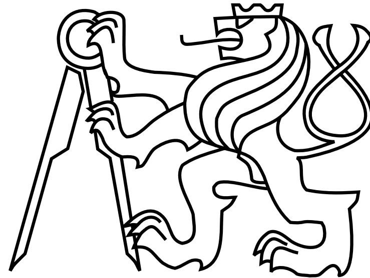
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Project

Sensor network simulator core

David Kubásek

Supervisor: Ing. Jan Kubr

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer networks

January 5, 2015

Aknowledgements

I would like to thank Ing. Jan Kubr, my supervisor, for valuable comments and advice.I would also like to thank to the ones who are close to me, because without their help and support this work would not be created. Last but not least, I would like to thank Mgr. Stepan Sindelar for his persistent struggle for the freedom of people of Tibet as his determination for that matter keeps inspiring me.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on January 5, 2015 ....

Abstract

This work focuses on wireless sensor networks. The nature of sensor networks brings new kinds of issues that have not been encountered in the design of traditional wired and wireless networks. Therefore new algorithms designed for sensor networks keep emerging. This work intends to provide an environment that would allow simulation of these new algorithms. Regarding the phenomena being simulated, this work particularly focuses on simulation of routing algorithms for sensor network and simulation and a technique called distributed phase shift beamforming.

Abstrakt

Tato práce se zabývá bezdrátovými senzorovými sítěmi. Senzorové sítě přinášejí nové druhy problémů, se kterými se při návrhu tradičních drátových i bezdrátových sítí nepočítalo. Proto vznikají nové algoritmy zaměřené na senzorové sítě. Tato práce si klade za cíl poskytnout prostředí, které umožní simulaci takových algoritmů. Co se týče konkrétních simulovaných jevů, tato práce se zaměřuje na směrovací algoritmy pro senzorové sítě a techniku zvanou beamforming.

Contents

1 Introduction **1**
- 1.1 Introduction . . . 1
- 1.2 Contents of this work . . . 2

2 Specification **3**
- 2.1 Aim of this work . . . 3
- 2.2 Requirements . . . 3

3 Analysis **5**
- 3.1 Focus of this work . . . 5
  - 3.1.1 Distributed phase shift beamforming . . . 5
  - 3.1.2 Geographic routing . . . 6
- 3.2 General concerns of sensor network simulation . . . 7
  - 3.2.1 ISO OSI coverage . . . 7
  - 3.2.2 Physical layer concerns . . . 7
  - 3.2.3 Link layer concerns . . . 8
  - 3.2.4 Network layer concerns . . . 8
- 3.3 Known simulators . . . 9
  - 3.3.1 OMNeT++ . . . 9
    - 3.3.1.1 INET framework . . . 9
    - 3.3.1.2 Castalia . . . 9
  - 3.3.2 NS-2 . . . 9
  - 3.3.3 TOSSIM . . . 10
  - 3.3.4 Shawn . . . 10
- 3.4 Result of the analysis . . . 10
- 3.5 Proposed solution . . . 11

4 Implementation **17**
- 4.1 Design . . . 17
  - 4.1.1 Simulator core . . . 17
  - 4.1.2 Node . . . 20
    - 4.1.2.1 Protocol stack . . . 20
    - 4.1.2.2 Physical layer . . . 22
    - 4.1.2.3 Data link layer . . . 23
    - 4.1.2.4 Medium access control sub-layer . . . 24

4.1.2.5 LLC built in sub-layer 25
4.1.2.6 LLC user controlled sub-layer 28
4.1.2.7 Network layer 29
4.1.3 Messages 31
4.2 Implementation 31
4.3 Input 32
4.4 Output 36
4.5 Portability 36

5 Testing **39**
5.1 Unit testing 39
5.2 Integration testing 42

6 Conclusion **45**
6.1 Conclusion 45
6.2 Future work 46

A Installation guide **51**
A.1 Recommended installation 51

List of Figures

3.1 Comparison of radiation patterns of a single transmitter and two transmitters using beamforming technique [6] 6
3.2 ZigBee protocol stack 7

4.1 operation of the simulation scheduler 18
4.2 Simulation node protocol stack 21
4.3 Event flow between the layers 22
4.4 operation of the physical layer 23
4.5 send request handling by the mac sub-layer 24
4.6 receive request handling by the mac sub-layer 25
4.7 State machine describing the operation of the LLC built in sub-layer 26
4.8 Flowchart describing handling of LLC blocking send event 28
4.9 Flowchart describing handling of LLC send event with acknowledgement . . . 29
4.10 Flowchart describing handling of LLC blocking send event with acknowledgement 30
4.11 Flowchart describing handling of LLC recv event 31
4.12 Flowchart describing handling of LLC pass event 32
4.13 Flowchart describing operation of a user routine 35

List of Tables

3.1 Compliance of known solutions with the requirements of this work. Note that "Y" denotes full compliance, "P" denotes partial compliance, and "N" denotes no compliance 10

Chapter 1

Introduction

1.1 Introduction

During the last couple of years sensor networks became a very attractive field of study for many researchers [10].

A sensor network typically consists of a large amount of inexpensive nodes that are usually quite limited in terms of computational capabilities, memory capacity, communication speed, area coverage of antennas, especially compared to traditional wireless networks [10]. Whereas traditional wireless networks require some infrastructure such as base stations (which are then serving as masters in master/slave communication paradigm in the network) to work properly, sensor networks are completely unstructured [20].

The nature of the sensor networks brings new kinds of issues that have not been encountered in the design of traditional wired and wireless networks [2]. Especially the lack of resources on each node implies a need for new algorithmic ideas that combine methods of distributed computing with traditional centralized algorithms [10].

In large sensor networks, traditional stateful routing algorithms become essentially useless due to the inability of the node to keep the required information about the state (i.e. a routing table). Therefore new routing algorithms, including greedy stateless routing algorithms and geographic routing algorithms have been designed.

As a node of a sensor network is typically powered by a battery, it is quite desirable to extend the battery life. A few techniques that indirectly achieve that emerged. For example distributed phase shift beamforming technique, that reduces the total power spent on a transmission by reducing the electromagnetic interference in the network [6].

In order to verify that some ideas, conclusions, or algorithms concerning sensor networks are valid, it is quite desirable to have these tested. Basically, there are two approaches to achieve that, that are fundamentally different. The first is to actually perform the test on a real sensor network. This approach probably delivers very precise results. However, there are some practical difficulties that emerge when it comes to actually using a real sensor network, "It is difficult to operate and debug such systems. This may have contributed to the fact that only very few of these networks have yet been deployed. Real-world systems typically consist of roughly a few dozen sensor nodes, whereas future scenarios anticipate networks of

several thousands to millions of nodes"[10]. The other approach is to design a model of a sensor network and test the algorithms upon that model using a simulation.

This work focuses on simulations of sensor networks. The aim of this work is to find a suitable simulation environment for specific algorithms and techniques that were designed for sensor networks, namely this work focuses on simulation of routing algorithms for sensor networks and on simulation of distributed phase shift beamforming technique. These are further discussed in chapter 3.

1.2 Contents of this work

- **Chapter 2** defines the problem that this work intends to solve and specifies the requirements that the solution is supposed to meet.
- **Chapter 3** analyses known solutions, rates these solutions in terms of the requirements defined in 2 and based on the results if this analysis proposes a solution and discusses it's design.
- **Chapter 4** thoroughly describes the details of the final solution, including the detailed design of the components that the solution consists of and implementation caveats.
- **Chapter 5** describes the methods that were used to assure the quality of the final solution.
- **Chapter 6** summarizes this work, rates the final solution in terms of the requirements defined in 2 and suggests a future work that might be continuation of this work.

Chapter 2

Specification

This chapter describes the expected result of this work and then it defines requirements that the result should meet.

2.1 Aim of this work

The main goal of this work is to provide an appropriate simulation environment for sensor networks. This could either mean finding an existing simulator that meets the requirements defined in 2.2, or finding an appropriate simulation environment and modifying it to meet the requirements (for example via developing a plugin to that environment), or developing a standalone simulator if the other options turn out not to be sufficient in terms of requirements.

2.2 Requirements

Two areas of interest have been identified as the key concerns of this work:

- **Routing algorithms simulation:** The final simulator shall allow the user to simulate the behavior of wide range of routing algorithms in a *simple* and *straightforward* way. The design should respect the nature of sensor networks and therefore incorporate the support of sensor network specific algorithms such as *geographic routing*.

- **Physical layer simulation:** The final simulator shall allow the user to define a model of physical layer that supports advanced media access control techniques, such as distributed phase shift beamforming.

The simulator should also show reasonable runtimes for each simulation task (even for excessive amount of nodes simulated). That means that routing algorithm simulation runtime should not be prolonged by complicated calculations in physical layer and vice versa. In order to provide this functionality the OSI layers of the simulation must be clearly (by well defined interface) separated.

Fundamental requirements that the output of this work is supposed to meet are summarized in the following list:

1. Advanced transmission control that allows simulation of advanced media access control techniques, such as distributed phase shift beamforming
2. Independent model of each layer of the protocol stack that is easily exchangeable for a different one in the future
3. Simple user interface of each layer of the protocol stack that allows the user to implement the protocol logic in a straightforward way
4. Reasonable simulation runtime even for large networks

Chapter 3

Analysis

Section 3.1 describes the most important phenomena to be simulated. Section 3.2 discusses general aspects to keep in mind when simulating a wireless sensor network. Section 3.3 describes known solutions and critically evaluates their advantages and disadvantages with regard to the requirements defined in 2. Section 3.4 concludes the analysis and upon it's result section 3.5 proposes a solution.

3.1 Focus of this work

In order to design a truly useful simulator it is necessary to choose which phenomena require more detailed model and which can be simplified. This approach may be beneficial since the important details are included in the model of the simulated phenomenon while the run time of the simulator does not exceed the reasonable limits.

3.1.1 Distributed phase shift beamforming

The first specific of this work is a focus on distributed phase shift beamforming (DPSBF)[1]. Beamforming technique is used to affect the radiation pattern of the transmitter in order to save energy and/or to lower the amount of electromagnetic interference in the sensor network. These effects are achieved by using more (than one) antennas to transmit the same signal simultaneously (of course this requires a proper synchronization of the antennas). The resulting radiation pattern is affected by the addition of multiple signals. There are directions where the signal is amplified and directions where the signal is attenuated. Figure 3.1 depicts this situation. The two nodes u and v that are denoted by circles around them represent the transmitters in this situation. The node w denoted by a square around it represents a receiver. The continuous line shows the range of the transmission when both transmitters are transmitting together using DPSBF, whereas the dashed lines represent the range each of nodes u and v would cover if it was transmitting alone using a standard transmission. The figure clearly shows that area affected by the transmission using DPSBF is smaller than the area (and therefore the amount of nodes) affected by a standard transmission [6].

[1] Beamforming or spatial filtering is a signal processing technique used in sensor arrays for directional signal transmission or reception.[22]

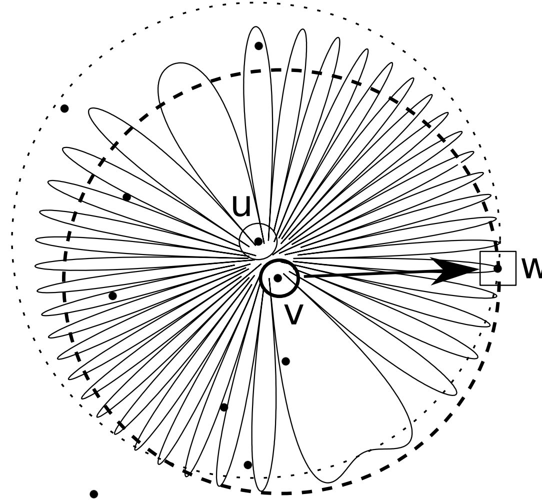


Figure 3.1: Comparison of radiation patterns of a single transmitter and two transmitters using beamforming technique [6]

The name *distributed* comes from the fact that there is no antenna array and therefore no central transmitter, the adjacent nodes of the network are used to shape the radiation pattern instead.[6]. Description of how such nodes are synchronized is out of the scope of this work. Any simulator that is able to simulate DPSBF must incorporate a detailed (and customizable) model of the lowest layers of the protocol stack.

3.1.2 Geographic routing

The second specific issue this work focuses on are the routing algorithms used in wireless and ad hoc networks. Geographic routing algorithms use the information about the geographical position of the network nodes in order to make packet forwarding decisions. These are attractive for sensor networks because they do not need to exchange routing information and therefore work near stateless and require almost no memory for routing (which is desirable since the resources are usually scarce in wireless sensors)[14]. Of course not only geographic routing algorithms are used in sensor networks, but these require the most specific attributes of the simulation environment. Mainly because geographic routing algorithms require the sensors to know their position and the position of the destination. In order to design models of such algorithms the simulation environment must provide the information about the geographical position of the network nodes in the user space.

3.2 General concerns of sensor network simulation

3.2.1 ISO OSI coverage

When simulating a sensor network, it is not absolutely necessary to cover all the layers of the `ISO OSI` model[2]. The reason for this assumption is that sensor networks differ from standard wired networks mainly in low level layers of the model. As an example that supports this assumption, consider *ZigBee*. ZigBee is a specification for a suite of high level communication protocols used to create personal area networks built from small, low-power digital radios. ZigBee is based on `IEEE 802.15.4`[36]. Figure 3.2 displays ZigBee protocol stack. As can be seen, the higher level OSI layers are merged into one application layer. So instead of implementing a traditional stack with seven layers, ZigBee implements only four layers. Note that these four layers differ from the first four layers of the TCP/IP stack (for example the network layer (3) of the ZigBee stack actually implements logical link control [8] that is in traditional networks covered by data link layer (2) [31]). In principle, this work shall follow the same attitude. Only OSI layers 1-3 will be covered.

