these properties, this study will consider diffusion-based molecular communication as the communication mechanism between nanomachines. Figure 2.2 depicts components of a typical diffusion-based molecular communication between two nodes.

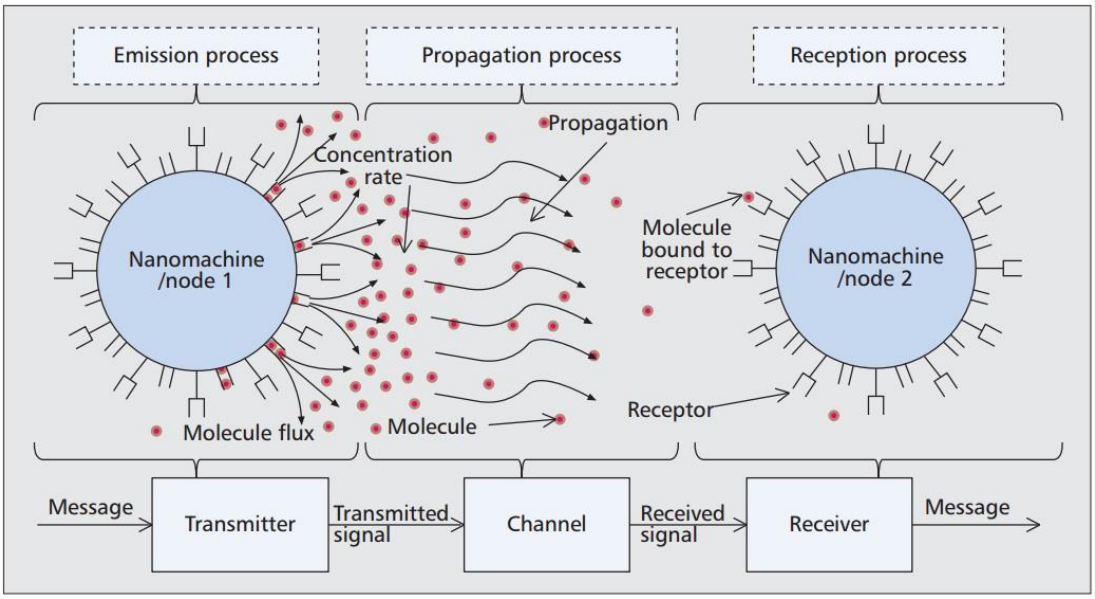

Figure 2.2: Diffusion-based molecular communication process with two nodes (Reproduced from [7])

Main components of this communication mechanism are: emission process, propagation and reception process of information molecules. These processes are explained in the following sections.

2.5.1 Emission Process

Emission process mainly consists of encoding the message and sending it toward the destination. Encoding phase is about translating the information into molecular signals. A molecular signal can be the variation in structure, type or concentration of

the signal molecules. Three different modulation schemes can be considered to encode the message into a molecular signal [12]:

- **Concentration Shift Keying (CSK)**. This technique is similar to pulse-amplitude modulation in traditional networks. Concentration shift keying can be used to encode N-bit messages by 2^N different levels of concentration of released molecules [13]. For instance, to encode one bit of information, sender can use two different amount of released molecules depending on the bit value. Receiver then can decode the information according to the received concentration level.
- **Pulse Position Modulation (PPM)**. PPM can be used to encode N-bit of information by transmitting each pulse of molecules in one of 2^N possible time slots [14]. That is, the sender transmits the molecules in the beginning of one of possible time slots and the receiver then can decode the information by knowing the time slot in which the molecules were released.
- **Molecule Shift Keying (MoSK)**. In molecule shift keying, N-bit of information can be encoded by 2^N different types of released molecules [13]. That is, the transmitter, based on the symbol it intends to transmit, releases certain amount of one of different types of molecules available. The receiver then can decode the information based on the type and concentration of received signal during a time slot.

After encoding the message, the sender releases the signal molecules into the surrounding medium (e.g., by opening its gap junction channels deployed in its cell membrane).

2.5.2 Propagation Process

In diffusion-based molecular communication, after information molecules are released by the sender into the medium, they propagate by means of diffusion from the regions of high concentration to the regions of lower concentration. According to [15], molecular diffusion can be categorized into following groups:

- **Normal diffusion**. In normal diffusion, interaction among released molecules, such as their collision with each other and chemical reactions and electrostatic forces among them can be neglected. With this assumption movement of particles are independent of each other and the diffusion of particles can be modeled by Brownian motion using Fick's laws of diffusion.
- **Anomalous diffusion**. In situations where fluid is not in equilibrium or when concentration of transmitted molecules is very high and collision among particles cannot be neglected, anomalous diffusion happens. In this case, correlated random walk can be used to model the diffusion.

In this thesis, we assume that medium is in equilibrium or very close to equilibrium and concentration of released particles is much lower than density of the medium. Hence, the movement of different particles are independent of each other and can be modeled by using the normal diffusion.

2.5.3 Reception Process

Reception process happens when receiver senses the transmitted molecules in its sensing area. Surface of a receiver cell is usually covered by chemical receptors (e.g., protein receptors) which can sense and react to the signal molecules. When a signal molecule binds to a receptor, a chemical reaction occurs. These chemical reactions can be used to decode the transmitted messages at the receiver.

Chapter 3

MODELING MOLECULAR COMMUNICATION PROCESSES

3.1 Related Work

As mentioned previously, diffusion-based molecular communication (DMC) is considered as a promising paradigm and the most practical for communication between nanomachines. Several studies have focused on this area and various aspects of DMC have been modeled analytically by different authors. In [13], based on the unique properties of this communication paradigm, authors proposed two modulation techniques, Concentration Shift Keying (CSK) and Molecule Shift Keying (MoSK). In [16, 17], the main characteristics of diffusion-based propagation channel are explored. In another contribution [18], achievable information rates for nano-communication channel are investigated by comparing simulation results to the results obtained from analytical channel models. However, the noise sources in DMC are analyzed for the first time in [19]; two sources of noise are introduced as particle sampling noise and particle counting noise. Particle sampling noise comes from the difference between the ideal particle concentration rate, which samples the information signal, and the actual transmitted concentration rate and it happens due to the discreetness of the particles that compose the transmitted signal. Particle counting noise accounts for the noise in the process of reception due to the influence of diffusion of particles on the measurement of the concentration at the receiver. In [20, 21], characterizing and modeling the reception process at the receiver

nanomachine, based on kinetics of reaction in ligand-receptor binding, has been described by the authors.

These works were mostly based on analytical methods and therefore simulators can be used for validation of these models and simplifying the development of new communication techniques and more accurate models. Furthermore, Monte Carlo simulations can be used when there are no close form solutions for theoretical models or in the more complex cases where analytical analysis is not feasible. In the following, an overview of relevant simulators to the field of molecular communication is given.

Molecular dynamics simulators can be used to accurately define models of molecules and atoms and simulate their movement and atomic interactions within the biomolecular systems. They are usually scalable to run on high-end parallel platforms with hundreds of processor cores. Some well-known simulators include: NAMD [22], a parallel molecular dynamics simulator designed for high performance simulation of large biomolecular systems; GROMACS [23], is another parallel molecular dynamics simulator that can utilize both CPU and GPU cores to simulate the Newtonian equations of motion for systems with hundreds to billions of particles and LAMMPS [24], a large-scale atomic and molecular dynamics simulator which can run on single processors or in parallel using message-passing techniques.

Despite high performance and various capabilities of these simulators, they have not been designed for diffusion-based molecular communication networks. As a consequence, they cannot be used to measure important parameters in communication networks such as throughput, signal amplitude, delay and noise.

Hence, some new simulation frameworks have been developed by different authors specifically for DMC.

NanoNS [25] is based on the famous ns-2 simulator and can simulate three-dimensional molecular diffusion based on Fick's laws of motion. It partitions the propagation medium into lattice sides and flux of particles happens between adjacent volume cells. Furthermore, the reception process can be modeled as one of the three models: NoReaction, Gillespie [26] or Berg [27]. However, NanoNS framework depends on the ns-2 simulator and cannot be considered as a stand-alone simulator.

N3Sim [28] is another DMC simulator which models both the cases of normal and anomalous diffusion and includes a harvesting mechanism which can collect messenger molecules from the medium and store them into a molecule reservoir for use in future transmissions. Moreover, some number of predefined waveforms are included for the emitter nodes that can be used to shape emission of particles. However, authors did not justify their model for anomalous diffusion and how it is possible to integrate inertia forces and collision among particles with normal diffusion equation of Brownian motion. Furthermore, this simulator does not model the reception process at the receiver nodes.

Authors in [29] introduced a DMC simulator that can be used to simulate Brownian molecular diffusion under multiple boundary conditions. In this simulator nano objects can be modeled as both mobile and static nodes. Furthermore, as a reception model, receptors are distributed over surface of receiver nodes and reception happens if messenger molecule collides with one of the carrier-compliant receptors. However

reception process does not take into account the interaction between the receptor and particles and assumes that binding event is always successful.

The contribution of this thesis is to present a simulator for the diffusion-based molecular communication networks. The unique features of this simulator are its ability to model a reception process which is inspired from the ligand-receptor binding kinetics in living cells; a graphical user interface and visualization capabilities that can display simulation elements by 3-D graphics and its composite design that lets new modules easily developed and added to the simulator based on needs of users and simulation model.

3.2 Emission Process Model

In the emission process, first information is encoded into molecular signals then, transmitter releases the signal molecules into the surrounding medium (e.g., by opening its gap junction channels deployed in its cell membrane). In this thesis, transmitter nanomachines are modeled as the ideal transmitters (i.e., transmitter does not affect the propagation of the released molecules in the medium). Moreover, molecules can be released from a point source or a spherical source. In the former case, all the molecules will be released to a specified point that denotes the location of the transmitter in the simulation. In the latter case, molecules will be released randomly inside a sphere centered at the transmitter location in the simulation. In order to model a gap junction channel in a cell, a transmitter can be used in conjunction with a spherical node in the simulation, as depicted in the Figure 3.1:

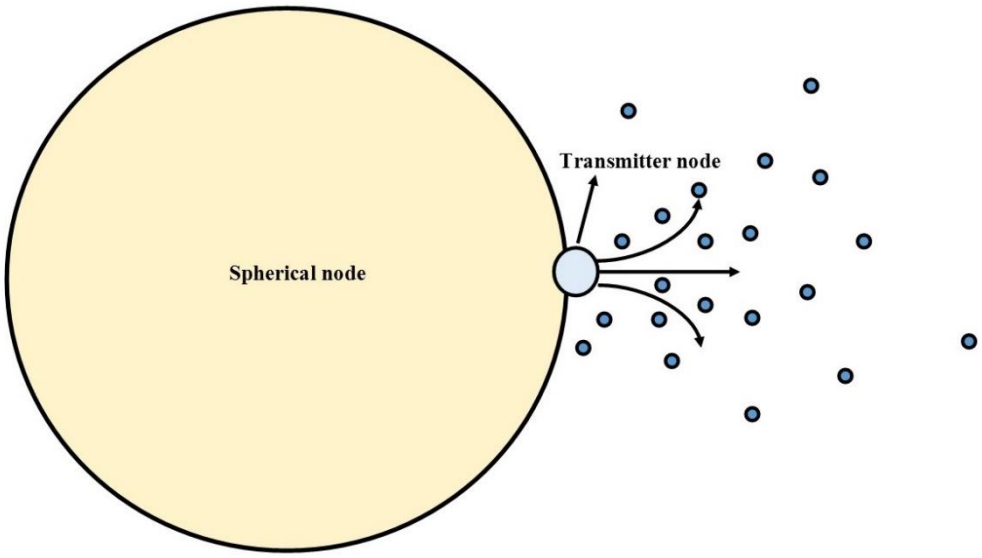

Figure 3.1: Modeling a gap junction channel in a cell with a transmitter node and a spherical node

3.3 Normal Diffusion Model

In this work, we assume that the concentration of transmitted particles is much lower than the concentration of the propagation medium molecules and also medium is in equilibrium state. Hence, the interactions among carrier molecules, i.e. collisions and electrostatic forces, can be neglected and movement of each particle is independent of other particles and diffusion process follows Brownian motion and Fick's laws of diffusion [15, 12].

The propagation of molecular signal based on above assumptions can be modeled using the Wiener process [30, 12]. In three dimensions, the position of the Wiener process *W(t)* is given by:

$$\mathrm{W(t)} = \left(W_x(\mathrm{t}), W_y(\mathrm{t}), W_z(\mathrm{t})\right) \tag{3.1}$$

where $W_x(t)$, $W_y(t)$, and $W_z(t)$ are independent and identically distributed (i.i.d) random processes and represent the *x*, *y*, and *z* components of the molecule position at time *t*, respectively. Changes in these random processes in each interval of *dt* seconds are given by:

$$W_x(t + dt) = W_x(t) + \sqrt{2\mathrm{Ddt}} * N(0,1) + v_x dt \quad (3.2)$$

$$W_y(t + dt) = W_y(t) + \sqrt{2\mathrm{Ddt}} * N(0,1) + v_y dt \quad (3.3)$$

$$W_z(t + dt) = W_z(t) + \sqrt{2\mathrm{Ddt}} * N(0,1) + v_z dt \quad (3.4)$$

where *N(0,1)* is a random number generated from standard normal distribution and v_x, v_y and v_z are the *x*, *y*, and *z* components of the drift velocity and *D* is diffusion coefficient of the molecule propagating in the given medium and its value in a fluidic medium is given by:

$$D = \frac{KT}{6\pi\eta r_m} \quad (3.5)$$

where $k = 1.38 \cdot 10^{-23}$ J/K is the Boltzmann constant, *T* is the temperature (in *K*), η is the dynamic viscosity of the fluid, and r_m is the radius of the diffusing molecule.

Position of each particle at each time step in the simulation is cumulative sum of its previous displacements calculated from equations (3.2) - (3.4).

3.3.1 Behavior at Boundaries

Diffusion simulation should check if particles encounter any boundaries; the first case is when particles collides with the surface of a receiver node which is not in the

transparent mode (i.e. particles can freely propagate through receiver without encountering any collision or resistance). The second case is when particles hit the reflective surface of the simulation bounds if a bounded space is chosen for the simulation. Independent of which case that might happen, three steps are needed at each time step to manage the collisions in the simulation:

1- Check if collision happened
2- Calculate the point of collision
3- Calculate the reflection vector

Each receiver and boundary object in the simulator is responsible for managing collision of particles with its surface. In the following sections, details of the collision management for the receiver and boundary nodes are given.

3.3.1.1 Collisions with Receiver Node

Each receiver node in the simulation, at each time step checks to detect if any particle collides with its surface. This is done by checking distance of the particle to the center of the receiver which assumed to have a spherical shape, inspired from living cells. If this distance is less than receiver radius then collision is detected. Next step is to calculate point of the collision (P) as is illustrated in the Figure 3.2:

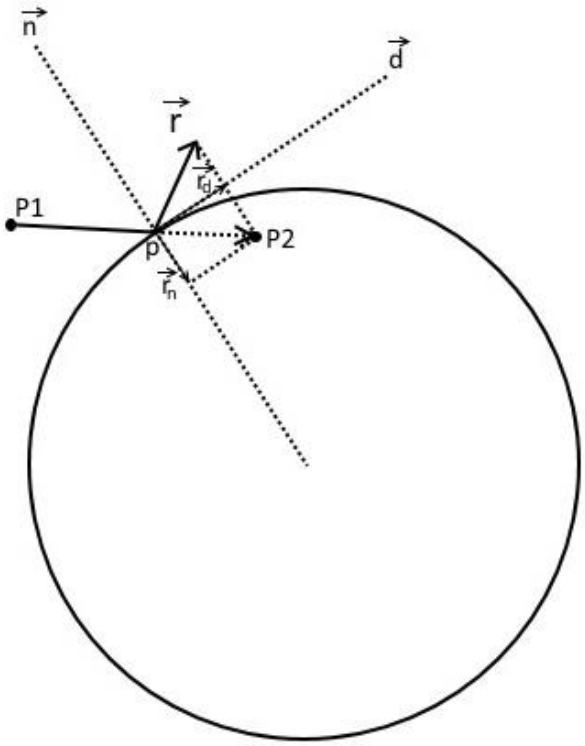

Figure 3.2: Collision of a particle to the receiver node

P_1 (x_1, y_1, z_1) is the previous location of the particle and $P_2(x_2, y_2, z_2)$ is the current location if particle did not collide with the receiver. The vector passing through P_1 and P_2 in the space can be represented as:

$$(P_2 - P_1)t + P_1 = \big((x_2 - x_1)t + x_1\big)\hat{\imath} + \big((y_2 - y_1)t + y_1\big)\hat{\jmath} + ((z_2 - z_1)t + z_1)\hat{k} \quad (3.6)$$

The collision point (P) is located on this vector and on the surface of the receiver. Hence, it should satisfy the following equation:

$$\big((x_2 - x_1)t + x_1\big)^2 + \big((y_2 - y_1)t + y_1\big)^2 + ((z_2 - z_1)t + z_1)^2 = r^2 \quad (3.7)$$

Solving equation (3.7) for *t* gives two points which only one of them resides in between P_1 and P_2 and is the valid collision point (*P*).

Having the collision point (*P*), the next step is to calculate reflection vector ***r***. In order to find ***r*** we need to calculate unit vectors $\hat{n}$ which is normal to the surface of the receiver and $\hat{d}$ which is tangent to the surface. If $\vec{C}$ represents location vector of the receiver's center, then $\hat{n}$ can be easily calculated as follows:

$$\hat{n} = \frac{(\vec{P}-\vec{C})}{\|\vec{P}-\vec{C}\|} \tag{3.8}$$

Now, if we name direction vector of the particle by $\overrightarrow{V} = (\vec{P}_2 - \vec{P})$, then projection of $\vec{V}$ in the $\hat{n}$ direction can be calculated as:

$$\overrightarrow{r_n} = (\vec{V}.\hat{n})\hat{n} \tag{3.9}$$

Similarly, projection of $\vec{V}$ in the $\hat{d}$ direction can be calculated as:

$$\overrightarrow{r_d} = (\vec{V} - \overrightarrow{r_n}) \tag{3.10}$$

Finally, the reflection vector $\vec{r}$ is calculated from (see Appendix A):

$$\vec{r} = \overrightarrow{r_d} - e\overrightarrow{r_n} \tag{3.11}$$

where *e* is the coefficient of restitution and ranges from 0 for completely inelastic collision to 1 for elastic collision [31]. The final position of the particle after the collision will be $\overrightarrow{P} + \vec{r}$.

