Poznań University of Technology
Faculty of Computing Science
Institute of Computing Science

Bachelor's thesis

DESIGN AND IMPLEMENTATION OF COMMUNICATION PROTOCOLS FOR SELF-ORGANIZING MULTI-HOP AD HOC NETWORKS USING XBEEPRO-868 PLATFORM

Amadeusz Juskowiak, 106453
Tomasz Kuczma, 106548
Mateusz Rybarski, 106572
Maciej Żurad, 106537

Supervisor
Professor Jerzy Brzeziński

Poznań, 2015

Temat
pracy dyplomowej inżynierskiej
nr

[redacted]

Politechnika Poznańska

Wydział Informatyki
Instytut Informatyki

[redacted]

Studia stacjonarne I stopnia

Kierunek: Informatyka
Specjalność: -

Zobowiązuję/zobowiązujemy się samodzielnie wykonać pracę w zakresie wyspecyfikowanym niżej. Wszystkie elementy (m.in. rysunki, tabele, cytaty, programy komputerowe, urządzenia itp.), które zostaną wykorzystane w pracy, a nie będą mojego/naszego autorstwa będą w odpowiedni sposób zaznaczone i będzie podane źródło ich pochodzenia.

[redacted]	Imię i nazwisko	Nr albumu	Data i podpis
Student:	Amadeusz Juskowiak	106453	
Student:	Tomasz Antoni Kuczma	106548	
Student:	Mateusz Rybarski	106572	
Student:	Maciej Marcin Żurad	106537	

[redacted]	
Tytuł pracy:	Projekt i implementacja protokołów komunikacyjnych dla samoorganizujących się wieloetapowych sieci ad hoc z użyciem platformy sprzętowej XbeePRO-868
Wersja angielska tytułu:	*Design and implementation of communication protocols for self-organizing multi-hop ad hoc networks using XbeePRO-868 platform*
Dane wyjściowe:	Literatura na temat bezprzewodowych mobilnych sieci ad hoc. Literatura, standardy oraz dokumentacja techniczna protokołów routingu dla bezprzewodowych i mobilnych sieci ad hoc. Dokumentacja techniczna dla platformy sprzętowej XbeePRO-868
Zakres pracy:	Projekt protokołów komunikacyjnych wraz z dowodami ich poprawności; implementacja protokołów w języku C++; przygotowanie platformy sprzętowej oraz testy efektywnościowe opracowanych rozwiązań.
Miejsce prowadzenia prac:	Instytut Informatyki Politechniki Poznańskiej
Termin oddania pracy:	30.01.2015 r.
Promotor:	prof. dr hab.inż. Jerzy Brzeziński

Z-ca Dyrektora Instytutu Informatyki
ds. Kształcenia
dr hab. inż. Maciej Zakrzewicz, prof. nadzw.

Dyrektor Instytutu

Dziekan

Poznań,

Miejscowość, data

Acknowledgements

Foremost, we would like to express our sincere gratefulness to Prof. Jerzy Brzeziński for his guidance, motivation and immense knowledge. His counselling helped us during research and writing of this thesis.

Besides our supervisor, we are pleased to thank all the members of Distributed Systems Group, especially Dr. Michał Kalewski. Without them this thesis would not have come into being. We could not forget about thanking the rest of Poznan University of Technology community. Over the years, their mentoring was inevitable process which consolidated our life goals.

Moreover, we would like to thank two companies, Amano Interactive and Chembus, for the inspiration and generous support.

Last but not least, we feel need to express gratitude to our parents, families and friends. They did believe in us and their labour formed each of us as beings capable of expressing themselves.

The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up.

His work is like that of the planter — for the future.
His duty is to lay the foundation for those who are to come, and point the way.

NIKOLA TESLA

Abstract

Wireless ad hoc networks consists of autonomous hosts which have equal status within a network. Unlike managed wireless networks, ad hoc networks are decentralized, there is no pre existing structure. Each host can participate in routing by forwarding packets. Decentralized nature of ad hoc networks makes them more reliable than managed wireless networks, because failure of one host does not cause failure of whole network. This feature makes them also more suitable for some kinds of applications, like sensoric networks.

Message delivery protocols are fundamental for applications that use the network, especially when it comes to sending messages to host that is not directly accessible. Assuming that hosts can be spread over a large area and distance between any pair of adjacent hosts can be large, temporary failures of links and network topology changes should be taken into account. This makes delivery guarantee a hard problem. Another problem is to provide self-organizing network. Joining to network should be as simple as possible and maintaining topology should happen in automatic way.

In context of the above observations, in this thesis we propose new communication protocols that provide self-organizing networks and try to ensure delivery of messages between any pair of hosts in a network. Moreover, we provide sample implementation of these protocols. Firstly, motivation and available solutions are shown, then designed protocols are described in Chapter 2. Chapters 3 and 4 contain detailed description of provided implementation of protocols and sample applications respectively. Then, in Chapter 5, different kinds of tests have been carried out.

Test results confirmed that provided solutions and implementation are in line with expectations and assumptions.

Contents

Chapter 1

Introduction

In recent years there is a growing demand for wireless communication, mostly caused by an increasing number of mobiles devices, such as smartphones, tablets, smart watches and home automation appliances. Lately these devices has got more popular, due to technological advances in hardware that resulted in lower prices and therefore more accessibility. Usually communication between those devices is achieved with the help of *Internet*, to which they connect through a wireless or cellular network. On the other hand direct communication is rarely used, even if a group of devices is in close neighbourhood. This solution has its advantages, such as limited responsibility. The mere thing they are responsible for is handling the communication with a device connecting it to the Internet. However, the massive disadvantage comes from relying on this provided infrastructure, which when fails then no communication can be performed. In order to not depend on any pre-established infrastructure another approach for enabling communication is needed.

One well described solution could be an *ad hoc* network [Per08]. Ad hoc is a Latin phrase for *"Formed for or concerned with specific purpose"*, *"Improvised and often impromptu"* [Dic11]. Ad hoc network is a decentralized network, where every device connected to it has the same responsibilities and can act both as a client and server simultaneously. Ad hoc networks can be divided into two categories *single-hop* and *multi-hop*, single-hop implies that the communication between two hosts is done without the help of intermediate hosts, while multi-hop network allows for communication assisted by intermediate hosts. Such a network has a possibility of having a much greater geographical span compared to a single-hop approach, because two hosts do not need to be in transmission range. However, there are substantial consequences of this approach. Now each host is responsible for routing traffic between other hosts. This demands all hosts to posses knowledge about the current network topology, but since it is decentralized network, it brings all the problems that can appear in distributed system.

Another form of comparing different ad hoc networks is to look at how they handle topology changes. A *self-organizing* [Poo00] network reacts to those alterations without any assistance, it automatically discovers new hosts willing to join the network as well as drops the hosts that are not responding. These differences are then flooded [LK01] throughout the network to ensure that all hosts inside the network are notified about the changes. In this thesis a self-organizing multi-hop ad hoc network is studied. Since ad hoc networks are usually deployed on small wireless devices, they usually have plenty of limitations on resources such as battery power, bandwidth, CPU's computational power and memory capacity. These constraints have to be taken into account, while designing such networks.

As a result of being independent from any pre-established infrastructure, ad hoc networks find many applications. One example is military, which nowadays uses computers extensively in

battlefields. Because of this, features like fault tolerance and on demand setup are of utmost importance, and therefore the use of ad hoc networks for army purposes is widely studied [JJV08]. It is worth nothing that in case of army application, an important assumption of high mobility has to be included. Ad hoc networks where each node can move without any constraints is known as *MANET* [BRG99], an abbreviation for mobile ad hoc network. Another excellent example of application that would greatly benefit from ad hoc approach is post disaster relief. In this case, there is also close to zero infrastructure availability, caused by flood, earthquake or other catastrophic event. Yet another application for instance is wireless sensor network monitoring gas pipes that are spread across a country, where access to cellular network can be limited or impossible. Further adoptions worth mentioning are gaming on portable devices, ubiquitous flea market [SNS03] as well as assistance during official meetings. More mobile games nowadays feature multi-player modes allowing to play with friends nearby. With the use of ad hoc network, rather than connecting to a server on the Internet, players can create a single-hop ad hoc network and communicate that way. A ubiquitous flea market is a concept, where users could buy and sell items. As they walk, application would search inside the network for an item match, and upon match request users to make a physical contact. Official meetings or school classes can also take advantage of ad hoc technology. People can easily exchange data within a network, that would be created dynamically just for this purpose. Of course security is something that has to be considered. Finally, Internet of Things *(IoT)* is a raising area of research. Its intention is to create a massive network of inter connected devices using existing infrastructure such as Internet. However, over the last decade use ad hoc networks was investigated to supplement the Internet [RTB+13]. The word *Things* from the definition refers to broad range of devices, including heart monitoring implants, energy management sensing and actuation systems, home automation systems and environment monitoring systems. These are just a few examples from a wide group of real world applications.

When studying currently available products based on ad hoc network, it can be observed that the market is still in its infancy. Moreover, there exist a number of available solutions to different problems that rely on existing infrastructure. Combining those two facts allows for creating a product, which would not only innovate, but could also compete with a lower price. One example of it, would be a modern timing system used in different sport events, such a marathons and triathlons. Main task of this system is to exchange data between points placed within the race route. These points would collect timestamps of each participant passing by. Data would be aggregated in real-time at a single node in order to calculate intermediate and lap times as well as reveal any cheaters. In already existing solutions data is sent by GPRS modules (for an extra fee paid to GSM operator). Therefore, not only it depends on the infrastructure, but also generates more costs associated with data transfers. Developing a modern timing system built with ad hoc network as its base for communication, would therefore be more resistant to failures and additional redundancy would increase this resistance even further. Data transfer costs can also be removed depending on the type of communication used between two adjacent nodes. Using a publicly available frequency band is an example of this. There is currently no timing system on the market, which is based on ad hoc approach. The reason for this is mostly due to the fact that creating such product comes with more responsibilities. Finally, it is the software or hardware of the device that is responsible for routing packets throughout the network, since there is no existing infrastructure. Moreover, ad hoc networks face many other problems such as security and manufacturing a final product that clients would be happy with is not an easy task. However, for a timing system, weaker assumptions can be made, which greatly simplify the design of the network, for instance: predefined shared key required to join network and no network mobility.

That being said, motivation for this thesis came precisely from observing the market of cur-

rent timing systems and noticing that an ad hoc approach could seriously disrupt it. Therefore, the goal of this thesis is to design and implement communication protocols that implement self-organizing multi-hop long distance ad hoc network. Long distance means that it allows for a long distance communication between two adjacent nodes. Together with multi-hop feature of the network, it implies that the network can have an immense geographical span. Consequently, these protocols would fit extremely well as a core of a modern timing system used in sport competitions, where gap between two points can be large. In order to support long range communication, radio modules from Digi International [Dig15], namely XBeePro series were chosen as a base for transmission. There are two available models, which operate on two different frequencies. First one being XBeePro-868, that uses 868MHz frequency and the second, XBeePro-900 that works on 900MHz frequency. The latter one implements a proprietary mesh network protocol, where each node acts as a router [JS03]. However, in Europe there is legal problem. European Union directive 87/372/EWG [Eur09, Eur11] does not allow public access on 900MHz frequency. In that case to create mesh network using XBeePro in Europe it is necessary to implement own multi-hop ad hoc network and use XBeePro-868 [Dig14b].

Knowing the hardware, the network is going to operate on, following assumptions were made in order to reduce complexity of the project. No network mobility, which otherwise would complicate routing and discovery protocols. Security of the network that relies on the hardware and users, meaning XBeePro-868 modules are responsible for encryption and that only authorized people know the password. Therefore, this network could not be used for applications that involve public access, because the protocol does not solve typical security problems such as Man-in-the-Middle and Denial of Service attacks [HBC01]. Another important assumption concerns network partitions. Partition occurs if during operation of network there exist two nodes without a path. In the design of the protocol it is assumed that permanent network partition cannot arise.

After determining assumptions, these functional and non-functional requirements were defined to ensure that the designed protocols could be used in different applications, for instance previously described timing system or for that matter any data collecting system, that would have static nodes (node with no mobility). First important functional requirement is that system shall handle topology changes. Meaning new nodes joining the network shall get notified about current network topology and vice versa. It also implies that temporary failures of nodes or links will be noticed and propagated throughout the network as well. Second functional requirement concerns packet delivery, and it states that the protocol shall deliver a packet sent from node A to node B eventually. However, it can deliver the same packet more than once. On the other hand non-functional requirements are qualities of the system. One requirement for the implementation of the protocol is to be portable for all POSIX[1] compliant operating systems with access to essential dependencies. Second quality requirement is to abstract and hide routing from the end user of network. Another non-functional requirement, which is partially coupled with the previous one, is to provide an API[2] of end-point to end-point packet transmission to the end user.

The thesis was written with the following structure. In Chapter 2, general project concept is discussed. Starting with the description of data types and variables that the protocol is using. Then each part of the protocol is explained. Consisting of network topology maintenance and discovery, as well as packet routing and mechanism that try to ensure packet delivery. The chapter ends with proof of correctness for the described protocol. Chapter 3 contains a sample implementation of the protocol discussed in the previous chapter. It begins with detailed explanation of the

[1] POSIX is abbreviation for Portable Operating System Interface and is an operating system compatibility standard

[2] API stands for Application Programming Interface

chosen hardware that is directly responsible for radio communication. It also shows the control system platform that actually runs the implementation, giving a rundown of multiple choices for such control system. Next, the implementation itself is described. Exact modules responsible for different tasks such as communication with the radio hardware, providing API for the user, routing itself are illustrated. After simulator used for performance analysis is also described. At the end testing methodology applied during development process is also explained. Next, Chapter 4 discusses sample applications of the implemented protocol using provided API. It shows 3 different applications: echo, temperature reading and console together with detailed implementations. Finally, other possible areas of usage are mentioned. Chapter 5 focuses on evaluating performance of the protocol implementation with regards to network size, topology and environment. It starts with assessing complexity of topology discovery process. Afterwards, load and spike testing is described and performed. Then, harsh environment testing is conducted. Finally, analysis of all results is carried out. Eventually, summary of the study is discussed in Chapter 6.

In this work, specific tasks were divided among the group. Implementation of the protocol was assigned to three people: Tomasz Kuczma, Mateusz Rybarski and Maciej Żurad. The design of the protocol was also split into parts. Delivery guarantee was done by Tomasz Kuczma and Mateusz Rybarski, while discovery and maintenance of topology was designed by Amadeusz Juskowiak. Routing of packets was developed by Amadeusz Juskowiak. Last part of the protocol, proof of correctness was completed by Maciej Żurad. Network simulator and its further integration with various tools was done by Amadeusz Juskowiak and Mateusz Rybarski. While integration, performance and reliability tests together with fixtures were conducted by Tomasz Kuczma. Software engineering issues and unit testing were solved by Tomasz Kuczma and Maciej Żurad. Finally, implementation of sample applications was done by Mateusz Rybarski, however the driver for the end user was designed and implemented by Amadeusz Juskowiak.

Chapter 2

Project concept

In following chapter, general concept of communication protocols provided in this thesis is described. Then correctness of various properties of these protocols are proven. In context of protocols and algorithms operating with network, there is a need to choose network representation used in this thesis. Because of nature of designed protocols, where in every pair of adjacent nodes, both of them have to see each other to make communication possible, undirected graph $G = (V, E)$ is chosen. This graph consists of a set of nodes V and a set of edges E.

2.1 Basis of operation

This section describes basis of operation and algorithms used in designed protocols. Following subsections show how topology of the network is discovered and maintained, how packets with data sent by user applications are routed within the network and how the protocols try to ensure delivery of these packets. Firstly, there is a need to define data types, operations and events used in designed protocols.

2.1.1 Data types, variables, operations and events

Following operations, events and data types are used in designed protocols.

Data types

Pseudocode 1 shows data types used in protocols. It is worth to notice, that *data* field of *DATA* packet type is not fully defined here. This type depends on implementation details, so its full definition is omitted here. Different implementations can send via network different types of data.

Variables

At every node, variables from pseudocode 2 are defined. `self` represents node itself. In addition, every node contains graph that represents its knowledge about the state of the network. In the rest of this chapter, it will be named *local network*. Inside pseudocodes, G stands for whole local network, V stands for a set of known nodes, E stands for a set of edges known to given node.

Operations

Operations that take argument of type *PACKET* can also take argument of any type that extends *PACKET*.

- `add_node(node : NODE)` – adds node to graph that represents local network,

Pseudocode 1: Data types used in protocols

```
type NODE : Integer;                                    /* Unique identifier of node */
type EDGE is array[1..2] of Node;                /* Represents edge between two nodes */
type EDGE_PARAMETERS is record of
    good : Integer;                                      /* Parameters of given edge */
    retries : Integer;
    errors : Integer;
end
type PATH is list of NODE ;                                  /* The path of a packet */
/* types of packets sent between nodes                                              */
type PACKET is record of
    ;                                  /* Base type of packets used in algorithms */
end
type NODE_BROADCAST extends PACKET is record of
    node : NODE;
end
type GRAPH extends PACKET is record of
    edges : list of EDGE;
end
type EDGE_DROP extends PACKET is record of
    edge : EDGE;
end
type DATA extends PACKET is record of
    id : Integer;                                    /* unique identifier of packet */
    source : NODE;
    destination : NODE;
    data : ... ;                                        /* data sent between nodes */
    visited : list of NODE;
end
type ACK extends PACKET is record of
    id : Integer;                          /* identifier of corresponding DATA packet */
    status : Integer;
    path : PATH;
    params : list of EDGE_PARAMETERS
end
```

Pseudocode 2: Variables defined at every node

```
self : NODE;
packet_history : map of (DATA, timestamp);
```

- `remove_node(node : NODE)` – removes node from graph that represents local network,
- `add_edge(a : NODE, b : NODE)` – adds edge from a to b to graph that represents local network,
- `add_edge(edge : EDGE)` – adds edge to graph that represents local network,
- `remove_edge(edge : EDGE)` – removes edge from graph that represents local network,
- `calculate_antireliability(G)` – calculates reliability of edges in given graph, described in packet routing section,
- `path(source : NODE, destination : NODE) : PATH` – obtains and returns shortest path from source to destination in graph G,

- `calculate_timeout(path : PATH) : Timestamp` – calculates and returns timeout for acknowledgement packet for given path,
- `history_remove(id : Integer)` – finds and removes from `packet_history` entry for packet with given `id`,
- `history_find(id : Integer) : DATA` – finds in `packet_history` and returns entry for packet with given `id`,
- `deliver(receiver : NODE, packet : DATA)` – delivers packet of type *DATA* from current node to node `Receiver`
- `receive(sender : NODE, packet : PACKET)` – receives packet at current node from node `Sender`
- `send(receiver : NODE, packet : PACKET)` – sends packet from current node to adjacent node `Receiver`,
- `broadcast(packet : PACKET)` – sends packet from current node to every adjacent node.
- `error(message : String)` – reports error to application, used only when there is no route to destination,
- `process(data)` – process data by application.

Events

Events that take argument of type *PACKET* can also take argument of any type that extends *PACKET*.

- `e_timeout_expires(packet : DATA)` – occurs when timeout for given *DATA* packet expires,
- `e_antireliability_exceeds(edge : EDGE)` – occurs when antireliability for given edge becomes too high,
- `e_receive(sender : NODE, packet : PACKET)` – occurs when `receive` operation is invoked at current node,
- `e_deliver(receiver : NODE, packet : DATA)` – occurs when `deliver` operation is invoked at current node.

2.1.2 Network topology discovery

To provide self-organizing network, topology has to be discovered at the beginning of operation. Solving this problem is divided into two steps:

- Informing visible nodes that new node wants to join the network,
- Informing rest of the network about new node and edges in the network.

To join the network, new node broadcasts packet of type *NODE_BROADCAST*. This packet contains identifier of broadcasting node that is unique. If any other node receives this packet, `e_receive(sender : NODE, packet : NODE_BROADCAST)` event occurs and procedure from pseudocode 3 is executed on the receiving node. Node received in this packet and edge from this node to itself is added, unless existing in the graph that represents local knowledge of the network. If

Pseudocode 3: Receiving NODE_BROADCAST packet event handler

```
when e_receive(sender : NODE, packet : NODE_BROADCAST):
    if node ∉ V then
        add_node(packet.node);
    end
    if {self, packet.node} ∉ E then
        add_edge(self, packet.node);
        p_graph : GRAPH;
        p_graph.nodes := get_edges();
        broadcast(p_graph);
    end
end
```

the edge was added, every edge in the graph is broadcasted using packet of type *GRAPH*. *GRAPH* packet contains information about every edge that is known to broadcasting node. When packet of type *GRAPH* is received, `e_receive(sender : NODE, packet : GRAPH)` occurs and procedure from pseudocode 4 is executed. Edges from the packet are merged with local network graph. If it has changed, new packet of type *GRAPH* is broadcasted with current graph.

Pseudocode 4: Receiving GRAPH packet event handler

```
when e_receive(sender : NODE, packet : GRAPH):
    change : Bool;
    change := False;
    foreach edge : packet.edges do
        if edge ∉ E then
            add_edge(edge);
            change := True;
        end
    end
    if change = True then
        p_graph : GRAPH;
        p_graph.nodes := get_edges();
        broadcast(p_graph);
    end
end
```

2.1.3 Network topology maintenance

After discovering topology, it is needed to ensure that nodes are still able to deliver packets via edges in graph and nodes are still working. This is done in two steps:

- Removing faulty edges from graph,
- Rediscovering nodes and edges in case of temporary breakdown.