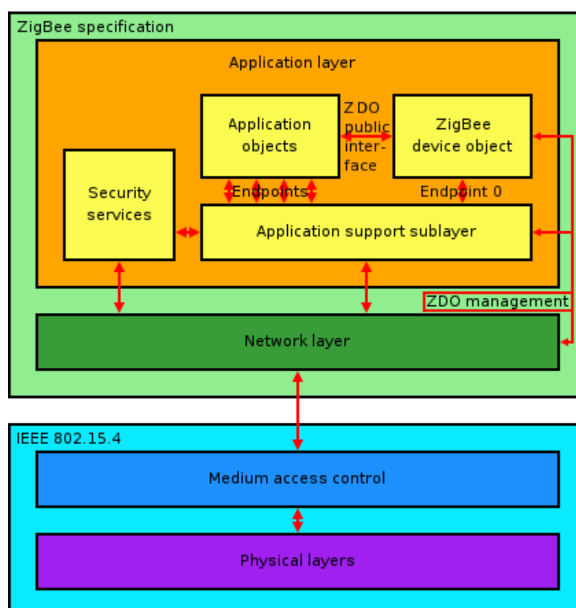


Figure 3.2: ZigBee protocol stack
[36]

3.2.2 Physical layer concerns

Before evaluating constraints of physical layer, essential parameters of radio transmission must be defined. Physical layer constraints should be expressed with regard to these parameters. Basically, the sender node transmits data through an antenna with certain gain[3] using

[2] OSI = Open system interconnections model. It specifies abstract layers and each of this layers has defined functions. It is a standard for systems interconnections [31]

[3] Power gain (or simply gain) is a unitless measure that combines an antenna's efficiency and directivity[21]

certain power into an open space. Some of the power is lost on the way to the receiver[4] and on the receiving side of the transmission there is another antenna with (in general) different gain. If the power level of the signal on the receiving side is sufficient, receiver considers it as data, otherwise the signal is considered to be noise.

As mentioned earlier, it is quite necessary for the simulation environment to support simulation of nodes having antennas with various gains transmitting in various directions. The radiation pattern of an antenna is a plot of the relative field strength of the radio waves emitted by the antenna at different angles. It is typically represented by a three dimensional graph, or polar plots of the horizontal and vertical cross sections. The pattern of an ideal isotropic antenna, which radiates equally in all directions, would look like a sphere. Many nondirectional antennas, such as monopoles and dipoles, emit equal power in all horizontal directions, with the power dropping off at higher and lower angles; this is called an omnidirectional pattern and when plotted looks like a torus or donut.[21]. Some representation of the radiation pattern should be implemented by the simulation environment in order to support various types of radiation pattern. On the other hand the simulator shall provide different types of free space path loss calculation. Of course, the environment should recognize interference on the physical layer. Advanced techniques using constructive interference, such as beamforming shall be simulable in the environment.

3.2.3 Link layer concerns

As link layer simulation is not the primary goal of this work, a simple model of this layer, e.g. `Ideal planning` is sufficient. In this case, ideal planning means planning of transmission in a manner where there are no collisions possible. If conditions preventing a node from transmission, such as network congestion occur, the data is being dropped until this condition vanishes. Though the simulator should provide independent model of the link layer that could be easily exchanged for a different model (for example implementing some real protocol) in case ideal planing turns out to be insufficient in the future.

3.2.4 Network layer concerns

One of the main goals of this work is to provide viable environment for simulating algorithms designed especially for sensor networks (for example geographical routing algorithms, for more information refer to [20] or [2]). This concerns both existing algorithms and their possible modifications and on the other hand brand new algorithms. This implies that network layer model must be easily exchangeable with a completely different model. Due to the complexity and nature of geographical routing algorithms it is quite appropriate that the network layer model of each node has the ability to access the global information about the network model, such as network topology etc. This requirement comes from the fact that the global knowledge of the network in each node could significantly simplify the implementation of such algorithms within the simulation environment.

[4] In telecommunication, free-space path loss (FSPL) is the loss in signal strength of an electromagnetic wave that would result from a line-of-sight path through free space (usually air), with no obstacles nearby to cause reflection or diffraction [27].

3.3 Known simulators

This section describes known simulators that may be suitable for sensor network simulation, and critically evaluates their features and properties with respect to requirements defined in 2. Table 3.1 summarizes the compliance of the known solutions with the requirements defined in chapter 2.

3.3.1 OMNeT++

The "Objective Modular Network Testbed in C++" is an objective-oriented modular discrete event simulator[10]. OMNeT++ itself is just a simulation core and the domain-specific functionality is provided by model frameworks that are developed more or less independently on OMNeT++.

3.3.1.1 INET framework

The INET Framework is an open-source communication networks simulation package for the OMNeT++ simulation environment. even though it provides out of the box support for a lot of network protocols, it lacks support of sensor network specific protocols. On the other hand there are extensions of INET framework such as **MiXiM**, that provide sensor network specific features.MiXiM shows supreme support of physical layer of sensor networks. It provides various out of the box signal propagation models (called `AnalogueModel`), various media access control techniques (called `MacLayer`) and user-defined model of `Decider` which determines whether received signal is data or noise. The biggest disadvantage of using INET framework is its complexity that disallows the user to effectively simulate large networks. Another consequence of the frameworks complexity is complicated definition of the models.

3.3.1.2 Castalia

Castalia is another framework based on OMNeT++ supporting wireless sensor networks. It can be used by researchers and developers who want to test their distributed algorithms and/or protocols in realistic wireless channel and radio models, with a realistic node behavior especially relating to access of the radio[5]. Castalia provides advanced channel model based on empirically measured data that allows the path loss to change over time and nodes to move. It also provides a couple of out of the box MAC and routing algorithms. The disadvantages are basically the same as the ones mentioned in 3.3.1.1 and since these disadvantages are not a specific disadvantages of a single framework, it is reasonable to assume that any framework built upon OMNeT++ simulation library might show such disadvantages.

3.3.2 NS-2

NS-2 (Network simulator 2) is a general simulator originally designed for ip networks [37]. On one hand, NS-2 provides a considerable range of protocols that are specific for wireless sensor network, including sensing channels, sensor models, battery models, and lightweight protocol stacks for wireless micro sensors. On the other hand "the highly detailed packet level simulations lead to a runtime behavior closely coupled with the number of packets that

Solution	Transmission Control	Exchangeable Model	User Interface	Short Runtime
MIXIM	Y	Y	N	N
Castalia	Y	P	N	N
Shawn	N	N	Y	Y
NS-2	Y	N	N	N
TOSSIM	N	N	N	Y

Table 3.1: Compliance of known solutions with the requirements of this work. Note that "Y" denotes full compliance, "P" denotes partial compliance, and "N" denotes no compliance

are exchanged, making it virtually impossible to simulate really large networks. In principle, Ns-2 is capable of handling up to 16,000 nodes, but the level of detail of its simulations leads to a runtime that makes it hopeless to deal with more than 1,000 nodes [10]. Another considerable drawback of NS-2 is that it uses TCL[5] for model definitions, which may lead to difficulties while using NS-2 [37].

3.3.3 TOSSIM

TOSSIM is a platform specific simulator that simulates TinyOS[6] motes (sensors) at the bit level. It actually compiles the code written for TinyOS into an executable that can be run on a standard PC equipment [10]. While TOSSIM might be a powerful tool for testing code designated for the TinyOS motes, it is virtually useless for simulating generic sensor networks.

3.3.4 Shawn

Shawn is a sensor network simulator developed because the runtimes of other network simulators became unbearable when the amount of nodes reached a certain limit (hundreds of thousands). Compared to other network simulators Shawn runs very quickly due to the fact that it is not actually simulating the network stack. Instead of simulating some phenomenon, it simulates the effects caused by the phenomenon (e.g. instead of simulating MAC layer, it simulates packet loss and data corruption).[10] This basically implies that it lacks support for any low-level specific protocols. On the other hand it supports very large networks.

3.4 Result of the analysis

Network simulators can be divided into two categories:

[5] TCL is a scripting language created by John Ousterhout. Originally "born out of frustration", according to the author, with programmers devising their own languages intended to be embedded into applications, Tcl gained acceptance on its own. It is commonly used for rapid prototyping, scripted applications, GUIs and testing. Tcl is used on embedded systems platforms, both in its full form and in several other small-footprint versions [32].

[6] "TinyOS is a free and open source software component-based operating system and platform targeting wireless sensor networks (WSNs)" [34].

- **Heavy weight simulators** like OMNeT++ or NS-2 tend to implement every protocol possible on each OSI layer. This results in a very complex simulation environment. The main advantage of such approach (considering this work's focus) is the ability to simulate advanced low-level techniques such as beamforming. On the other hand it comes with the price. Heavy weight simulators are not able to simulate networks consisting of large amount of nodes in a finite time.

- **Light weight simulators** like Shawn tend to simplify the physical aspects of the network into statistical models. This effectively makes it impossible to simulate advanced low-level techniques but on the other had the simulation times are better compared to heavy weight simulators. Also the definition of the model to be simulated is significantly simpler for the user.

The solution provided by this work shall combine the advantages of both approaches. It shall simplify phenomena that are out of focus of this work using the statistical approach in order to lower the complexity and accelerate the simulation. At the same time it shall support the advanced features of physical layer. It will make the implementation of routing algorithms easier by providing the global information about the network to each layer.

3.5 Proposed solution

As concluded in 3.4, the requirements defined in 2 are not met by known simulation environments (there is no known solution that meets all the requirements). The problem of the lightweight environments is the lack of support of low level protocols, on the other hand the problem of the heavyweight environments is the complexity of usage and long run times.

The *lightness* can be achieved by designing a simulation environment just for the specific area of sensor networks without the ambition to simulate different phenomena in the future. This approach leads to the lightest possible solution because there is no need for generalization. The simplicity of use could be achieved by moving the control flow[7] into the user space.

Note that even though this has not been mentioned before, all the solutions described in 3.3 (except for TOSSIM) were in fact discrete event simulators. A discrete event simulator is a simulator that considers time a discrete (integer) variable and processes one event at each *moment* of time. User-defined actions are then invoked when specific events (hooked to those actions) occur[12]. The fact that user-defined actions are triggered by events means that the user-defined action only gets the program control flow for the time that is necessary in order to handle the event. This behavior is highly undesirable when designing the logic of the sensor node (e.g. the routing algorithm), since it forces the user to organize the code in an unnatural way. Instead of dividing the code into units that cover the same logic, the code ends up divided into units handling each event. As a consequence the code of one algorithm in *unintentionally* divided into those units, which can cause trouble writing, reading and maintaining the code.

[7]In computer science, control flow (or alternatively, flow of control) refers to the order in which the individual statements, instructions or function calls of an imperative or a declarative program are executed or evaluated.[25]

Since one of the most important requirements that this work is supposed to meet is to allow a simple and straightforward way of implementation of routing algorithms (see 2), the aforementioned problems with the event-based approach should be resolved. [12] defines three different paradigms of discrete simulations[8]:

1. **Activity oriented paradigm:** Under activity-oriented paradigm, the time is broken into tiny slices. As in any other discrete simulation, time is represented as a discrete variable. In activity oriented paradigm, each time this variable is incremented, the state of the system is checked and if some new event occurs, it is properly handled and time is increased (by the same tiny slice). The biggest drawback of this paradigm is that there is a lot of time *moments* when nothing happens, which may cause the simulation to run significantly slower than necessary. The biggest advantage is the simplicity of design.

2. **Event oriented paradigm:** Event oriented paradigm speeds up the execution by "skipping" the moments that generate no event and are therefore not interesting for the simulation. This is achieved by implementation of *event set*, the set of all the pending events with time stamps indicating when each event is due. This approach completely eliminates the busy waiting for the next event and therefore brings a major performance improvement in comparison to the activity oriented paradigm. However, under event oriented paradigm, the simulation spends a major portion of time finding the event with the lowest time stamp (i.e. the event that should be invoked next). [12] suggest that a reasonable implementation of the event set is a heap-based priority queue, that minimizes the finding time of the next event significantly. Note that it is virtually impossible to have two different events happen simultaneously under the event oriented paradigm. Even if two different events were assigned the same time stamp, they would be invoked sequentially.

3. **Process oriented paradigm:** The name of the process oriented paradigm comes from the resemblance to the Unix processes - under the process oriented paradigm each simulation activity is modeled by a process. Of course the *event set* is also implemented under process oriented paradigm and is basically the same as the one employed under event based paradigm. The chief virtue of the process oriented paradigm is the implied modularity of the code that the event based approach lacks. "The process-oriented paradigm produces more modular code. This is probably easier to write and easier for others to read. It is considered more elegant, and is the more popular of the two main world views[9] today."[12]. The usage of the actual processes is considered an outdated approach. [12] suggests that usage of threads to implement process oriented paradigm is more suitable than the actual processes. "Indeed one could write process-oriented simulations using Unix processes. However, these would be inconvenient to write, difficult to debug, and above all they would be slow."[12]. For more information about threads and processes, refer to any good book focusing on operating systems.