3.3.1.2 Collisions with the Boundary of the Environment

The default supported shape for the simulation space bounds is the box shape, although different boundary shapes can be developed based on needs of the simulation scenario. Collision in this case happens when a particle moves outside of the defined bounds of the simulation. In order to find the collision point (*P*), like the previous case, we find intersection of the line that passes through P_1 and P_2 and the edges of the simulation bounds. Since this line hits more than one edge of the box, the valid collision point (*P*) will be the point that resides in between two points P_1 and P_2 and is closest to the point P_1. The calculation of the reflection vector is also similar to the receiver case with one exception that is depicted in Figure 3.3:

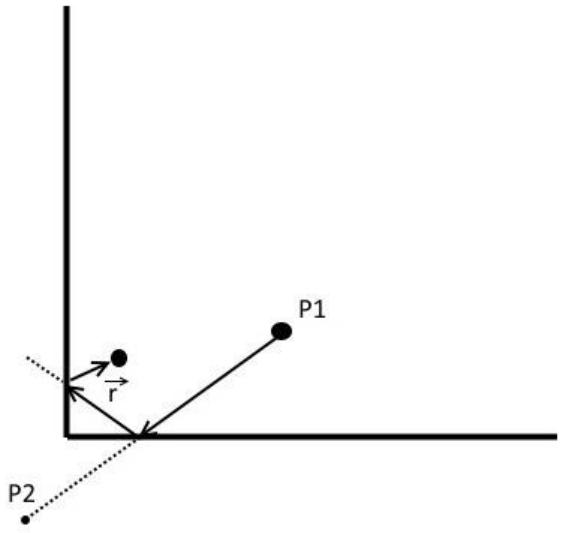

Figure 3.3: Collision of the particle to the simulation boundary

This special case happens when reflected particle collides for the second time with the simulation bound. Special care is needed in the simulator code to detect this extra collision and reflect the particle for the second time.

3.4 Reception Process Model

In order to properly model the reception process at the receiver node, we need a precise understanding of the kinetics of reaction between ligands and receptors in living cells.

In biological chemistry, receptors are transmembrane proteins placed in the cell membrane of cells in living organisms and ligands are compatible molecules that can bind to a receptor protein [20]. Surface of a receiver cell is usually covered by these chemical receptors which can sense and react to the messenger molecules. In this work, we assume that these receptors are homogeneously placed on the surface of receivers and each of them will only bind to the compatible carrier molecules. When a ligand binds to a receptor, signal transduction occurs. Therefore, the amplitude of the received signal at the receiver is proportional to the number of bound chemical receptors [21]. Figure 3.4 shows a simple case of binding of ligand to receptor.

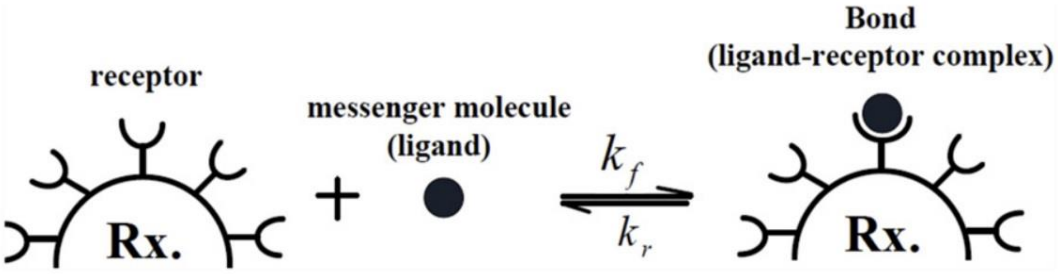

Figure 3.4: Schematic of a simple ligand–receptor binding (reproduced from [21])

In order to measure the number of receptor/ligand bonds, the deterministic reaction rate equation can be used as follows [32]:

$$\frac{dn_b(t)}{dt} = k_f C_R(t)\big(N_R - n_b(t)\big) - k_r n_b(t) \tag{3.12}$$

where $n_b(t)$ is the number of bound receptors at time t, k_f is the rate of particle binding, k_r is the rate of particle release, $C_R(t)$ is the particle concentration at the receiver and N_R is the total number of chemical receptors at the receptors area on the receiver. Particle binding rate (k_f) can be expressed as [20]:

$$k_f = Ze^{\frac{-E_a}{K_B T}} \tag{3.13}$$

where Z denotes average collision frequency and the number of collisions that happen between an unbound receptor and a particle; E_a is the activation energy and means a minimum kinetic energy a particle must have to start a chemical reaction with the collided receptor; K_B is the Boltzmann constant, T is the temperature (in K) and $e^{\frac{-E_a}{K_B T}}$ is the probability that any given collision has a higher energy than the activation energy and results in a reaction. Particle release rate (k_r) depends on physical characteristics of the propagation medium and the ligand–receptor bond and is assumed to be constant during the simulation.

Based on above mentioned facts about the ligand-receptor binding kinetics, the following assumptions are made in order to simulate the reception process [20]:

1- Receptors are uniformly distributed over one or more areas (Receptors Area) on the receiver surface which is the intersection of a cone and surface of a spherical receiver node as illustrated in Figure 3.5:

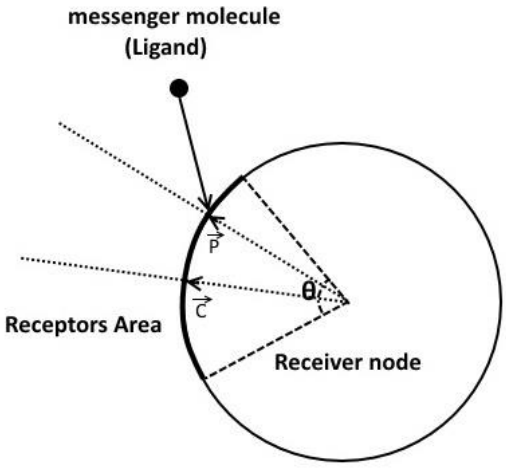

Figure 3.5: Illustration of the receptors area on a receiver node

2- The binding reaction can happen when a particle collides with a free receptor on the Receptors Area and orientation of the collision does not matter.

3- The probability (P_h,) that a particle hits an unbound receptor given that it collides the Receptors Area on a receiver, is defined as the fraction of area covered by the unbound receptors to the total area of the Receptors Area. In order to accurately calculate P_h, a similar equation in [21] is modified as follows to account for the effect of particle radius on the collision cross section:

$$P_h(t) = \frac{(N_R - n_b(t))\pi(r_{rcp}+r_p)^2}{2\pi r_{rec}^2(1-\cos(\frac{\theta}{2}))} \quad (3.14)$$

where r_{rcp} is the radius of a receptor, r_p is the radius of the particle, r_{rec} is the radius of a receiver nanomachine and θ is the aperture of a cone that defines the Receptors Area on a receiver (Figure 3.5).

4- The binding reaction between a receptor and a colliding messenger molecule happens only if the messenger molecule has a higher kinetic energy than the required activation energy. The kinetic energy of a molecule P at the time step t_n, $E_k^p(t_n)$, is given as follows [20]:

$$E_k^p(t_n) = \frac{1}{2} m_p \left| V_p(t_n) \right|^2 \tag{3.15}$$

where m_p is the particle mass and $V_p(t_n)$ is its velocity at time step t_n in the simulation. However, since it is not possible to measure instantaneous velocity of a particle subject to Brownian motion, we use (3.13) to define P_r as intrinsic probability of reaction upon a given collision between an unbound receptor and a particle:

$$P_r = e^{\frac{-E_a}{K_B T}} \tag{3.16}$$

5- $n_b(t)$ is linear in the duration of a simulation time step. Hence, discretized $k_r n_b(t)$ can be expressed as:

$$\begin{aligned} k_r n_b(t_n) &= k_r \int_0^{\Delta t} n_{b_0} + \frac{\Delta n_b(t_n)}{\Delta t} t dt \\ &= k_r (n_{b_0} \Delta t + \frac{\Delta n_b(t_n) \Delta t}{2}) \end{aligned} \tag{3.17}$$

where n_{b_0} is number of bound receptors at the beginning of time step t_n, $\Delta n_b(t_n)$ is the change to the number of bonds during the time step t_n and Δt is the duration of each time step in the simulation.

At each simulation time step, collision of a particle to the receptors area of a receiver node is detected if the angle between vectors $\vec{P}$ (passes through the collision point and center of the receiver node) and $\vec{C}$ (passes through the center of the receiver node and the center of the receptors area) is less than half of the cone's aperture θ as illustrated in Figure 3.4. If a collision detected then corresponding particle will hit an

unbound receptor with probability $P_h(t_n)$ and if it successfully hits the receptor then particle will bind to the receptor with probability P_r and the $n_b(t_n)$ will be increased. Finally, after all particles are checked for the contribution to $n_b(t_n)$, then number of released particles will be deducted from $n_b(t_n)$ based on (3.17) to calculate the final value of the number of receptor/ligand bonds in the current time step.

Chapter 4

SIMULATOR DESIGN AND IMPLEMENTATION

4.1 Design Criteria

Nano communication networking is a relatively new field of research that attracted many scientists recently. Up to the time of this writing, there are not any functional nanomachine constructed nor any real implementation of nanonetworks yet reported. Moreover, researchers are still working on different communication paradigms that can be applicable in the nano-scaled environments; even for the molecular communication paradigm, many different methods for modeling the propagation channel, modulation etc. have been proposed.

Hence, based on above mentioned factors, the objective of this thesis is not to only develop a single-purpose diffusion based simulator, But also to develop a platform that can be easily extended and modified to meet the desired functionalities that can be changed over the time based on the model needs. Therefore, **modularity** and **extensibility** are the main criteria used in the design and implementation of this simulator.

4.2 Composite Design

Designing applications in a *monolithic* style can lead to an application that is very hard to maintain or extend. In this context, “monolithi” refers to a software that consists of components which are tightly coupled together and there is no clear separation of concerns and functionality between them. It is very hard to add new

features or replace existing features in the applications that are designed and built in this way without breaking other portions of the system.

An effective solution for these problems is to partition the application into a number of semi-independent, loosely coupled components that can then be integrated into an application *shell*. The "shell" acts as the host for the application components and defines the overall layout and structure of the application, while it is generally unaware of the exact components that it will host. The shell also defines the top-level visual element of the software that will then host the different user interfaces provided by the loaded application modules. This solution provides many merits including the following:

- **Extensibility**. New functionalities or components can be easily integrated into the application or replaced with alternative implementations at run time.
- **Flexibility**. Current parts can be easily modified and updated independently and without affecting other parts due to the loosely-coupled components. Having loosely-coupled components are possible by using the Dependency Injection (DI) pattern [33]. It means each component can use services offered by other components without having a direct reference to them and dependencies between them are fulfilled at runtime by the dependency injection method.
- **Reuse**. The fact that each module can be easily developed and tested individually promotes the reuse of the modules across applications.

To achieve mentioned goals and implement these patterns, this work utilizes the Microsoft Prism Library [34] which targets Microsoft .Net framework and includes

guidance and necessary tools to accomplish this task. This simulator can be run on any PC that runs Windows 7 or newer version of the Microsoft operating system with at least one gigabyte of free RAM. A quick start guide that gives the reader the minimum instructions to run the simulator is provided in Appendix B.

4.3 Simulator Architecture

Following the previously mentioned design patterns, this simulator is composed of a main component which provides a shell for three other module components. The main component also creates a dependency injection container which loads the application modules and fulfills dependencies between them. Moreover, it creates the top-level window which defines regions that other modules then can load their user interfaces into.

The overall layout of the simulator top-level window design and its implementation are shown in Figure 4.1 and 4.2 respectively.

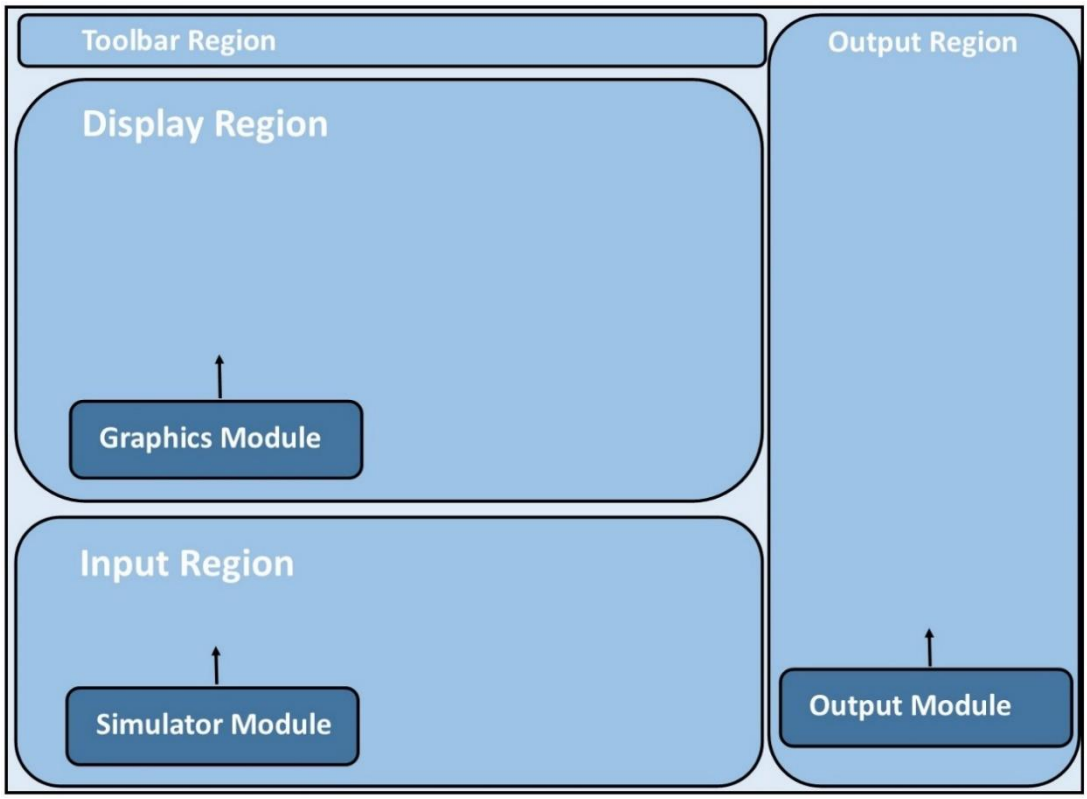

Figure 4.1: Layout of the Top-level window of the simulator user interface

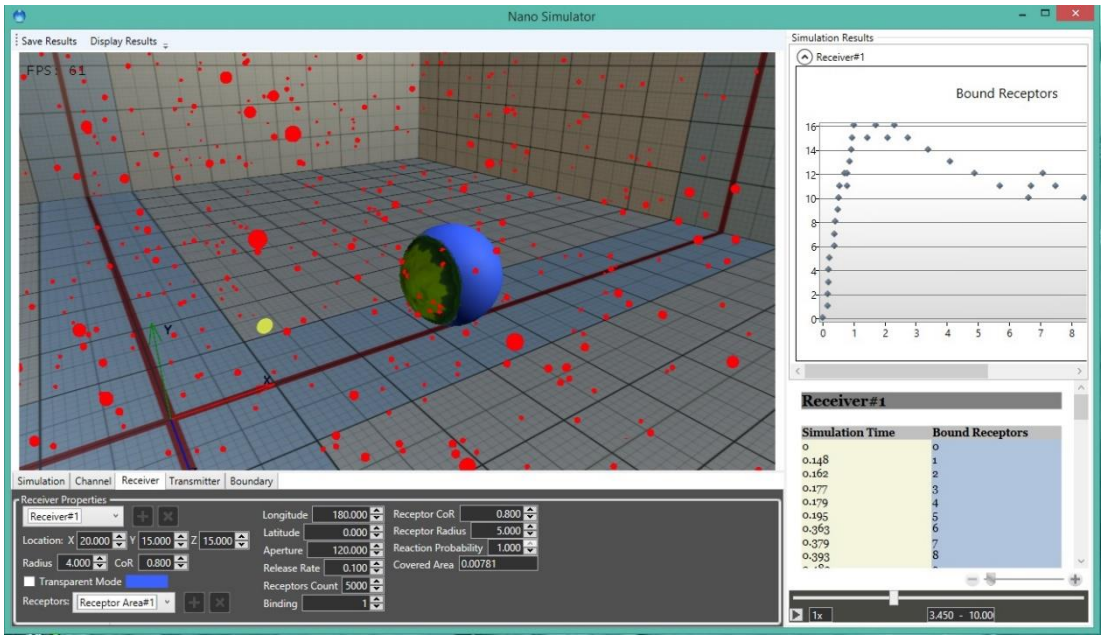

Figure 4.2: Final view of the simulator user interface when modules load their user interfaces into the top-level window

In the following sections, functionality and purpose of each three module components will be explained.

4.3.1 Graphics Module

This module invokes the Digital Rune Game Engine [35] which is based on Microsoft XNA framework [36], to provide visualization and 3D graphic rendering support for the rest of the application modules.

After being loaded into the program, this module loads its graphic screen into the "Display Region" of the application top-level window. Also, it provides a simple interface for other modules which they can use to render 2D/3D graphics to the screen. Although rendering capability of game engines are vast, for the sake of simplicity of use and based on needs of a typical simulator application, this module defines simple methods to render different geometry shapes and texts into the screen. Moreover, this module implements a particle system which gets the position of particles in each frame as an input and creates the animation of particle movements. This animation then can be played, starting from any given frame number by the user. A sample screen that demonstrates some of this module capabilities is depicted in Figure 4.3.

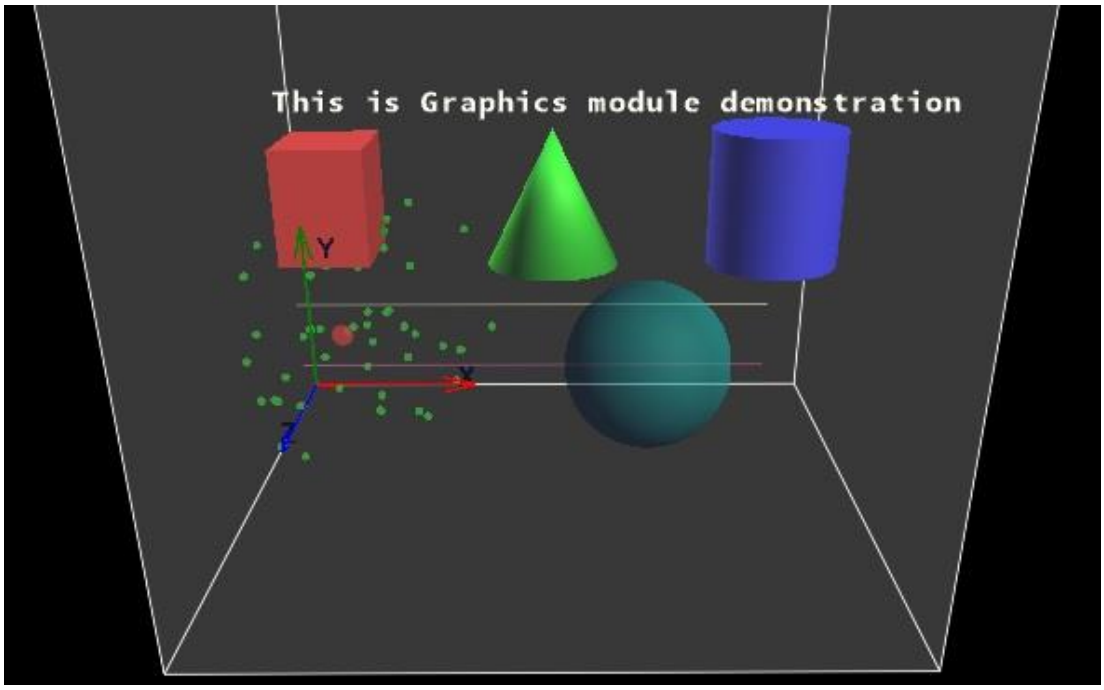

Figure 4.3: Demonstration of the Graphics module

4.3.2 Output Module

This module creates an output service for the other modules and mainly consists of three different parts:

The first part is the charting section which provides simple methods through its interface to create and display various kind of charts for the given data series. Three kinds of supported charts in the current version are: line charts, area charts and scatter charts.