Firstly, when antireliability measure for given edge exceeds threshold value (antireliability measure is described widely in Packet routing section), `e_antireliability_exceeds(edge : EDGE)` event occurs and code from procedure shown in pseudocode 5 is executed. Packet of type *EDGE_DROP* is broadcasted to inform other nodes, that given edge should be removed from graph because there is a possibility undelivering packet using this edge.

Pseudocode 5: Handling e_antireliability_exceeds event

```
when e_antireliability_exceeds(edge : EDGE):
    p_edge_drop : EDGE_DROP;
    p_edge_drop.edge := edge;
    broadcast(p_edge_drop);
end
```

When packet of type *EDGE_DROP* is received, `e_receive(sender : NODE, packet : EDGE_DROP)` event occurs (pseudocode 6). The received edge is removed from the graph. If it was removed successfully, the *EDGE_DROP* packet is broadcasted again from current node. Otherwise, edge was deleted earlier or have never existed for this node and *EDGE_DROP* packet is not broadcasted. This operation allows to remove faulty edges in whole network. If node without any edge exists in the network, it should be removed locally from graph.

Pseudocode 6: Receiving EDGE_DROP packet event handler

```
when e_receive(sender : NODE, packet : EDGE_DROP):
    if edge ∈ E then
        remove_edge(packet.edge);
        p_edge_drop : EDGE_DROP;
        p_edge_drop.edge := packet.edge;
        broadcast(p_edge_drop);
        if ∄{a, b} ∈ E : packet.edge[1] ≡ a ∨ packet.edge[1] ≡ b then
            remove_node(packet.edge[1]);
        end
        if ∄{a, b} ∈ E : packet.edge[2] ≡ a ∨ packet.edge[2] ≡ b then
            remove_node(packet.edge[2]);
        end
    end
end
```

Secondly, once in a while, from every node packet of type *NODE_BROADCAST* is broadcasted and procedure similar to topology discovering is happening in the network. This allows rediscovering nodes and edges if they were removed from graph in case of broadcasting *EDGE_DROP* packet.

2.1.4 Packet routing

As mentioned at the beginning of this chapter, network is represented by undirected weighted graph. Packet routing determines shortest path in this graph, from packet source node to packet destination node. Important thing is that following rules are applied only for packets of type *DATA*. These packets contains addresses of source and destination nodes, data that needs to be delivered and list of visited nodes by packet.

Following subsections describe how antireliability measure used as weights on edges is calculated, how packet is delivered to destination node and how parameters used to calculate antireliability measure on edges are updated with acknowledgement packets.

Antireliability measure

Network is represented by undirected weighted graph, where antireliability measure acts as edge weight. This measure is a real number greater than or equal to 0. Greater the measure is, reliability

of given edge is lower.

Antireliability measure evaluation is based on parameters stored for every edge in the graph:

- *GOOD* which determines number of delivered packets using given edge,
- *ERRORS* which determines number of undelivered packets using given edge,
- *RETRIES* which determines number of retries of packets on given edge.

It is worth to notice, that every increase of *ERRORS* parameter makes *RETRIES* parameter increased at least by 1. Equation 2.1 shows how antireliability measure Am for given edge is calculated:

$$Am = \frac{RETRIES * (ERRORS + 1)}{GOOD + 1} \tag{2.1}$$

Packet delivery

In order to send *DATA* packet from source node to destination node, an antireliability measure is calculated for every edge in the graph as described above. Then, the path from source node to destination node is obtained using Dijkstra's shortest path algorithm [Dij59] with antireliability measures as edge weights. Subsequently, *DATA* packet is sent to first node on obtained path. There is possibility, that path obtained will be empty. That means destination is inaccessible from current node. If such situation occurs, proper error should be raised to application layer. This process is shown in pseudocode 7.

Pseudocode 7: e_deliver event handler

```
when e_deliver(receiver : NODE, packet : DATA):
    calculate_antireliability(G);
    path : PATH;
    path := path(self, receiver);
    if path ≠ ∅ then
        send(path.first, packet);
        t : Timestamp;
        t := calculate_timeout(path);
        if packet.source = self then
            history.add(packet, t);
        end
    else
        if packet.visited ≠ ∅ then
            p_ack : ACK;
            p_ack.id := packet.id;
            p_ack.status := self;
            p_ack.path := packet.visited;
            send(p_ack.path.last, p_ack);
        else
            error("No route to destination");
        end
    end
end
```

When packet of type *DATA* is received on intermediate node, list of visited nodes stored in packet is updated with current node, antireliability measures are evaluated at current node and path from current node to destination node is obtained, trying to avoid already visited nodes. Afterwards, packet with data is sent to first node at newly obtained path. If packet of type *DATA*

reaches its destination, acknowledgement packet is sent back to the source node and received data can be processed by node. This process is presented in pseudocode 8.

Pseudocode 8: Receive DATA packet event handler

```
when e_receive(sender : NODE, packet : DATA):
    if self = packet.destination then
        p_ack : ACK;
        p_ack.id := packet.id;
        p_ack.status := 0;
        p_ack.path := packet.visited;
        send(p_ack.path.last, p_ack);
        process(packet.data);
    else
        packet.visited.add(self);
        deliver(self, packet.destination, packet);
    end
end
```

Acknowledgement packets

Acknowledgement packets (packets of type *ACK*) are sent back from destination node to source node, by exactly the same path the corresponding packet was delivered to the destination node. The *ACK* packet contains parameters of every edge on the path of corresponding *DATA* packet, which are used to update graph on intermediate and source nodes. Process of updating these parameters strongly depends on implementation details and used hardware, so it is not widely described in this chapter. If *DATA* packet could not be delivered, *ACK* packet contains also information about last node which could not deliver the packet.

2.1.5 Ensuring delivery

One of main goals of this project is to try to ensure that *DATA* packet sent from source node will be delivered to destination node or message with error will be send to application when packet could not be delivered. As it is mentioned in assumptions section of first chapter, it is assumed that packet will be delivered to its destination node, but there are cases when packet can be delivered more than once. To aim this goal, two mechanisms are used: acknowledgement packets and timeouts. For purpose of delivery guarantee, there is a need to remember at every node, every packet that was sent or was transmitted . This is remembered in `packet_history`.

Acknowledgement packets

Acknowledgement packets are very important to delivery process. After *DATA* packet reaches its destination, *ACK* packet is sent to *DATA* packet source node by exactly the same way corresponding *DATA* packet was transmitted. When *ACK* packet reaches corresponding *DATA* packet source node, this node knows that packet was delivered successfully. As it is mentioned in previous subsection, acknowledgement packets are also used when *DATA* packet can not be delivered. In this case, *ACK* packets contains information about node that could not deliver *DATA* packet to next node. When source node receives *ACK* packet with such information, it knows that delivery of *DATA* packet failed and this packet should be retransmitted (probably with different path, if it is possible). Whole procedure is shown on pseudocode 9.

Pseudocode 9: Receive ACK packet event handler

```
when e_receive(sender : NODE, packet : ACK):
    update_edges_parameters();
    packet.path.popback();
    /* intermediate node                                          */
    if packet.path ≠ ∅ then
        send(packet.path.last, packet);
    else
        /* packet undelivered                                     */
        if packet.status ≠ 0 then
            data_packet : DATA;
            data_packet := history_find(packet.id);
            history_remove(packet.id);
            deliver(data_packet.destination, data_packet);
        else
            history_remove(packet.id);
        end
    end
end
```

It should be noticed that when source node of corresponding *DATA* packet receives *ACK* packet, it should remove entry about *DATA* packet from `packet_history`. There are cases when *ACK* packets can not reach source node of corresponding *DATA* packet. These situations are resolved using timeouts mechanism described below.

Timeouts

As mentioned, timeouts mechanism repeats delivery process of *DATA* packet when corresponding acknowledgement packet does not reach source node. For every outgoing packet, receipt time of corresponding acknowledge packet is calculated. Equation 2.2 shows how this time is calculated:

$$T = 2c \cdot \left(\sum_{e \epsilon E_p} \left(d_e \cdot \left(1 + \frac{RETRIES_e}{max\{GOOD_e - RETRIES_e, 1\}} \right) \right) + \sum_{v \epsilon V_p} t_v \right) \tag{2.2}$$

Where:

$RETRIES$, $GOOD$ are parameters of given edge, described in antireliability measure subsection,

E_p is a set of edges in packet path,

d_e is a delay on given edge,

V_p is a set of nodes in packet path,

t_v is a time of processing single packet on given node

c is a constant greater or equal 1.

Timeout is calculated at source node of *DATA* packet and stored in `packet_history`. Action, which takes place after timeout expires is shown on pseudocode 10. After timeout expires, *DATA* packet is retransmitted (possibly with a different path) and new timeout is calculated.

Pseudocode 10: e_timeout_expires event handler

```
when e_timeout_expires(packet : DATA):
    history_remove(packet.id);
    deliver(packet.destination, packet);
end
```

2.2 Proof of correctness

The following section's plan is to show what correctness of a protocol is and what is required to prove such thing.

2.2.1 Assumptions

This section focuses on presenting assumptions about network and the environment that surrounds it. Furthermore, based on these assumptions and special cases of the protocol that can appear, it will be clear which properties the protocol has.

Critical assumptions for understanding the protocol are:

- *fail-stop* node [SS83] – means that the process (in this case a node) when fails, will go into halt mode instead of executing erroneous state. This also indicates that other processes will be able to detect that and deal with it by not trying to communicate. Fail-stop also implies that a process will lose all state after failure, therefore it is equivalent to introducing a new node.
- *non-permanent network partitions* – is a very important assumption for the correctness of the protocol. It promises that if a network partition occurs, it will not be perpetual. A network partition happens when a failure of a single node leads to a situation, where there exist two correct nodes unable to communicate with each other.
- *changing topology* – one way of permanently changing topology is to introduce new nodes. However, this does not provide any bad consequences unlike permanent failures, which would highly complicate the proof of correctness.
- *perfect links* [CGR11] – every two adjacent nodes are in a perfect link. Meaning that if the nodes are correct (neither sender nor receiver is down), then if one process sends a message to its neighbour, neighbour will obtain it eventually. This property holds because of the nature of XBeePro-868 hardware, which handles retransmits by itself, and if the receiver node is down then it returns an error status. It also implies no duplication of messages between those two nodes and a requirement of an originator of message. No message can be received without it being sent first.

2.2.2 Special cases

The protocol uses acknowledgments and timeouts as a way of telling the originator of message if it has been successfully or unsuccessfully delivered. Because of this, there is a situation, which makes it impossible to prove that the protocol has a *Reliable Delivery* property [CGR11]. This property says that if a correct process requests to send a message m to a correct process q, then p eventually delivers m to q.

The situation that causes this property to fail is depicted in the figure 2.1 and is the following: Process p sends a message m and correctly delivers it to process q, after receiving the message

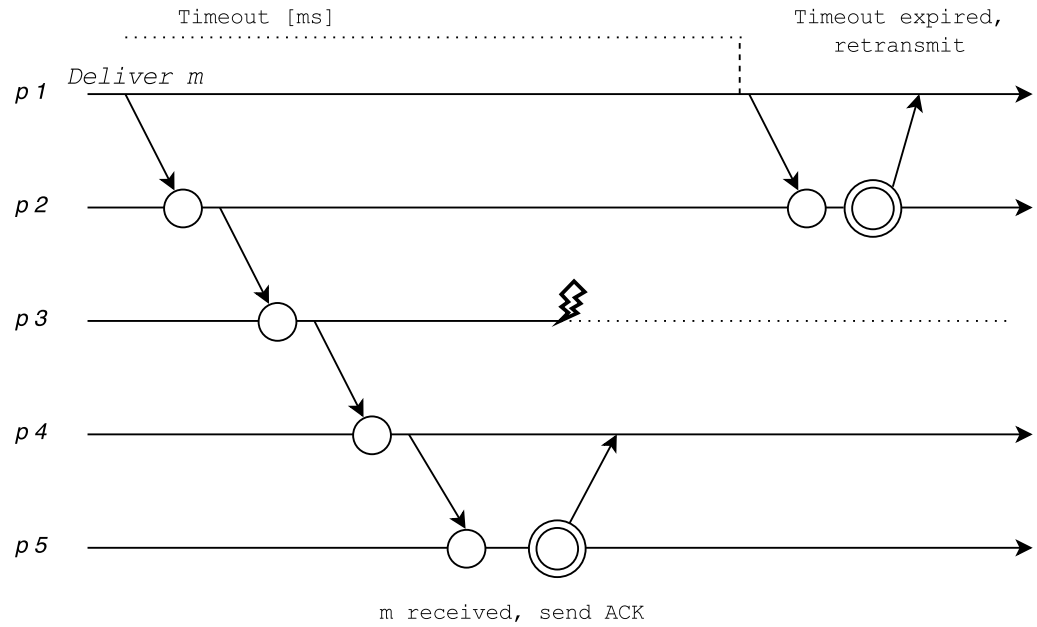


Figure 2.1: ACK returns undelivered, after delivering message

process q sends back the acknowledgment message ACK to process p. While the ACK is being routed to originator, the network is temporary partitioned, leaving the ACK message without a path to originator. This after some time t causes timeout to expire and originator attempts to retransmit the message again. However, the network stays partitioned and eventually originator reports that the message has been unsuccessfully delivered. This shows that the reliable delivery property does not hold, since reporting undelivered at the originator does not guarantee anything. Receiving acknowledgment without error status means of course that the message has been successfully delivered, but that is not enough for proving *Reliable Delivery*. Following logical statements 2.3 show this property. It can be seen that receiving status OK implies that message was delivered. However, delivering the message does not imply that status OK will be received. Same case applies to receiving status NOT_OK, which doesn't imply that message was not delivered.

$$\left.\begin{aligned} &OK \rightarrow Delivered, \\ &\neg(Delivered \rightarrow OK), \\ &\neg Delivered \rightarrow NOT\_OK, \\ &\neg(NOT\_OK \rightarrow \neg Delivered), \end{aligned}\right\} \quad \text{Delivery to ACK relation} \tag{2.3}$$

2.2.3 Proving correctness of the protocol

To consider any distributed protocol correct requires two classes: *safety* and *progress* to be satisfied [Lam77, ADS86]. Safety properties enforce that something cannot happen. They are used to prevent anything bad from happening. However, only specifying safety properties is not enough, a trivial program that does not do anything would count as correct. Therefore the second class of progress properties is needed. Progress properties (also known in literature as *liveness* properties [CGR11]) are there to ensure that something good occurs. Moreover, these properties state that for any time t, they will be fulfilled at time $t' > t$.

The properties of protocol are defined in 2.1. Immediately it can be observed that *Stubborn delivery* is a *progress* property and *No creation* is a *safety* property. Stubborn delivery is weaker

Table 2.1: Interface and properties of Stubborn point to point links

Module:
Name: StubbornPointToPointLinks, **instance** *sl*.

Events:
Request: ⟨*sl, Send* | *q, m*⟩ : Requests to send message *m* to process *q*.
Indication: ⟨*sl, Deliver* | *p, m*⟩ : Delivers message *m* sent by process *p*.

Properties:
SL1: *Stubborn delivery:* If a correct process *p* sends a message *m* once to a correct process *q*, then *q* delivers *m* an infinite number of times.
SL2: *No creation:* If some process *q* delivers a message *m* with a sender *p*, then *m* was previously sent to *q* by process *p*.

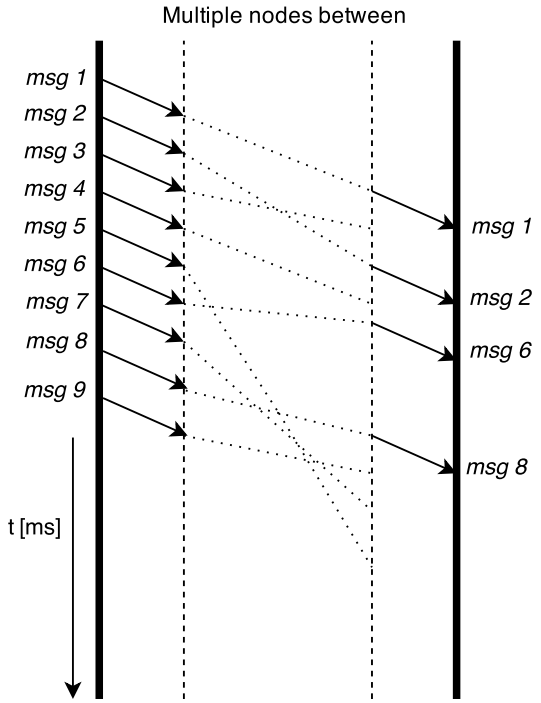


Figure 2.2: Stubborn Links time graph

of course than most progress properties, most notably reliable delivery. It assures that if and only if the message *m* is transmitted infinite number of times from a process *p*, then the process *q* will receive it an infinite number of times. Figure 2.2 shows a time line of the same frame going through the network, and either delivering or not delivering.

Theorem 1. *If two presented properties* Stubborn delivery *and* No creation *are met, then the whole protocol implements Stubborn Point to Point links interface.*

Correctness Arguments. It is trivial to prove that the *No creation* property holds. Previous assumption that the network is operating on perfect links, makes it very easy to say that this property cannot be violated, since perfect links are much stronger property than e.g. fair-loss links [CGR11] which already cover *No creation* property. The progress property on the other hand, requires to look at the protocol and assumptions. Because Stubborn delivery sends a message

infinite number of times, then the receiver will get also infinite number of messages. The actual difference in number might be huge between sent and received messages, but looking at infinity it holds. Assumption about temporary network-partitions is very important, because retransmitting messages infinite number of times, doesn't assure the receiver that he is also going to get infinite number of messages. In fact if the partitions could be permanent, the property would not hold. Obviously the protocol is not efficient if the message was received and the other side doesn't need it anymore. Therefore, the implementation of the protocol sends acknowledgment packets, which stop sending the same message. This is merely an implementation detail, so it can be done. □

Chapter 3

Project

Algorithm implementation is required to study its behaviour in real life situations and perform various tests. Two distinct implementation sections can be designated – hardware-related design and software implementation. Software implementation requires connection to physical radio device or simulator environment.

3.1 Infrastructure

Proposed algorithm does not require any specific hardware platform, however its design process was inspired by XBeePro-868 devices produced by Digi International Inc. Although XBeePro-868 device provides support for point-to-point and point-to-multipoint, implementation of mesh networking is omitted in the current radio firmware.

XBee-Pro 868 provides physical, mac and security layers upon which our network layer is built. The radio module connects with controller (such as personal computer, embedded system or microcontroller) using industry standard serial communication. Therefore communication in heterogeneous computer environment is possible. It should be noticed that line-of-sight distance between two adjacent nodes can be as long as 40 kilometers with proper antennas.

3.1.1 XBeePro-868 radio module

Digi XBeePro-868 radio module is complete radio solution for point-to-point and point-to-multipoint communication. Essential components such as UART transceiver, radio modem and amplifier were embedded into small footprint (figure 3.1) module. Complete module documentation is available in manufacturer provided manual [Dig14b].

Radio amplifier transmits signal with 315mW power, which combined with built-in high signal-to-noise receiver and proper antenna allows to create stable link over 40km line-of-sight distance.

Specification

XBeePro-868 module requires 3.3V power source capable of delivering current up to 500mA. Built-in UART allows variety of data rates (1200 - 230400bps) to be used with communication to controller. Overview of the parameters is available in table 3.1, however complete specification is available in the datasheet [Dig11a].

Modules implement three networking layers – physical baseband layer, mac layer and security layer. Modules are addressed with 64-bit MAC address set by the manufacturer. Beside that nodes need to be within same network (determined by user configurable network identifier). Transmissions are encrypted using symmetrical AES-128 algorithm [DR91] (its support is built-in

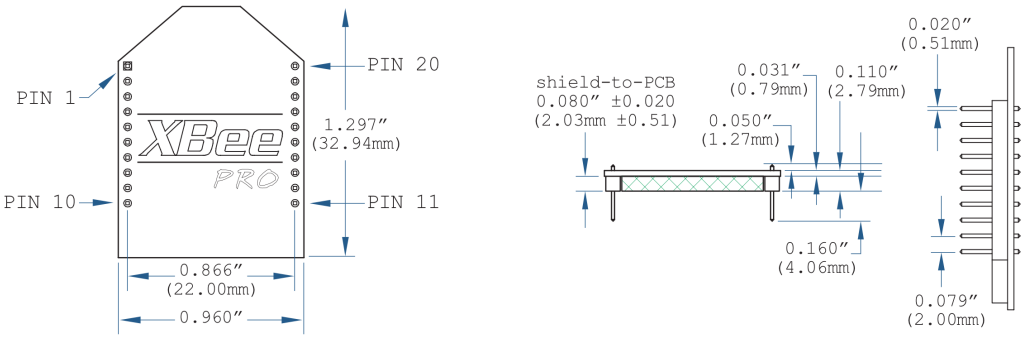


Figure 3.1: XBeePro-868 mechanical drawing [Dig11a]

Table 3.1: XBeePro-868 specification

Parameter	Value
Data rate	24Kbps
Indoor range	Up to 500m
Outdoor range	Up to 40km
Transmit power	1mW (0dBm) to 315mW (+25dBm)
Receiver sensitivity	-112dBm
Frequency band	868 MHz (SRD g3 Band 869.525MHz)
Addressing	64-bit MAC address, 16-bit Network ID
Encryption	128-bit AES

into hardware and transparent to user). Therefore knowledge of network identifier, mac addresses and encryption key is required to transfer and receive messages with these devices, making it quite secure solution.