[8]Note that there exist other types of simulations than discrete ones (e.g. continuous simulation or process simulation), but these are out of the focus of this work, because they are suitable for simulating different phenomena.[24]

[9]Note: by the two main world views the author means the event oriented paradigm and the process oriented paradigm

Considering the paradigms used in discrete simulations that are defined above, the process oriented paradigm seems to be the most suitable for the needs of this work. However, designing a process driven discrete event simulator requires a deeper thought. For example, the most straightforward idea would be to implement each sensor network node as a thread. Consider simple following example that is using pthreads[10]. The only purpose of this example is to find out how many pthreads can be spawned inside one process.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4  #include <errno.h>
5  #include <stdlib.h>
6  #include <string.h>
7
8  #define THREAD_COUNT 1024
9
10 void *
11 routine (void *arg)
12 {
13     sleep(10);
14     pthread_exit(NULL);
15 }
16
17 int
18 main (void)
19 {
20     pthread_t   th[THREAD_COUNT];
21     int         ret;
22     int         i;
23
24
25     for (i = 0; i < THREAD_COUNT; ++i) {
26         ret = pthread_create(&th[i], PTHREAD_CREATE_JOINABLE, routine, NULL);
27         if (ret != 0) {
28             fprintf(stderr, "pthread_create(%d): %s\n", i, strerror(errno));
29              exit(EXIT_FAILURE);
30         }
31     }
32
33     for (i = 0; i < THREAD_COUNT; ++i) {
34         ret = pthread_join(th[i], (void **)NULL);
35         if (ret != 0) {
36             fprintf(stderr, "pthread_join(%d): %s\n", i, strerror(errno));
37              exit(EXIT_FAILURE);
38         }
39     }
40
41     exit(EXIT_SUCCESS);
42 }
```

Listing 3.1: pthread test

The code was compiled on a 64 bit CentOS linux machine, compiled using gnu compiler collection, with standard c flags. The result of running the compiled code on the same

[10] Pthreads are POSIX implementation of threads. for more information refer to [13]

machine follows:

```
[user@hostname tmp]$ gcc -o thread -lpthread thread.c -Wall -Werror -ansi
[user@hostname tmp]$ ./thread
pthread_create(384): Resource temporarily unavailable
[user@hostname tmp]$
```

The output of the binary clearly states that it was able to successfully spawn 383 threads inside one process, but it was not able to spawn any more threads. The pthread manual states that this error is returned if: "Insufficient resources to create another thread, or a system-imposed limit on the number of threads was encountered. The latter case may occur in two ways: the RLIMIT_NPROC soft resource limit (set via setrlimit(2)), which limits the number of process for a real user ID, was reached; or the kernel's system-wide limit on the number of threads, /proc/sys/kernel/threads-max, was reached."[13]. Either way the error was caused by the system inability to create another thread. Please note that the amount of the threads that can be created depends on the amount of system resources available and can be tuned. Firstly, the resources can be expanded (e.g. by running the code on a machine with more resources available), secondly, the pthreads themselves can be tuned using `pthread_attr` structure. User can for example tune the stack size of each thread. In this case, the most likely reason preventing new threads from spawning is the lack of memory on the machine. It is reasonable to assume, that by reducing the stack size of each thread to a half of it's default value, ability to spawn roughly twice as many threads could be achieved. However, such tuning comes with the price. Reducing the stack size of a thread is a risky move because it increases the likelihood of the stack overflow error. Considering the nature of the sensor networks (number of nodes is usually large, it can exceed 10000 [10]), this lack of threads is fairly limiting.

There are other ways to deal with the lack of physical threads. One of them is thread pooling. Thread pool pattern works in a way that threads that are available serve the tasks that are waiting to be served in queues[33]. As far as this work is concerned, the nodes of the sensor network could be scheduled as tasks and could be served by a finite amount of threads. On the other hand, the problem with the thread pool is that it does not ensure full concurrency (any amount of nodes can perform actions simultaneously) in the simulated network. Last but not least this approach would bring new issues with synchronization of the threads in the pool.

Even though the process oriented paradigm seem to be the most suitable tool in regards to this work's needs, the use of physical threads on the other hand would not be the optimal way to implement the paradigm. In fact, to ensure concurrency in the simulated network, there is no need for *real* concurrency on the operating system level. It would be just enough to be able to swap context between the execution units of the nodes. Fortunately, there are tools that provide such a feature. The concept of coroutines[11] provides such behavior. It uses non-local jumps in order to alternate between two or more routine (called coroutines).

[11] "Coroutines are computer program components that generalize subroutines for nonpreemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations. Coroutines are well-suited for implementing more familiar program components such as cooperative tasks, exceptions, event loop, iterators, infinite lists and pipes."[26]

Coroutines basically provide multitasking features in the operating system user space[18]. There is a couple of tools that implement coroutines:

- `setjmp.h`: The system header `setjmp.h` provides two functions: `setjmp()` and `longjmp()`. `Setjmp()` stores the current calling stack and `longjmp()` returns to the state stored by `setjmp()`. It just an implementation of the non-local jump.[16]
- `ucontext.h`: The `ucontext.h` header provides more advanced interface for creating coroutines, but the principle is the same as in `setjmp.h`. It allows the user to store and swap execution between different contexts.[15]
- `GNU pth`: "Pth is a very portable POSIX/ANSI-C based library for Unix platforms which provides non-preemptive[12] priority-based scheduling for multiple threads of execution (aka "multithreading") inside event-driven applications. All threads run in the same address space of the server application, but each thread has it's own individual program-counter, run-time stack, signal mask and errno variable"[9]. As the author says, pth library provides thread-like feeling in a user space. Internally, pth is implemented using `ucontext.h`. Basically, pth provides the concept of coroutines wrapped in a thread-like interface.

Because of the reasons mentioned in this section, this work implements a sensor network simulator as an process oriented discrete event simulator using the GNU pth library to model the processes.

[12] The mentioned non-preemptive (or cooperative) scheduling of the threads simply means that the thread **has to** yield the control to the other threads voluntarily.

Chapter 4

Implementation

Section 4.1 describes the design of the solution in detail. Section 4.2 explains the technical details of the implementation, such as implementation language choice, used libraries and tools. Then it discusses the complicated parts and caveats of the implementation as well as the inputs and outputs of the resulting software product. At the end it defines requirements for the target system of the simulation application that need to be met in order for the software product to compile and execute correctly.

4.1 Design

This section describes the design of the proposed solution in detail. First it describes the simulation core which implements the process-driven paradigm simulator proposed in section 3.4 and highlights the parts where the design of the simulation core diverts from the proposed paradigm.

Then it describes an entity representing a single sensor called *node* (note that node is an entity producing the simulation events that enter the simulation core and receiving the events that leave the simulation core), it's properties and behavior. This description includes information about the protocol stack that the node implements and the layers of the protocol stack. Behavior and user accessibility of each layer of the protocol stack is discussed.

At it's end, this section introduces *messages*, a tool designated for delivery of the user data directly between the user accessible sections without affecting the simulation.

4.1.1 Simulator core

First of all, it is necessary to define what an *event* means in context of this work. Let us consider a typical simulation task: a line of customers waiting for an atm. In that context, the possible events are:

- a customer coming to the end of the line
- a customer at the front of the line starts drawing money
- a customer being served finished transaction and leaves

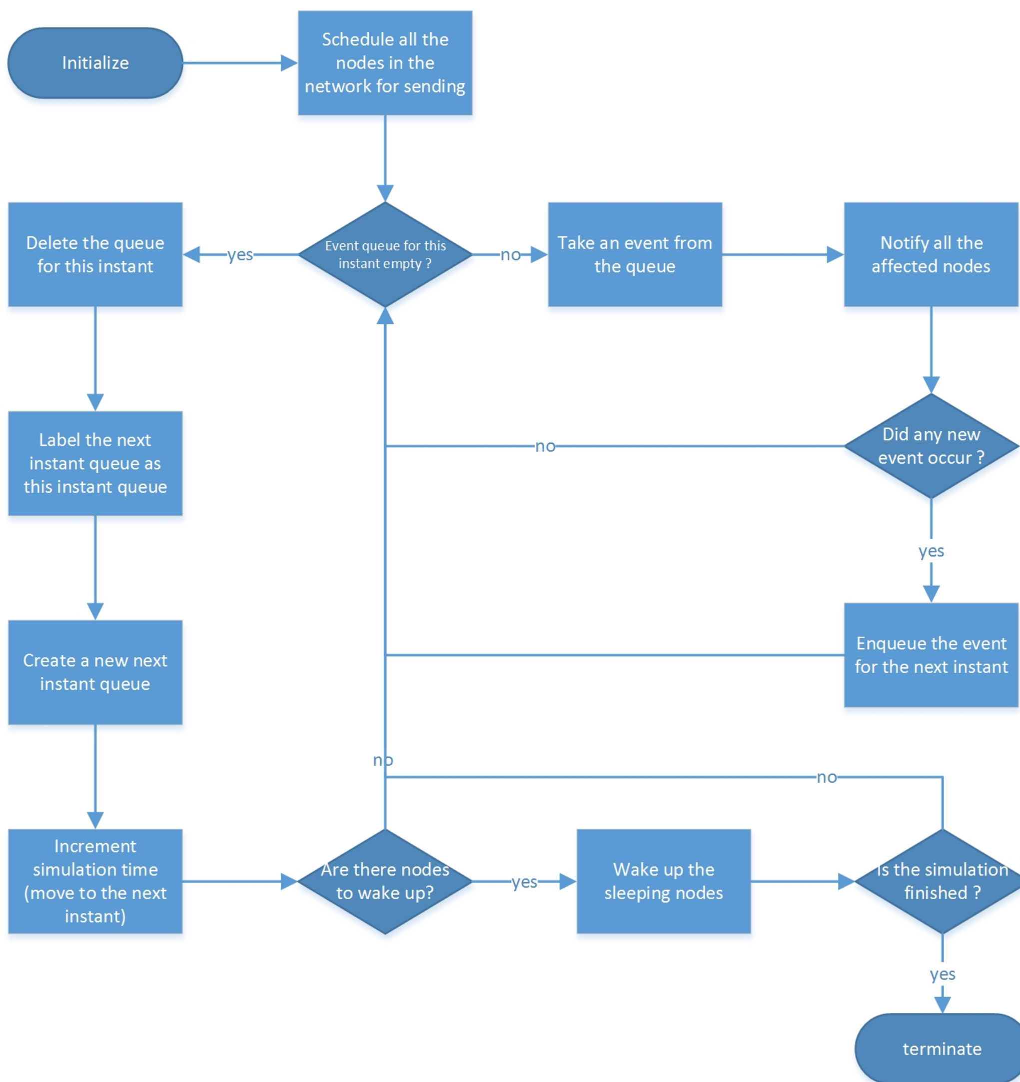


Figure 4.1: operation of the simulation scheduler

Let us define the possible events of the simulator that is the subject of this work in the same manner.

- a node starts transmitting data
- a node stops transmitting data
- a node starts receiving data
- a node stops receiving data

Of course this is just a rough draft of a definition, however it is clear already, that these events overlap in terms of meaning. When some node(s) start transmitting, some

other node(s) start receiving. Therefore as far as the simulation core is concerned, there is only one kind of event defined, an event that represents a data transmission, that contains information about the sources, destinations and duration of the transmission. This definition connects the sources and destinations of the transmission, together with the duration, which makes the implementation simpler. Please note that even though the event definition is connected to the transmission itself, the nodes are still represented as threads (coroutines).

Having the simulation event well defined, it is possible to define the event set mentioned in 3.5. Since the simulation core needs to ensure the concurrency of the nodes (more simultaneous transmissions) in the network, the simulation time must be properly synchronized. The synchronization technique used in this work is based on [3]. Fair threads[3] describes a library that combines usage of the native threads (pthreads) and user space threads (coroutines). The paper defines the term *instant* as the period of time between each synchronization of the cooperative threads. During each instant, the scheduler "fairly" gives each thread the opportunity to execute. When all threads linked to the scheduler performed what they needed during the instant, the scheduler moves to the next instant[3]. This work uses a modified version of this synchronization technique. Of course, this kind of synchronization requires two event sets instead of one. The first contains events that are due during this instant, the second contains events that will be due during the next instant. Note that the events that are present at the second event set, occurred during the current instant. The described synchronization technique gives the nodes just a "feeling" of concurrency, the processing of the "concurrent" events is still sequential. Using two event sets and using *instants* for synchronization are the two most significant deviations from the process oriented simulation paradigm. The previous paragraph defined *duration* as a parameter of the event. It may now become clearer why. The duration of the event specifies the number of instants the transmission (represented by the event) takes to finish. And for this amount of instants the event is present in the core. Thus the contents of the event set that is currently being processed actually contains all the transmission that are currently taking place in the network.

Another feature that the core of the simulator provides is the support of timed waiting[1]. It is clear that implementation of timed waiting in the simulation environment as proposed here requires some context swapping. It may seem that this behavior could be modeled as an event. However this work uses events strictly for modeling transmissions. Therefore timed waiting is handled in a little bit different way than transmissions. For this purpose this work defines another data structure as the part of the simulation core. This structure is just an associative array to which the key is the instant (when the timed waiting ends) and the value is the list of nodes (to be woken up at this instant). The scheduler just checks if there are nodes to be woken up at the end of each instant and then proceeds as defined in the previous paragraph.

The last things that need to be defined are the beginning and terminating conditions. The beginning of the simulation is the invocation of the simulation scheduler. The termination condition is defined as a certain number of instants that the simulation scheduler is supposed to perform.

The flowchart in Figure 4.1 depicts the operation of the simulation scheduler. First the simulation scheduler schedules all the nodes for sending. Note that it is up to the user

[1] An example of timed waiting is standard `sleep(int seconds)` function

to decide what nodes will actually perform the transmission and what nodes will wait for reception or perform timed waiting. Then the scheduler processes all the events that are due during the current instant. Any new event that might occur during this process is scheduled for the next instant. When all the events that are due during the current instant are properly handled, the event set assigned to this instant is just dropped (note that the set is empty) and replaced by the event set assigned to the next instant (which is replace by an empty event set). Before moving to the next instant, the simulation scheduler wakes up all the nodes that are supposed to be woken up during the current instant.

The core of the simulator is naturally not accessible by the user. The user defines actions for each node instead. As the main focus of this work are advanced media access techniques (i.e. distributed phase shift beamforming) and routing algorithms, OSI layers 2 and 3 are fundamental regarding the user interface. The parameters of the user interface on these layers are described in detail in 4.1.2.6 and 4.1.2.7 for the layer 2 and layer 3 respectively.

4.1.2 Node

Since a network node is a key building element of the network to be simulated, this chapter describes the attributes of the nodes thoroughly. A network node is the element of the simulation that directly communicates with the simulation core. It is the element the events are invoked upon (more precisely, it is the physical layer of a node). Each node has a unique location in space, which in this work is considered a flat plane (in order to keep the simulation simple). Therefore the position of a node in space is defined by two spatial parameters (x and y). These parameters are defined by the user during the simulation initialization. Another parameter of a node is a table of neighbors (item in a table is an identification of another node with associated transmitting power needed to reach the node). Note that this parameter in no way simulates the real environment, it is just a convenience parameter that allows reducing the amount of computations during the simulation, since it is computed during the initialization. Chapters 2 and 3 specify that the user needs to access the global information of the simulation, therefore another convenience parameter is the reference to the global data.

Since there are expected user defined protocols on layers 2 and 3, it is highly desirable to assign a coroutine to each OSI layer of each network node[2]. Having a separate coroutine for each OSI layer of each node gives the user a "feeling" of continuous execution on each of these layers. Given that the user does not know about the context switching, developing of user defined protocols is simplified significantly.

4.1.2.1 Protocol stack

The node is designed to implement a three layer protocol stack. The protocol stack of the simulation node is depicted in figure 4.2. Note that the data link layer needs to be divided into sub-layers. This is explained in 4.1.2.3. All the layers have some common attributes:

- Each layer remembers the reference to the parent node, upper and lower layers.

[2] In order to keep the description clear, all the previous mentions of network nodes in this work did not take this fact into consideration.

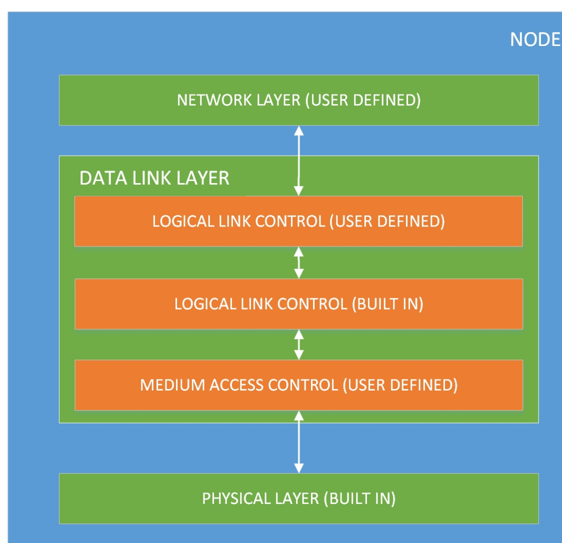


Figure 4.2: Simulation node protocol stack

- Each layer has communication ports to communicate with the rest of the simulation. First communication port is called event port and is designated for communication with the upper and lower layer and is only accessible from these layers. The second communication port is called message port and is used for direct communication between any two layers of any two nodes inside the simulation. The communication using messages is **not** the part of the simulation and is designated for transferring the user data between nodes without affecting the simulation. For more information about messages refer to 4.1.3.
- As mentioned before, each layer has it's own execution unit (coroutine).

Figure 4.2 shows that not all the layers of the protocol stack are user defined. At the first sight, it may seem that having some layers built in could make the simulator less universal. Actually, it is the other way around. The physical layer as the bottom of the protocol stack has to be more or less built in, because it directly communicates with the simulation core. Plus, all the "physical" parameters are available at the link layer as well. The other built in layer (LLC) is a part of the model just for the user convenience. This layer actually does not exist in the real sensor networks (as it duplicates the LLC sub-layer), but here it provides some extra features that may or may not be useful for the user. The important fact is, that if the extra features are not used, the built in LLC layer acts as if not present at the protocol stack (just passes data between the neighboring layers).

Design of the built in layers differs fundamentally from the design of the user defined layers. The control flow in the user defined layers is a responsibility of the user, whereas in the built in layers it is a responsibility of the simulator itself. Therefore the built in layers can be designed and implemented as state machines. On the other hand, on the user defined layers, the simulator only implements encapsulation of data passing and context swapping between the layers.

Data flow between the layers is implemented as *events*. It is important to realize that in context of a node and it's internals, the term *event* (a node event) has a fundamentally different meaning than the event in context of the simulation core (a simulation event), which is defined in section 4.1.1. Simulation event is a term used by [12] and is directly connected to the simulation itself. It is the event that is the member of the event set of the event simulator. On the other hand, a node event is a method to pass information between the internal parts of a node, which in context of a process oriented event simulator is a *process*. As a *process* (a node) is basically opaque to the simulator core in terms of what the process is actually doing and how, *node events* are not known by the simulation core. Thus a *node event* is in no way a event in terms of the event simulation.

To avoid ambiguity of the terms, on the user defined layers this work strictly uses the terminology of node events as follows:

- `SEND` send event represents a request from the upper layer to the lower layer to send data
- `ACCEPT` accept event represents a request from the lower layer to the upper layer to pass data to send (if any)
- `RECV` recv event represents a request from the upper layer to the lower layer to receive data
- `PASS` pass event represents a request from the lower layer to the upper layer to accept the received data