The second part provides a document viewer that can display rich texts and tables. This part defines simple methods to add text with different formats and tables to the document. The resulting document then will be displayed in the document viewer of the GUI of this module which is loaded into the “Output Region” of the top-level application window.

Last part adds playback features for the animation of particles created in the graphics module. Using its simple GUI, users can play/pause the animation with different speeds or jump to any frame through the animation time. A sample view of this module is shown in Figure 4.4.

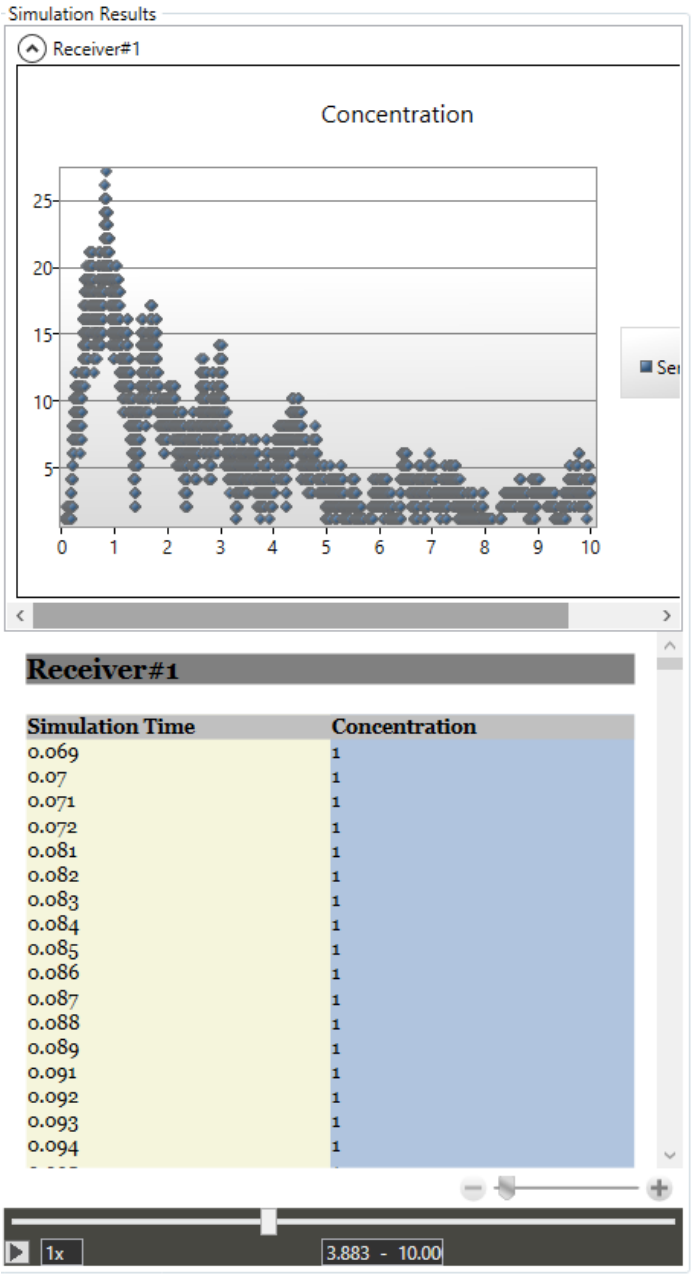

Figure 4.4: Demonstration of the Output module

4.3.3 Simulator Module

This module can be considered the main component of the application in terms of functionality and consists of two major parts:

1- A graphical user interface (GUI) that registers itself with the “Input Region” of the application top-level window and is used for setting the simulation parameters and inputs through interaction with users.
2- A logical model that defines simulator objects and relations between them which together will form the simulation logic and behavior.

In Figure 4.5, the diagram of simulator objects and relations between them is shown. In this diagram each arrow from an object to others represents that source object has a reference to the destination objects and can directly access them.

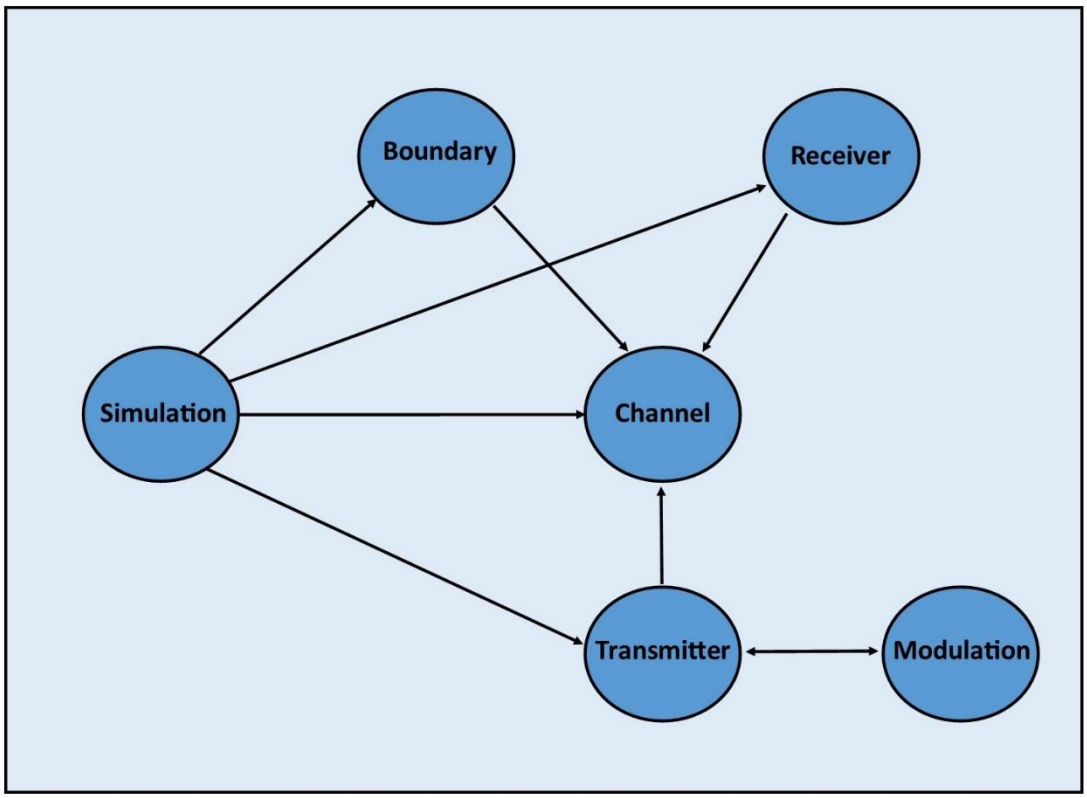

Figure 4.5: Simulator objects and relations between them

In the following section, functionality of each object and its behavior in the simulation logic will be explained.

4.4 Simulation Logic

The simulation logic in this project is implemented through set of objects and their behaviors and relations. Each object encapsulates a distinct functionality and responsibility in the simulation logic. Description of each object and its specific role will be explained next.

4.4.1 Simulation Object

This object acts as a coordinator for the rest of objects in the simulation. It is responsible for creating other objects and running the simulation logic. When simulation is started, it keeps track of the current simulation time and at each time step it will update the "Channel object". Furthermore, after simulation is finished it will invoke "Output Module" to display results on demand. Table 4.1 contains a short description of input parameters used in this object.

Table 4.1: List of input parameters for the "Simulation" object

Duration	Duration of simulation in seconds.
Time Step	Duration of each time step in seconds.
Sample Rate	Number of frames per second with which particle positions will be recorded for the playback animation
Seed	Random number generator seed (integer value).
Number of Runs	Number of times simulation will run. After first run, different time variant seed will be used in each run.

4.4.2 Channel Object

This object models propagation medium and keeps track of all information molecules at each time step during the simulation run. After it is updated by the "Simulation"

object at each time step, it will move each particle based on the defined diffusion model. Next it will notify rest of the objects though an event so they can act on the updated channel accordingly. Depending on features of the physical channel that is modeled in the simulation, different channel objects can be developed to meet the desired requirements, i.e., channel that models anomalous diffusion types or other propagation mechanisms. Default channel model included in the simulator is for modeling normal diffusion with drift. Table 4.2 lists input parameters related to this object in the simulator GUI.

Table 4.2: List of input parameters for the "Channel" object

Channel Drift	Velocity vector of the propagation medium ($\mu m/s$)
Temperature	Temperature of the propagation medium (k)
Viscosity	Viscosity of the propagation medium

4.4.3 Transmitter Object

Transmitter is responsible for emitting particles to the propagation medium based on the modulation scheme. Therefore it has a direct access to the "Channel" object and has a bi-directional relation with the "Modulation" object. Modulation object determines when and how messenger molecules are needed to be released into the channel and invokes the transmitter to do so. This decision is made based on the modulation technique that this object implements. Various modulation objects can be developed to model different modulation schemes and each transmitter then can choose which modulation to use. A simple pulse generator comes as a default modulator with the simulator that can merely be used to investigate the physical properties of the propagation channel and the reception process. This information

then can be used to model and create different modulation techniques to send binary data over the communication network. Adjustable parameters for the "Transmitter" object and "Modulation" object are shown in the Table 4.3:

Table 4.3: List of input parameters for the "Transmitter" and "Modulation" objects

Location	Location of the transmitter in the simulation space.
Particle Binding	An integer number that represents the affinity of particles. Particles only bind to the receptors with the same binding number.
Emission Radius	Radius of a sphere (μm) centered at the transmitter location which emitted particles initial location will be distributed randomly inside its bounds.
Wave Count	Number of pulses to be released into medium.
Particles per Wave	Number of particles that will be released in each wave.
Wave Period	Duration between two consequent waves in simulation steps.
Particle Radius	Radius of messenger molecule in nanometer.
Impulse Duration	Duration of each pulse in simulation steps.
Release Time	Simulation step number which first wave will be released at.

4.4.4 Receiver Object

Receiver objects have a direct reference to the "Channel" object and they will monitor propagation channel at each time step in the simulation run to detect presence of messenger molecules at their sensing area. Each receiver can act in one of the two operation modes, namely "Transparent" and "Ligand/Receptor" modes. In the transparent mode, receivers merely remain transparent to the presence of the particles in their sensing area. This means particles can freely propagate through them without encountering any collision or resistance. This mode is useful when one is interested to study effect of the propagation medium on the transmitted signal or to

measure concentration at the given location at different times. The Ligand/Receptor mode on the other hand, can be used to model the reception process for each receiver node based on the ligand-receptor kinetics in living cells. The purpose of this mode is to define a model inspired by the reception process in living cells in biology systems. In this operating mode, each receiver checks if particles collide with its surface and acts accordingly depending on where exactly these particles collided. For each particle that binds successfully to one of receiver's receptors, that particle will be marked as bound so the channel knows that this particle should not move in the next simulation time step. Table 4.4 catalogues input parameters related to the "Receiver" object.

Table 4.4: List of input parameters for the "Receiver" object

Location	Location of center of the receiver in the simulation space
Radius	Radius of the receiver (μm)
CoR	Coefficient of Restitution for the receiver surface
Longitude	Longitude of center of the receptor area
Latitude	Latitude of center of the receptor area
Aperture	Aperture of the receptor cone originating from receiver center
Binding	Represents affinity of the receptor. Only particles with the same number bind to this receptor kind
Receptors Count	Number of receptors in the corresponding receptor area
Receptor Radius	Radius of each receptor (nm)
Receptor CoR	Coefficient of Restitution for the receptor area surface
Reaction Probability	Intrinsic probability of reaction upon a given collision between an unbound receptor and a particle
Release rate	The rate at which bonds between particles and receptors breaks and particles release back to the medium

4.4.5 Boundary Object

The purpose of this object is to provide a bounded space for the Brownian motion. It has a reference to the "channel" object and monitors the channel to check if any particle collided with its bounds. If any collision is detected, it will reflect that particle back to inside its bounds. Moreover, each edge can be set to absorb the colliding particles and remove them from the propagation medium. Different boundary shapes can be developed based on needs of the simulation scenario. By default a box shaped boundary object is included in the simulator and its input parameters are listed in Table 4.5:

Table 4.5: List of input parameters for the "Boundary" object

Location	Location of bottom-left edge of the boundary box
Size	Width, height and depth of the boundary box (μm)
CoR	Coefficient of Restitution for colliding particles with the boundary surface.

Chapter 5

RESULTS

In this chapter, we conduct a set of simulations to evaluate the functionality of the simulator. In Section 5.1, we compare results obtained from simulations to that of analytical models when receivers are set to the transparent mode. Furthermore, we investigate the effect of the counting noise (Brownian noise) and intersymbol interference (ISI) as two important factors that affect the reception of molecules at the receiver [12]. In Section 5.2, operation of the receivers in the Ligand-Receptor mode will be explored and effect of different parameters on the reception process will be investigated. All the simulations in this chapter have been executed on a PC with Intel Core i7-2630QM CPU and 8 GB of RAM running Microsoft Windows 8.1 operating system. On average each run of simulation took five minutes to complete, although this time can change depending on input parameters such as simulation duration, time step and number of particles.

5.1 Transparent Mode Receivers

Transparent receivers are sometimes named *Ideal* receivers in the literature. It stems from the fact that, these receivers are able to detect and receive all the molecules in their sensing area. Although in the reality this is not possible, but for the sake of simplicity, many researchers used this model in their analysis. Hence, this mode of reception is implemented in this simulator and in this section we look into the behavior of receivers in this mode.

5.1.1 Effect of receiver distance on the received signal

Based on Fick's second law of diffusion, average molecular concentration at position X and time t after the transmission when an impulse of Q particles is transmitted at time zero can be expressed as:

$$\bar{U}(X,t) = \frac{Q}{(4\pi Dt)^{\frac{3}{2}}} \exp\left(\frac{-|X-X_{tx}|^2}{4Dt}\right) \quad (5.1)$$

where D is the diffusion coefficient and X_{tx} is the position of the transmitter.

In the first scenario, in order to compare the simulation results with the numbers obtained from (5.1), three receivers are placed at distances: 30, 45, and 60 micrometers from the transmitter. These numbers are chosen based on the typical ranges that have been suggested for a short-range molecular communication [3]. Table 5.1, lists input parameters of the simulator for this case.

Table 5.1: Input parameters for the first scenario simulation

Duration (*sec*)	Time step (*sec*)	Q	Viscosity ($Pa{\cdot}s$)	T (K)	Receiver radius (μm)	Particle Radius (nm)
70	0.0005	2000	0.001	310	7	4

In order to estimate mean values, the simulation was run 5 times and results are obtained by averaging the realizations from each run.

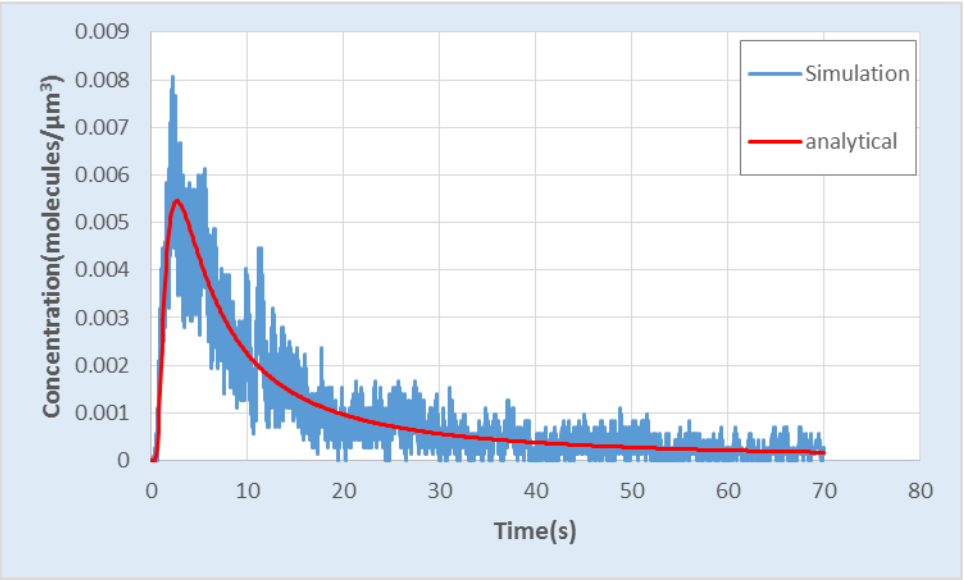

Figure 5.1: Comparing concentration of the received molecules at the receiver placed at 30µm distance from the transmitter using the results obtained by the simulator and values from analytical model

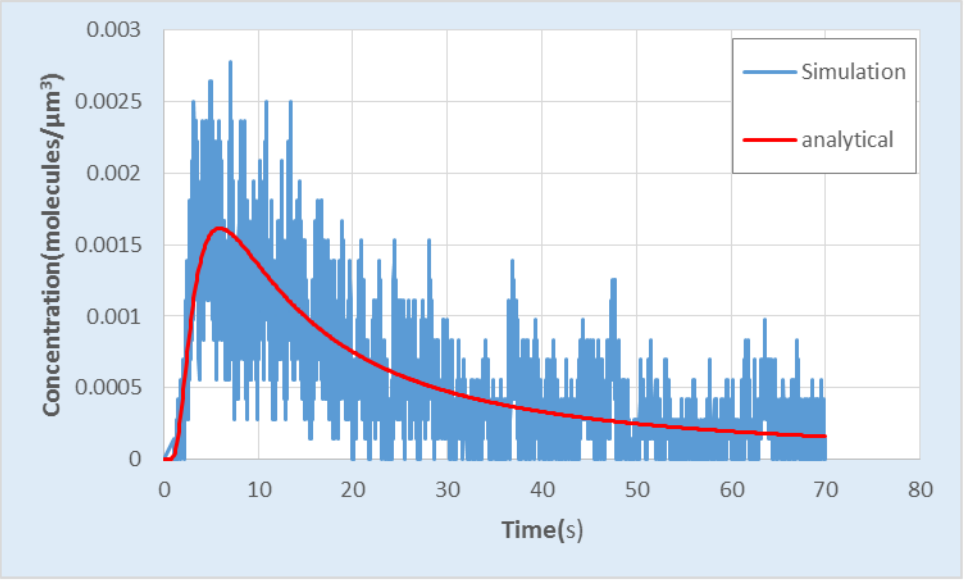

Figure 5.2: Comparing concentration of the received molecules at the receiver placed at 45µm distance from the transmitter using the results obtained by the simulator and values from analytical model

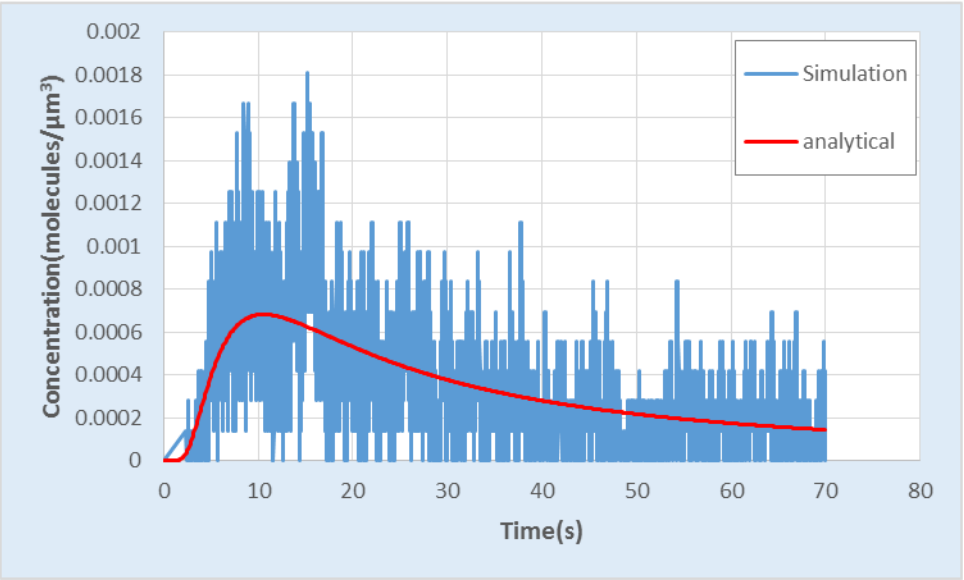

Figure 5.3: Comparing concentration of the received molecules at the receiver placed at 60μm distance from the transmitter using the results obtained by the simulator and values from analytical model

From Figures (5.1) – (5.3), we can observe the perturbation of concentrations obtained from the simulation around the mean concentrations calculated for the center of the three receivers using (5.1). Concentrations obtained from the simulation are calculated by counting the number of received molecules in each time step and dividing it by the volume of the receiver. It worth noting that, in order to more accurately compare the concentrations obtained from simulation to the values calculated from (5.1), one should integrate the latter over the receiver volume:

$$C(t) = \frac{\iiint_{V_{rec}} \bar{U}(X,t)dv}{V_{rec}} \tag{5.2}$$

Nevertheless, the perturbations observed are mainly due to the Brownian noise which results from the fact that the number of particles present for counting in the receiver sensing area, is influenced by molecules random motion and subjected to change in each time step. Moreover, for the receivers further away from the transmitter, it can

be seen that, variance of the concentration from its mean analytical values increases significantly. These increases can be explained by the fact that the mean squared displacement of the particles is proportional to the time elapsed:

$$\bar{x}^2 \sim Dt \tag{5.3}$$

As a result, particles, which diffuse to the further receivers, tend to move over greater distances in each time step, and consequently more counting noise (Brownian noise) is produced.