Interfacing with other electronics is conducted using 3.3V CMOS level pins. The pinout is described in table 3.2.

Table 3.2: XBeePro-868 pinout

Pin number	Type	Function
1	Power	3.3V power supply
2	Output	UART data output
3	Input	UART data input
4	Input/Output	GPIO
5	Input	Reset trigger
6	Input/Output	GPIO, PWM or RX signal strength indicator
7	Input/Output	GPIO or PWM
8	Not connected	None
9	Input/Output	GPIO or sleep trigger
10	Power	Ground
11	Input/Output	GPIO or analog input
12	Input/Output	GPIO or serial CTS signal
13	Output	GPIO or sleep indicator
14	Power	Voltage reference for analog inputs
15	Input/Output	GPIO or connection indicator
16	Input/Output	GPIO or serial RTS signal
17	Input/Output	GPIO or analog input
18	Input/Output	GPIO or analog input
19	Input/Output	GPIO or analog input
20	Input/Output	GPIO or analog input

Table 3.3: Radio power levels

Level	Transmission power	Current
0	1mW	85mA
1	23mW	150mA
2	100mW	280mA
3	158mW	350mA
4	350mW	500mA

(a) RP-SMA connector (b) U.FL connector (c) Wire antenna

Figure 3.2: Available XBeePro-868 variants[Dig14a]

Radio parameters

Radio transmitter is capable of transmitting signal up to 315mW (+25dBm). Its power can be limited with 5 discrete levels presented in table 3.3. Receiver is rated with -112dBm sensitivity. Radio works in 868MHz frequency band (in specific SRD g3 Band 869.525MHz) which is available in Europe and its usage is regulated by ETSI standard [Eur11] and EU directive [Eur09].

The module is capable to transfer data with 24Kbps data rate, however its duty cycle is limited to 10%[1]. Duty cycle causes effective throughput to be limited to 2.4Kbps. Duty cycle implementation by Digi allows to send data for 6 minutes in total in each hour. For remaining 54 minutes device is in passive mode – it can only receive frames as transmitting is disabled.

Modules are available with various antenna connectors – RP-SMA connector (figure 3.2a), U.FL connector (figure 3.2b) or built-in wire antenna (figure 3.2c). In total 5 devices were used during development, four devices with RP-SMA connectors and one device with built-in wire antenna.

RP-SMA connector is more reliable than U.FL, as U.FL was designed to connect small embedded antennas, while RP-SMA is industry acceptable external low-power antenna connector.

Tests were conducted using small +2dB omnidirectional antennas (figure 3.3). For large distance communication high gain antennas are required. In this case antenna specification should be selected depending on environment and required distance. Such selection and appropriate calculations and simulations are not part of the protocols concept.

Communication methods

XBeePro-868 requires at least four pins (power supply, ground, data input, data output) connected to make UART connection available. Parameters of the connection such as data rate, parity, stop bytes and flow control must be the same on the module and its controller.

Two modes of communication are available – transparent mode and API mode.

Transparent mode

[1] Duty cycle is limited by ETSI standard. [Eur11]

Figure 3.3: Omnidirectional +2dB 868MHz antenna with RP-SMA connector[Fac15]

In transparent mode any bytes sent to data input pin are encapsulated automatically into frame and sent to other XBee device. When any data are received they are decapsulated and sent to the controller using data output pin. Therefore such two devices emulate direct serial port connection.

Devices must be configured in order to use this mode – two devices must be paired together.

API mode

API mode allows for more advanced transmissions. Controller is required to provide correct frames into data input pin. When data are received, frame is not decapsulated – raw frame is sent out using data output pin. Other frames can be generated internally and sent by XBee device as response to various events.

In this mode programmer can explicitly specify destination of a frame. In similar fashion received frames contain explicitly source mac address. This behavior is practical and it is used in software implementation of the protocols concept.

Configuration

XBeePro-868 modules are configured using AT-inspired command set.

Commands can be entered in various ways. AT Command Mode can be entered using following algorithm – wait one second, enter `+++`, wait one second. In this mode settings can be managed using AT syntax [Dig14b, p. 16]. When module is in API mode, AT command can be encapsulated into command frame [Dig14b, p. 38, 43]. It is possible to execute AT command on remote XBeePro-868 module from another reachable device using remote command frame [Dig14b, p. 42-43]. Entering AT Command Mode in API mode is still possible using the same algorithm.

Values are stored in built-in reprogrammable EEPROM memory. To replace stored values command `ATWR` is required.

Assortment of parameters are configurable – comprehensive list of settings and associated AT commands is available in the manual [Dig14b, p. 27-34]. Some of parameters are read only, an essential one is mac address which can be read using `ATSH` and `ATSL` commands.

In order to make use devices as radio transceiver for implementation of thesis algorithm some settings must have been changed – list of settings changed for each device is available in table 3.4.

Table 3.4: List of changed settings

AT command	Default value	New value	Description
AP	0x00	0x01	Enables API mode
BD	0x03	0x07	115200bps baud rate
NI	' '	various	Node name
EE	0x00	0x01	Enables AES encryption
KY	unknown	'HelloWorld'	Encryption key
PL	0x04	depends on test	Power level (see table 3.3)

3.1.2 Control system

XBee868-Pro can not be completely functional without control device connected to the module using UART connection. Control system is platform for executing software implementation of the routing algorithm. Radio module can be connected directly to the control system using UART or using some sort of adapter if UART is not available on target. This allows variety of systems to be used as controllers such as personal computers, embedded systems, industrial computers or even microcontrollers.

Implementation was developed and designed to work on two platforms – embedded system and personal computer. It is assumed both platforms are using POSIX compatible operating system.

Embedded system

Embedded system is type of computer system designed to be part of larger electronic, robotic or mechanical system. Embedded systems implements various industrial protocols – UART, required for communication with XBeePro-868 module, is usually built-in.

RaspberryPi

The embedded system of choice for the thesis is RaspberryPi [Ras15]. It was not designed as typical industrial embedded system, but small, extremely affordable computing platform. One piece of RaspberryPi, platform capable of performing around 900MIPS [Lon14], costs $25-$35 depending on configuration. Most recent specifications of the platform can be obtained at RaspberryPi Foundation website at `http://raspberrypi.org/products/`.

Even if RaspberryPi was designed for consumer and educational market, it still follows principle of embedded system. Various interfaces are available (i.e. UART, GPIO, SPI or Ethernet), however some of them are not available in every model.

Four different models were available at time of the writing. Model B+ (visible on figure 3.4) was chosen as it was widely available at this time, yet any model of RaspberryPi is suitable as control system. Processor architecture of RaspberryPi is ARMv6, so ARMv6 compatible Linux operating system is installed.

Connection

Serial port can be easily accessed on RaspberryPi. Voltage levels both on XBeePro-868 and RaspberryPi are the same 3.3V CMOS level. Therefore it is possible to connect the devices directly without level translator – example pin connection is provided in table 3.5. For development purposes connection using jumper cables can be made, however for industrial or any commercial usage proper carrier board should be designed.

Power supply on the board generates 3.3V, although only about 100mA of current is left for external use. It is possible to power the radio from onboard power supply, however the transmission

Figure 3.4: RaspberryPi Model B+ [Ras15]

Table 3.5: Connection between RaspberryPi Model B+ and XBeePro-868

RaspberryPi		XBeePro-868	
Function	Pin	Pin	Function
3.3V power supply	1	1	VCC
TX signal	8	3	Data output
RX signal	10	2	Data input
Ground	39	10	Ground

power should be limited to 1mW. Usage of greater transmission power requires external power supply.

Personal computer

Personal computer is general purpose computer available for individuals. Any modern x86 platform with POSIX operating system (such as a distribution of Linux or commercial OS X system) should be suitable as control system for XBee module. However modern computers do not incorporate RS-232 serial port nor UART CMOS 3.3V level serial port.

Though personal computer is ideal platform for routing application execution, usage of adapters and protocol translators is usually required. USB is standard interface available in almost all personal computers. There are commercially available USB to UART converters and they can be used to connect XBeePro-868 module to the computer. In addition carrier boards specifically made in mind of connection XBee series devices with computers using USB are commercially available.

Digi XBee Development Board

XBee devices manufacturer Digi provides complex developer board (manufacturer part number *XBIB-U-DEV*) capable of communication with computer using USB (pictured in figure 3.5). Board is supplied with external power source so usage of full transmission power is possible.

The development board can be used to check every feature of XBeePro-868 devices, however consulting its guide [Dig11b] is recommended as the thesis does not exploit every feature.

Two pieces of Digi XBee Development Board were used during testing and implementation process of the thesis.

Figure 3.5: Digi XBee Development Board[Dig11b]

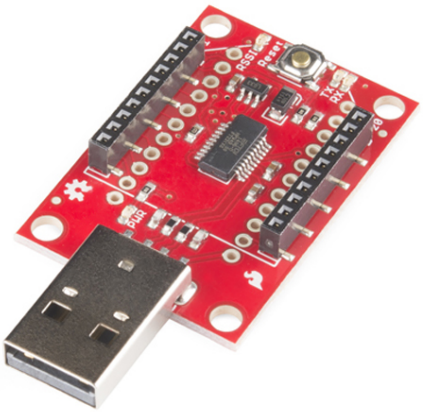


Figure 3.6: Sparkfun XBee Explorer Dongle[Spa13]

Sparkfun XBee Explorer Dongle

Independent manufacturer Sparkfun produces XBee Explorer Dongle (pictured in figure 3.6)[Spa13]. It is much simpler device than previously described development board – it connects directly into USB port and drains power from it. Transmission at full power is not recommended, however it possible, depending on USB power capabilities of the personal computer. Main advantage of this device is price, it is 5 times cheaper than competitor's device.

Two pieces of Sparkfun XBee Explorer Dongle were used during testing and implementation process of the thesis.

3.2 Software implementation

The protocols proposed in the thesis are implemented in C++11 language for POSIX platforms (implementation was tested on Linux x86_64/arm11 and OS X x86_64 machines). C++11 language

is chosen for its efficiency, low level system integration and well provided object oriented paradigm. Compared to previous editions of the language, new features such as lambda expressions, native threads or unordered sets simplifies overall implementation. The code is portable and works in heterogeneous environment. Build process and dependencies retrieval is automated with Cmake, subversion and git systems. The most recent version of toolchain is recommended, yet the software was developed using LLVM 3.5 and G++ 4.8 compilers.

Router classes and structures are contained in `PUT::CS::XbeeRouting` namespace. *Router* code is divided into functional sections such as:

- *Driver* – provides communication between network and client applications,
- *Device connector* – parses or serializes *Frames* and communicates with XBee device over serial port,
- *Router* – routes *Packets* and maintains *Network* topology,
- *Dispatcher* – ensures delivery of *Packets* and keeps them in *History*.

Language of choice for the Simulator was Ruby 2.2 – its great standard library and flexibility allowed to implement portable simulator code in readable manner.

Simulator code is divided into DSL[2] providers, low-level part, data structures and logic.

3.2.1 Building process

Code developed in the thesis is available online in git repository at `https://github.com/alfanick/xbee868-routing` or `https://git.amanointeractive.com/poznan-university-of-technology/xbee868-routing-code/tree/master`. It can be provided in other form upon request by any of the authors.

Dependencies

Building process is designed to be as simple as possible and requires from user minimal dependencies.

Driver and router

Essential requirements are:

- POSIX compatible operating system such as Linux 2.6+, FreeBSD 10.0+, OS X 10.5+ – low-level system calls are based upon POSIX standards,
- C++11 compiler (clang 3.5+ – `http://llvm.org/releases/download.html#3.5` or g++ 4.8+ – `https://gcc.gnu.org/mirrors.html`) – compiler used to generate binaries for given system,
- Cmake 2.8.8+ – `http://cmake.org` – portable out-of-source building system,
- Redis 2.6+ server – `http://redis.io/download` – publish-subscribe pattern provider,
- Git client – `http://git-scm.com` – distributed version control system used for thesis code and its dependencies,

[2]DSL – Domain Specific Language

- Subversion client – `https://subversion.apache.org` – version control system used for dependencies retrieval.

Following dependencies are obtained automatically by Cmake:

- Hiredis – `https://github.com/redis/hiredis` – Redis client library for C/C++,
- google-glog – `https://code.google.com/p/google-glog/` – Logging library for C/C++,
- googletest – `https://code.google.com/p/googletest/` – C++ unit testing toolchain.
- googlemock – `https://code.google.com/p/googlemock/` – mocking library used in unit testing.

Simulator

Simulator requires Ruby 2.2+ implementation (standard MRI Ruby `https://www.ruby-lang.org/en/downloads/` is recommended) with croupier-rb 2.0+ (`https://github.com/croupiers/croupier-rb`) and redis-rb (`https://github.com/redis/redis-rb`) gems[3].

Building

After obtaining source code and fulfilling essential requirements run `./configure` while being connected to the Internet. The script will generate build files for given platform. Additional dependencies will be downloaded and built.

To build the driver, router and tests use `./build.sh` – binaries will be generated inside `bin/` directory.

3.2.2 Data structures

In order to truly represent processed data and simplify algorithms and their implementations assortment of data structures must be introduced.

As the thesis covers low-level processing and data, using types which clearly specify their size is preferred (such as `uint8_t`, `uint16_t` or `uint64_t`. To make code follow real world data types some aliases were introduced as visible in table 3.6.

Table 3.6: Introduced typedefs

Type	Original type	Description
`PacketId`	`uint32_t`	Globally unique `Packet` identifier.
`Address`	`uint8_t`	Unique `Node` address.
`Path`	`std::deque<Address>`	`Packet` path.
`Edge`	`Address[2]`	Edge in `Network` graph.
`Destination`	`std::pair<float, Address>`	Cost to given `Node`.

Frame

`Frame` is lowest level data processed in the *Router* software. Frames allow standardized bidirectional communication with Xbee device over serial port. XBee module manufacturer[4] specifies complete API manual [Dig14b]. Frame consists of general API Frame [Dig14b, p. 35] which encapsulated API-Specific Data (figure 3.7).

[3] 'gem' is name for user provided code library in Ruby environment
[4] Digi International, Inc. – `http://www.digi.com`

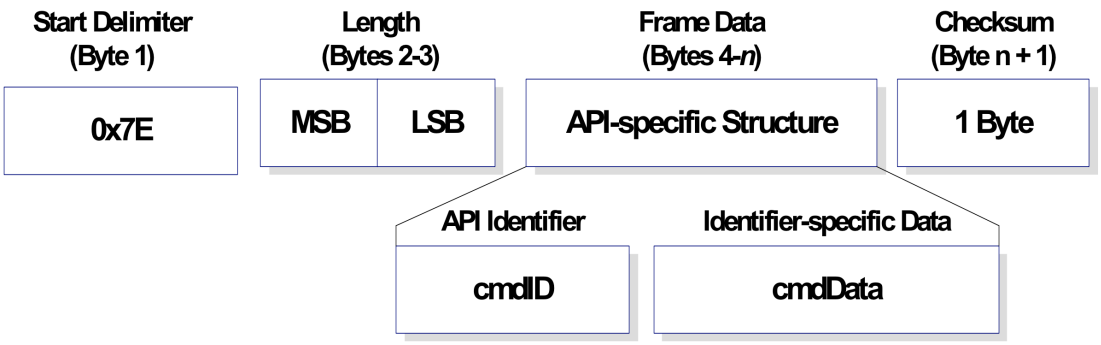


Figure 3.7: API Frame structure[Dig14b]

Frame structure is inspired by frame specification written by the manufacturer [Dig14b]. Therefore `struct Frame` exists mimicking API Frame (as seen in listing 3.1) and encapsulates `union Frame::data` consisting of API-Specific Data depending on given `enum class Frame::Type`. Relationships between type of frames and their functions are presented in table 3.7.

Listing 3.1: Simplified `Frame` structure

```
typedef struct Frame {
  // size of data union
  uint16_t length;

  // identifier of data frame struct (actual type is enum class Type : uint8_t)
  uint8_t type;

  // encapsulated frame struct
  union {...} data;

  // checksum of the frame
  uint8_t checksum;
}
```

Table 3.7: XBee frame types

Structure name	Type	Direction	Description
`ModemStatusFrame`	`0x8A`	Response	Radio module status
`CommandFrame`	`0x08/0x09`	Request	AT command request
`CommandResponseFrame`	`0x88`	Response	AT command response
`RemoteCommandFrame`	`0x17`	Request	AT command request for remote XBee device
`RemoteCommandResponseFrame`	`0x97`	Response	Remote XBee device AT command response
`TransmitFrame`	`0x10`	Request	Simplified data transmit request
`ExplicitTransmitFrame`	`0x11`	Request	Detailed data transmit request
`StatusFrame`	`0x8B`	Response	Status of last transmit request
`ReceiveFrame`	`0x90`	Response	Simplified data receipt
`ExplicitReceiveFrame`	`0x91`	Response	Details data receipt

Implementation of *Router* requires usage only of some of the frames:

- `ModemStatusFrame` – used for discovery device availability,
- `CommandFrame` – used for sending AT commands for configuring and managing XBee device,

- `CommandResponseFrame` – used for AT commands response,
- `TransmitFrame` – used for sending `Packets`,
- `StatusFrame` – used for acknowledge of transmission status and its statistics,
- `ReceiveFrame` – used for receiving `Packets`.

Each `TransmitFrame` and `StatusFrame` require locally unique identifier (`Metadata::frame_id`) so they can be paired together when status for given transmission is received. Identifier is one byte long, so total of 255 frames could be transmitted before their statuses are received (for identifier value `0` no `StatusFrame` will be generated).

Serialization

Filled out `Frame` can be serialized to sequence of bytes using `unsigned char* Frame::serialize(int & length)`. Such string can be transmitted and interpreted in XBee device. Serializing method uses `memcpy` call to serialize `Frame::data` to sequence of bytes, yet it is important to realize differences in byte ordering. Compiler is let know to omit memory paddings with appropriate `#pragma` calls. Hence serialization is much quicker than manual conversion using separate bit operations. During serialization, start byte `0x7E` is added, length is amended accordingly and checksum is calculated. Resulting string is compliant with XBee documentation.

Deserialization

Deserialization process is parallel to serialization. Sequence of bytes can be deserialized into existing `Frame` using `void Frame::unserialize(unsigned char* frame)`. Length of the string is determined using second and third byte, as it is a part of API Frame specification. Deserialization uses `memcpy` function to insert bytes into `union Frame::data`, same cautions about byte ordering are liable. After unserialization it is possible to read specific union field depending on `Frame::type` and process received data.

Packet

`Packet` is the only data type exchanged on network level. `Packet` is encapsulated as data into `ReceiveFrame` or `TransmitFrame` when received or transmitted. `Packet` structure is designed in similar manner as `Frame` structure (as in listing 3.2). Every packet can transmit different data according to its `type`. However the data stored in `union data` are usually primitive types (as seen in table 3.8).

Listing 3.2: Simplified `Packet` structure

```
struct Packet {
  // packet type (implemented as enum class Type : uint8_t)
  uint8_t type;

  // source address (address of last visited node)
  Address source;

  // destination address
  Address destination;

  // origin address (address of first visited node)
  Address origin;
```

```
  // number of quanta of data if applicable (depends on type)
  uint8_t length;

  // port tag associated with packet
  uint8_t port;

  // data depending on packet type
  union {...} data;

  // mac address of source
  uint64_t mac;

  // constant packet id generated on node where its value is 0
  uint8_t packet_id = 0;

  // status used when packet is Ack (0 means delivery, other value is Address of node where error
       was noticed)
  Address status = 0;

  // list of visited nodes during transmission of the packet
  Path visited;
};
```

Table 3.8: Network Packet types

Type	Union field name	Data type	Description
`Internal=0x00`	`frame`	`Frame*`	Used to transport low-level `Frame` to higher layer (usually with `StatusFrame`).
`Data=0x01`	`content`	`uint8_t*`	Sequence of bytes containing message.
`Ack=0x03`	`parameters`	`RemoteParameters*`	Array of link statistics collected while routing `Ack` packet.
`NodeBroadcast=0xFE`	`address`	`Address`	`Address` of broadcasting `Node`.
`EdgeDrop=0xFD`	`edge`	`Edge`	`Edge` to be removed from local `Network`.
`Graph=0xFB`	`edges`	`Edge*`	Array of `Edges` representing new links between nodes.

Packet can be serialized into `Frame` encapsulating `TransmitFrame` using `Frame* Packet::to_frame()`. Serialization is done in different ways depending on `Frame::type` – it is achieved using byte operations and previously mentioned `memcpy` procedure where possible. Deserialization is done in reversed way, comparably to serialization – packet can be deserialized from `Frame` (usually containing `ReceiveFrame`) into existing `Packet` using `Packet::Type Packet:from_frame(Frame* frame)`.

Constant `uint64_t Frame::BROADCAST` is defined and contains local broadcast address for sending packets adjacent to local device.

Packet identifier

Packet identifier is globally unique in the network until corresponding acknowledgement packet is received. It can be acquired using `PacketId Packet::id()` function.

Packet identifier is generated from `destination`, `source` and `packet_id` fields. Origin node sets `packet_id` to generated `Metadata::frame_id` which is locally unique. Therefore tuple `(destination,`

source, packet_id) is unique in whole network and allows to distinguish packet on any intermediate node.