These events are depicted in figure 4.3.

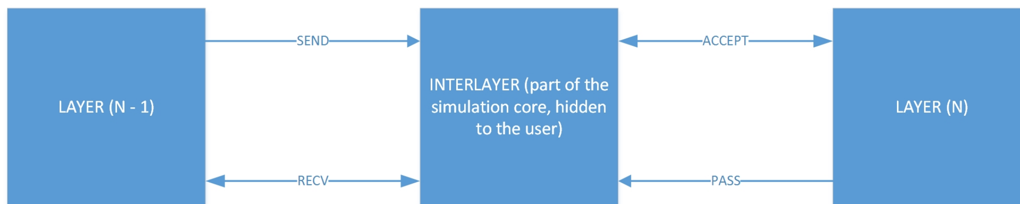


Figure 4.3: Event flow between the layers

4.1.2.2 Physical layer

Since the physical layer in real sensor networks is usually a simple radio transmitter/receiver with just one antenna, the model of the physical layer is a simple radio emulator. Even though the radio emulation is not accessible by the user as it is internal part of the simulation environment, it can be easily replaced by a more complicated model in the future. The physical layer model has two notable attributes. First, the radio is not able to receive any signal while transmitting and vice versa. Second, the radio is able to recognize that collisions of the received signal.

The state machine depicted in figure 4.4 describes the operation of the physical layer. In the *idle* state the radio waits for events. When asked to transmit, it moves to the state

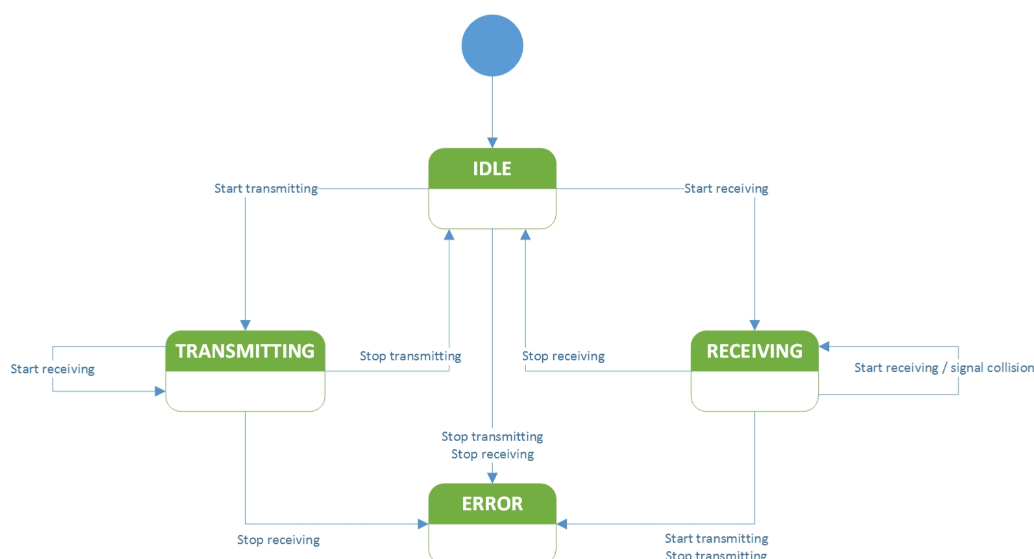


Figure 4.4: operation of the physical layer

transmitting. When asked to receive data it moves to the state *receiving*. When asked to receive data in the state *transmitting*, it just ignores the request (this implements the inability to receive anything when the radio is transmitting), when asked to stop the transmission, it moves to the state *idle*. While in state *receiving*, when asked to receive data, it signals a collision and stays in the same state. When asked to transmit data, it signals an error. Any other requests that are not mentioned above cause an error.

4.1.2.3 Data link layer

The data link layer model of the simulation environment proposed by this work is quite elaborate. The reason is simple, the data link layer of the sensor networks is quite different, and in a way, more complicated than the one known from TCP/IP stack. In TCP/IP, flow control and error management is taken care of by the transport layer (TCP). This implies that need for flow control of the data link layer in TCP/IP is reduced[30]. On the other hand, in the sensor networks, flow control and error management is still managed by the data link layer[7]. This implies that the data link layer in sensor network is more robust than the one used in wired networks and that LLC[3] actually performs more operations than just multiplexing the network layer protocols. Therefore it is quite desirable to actually separate the model of MAC[4] and LLC.

On one hand, it is convenient to allow the user to operate on LLC sub-layer in order to ensure modularity reusablity and universality, on the other hand implementing data acknowledgement protocols might be bothersome for the potential users. This work proposes a compromise between user control and user convenience on LLC sub-layer. The compromise is achieved by splitting the LLC sub-layer into two parts. The bottom part is built in in

[3]logical link control sub-layer of the data link layer
[4]media access control sub-layer of the data link layer

the simulation core and is opaque user, whereas the uppermost part is fully controlled by the user. The built in part implements a simple acknowledgement protocol, which the user part is free to use or not. If the user decides not to use the built in features, the built in part actually behaves as if not present. The following sections describe the design of the sub-layers defined in this paragraph.

4.1.2.4 Medium access control sub-layer

The model of MAC sub-layer is completely controlled by the user including the control flow. There is only an interface built in in the simulation core that provides connection to the neighboring layers (especially context switching that is unnoticed on the user side) and global information about the simulation. The actions that can be performed from the user space are following:

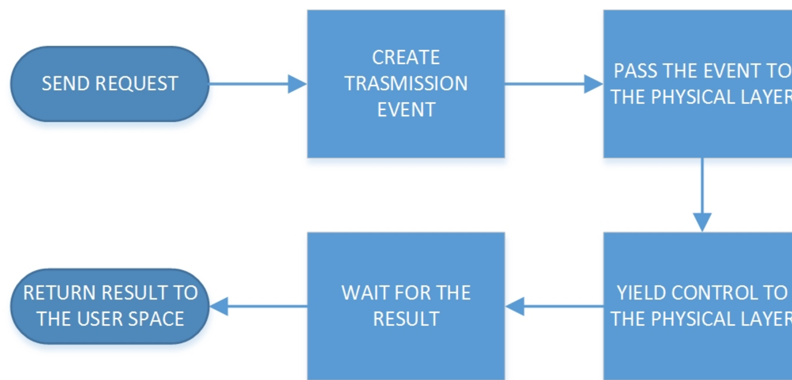


Figure 4.5: send request handling by the mac sub-layer

1. `SEND` When a MAC send event is invoked, MAC just converts the event into an event acceptable by the physical layer (in this case asks for a transmission start), passes the event to the physical layer and yields control (swaps context) to it. The process is depicted in figure 4.5.

2. `RECV` Handling of the receive request is a bit more complicated. First MAC registers a timeout event (see 4.1.1) in the simulation core and then yields the control to the physical layer. When it regains control, it checks whether the timeout expired. If not, the whole process is repeated until an event is received or the timeout expires. If there is an event of an appropriate type received, the data is passed to the user space. Eventually the context is swapped to the user space. Flowchart in figure 4.6 depicts this process.

3. `ACCEPT` Accept request means that the user wants to acquire data from the upper layer. If there were any data passed to the event port in the past by the upper layer, the data are converted and passed into the user space. Otherwise an error is returned.

4. `PASS` Pass request means that the user wants to pass the received data to the upper layer. If such request is claimed, the data together with the control are passed to the upper layer.

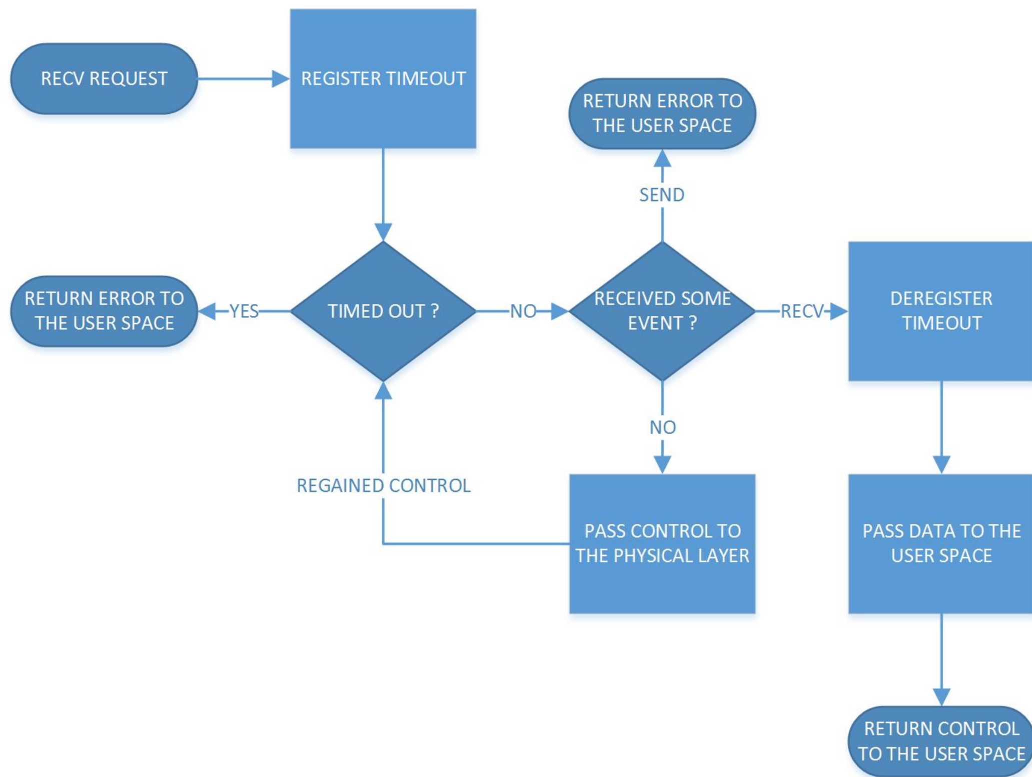


Figure 4.6: receive request handling by the mac sub-layer

5. `NOTIFY` Notify request means that the user wants to notify the upper layer about the result of the previous send request. If such request is claimed, the result together with the control are passed to the upper layer.

6. `WAIT` When a wait event is invoked, the MAC passes the control flow to the simulation scheduler and waits for any event that might occur. When it regains the control flow it notifies the caller about the type of the event that was received.

4.1.2.5 LLC built in sub-layer

As mentioned in 4.1.2.1, LLC built in is a sub-layer designed just for the user convenience. It provides the features that are not interesting from a simulation point of view. Nevertheless, if there is a need to replace the built-ins of the LLC layer in the future, all the features provides by this layer may be re-implemented by the user in the LLC user sub-layer. The provided features are:

- *addressing*: LLC built in provides basic addressing. The address is equal to the id of the node. LLC addressing feature includes dropping of frames that are received but not addressed to the node.
- *data acknowledgment*: Data acknowledgment is carried out using simple *ack* frames that contain just an acknowledgement flag and no data. These frames are sent by

the receiver to the sender after the receiver checks that the received data is all right. Data acknowledgement is managed on per-frame basis. That means that the upper layers decide whether the sent data should or should not be acknowledged for each send request.

- *buffering*: LLC built in buffers incoming frames that are for some reason not read immediately by the upper layer (RX queue), as well as it buffers the frames that should be sent and cannot be sent immediately (TX queue).

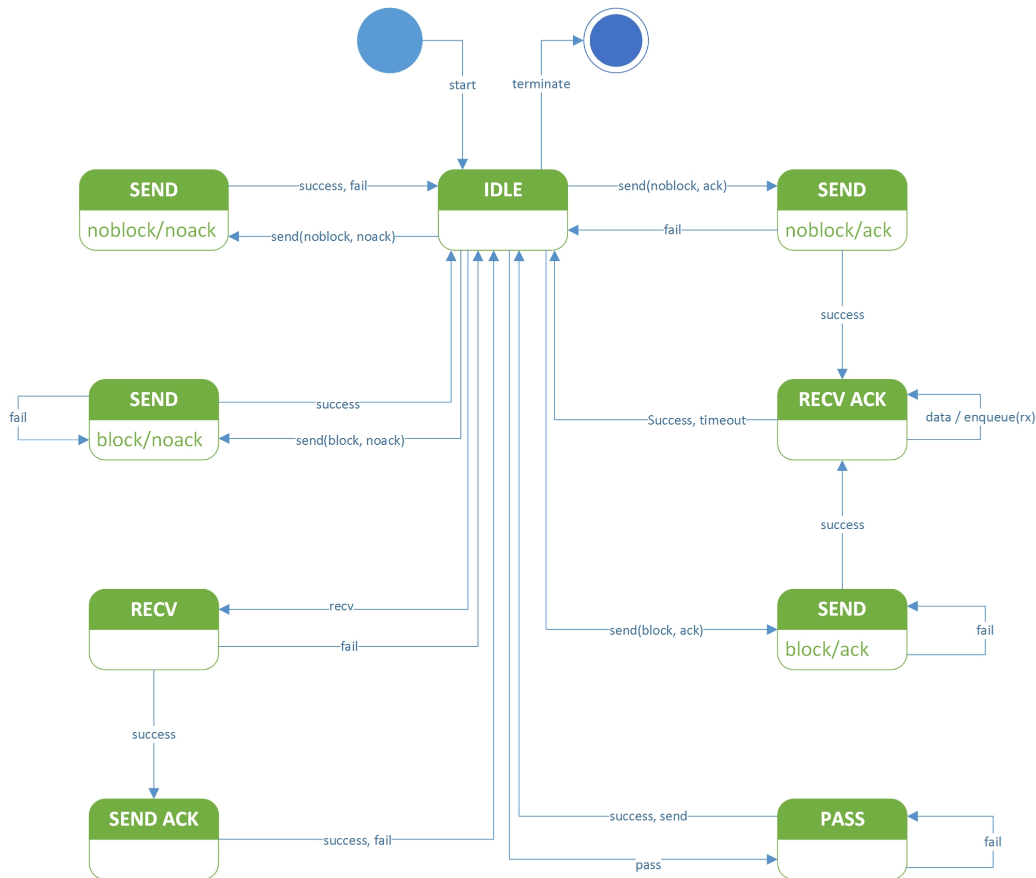


Figure 4.7: State machine describing the operation of the LLC built in sub-layer

State machine in figure 4.7 depicts the operation of the LLC built in sub-layer. Note that the state representing an error is not shown in order to keep the scheme of the state machine clear. Basically, upon any event that occurs, LLC state machine moves to a state designated for handling that event. The actual handling of the event consist of transformation of the data to an appropriate form, passing that data to an appropriate neighboring layer, swapping context to that layer and waiting for the result. Waiting for result means waiting until the control flow is regained and checking the result of the pending operation. This is straightforward in case of simple action like sending data without acknowledgement, where the control is regained almost immediately, since the only thing that needs to happen is the check whether medium is available (on MAC layer) and the only possible results are *success*

or *fail*. On the other hand, some operations, like sending with acknowledgement require more robust logic for the result checking. In this case LLC has to handle different events that may occur before the acknowledgement is received, for example reception of a different data (by enqueuing them in the RX queue). When there are no pending events to be handled, LLC state machine stays in the *idle* state.

List of possible events and description of their handling follows:

1. `SEND` The send event on LLC built in sub-layer implements the data acknowledgement feature and is available in following forms:

    - `SEND_NONBLOCKING_NOACK` An attempt to send the data is made and the result is immediately returned to the upper layer. Event handler for a simple send event works analogically to the one described in 4.1.2.4, which is depicted in 4.5.

    - `SEND_NONBLOCKING_ACK` An attempt to send the data is made. In case of failure, the result is immediately passed to the upper layer along with the control. In case of success, the *event handler*[5] waits for the acknowledgement frame. It returns a result along with the control to the upper layer when the acknowledgement is received or the waiting for acknowledgement times out. This process is depicted in figure 4.8.

    - `SEND_BLOCKING_NOACK` An attempt to send the data is made. In case of failure, the context is swapped to the simulator core. Then the event handler waits for control. This loop is repeated until the condition that prevents the event handler from sending the data vanishes. After the data is sent, the result is returned to the upper layer along with the control. This process is depicted in figure 4.9.

    - `SEND_BLOCKING_ACK` An attempt to send the data is made. In case of failure, the context is swapped to the simulator core. Then the event handler waits for control. This loop is repeated until the condition that prevents the event handler from sending the data vanishes. After the data is sent, the event handler waits for the acknowledgement frame. It returns a result together with the control to the upper layer when the acknowledgement is received or the waiting for acknowledgement times out. This process is depicted in figure 4.10.

2. `RECV` Incoming data are picked up from the event port and stored in the RX queue for the future use. If an acknowledgement is required for the incoming data, the event handler sends an acknowledgement in a way described in `SEND_NONBLOCKING_NOACK`. Then the control is passed to the simulation core. This process is depicted in figure 4.11.

3. `PASS` The control is immediately returned to the upper layer. Either with data, in case the RX queue is not empty, or with an error notification in the opposite case. This process is depicted in figure 4.12.

[5] event handler is a block of the simulator code designated for handling a specific event

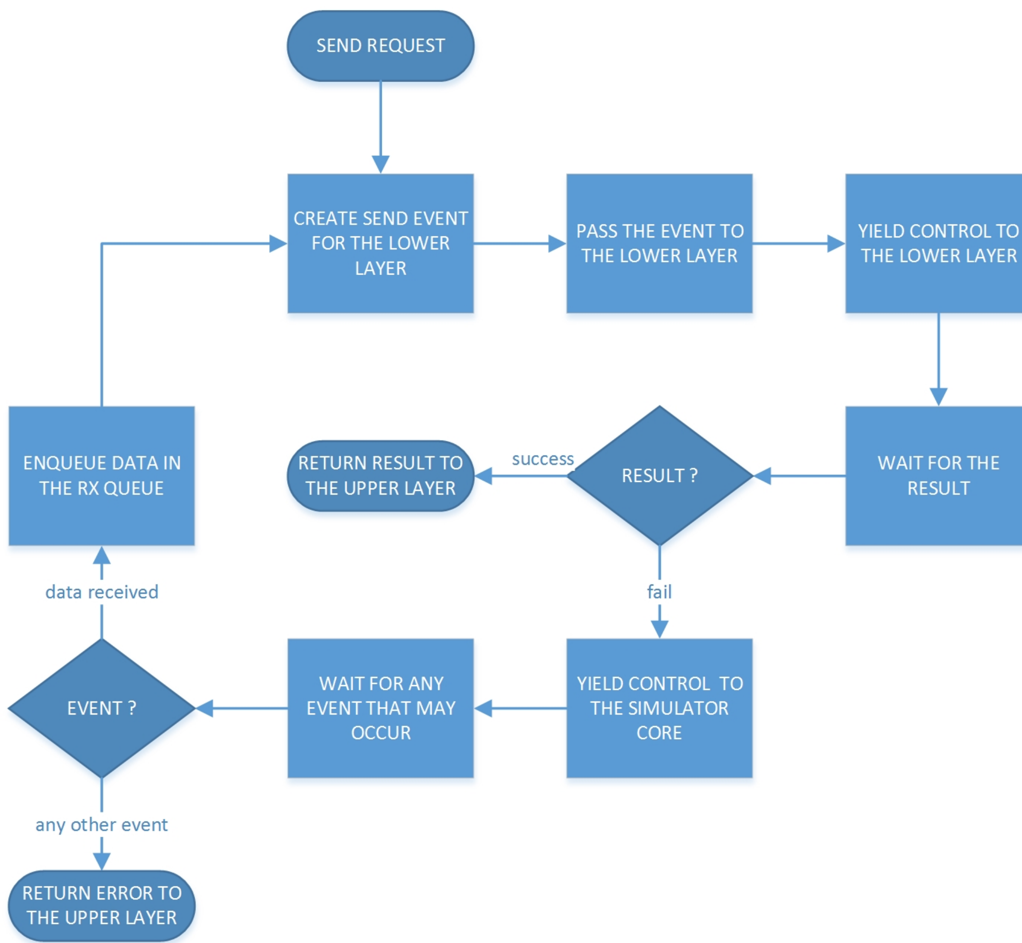


Figure 4.8: Flowchart describing handling of LLC blocking send event

4.1.2.6 LLC user controlled sub-layer

User controlled LLC sub-layer is the uppermost sub-layer of the data link layer. It's architecture is similar to all the other layers that are under user control. The user defined routine is executed by the execution unit assigned to the layer. This routine communicates with the rest of the simulation by requesting actions from the provided interface. The LLC user controlled sub-layer basically wraps the features implemented by 4.1.2.5, so that they are accessible to the user while the user still has a dedicated execution unit. The actions that can be performed from the user space are following:

1. SEND Same forms of send request are available as the ones defined in 4.1.2.5. LLC user layer just creates an appropriate event and passes it to the lower layer. Then the event handler waits for the result which is then returned to the user space.

2. RECV An event requesting data is passed to the lower layer along with the control flow. Than the event handler waits for any event that might occur. When the data event is received or when the user defined timer times out, the result is passed to the user

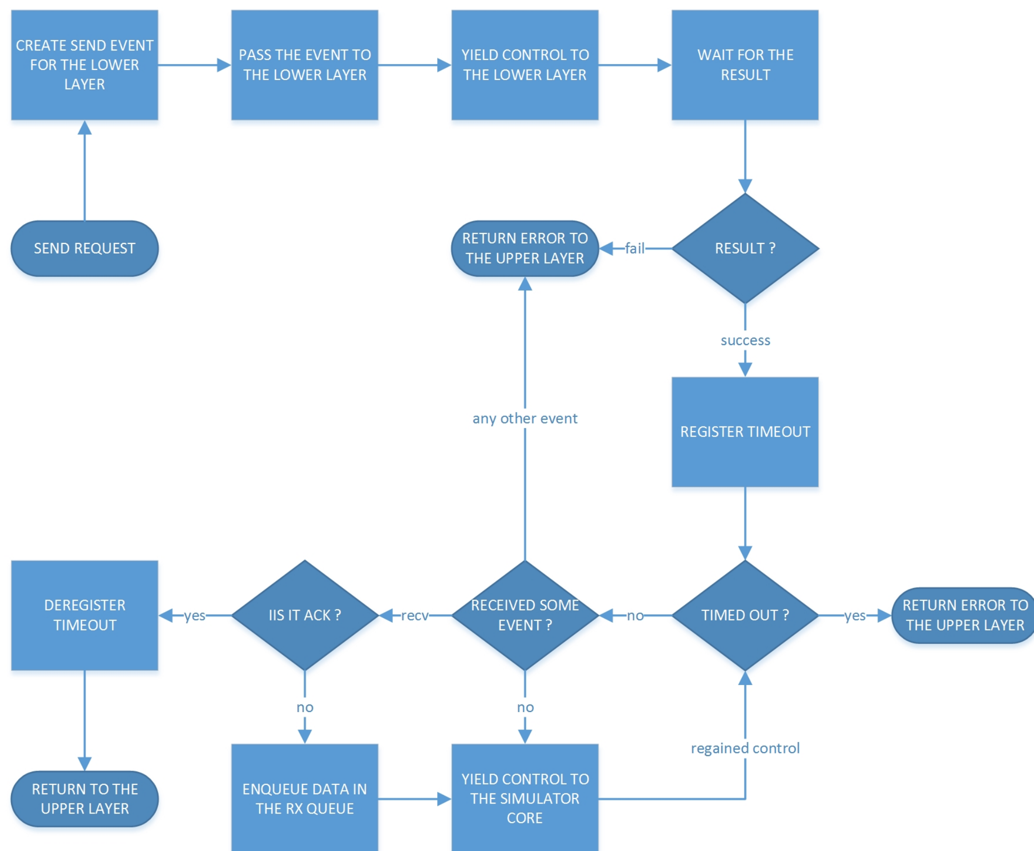


Figure 4.9: Flowchart describing handling of LLC send event with acknowledgement

space. Event handler for receiving event works analogically to the one described in 4.1.2.4, which is depicted in 4.6.

3. PASS Pass action always succeeds, because it only passes the control along with the user supplied data to the upper layer.

4. WAIT Wait action allows the routine in the user space perform conditional waiting[6]. The event handler just waits for any event. When an event occurs, the type of that event is returned to the user space.

4.1.2.7 Network layer

The network layer is the uppermost layer of the node model. As it is one of the user control layers, it's architecture is similar to all the other layers that are under user control. The user defined routine is executed by the execution unit assigned to the layer. This routine communicates with the rest of the simulation by requesting actions from the provided interface. The available actions on the network layer are:

---

[6]conditional waiting is the process where the calling execution unit is suspended until a specific condition occurs. An example may be timed waiting, where the condition is expiration of predefined amount of time

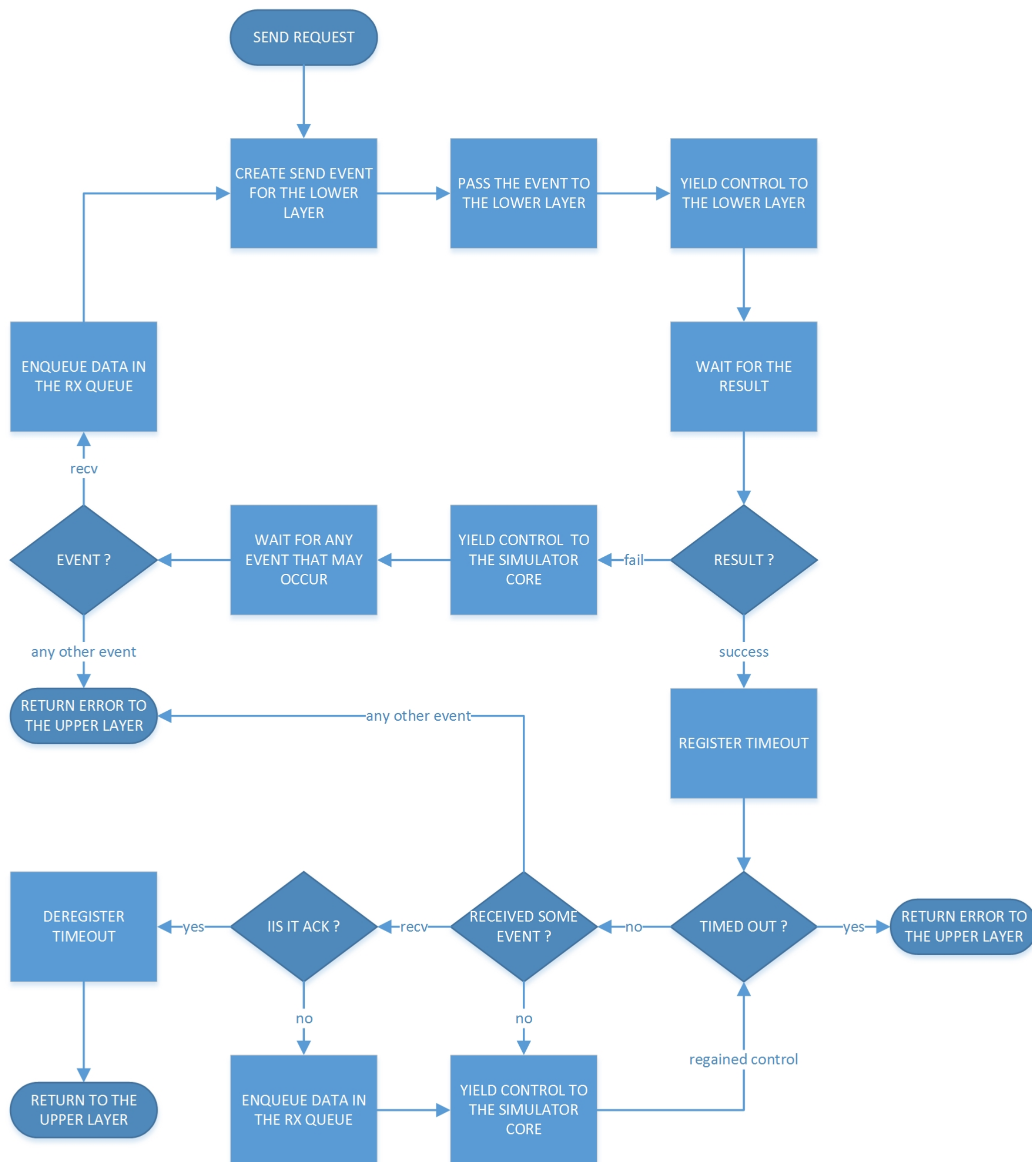


Figure 4.10: Flowchart describing handling of LLC blocking send event with acknowledgement

1. SEND: send event handler of the network layer works in an analogous way to the user LLC send event handler, which is described in 4.1.2.6. The action of the event handler consists of requesting send on the lower layer and waiting for the result.

2. RECV: Event handler for receiving event works analogically to the one described in 4.1.2.4, which is depicted in 4.6.

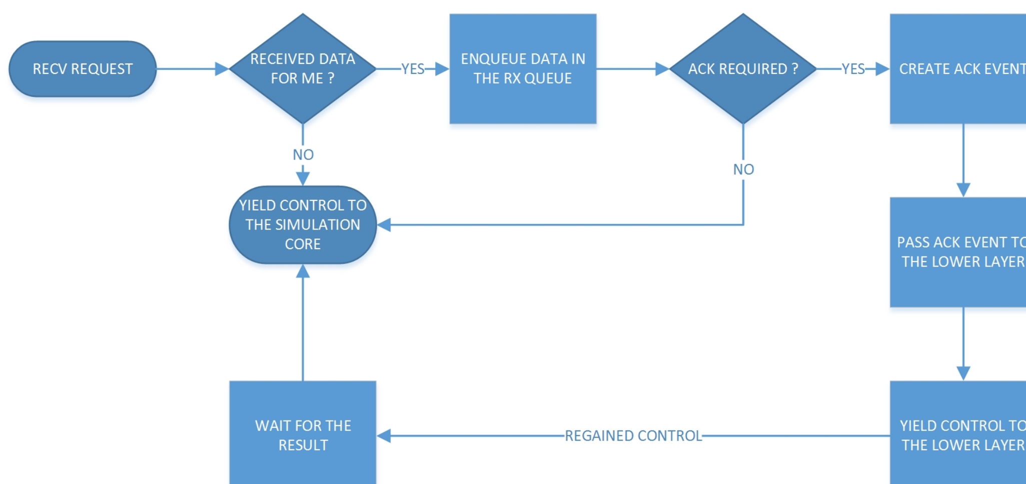


Figure 4.11: Flowchart describing handling of LLC recv event

4.1.3 Messages

All the structures representing layers of the simulated protocol stack incorporate an *inbox* designated for messages of any kind that the application in the user space may need to transfer between the nodes. This may become useful in case there is any information that needs to be delivered to a different node but there is a valid reason not to send this information as data through the simulation core. A typical use case of such messages is a simulation in which the user is not particularly interested in the contents of the frames that are sent between the network nodes (e.g. uses a data generator) but at the same time needs to transfer some control data between the nodes.

The messages are addressed to a node and a layer, are guaranteed to be delivered to the destination. The messages are delivered to the destination immediately, but the context is not swapped to the recipient. Therefore the recipient can "read" the message no sooner than the moment when the context is swapped to the recipient (an event is delivered to the recipient). The simulation core is in no way responsible for receiving the messages, i.e. the user defined routines **must** check the *inbox* for new messages periodically, because there is no other way to determine whether a new message has been received.

4.2 Implementation

This section describes technical details of the implementation. First it explains what inputs are needed for the simulation environment to run. It focuses on the details the user *must not* omit in order to preserve the correct operation environment, Then it describes what outputs user can expect. At it's end, this section defines what requirements need to be met in order for correct compilation, installation and execution of the simulator.

The goal of this work is to provide a core of a universal sensor network simulator. All the features that are implemented are discussed in 4.1. It is clear that a product of this work is not a full simulator. In order to perform a simulation, a user must provide at least

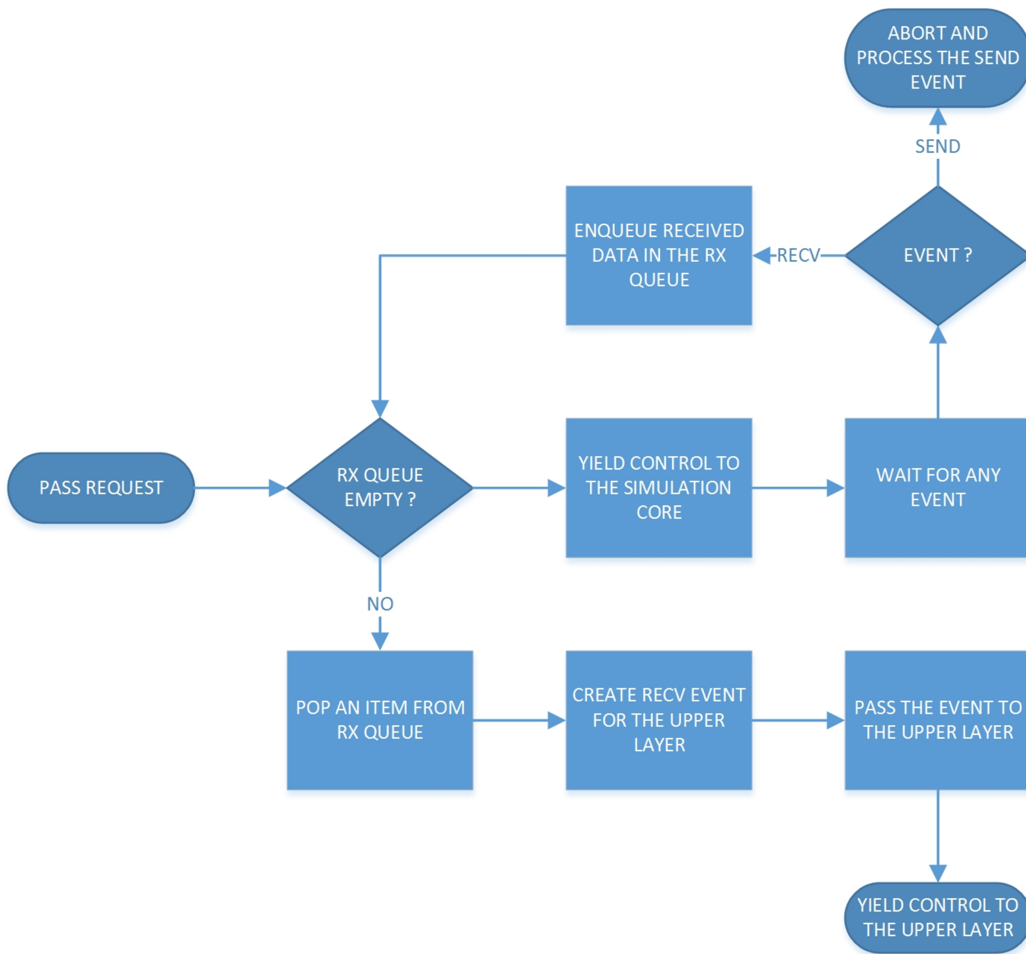


Figure 4.12: Flowchart describing handling of LLC pass event

implementation of the user defined layers. Therefore the core of the simulator is implemented as a program library.

Language chosen for the implementation is `C`. There exist simulation environments that are implemented in languages outside of `C/C++` family, for example the one described in [12]. These simulators may be suitable for some light weight application, but when it comes to performing heavy weight simulations (e.g. sensor network simulations), `C` is the most suitable choice, because it shows the shortest run times due to it's "closeness" to the operating system and hardware.

4.3 Input

Because the simulation core is implemented as a library, it cannot use direct inputs (e.g. configuration file), instead it takes it's inputs as parameters of the initialization call. There are three inputs that are necessary in order for the library to initialize correctly:

1. **transmission function** The simulation core uses the transmission function to compute the effects of a given data transmission in the network (actually it calculates the radiation pattern of an/a set of antenna/s). It gives the user complete freedom to implement any possible radiation pattern of a single antenna or even a virtual antenna array. Annotation of the transmission function is captured in listing 4.1. Transmission function is supposed to decide who are the transmitters and who are the receivers, what power should each transmitter use to transmit and what power each receiver receives when data is delivered between two nodes with given ids. Of course the function has access to the global simulation data. Note that transmission function takes a parameter of type `void *` designated for user data. This parameter may be used for example for beamforming control. This parameter originates in `send()` functions on user defined layers, so when a user calls the function `send()` on any layer with the given parameter, this parameter is passed to the transmission function.