In order to investigate the effect of number of simulation runs on the obtained results, another simulation conducted with the same setup as before, this time using only one realization of the Wiener process. The result can be seen in Figure 5.4:

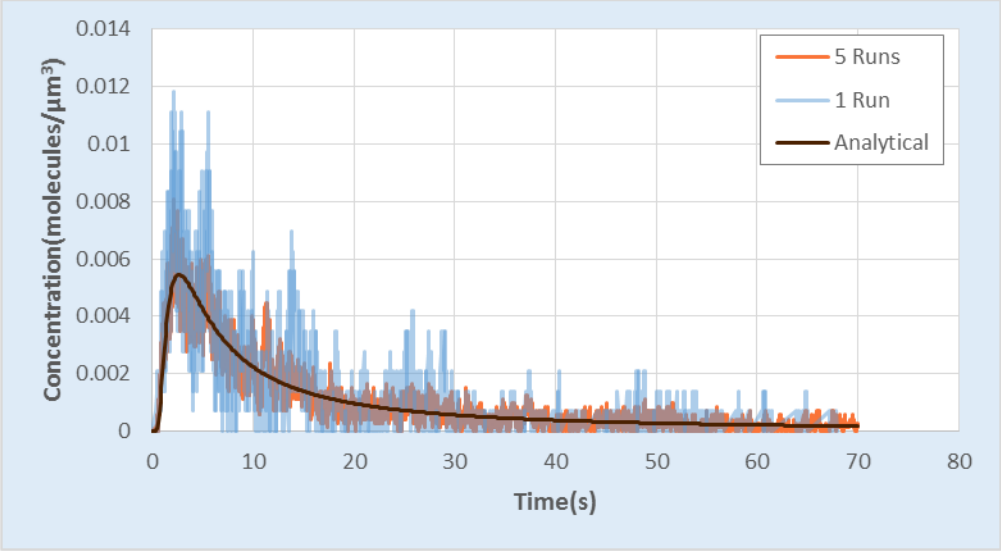

Figure 5.4: Comparing concentration of the received molecules at the receiver placed at 30µm distance from the transmitter using the results obtained by the simulator after one and five runs and values from analytical model

It is obvious from the Figure 5.4 that, averaging over more number of realizations results in closer numbers to that of mean analytical values. Furthermore, from Figures (5.1) – (5.4), we can conclude that in order to estimate mean values for receivers further away, generally one needs to run more simulations.

In the second scenario, we are interested to measure the effect of the distance on the intersymbol interference at the receivers. Similar to the previous scenario, three receivers are placed at the distances: 30, 45, and 60 micrometers from the transmitter. Table 5.2 lists input parameters of the simulator for this case.

Table 5.2: Input parameters for the second scenario simulation

Duration (*sec*)	Time step (*sec*)	*Q*	Viscosity (*Pa·s*)	*T* (*K*)	Receiver radius (*μm*)	Particle Radius (*nm*)
50	0.0005	5*2000	0.001	310	7	4

Five impulses of 2000 particles each are released by the transmitter with the 10 second intervals in order to measure the intersymbol interference at the receivers. The simulation was run 5 times and the results are obtained from averaging the realizations from each run.

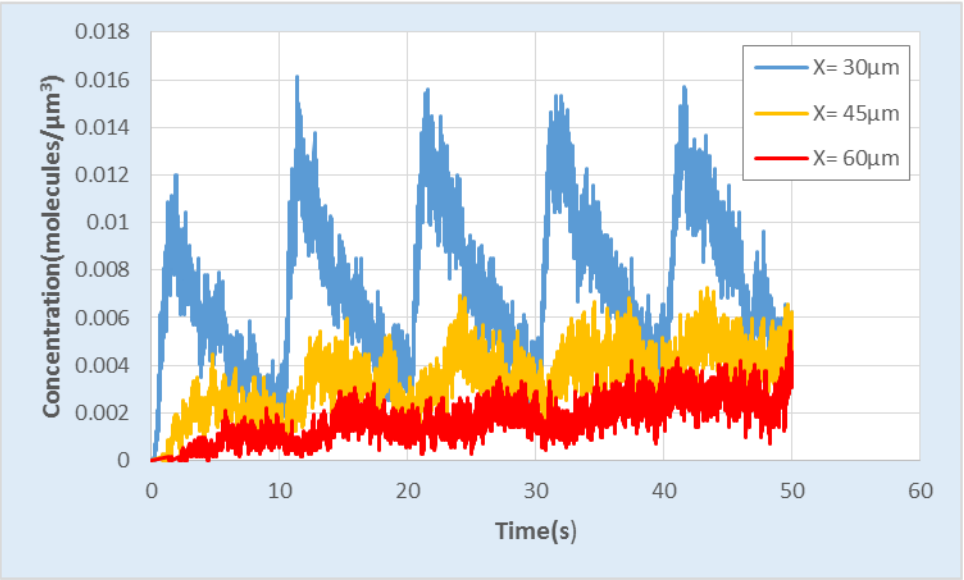

Figure 5.5: Intersymbol interference at receivers placed at the distances 30, 45, and 60 micrometers from the transmitter

Considering Figure 5.5, it is understood that as distance grows intersymbol interference has a greater effect on the received signal. It can be explained by the fact that, at the larger distances from the transmitter received signal decays more slowly from its peak value (Figure 5.3), hence there are more number of molecules from the previous impulses present at the receiver sensing area when new ones arrive.

5.1.2 Effect of receiver size on the received signal

In this section, we will investigate the effect of the receiver size on the received signal and noise at the receivers. This will be done through two simulations sets. In the first simulation, three receivers with radii of 4, 7 and 10 micrometer respectively will be placed at 30µm of the transmitter and simulation will be run for 5 times to get the average values. The radius of the receivers are chosen to be close to the size of a typical living cell [37]. Rest of input parameters are same as of Table 5.2.

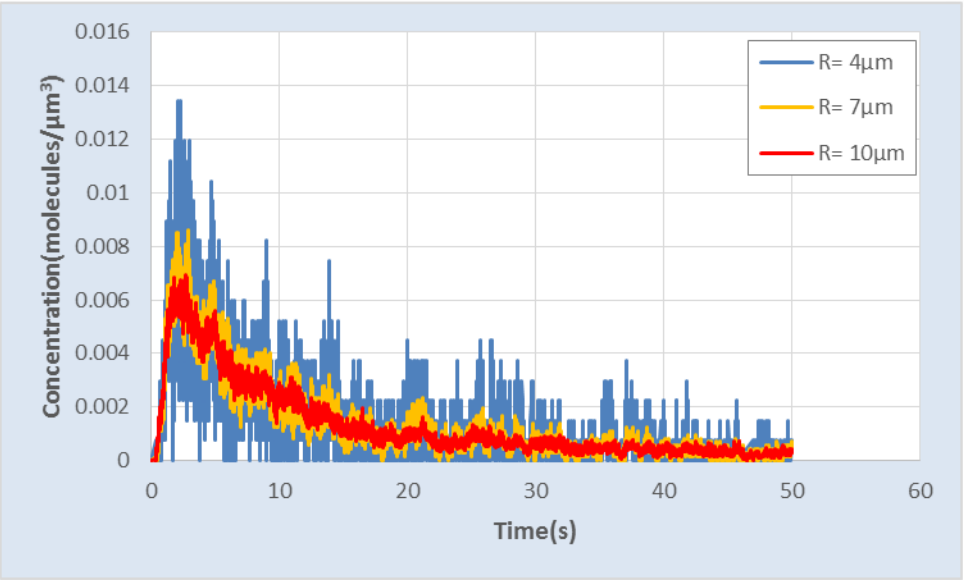

Figure 5.6: Received signal at three receivers with radius 4, 7 and 10µm

According to Figure 5.6, as radius of receiver increases, the Brownian noise at the receiver will decrease. This observation and the previous ones which explored the effect of distance on the Brownian noise, are consistent with the following equation which relates magnitude of the Brownian noise to the concentration at the receiver and the receiver volume [19]:

$$RMS(N_b) = \sqrt{\frac{C_R}{(\frac{4}{3})\pi r_{rec}^3}} \tag{5.4}$$

where $RMS(N_b)$ is the root mean square value of the Brownian noise.

In the second set, over five runs, five impulses of 2000 particles each are released by the transmitter with the 10 second intervals in order to measure the intersymbol interference at the receivers.

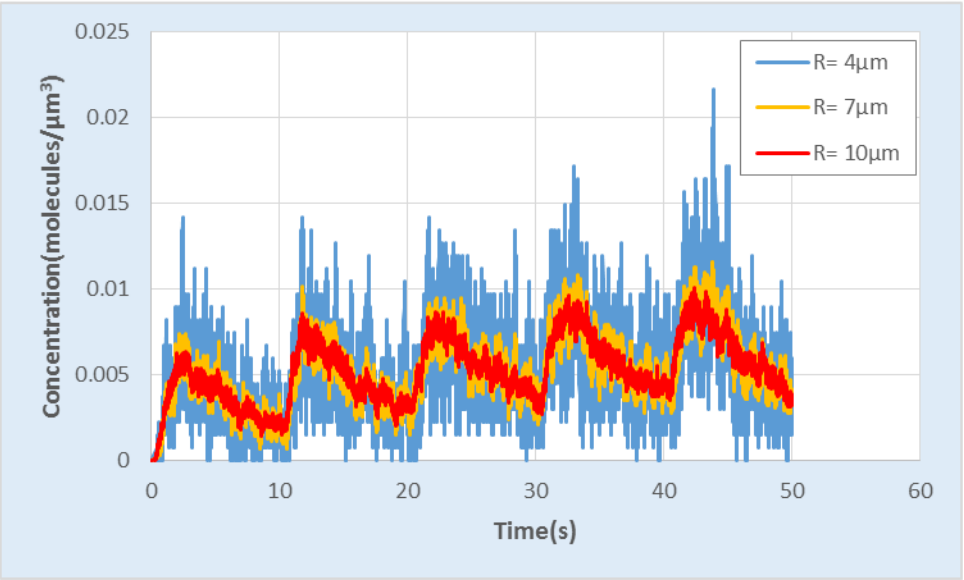

Figure 5.7: Intersymbol interference at three receivers with radius 4, 7 and 10µm

Interestingly, Figure 5.7 shows that receiver size does not affect intersymbol interference at the receiver. Hence, this kind of noise is merely affected by receiver distance and signal frequency.

Although averaging over numerous realizations will result in a closer values to the mean analytical numbers, noise in the received signal will not completely diminish. Hence, the knowledge of intersymbol interference and counting noise presented above can be used in the design of optimum receivers [12].

5.2 Receptor Mode Receivers

In the previous section, we observed the characteristics of the ideal receiver. However, in reality, a receiver cannot receive all the carrier molecules in its sensing area and the reception of molecules occurs through the reaction with the receptors embedded on the receiver surface. Moreover, particles cannot move through the receiver and they will reflect over a collision that does not lead to a bond. Number of

bound receptors in the ligand-receptor binding process in living cells, at the steady state, can be expresses as follows [32]:

$$L + R \underset{k_r}{\overset{k_f}{\rightleftarrows}} N_b \tag{5.5}$$

where, L is the number of ligands available for binding, R is the number of free receptors, N_b is number of bonds and k_f and k_r are corresponding reaction rate and release rates. This equation denotes that at the steady state, the rate at which bonds will form will be equal to the release rate. In order to evaluate behavior of this model in the simulator, a transmitter and a receiver are placed inside a bounded space. The boundary will ensure that concentration of molecules (ligands) is constant throughout the simulation. Table 5.3, shows input parameters for this case:

Table 5.3: Input parameters for the bounded space scenario

Duration (*sec*)	Q	Receiver radius (μm)	Receptor Radius (nm)	Particle Radius (nm)	Reaction Probability (P_r)
50	2000	7	5	4	0.7

Reaction probability (P_r) depends on the activation energy and the temperature of the environment (see Eq. 3.16). Generally, activation energy in the chemical reactions is in order of kilo joule per mole. But, it greatly varies based on the different parameters such as the temperature and type of the reactants. Therefore, the chosen value of P_r in Table 5.3 can be considered as an upper bound for P_r. Parameters

which are not listed in Table 5.3 are same as before. Simulation was run 5 times for different set of receptors and release rates and the results are averaged.

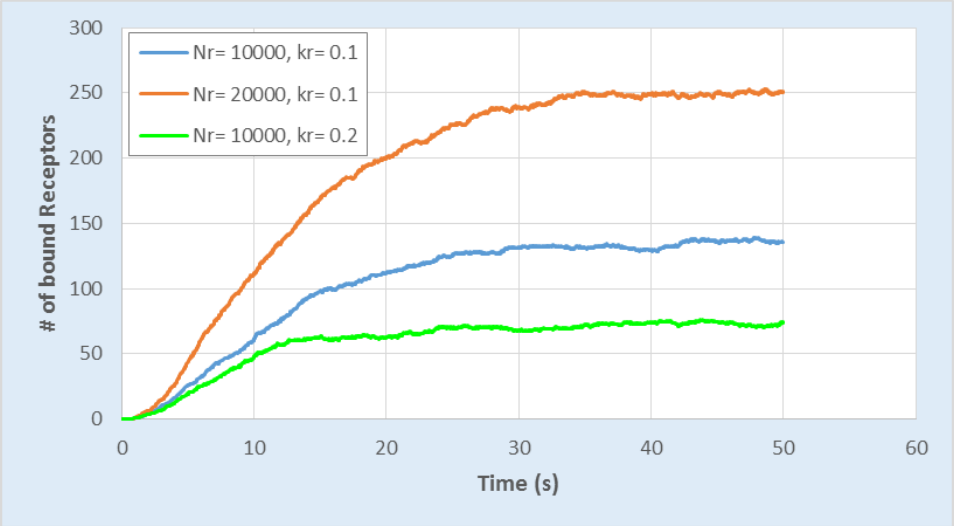

Figure 5.8: The effect of number of receptors (N_r) and release rate (k_r) on the number of receptor/ligand bonds in the steady state

From Figure 5.8, it is evident that increasing the number of receptors results in more bonds while also increasing the time that system reaches the steady state. On the other hand, greater release rate numbers leads to decreased number of bonds and the time that system reaches steady state. The steady state number of receptor/ligand bonds can be considered as the reception capacity of the receiver for the given particle concentration.

In a typical diffusion-based molecular communication however, a transmitter sends information symbols through number of impulses of messenger molecules that decay over time at the receiver. Hence, in order to investigate the behavior of the reception process for this case, some set of simulations will be conducted under different parameters that might affect the reception process.

In the first scenario, a receiver with radius of 7 micrometer will be placed at the 30 micrometer distance from the transmitter and series of simulation will be performed for different values of receptors and release rates. Each simulation set will be run for 5 times and results will be averaged.

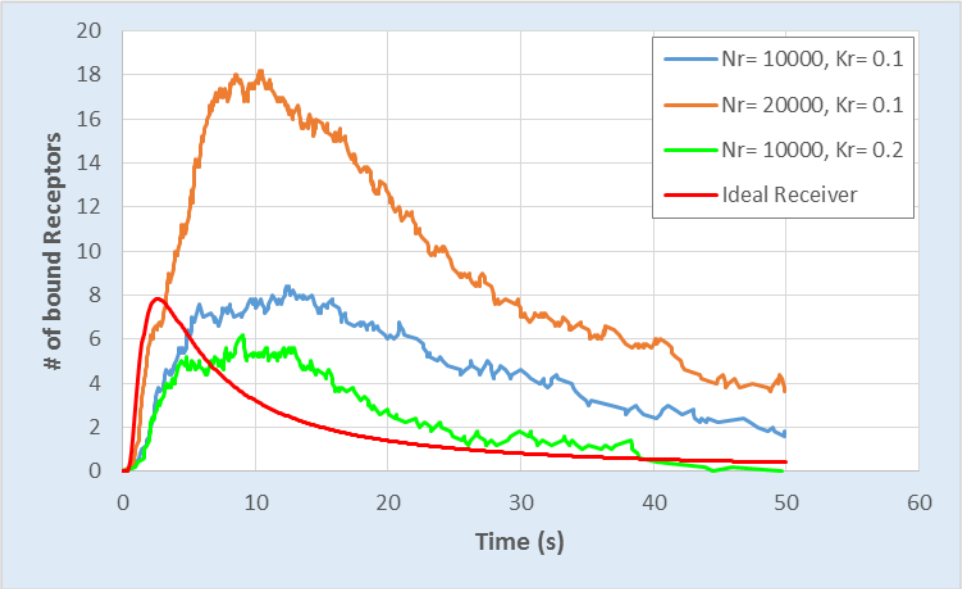

Figure 5.9: The effect of number of receptors (N_r) and release rate (k_r) on the number of receptor/ligand bonds in the receiver placed at 30μm from the transmitter

In order to compare the number of sensed particles for an ideal receiver, which is placed at the same distance from the transmitter and has the same radius as the simulated receiver, to the number of bound receptors, the concentration of the particles at the center of the ideal receiver (calculated numerically from Eq. 5.1) is multiplied by the volume of the ideal receiver and the results are depicted in Figure 5.9.