Internal packet

Internal packet is used to encapsulate `Frame` so they can be analyzed in higher layers such as *Dispatcher* or *Router*. Internal packet is generated when in `Router::receive()` run across `Frame` different then `ReceiveFrame`.

Acknowledgement packet

Acknowledgement packet is sent from destination node to origin node via original packet visited path. On each visited node `struct RemoteParameters` (listing 3.3 is appended to the packet. Statistics are used to update antireliability measure on intermediate links.

It is important to notice that `Packet::packet_id` for acknowledgement packet and corresponding data packet are the same, so are the computed `PacketId Packet::id()` values.

Listing 3.3: `RemoteParameters` structure

```
struct RemoteParameters {
  // destination address of the statistics
  Address hop = 0;

  // given transmission edge delay in miliseconds
  uint16_t delay = 0;

  // number of undelivered packets during given transmission
  uint8_t errors = 0;

  // number of packet retries on given edge
  uint8_t retries = 0;
};
```

Each of parameters can max out, but never overflow. Knowledge of origin address and each of the hops addresses allows to update antireliability measure on every intermediate link.

Other case when Acknowledgement packet is sent is when transmission error occurs – `Packet::status` represents address of last node visited. `RemoteParameters` are still collected, yet they do not reach destination node.

History

`History` is responsible for saving data packets in memory until their acknowledgement packets are received. If acknowledge is not received before timeout or its status is negative, retransmission of the packet is possible without any actions from client applications. Even if retransmissions fail, data stored in history can be used to return the message back to the sender application.

`History` keeps relationship between packets `Metadata` and their `Packet::id()` and `Metadata::frame_id` (as can be seen in listing 3.4). `Metadata` store various parameters (listing 3.5 such as send time, local frame identifier, timeout or retransmission counters. These values are used to control packet routing and maintain safe delivery.

Listing 3.4: Simplified `History` class

```
class History {
  private:
    // map containing relationship between packet identifier and metadata
    PacketsHistory packets_history;
```

```
    // map containing relationship between local frame identifier and metadata
    FramesHistory frames_history;

    // bitset guarding frame identifier occupation
    bitset id_occupation;
  public:
    // add packet to history give it identifier
    uint8_t watch(Packet* packet, Path path, time_point timeout);

    // retrieve metadata by packet identifier
    Metadata* meta(Packet* packet);

    // remove metadata from history by packet identifier
    Metadata* erase(Packet* packet);

    // release frame identifier (after status frame is received)
    void release_id(Frame* frame);
};
```

Listing 3.5: `Metadata` structure

```
struct Metadata {
  // data packet whose metadata are stored
  Packet* packet;

  // computed packet identifier
  uint32_t packet_id;

  // history of paths (path can be different with every retransmission)
  std::vector<Path> path_history;

  // current Xbee frame identifier
  uint8_t frame_id;

  // aggregated statistics of the packet
  RemoteParameters frame_status;

  // time of last packet transmission
  time_point send_time;

  // calculated timeout for acknowledgement packet
  time_point timeout;

  // number of retransmission
  atomic_ushort retransmission_counter;
};
```

Dispatcher uses `History::watch()` to compute next free frame identifier and add the packet to the `History`. It also used to store and check acknowledge timeout and retransmission counter. When `Packet` is delivered or timeout expires *Dispatcher* removes `Packets` and its `Metadata` from `History`.

Node

`Node` is representation of single network node (listing 3.6).

Listing 3.6: Simplified `Node` class

```
class Node {
  public:
    // network address
    Address address;

    // mac address (applicable only for adjacent nodes)
    uint64_t mac;

    // node name (applicable only for self node)
    string name;

    // marks self node
    bool self;
};
```

Every vertex inside `Network` graph contains `Node`. Self node is special case and only one self node can exist in the `Network`. Only self `string Node::name` is known to router, as it requires AT command to obtain the name, however `uint64_t Node::mac` addresses of direct neighbours are known and `Address Node::Address` are known for every discovered `Node` in the `Network`.

Network

`Network` represents graph containing connections between nodes. Every vertex incorporate `Node` and for every edge associated is `Parameters` structure (listing 3.7).

Network is represented as undirected graph. Internally graph is represented as `unordered_map< Address, unordered_map<Address, Parameters*>>` as compromise between access complexity and memory usage.

Listing 3.7: Edge `Parameters` structure

```
struct Parameters {
  // total number of delivered packets
  uint32_t good = 0;

  // total number of undelivered packets
  uint32_t errors = 0;

  // total number of packet retries
  uint32_t retries = 0;

  // averaged delay on given edge (in milliseconds)
  uint16_t delay = 10;

  // antireliability measure based on prior parameters
  float antireliability();
};
```

`Network` constructor requires self node address, so first node can be created and accessed using `Node* Network::self()`. Elementary operations such as adding or removing nodes or edges are implemented (listing 3.8). Operation of merging two graphs is implemented, as it is used with `Packet::Type::NodeBroadcast` packet. Overflow-safe updating of edge `Parameters` is implemented using `Network::update()`. Summarized list of operations available in `Network` is presented in listing 3.8.

Listing 3.8: Simplified `Network` class

```
class Network {
  public:
    // constructor - adds self node
    Network(Address self);

    // try to add new nodes and edges to current graph
    bool merge(Edge* edges, uint8_t length);

    // try to add new edge
    bool add_edge(Address a, Address b);

    // try to add new node
    bool add_node(Address a);

    // check for edge existence
    bool adjacent(Address a, Address b);

    // remove edge
    bool drop(Address a, Address b);

    // get edge parameters
    Parameters* edge(Address a, Address b);

    // get node parameters
    Node* node(Address a);

    // get self node
    Node* self();

    // convert graph to list of edges
    Edge* graph(uint8_t &length);

    // get mac address of given node address
    uint64_t mac(Address a);

    // get node address from mac address
    Address from_mac(uint64_t mac);

    // update edge parameters
    void update(Address a, Address b, uint8_t retries, uint8_t error, uint16_t delay);

    // find most reliable path between two nodes without visiting already visited nodes
    Path path(Address from, Address to, Path visited);
};
```

Essential operations

Essential operations such as `bool Network::add_edge(Address a, Address b);`, `bool Network::add_node(Address a);`, `bool Network::merge(Edge* edges, uint8_t length);` and `bool Network::drop(Address a, Address b);` do not generate error when node or edge does not exists (when removing) or does exist (when adding). Instead, boolean value `true` is returned when operation is successful (that is when graph changes after the operation) and `false` is returned when operation does not change the graph.

Furthermore `bool Network::add_edge(Address a, Address b);` and `bool Network::merge(Edge* edges, uint8_t length);` operations will add missing nodes, so edge between could be created.

Edge parameters update

Figure 3.8: Delay calculating method

$$delay = \lceil \frac{delay_{previous} + delay_{new}}{2} \rceil \tag{3.1}$$

Edge `Parameters` are used to calculate antireliability measure and timeouts. Most recent link statistics are preferred. Method `void Network::update(Address a, Address b, uint8_t retries, uint8_t error, uint16_t delay);` is preferred way of updating edge parameters, as it ensures no overflow will happen.

Additionally `uint16_t Parameters::delay` is calculated as moving average (equation 3.1) – previously calculated delay and current delay (with delay assumed to be $10ms$ at the beginning) are averaged together.

Most reliable path

As described in the routing algorithm, path between *source* and *destination* is assumed to be most reliable when sum of antireliability measures on intermediate edges is minimal. Therefore to obtain most reliable `Path`, antireliability measure is treated as edges weight and path with smallest cost is pretended to be the most reliable one. To avoid cycles previously visited nodes should not be visited again. List of previously visited nodes is included when computing path and cost of visiting already visited nodes is increased drastically.

To compute such `Path` well known Dijkstra shortest path algorithm [Dij59] is used. Implementation based on priority queues [Bar98] allows to efficiently solve shortest path algorithm between two nodes in the network. Built in `std::priority_queue` is used with custom comparator based on `Parameters::antireliability()`.

3.2.3 Driver

Driver provides way for any application to communicate with the network based upon routing system implemented in the thesis. It is a layer of abstraction hiding hardware, routing operations and network topology.

Well designed driver ensures following concepts:

- adds minimal performance overhead,
- allows bidirectional communication with any device in the network,
- hides network topology,
- has minimal dependencies,
- works in heterogeneous environment,
- easily integrates with client applications.

Publish-Subscribe pattern

Publish-Subscribe pattern allows publisher to send message to given channel for any number of clients, furthermore any number of clients can listen for any channel, even if there is no publishers in observed channel [CMPC04].

To be specific Topic-Based Publish-Subscribe pattern [EFGK03] is used in *Driver* implementation as inter-process communication tool. In this approach publisher labels every message with topic, while can subscribe to patterns which can match any number of topics.

Following topic naming convention is used internally by *Driver*:

```
network:PORT:DESTINATION:SOURCE
undelivered:PORT:DESTINATION:SOURCE
```

Where `PORT` is number in range $[1, 255]$, `DESTINATION` and `SOURCE` are accordingly destination and source addresses. Addresses can be numbers in range $[1, 255]$ or token `self` which explicitly marks local node.

Any of the parameters can be replaced with `*` (asterisk) – the matching operator. Therefore it is possible to subscribe and receive message with any combination of tuple `(PORT, DESTINATION, SOURCE)`. Interesting and useful patterns with explanations are provided in table 3.9.

Table 3.9: Example patterns

Pattern	Port	Destination	Source	Description
`network:13:2:self`	13	2	self	Transmission from `self` to node `2` on port `13` (used to send message).
`undelivered:13:2:self`	13	2	self	Undelivered message from `self` to node `2` (used when message could not be delivered by any means).
`network:*:*:self`	any	any	self	Used in *Router* to intercept any messages that need to be sent and route them to destination node.
`undelivered:*:*:self`	any	any	self	Used in clients to intercept any undelivered messages from `self`.
`network:13:self:2`	13	self	2	Receipt from `2` on port `13` (used when waiting for messages from given node on given port).
`network:13:self:*`	13	self	any	Used when waiting for messages on given port from any node.
`network:*:self:*`	any	self	any	Used to receive any message for `self` node.

Redis

Redis is open source in-memory key-value database and distributed implementation of Publish-Subscribe pattern. It has low memory footprint and low performance overhead, although it does implement Publish-Subscribe with topic pattern matching.

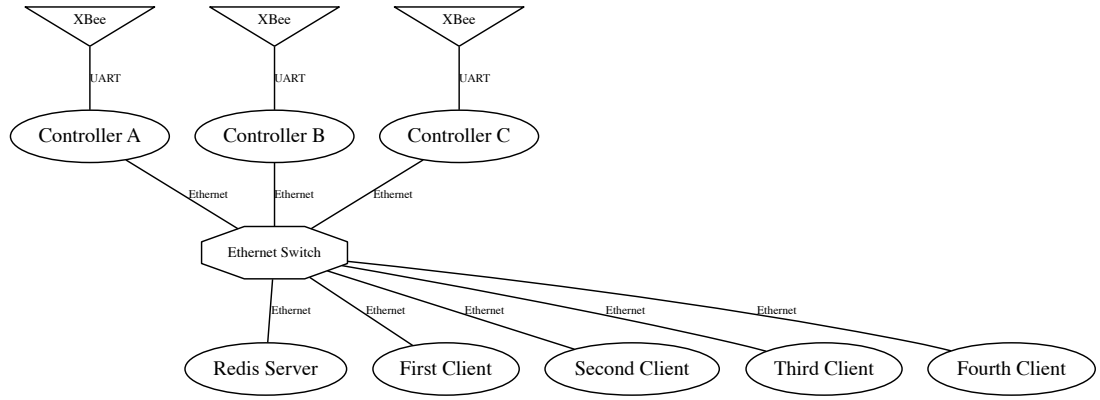
Using C/C++ driver Hiredis and few commands (as shown in table 3.10) satisfies requirements for *Router*. For each machine only one instance of Redis server is required, even if there is more than one *Router* instance (as happens on developer machines or when using *Simulator*). To prevent topic collisions, every *Router* may introduce each own prefix (a number in range $[1, 255]$, generating ie. topic `13/network:*:*:self` instead of `network:*:*:self`).

Usage of Redis comes up with additional property – it is possible to connect *Driver* to remote instance of Redis server. As a consequence it is feasible to connect multiple remote applications to local machine with *Router* and XBee device. Multiple complex topologies can be created with this feature as shown by examples in figure 3.9.

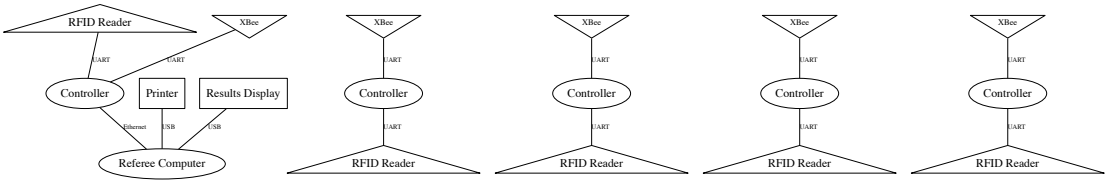
Table 3.10: Redis commands[Red15]

Command	Complexity	Description
`PSUBSCRIBE pattern`	$O(N)$	Subscribes client to given `patterns` (with `*` allowed).
`PUNSUBSCRIBE pattern`	$O(N+M)$	Unsubscribes client from given `patterns`.
`PUBLISH topic message`	$O(K+M)$	Publish `message` to given `channel`.

Where N is number of patterns client already subscribed, M is total number of patterns in system, K is number of clients subscribed to given channel.



(a) Multiple applications connected to multiple routers using Ethernet



(b) Possible sport timing system architecture

Figure 3.9: Example networking topologies

Implementation details

Driver part is implemented with three classes:

- `Driver` – bridge between C++ and network, most important part from developer point of view,
- `Manager` – creates `Subscriptions` with given topic prefix,
- `Subscription` – subscribes to single Redis topic, runs in separate thread.

`Driver` interacts with applications in asynchronous matter. C++11 language features such as `std::function` (extension of function pointer concept) is used to bind code with message receipt events – any type of code such as functions, methods or lambda can be used.

`Driver` defines function types as follows:

Listing 3.9: `Driver` handlers types

```
// Exhaustive handler with destination address, port, source address, data and data length
typedef std::function<void (Address, uint8_t, Address, uint8_t*, size_t)> Action;

// Simplified handler with source address, data and data length
typedef std::function<void (Address, uint8_t*, size_t)> LocalAction;
```

Constant `Driver::SELF` is defined and recognized in `Driver`. It should be used to mark or check local address.

With previously introduced handler types following `Driver` methods connect network with code:

- `void self(Action action);` – executes `action` when any message to `self` is received,
- `void listen(uint8_t port, LocalAction action);` – executes `action` when message on given `port` to `self` is received,
- `void deliver(Address destination, uint8_t port, uint8_t* data, uint8_t length);` – sends given `data` to `destination` on given `port`,
- `void undelivered(Action action);` – executes `action` when message from `self` could not be `delivered`.

These methods abstract network topology and hide routing, yet they follow the concept of reliable network implemented in the thesis. It is possible to *send* messages, *receive* messages and get notified of *undelivered* messages.

Presented code listing 3.10 shows different ways to interact with the network in `echo` style application. It listens on port 13 and sends back to source any received message, in case of delivery failure `handle_error` is executed. Overall three threads are created – one for main function, one for listener and one for undelivered messages listener. Program is running until killed from outside environment.

Listing 3.10: Example `echo` application

```
#require <iostream>
#require <unistd.h>
#require <driver/driver.h>

using namespace std;
using namespace PUT::CS::XbeeRouting;

// undelivered messages handler
void handle_errors(Address destination, uint8_t port, Address source, uint8_t* data, size_t length)
    {
  cerr << "Could␣not␣deliver␣data:␣'" << data << "'␣to␣" << destination << endl;
}

// basic echo program
int main() {
  // driver instance
  Driver driver;

  // listen for incoming messages on port 13 (lambda function)
  driver->listen(13, [](Address source, uint8_t* data, size_t length) {
    // send back data
    driver->deliver(source, 13, data, length);
    cout << "Received␣'" << data << "'␣from␣" << source << endl;
  });

  // handle undelivered messages (function pointer)
  driver->undelivered(handle_errors);

  // let other threads execute
```

```
  pause();
  return 0;
}
```

Driver is build as shared library available in `bin/libxbee_network.so` (or `libxbee_network.dylib` on OS X platforms). By default `Driver` connects to local Redis server instance by Unix socket at path `/tmp/redis.sock`. *Driver* can be further configured using environment variables presented in table 3.11. Any program using *Driver* shared library reacts to previously mentioned environment variables.

Table 3.11: *Driver* environment variables

Variable	Type	Default	Description
REDIS_HOST	network address	none	network address of Redis server (using Unix socket if not provided)
REDIS_PORT	number	6379	Redis server port
REDIS_DATABASE	number	none	prefix of topics in case of using multiple *Router* instances with same Redis server

3.2.4 Dispatcher

`Dispatcher` is responsible for successful delivery of `Packet` or discovering delivery failure. Instance of `Dispatcher` is created by `Router`. Implementation of Dispatcher realize diversity of operations as seen in listing 3.11.

Listing 3.11: Basic `Dispatcher` class

```
class Dispatcher {
  public:
    // constructs instance of dispatcher with access to XBee device, network graph and driver
    Dispatcher(Xbee& x, Network& n, Driver& d);

    // scan incoming packet
    void scan(Packet* p);

    // analysis of data packet
    void handle_data(Packet* p);

    // analysis of acknowledge packet
    void handle_ack(Packet* p);

    // analysis of internal packet (other than receive xbee frame)
    void handle_internal(Packet* p);

    // send packet with delivery guarantee
    bool deliver(Packet* p);

    // send packet to adjacent node
    bool send(Packet* p);

    // locally broadcast packet
    void broadcast(Packet* p);

    // check for timeouted packet
    int tick();

    // retransmit packet
```

```
    bool retransmit(Metadata* meta);

    // retransmit packet if number of retransmissions is not maxed out
    bool try_retransmit(Metadata* meta);
};
```

`Dispatcher` can scan every incoming `Packet` for data required to guarantee successful packets delivery. Moreover it is charge of updating network parameters, based on statistics carried within acknowledge packets.

Packet analysis

Every `Packet` received by `Router` is scanned by `Dispatcher::scan()` before further processing. Various parameters are obtained depending on packet type. To make implementation more readable, handling method depending on packet type were introduced.

Data packets

`Packet` is considered as successfully delivered when `Packet` destination is self node. In this case positive acknowledge packet is generated and sent towards source node.

Internal packets

Internal packets encapsulated XBee `Frame` other than `ReceiveFrame`. Frame of type `StatusFrame` is processed inside `Dispatcher::handle_internal(Packet* packet)` as it contains delivery status information of last send `Frame`.

Every `StatusFrame` is associated with `frame_id`, that being so `Metadata` is retrieved from the `History`. Statistics of given `Packet`, stored inside metadata, such as delay, number of errors or retries are updated. Using the same statistics parameters of link between self and status source are updated.

If delivery status is negative, packet retransmission occurs (using `Dispatcher::try_retransmit`), unless retransmissions are no allowed any longer for this packet. In this case, negative acknowledge packet is sent towards packet source.

Receipt of positive acknowledge packet causes corresponding `Packet` and its `Metadata` to be removed from `History`, as they no longer be useful.

Acknowledge packets

Acknowledge packets have the same `PacketId` as corresponding data packet, so `Metadata` is easily retrieved from the `History`.

Acknowledge packet carries status and edge parameters in form of array of `RemoteParameters`. Local `Network` graph is updated with these statistics. Every acknowledge packet passing through intermediate node refreshes local graph, so statistics used to calculate antireliability measures are most recent.

Local node parameters retrieved from `History` for associated packet are appended to acknowledge packet. If local node is not associated packet source (origin of acknowledge packet) it is passed to next node (using `Dispatcher::send()`) following the same path the original packet was routed.

Otherwise if acknowledge packet reached its destination and its status is negative, retransmission of corresponding data packet occurs (if it is still allowed).

Transmission routines

Different packet transmission routines are implemented (as seen in table 3.12). Method `Dispatcher::deliver()` is most important in context of the thesis, as it ensures `Packet` delivery.

Table 3.12: `Dispatcher` transmission methods

Method	Range	Description
`send()`	physical neighbours	Send packet to locally adjacent neighbour without any guarantee of success
`broadcast()`	physical neighbours	Send packet to every locally adjacent neighbour (broadcast) without any guarantee of success
`deliver()`	any node in network	Send packet to any node in the network, guarantees delivery or information in case failed delivery

Both `send` and `broadcast` methods basically encapsulate `Packet` in `TransmitFrame` with proper MAC address and do not receive `StatusFrame` in response. Message is delivered back to Redis using `Driver::deliver_back()` in case the source node is not adjacent to destination node (message can be recognized as undelivered in client applications).

Delivery commitment

Delivery commitment in `deliver`, implemented accordingly to algorithm described in previous chapter, requires more processing. Most reliable path is calculated using `network.path(self->address , packet->destination, packet->visited)`. If such path could not be found and self node is source node (not intermediate one), message is delivered back in same manner as mentioned previously.