```
1  typedef void (*pdsns_transmission_fun)   (
2  /* input: global pdsns structure */       pdsns_t *,
3  /* input: source id */                    uint64_t,
4  /* input: destination id */               uint64_t,
5  /* output: array of source nodes */       pdsns_node_t **,
6  /* output: transmission power of src */   double *,
7  /* output: number of sources */           size_t *,
8  /* output: array of dest nodes */         pdsns_node_t **,
9  /* output: transmission pwr of dsts */    double *,
10 /* output: number of destinations */      size_t *,
11 /* user data */                           void *
12                                           );
```

Listing 4.1: annotation of the transmission function

2. **neighborhood function** The neighbor function is very similar to the transmission function. The simulation core uses it to compute the neighborhood of each node during the initialization. Because the topology of any wireless network highly depends on the radiation pattern of all the antennas in the network, this function is also user defined. The neighborhood function is supposed to output all the nodes that are in reach of the inspected node a the power that is needed to reach each of the neighboring nodes. Unlike the transmission function, this function is only called during the initialization. The annotation of the neighborhood function is captured in listing 4.2.

```
1  typedef void (*pdsns_neighbor_fun)        (
2  /* input: global pdsns structure */       const pdsns_t *,
3  /* input: inspected node */               const pdsns_node_t *,
4  /* output: array of neighbors */          pdsns_node_t **,
5  /* output: neighbor pwr */                double *,
6  /* output: neighbor count */              size_t *
7                                            );
```

Listing 4.2: annotation of the neighborhood function

3. **Path to the network file** A network file defines basic parameters of all the nodes in the network. The network file uses XML format. The syntax is quite simple. The only two tags are `<network>` which is the wrapping tag and `<node>`. Node is a child of

network (in XML terms). Wheres <network> has no attributes, <node> has four mandatory attributes:

- **x** the first planar coordinate (signed integer type)
- **y** the second planar coordinate (signed integer type)
- **sensitivity** The smallest amount of power that the node is able to recognize as data (any smaller power is always considered noise). The data type is `double`.
- **maxpwr** maximal power the node is able to use for transmitting (double)

A simple example of a network file is captured in listing 4.3.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <network>
3     <node x="3" y="4" maxpwr="1.0" sensitivity="0.01"/>
4     <node x="-1" y="2" maxpwr="1.0" sensitivity="0.01"/>
5 </network>
```

Listing 4.3: network file

Before actually running a simulation the user must define the model that is actually going to be simulated. The library provides a skeleton of this model, but the user must define the details using the user routines for the user controlled layers. These routines are similar to the user routines of *pthreads* that are passed to `pthread_create()`[13] and are treated exactly in the same way. The only difference is, that the execution unit is not a pthread, but a GNU portable thread. These routines should be implemented with care and the user should anticipate that these routines are called on each node's corresponding layer as soon as the corresponding thread spawns. The routines should return no sooner than the end of simulation is signalized. Returning from such a routine implies the end of existence of the corresponding thread (and consequently the corresponding layer of the affected node). The annotation of these routines can be reviewed in listing 4.4

```
1 /* main function of the MAC sublayer */
2 typedef void (*pdsns_usr_mac_fun)        (pdsns_mac_t *);
3 /* main function of the LLC sublayer */
4 typedef void (*pdsns_usr_link_fun)       (pdsns_link_t *);
5 /* main function of the network layer */
6 typedef void (*pdsns_usr_net_fun)        (pdsns_net_t *);
```

Listing 4.4: user routines for the user controlled layers

It is very important to note that these routines are directly accessed by the `GNU pth` library and realize what consequences it brings to the user. Basically, these routines are only interrupted when they perform a blocking call (e.g. `SEND` or `RECV` or `WAIT`). There is no way to interrupt them from outside, as this is the nature of the coroutines[9]. Therefore these routines are required to behave "nicely" in a way that they deliberately yield control from time to time by calling some blocking calls. For the same reason these routines must respect the termination condition of the simulation. If the termination condition is disrespected, the routine is canceled in a similar way a unix process would be by calling `kill -SIGKILL`[29]. Expected behavior of a user routine is depicted by a flowchart in figure 4.13.