From Figure 5.9, we can observe that increasing the number of receptors will result in the higher received signal peak value. Furthermore, increasing the release rate will increase the rate at which molecules will be released from their bond and causes less

peak value and faster signal decays at the same time. Moreover, we can see that, peak values and decay times can be completely different from the analytical models, depending on number of receptors and release rate. This is an important observation that can affect design of optimum receivers and modulation schemes that are based on the ideal models.

Here we define a performance metric for comparing the quality of received signal as the ratio of the received signal peak value to the time width of its half max value [38]:

$$F_d = \frac{\text{peak value}}{\Delta t \text{ of half max value}} \tag{5.6}$$

The larger this ratio the taller and narrower the signal shape at peak will be. Ideally for the smaller error in the detection of the received signal, the peak shape must be as tall and as narrow as possible. Hence, larger values of F_d are desirable. Values of the F_d for this scenario are listed in Table 5.4:

Table 5.4: Values of the F_d for the first scenario

Receiver Parameters	Analytical	N_r= 20000 k_r= 0.1	N_r= 10000 k_r= 0.2	N_r= 10000 k_r= 0.1
F_d	1.1	0.82	0.39	0.29

F_d in Table 5.4 for the analytical received signal is calculated by discretizing the values of (5.1) for the arbitrarily selected time interval of 0.01 seconds. Δt then is found for the two time intervals in which the received signal value was

approximately half of the signal peak value. It can be seen that analytical value for the ideal receiver has the highest ratio and can be considered as the upper bound of F_d. Next highest value is for the receiver with 20000 receptors. Comparing two receivers with the same number of receptors we can see that higher release rate results in a larger ratio.

In order to explore the effect of receiver size on the received signal, in the second scenario we will increase the size of the receptor to 10µm and leave other parameters same as the previous scenario. Increasing the receiver size should increase the chance that particles collide with the receiver and hence increase the chance that reactions happen.

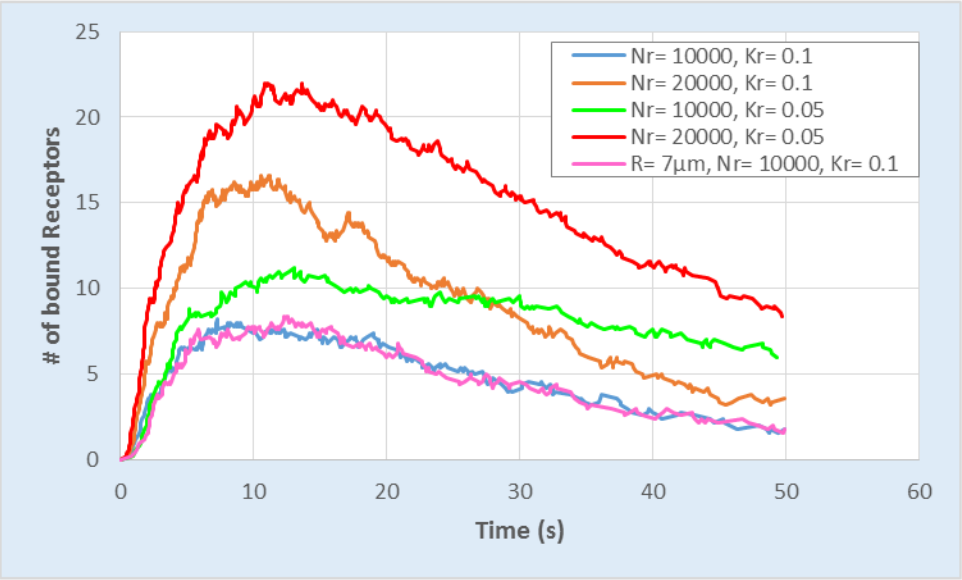

Figure 5.10: The effect of number of receptors (N_r) and release rate (k_r) on the number of receptor/ligand bonds in the receiver with radii of 10µm

An interesting observation from Figure 5.10 is that the received signal for the both receivers with radius of 7 and 10 micrometer is almost identical when number of bound receptors and release rate are the same. This is due the fact that, even when

the larger receiver experiences more collisions (12851 versus 5445), fewer number of collisions lead to a reaction when number of receptors are identical. Values of the F_d for this scenario are shown in Table 5.5:

Table 5.5: Values of the F_d for the second scenario

Receiver Parameters	N_r= 20000 k_r= 0.1	N_r= 20000 k_r= 0.05	N_r= 10000 k_r= 0.1	N_r= 10000 k_r= 0.05
F_d	0.614	0.56	0.28	0.23

Again we can see that the highest number of F_d, corresponds to the largest number of receptors and release rate. By comparing the case that F_d is equal to 0.56 in Table 5.5 with the case that F_d is equal to 0.28 we can conclude that extra number of receptors has greater effect on the F_d than larger release rate. Overall, based on above results, we can conclude that the ratio F_d increases with the number of receptors and release rate.

Finally, in the third scenario, we will release five waves of 2000 molecules with period of 10 seconds to study behavior of the receiver under consequent impulses. The receiver size is 7 micrometer and is placed at 30 micrometer distance from the transmitter.

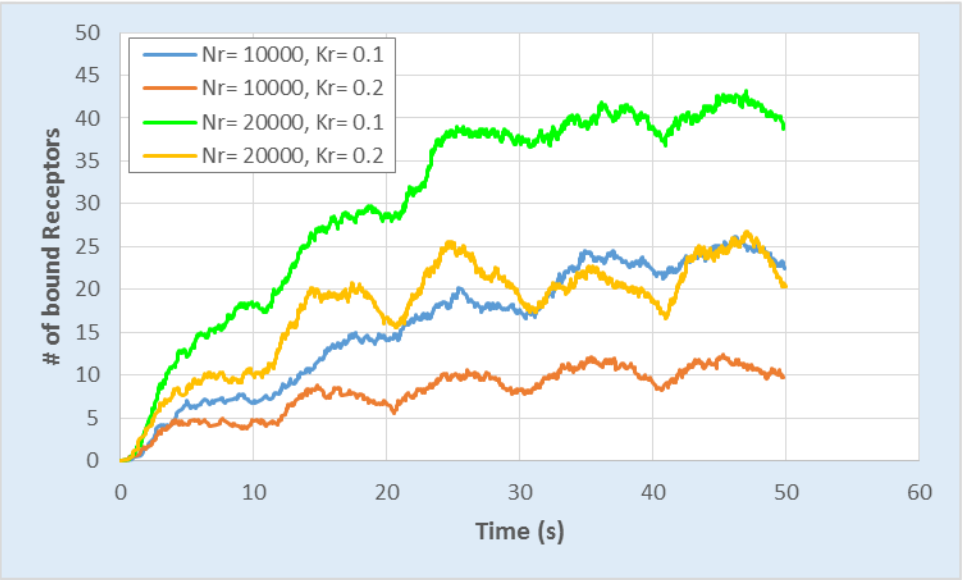

Figure 5.11: The effect of number of receptors (N_r) and release rate (k_r) on the number of receptor/ligand bonds when 5 waves of 2000 molecules transmitter every 10 seconds

According to Figure 5.11 we can say that receivers with the higher number of receptors and lower release rate, have a higher capacity in terms of number of receptor/ligand bonds they can achieve before this number starts to drop. However, in order to have a more distinct received signals, we can conclude that having a higher number of receptors and release rate on the receiver is more desirable. Nevertheless, if we are interested in designing modulation schemes for molecular communications and these two factors are fixed, i.e. using biological nanomachines with certain characteristics, then simulations or experiments are inevitable in order to find the proper modulation parameters.

The summary of important findings in this chapter is as follows:

- The Brownian noise at the receiver increases with distance from the transmitter and decreases with receiver size.

- Intersymbol interference at the receiver is merely affected by receiver distance and signal frequency.
- In order to estimate mean values for receivers further away, generally one needs to run more simulations.
- Increasing the number of receptors will result in a higher received signal peak values.
- Increasing the release rate will increase the rate at which particles will be released from their bonds and causes lower peak values and faster signal decays at the same time.
- In the receptor mode, peak values and decay times can be completely different from the analytical models.
- The ratio F_d (5.6) increases with the number of receptors and release rate.
- In order to have a more distinct received signals, having a higher number of receptors and release rate on the receiver is more desirable.

Chapter 6

CONCLUSION

In this work, diffusion-based molecular communication as a promising communication paradigm has been described. Furthermore, a realistic reception process has been studied and its implementation details are presented. Next, design and implementation of the simulator is demonstrated and it was shown that a composite design provides many merits including extensibility, flexibility and reusability of the application modules. In order to evaluate the functionality of the simulator when receiver operates in transparent mode and receptor mode, many simulations have been conducted for each case. In the transparent mode, we show that the Brownian noise at the receiver increases with distance from the transmitter and decreases with receiver size. Moreover, it was concluded that intersymbol interference is merely affected by the transmitted signal frequency and receiver distance from the transmitter. In the receptor mode, a performance metric for comparing the quality of received signal was introduced and it was shown that having larger numbers of receptor in the receiver and greater values of release rate yields more desirable results.

The main contribution of this thesis is to present a simulator for the diffusion-based molecular communication networks. The unique features of this simulator are its ability to model a reception process which is inspired from the ligand-receptor binding kinetics in living cells; a graphical user interface and visualization

capabilities that can display simulation elements by 3-*D* graphics and its composite design that lets new modules easily developed and added to the simulator based on the needs of users and simulation model.

Future work, will involve adding more features to the simulator and developing different modulation schemes that can be used to transmit binary data over the network as well as modeling the anomalous diffusion that accounts for the interaction between messenger molecules.

REFERENCES

[1] N. Taniguchi, "On the Basic Concept of 'Nano-Technology'," in *Proc. Intl. Conf. Prod.Eng*, Tokyo: Japan Society of Precision Engineering, 1974.

[2] T. Suda, M. Moore, T. Nakano, R. Egashira & A. Enomoto, "Exploratory research on molecular communication between nanomachines," in *Proceedings of Genetic and Evolutionary Computation Conference*, 2005.

[3] l. F. Akyildiz, F. Brunetti, C. Blázquez, "Nanonetworks: A new communication paradigm," *Computer Networks,* vol. 52, no. 12, pp. 2260-2279, 2008.

[4] I. F. Akyildiz, J. M. Jornet, M. Pierobon, "Nanonetworks: A New Frontier in Communications," *Communications of the ACM,* vol. 54, no. 11, pp. 84-89, 2011.

[5] E. Drexler, Nanosystems: Molecular Machinery, Manufacturing, and Computation, John Wiley and Sons Inc, 1992.

[6] I. F. Akyildiz, J. M. Jornet & C. Han, "Terahertz Band: Next Frontier for Wireless Communications," *Physical Communication (Elsevier)* , vol. 12, pp. 16-32, 2014.

[7] I. F. Akyildiz, F. Fekri, R. Sivakumar, C. Forest & B. Hammer, "MoNaco: Fundamentals of Molecular Nano-Communication Networks," *IEEE Wireless*

Communication, vol. 19, no. 5, pp. 12-18, 2012.

[8] T. Nakano, M. Moore, A. Enomoto & T. Suda, "Molecular Communication Technology as a Biological ICT," *Biological Functions for Information and Communication Technologies,* pp. 49-86, 2011.

[9] T. Nakano, M. Moore, F. Wei, A. Vasilakos & J. Shuai, "Molecular Communication and Networking: Opportunities and Challenges," *IEEE Transactions on NanoBioscience,* vol. 11, no. 2, pp. 135 - 148, 2012.

[10] M. Gregori & I. Akyildiz, "A New NanoNetwork Architecture Using Flagellated Bacteria and Catalytic Nanomotors," *IEEE Journal On Selected Areas In Communications,* vol. 28, no. 4, pp. 612-619, 2010.

[11] L. P. Giné & I. F. Akyildiz, "Molecular communication options for long range nanonetworks," *Computer Networks,* vol. 53, no. 16, pp. 2753-2766 , 2009 .

[12] H. ShahMohammadian, G. G. Messier & S. Magierowski , "Optimum receiver for molecule shift keying modulation in diffusion-based molecular communication channels," *Nano Communication Networks,* vol. 3, no. 3, p. 183–195, 2013.

[13] I. F. Akyildiz, H. B. Yilmaz, M. S. Kuran & T. Tugcu, "Modulation Techniques for Communication via Diffusion in Nanonetworks," in *IEEE International*

Conference on Communications, Kyoto, 2011.

[14] S. Kadloor, R. Adve & A. Eckford, "Molecular communication using Brownian motion with drift," *IEEE Transactions on Nanobioscience,* vol. 11, no. 2, pp. 89-99, 2011.

[15] L. Vlahos, H. Isliker, Y. Kominis & K. Hizanidis, "Normal and Anomalous Diffusion: A Tutorial," in *Order and Chaos*, 2008.

[16] I. Llatser, E. Alarcon & M. Pierobon, "Diffusion-based channel characterization in molecular nanonetworks," in *IEEE Conference on Computer Communications Workshops*, 2011.

[17] M. Pierobon & I. F. Akyildiz, "A physical end-to-end model for molecular communication in nanonetworks," *IEEE Journal on Selected Areas in Communications,* vol. 28, no. 4, pp. 602-611, 2010.

[18] D. Arifler, "Capacity analysis of a diffusion-based short-range molecular nano-communication channel," *Computer Networks,* vol. 55, no. 6, p. 1426–1434, 2011.

[19] M. Pierobon & I. F. Akyildiz, "Diffusion-Based Noise Analysis for Molecular Communication in Nanonetworks," *IEEE Transactions on Signal Processing,* vol. 59, no. 6, pp. 2532 - 2547, 2011.

[20] M. Pierobon & I. Akyildiz, "Noise Analysis in Ligand-Binding Reception for Molecular Communication in Nanonetworks," *IEEE Transactions on Signal Processing,* vol. 59, no. 9, pp. 4168 - 4182, 2011.

[21] H. ShahMohammadian, G. G. Messier & S. Magierowski, "Nano-machine molecular communication over a moving propagation medium," *Nano Communication Networks,* vol. 4, no. 3, p. 142–153, 2013.

[22] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry,* vol. 26, no. 16, p. 1781–1802, 2005.

[23] E. Lindahl, B. Hess & D. van der Spoel, "GROMACS 3.0: A package for molecular simulation and trajectory analysis," *Journal of Molecular Modeling,* vol. 7, no. 8, pp. 306-317, 2001.

[24] S. Plimpton, P. Crozier & A. Thompson, "LAMMPS Molecular Dynamics Simulator," Sandia National Laboratories, [Online]. Available: http://lammps.sandia.gov/.

[25] E. Gul, B. Atakan & O. B. Akan, "NanoNS: A nanoscale network simulator framework for molecular communications," *Nano Communication Networks,* vol. 1, no. 2, p. 138–156, 2010.

[26] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *The Journal of Physical Chemistry*, vol. 81, no. 25, p. 2340–2361, 1977.

[27] H. Berg, Random Walks in Biology, Princeton: Princeton University, 1993.

[28] g. Llatser, D. Demiray, A. Cabellos-Aparicio, D. T. Altilar and E. Alarcón, "N3Sim: Simulation framework for diffusion-based molecular communication nanonetworks," *Simulation Modelling Practice and Theory,* vol. 42, pp. 210-222, 2014.

[29] L. Felicetti, M. Femminella & G. Reali, "A simulation tool for nanoscale biological networks," *Nano Communication Networks,* vol. 3, no. 1, pp. 2-18, 2011.

[30] T. Nakano, A. W. Eckford & T. Haraguchi, Molecular Communication, Cambridge University Press, 2013.

[31] F. Beer, E. J. Jr., E. Eisenberg & P. Cornwell, Vector Mechanics for Engineers: Dynamics, McGraw-Hill, 2009, pp. 821-823.

[32] D. A. Lauffenburger & J. J. Linderman, Receptors: models for binding, trafficking, and signalling, New York: Oxford University Press , 1993.

[33] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," 23 01 2004. [Online]. Available:

http://www.martinfowler.com/articles/injection.html.

[34] Microsoft, "Microsoft prism," 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/ff648465.aspx.

[35] "DigitalRune Game Engine," DigitalRune, 2014. [Online]. Available: http://www.digitalrune.com/,.

[36] "Microsoft XNA Framework," Microsoft , [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=20914.

[37] R. A. Freitas Jr., Nanomedicine, Volume I: Basic Capabilities, Georgetown, TX: Landes Bioscience, 1999.

[38] F. Nariman, G. Weisi & A. W. Eckford, "Tabletop molecular communication: text messages through chemical signals," *PloS one*, vol. 8, no. 12, p. e82935, 2013.

APPENDICES

Appendix A: Inelastic Collisions

When two particles collide with each other, their velocities after impact are unknown in magnitude as well as in direction. In order to find the velocity vectors after impact, we consider as coordinate axes, the *n* axis along the common normal to the surface in contact between the two particles at the time of impact and *t* axis along their common tangent. Assuming that the particles are perfectly smooth and frictionless, the only impulse exerted on the particles during the impact are due to internal forces directed along the *n* axis. Hence, the velocity of each particle along the *t* axis remains unchanged after the impact and we can write:

$$(v_A)_t = (\acute{v}_A)_t \qquad (v_B)_t = (\acute{v}_B)_t \tag{A.1}$$

Also, the total momentum of the two particles along the *n* axis is conserved and is as follows:

$$m_A(v_A)_n + \quad m_B(v_B)_n = m_A(\acute{v}_A)_n + m_B(\acute{v}_B)_n \tag{A.2}$$

Moreover, velocities of the two particles after impact is related to their velocities before impact as follows:

$$(\acute{v}_B)_n - \ (\acute{v}_A)_n = \ e[(v_A)_n - \ (v_B)_n] \tag{A.3}$$

where *e* is the coefficient of the restitution.

Solving equations (A.2) and (A.3) for $(\acute{v}_B)_n \ and \ (\acute{v}_A)_n$ yields:

$$(\acute{v}_A)_n = \frac{(m_A - em_B)(v_A)_n + (1+e)m_B(v_B)_n}{m_A + \ m_B}$$

$$(\acute{v}_B)_n = \frac{(m_B - em_A)(v_B)_n + (1+e)m_A(v_A)_n}{m_A + m_B} \tag{A.4}$$

In the case that one particle collides with another static rigid body which is significantly larger than the particle (i.e., $m_B \gg m_A \ and \ v_B = 0$), we can write:

$$(\acute{v}_A)_n = \frac{(m_A - em_B)(v_A)_n}{m_A + m_B}$$

$$= \lim_{m_B \to \infty} \frac{(m_A - em_B)(v_A)_n}{m_A + m_B} = -e(v_A)_n \qquad \text{(A.5)}$$

From (A.1) and (A.5), we can see that the final velocity of the particle in *n* direction is in the opposite direction and only modified by the coefficient *e* while velocity in *t* direction remains unchanged.