Packet is added to `History` and `frame_id` is obtained using `history.watch(packet, path, timeout (packet, path))`. Therefore `Metadata` is created for the packet and its delivery status is monitored. Packet's acknowledgement must be received within `timeout` otherwise, after number of consequent failures packet will be considered as lost. It will be delivered back to Redis as undelivered if failure is detected on source node, otherwise acknowledge packet marking failure will be send back to source, causing delivering back to Redis.

Packet retransmissions

Method `bool Dispatcher::retransmit(Metadata* meta)` was designed to support `Packet` retransmission. It is implemented comparably to `deliver` method, however as the `Packet` already exists in the `History` there is no need to use `History::watch()`. Instead new `Metadata::frame_id` is issued and updated in the `History`. Both new `Metadata::send_time` and `Metadata::timeout` are assigned. Packet is serialized and encapsulated into `Frame`, as such it is sent to XBee device.

Retransmission count of given packet is restricted by constant `Metadata::retransmission_max = 5`. Helper method `bool Dispatcher::try_retransmit(Metadata* meta)` was created which retransmit `Packet` stored in the `History`. With each retransmission adequate `Metadata::retransmission_counter` is incremented, until retransmission count reach `retransmission_max`. In such case the `Packet` is assumed to be undelivered and it is returned to client application.

Timeout system

Constructor of `Dispatcher` class creates background thread responsible for detecting data `Packet` with expired acknowledge timeout.

In this thread `int Dispatcher::tick()` method is executed periodically (with period set as `miliseconds Dispatcher::tick_sleep_time = 100`). Timeout is calculated for every `Packet` individually using

`time_point Dispatcher::timeout(Packet* p, Path& path)` as part of `deliver` method. Timeout method calculates deadline for acknowledge packet to return, accordingly to equation 2.2 – timeout depends upon `Path` to destination and intermediate link statistics such as delay, retries, errors and deliveries count.

For each `Packet` contained in `History` previously computed timeout is compared with current timestamp. If acknowledge packet is not received before timeout, retransmission is performed unless maximal retransmission tries was conducted. In this case the packet is marked as undelivered and its `Metadata` is removed from `History`.

3.2.5 Router

`Router` class is entry point of router application. It is responsible for routing process and topology discovery. Router creates connection with physical XBee device and obtains basic parameters from the device. Router works in simple processing loop as shown in listing 3.12.

Listing 3.12: `Router` processing loop pseudocode

```
while (should_run) {
  // receive next frame from xbee
  frame = xbee.receive();
  // deserialize packet
  packet = packet->from_frame(packet);

  // allow dispatcher to process the packet
  dispatcher.scan(packet);

  // data packet
  if (packet->type == Packet::Type::Data) {
    // if packet reached its destination
    if (packet->destination == self) {
      // publish it to redis so other applications can read the message
      driver.publish(packet);
    }
    // if this is intermediate node
    else {
      // let dispatcher move packet forward
      dispatcher.deliver(packet);
    }
  }
  // topology packet
  else {
    // analyze packet message so network topology can be updated
    proceed_with_topology_maintenance(packet);
  }
}
```

The pseudocode shows basis of operation, however, `Router` code is more complex (listing 3.13) and closely reflects algorithms presented in previous chapter.

Listing 3.13: Basic `Router` class

```
class Router {
  // constructs router with xbee device connected to given serial port and given address as self
       node address
  Router(char* serial_port, Address address);

  // creates router instance and processing loop
```

```
  static void run(char* serial_port, Address address);

  // receive and deserialize packet from xbee
  Packet* receive();

  // process next packet, behaves adequately to packet type
  void process();

  // broadcast heartbeat packet to let other nodes know about its existence
  void heartbeat();
};
```

Initialization

Instance of `XBee` device connector (described in previous section) is created. Radio power level is configured and basic parameters such as node name, network address and local MAC address are obtained. As a result `Node* Network::self()` is configured.

Router listens on channel `network:*:*:self` using `Driver::self`, so messages that need to be sent to other nodes from external applications can be intercepted and delivered using `Dispatcher::deliver()`.

Additional thread, broadcasting `Packet::Type::NodeBroadcast` every 15 seconds acts as heartbeat process and is required for proper topology discovery and maintenance process.

Receiving process

`Packet` is received from XBee device using blocking method `Packet* Router::receive()`. `Frame` acquired from XBee device is analyzed. `Packet` is deserialized from `Frame::Type::Receive`, however, other types of `Frame` cause `Packet::Type::Internal` to be generated and surround the frame.

Processing packets

After receiving the packet, it is let to be processed by `Dispatcher` in the first place. Then, packet is processed adequately to its type. Some packets carry data, other one can be acknowledge or topology maintenance packets. `switch` statement is to differentiate actions for the packet.

Data packet

Data packet can be processed in two ways, depending on its destination. If given node is packet destination (that is `self->address == packet->destination`), then contained data is published to Redis using `Driver::deliver` method. At this moment acknowledge packet has been already sent by `Dispatcher`.

Otherwise, given node is intermediate node on path to packet destination. As so, packet must be routed to its destination. After adding `Address self->address` to list of packet waypoints, the packet is sent further using `Dispatcher::deliver()`.

Acknowledge packet

On `Router` layer, acknowledge packets are used just for logging purposes.

NodeBroadcast packet

`NodeBroadcast` packets are received as `Node` heartbeat. As designed in topology discovery algorithm, edge between self node and node address contained in the packet is added to the `Network` graph. If graph was updated (that is self node did not have the edge in the graph already), current state of the graph is locally broadcasted.

EdgeDrop packet

`EdgeDrop` packets cause removal of `Edge` contained in the packet from self `Network` graph. Packet is further propagated (using `Dispatcher::broadcast`), only if the connection defined `Edge` was existing in the graph.

Graph packet

`Graph` packets are received from adjacent nodes after successful recognizing of new node. It is assumed that received graph may contain new nodes and edges. As so `Network::merge` operation is executed. If state of the local `Network` graph was changed, updated local graph is broadcasted.

3.2.6 Simulator

In order to make design and development process more reliable XBeePro-868 simulator was developed. Algorithm and implementation should be tested on various network topologies and in different environments. Such tests are not possible with five radio modules owned by authors. Tests could have been conducted using larger amount of XBee modules, however it is not an economical solution. In addition these tests could not have been reliable as real world environment can behave in unpredictable way.

Therefore simulator was created – one instance of the simulator can simulate network of physical XBeePro-868 nodes. The simulator uses POSIX feature such as pseudo terminals (called PTY). On POSIX platforms physical serial ports are represented as special files available as `/dev/tty*` path. Similarly PTY are special files (usually available as `/dev/pty*`) with similar behaviour to serial port – read and write operations are available on both sides of the PTY. As a result it is possible to deceive the router application and pass path to pseudo terminal instead path to real serial port – no changes in router code are required.

Simulator is developed in Ruby language, because of its great standard library and vast available community provided code as well as metaprogramming availability. It is possible to use simulator as standalone application using `bin/simulator` or as library used in other Ruby code. When used as standalone application list of paths to PTY and names of associated virtual XBeePro-868 devices is printed to standard output, various logs are printed to standard error stream.

Simulator splits network topology from environment definition. Network topology is definition of undirected graph with possible links between XBeePro-868 devices. Environment is file describing behaviour of nodes and links at any moment of time. Parameters such as node power state, link delay and number of frame retries and errors can be configured.

Basis of operation

In order to use simulator, network topology must be provided. Network topology is represented by undirected graph, where vertices represent XBeePro-868 module (identified by name) and edges represent aerial links between the modules. There is no theoretical limit on number of nodes or edges, however simulation efficiency can decrease when large dense graph is created

Table 3.13: Environment parameters

Parameter	Subject	Value	Description
Power	node	boolean	Power state of the node. Powered off devices cannot receive or transmit frames.
Retries	edge	number	Number of frame retries required to transfer frame between two nodes.
Delay	edge	number	Time necessary to transfer one frame between two nodes (for each retry delay is accumulated).
Errors	edge	boolean	Determines if frame is delivered when using given link.

Inspired by real world, environment definition must be supplied – environment defines behaviour of nodes and links. In real world node can be either powered on or off, links introduce delay depending on propagation time and distance and various events can contribute to number of transfer retries and errors (parameters summary is available in table 3.13).

Furthermore any of the parameters can be common for every node and link or can be defined for specific node or link. The environment uses directed version of network topology – it is possible to specify different parameters for two directions of same aerial link.

Moreover parameters values can be represented using probability density functions. This approach allows accurate modeling of real world behaviour. Various probability distributions are available, however, Gaussian (normal), Poission and degenerate distributions are most useful in this case [Tij12].

Output of probability density functions can be scaled and biased so parameters can be retained in specific ranges. Additionally probability density functions can be specified differently depending on time increasing reality of the simulation's model. Therefore value of specific parameter p is defined as value of stochastic process $X_{p,id}(t)$ in given time point t for link or node id as presented in following equation:

$$X_{p,id}(t) = a_{p,id} \cdot f_{p,id}(x,t) + b_{p,id}$$

Where:

$a_{p,id}$ is scale of parameter p,

$b_{p,id}$ is bias of parameter p,

$f_{p,id}(x,t)$ is random value generated using probability density function at given time t.

Parameters values are calculated in different ways as they need to be converted to appropriate types using following equations:

- power must be converted to boolean value

$$power_a(t) = \begin{cases} 0 & X_{power,a}(t) < 1 \\ 1 & X_{power,a}(t) \geq 1 \end{cases}$$

- retries cannot exceed maximum number of retries

$$retries_{a \to b}(t) = \begin{cases} X_{retries,a \to b}(t) & X_{retries,a \to b}(t) < retries_{max} \\ retries_{max} & X_{retries,a \to b}(t) \geq retries_{max} \end{cases}$$

- delay must incorporate number of retries

$$delay_{a \to b}(t) = X_{delay,a \to b}(t) \cdot (retries_{a \to b}(t) + 1)$$

- link error parameter must be converted to boolean value

$$errors_{a\to b}(t) = \begin{cases} 0 & X_{errors,a\to b}(t) \leq 0 \\ 1 & X_{errors,a\to b}(t) > 0 \end{cases}$$

Where:

a, b are node identifiers,

t is time point,

$retries_{max}$ is maximum number of frame retries.

It can be noticed that in given time t effective networking graph can change. If a node is marked as powered off, depending on $power_a(t)$, the node is effectively removed from the graph, so any transmission to the node will fail. Similarly if an edge's $errors_{a\to b}(t)$ parameter denotes link failure, transmission from a to b will fail. Therefore environment configuration can dynamically change effective network graph (examples can be seen in figure 3.10).

Processing loop

With presented parameters definition, following processing loop is executed for each node.

A frame is received from given node pseudo terminal. Frame is deserialized from raw byte sequence.

If frame is `CommandFrame` type appropriate radio parameters are set or read and `CommandResponseFrame` is sent back to the node.

If frame is `TransmitFrame` decapsulated data are passed to destination nodes' `receive` method. In case of broadcast transmission, the data are passed to every adjacent node in effective graph. If transmission can occur, that is effective graph computed using environment state in given time contains edge connecting source and destination node, `ReceiveFrame` is created.

The data are encapsulated into the `ReceiveFrame`, after delay depending on $delay_{a\to b}(t)$ the receive frame is serialized and sent over destination's pseudo terminal to destination's application. `StatusFrame` is generated and sent to source node pseudo terminal. In case of transmission failure (depending on $power_b(t)$ and $errors_{a\to b}(t)$) the status frame contains appropriate statistics.

Every event in processing loop creates log entry tagged with node name, frame data and time relative to start time of simulation. Probability related functions are implemented using community developed `courier-rb` library.

Input files

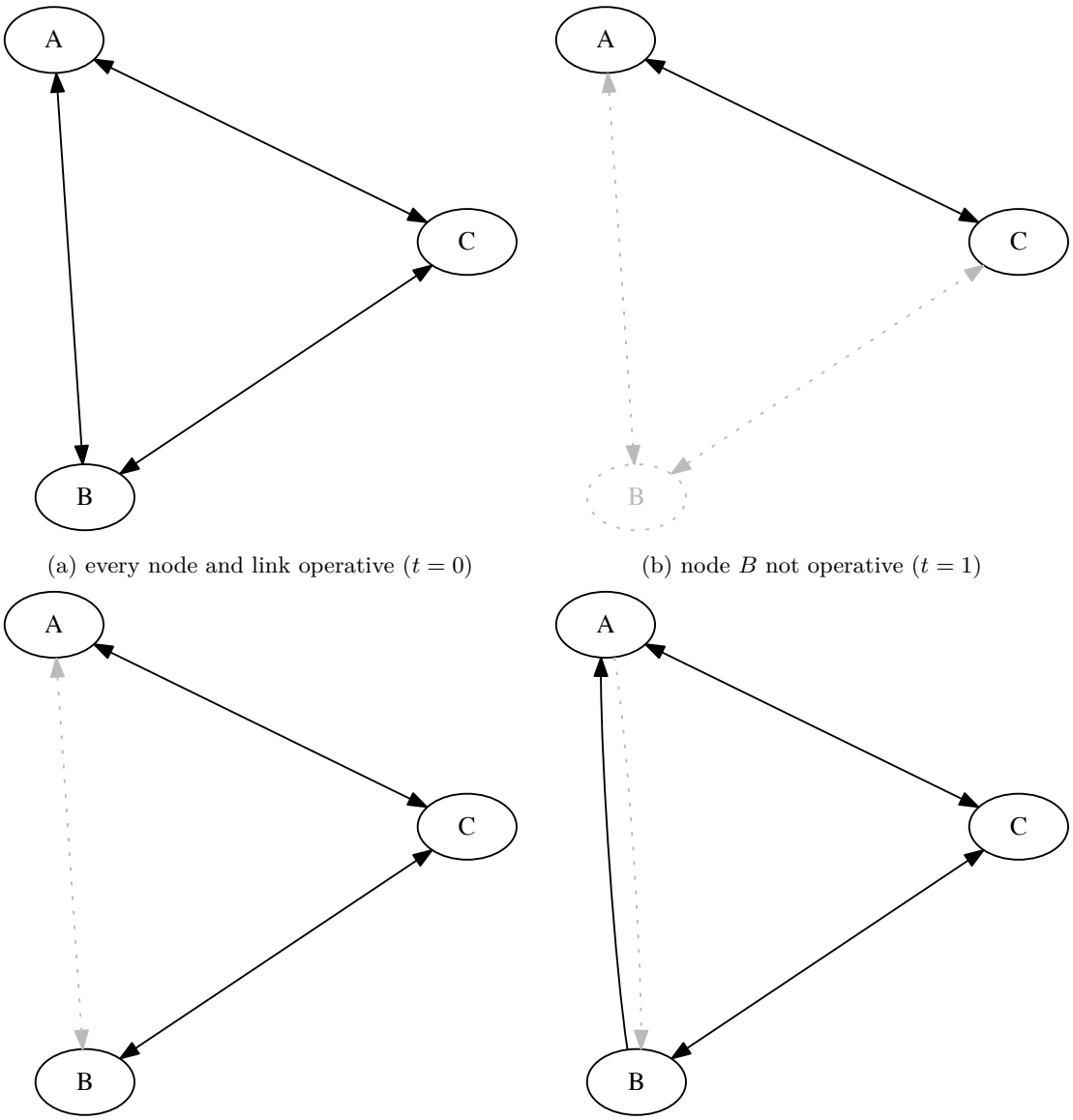
Simulator requires two input files. These files describe network topology and environment. It can be noticed that environment definition is quite detailed and may require complex input file.

In order to make input files human readable YAML file format was chosen. YAML is human readable, indentation based, popular data format. It is assumed syntax in examples provided below is self-explanatory. Complete description can be found in reference card [YAM06].

Nodes are identified by name, so any unique number or string can be node identifier.

Network topology

Network topology is represented as adjacency map – every node is associated with list of adjacent nodes. Example chain topology network can be seen in listing 3.14. By convention network definitions are stored in `*.network.yml` files.



(a) every node and link operative ($t = 0$)

(b) node B not operative ($t = 1$)

(c) link $A \to B$ and $B \to A$ not operative ($t = 2$)

(d) only link $A \to B$ not operative ($t = 3$)

Figure 3.10: Network graph in various moments of time
Effective graph is shown in black.

Listing 3.14: Seven nodes chain network topology

```
1: [2]
2: [1, 3]
3: [2, 4]
4: [3, 5]
5: [4, 6]
6: [5, 7]
7: [6]
```

As network topology graph is undirected, both direction entries are redundant and may be omitted (as can be seen in listing 3.15).

Listing 3.15: Three nodes complete graph

```
alfa:
  - beta
beta:
```

```
  - gamma
gamma:
  - alfa
```

Two different syntaxes can be observed in listings 3.14 and 3.15, presenting two admissible ways of enumerating lists in YAML.

Environment definition

Environment definition requires file structure which allows to specify common or specific parameters distribution for nodes and edges in given time points. Therefore environment file structures is divided into time range which can start in some absolute time point or after delay relative to previous time point. Definition of example "perfect" environment can be seen in listing 3.16 – such environment guarantees transfer of every frame.

Listing 3.16: Example perfect environment definition

```
start:
  point: 0

  edges:
    all:
      delay: 5
      retries: 0
      errors: 0

  nodes:
    all:
      power: 1
```

Each time range is labeled with identifier and its start time point must be specified. To specify absolute time moment `point` key must be used, in order to specify time relative to previous time point `delay` key can be used.

Most recent parameters definitions are used – if parameter for given node or frame is not defined in current time frame, last definition is used. In each time frame parameters for edges can be specified (in section `edges`) so as nodes parameters (in section `nodes`). A key `all` can be used to set parameters of every edge or node. To set parameters for specific node, its name must be used as key in `nodes` section. In similar fashion setting link between node `a` and node `b` parameter requires to use array `[a, b]` as key in `nodes` section.

Parameter distribution can be specified in two ways, depending on required distribution. Degenerate distribution can use simplified syntax as it can be defined using one constant. Other distributions require full distribution definition. To pair parameter with distribution a parameter name must be used as a key and list of properties must be passed, where `distribution` key is required (for degenerate distribution single number can be passed instead of properties).

Every distribution can use `bias` and `scale` parameter, however, any distribution requires other specific constants. Distribution names and their required arguments are dependent on `croupier-rb` implementation – list of useful distribution names and their parameters can be seen in table 3.14. More distribution specifications are available in `croupier-rb` gem documentation.

Complex example presenting described parameters is available in listing 3.17. Three time ranges will occur – `start` at $t = 0$, `storm` at $t = 10$ and `back_to_normal` at $t = 60$. Short syntax for degenerate (constant) distribution, parameters specific for edge `[alfa, gamma]` and node `beta` and usage of YAML references can be noticed.

Table 3.14: Available probability distributions

Distribution name	Default parameters	Description
normal	$mean = 0.0$, $std = 1.0$	Normal (Gaussian) distribution with given mean and standard deviation.
poission	$lambda = 50$	Poisson distribution with given lambda.
degenerate	$constant = 42$	Discrete distribution which returns the same value each time.
uniform	$included = 0.0$, $excluded = 1.0$	Uniform distribution generating number between *included* and *excluded*, where each value is equally likely.

Listing 3.17: Complex environment example

```
start:
  point: 0

  edges:
    all: &perfect_edges
      delay: 10
      retries: 0
      errors: 0

    [alfa, gamma]:
      retries:
        distribution: uniform
        included: 0
        excluded: 5

  nodes: &nodes_on
    all:
      power: 1

storm:
  delay: 10

  edges:
    all:
      delay:
        distribution: uniform
        included: 20
        excluded: 50

  nodes:
    beta:
      distribution: normal
      mean: -1
      bias: -0.2

back_to_normal:
  point: 60

  edges:
    <<: perfect_edges

  nodes:
    <<: nodes_on
```

Implementation summary

Simulator code is divided into multiple parts however following classes has crucial application:

- `XBee::Node` – single XBeePro-868 node, communicates over its own pseudo terminal, deserializes frames and responses to various events,
- `XBee::Network` – manages graph of nodes, it is responsible for calculating effective graph and properties,
- `XBee::Frame` – representation of XBeePro-868 frame.

Classes' implementation were divided into multiple partial classes files grouped by implemented functionality type.

Frame implementation

Implementation of XBeePro-868 frames is required to operate on data serialized as raw bytes sequences. Every frame documented in XBeePro-868 manual [Dig14b] is implemented as Ruby class inside `XBee::Frame` module.

Domain Specific Language

Domain Specific Language was introduced to overcome problem of frames implementation. Domain Specific Languages are computer languages designed to be specialized in distinct problem [VDKV00]. Human friendly DSL to solve the problem was designed as superset of Ruby language. It is executable Ruby code even though it looks like new programming language.

Listing 3.18: `TransmitFrame` implementation

```
frame 0x10, :transmit,
  id: 'C',
  mac: 'Q>',
  network: 'S>',
  command: 'a2',
  status: 'C',
  data: '*a'
```

Using various metaprogramming methods, procedure `XBee::Frame::frame(id, name, **args)` dynamically create class which represents XBee frame identified by `id` and `name`. Field structure is specified using `**args` dictionary – every field is specified by name and `Array::pack` compatible string [Rub13]. Therefore each frame can be specified as in similar way as example `TransmitFrame` listing 3.18. Each frame implementation supports serialization using `to_bytes` and deserialization using `from_bytes`.

Supported events

Only subset of operations supported by XBeePro-868 device is implemented – the subset covers every frame required for proper router simulation. Implementation reacts to frame received frames such as `TransmitFrame` and `CommandFrame`.