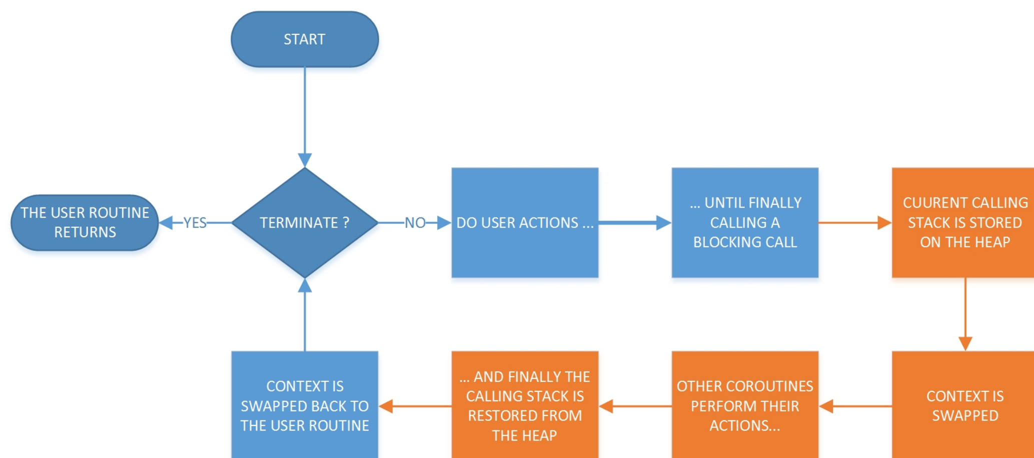


Figure 4.13: Flowchart describing operation of a user routine

Another important issue to take into consideration when implementing the user routines is the fact that the context actually gets swapped. As can be seen in figure 4.13, context swapping basically consist of storing of the current calling stack[7] on the heap[8], calling stack of the destination coroutine is restored and the context is swapped to the destination coroutine. This process is reversed when the control is returned to the user routine.

An important consequence of the stack overwriting is that during the period when other routine is in control, the stack of the user routine is invalidated, which among other things mean, that all the local variables of the user routine are invalid during that period. Therefore passing addresses (pointers) of local variable to the blocking calls **shall lead to the memory corruption**.

Thus the recommended approach to implementing the user routines is following. Instead of allocating variables on the stack, the user should define a structure containing all the variables the user wishes to have on the stack. When the user routine is invoked, an instance of this structure should be dynamically allocated on the heap. This approach is captured in listing 4.5. Addresses of variables that are inside such a structure are safe to be passed to the blocking calls.

```
1 struct usrstack
2 {
3     type1   *var1;
4     type2   *var2;
5     ...
6 };
7
```

[7]"In computer science, a call stack is a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack". Although maintenance of the call stack is important for the proper functioning of most software, the details are normally hidden and automatic in high-level programming languages. Many computer instruction sets provide special instructions for manipulating stacks."[23]

[8]Being an opposite to stack memory, that is allocated statically, heap memory is a memory space designated for dynamic allocations by the program.

```
8 void
9 usr_routine (some arguments)
10 {
11     struct usrstack     *stack;
12 
13     stack = (struct usrstack *)malloc(sizeof(struct usrstack));
14     ...
15     blocking_call(&stack->var1);
16     ...
17     free(stack);
18     return;
19 }
```

Listing 4.5: correct stack allocation by the user routine

4.4 Output

The library core is versatile enough to have virtually no clue about what the user intends to simulate and what outputs the user expects. It is therefore the responsibility of the user application to define and deliver the desired outputs. There is a sufficient infrastructure for the user allowing for the collection of the information about the simulation.

4.5 Portability

The simulation library can be compiled and linked against under any environment that meets following requirements:

- `Automake` The simulation library is built using `Automake` together with `libtool`. Therefore any target platform must provide `Automake` package. "Automake is a tool for automatically generating Makefile.in files compliant with the GNU Coding Standards. Automake requires the use of Autoconf."[1] For more information about `Automake` and `libtool` refer to [1].

- `GLib` "GLib provides the core application building blocks for libraries and applications written in C. It provides the core object system used in GNOME, the main loop implementation, and a large set of utility functions for strings and common data structures."[4]. The simulation library requires `GLib` in version `2.0` or later available on the target system. For more information about `GLib`, refer to [4].

- `libxml` "Libxml2 is the XML C parser and toolkit developed for the Gnome project (but usable outside of the Gnome platform), it is free software available under the MIT License. XML itself is a metalanguage to design markup languages, i.e. text language where semantic and structure are added to the content using extra "markup" information enclosed between angle brackets. HTML is the most well-known markup language. Though the library is written in C a variety of language bindings make it available in other environments."[19]. The simulation library requires `libxml` in version `2.0` or later available on the target system. For more information about `libxml`, refer to [19].

- `GNU pth` The GNU portable threads library is the library providing concurrency for the simulation core. It has already been discussed in 3.5. The simulation library requires `GNU pth` in version `2.0.7` or later available on the target system. For more information about `GNU pth`, refer to [9].

Chapter 5

Testing

This chapter describes the methods that were used to test the solution. First this section describes these methods in general, then it explains how they were used to ensure quality of the solution.

Testing is an important part of a software project life cycle as it is supposed to detect erroneous or any other undesired behavior before the software product is released. Chapter 5 describes the methods that were used to assure the quality of the software library that was described in chapter 4 and explains why these methods were used.

In order to assure quality of the software library, two phase testing has been employed:

1. **Unit testing:**"In computer programming, unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure. In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process."[35].

2. **Integration testing:** "Integration testing (sometimes called integration and testing, abbreviated I&T) is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing."[28]

5.1 Unit testing

There exist libraries that provide advanced unit testing for plain `C`, for example *Check*, *AceUnit*, or *GNU Autounit*, but these are designated for mission-critical or much larger

projects than this work actually is[11]. For purposes of this work, an old fashion style of writing simple application tests using `assert` is quite enough.

All the data structures that are defined and used in the library were subjected to unit testing. As an example of an unit test, consider a test of an universal queue that is used for example as a RX and TX queue on LLC built in layer (discussed in section 4.1.2.5) or as the *event set* of the simulation core (discussed in chapter 4.1.1). Unit test testing of the queue is captured in listing 5.1.

```
#include <assert.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include "libpdsns.h"

int
main (void)
{
    pdsns_queue_t    *q;
    int              *val, ret, i;


    /*************************** create ***************************************/
    q = pdsns_queue_init(free);
    if (q == NULL)
        perror("queue_init"), exit(EXIT_FAILURE);

    /* a new queue must be empty */
    assert(pdsns_queue_empty(q));

    /***************************** push ***************************************/

    if ((val = (int *)malloc(sizeof(int))) == NULL)
        perror("malloc"), exit(EXIT_FAILURE);

    *val = 42;
    ret = pdsns_queue_push(q, (void *)val);
    if (ret == PDSNS_ERR)
        perror("queue_push"), exit(EXIT_FAILURE);

    /* size must be 1 now */
    assert(pdsns_queue_size(q) == 1);

    if ((val = (int *)malloc(sizeof(int))) == NULL)
        perror("malloc"), exit(EXIT_FAILURE);

    *val = 666;
    ret = pdsns_queue_push(q, (void *)val);
    if (ret == PDSNS_ERR)
        perror("queue_push"), exit(EXIT_FAILURE);

    /* size must be 2 now */
    assert(pdsns_queue_size(q) == 2);

    /* i.e. queue is not empty */
    assert(! pdsns_queue_empty(q));
```

```
48
49     /***************************** pop ****************************************/
50     val = pdsns_queue_pop(q);
51
52     /* expecting a value now */
53     assert(val);
54
55     /* first out must be 42 */
56     assert(*val == 42);
57
58     /* size must be 1 */
59     assert(pdsns_queue_size(q) == 1);
60
61     free(val);
62
63     val = pdsns_queue_pop(q);
64
65     /* expecting a value now */
66     assert(val);
67
68     /* second out must be 666 */
69     assert(*val == 666);
70
71     /* size must be 0 */
72     assert(pdsns_queue_size(q) == 0);
73
74     /* i.e. queue is empty */
75     assert(pdsns_queue_empty(q));
76
77     free(val);
78
79     /**************************** misc ****************************************/
80     /* expecting no value from empty queue */
81     for (i = 0; i < 3; i++) {
82         val = pdsns_queue_pop(q);
83         assert(val == NULL);
84     }
85
86     /************************* cleanup ****************************************/
87     if ((val = (int *)malloc(sizeof(int))) == NULL)
88         perror("malloc"), exit(EXIT_FAILURE);
89
90     *val = 42;
91     ret = pdsns_queue_push(q, (void *)val);
92     if (ret == PDSNS_ERR)
93         perror("queue_push"), exit(EXIT_FAILURE);
94
95     /* expecting queue to clean up the value val, use valgrind to check */
96     pdsns_queue_destroy(q);
97     fprintf(stderr, "success\n");
98     exit(EXIT_SUCCESS);
99 }
```

Listing 5.1: unit test testing of an universal queue implemented by the library

When compiled and executed, such a test is expected to print `success` and return `0`. In

order to check that the memory is managed properly, the unit tests were run under *valgrind*[1]. The output of the test listed in 5.1 is shown below.

```
[user@hostname path]$ valgrind ./pdsns_queue_unit && echo $?
==13040== Memcheck, a memory error detector
==13040== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==13040== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==13040== Command: ./pdsns_queue_unit
==13040==
success
==13040==
==13040== HEAP SUMMARY:
==13040==     in use at exit: 0 bytes in 0 blocks
==13040==   total heap usage: 7 allocs, 7 frees, 92 bytes allocated
==13040==
==13040== All heap blocks were freed -- no leaks are possible
==13040==
==13040== For counts of detected and suppressed errors, rerun with: -v
==13040== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
0
```

5.2 Integration testing

As an integration test, a simple *"Hello World"* simulation using the simulation library has been implemented. Even though it is the simplest simulation that could be implemented using the simulation library, it's code is still quite extensive, due to the versatility of the library. Therefore the complete code of the *"Hello World"* simulation is not shown, nevertheless it is still part of the distribution package.

The simulated network consists of two nodes, one sending and the other receiving a "hello world" message. All the necessary user inputs are implemented as simple as possible and therefore do not model any real environment. These are used to test whether the simulation library work properly instead.

- neighborhood function creates a full topology (i.e. any node can directly transmit to any other node) where every node needs to use it's maximal power to reach any other node.
- transmission function returns all the nodes but the transmitting one as the recipients of any transmission. Receiving power of each node is equal to it's sensitivity.
- MAC and LLC user sub-layers just forward the request from neighboring layers.

[1] Valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of debugging and profiling tools. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure.[17]. It is commonly used to debug errors caused by poor memory management

- NET user layer designates one node for sending and the other nodes for receiving. The sending node sends data while the others wait for reception.

The full code of the "hello world" simulation is part of the distribution package.

Chapter 6

Conclusion

6.1 Conclusion

Aim of this work is to deliver a suitable simulation environment for simulating sensor networks. Namely, this work is focused on providing an environment for simulating routing algorithms designed specifically for sensor networks and for simulating distributed phase shift beamforming technique. These two simulation scenarios require the simulating environment to have specific attributes that are defined in chapter 2.

Existing solutions were analyzed and a conclusion was made that none of the analyzed solutions meets all the requirements defined by this work (refer to 3). Therefore a new approach combining the techniques of known solutions that were suitable to achieve the aim of this work, was employed. A standalone simulation library for sensor networks was designed and developed (refer to 4). On one hand, the final solution is versatile in terms of simulating sensor networks, on the other hand, the solution is not capable of simulating phenomena outside of the field of sensor (or similar) networks. Compliance of the final solution with the requirements specified in chapter 2 is summarized in following list:

1. *Advanced transmission control that allows simulation of advanced media access control techniques, such as distributed phase shift beamforming*: This point is satisfied by leaving the implementation of the radiation effects of a transmission on the user. User is also in control of `MAC` and `LLC` sub-layers of the data link layer. For further information, refer to 4.1.

2. *Independent model of each layer of the protocol stack that is easily exchangeable for a different one in the future*: This requirement is met by including only the simulation core into the final solution and leaving the large portion of the protocol stack for the user to implement. The layer of the protocol stack that is directly connected to the simulation core is the physical layer.

3. *Simple user interface of each layer of the protocol stack that allows the user to implement the protocol logic in a straightforward way*: The final solution implements process driven discrete event paradigm, which gives a user the "feeling" of continuous execution. Whereas other discrete event simulators expect the user to handle events when they emerge, the final solution lets the user handle events when convenient for the user.

4. *Reasonable simulation runtime even for large networks* Since this work only provides the simulation core, not the models of the phenomena to be simulated, it is virtually impossible make a proper measurement of the runtimes without actually implementing models for all the layers of the simulated protocol stack, which is not covered by this work. Therefore it cannot be determined at this moment whether this requirement is met.