Appendix B: Simulator Quick Start Guide

This simulator can be run on any PC that runs Windows 7 or newer version of the Microsoft operating system with at least one gigabyte of free RAM. However, two gigabyte of free RAM or more is recommended for most cases. Also, the Microsoft .Net framework 4.5 or newer versions are required to be installed on the system. This software does not require any installation and directly can be executed from its directory.

The following tips can be helpful while working with this application:

- In directory "NanoSimulator" of the software distribution, double click on "NanoSimulator.exe" to start the simulator.
- All the input parameters are briefly explained in the corresponding tables in Chapter 4. Moreover, moving the mouse cursor over an input parameter in the application will show a brief tooltip for that parameter.
- To start the simulation, at least one transmitter should be added to the simulation.
- The camera can be moved in the scene to have different views of the simulation, by pressing one of the following keys: A, W, S, D, R and F. Also, it can be rotated by moving the mouse to different directions while holding the right mouse button.

Appendix C: Source Codes

The simulation object source code:

```
public class Simulation : BindableBase, IDisposable
    {
        //------------------------------------------------------------
        #region Fields
        //------------------------------------------------------------
        private Random random;
        private decimal simulationTime;
        private Transmitter activeTransmitter;
        private Receiver activeReceiver;
        private ObservableCollection<Receiver> receivers;
        private ObservableCollection<Transmitter> transmitters;
        private Boundary boundary;
        private decimal deltaT;
        private bool useBounds;
        private int mumOfRuns;
        private int sampleRate;
        private BrownianChannel channel;
        private bool disposed;
        private DataManager dataManager;
        //------------------------------------------------------------
        #endregion
        //------------------------------------------------------------
        #region Constructor
        //------------------------------------------------------------
        public Simulation()
        {
            //Simulation Properties
            SimulationDuration = 5;
            Seed = 12345;
            mumOfRuns = 1;
            random = new Random(Seed);
            deltaT = 0.001M;
            sampleRate = 60;
             channel = new BrownianChannel(SimulationDuration, SampleRate,         deltaT,
        random);
            receivers = new ObservableCollection<Receiver>();
            Receivers = new ListCollectionView(receivers);
            Receivers.CurrentChanged += Receivers_CurrentChanged;
            transmitters = new ObservableCollection<Transmitter>();
            Transmitters = new ListCollectionView(transmitters);
            Transmitters.CurrentChanged += Sources_CurrentChanged;
            useBounds = false;
        }

        ~Simulation()
        {
            if (!disposed)
                Dispose();
        }

        public void Dispose()
        {
            foreach (var receiver in receivers)
            {
                receiver.Dispose();
            }
            foreach (var transmitter in transmitters)
            {
                transmitter.Dispose();
            }
            if (UseBounds)
                Boundary.Dispose();
            channel.Dispose();
            disposed = true;
        }
        #endregion
```

```
//------------------------------------------------------------
#region Properties & Events
//------------------------------------------------------------
public event EventHandler SimulationFinished;
public ListCollectionView Receivers { get; private set; }
public ListCollectionView Transmitters { get; private set; }

public Receiver ActiveReceiver
{
    get { return activeReceiver; }
    set { SetProperty(ref activeReceiver, value); }
}
public Transmitter ActiveSource
{
    get { return activeTransmitter; }
    set { SetProperty(ref activeTransmitter, value); }
}

public Boundary Boundary
{
    get { return boundary; }
    set { SetProperty(ref boundary, value); }
}
public decimal TimeStep
{
    get { return deltaT; }
    set { SetProperty(ref deltaT, value);
        if (deltaT != 0 && sampleRate > 1 / deltaT)
                SampleRate = (int)(1 / deltaT); }
}
public int SampleRate
{
    get { return sampleRate; }
    set { SetProperty(ref sampleRate, value);
        if (deltaT != 0 && sampleRate > 1 / deltaT) SampleRate = (int)(1 / deltaT); }
}
public int SimulationDuration { get; set; }
public int Seed { get; set; }
public int NumOfRuns
{
    get { return this.mumOfRuns; }
    set { SetProperty(ref this.mumOfRuns, value); }
}

public bool UseBounds
{
    get { return useBounds; }
    set
    {
        SetProperty(ref useBounds, value);
        if (value)
        {
            if (Boundary == null)
                Boundary = new Boundary(channel);
            else
            {
                Boundary.Dispose();
                Boundary = new Boundary(channel);
            }
        }
        else
        {
            if (Boundary != null)
            {
                Boundary.Dispose();
                Boundary = null;
            }
        }
    }
}

public BrownianChannel Channel
{
    get { return channel; }
```

```
        }

        public static DataManager DataManager;
        //------------------------------------------------------------
        #endregion
        //------------------------------------------------------------
        #region Public Methods
        //------------------------------------------------------------
        public void Run(SimulationStateNotification state)
        {
            DataManager = new DataManager(NumOfRuns, TimeStep);

            for (int i = 0; i < NumOfRuns; i++)
            {
                //Initializing
                if (i == 0)
                    random = new Random(Seed); //for the first run we always used the given
seed
                else
                    random = new Random();
                SetSimulationObjectProps(); //most come before ResetStates
                ResetStates();
                //Starting simulation loop
                while (simulationTime <= (SimulationDuration) && !state.Canceled) //there is
a small summation accurarcy and we need to deduct a small amount so it simulates last step
                {
                    channel.Update(simulationTime);// this will update receivers &
transmitters each time step
                    state.DisplayText = "Run# " + (i + 1) + "\nCurrent Time: " +
simulationTime.ToString("f3");
                    state.Progress = (double)Math.Round(100 * (simulationTime /
SimulationDuration), 1);
                    simulationTime += deltaT;//we start simulation from time 0
                }
                if (!state.Canceled)
                {
                    //state.Finished = true;
                    DataManager.CollectResult(receivers, i);
                }
            }

        var displayService = ServiceLocator.Current.GetInstance<IDisplayService>();
              displayService.CreateAnimation(Channel.Frames, Channel.MaxParticleCount,
        (float)SimulationDuration);

            state.Finished = true;
            OnSimulationFinished(null);
            //System.GC.Collect();
        }

        internal void AddReceiver()
        {
            var receiver = new Receiver(Channel, TimeStep, SimulationDuration);
            receivers.Add(receiver);
            receiver.Name = "Receiver#" + Receivers.Count.ToString();
            Receivers.MoveCurrentToLast();
        }

        internal void RemoveReceiver()
        {
            activeReceiver.Dispose();
            Receivers.Remove(activeReceiver);
        }

        internal void AddTransmitter()
        {
            var source = new Transmitter(Channel, TimeStep);
            transmitters.Add(source);
            source.Name = "Transmitter#" + Transmitters.Count.ToString();
            Transmitters.MoveCurrentToLast();
        }

        internal void RemoveTransmitter()
        {
```

```
            activeTransmitter.Dispose();
            Transmitters.Remove(activeTransmitter);
        }
        //------------------------------------------------------------
        #endregion
        //------------------------------------------------------------
        #region Private Methods
        //------------------------------------------------------------

        private void OnSimulationFinished(EventArgs e)
        {
            EventHandler simulationFinished = SimulationFinished;
            if (simulationFinished != null)
                simulationFinished(this, e);
        }

        void Receivers_CurrentChanged(object sender, EventArgs e)
        {
            ActiveReceiver = Receivers.CurrentItem as Receiver;
        }

        void Sources_CurrentChanged(object sender, EventArgs e)
        {
            ActiveSource = Transmitters.CurrentItem as Transmitter;
            if (ActiveSource != null)
                ActiveSource.Modulators.MoveCurrentToLast();
        }

        private void SetSimulationObjectProps()
        {
            Channel.SampleRate = SampleRate;
            Channel.SimulationDuration = SimulationDuration;
            Channel.TimeStep = TimeStep;
            Channel.Random = random;
            foreach (var receiver in receivers)
            {
                receiver.Channel = Channel;
                receiver.TimeStep = TimeStep;
                receiver.SimulationDuration = SimulationDuration;
            }
            foreach (var transmitter in transmitters)
            {
                transmitter.Channel = Channel;
                transmitter.TimeStep = TimeStep;
            }
        }

        private void ResetStates()
        {
            simulationTime = 0;
            Channel.ResetStates();
            foreach (var receiver in receivers)
            {
                receiver.ResetStates();
            }
            foreach (var transmitter in transmitters)
            {
                transmitter.ResetStates();
            }
            if (UseBounds)
                Boundary.ResetStates();
        }
        //------------------------------------------------------------
        #endregion

    }
```

The channel object source code:

```
public abstract class Channel : BindableBase, IDisposable
    {
        //--------------------------------------------------------------
        #region Fields
        //--------------------------------------------------------------
        List<IParticle> particles;
        List<IParticle> boundParticles;
        IDisplayService displayService;
        private int sampleRate;
        private int simulationDuration;
        private IFrame[] frames;
        private decimal RecordingTime;
        private int frameIndex;
        private decimal timeStep;
        private int maxParticleCount;
        private Random random;
        //--------------------------------------------------------------
        #endregion
        //--------------------------------------------------------------
        #region Constructor
        //--------------------------------------------------------------
        public Channel(int simulationDuration, int sampleRate, decimal
timeStep, Random random)
        {
            particles = new List<IParticle>();
            boundParticles = new List<IParticle>();
            displayService =
ServiceLocator.Current.GetInstance<IDisplayService>();
            this.simulationDuration = simulationDuration;
            this.sampleRate = sampleRate;
            this.timeStep = timeStep;
            this.RecordingTime = 1.0M / sampleRate;
            this.frameIndex = 0;
            this.random = random;
            //frames = new IFrame[simulationDuration * sampleRate];
            maxParticleCount = 0;
        }

        public void Dispose()
        {
            displayService = null;
            particles.Clear();
            BoundParticles.Clear();
            particles = null;
            frames = null;
            random = null;
        }
        #endregion
        //--------------------------------------------------------------
        #region Properties
        //--------------------------------------------------------------
        public event EventHandler ChannelUpdated;

        public Random Random
        {
            get { return random; }
            set { random = value; }
        }
```

```
        public decimal TimeStep
        {
            get { return timeStep; }
            set { timeStep = value; }
        }

        public int SimulationDuration
        {
            get { return simulationDuration; }
            set { simulationDuration = value; }
        }

        public int SampleRate
        {
            get { return sampleRate; }
            set { sampleRate = value; }
        }

        public List<IParticle> Particles
        {
            get { return particles; }
        }

        public List<IParticle> BoundParticles
        {
            get { return boundParticles; }
        }

        public IFrame[] Frames
        {
            get { return frames; }
        }

        public int MaxParticleCount
        {
            get { return maxParticleCount; }
            set { maxParticleCount = value; }
        }
        #endregion
        //------------------------------------------------------------
        #region Methods
        //------------------------------------------------------------
        public void Update(decimal simulationTime)
        {
            //first call should not move particles because simulation time is
zero. so this must come before OnChannelUpdated() and there are no particles
generated yet.
            MoveParticles();
            // let other objects do their things
            OnChannelUpdated();
            //Finally we record this frame.
            if (sampleRate > 0 && simulationTime >= (RecordingTime -
0.0000001M)) //needs correction for last frame to be recorded at end
simulation time
                RecordFrame();
            if (particles.Count > maxParticleCount)
                maxParticleCount = particles.Count;
        }

        public virtual void ResetStates()
        {
```

```
            particles.Clear();
            BoundParticles.Clear();
            this.RecordingTime = 1.0M / sampleRate;
            this.frameIndex = 0;
            frames = new IFrame[simulationDuration * sampleRate];
            maxParticleCount = 0;
        }

        private void OnChannelUpdated()
        {
            EventHandler handler = ChannelUpdated;
            if (handler != null) handler(this, EventArgs.Empty);
        }

        private void RecordFrame()
        {
            var frame = displayService.CreateFrame();
            frame.Particles = new IParticle[Particles.Count];
            for (int i = 0; i < frame.Particles.Length; i++)
            {
                // change it to deep copy
                frame.Particles[i] = new Particle()
                {
                    Binding = Particles[i].Binding,
                    DiffusionCoef = Particles[i].DiffusionCoef,
                    Last_X = Particles[i].Last_X,
                    Last_Y = Particles[i].Last_Y,
                    Last_Z = Particles[i].Last_Z,
                    ParticleColor = Particles[i].ParticleColor,
                    X = Particles[i].X,
                    Y = Particles[i].Y,
                    Z = Particles[i].Z,
                };
            }
            Frames[frameIndex] = frame;
            frameIndex++;
            RecordingTime += 1.0M / sampleRate;
        }
        protected virtual void MoveParticles() { }
        internal virtual void MoveParticleFromReceptor(IParticle particle,
DigitalRune.Mathematics.Algebra.Vector3F Location, float Radius) { }
        #endregion
        //---------------------------------------------------------------
    }


public class BrownianChannel : Channel
    {
        //---------------------------------------------------------------
        #region Fields
        //---------------------------------------------------------------
        NormalDistribution normalDist;
        //---------------------------------------------------------------
        #endregion
        //---------------------------------------------------------------
        #region Constructor
        //---------------------------------------------------------------
        public BrownianChannel(int simulationDuration, int sampleRate, decimal
timeStep, Random random)
            : base(simulationDuration, sampleRate, timeStep, random)
        {
```

```
            normalDist = new NormalDistribution();
            DriftX = 0;
            DriftY = 0;
            DriftZ = 0;
            Temperature = 310;
            Viscosity = .001f;
        }
        #endregion
        //---------------------------------------------------------------
        #region Properties
        //---------------------------------------------------------------
        public float DriftX { get; set; }
        public float DriftY { get; set; }
        public float DriftZ { get; set; }
        public double Viscosity { get; set; }
        public double Temperature { get; set; }
        #endregion
        //---------------------------------------------------------------
        #region Methods
        //---------------------------------------------------------------
        protected override void MoveParticles()
        {
            //base.MoveParticles(deltaTime);
            foreach (var particle in Particles)
            {
                if (!particle.BoundToReceptor)
                {
                    particle.Last_X = particle.X;
                    particle.Last_Y = particle.Y;
                    particle.Last_Z = particle.Z;
                    double dX = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftX * (double)TimeStep);
                    double dY = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftY * (double)TimeStep);
                    double dZ = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftZ * (double)TimeStep);
                    particle.X += dX;
                    particle.Y += dY;
                    particle.Z += dZ;
                }
            }
        }

        internal override void MoveParticleFromReceptor(IParticle particle,
DigitalRune.Mathematics.Algebra.Vector3F Location, float Radius)
        {
            double distance, dX, dY, dZ;
            particle.Last_X = particle.X;
            particle.Last_Y = particle.Y;
            particle.Last_Z = particle.Z;
            do
            {
                dX = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftX * (double)TimeStep);
                dY = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftY * (double)TimeStep);
                dZ = (normalDist.Next(Random) * Math.Sqrt(2 *
particle.DiffusionCoef * (double)TimeStep) + DriftZ * (double)TimeStep);
                double distX = particle.X + dX - Location.X;
                double distY = particle.Y + dY - Location.Y;
                double distZ = particle.Z + dZ - Location.Z;
                distance = distX * distX + distY * distY + distZ * distZ;
```

```
            } while (distance <= Radius * Radius);
            particle.X += dX;
            particle.Y += dY;
            particle.Z += dZ;
        }

        private void Job(int startIndex, int count)
        {
            for (int i = startIndex; i < startIndex + count; i++)
            {
                Particles[i].Last_X = Particles[i].X;
                Particles[i].Last_Y = Particles[i].Y;
                Particles[i].Last_Z = Particles[i].Z;
                double dX = (normalDist.Next(Random) * Math.Sqrt(2 *
Particles[i].DiffusionCoef * (double)TimeStep) + DriftX * (double)TimeStep);
                double dY = (normalDist.Next(Random) * Math.Sqrt(2 *
Particles[i].DiffusionCoef * (double)TimeStep) + DriftY * (double)TimeStep);
                double dZ = (normalDist.Next(Random) * Math.Sqrt(2 *
Particles[i].DiffusionCoef * (double)TimeStep) + DriftZ * (double)TimeStep);
                Particles[i].X += dX;
                Particles[i].Y += dY;
                Particles[i].Z += dZ;
            }
        }

        public override void ResetStates()
        {
            base.ResetStates();
        }
        #endregion
        //------------------------------------------------------------------

    }
```

The transmitter object source code:

```
public class Transmitter : StaticNanoObject, IDisposable
    {
        //------------------------------------------------------------------
        #region Fields
        //------------------------------------------------------------------
        private Color color;
        private Color particleColor;
        private int binding;
        private float radius;
        private Channel channel;
        private decimal timeStep;
        private Modulation activeModulator;
        private ObservableCollection<Modulation> modulators;
        private DigitalRune.Mathematics.Statistics.SphereDistribution
sphereDist;
        //------------------------------------------------------------------
        #endregion
        //------------------------------------------------------------------
        #region Constructor
        //------------------------------------------------------------------
        public Transmitter(Channel channel, decimal timeStep)
        {
```

```
            CreateModel();
            this.channel = channel;
            channel.ChannelUpdated += channel_ChannelUpdated;
            this.timeStep = timeStep;
            sphereDist = new
DigitalRune.Mathematics.Statistics.SphereDistribution();
            Location = new Vector3F(5, 15, 15);
            binding = 1;
            radius = 0.0f;
            activeModulator = new Modulation(this);
            modulators = new ObservableCollection<Modulation>();
            Modulators = new ListCollectionView(modulators);
            Modulators.CurrentChanged += Modulators_CurrentChanged;
            modulators.Add(activeModulator);
            particleColor = Colors.Green;
            sphereDist.InnerRadius = 0;
            sphereDist.OuterRadius = radius;
        }

        public void ResetStates()
        {
            ActiveModulator.ResetStates();
        }

        public void Dispose()
        {
            displayService.RemoveSceneNode(model);
            displayService = null;
            channel.ChannelUpdated -= channel_ChannelUpdated;
            channel = null;
            foreach (var modulator in modulators)
                modulator.Dispose();
        }
        //-----------------------------------------------------------------
        #endregion
        //-----------------------------------------------------------------
        #region Properties & Events
        //-----------------------------------------------------------------
        public decimal TimeStep
        {
            get { return timeStep; }
            set { timeStep = value; }
        }

        internal Channel Channel
        {
            get { return channel; }
            set { channel = value; }
        }

        public ListCollectionView Modulators { get; private set; }
        public Modulation ActiveModulator
        {
            get { return activeModulator; }
            set { SetProperty(ref activeModulator, value); }
        }

        public override Vector3F Location
        {
            get
            {
                return model.PoseWorld.Position;
```

```
            }
            set
            {
                model.PoseWorld = new Pose(value);

                OnPropertyChanged(() => this.Location_X);
                OnPropertyChanged(() => this.Location_Y);
                OnPropertyChanged(() => this.Location_Z);
                sphereDist.Center = value;
            }
        }

        public float Location_X
        {
            get { return Location.X; }
            set { Location = new Vector3F(value, Location.Y, Location.Z); }
        }
        public float Location_Y
        {
            get { return Location.Y; }
            set { Location = new Vector3F(Location.X, value, Location.Z); }
        }
        public float Location_Z
        {
            get { return Location.Z; }
            set { Location = new Vector3F(Location.X, Location.Y, value); }
        }

        public Color Color
        {
            get { return color; }
            set
            {
                SetProperty(ref this.color, value);
                displayService.SetColor(model, color);
            }
        }

        public Color ParticleColor
        {
            get { return particleColor; }
            set
            {
                SetProperty(ref this.particleColor, value);
            }
        }

        public int Binding
        {
            get { return binding; }
            set { SetProperty(ref binding, value); }
        }

        public float Radius
        {
            get { return radius; }
            set { SetProperty(ref radius, value); sphereDist.OuterRadius =
value; }
        }

        internal void ReleaseParticles(int particleCount)
        {
```

```
            double diffCoef = 1.38e-23 * (channel as
BrownianChannel).Temperature / (6 * Math.PI * (channel as
BrownianChannel).Viscosity * ActiveModulator.ParticleRadius* 1e-9);
            diffCoef *= 1e12;
            float x = Location.X;
            float y = Location.Y;
            float z = Location.Z;

            for (int i = 0; i < particleCount; i++)
            {
                if (Radius > 0)
                {
                    var loc = sphereDist.Next(Channel.Random);
                    x = loc.X;
                    y = loc.Y;
                    z = loc.Z;
                }
                Particle particle = new Particle()
                {
                    Binding = this.Binding,
                    DiffusionCoef = diffCoef,
                    X = x,
                    Y = y,
                    Z = z,
                    Last_X = x,
                    Last_Y = y,
                    Last_Z = z,
                    Radius = ActiveModulator.ParticleRadius,
                    ParticleColor = this.ParticleColor
                };
                channel.Particles.Add(particle);
            }
        }
        //------------------------------------------------------------------
        #endregion
        //------------------------------------------------------------------
        #region Methods
        //------------------------------------------------------------------
        private void CreateModel()
        {
            color = new Color()
            {
                A = 196,
                R = 196,
                B = 64,
                G = 64,
            };
            particleColor = new Color()
            {
                A = 255,
                R = 32,
                B = 32,
                G = 32,
            };
            model = displayService.AddEllipse(new Vector2F(0.5f, 0.5f),
Color.FromRgb(128, 64, 64), color, true);
        }

        void channel_ChannelUpdated(object sender, EventArgs e)
        {
            activeModulator.Update();
        }
```

```
        void Modulators_CurrentChanged(object sender, EventArgs e)
        {
            ActiveModulator = Modulators.CurrentItem as Modulation;
        }
        //------------------------------------------------------------
        #endregion
    }
```