Frame are handled inside `Node` class. Using metaprogramming another DSL feature was provided – `XBee::Node::receive(type, **args, &block)` and `XBee::Node::reply(type, **args)` methods.

Using `receive` method allows to filter incoming frame by its type and by contained data values. It allows to write event-based code as it can be seen in listing 3.19 – the example presents implementation of handling broadcast-type `TransmitFrame`.

Table 3.15: Commands available in simulator

Name	Description	Behaviour in simulator
FR	resets the radio	restores Node parameters back to default
NI	reads node name	name is set based on value in topology definition file
ID	reads network identifier	network identifier is constant 0xFFFE as instance of simulator can simulate only one network)
SL	reads lower eight bytes of MAC	MAC addresses are generated automatically by Network and they are guaranteed to be unique)
SH	reads higher eight bytes of MAC	these bytes are constant and set to 0x0013A200
PL	sets power level	no influence on simulation behaviour
MT	sets retransmission limit	sets $retries_{max}$
DB	reads signal strength	returns mock value

Listing 3.19: Broadcast TransmitFrame handler

```
receive :transmit, mac: 0xFFFF do |frame|
  broadcast frame.id, frame.data
end
```

Method reply enable creating, serialization and sending frame over pseudo terminal to client application. Example usage showing CommandResponseFrame reply with response to NI command is shown in listing 3.20.

Listing 3.20: NI command response

```
reply :command_response,
  id: frame.id,
  command: 'NI',
  status: 0,
  data: @id.to_s
```

Supported commands

Only subset of configuration commands is supported in the simulator. These commands are sufficient set to make router application fully operational. Implemented commands list is available in table 3.15. Further commands implementation can be easily added if application requires them.

3.3 Testing process

In order to ensure that the implementation of the network protocols was properly designed various tests has been conducted. Two testing strategies were chosen – unit testing and behaviour testing. Tests has been written concurrently with the implementation, so any flaws has been fixed as soon as they have been found.

3.3.1 Unit testing

Unit testing is concept of testing small individual units of software's source code [PM14]. Individual methods of implementation are tested in simplified environment. Therefore implementation errors can be found independently and their localizations can be easily determined.

Unit tests has been written with help of googletest library [Goo15]. *Network* and *Dispatcher* were seriously tested as core parts of the implementation. These tests were partitioned into tests cases (table 3.16) comprehensively testing various combinations of input data. In the end 15 test

Table 3.16: Test cases

Name	Testing Goal
`timeout`	Calculation of proper timeouts depending on collected edge statistics.
`add_node`	Ability to add new node unless node is already in the graph.
`node`	Accessing node by its address.
`nodeForbiddenAddress`	Awareness of invalid addresses when adding node or edge.
`add_edgeWhenNodesDoNotExist`	Creating nodes associated with new edge when these nodes are not already in the graph.
`add_edgeWhenNodesExist`	Addition of new edge to the graph.
`add_edgeSelfLoop`	Inability to add loopback connection.
`dropEdge`	Removing edge form the graph.
`mergeGraphs`	Merging two graphs should result in addition of non existing nodes and edges to target graph.
`mergeTheSameGraphs`	Merging edges already existing in target graph should not change the graph.
`pathSingleChain`	Most reliable path between two nodes should be calculated in basic chain graph.
`pathNotChainGraph`	Most reliable path between two nodes should be calculated in complex graph.
`pathInTreeGraph`	Most reliable path between two nodes should be calculated in tree.
`pathNoPath`	Empty path should be expected when route between two nodes cannot be calculated.
`pathNoChainGraphVisited`	Calculated path should not contain already visited nodes.

cases were formulated with over 120 assertions. Multiple flaws were found and fixed using this testing method.

3.3.2 Behaviour testing

Behaviour testing has been conducted as part of behaviour-driven development process. In this technique various behaviours are described as *scenarios* which consists of preconditions, actions and expected results [PM14]. The whole system, in our case implementation of network protocols, is tested at once so complex behaviour can be described and examined.

In case of the project, behaviour of protocols' implementation can be studied in various topologies and environment using previously designed simulator. As the simulator was implemented in Ruby language, *Cucumber* application [Asl15] has been chosen as it easily integrates with the simulator.

Scenarios of the tests were written in *Gherkin* language [Ye13]. Gherkin has natural-language-based syntax and it is used to describe various scenarios combined. Each scenario can be described by sequence of steps. Each step can be either precondition, action or result. Set of logically combined scenarios is called *feature*. Scenarios collected inside on feature can share common background – list of steps executed before every scenario. Example scenario, testing basic behaviour of chain network, is available in listing 3.21.

Listing 3.21: Feature describing basic behaviour of chain network

```
Feature: Testing basic chain network
  Background: 7 nodes network in perfect environment
    Given simulation of 02_chain network in perfect environment
    And every router is alive
    And topology is discovered
```

```
Scenario: Send from first node to second node
  When 1 sends to 2 message "hello"

  Then 2 receives from 1 message "hello"

Scenario: Send from first node to last node
  When 1 sends to 7 message "hello"

  Then 7 receives from 1 message "hello"
  And 1 receives acknowledge from 2
  And 2 receives acknowledge from 3
  And 3 receives acknowledge from 4
  And 4 receives acknowledge from 5
  And 5 receives acknowledge from 6
  And 6 receives acknowledge from 7
```

Variety of steps are provided in order to efficiently and accurately write scenarios. List of designed steps is available in table 3.17.

With possibilities given by implemented steps, comprehensive feature descriptions has been written. These features covers variety of network functionality such as topology discovery, packet routing, handling undelivered messages and behaviour in variety of environments. Behaviour tests made whole process of protocols' implementation much easier as every aspect of network behaviour was tested for proper operation.

Table 3.17: Designed Cucumber steps

Step	Description
timeout is *timeouts*	Set *timeout* for time-bound steps. Default timeout is 5 seconds.
simulation of *topology* network in *env* environment	Spawns simulator with given network *topology* file in given *env* environment file. Fails when simulation cannot be started within given *timeout*.
time is *timepoint*	Sets simulation time to given *timepoint*.
node *name* is down	Simulates power of node with given *name*.
router *name* is alive	Spawns instance of router on network node with given *name*. Fails when router is not spawned within *timeout*.
every router is alive	Starts router instance on every network node. Fails when routers are not spawned within *timeout*.
topology is discovered	Waits until every router discovers the topology. Fails when topology is not discovered on every router within *timeout*.
source sends to *destination* message "*message*"	Publishes *message* on designated Redis channel, effectively causing *message* send operation from *source* node to *destination* node.
node receives acknowledge from *destination*	Waits until *node* receives an acknowledge packet from corresponding data packet's *destination* node. Fails when acknowledge is not received within *timeout*.
destination receives from *source* messages "*message*"	Waits until *destination* node receives *message* from *source* node. Fails when *message* is not received within *timeout*.
source receives back message "*message*"	Waits for undelivered message originated from *source*. Fails if *message* is eventually delivered.
message route is *path*	Checks if *message* route is equal to given *path*.
source broadcasts data *data*	Checks if *source* node broadcasted raw *data* bytes within given *timeout*.
source transmits to *destination* data *data*	Checks if *source* has sent raw *data* bytes to given *destination* within *timeout*.
packet is broadcasted from *source*	Checks if given *packet* type (node or graph) is broadcasted from *source* node.

Chapter 4

Sample applications

This chapter describes sample applications created in order to present the usage of designed protocols. Each application is written in C++ 11 as a command line program and uses API of network driver described in section 3.2.3. Echo and temperature reading applications, beside driver API, use only standard C++ 11 library. Console application use GNU Readline[1] library in addition.

4.1 Echo

Echo is simple application that shows basics of receiving and sending data using designed network. Purpose of this application is to receive data and send back exactly the same data to the sender of received data.

Echo as an optional parameter takes a port number on which it accepts data packets. By default it listens on the port 15. After determining port number, application creates instance of `PUT::CS::XbeeRouting::Driver` class and invokes `listen` method on this instance. Listing 4.1 shows whole implementation of the program.

Listing 4.1: Echo application

```
#include "../driver/driver.h"

using namespace PUT::CS::XbeeRouting;

int main(int argc, char const* argv[]) {
  Driver* driver = new Driver();
  uint8_t port;

  if (argc == 1) {
    port = 15;
  } else {
    port = atoi(argv[1]);
  }

  driver->listen(port, [ = ](Address source, uint8_t* data, size_t length) {
    if (source != Driver::SELF)
      driver->deliver(source, port, data, length);
  });

  pause();
  return 0;
}
```

[1] http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html

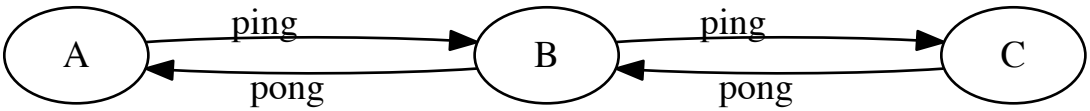


Figure 4.1: Example of using echo application

First argument of `listen` method is port in which echo application listen for packets. Second argument of `listen` method is handler for received messages. In this case, it simply sends back received data as new packet using `deliver` method of `PUT::CS::XbeeRouting::Driver` instance. The network from figure 4.1 reveals the following usage examples:

1. node A sends message to node C,
2. node C reads this message and sends the same data as in received message back,
3. node A receives message from node C with the same data as sent in point 1.

4.2 Temperature reading

Another example of application that uses designed protocols is temperature reading. Application is designed for RaspberryPi [UH12] with Raspbian operating system, but it should also work on notebooks and PCs with Debian like Linux distributions. It reads actual temperature of CPU and sends that value on demand. Similar to echo application, it takes one optional argument, which is port number for incoming requests from other nodes. By default, port number equals 7.

Application reads temperature of CPU from suitable system file and divides value by 1000 to obtain temperature in Celsius degree. Listing 4.2 shows how this is done:

Listing 4.2: Reading temperature of CPU

```
using namespace std;
float temperature() {
  ifstream f("/sys/class/thermal/thermal_zone0/temp");

  if (!f.is_open())
    return -1;

  int temperature = 0;
  f >> temperature;

  return temperature / 1000.0;
}
```

Main functionality of this application is similar to echo. Its code consists of obtaining port number, creating instance of `PUT::CS::XbeeRouting::Driver` class and invoking `listen` method on it. Implementation differs only in handling received data. This is shown in listing 4.3:

Listing 4.3: Handler for received data

```
driver->listen(port, [ = ](Address source, uint8_t* data, size_t length) {
    if (source != Driver::SELF) {
      if (strncmp((char*)data, "gettemp", 7) == 0) {
        driver->deliver(source, port, (uint8_t*)to_string(temperature()).c_str(), s.length());
      }
    }
  });
```

Handler checks if source node of received data is not self node. Then checks whether data is equal to "gettemp" which is proper request for fetching response from this application. When the above requirements have been meet, the obtained temperature is sent to request source using `deliver` method on `PUT::CS::XbeeRouting::Driver` instance.

Example use case for temperature reading application and network topology from figure 4.2 can be as follows:

1. node A sends "gettemp" message to node C,
2. node C reads temperature of its CPU,
3. node C sends this temperature to node A,
4. node A receives message with temperature from node C.

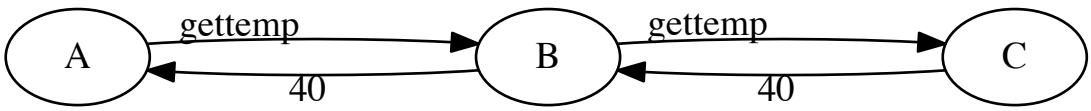


Figure 4.2: Example of using temperature reading application

4.3 Console

Applications described above react only when data is sent to them. Purpose of console application is to allow users to send data to chosen node at chosen port from command line. This application was very useful as debugging tool while implementing communication protocols.

Console application takes two command line arguments while launching: first one is port number (this is required argument), on which application listen for packets and second one is path to file or device to which information about packets will be written. As default this argument is set to `stdout`.

Whole application, excluding obtaining command line arguments, consists of two parts. First one creates `PUT::CS::XbeeRouting::Driver` instance and invokes `listen` method on this instance. This is shown in listing 4.4:

Listing 4.4: Creating Driver instance and invoking listen method

```
Driver* driver = new Driver();
driver->listen(port, [ = ](Address destination, uint8_t p, Address source, uint8_t* data, size_t
    length) {
  if (source == Driver::SELF)
    fprintf(descriptor, "packet␣send:\n\tto:␣%d,␣port:␣%d,␣length:␣%zu\n\tdata:␣%s\n\n",
        destination, p, length, data);
  else
    fprintf(descriptor, "packet␣received:\n\tfrom:␣%d,␣port:␣%d,␣length:␣%zu\n\tdata:␣%s\n\n",
        source, p, length, data);
});
```

First argument of `listen` method is port in which console application listen for packets. The second one is handler for incoming and outgoing packets. It simply writes to chosen file/device information about received/sent packets and data from these packets.

Second part of application is responsible for displaying command prompt and parsing commands typed by user. Listings 4.5 and 4.6 show how this is implemented.

Listing 4.5: Main loop of application

```
bool run = true;
char* input, shell_prompt[100];
while (run) {
  // create prompt
  snprintf(shell_prompt, sizeof(shell_prompt), "port:%d>␣", port);

  input = readline(shell_prompt);

  if (!input)
    break;

  if (strcmp(input, "") != 0)
    add_history(input);

  if (strcmp(input, "exit") == 0)
    run = false;
  else
    parse_command(input, driver, port);

  free(input);
}
```

Listing 4.5 shows main loop of application. It simply creates command prompt and displays it using `readline` function. This function also reads input typed by user and stores it in `input` variable. Then, if `input` variable is not null and is not an empty string, typed command is added to command history using `add_history` function, so it is accessible using up and down arrows (like in bash). `readline` and `add_history` functions come from GNU Readline Library. Next, it checks if command is not equal to "exit", which results exiting application. If it is not, then `parse_command` function, shown in listing 4.6, is invoked.

Listing 4.6: parse_command function

```
void parse_command(char* command, Driver* driver, uint8_t port) {
  char* action = (char*)malloc(sizeof(char) * 10);
  int destination = 0;
  char* data = (char*)malloc(sizeof(char) * 255);

  sscanf(command, "%s␣%d␣%[^\n]s", action, &destination, data);

  if ((strcmp(action, "deliver") == 0) || (strcmp(action, "d") == 0)) {
    driver->deliver(destination, port, (uint8_t*)data, strlen(data));
  }

  free(data);
  free(action);
}
```

This function parses command typed by user and invoke proper action for typed command. Command have following structure: `action destination data` where the only possible action is `deliver` (or shortcut: `d`) – sends packet to given node on given port using `PUT::CS::XbeeRouting::Driver::deliver` method, For example, `deliver 2 some_kind_of_data` will cause delivery some_kind_of_data to node 2 on port given as command line argument to console application.

4.4 Possible areas of usage

There are many places where designed protocols can be used and many applications which can be written to use them. As examples, system for measuring time during sport events and sensoric networks are presented below.

4.4.1 Sport timing

As mentioned before, one possible area of usage for designed protocols is measuring time during sport events like marathons, triathlons, ski country and so on. Whole system could look as follows. Along entire marathon track, there are gates that records timestamps for every runner when he passes through gate. At every gate and at arbiter position, there is XBeePro 868 [Dig11a] device connected to RaspberryPi [UH12]. These devices create one network using communication protocols and algorithms described before. At every gate, there are RFID [Wan06] aerials with controller and they are connected to RaspberryPi device. Every runner has RFID chip pinned to his leg. When he passes through gate, application on RaspberryPi gets through RFID aerials and controller unique identifier of runner stored in RFID chip, concatenates it with actual timestamp and sends this information to arbiter position via created at the beginning network. Then, at arbiter position, received data are matched with name of runner, stored with identifier in database and can be shown to arbiter and supporters.

4.4.2 Sensoric networks

Another area of usage for designed protocols is measuring different kinds of parameters in pipelines or other transmission networks. For example, protocols can be applied to measuring pressure in gas pipeline. Scenario is very similar to sport timing. Along entire pipeline, there are places where XBeePro 868 [Dig11a] connected to embedded system are placed. At control station, there is another XBeePro 868 device connected to computer that process received data. These devices create network described in previous chapters. Every RaspberryPi device along pipeline has connected pressure sensor. Application reads periodically pressure and sends this information to control station, where this information is processed. Obviously, there is nothing against the use of greater number of sensors with single RaspberryPi device, but the number of input/output ports in RaspberryPi and throughput of system should be taken into account.

Chapter 5

Performance evaluation

In order to analyze the protocols, variety of tests are performed. Firstly topology discovery time is investigated, than message routing was benchmarked using load tests, spike tests and harsh environment tests [Sub06]. Such tests should give back clear image of protocols' design and implementation behaviour. Analysis of protocols' behavior under the load and in prone to crashes environment is accomplished in the section 5.6 at the end of this chapter.

5.1 Methodology

All of tests have been carried out with simulator (subsection 3.2.6) to provide the required test conditions and avoid random interference. Moreover the use of a simulator provided tests repeatability. After measuring XBeePro-868 devices' transmission delay, delays on each link were simulated with normal distribution determined by 20ms mean and 1ms standard deviation. All other parameters that do not have direct influence upon test behaviour, such as message contents, have been generated randomly.

Message roundtrip time was chosen as benchmark measure. Therefore, each node was running *echo* application (section 4.1). Time measured between sending message from source node to destination and receiving it back from destination on source node is reliable indicator of protocols behavior. With such response time, protocols' and implementations' performance can be easily compared in various kinds of network topologies. Furthermore as short as possible roundtrip time is a desirable from the end-user perspective, because request-response is common network application workflow. Each test was performed 20 times and the reported results are the average values. Complete numerical values can be found in Appendix A.

5.1.1 Tests environment

Significant tests parameters are network topology and its size. Following topologies were selected for every test:

- chain graph – every node is connected to maximum two nodes and there is no cycles (figure 5.1),
- spider web graph – topology resulting of star and ring union (figure 5.1),
- random sparse graph – random graph generated using Gilbert model [Gil59] with 0.02 to 0.2 probability of edge existence, depending on network size (figures 5.3, 5.4).

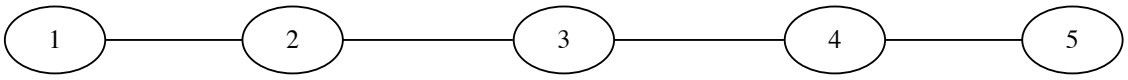


Figure 5.1: Example chain network visualization (5 nodes)

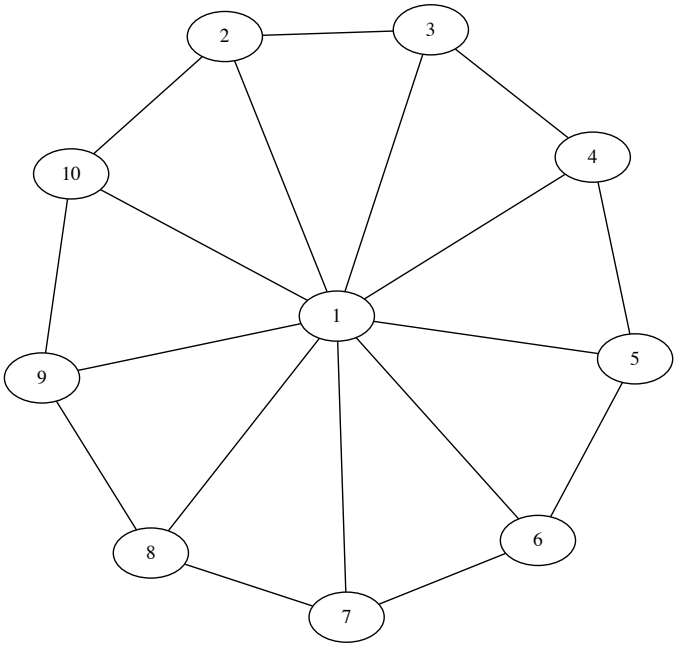


Figure 5.2: Example spider network visualization (10 nodes)

It is worth noticing that chain graph is a pessimistic network topology, so it holds the upper bound of roundtrip time, it is least reliable kind of topology, however it offers the greatest geographical span. Spider web graph can represent real world network built with radio modules with omnidirectional antennas arranged around one central node. However the most important topology is random sparse graph – it represents real world network, where every node has between two and four direct neighbours and nodes positions are distributed quite unpredictably.

For each topology, following order of graph sizes values n were used: 5, 10, 20, 30, 40, 50 to examine behaviour in variety of network sizes. Other parameters like number of sent messages will be individual for each tests and are presented in appropriate sections.

5.1.2 Results visualization

In order to facilitate the analysis of the collected results, test results are visualized on the line charts in appropriate sections. In each test case there is one line chart to the one topology. Charts present average response time dependence on the number of nodes. If test assumes variability of parameter m (number of messages), then on each chart suitable number of curves is presented.

Test results should confirm or refute the claims, that protocols work correctly on different conditions. The shape of the curves determine the behaviour of protocols. It is expected that with the increased order of graph the response time will also rise, as response time between furthest nodes is dependent on network diameter. However, this growth should not be exponential. Moreover curves for lower values of parameter m should be dominated by curves for greater value of this parameter.