6.2 Future work

This work only provides a simulation core, i.e. there are implemented no out of the box features, such as real protocols implementing layers of the simulated protocol stack. Readers are therefore encouraged to implement protocols used in sensor networks.

Bibliography

[1] *GNU Automake* [online]. Dostupné z: <http://www.gnu.org/software/automake/manual/>.

[2] BOUKERCHE, A. *ALGORITHMS AND PROTOCOLS FOR WIRELESS AND MOBILE AD HOC NETWORKS.* : John Wiley & Sons, Inc., 2009. ISBN 978-0-470-38358-2.

[3] BOUSSINNOT, F. Fair threads in C, 2002.

[4] BREUER, H. *GLib* [online]. Dostupné z: <https://developer.gnome.org/glib/>.

[5] *Castalia* [online]. Dostupné z: <http://castalia.research.nicta.com.au/index.php/en/>.

[6] CERNY, V. – MOUCHA, A. – KUBR, J. Interference cancellation by the usage of distributed phase shift beamforming, 2014.

[7] DIGI. XBee series 2 OEM RF modules, 2008.

[8] DIGI. Demystifying 802.15.4 and ZigBee, 2007.

[9] ENGELSCHALL, R. *GNU Portable Threads* [online]. Dostupné z: <http://www.gnu.org/software/pth/>.

[10] KROLLER, D. et al. Shawn: A new approach to simulating wireless sensor networks, 2005.

[11] MALEC, A. *Unit testing in C* [online]. Dostupné z: <http://check.sourceforge.net/doc/check_html/check_2.html>.

[12] MATLOFF, N. Introduction to Discrete-Event Simulation and the SimPy Language, 2008.

[13] *Pthread* [online]. Dostupné z: <http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html>.

[14] RUHRUP, S. Theory and Practice of Geographic Routing, 2009.

[15] *Setcontext* [online]. Dostupné z: <http://pubs.opengroup.org/onlinepubs/007908775/xsh/getcontext.html>.

[16] *Setjmp* [online]. Dostupné z: <http://pubs.opengroup.org/onlinepubs/009695399/functions/setjmp.html>.

[17] SEWARD, J. Valgrind, 2002. Dostupné z: <http://valgrind.org>.

[18] SHENE, K. *Building coroutines* [online]. 2001. Dostupné z: <http://www.csl.mtu.edu/cs4411.ck/www/NOTES/non-local-goto/>.

[19] VEILLARD, D. *The XML C parser and toolkit of Gnome* [online]. Dostupné z: <http://xmlsoft.org/>.

[20] WAGNER, D. – WATTENHOFFER, R. *Algorithms for Sensor and Ad Hoc Networks*. : Springer-Verlag, 2007. ISBN 978-3-540-74990-5.

[21] *Antenna (radio)* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Antenna_(radio)>.

[22] *Beamforming* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Beamforming>.

[23] *Call stack* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Call_stack>.

[24] *Computer simulation* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Computer_simulation>.

[25] *Control flow* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Control_flow>.

[26] *Coroutine* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Coroutine>.

[27] *Free-space path loss* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Free-space_path_loss>.

[28] *Integration testing* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Integration_testing>.

[29] *kill (command)* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Kill_(command)>.

[30] *Logical link control* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Logical_link_control>.

[31] *OSI model* [online]. Dostupné z: <http://en.wikipedia.org/wiki/OSI_model>.

[32] *Tcl* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Tcl>.

[33] *Thread pool pattern* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Thread_pool_pattern>.

[34] *TinyOS* [online]. Dostupné z: <http://en.wikipedia.org/wiki/TinyOS>.

[35] *Unit testing* [online]. Dostupné z: <http://en.wikipedia.org/wiki/Unit_testing>.

[36] *ZigBee* [online]. Dostupné z: <http://en.wikipedia.org/wiki/zigbee>.

[37] YU, F. A Survey of Wireless Sensor Network Simulation Tools, 2011.

Appendix A

Installation guide

Software library implementing the simulation core is called *libpdsns*. Libpdsns is supposed to be built using `automake` [1] and is distributed together with a standard configuration package for `automake`. The generic installation instructions (that also apply for libpdsns) are available in the `INSTALL` file of the package. These instruction are also available online at [1] and for complete clarity also mentioned below.

A.1 Recommended installation

1. Download or copy the installation package. preferably to a location like `/tmp` since the installation files are used only once during installation and are needed no more after the installation is finished.

2. run the `./autogen.sh` script.The output should look similar to this:

```
[user@hostname tmp]$ ./autogen.sh
libtoolize: putting auxiliary files in `.'.
libtoolize: copying file `./ltmain.sh'
libtoolize: Consider adding `AC_CONFIG_MACRO_DIR([m4])' to conf
igure.ac and
libtoolize: rerunning libtoolize, to keep the correct libtool m
acros in-tree.
libtoolize: Consider adding `-I m4' to ACLOCAL_AMFLAGS in Makef
ile.am.
configure.ac:5: installing `./config.guess'
configure.ac:5: installing `./config.sub'
configure.ac:2: installing `./install-sh'
configure.ac:2: installing `./missing'
src/Makefile.am: installing `./depcomp'
[user@hostname tmp]$
```

3. run the `./configure` script. The configure script has a lot of options. Consider using at least `-prefix=/some/nonsystem/path` because the path usually defaults

to /usr/lib which is default place for user libraries. Since the only application using libpdsns is going to be your simulation application, consider installing libpdsns to some custom path. The output should look similar to this:

```
[user@hostname tmp]$ ./configure --prefix=/opt/libpdsns
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a sed that does not truncate output... /bin/sed
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for fgrep... /bin/grep -F
checking for ld used by gcc... /usr/bin/ld
checking if the linker (/usr/bin/ld) is GNU ld... yes
checking for BSD- or MS-compatible name lister (nm)... /usr/bin
/nm -B
checking the name lister (/usr/bin/nm -B) interface... BSD nm
checking whether ln -s works... yes
checking the maximum length of command line arguments... 196608
0
checking whether the shell understands some XSI constructs... y
es
checking whether the shell understands "+="... yes
checking for /usr/bin/ld option to reload object files... -r
checking for objdump... objdump
checking how to recognize dependent libraries... pass_all
checking for ar... ar
checking for strip... strip
checking for ranlib... ranlib
checking command to parse /usr/bin/nm -B output from gcc object
... ok
```

```
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for dlfcn.h... yes
checking for objdir... .libs
checking if gcc supports -fno-rtti -fno-exceptions... no
checking for gcc option to produce PIC... -fPIC -DPIC
checking if gcc PIC flag -fPIC -DPIC works... yes
checking if gcc static flag -static works... no
checking if gcc supports -c -o file.o... yes
checking if gcc supports -c -o file.o... (cached) yes
checking whether the gcc linker (/usr/bin/ld -m elf_x86_64) sup
ports shared libr
aries... yes
checking whether -lc should be explicitly linked in... no
checking dynamic linker characteristics... GNU/Linux ld.so
checking how to hardcode library paths into programs... immedia
te
checking whether stripping libraries is possible... yes
checking if libtool supports shared libraries... yes
checking whether to build shared libraries... yes
checking whether to build static libraries... yes
checking for pth_init in -lpth... yes
checking for xmlReadFile in -lxml2... yes
checking for g_hash_table_new in -lglib-2.0... yes
checking size of time_t... 8
checking size of size_t... 8
checking size of long... 8
checking size of int... 4
checking size of short... 2
checking size of char... 1
configure: creating ./config.status
config.status: creating Makefile
config.status: creating doc/Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: executing depfiles commands
config.status: executing libtool commands
```

```
[user@hostname tmp]$
```

4. Build the library using `make` command. The output should look similar to this:

```
[user@hostname tmp]$ make
make  all-recursive
make[1]: Entering directory `/tmp/trunk'
Making all in doc
make[2]: Entering directory `/tmp/trunk/doc'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/tmp/trunk/doc'
Making all in src
make[2]: Entering directory `/tmp/trunk/src'
/bin/sh ../libtool --tag=CC   --mode=compile gcc -std=gnu99 -DH
AVE_CONFIG_H -I. -I..     -Wall -Werror -O0 -ggdb  -Wall -Werro
r -O0 -ggdb -I/usr/include/libxml2 `pkg-config --cflags glib-2.
0` -MT libpdsns.lo -MD -MP -MF .deps/libpdsns.Tpo -c -o libpdsn
s.lo libpdsns.c
libtool: compile:  gcc -std=gnu99 -DHAVE_CONFIG_H -I. -I.. -Wal
l -Werror -O0 -ggdb -Wall -Werror -O0 -ggdb -I/usr/include/libx
ml2 -I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include -MT l
ibpdsns.lo -MD -MP -MF .deps/libpdsns.Tpo -c libpdsns.c  -fPIC
-DPIC -o .libs/libpdsns.o
libtool: compile:  gcc -std=gnu99 -DHAVE_CONFIG_H -I. -I.. -Wal
l -Werror -O0 -ggdb -Wall -Werror -O0 -ggdb -I/usr/include/libx
ml2 -I/usr/include/glib-2.0 -I/usr/lib64/glib-2.0/include -MT l
ibpdsns.lo -MD -MP -MF .deps/libpdsns.Tpo -c libpdsns.c -o libp
dsns.o >/dev/null 2>&1
mv -f .deps/libpdsns.Tpo .deps/libpdsns.Plo
/bin/sh ../libtool --tag=CC   --mode=link gcc -std=gnu99  -Wall
 -Werror -O0 -ggdb  -Wall -Werror -O0 -ggdb -I/usr/include/libx
 ml2 `pkg-config --cflags glib-2.0`   -o libpdsns.la -rpath /tm
 p/libpdsns/lib libpdsns.lo  -lglib-2.0 -lxml2 -lpth
libtool: link: gcc -shared  .libs/libpdsns.o   -lglib-2.0 -lxml
2 -lpth    -Wl,-soname -Wl,libpdsns.so.0 -o .libs/libpdsns.so.0
.0.0
libtool: link: (cd ".libs" && rm -f "libpdsns.so.0" && ln -s "l
ibpdsns.so.0.0.0" "libpdsns.so.0")
libtool: link: (cd ".libs" && rm -f "libpdsns.so" && ln -s "lib
pdsns.so.0.0.0" "libpdsns.so")
libtool: link: ar cru .libs/libpdsns.a  libpdsns.o
libtool: link: ranlib .libs/libpdsns.a
libtool: link: ( cd ".libs" && rm -f "libpdsns.la" && ln -s "..
/libpdsns.la" "libpdsns.la" )
make[2]: Leaving directory `/tmp/trunk/src'
make[2]: Entering directory `/tmp/trunk'
```

```
make[2]: Leaving directory `/tmp/trunk'
make[1]: Leaving directory `/tmp/trunk'
[user@hostname tmp]$
```

5. And finally install the library to the path you previously specified by the `-prefix` argument to the `./configure` command, using `make install` command. Note that you must have a permission to write the target path, so when installing to standard system paths, you may need to become root beforehand. The output should look something like this:

```
[user@hostname tmp]$ make install
Making install in doc
make[1]: Entering directory `/tmp/trunk/doc'
make[2]: Entering directory `/tmp/trunk/doc'
make[2]: Nothing to be done for `install-exec-am'.
test -z "/tmp/libpdsns/share/doc/libpdsns" || /bin/mkdir -p "/t
mp/libpdsns/share/doc/libpdsns"
 /usr/bin/install -c -m 644 dip.pdf '/tmp/libpdsns/share/doc/li
 bpdsns'
make[2]: Leaving directory `/tmp/trunk/doc'
make[1]: Leaving directory `/tmp/trunk/doc'
Making install in src
make[1]: Entering directory `/tmp/trunk/src'
make[2]: Entering directory `/tmp/trunk/src'
test -z "/tmp/libpdsns/lib" || /bin/mkdir -p "/tmp/libpdsns/lib
"
 /bin/sh ../libtool   --mode=install /usr/bin/install -c   libp
 dsns.la '/tmp/libpdsns/lib'
libtool: install: /usr/bin/install -c .libs/libpdsns.so.0.0.0 /
tmp/libpdsns/lib/libpdsns.so.0.0.0
libtool: install: (cd /tmp/libpdsns/lib && { ln -s -f libpdsns.
so.0.0.0 libpdsns.so.0 || { rm -f libpdsns.so.0 && ln -s libpds
ns.so.0.0.0 libpdsns.so.0; }; })
libtool: install: (cd /tmp/libpdsns/lib && { ln -s -f libpdsns.
so.0.0.0 libpdsns.so || { rm -f libpdsns.so && ln -s libpdsns.s
o.0.0.0 libpdsns.so; }; })
libtool: install: /usr/bin/install -c .libs/libpdsns.lai /tmp/l
ibpdsns/lib/libpdsns.la
libtool: install: /usr/bin/install -c .libs/libpdsns.a /tmp/lib
pdsns/lib/libpdsns.a
libtool: install: chmod 644 /tmp/libpdsns/lib/libpdsns.a
libtool: install: ranlib /tmp/libpdsns/lib/libpdsns.a
libtool: finish: PATH="/usr/lib64/qt-3.3/bin:/usr/local/bin:/us
r/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/davak/omnet/om
netpp-4.4/bin:/home/davak/bin:/home/davak/omnet/omnetpp-4.4/bin
:/sbin" ldconfig -n /tmp/libpdsns/lib
```

```
----------------------------------------------------------------------
Libraries have been installed in:
   /tmp/libpdsns/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,-rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.c
   onf'

See any operating system documentation about shared libraries f
or
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
test -z "/tmp/libpdsns/include" || /bin/mkdir -p "/tmp/libpdsns
/include"
 /usr/bin/install -c -m 644 libpdsns.h '/tmp/libpdsns/include'
make[2]: Leaving directory `/tmp/trunk/src'
make[1]: Leaving directory `/tmp/trunk/src'
make[1]: Entering directory `/tmp/trunk'
make[2]: Entering directory `/tmp/trunk'
make[2]: Nothing to be done for `install-exec-am'.
test -z "/tmp/libpdsns/share/doc/libpdsns" || /bin/mkdir -p "/t
mp/libpdsns/share/doc/libpdsns"
 /usr/bin/install -c -m 644 README '/tmp/libpdsns/share/doc/lib
 pdsns'
make[2]: Leaving directory `/tmp/trunk'
make[1]: Leaving directory `/tmp/trunk'
[user@hostname tmp]$
```

6. Enjoy!