The receiver object source code:

```
public class Receiver : StaticNanoObject, IDisposable
    {
        #region Fields
        //------------------------------------------------------------
        private Stack<ReceptorArea.Coordinate> presetLocations;
        private Color color;
        private ReceptorArea activeReceptor;
        private bool isTransparent;
        private float coefOfRestitution;
        private float radius;
        private Channel channel;
        private decimal timeStep;
        private int simulationDuration;
        private decimal simulationTime;
        private Dictionary<ReceptorArea, decimal> currentNumOfBounds;
        private float[] totalReceiverBounds;
        private int stepIndex;
        //------------------------------------------------------------
        #endregion
        //------------------------------------------------------------
        #region Constructor
        //------------------------------------------------------------
        public Receiver(Channel channel, decimal timeStep, int
simulationDuration)
        {
            CreateModel();
            ReceptorAreas = new ObservableCollection<ReceptorArea>();
            ReceptorsView = new ListCollectionView(ReceptorAreas);
            ReceptorsView.CurrentChanged += ReceptorsView_CurrentChanged;
            Location = new Vector3F(20, 15, 15);
            Radius = 4;
            coefOfRestitution = 0.8f;
            isTransparent = false;
            presetLocations = InitLocations();
            this.channel = channel;
            channel.ChannelUpdated += channel_ChannelUpdated;
            this.timeStep = timeStep;
            simulationTime = 0; //
            this.simulationDuration = simulationDuration;
            stepIndex = -1;
        }

        public void Dispose()
        {
            foreach (ReceptorArea receptor in ReceptorAreas)
```

```
            {
                receptor.Dispose();
            }
            //Receptors.Clear();
            displayService.RemoveSceneNode(model);
            displayService = null;
            presetLocations.Clear();
            channel.ChannelUpdated -= channel_ChannelUpdated;
            channel = null;
            totalReceiverBounds = null;
            currentNumOfBounds = null;
        }
        //------------------------------------------------------------
        #endregion
        //------------------------------------------------------------
        #region Properties & Events
        //------------------------------------------------------------
        public ObservableCollection<ReceptorArea> ReceptorAreas { get; private
set; }
        public ListCollectionView ReceptorsView { get; private set; }

        public int SimulationDuration
        {
            get { return simulationDuration; }
            set { simulationDuration = value; }
        }
        public decimal TimeStep
        {
            get { return timeStep; }
            set { timeStep = value; }
        }
        public Channel Channel
        {
            get { return channel; }
            set { channel = value; }
        }

        public float CoefOfRestitution
        {
            get { return coefOfRestitution; }
            set { SetProperty(ref coefOfRestitution, value); }
        }

        public ReceptorArea ActiveReceptor
        {
            get { return activeReceptor; }
            set { SetProperty(ref this.activeReceptor, value); }
        }

        public bool IsTransparent
        {
            get { return isTransparent; }
            set
            {
                SetProperty(ref this.isTransparent, value);
                if (value)
                    displayService.MakeTransparent(this.model, .70f);
               else
                    displayService.MakeOpaque(this.model);
            }
        }
```

```
public override Vector3F Location
{
    get
    {
        return model.PoseWorld.Position;
    }
    set
    {
        model.PoseWorld = new Pose(value);
        foreach (ReceptorArea receptor in ReceptorAreas)
        {
            receptor.UpdateSphereProperties(Radius, Location);
        }
        OnPropertyChanged(() => this.Location_X);
        OnPropertyChanged(() => this.Location_Y);
        OnPropertyChanged(() => this.Location_Z);
    }
}

public float Location_X
{
    get { return Location.X; }
    set { Location = new Vector3F(value, Location.Y, Location.Z);  }
}
public float Location_Y
{
    get { return Location.Y; }
    set { Location = new Vector3F(Location.X, value, Location.Z); }
}
public float Location_Z
{
    get { return Location.Z; }
    set { Location = new Vector3F(Location.X, Location.Y, value); }
}

public float Radius
{
    get { return radius; }
    set
    {
        radius = value;
        displayService.SetSize(model, radius);
        foreach (ReceptorArea receptor in ReceptorAreas)
        {
            receptor.UpdateSphereProperties(radius, Location);
        }
    }
}

public Color Color
{
    get { return color; }
    set
    {
        SetProperty(ref this.color, value);
        displayService.SetColor(model, color);
    }
}

public float[] Concentration { get { return totalReceiverBounds; } }
//------------------------------------------------------------
#endregion
```

```
        //-------------------------------------------------------------
        #region Methods & Event Handlers
        //-------------------------------------------------------------
        public void AddReceptor()
        {
            ReceptorArea receptor = null;
            if (presetLocations.Count > 0)
            {
                var receptorCoord = presetLocations.Pop();
                receptor = new ReceptorArea(receptorCoord, Radius, Location,
(int)(simulationDuration / timeStep + 1)) { Name = "Receptor Area#" +
(ReceptorAreas.Count + 1).ToString() };
                ActiveReceptor = receptor;
                ReceptorAreas.Add(receptor);
                model.Children.Add(receptor.Decal);
                //model.Parent.Children.Add(receptor.Decal);
                receptor.UpdateSphereProperties(Radius, Location);
            }
            else
            {
                receptor = new ReceptorArea(new ReceptorArea.Coordinate() {
Latitude = 0, Longtitude = 90, Aperture = 60 }, Radius, Location,
(int)(simulationDuration / timeStep + 1)) { Name = "Receptor Area#" +
(ReceptorAreas.Count + 1).ToString() };
                ReceptorAreas.Add(receptor);
                ActiveReceptor = receptor;
                model.Children.Add(receptor.Decal);
            }
            ReceptorsView.MoveCurrentToLast();
        }

        public void RemoveReceptor()
        {
            ReceptorArea receptor = ReceptorsView.CurrentItem as ReceptorArea;
            ReceptorAreas.Remove(receptor);
            presetLocations.Push(receptor.Coordinates); //pushing back freed
location to the collection
            receptor.Dispose();
        }

        public void ResetStates()
        {
            stepIndex = -1;
            simulationTime = 0;
            int simulationSteps = (int)(simulationDuration / timeStep + 1);
            totalReceiverBounds = new float[simulationSteps];//+1 is for start
recording at simulation time 0
            currentNumOfBounds = new Dictionary<ReceptorArea, decimal>();
            foreach (var receptor in ReceptorAreas)
            {
                receptor.ResetStates(simulationSteps);
                currentNumOfBounds[receptor] = 0;
            }

        }

        private void ReceptorsView_CurrentChanged(object sender, EventArgs e)
        {
            ActiveReceptor = ReceptorsView.CurrentItem as ReceptorArea;
        }

        private void channel_ChannelUpdated(object sender, EventArgs e)
```

```
        {
            stepIndex++;
            foreach (ReceptorArea receptor in ReceptorAreas)
            {
                receptor.Update(simulationTime);
            }
            CheckForCollision();
            simulationTime += timeStep; //it comes last because first update
is for simulation time 0;
        }

        private void CheckForCollision()
        {
            //List<IParticle> particlesToRemove = new List<IParticle>();

            foreach (var particle in channel.Particles)
            {
                bool collidedWithReceptor = false;
                double distX = particle.X - Location.X;
                double distY = particle.Y - Location.Y;
                double distZ = particle.Z - Location.Z;
                double distance = distX * distX + distY * distY + distZ *
distZ;
                //first we check if this particle hits the receiver (sphere)
                if (distance <= Radius * Radius)
                {
                    if (IsTransparent) //in this case particles simply move
without any reflection or absorbtion
                    {
                        totalReceiverBounds[stepIndex]++;
                    }
                    else
                    {
                        //first we calculate collision point
                        Vector3D collisionP = CalcCollisionPoint(particle);
                        //in this case we need to check f particle hits one of
the receptors on the receiver
                        foreach (var receptorArea in ReceptorAreas)
                        {
                            //first we calculate the angle between patch
center and particle vector originated from sphere center
                            double teta =
MathHelper.ToDegrees(Vector3D.GetAngle((collisionP - Location),
receptorArea.Center));
                            if (teta < receptorArea.Aperture / 2)
                            { //particle collided with this patch
                                receptorArea.Hits++;
                                if (receptorArea.Binding != particle.Binding)
                                {
                                    ReflectParticle(particle, collisionP);
                                }
                                else //check if collision results in a bound
between free receptor and particle
                                {
                                    if (CheckForBound(particle, receptorArea))
                                    {

receptorArea.BoundReceptores[stepIndex]++;
                                        particle.BoundToReceptor = true;
                                        particle.X = collisionP.X;
                                        particle.Y = collisionP.Y;
                                        particle.Z = collisionP.Z;
```

```
                                    channel.BoundParticles.Add(particle);
                                }
                                else
                                {
                                    receptorArea.Missses++;
                                    ReflectParticle(particle, collisionP);
//in this case particle continues its BMotion
                                }
                            }
                            //now that particle collided with this
receptor, no need to check other patches
                            collidedWithReceptor = true;
                            break;
                        }
                    }
                    //finally if partile did not hit any receptors it will
reflect from surface of sphere.
                    if (!collidedWithReceptor)
                        ReflectParticle(particle, collisionP);
                }
            }
            //now we check for released bounds
            foreach (var receptorArea in ReceptorAreas)
            {
                currentNumOfBounds[receptorArea] =
currentNumOfBounds[receptorArea] +
(decimal)receptorArea.BoundReceptores[stepIndex] -
(decimal)receptorArea.ReleaseRate * timeStep *
(currentNumOfBounds[receptorArea] +
(decimal)receptorArea.BoundReceptores[stepIndex] / 2);
                receptorArea.BoundReceptores[stepIndex] =
(float)Math.Round(currentNumOfBounds[receptorArea]);
                totalReceiverBounds[stepIndex] +=
receptorArea.BoundReceptores[stepIndex];
                if (stepIndex > 0)
                {
                    float releasedParticles =
receptorArea.BoundReceptores[stepIndex - 1] -
receptorArea.BoundReceptores[stepIndex];
                    if (releasedParticles > 0)
                    {
                        //in this case some particles released and we must
update channel
                        for (int i = 0; i < releasedParticles; i++)
                        {
                            channel.BoundParticles[0].BoundToReceptor = false;
                            //after particles released they will continue
their Brownian motion

channel.MoveParticleFromReceptor(channel.BoundParticles[0], Location, Radius);
                            channel.BoundParticles.RemoveAt(0);
                        }
                    }
                }
            }
        }

        private bool CheckForBound(IParticle particle, ReceptorArea
receptorArea)
        {
```

```
            int boundReceptors = (int)receptorArea.BoundReceptores[stepIndex]
+ stepIndex > 0 ? (int)receptorArea.BoundReceptores[stepIndex - 1] : 0;
            double hitProb = receptorArea.CalcHitProbability(particle.Radius,
boundReceptors);
            double reactionProb = receptorArea.ReactionProbability;
            double boundProbability = hitProb * reactionProb;
            bool reactionSucceed = (Channel.Random.NextDouble() <=
boundProbability);
            return reactionSucceed;
        }

        private void ReflectParticle(IParticle particle, Vector3D collisionP)
        {
            Vector3D dist_AfterCollision = new Vector3D(particle.X -
collisionP.X, particle.Y - collisionP.Y, particle.Z - collisionP.Z);
            Vector3D reflectionVector = CalcReflectVector(dist_AfterCollision,
collisionP);
            var reflectedPoint = collisionP + reflectionVector;
            particle.Last_X = collisionP.X;
            particle.Last_Y = collisionP.Y;
            particle.Last_Z = collisionP.Z;
            particle.X = reflectedPoint.X;
            particle.Y = reflectedPoint.Y;
            particle.Z = reflectedPoint.Z;
        }

        private Vector3D CalcCollisionPoint(IParticle particle)
        {
            // first we will calculate point of collision on the sphere
            Vector3D p1, p2;
            p1 = new Vector3D() { X = particle.Last_X - Location.X, Y =
particle.Last_Y - Location.Y, Z = particle.Last_Z - Location.Z };
            p2 = new Vector3D() { X = particle.X - Location.X, Y = particle.Y
- Location.Y, Z = particle.Z - Location.Z };
            double a = (p2.Z - p1.Z) * (p2.Z - p1.Z) + (p2.Y - p1.Y) * (p2.Y -
p1.Y) + (p2.X - p1.X) * (p2.X - p1.X);
            double b = 2 * (p1.X * (p2.X - p1.X) + p1.Y * (p2.Y - p1.Y) + p1.Z
* (p2.Z - p1.Z));
            double c = p1.X * p1.X + p1.Y * p1.Y + p1.Z * p1.Z - Radius *
Radius;
            double t1 = (-b + (float)Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
            double t2 = (-b - (float)Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
            Vector3D cp1 = (p2 - p1) * t1 + p1; //first calculated point
            Vector3D cp2 = (p2 - p1) * t2 + p1; //second calculated point
            // now we have to check which of this collision points is the
valid point
            //Valid point is the one that is between p1 and p2
            Vector3D collisionPoint = cp1 + Location; //we assume that cp1 is
the answer and check if it is the otherwise
            if (p1.X < p2.X)
            {
                if (cp1.X >= p2.X || cp1.X <= p1.X)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
                }
            }
            else
                if (cp1.X >= p1.X || cp1.X <= p2.X)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
```

```
                }
            if (p1.Y < p2.Y)
            {
                if (cp1.Y >= p2.Y || cp1.Y <= p1.Y)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
                }
            }
            else
                if (cp1.Y >= p1.Y || cp1.Y <= p2.Y)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
                }
            if (p1.Z < p2.Z)
            {
                if (cp1.Z >= p2.Z || cp1.Z <= p1.Z)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
                }
            }
            else
                if (cp1.Z >= p1.Z || cp1.Z <= p2.Z)
                {
                    collisionPoint = cp2 + Location;
                    return collisionPoint;
                }
            return collisionPoint;
        }

        private Vector3D CalcReflectVector(Vector3D direction, Vector3D
collisionPoint)
        {
            Vector3D n = (collisionPoint - Location).Normalized; //for the
sphere
            Vector3D projection_n = Vector3D.Dot(n, direction) * n;
            Vector3D projection_d = direction - projection_n;
            Vector3D reflectionVector = projection_d - CoefOfRestitution *
projection_n;
            return reflectionVector;
        }

        private Stack<ReceptorArea.Coordinate> InitLocations()
        {
            Stack<ReceptorArea.Coordinate> presetLocations = new
Stack<ReceptorArea.Coordinate>();

            ReceptorArea.Coordinate location14 = new ReceptorArea.Coordinate()
{ Longtitude = 0, Latitude = -90, Aperture = 60 };
            presetLocations.Push(location14);

            ReceptorArea.Coordinate location13 = new ReceptorArea.Coordinate()
{ Longtitude = 0, Latitude = 90, Aperture = 60 };
            presetLocations.Push(location13);

            for (int i = 3; i >= 0; i--)
            {
                ReceptorArea.Coordinate location = new
ReceptorArea.Coordinate() { Longtitude = 45 + i * 90, Latitude = -45, Aperture
= 60 };
```

```
                presetLocations.Push(location);
            }

            for (int i = 3; i >= 0; i--)
            {
                ReceptorArea.Coordinate location = new
ReceptorArea.Coordinate() { Longtitude = 45 + i * 90, Latitude = 45, Aperture
= 60 };
                presetLocations.Push(location);
            }

            for (int i = 3; i >= 0; i--)
            {
                ReceptorArea.Coordinate location = new
ReceptorArea.Coordinate() { Longtitude = i * 90, Latitude = 0, Aperture = 60
};
                presetLocations.Push(location);
            }

            return presetLocations;
        }

        private void CreateModel()
        {
            color = new Color()
            {
                A = 255,
                R = 30,
                B = 240,
                G = 240,
            };
            this.model = displayService.AddSphere(Radius, color);
        }
        //------------------------------------------------------------------
        #endregion




        //------------------------------------------------------------------
        #region Internal types
        //------------------------------------------------------------------
        public class ReceptorArea : BindableBase ,IDisposable
        {
            //Fields
            private float[] boundReceptores;
            private float receiverRadius;
            private Vector3F sphereLocation;
            private decimal simulationTime = 0;
            private IDisplayService displayService;
            private float coefOfRestitution;
            private float receptorRadius;
            private float reactionProbability;
            private float releaseRate;
            private uint totalReceptors;
            private int binding;

            //------------------------------------------------------------------
            #region Constructor
            //------------------------------------------------------------------
```

```
            public ReceptorArea(Coordinate coordinate, float sphereRadius,
Vector3F sphereLocation, int stepCounts)
            {
                this.receiverRadius = sphereRadius;
                this.sphereLocation = sphereLocation;
                displayService =
ServiceLocator.Current.GetInstance<IDisplayService>();
                Decal = displayService.GetDecalNode();
                this.Coordinates = coordinate; // needs to be saved seperately
to maintain preset locations
                this.Latitude = coordinate.Latitude;
                this.Longtitude = coordinate.Longtitude;
                this.Aperture = coordinate.Aperture;
                totalReceptors = 100;
                binding = 1;
                coefOfRestitution = 0.8f;
                reactionProbability = 1;
                receptorRadius = 5;
                releaseRate = 0.1f;
                //concentration = new int[stepCounts];
            }
            //-----------------------------------------------------------------
            #endregion
            //-----------------------------------------------------------------
            #region Properties
            //-----------------------------------------------------------------
            public string Name { get; set; }
            public float[] BoundReceptores { get { return boundReceptores; } }
            public Vector3F Center { get; private set; }
            public Vector3F GlobalCenter { get; private set; }
            public float Diameter { get; private set; }
            public DecalNode Decal { get; private set; }
            private float latitude;
            public float Latitude { get { return
MathHelper.ToDegrees(latitude); } set { SetProperty(ref latitude,
MathHelper.ToRadians(value)); SetDecalProperties(); } }
            private float longtitude;
            public float Longtitude { get { return
MathHelper.ToDegrees(longtitude); } set { SetProperty(ref longtitude,
MathHelper.ToRadians(value)); SetDecalProperties(); } }
            private float aperture;
            public float Aperture { get { return
MathHelper.ToDegrees(aperture); } set { SetProperty(ref aperture,
MathHelper.ToRadians(value)); SetDecalProperties(); OnPropertyChanged(() =>
this.CoveredArea); } }
            public int Hits { get; set; }
            public int Missses { get; set; }
            public Coordinate Coordinates { get; private set; }
            public uint TotalReceptors
            {
                get { return totalReceptors; }
                set { SetProperty(ref totalReceptors, value);
OnPropertyChanged(() => this.CoveredArea); }
            }
            public int Binding
            {
                get { return binding; }
                set { SetProperty(ref binding, value); }
            }
            public float CoefOfRestitution
            {
                get { return coefOfRestitution; }
```

```
                set { SetProperty(ref coefOfRestitution, value); }
            }

            public float ReactionProbability
            {
                get { return reactionProbability; }
                set { SetProperty(ref reactionProbability, value); }
            }

            public float ReceptorRadius
            {
                get { return receptorRadius; }
                set { SetProperty(ref receptorRadius, value);
OnPropertyChanged(() => this.CoveredArea); }
            }

            public float ReleaseRate
            {
                get { return releaseRate; }
                set { SetProperty(ref releaseRate, value);}
            }

            public string CoveredArea
            {
                get
                {
                    return (TotalReceptors * Math.PI *
Math.Pow(ReceptorRadius, 2) / (2 * Math.PI * Math.Pow(receiverRadius * 1e+3,
2) * (1 - Math.Cos(aperture / 2)))).ToString("f5"); ;
                }
            }

            internal double CalcHitProbability(double particleRadius, int
boundedReceptors)
            {
                double hitProb = (TotalReceptors - boundedReceptors) * Math.PI
* Math.Pow((ReceptorRadius + particleRadius), 2)/ (2 * Math.PI *
Math.Pow(receiverRadius * 1e+3, 2) * (1 - Math.Cos(aperture / 2)));
                return hitProb;
            }

            internal void ResetStates(int simulationSteps)
            {
                boundReceptores = new float[simulationSteps];
                Hits = 0;
                Missses = 0;
            }

            public void Dispose()
            {
                displayService.RemoveSceneNode(Decal);
                Decal = null;
                displayService = null;
                Coordinates = null;
            }
            #endregion
            //-----------------------------------------------------------------
            private void SetDecalProperties()
            {
                if (Decal.Parent != null)
                {
                    CalcCenterofPatch();
```

```
                    CalcDiameter();
                    QuaternionF orientation =
QuaternionF.CreateRotation(ConstantsF.PiOver2 - longtitude, Vector3F.Up, -
latitude, Decal.PoseLocal.ToWorldDirection(Vector3F.Right), 0, Vector3F.UnitZ,
true);
                    //we need to rescale Decal sizes and position because they
are scaled by decal parrent receiver node.
                    Decal.PoseLocal = new Pose(Center /
Decal.Parent.ScaleLocal.X * 1.05f, orientation);
                    Decal.Width = Diameter / Decal.Parent.ScaleLocal.X;
                    Decal.Height = Diameter / Decal.Parent.ScaleLocal.X;
                    Decal.Depth = Decal.Height;
                }
            }

            private void CalcDiameter()
            {
                this.Diameter = (float)(2 * receiverRadius * Math.Sin(aperture
/ 2));
            }

            private void CalcCenterofPatch()
            {
                float y = (float)(Math.Sin(latitude) * receiverRadius);
                float z = (float)(Math.Cos(latitude) * Math.Sin(longtitude) *
receiverRadius);
                float x = (float)(Math.Cos(latitude) * Math.Cos(longtitude) *
receiverRadius);
                this.Center = new Vector3F(x, y, z);
                this.GlobalCenter = sphereLocation + Center;
            }

            internal void UpdateSphereProperties(float sphereRadius, Vector3F
sphereLocation)
            {
                this.receiverRadius = sphereRadius;
                this.sphereLocation = sphereLocation;
                SetDecalProperties();
                OnPropertyChanged(() => this.CoveredArea);
            }
            //Used for updating simulationTime and patch logic
            internal void Update(decimal simulationTime)
            {
                this.simulationTime = simulationTime; //Updates local
simulation Time
            }

            //-----------------------------------------------------------------
            public class Coordinate
            {
                public float Latitude { get; set; }
                public float Longtitude { get; set; }
                public float Aperture { get; set; }
            }
        }
        #endregion
    }
```