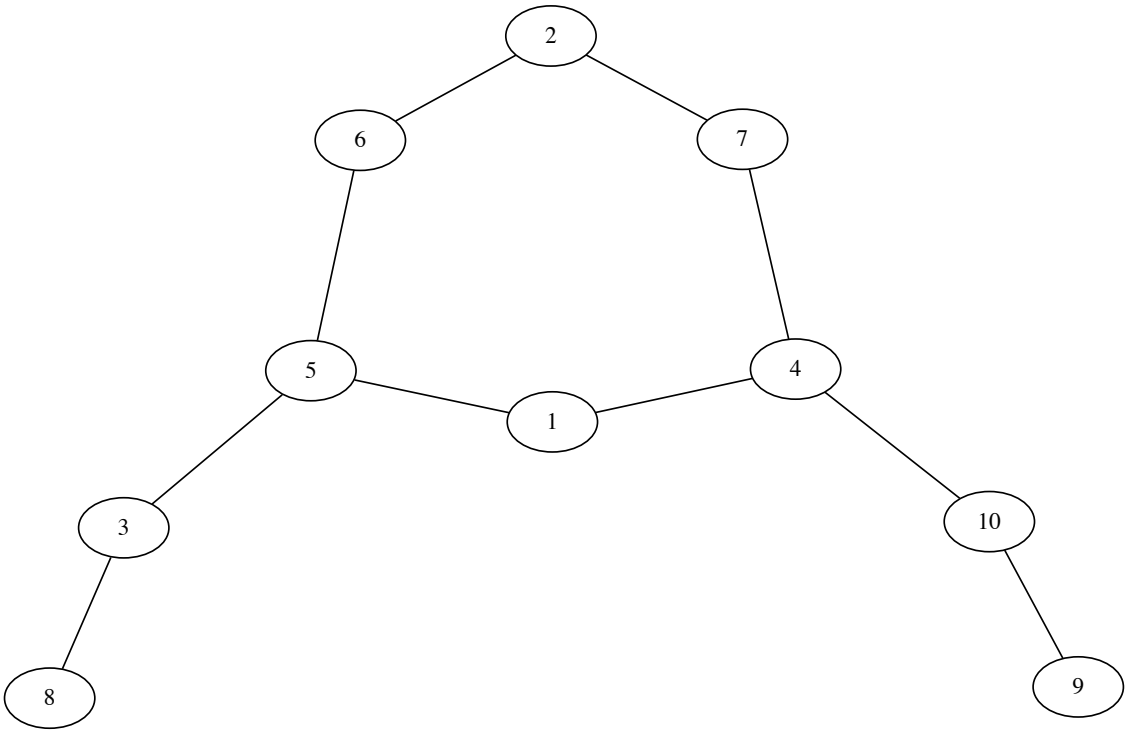


Figure 5.3: Example sparse random network visualization (10 nodes)

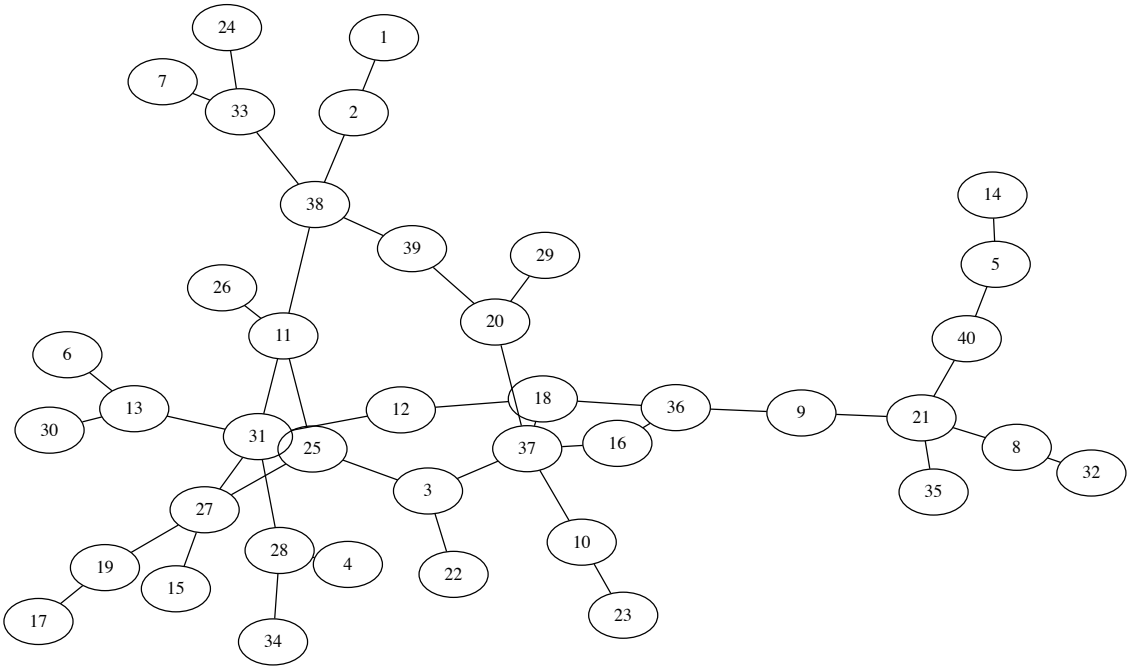


Figure 5.4: Example sparse random network visualization (40 nodes)

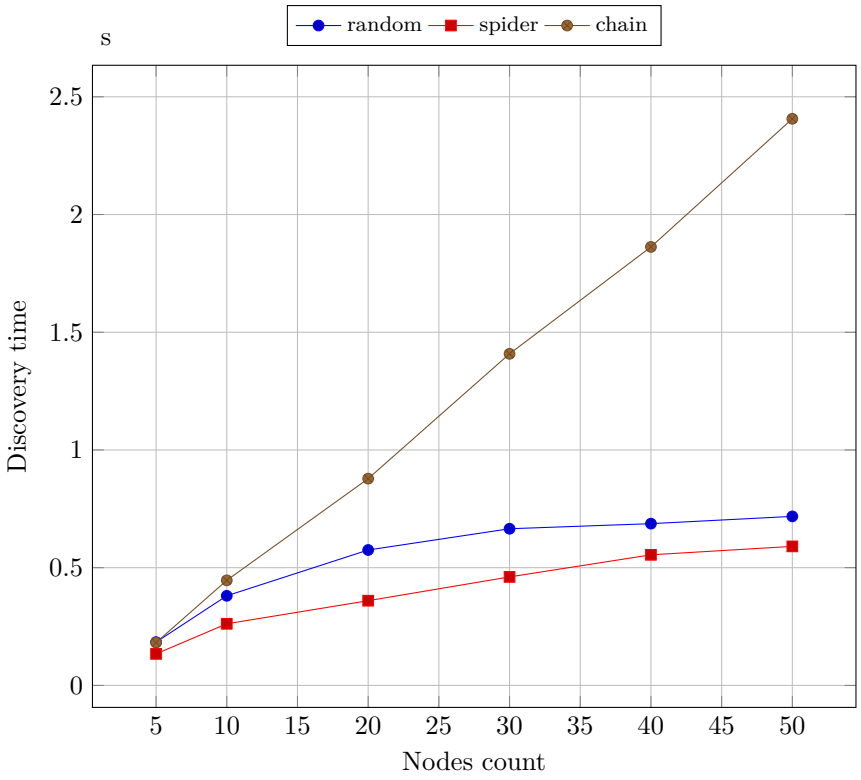


Figure 5.5: Complete topology discovery time

5.2 Topology discovery

Important feature of the presented protocol is self-organization, an essential part of this feature is topology discovery. It is limited by sending out a broadcast in wireless network. Multi-hop nature makes it only more difficult and time-consuming. In order to examine how the protocols handle topology discovery, the following test was conducted.

In the same moment all nodes become enabled (their state is changed from passive to active) and they join the network, this moment of time will be referred to as start time. Moment when last node discovers complete network topology (that is every node achieved consistent and final state of network topology) is called discovery end time. Difference between these two times, discovery end time and start time, is interpreted as complete topology discovery time.

It is assumed that all nodes initiate topology discovery protocol at the same time. This behaviour allows discovering various parts of the network at the same time. However, some mutual interference effects can be observed with adjacent nodes.

In figure 5.5, it can be seen that chain topology discovery is slowest as each node is connected to one or two other nodes only. That fact causes chain network to be the worst case, yet still it can be noticed that complexity is linear in network size domain.

In case of spider web topology, observed outcome seems to be optimal. Increasing the network's size results in slight increase of topology detection time and the curve is resembling logarithmic complexity. The fact of quick reach of a consistent state results from small network diameter and constant degree for all vertices (excluding the central node). Detection of a new node can be propagated to every other node in a very small time, as the central node is connected to every node. These factors make this topology an optimal one.

Topology discovery in random sparse graph seems to be little worse than in spider web topology.

It can be seen, that their curves are closely related. As random graph reflects real world scenario, measured 750ms required to discovery topology on every node seems promising.

It is worth mentioning that after network topology is discovered (the consistent state is reached), new node joining the network will obtain complete network graph in constant time. As only one heartbeat and graph transmission from any adjacent node is required to acquire network graph, this operation can be completed in less than 50ms. However, propagation of this new node in to whole network is bound by the laws presented before.

5.3 Load testing

Load testing is measuring network behaviour when it is under load. In case of this test it is necessary to simulate load. To realize this task, 20 messages are continuously sent between random nodes.

Test system sends m number of special messages from node A to node B and measures the roundtrip time. Where A and node B nodes are the outermost vertices on the graph diameter. Eventually time between sending first of m messages and receiving back the last message is used as measure. Presented numbers of special messages (m) are used: 5, 10, 20, 30, 40, 50.

Chain topology chart (figure 5.6) shows linear response time rise with increasing number of nodes. As chain topology is worst case scenario, it strongly dominates other topologies' behaviour.

In spider web topology load test (figure 5.6), it can be noticed that roundtrip time is not dependent upon order of the graph. This is correct behaviour due to to the fact, that in average case at most one intermediate node is required to connect pair of nodes. In consequence response time is more less constant.

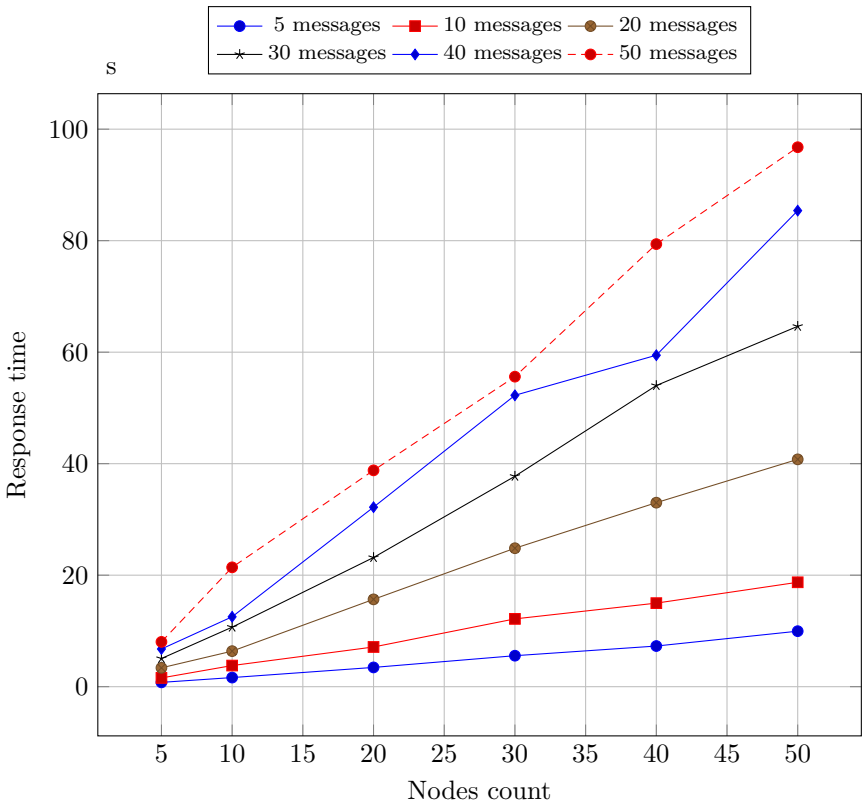


Figure 5.6: Load test in chain network

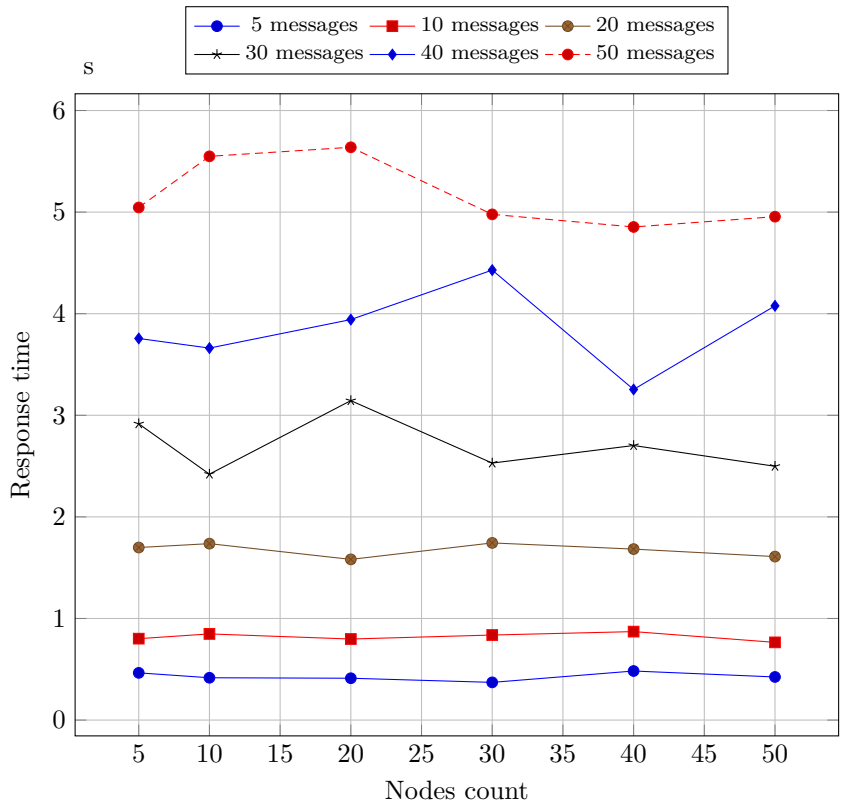


Figure 5.7: Load test in spider network

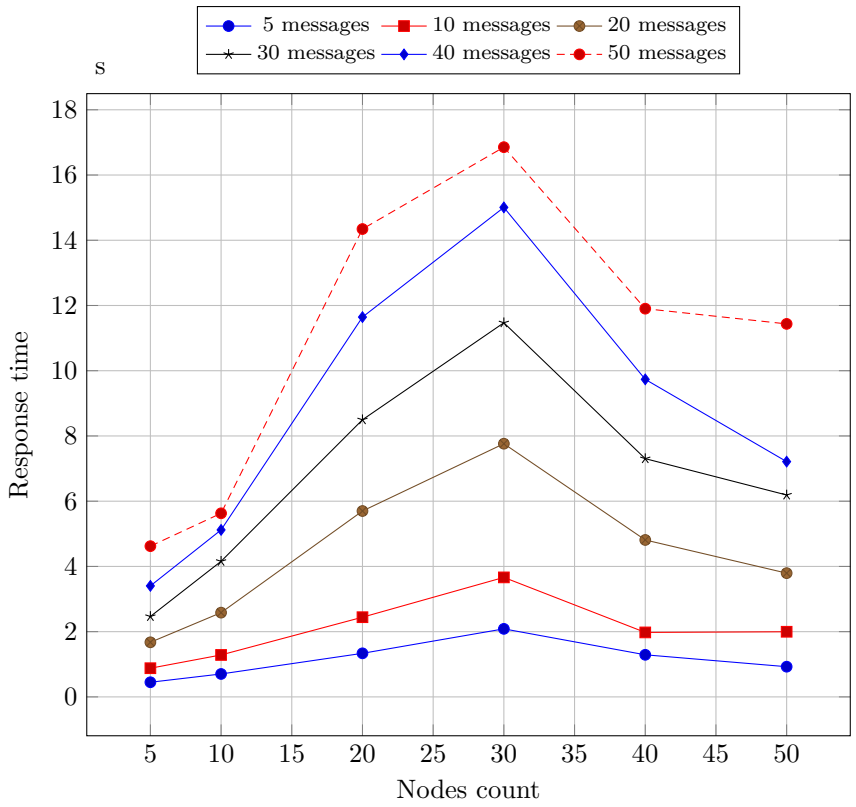


Figure 5.8: Load test in sparse random network

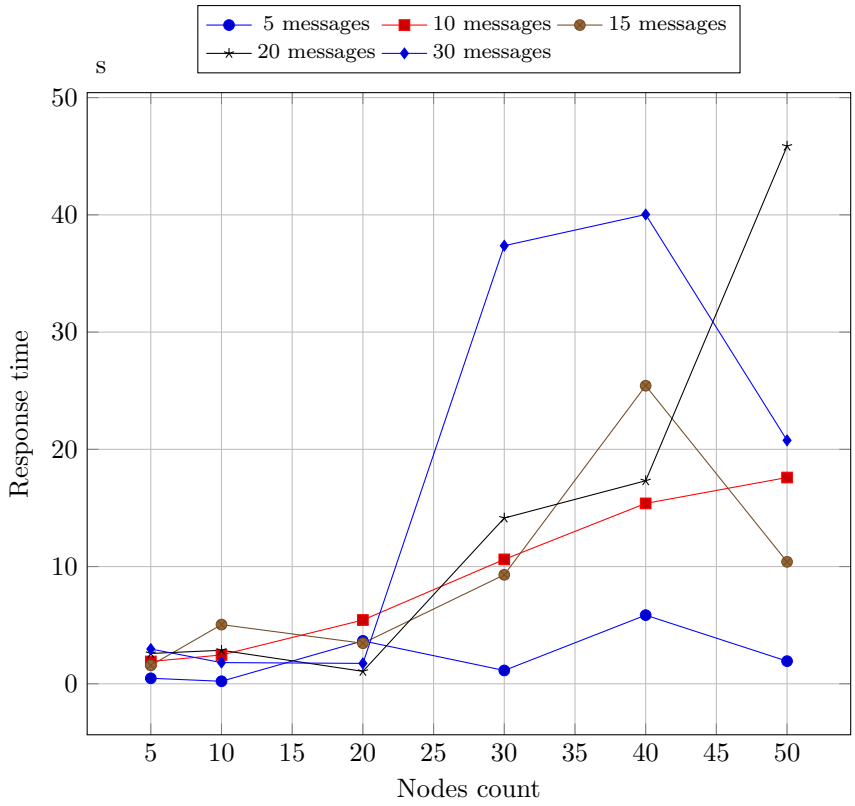


Figure 5.9: Spike test to random node on chain network

Results for random sparse graph (figure 5.8) are the most interesting as this topology reflects real world situations. It can be seen, that the curves rise a peak at 30 nodes – this phenomenon is closely correlated with graph diameter. The graph diameter grows with the order of graph, until some threshold number of nodes is reached, where the diameter starts decreasing. This is a result of the chosen graph generation method.

5.4 Spike testing

Spike testing is measuring system response when system load suddenly increased.

Two useful varieties of this test were performed:

- messages are sent from each node to other randomly chosen node (figures 5.9, 5.10, 5.11),
- messages are sent from each node to their furthest node (figures 5.12, 5.13, 5.14).

It should be noticed that transfers occur concurrently, that is each node starts sending m messages independently in the same time, so in total $m \cdot n$ messages are transferred. Tests were performed for following number m of messages: 5, 10, 15, 20, 30.

Measured roundtrip times are lower in comparison with load tests results, even though multiple transfers occur in parallel. This is a consequence of a smaller number of messages circulating in the network. Spider web graph still is leading network topology, no matter if transfers occur between random or furthest nodes. Observed behaviour of network based on random sparse graph is acceptable as roundtrip times are far better than values of chain network, which is a pessimistic topology.