The boundary object source code:

```
public class Boundary : BindableBase, IDisposable
    {
        //--------------------------------------------------------------
        #region Fields
        //--------------------------------------------------------------
        FigureNode model;
        IDisplayService displayService;
        Vector3F dimension;
        float coefOfRestitution;
        bool leftAbsorbs;
        bool rightAbsorbs;
        bool topAbsorbs;
        bool buttomAbsorbs;
        bool frontAbsorbs;
        bool backAbsorbs;
        private Channel channel;
        private enum Plane
        {
            Left,
            Right,
            Top,
            Buttom,
            Front,
            Back
        }
        private bool[] collisionPlane;
        //--------------------------------------------------------------
        #endregion
        //--------------------------------------------------------------
        #region Constructor
        //--------------------------------------------------------------
        public Boundary(Channel channel)
        {
            displayService =
ServiceLocator.Current.GetInstance<IDisplayService>();
            Dimension = new Vector3F(30f, 30f, 30f);
            coefOfRestitution = 0.8f;
            collisionPlane = new bool[6];
            this.channel = channel;
            channel.ChannelUpdated += channel_ChannelUpdated;
            CreateModel(); // this goes last due to Dimension needs to be set
first
        }

        public void Dispose()
        {
            displayService.RemoveSceneNode(model);
            displayService = null;
            channel.ChannelUpdated -= channel_ChannelUpdated;
            channel = null;
        }

        public void ResetStates()
        {
            for (int i = 0; i < collisionPlane.Length; i++)
            {
                collisionPlane[i] = false;
            }
```

```
}
//------------------------------------------------------------
#endregion
//------------------------------------------------------------
#region Properties & Events
//------------------------------------------------------------
public float CoefOfRestitution
{
    get { return coefOfRestitution; }
    set { SetProperty(ref coefOfRestitution, value); }
}

public bool LeftAbsorbs
{
    get { return leftAbsorbs; }
    set { SetProperty(ref leftAbsorbs, value); }
}
public bool RightAbsorbs
{
    get { return rightAbsorbs; }
    set { SetProperty(ref rightAbsorbs, value); }
}
public bool TopAbsorbs
{
    get { return topAbsorbs; }
    set { SetProperty(ref topAbsorbs, value); }
}
public bool ButtomAbsorbs
{
    get { return buttomAbsorbs; }
    set { SetProperty(ref buttomAbsorbs, value); }
}
public bool FrontAbsorbs
{
    get { return frontAbsorbs; }
    set { SetProperty(ref frontAbsorbs, value); }
}
public bool BackAbsorbs
{
    get { return backAbsorbs; }
    set { SetProperty(ref backAbsorbs, value); }
}

public Vector3F Location
{
    get
    {
        return model.PoseWorld.Position;
    }
    set
    {
        model.PoseWorld = new Pose(value);
        OnPropertyChanged(() => this.Location_X);
        OnPropertyChanged(() => this.Location_Y);
        OnPropertyChanged(() => this.Location_Z);
    }
}
public float Location_X
{
    get { return Location.X; }
    set { Location = new Vector3F(value, Location.Y, Location.Z); }
}
```

```
        public float Location_Y
        {
            get { return Location.Y; }
            set { Location = new Vector3F(Location.X, value, Location.Z); }
        }
        public float Location_Z
        {
            get { return Location.Z; }
            set { Location = new Vector3F(Location.X, Location.Y, value); }
        }

        public Vector3F Dimension
        {
            get
            {
                return dimension;
            }
            set
            {
                dimension = value;
                OnPropertyChanged(() => this.Dimension_X);
                OnPropertyChanged(() => this.Dimension_Y);
                OnPropertyChanged(() => this.Dimension_Z);
            }
        }
        public float Dimension_X
        {
            get { return dimension.X; }
            set { dimension = new Vector3F(value, dimension.Y, dimension.Z);
CreateModel(); }
        }
        public float Dimension_Y
        {
            get { return dimension.Y; }
            set { dimension = new Vector3F(dimension.X, value, dimension.Z);
CreateModel(); }
        }
        public float Dimension_Z
        {
            get { return dimension.Z; }
            set { dimension = new Vector3F(dimension.X, dimension.Y, value);
CreateModel(); }
        }
        //-----------------------------------------------------------------
        #endregion

        //-----------------------------------------------------------------
        #region Methods
        //-----------------------------------------------------------------
        private void CreateModel()
        {
            if (model != null)
                displayService.RemoveSceneNode(model);
            var color = new System.Windows.Media.Color()
            {
                A = 32,
                R = 240,
                B = 240,
                G = 240,
            };
            model = displayService.AddBoundary(Dimension, color);
        }
```

```
        void channel_ChannelUpdated(object sender, EventArgs e)
        {
            CheckForCollision();
        }

        private void CheckForCollision()
        {
            List<IParticle> particlesToRemove = new List<IParticle>();

            foreach (var particle in channel.Particles)
            {
                //first we check if this particle hits the boundary (cubic)
                if (ParticleCrossEdges(particle))
                {
                    //first we calculate collision point
                    Vector3D p2 = new Vector3D(particle.X, particle.Y,
particle.Z);
                    Vector3D p1 = new Vector3D(particle.Last_X,
particle.Last_Y, particle.Last_Z);
                    Vector3D collisionP = CalcCollisionPoint(p1, p2);
                    if (ParticleHitsAbsorbingPlane())
                        particlesToRemove.Add(particle);
                    else
                        ReflectParticle(particle, collisionP);
                }
            }
            //here we need to remove absorbed particles from the channel
            foreach (var particle in particlesToRemove)
                channel.Particles.Remove(particle);
        }

        private bool ParticleHitsAbsorbingPlane()
        {
            Plane collidedPlane = Plane.Top;
            for (int i = 0; i < collisionPlane.Length; i++)
            {
                if (collisionPlane[i])
                {
                    collidedPlane = (Plane)i;
                    break;
                }
            }
            switch (collidedPlane)
            {
                case Plane.Top:
                    if (TopAbsorbs)
                        return true;
                    else
                        return false;
                    break;
                case Plane.Buttom:
                    if (ButtomAbsorbs)
                        return true;
                    else
                        return false;
                    break;
                case Plane.Left:
                    if (LeftAbsorbs)
                        return true;
                    else
                        return false;
```

```
                    break;
                case Plane.Right:
                    if (RightAbsorbs)
                        return true;
                    else
                        return false;
                    break;
                case Plane.Back:
                    if (BackAbsorbs)
                        return true;
                    else
                        return false;
                    break;
                case Plane.Front:
                    if (FrontAbsorbs)
                        return true;
                    else
                        return false;
                    break;
            }
            return false;
        }

        private bool ParticleCrossEdges(IParticle particle)
        {
            var edge = Location + Dimension;
            //inside => outside
            if (particle.X < Location.X || particle.X > edge.X)
                if (particle.Last_X > Location.X && particle.Last_X < edge.X)
                    return true;
            if (particle.Y < Location.Y || particle.Y > edge.Y)
                if (particle.Last_Y > Location.Y && particle.Last_Y < edge.Y)
                    return true;
            if (particle.Z < Location.Z || particle.Z > edge.Z)
                if (particle.Last_Z > Location.Z && particle.Last_Z < edge.Z)
                    return true;
            //outside -> inside
            if (particle.X > Location.X && particle.X < edge.X)
                if (particle.Last_X < Location.X || particle.Last_X > edge.X)
                    return true;
            if (particle.Y > Location.Y && particle.Y < edge.Y)
                if (particle.Last_Y < Location.Y || particle.Last_Y > edge.Y)
                    return true;
            if (particle.Z > Location.Z && particle.Z < edge.Z)
                if (particle.Last_Z < Location.Z || particle.Last_Z > edge.Z)
                    return true;
            return false;
        }

        private void ReflectParticle(IParticle particle, Vector3D collisionP)
        {
            //Vector3D dist_Total = new Vector3D(particle.X - particle.Last_X,
particle.Y - particle.Last_Y, particle.Z - particle.Last_Z);
            var edge = Location + Dimension;

            Vector3D dist_AfterCollision = new Vector3D(particle.X -
collisionP.X, particle.Y - collisionP.Y, particle.Z - collisionP.Z);
            Vector3D reflectionVector = CalcReflectVector(dist_AfterCollision,
collisionP);
            var reflectedPoint = collisionP + reflectionVector;
            particle.Last_X = collisionP.X;
            particle.Last_Y = collisionP.Y;
```

```
            particle.Last_Z = collisionP.Z;
            particle.X = reflectedPoint.X;
            particle.Y = reflectedPoint.Y;
            particle.Z = reflectedPoint.Z;
            // we need to check for the rare cases when reflected particle
hits the boundary again. then it needs to be reflected again
            if (ParticleCrossEdges(particle))
            {
                collisionP = CalcCollisionPoint(collisionP, reflectedPoint);
                ReflectParticle(particle, collisionP);
                return;
            }
        }

        private Vector3D CalcCollisionPoint(Vector3D p1, Vector3D p2)
        {
            ResetStates(); //reset last collision plane
            var edge = Location + Dimension;

            var direction = p2 - p1;
            double[] distances = new double[6];
            Vector3D[] collisionPoints = new Vector3D[6];

            double t = (Location.X - p1.X) / direction.X;
            double x = Location.X;
            double y = p1.Y + t * direction.Y;
            double z = p1.Z + t * direction.Z;
            Vector3D cp = new Vector3D(x, y, z);
            distances[(int)Plane.Left] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Left] = cp;

            t = (edge.X - p1.X) / direction.X;
            x = edge.X;
            y = p1.Y + t * direction.Y;
            z = p1.Z + t * direction.Z;
            cp = new Vector3D(x, y, z);
            distances[(int)Plane.Right] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Right] = cp;

            t = (Location.Y - p1.Y) / direction.Y;
            x = p1.X + t * direction.X;
            y = Location.Y;
            z = p1.Z + t * direction.Z;
            cp = new Vector3D(x, y, z);
            distances[(int)Plane.Buttom] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Buttom] = cp;

            t = (edge.Y - p1.Y) / direction.Y;
            x = p1.X + t * direction.X;
            y = edge.Y;
            z = p1.Z + t * direction.Z;
            cp = new Vector3D(x, y, z);
            distances[(int)Plane.Top] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Top] = cp;

            t = (Location.Z - p1.Z) / direction.Z;
            x = p1.X + t * direction.X;
            y = p1.Y + t * direction.Y;
            z = Location.Z;
            cp = new Vector3D(x, y, z);
            distances[(int)Plane.Back] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Back] = cp;
```

```
            t = (edge.Z - p1.Z) / direction.Z;
            x = p1.X + t * direction.X;
            y = p1.Y + t * direction.Y;
            z = edge.Z;
            cp = new Vector3D(x, y, z);
            cp = new Vector3D(x, y, z);
            distances[(int)Plane.Front] = (cp - p1).LengthSquared;
            collisionPoints[(int)Plane.Front] = cp;

            int minIndex = 0;
            while (!IsBetweenTwoPoints(p1, p2, collisionPoints[minIndex]))
                minIndex++;
            double min = distances[minIndex];
            for (int i = minIndex + 1; i < distances.Length; i++)
            {
                if (distances[i] <= min && IsBetweenTwoPoints(p1, p2,
collisionPoints[i]))
                {
                    minIndex = i;
                    min = distances[i];
                }
            }
            collisionPlane[minIndex] = true;

            return collisionPoints[minIndex];
        }

        private bool IsBetweenTwoPoints(Vector3D p1, Vector3D p2, Vector3D
point)
        {
            if (p1.X < p2.X)
            {
                if (point.X >= p2.X || point.X <= p1.X) //equality is a must
here
                {
                    return false;
                }
            }
            else
                if (point.X >= p1.X || point.X <= p2.X)
                {
                    return false;
                }
            if (p1.Y < p2.Y)
            {
                if (point.Y >= p2.Y || point.Y <= p1.Y)
                {
                    return false;
                }
            }
            else
                if (point.Y >= p1.Y || point.Y <= p2.Y)
                {
                    return false;
                }
            if (p1.Z < p2.Z)
            {
                if (point.Z >= p2.Z || point.Z <= p1.Z)
                {
                    return false;
                }
```

```
            }
            else
                if (point.Z >= p1.Z || point.Z <= p2.Z)
                {
                    return false;
                }
            return true;
        }

        private Vector3D CalcReflectVector(Vector3D direction, Vector3D
collisionPoint)
        {
            Plane collidedPlane = Plane.Top;
            for (int i = 0; i < collisionPlane.Length; i++)
            {
                if (collisionPlane[i])
                {
                    collidedPlane = (Plane)i;
                    break;
                }
            }
            Vector3D n = Vector3D.Zero;
            switch (collidedPlane)
            {
                case Plane.Top:
                    n = Vector3D.Down;
                    break;
                case Plane.Buttom:
                    n = Vector3D.Up;
                    break;
                case Plane.Left:
                    n = Vector3D.Right;
                    break;
                case Plane.Right:
                    n = Vector3D.Left;
                    break;
                case Plane.Back:
                    n = Vector3D.Backward;
                    break;
                case Plane.Front:
                    n = Vector3D.Forward;
                    break;
            }

            Vector3D projection_n = Vector3D.Dot(n, direction) * n;
            Vector3D projection_d = direction - projection_n;
            Vector3D reflectionVector = projection_d - CoefOfRestitution *
projection_n;
            return reflectionVector;
        }

        //-----------------------------------------------------------
        #endregion
    }
```