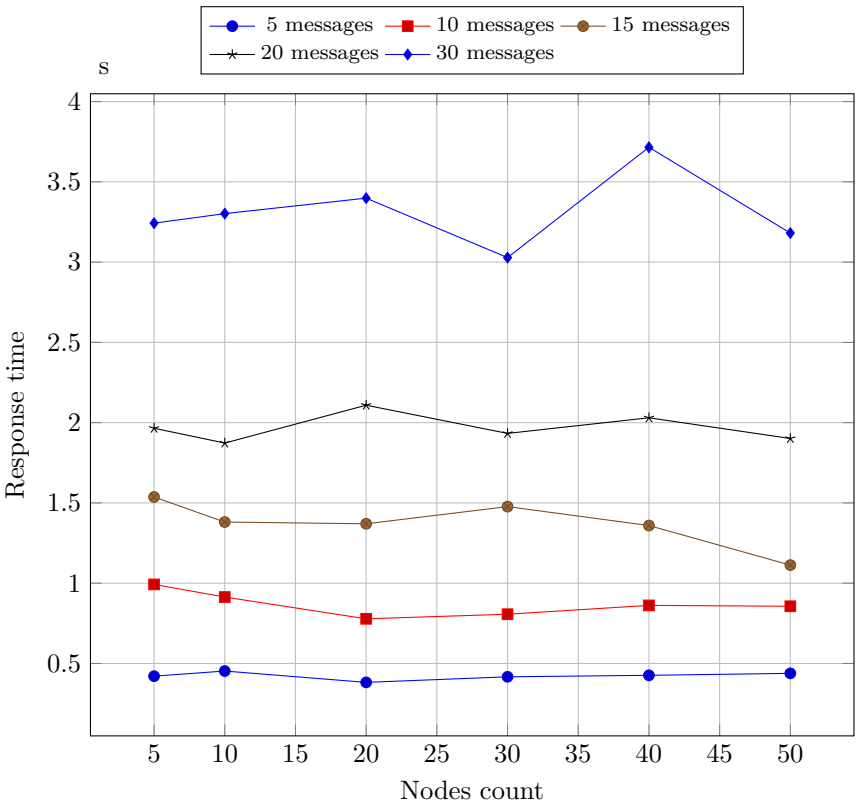


Figure 5.10: Spike test to random node on spider network

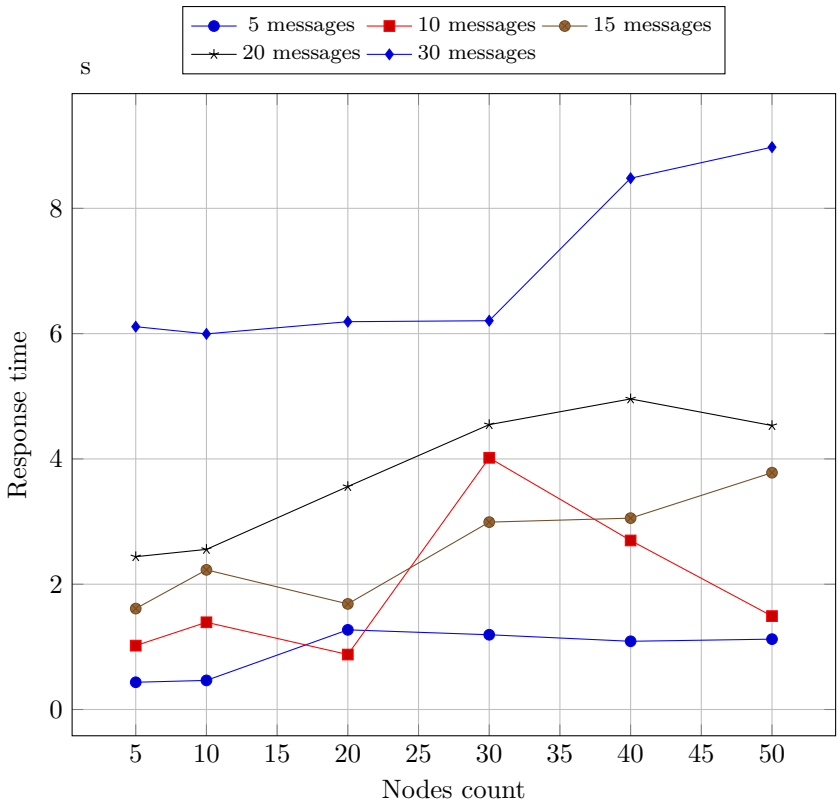


Figure 5.11: Spike test to random node on random network

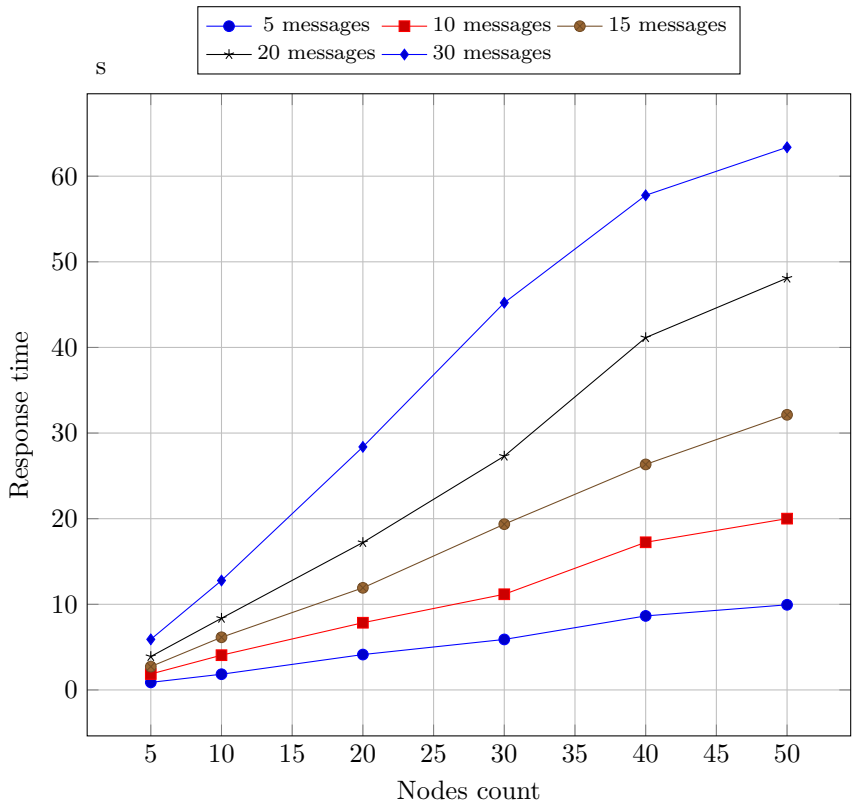


Figure 5.12: Spike test to furthest node on chain network

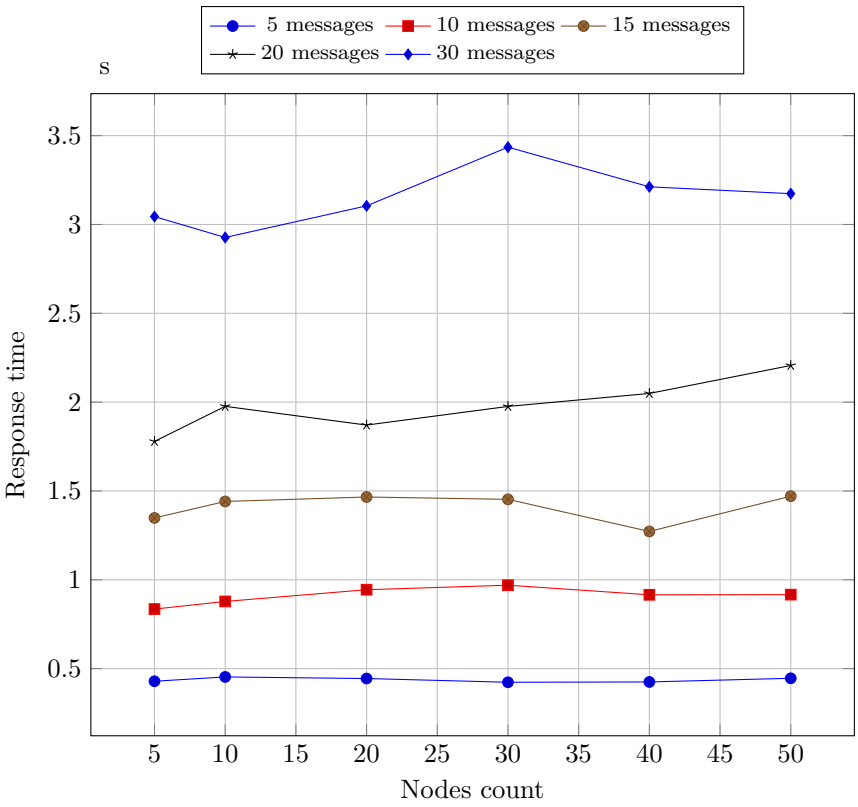


Figure 5.13: Spike test to furthest node on spider network

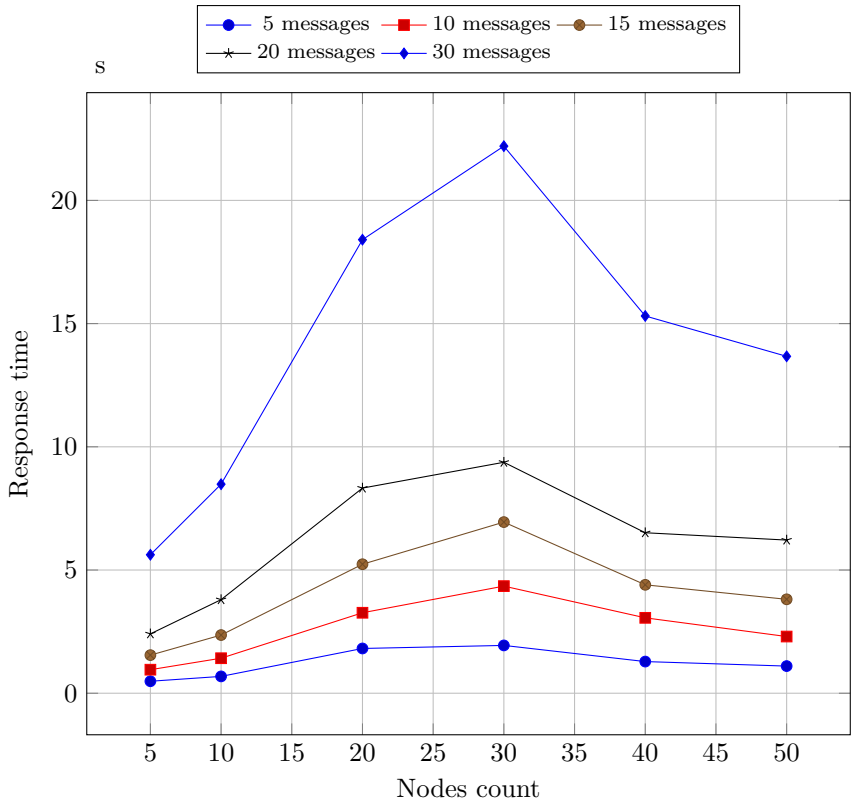


Figure 5.14: Spike test to furthest node on random network

Conclusively, it can be noted that concurrent messages in network do not cause unpredictable behaviour.

5.5 Harsh environment testing

Harsh environment is defined as network, where randomly occurring failures of nodes or retransmissions can happen. In real life situation network effective topology can change as a result of bad weather, power failures or interferences.

With simulator implemented in the thesis it is possible to mimic such environment. Therefore it is possible to test protocols and their implementation easily in controlled conditions.

The harsh environment test is based on load test, however different environment is assumed:

a) constant 50% probability of retransmission on every edge (figure 5.15),

b) constant 75% probability of retransmission on every edge (figure 5.16),

c) 40% probability of retransmission on every edge 100ms after start (figure 5.17).

As harsh environment test is closely related to real world situation, tests were conducted only on random sparse graphs. Known number m of messages (where m is 5, 10, 15, 20) are sent from random node A to its furthest node and roundtrip time is measured. Time required to receive back every sent message is used as measure.

Harsh environment testing is fundamental in estimating protocol's stability in prone to crashes real world environment.

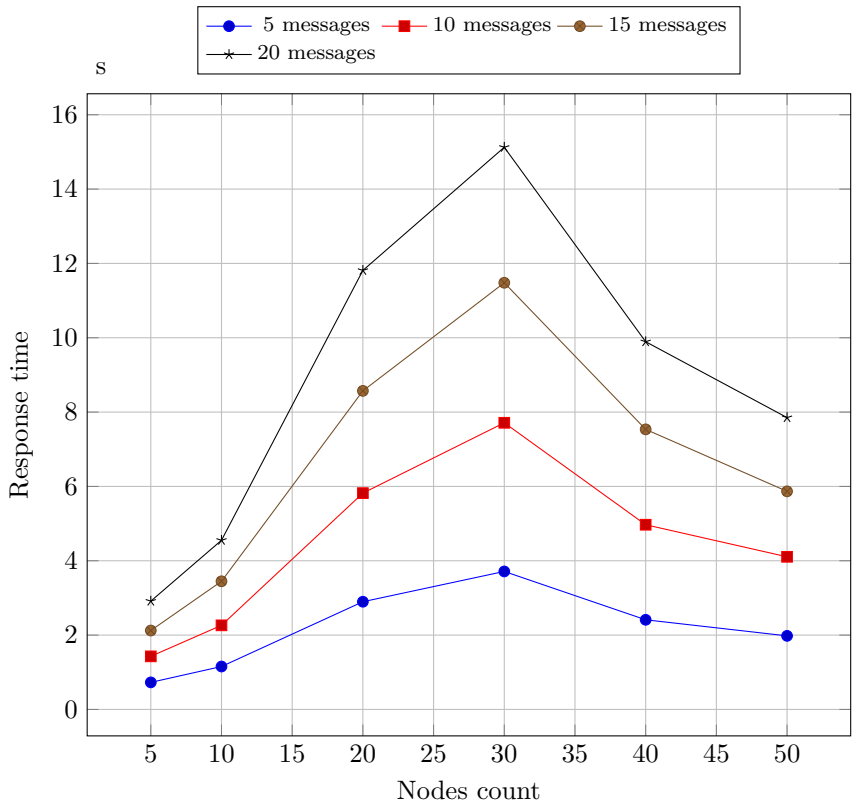


Figure 5.15: Harsh environment testing with 50% probability of retransmission

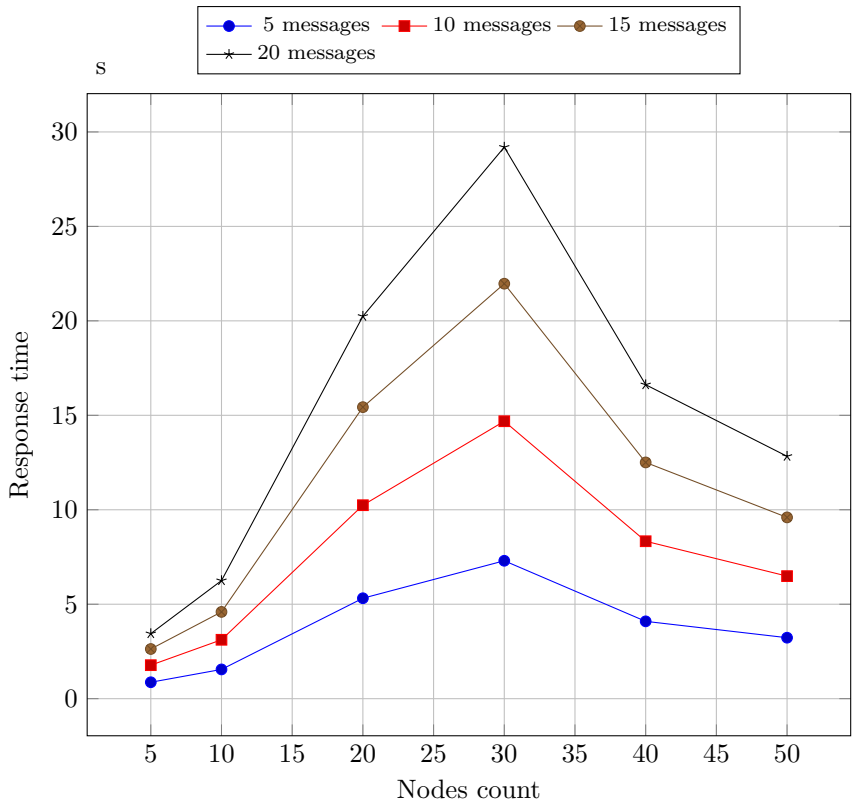


Figure 5.16: Harsh environment testing with 75% probability of retransmission

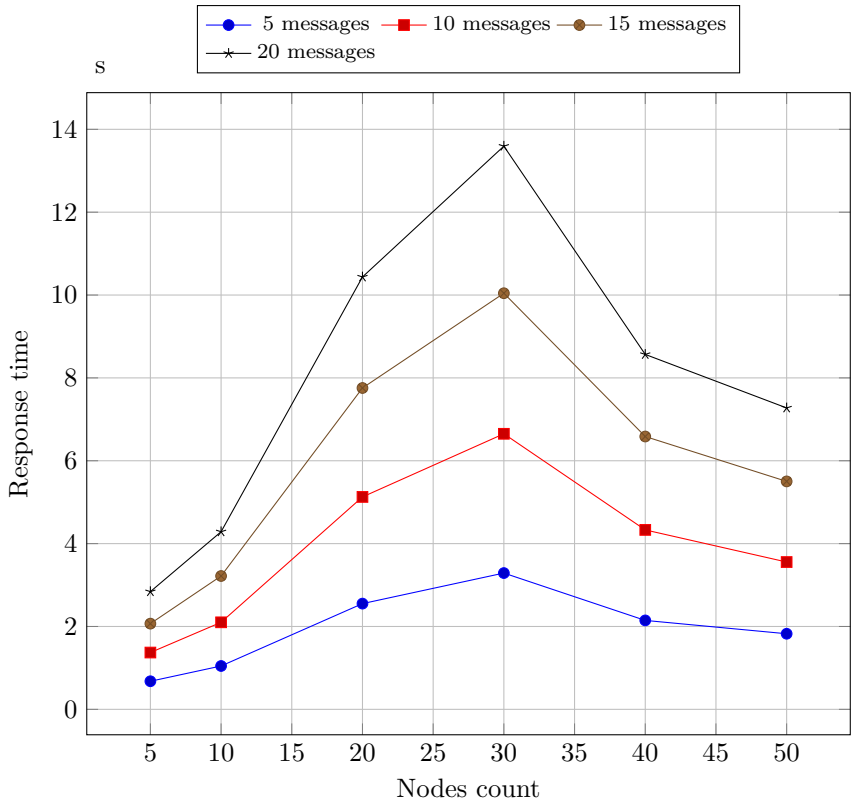


Figure 5.17: Harsh environment testing with 40% probability of retransmission 100ms after start

In presented charts similar peak as in load tests can be noticed, however its source is the same as previously described. In comparison with load tests 5.8, measured roundtrip times have increased considerably, yet still shapes of the curves are comparable. Time increase results from packet retransmissions required to ensure the delivery (during the test no undelivered messages were found). Correct protocols behaviour can be observed, even when environment parameters change in time, as in figure 5.17.

5.6 Summary

Performed tests provide sufficient information about the quality of protocols and their implementation. Tests confirm the protocol's ability to be scalable in context of the order of graph and network diameter. It can be noticed that computation time is minor part compared with transmission time, especially when data rate is 24Kbps.

In fact, the benchmark confirmed that designed protocols are adequate for use in real world situations. Tests based on random sparse graph topology were useful in prediction of behaviour in physical world. Topology can be discovered in satisfactory time. Messages can be transmitted effectively, even when under constant load. Moreover network can adapt to changing environment, as it makes use of redundant links in case of other link failures.

Chapter 6

Conclusions

In this thesis we provided the concept of brand new communication protocols for self organizing multi-hop long distance ad hoc networks. As part of the design process, it was proved that these protocols have *No creation* and *Stubborn delivery* properties. We also proved that having these two properties results in protocols correctness and implementing *Stubborn Point to Point links* interface.

Furthermore, sample implementation of designed protocols with example applications, such as echo, temperature reading and console is provided. Also example hardware platform based on XBeePro-868 and RaspberryPi is set up. As there was limited number of devices, network simulator that allows to simulate various network topologies and environment conditions was developed for the purposes of tests.

Various performance tests were conducted using this simulator. Load tests, spike tests and harsh environment tests shown, that scalability of designed protocols depends on size and topology of the network and matches performance expectations. Furthermore, harsh tests also shown ability of protocols to self-organize/reorganize while reliability of edges change.

In addition to conducted tests, provided implementation in connection with example applications and hardware setup shown that designed protocols can be used in real applications.

6.1 Comparing to existing solutions

Comparing to existing solutions, such as cellular network, ad hoc networks that use designed protocols have many advantages. Firstly, cellular network is not available everywhere, especially away from cities and roads. It is very important in case of measuring time during sport events. Triathlons, for example, can take place away from city and cellular network range. Thanks to the fact, that designed protocols implement multi-hop ad hoc network, large area may fall within a range of the network. And because range of single XBeePro-868 devices with proper antenna is up to 40 kilometers, check point gates that measures time does not have to be close to each other.

Joining new device to the existing network is very simple in provided implementation and is not more complex than in cellular networks. To connect the device to the network, turning the device on and inputting unique address is sufficient. Because designed protocols implement self organizing ad hoc network, device discovers network topology by itself. Furthermore, thanks to self-organizing property, network is able to react and reorganize if any temporary links disturbances occur.

Another advantage, comparing to cellular network is lack of data transfer charge. Cellular network operators charge for even small data transfer, while sending data through ad hoc network

is free if proper bandwidth is used.

The biggest advantage of provided solution is reliability. Failure of one device does not cause failure of whole network. In cellular, failure of base station can cause unavailability of network on large area.

6.2 Further research

Although the goal of this thesis has been accomplished, there is couple directions in which further research can follow:

1. As mentioned above, joining to the network requires setting unique address for each device. Automatic address configuration protocol is worth of investigation for making joining process even simpler.

2. Greater ensurance of message delivery can be investigated. Provided solution tries to ensure, that message will be delivered to destination node, but there is still no certainty about this. Increasing delivery ensurance is worth of researching.

3. Because of number of available XBeePro-868 devices was limited, performance evaluation was conducted using simulator. Conducting complex tests with usage of physical devices is worth of investigation.

Appendix A

Performance evaluation dataset

Table A.1: Topology discovery time (time in ms)

	Topology type		
Nodes	random	spider	chain
5	183.83	133.89	182.05
10	380.59	261.67	446.28
20	575.14	359.77	878.22
30	665.37	460.84	1 408.42
40	687.04	554.67	1 862.36
50	717.91	590.51	2 406.63

Table A.2: Load test in chain network (time in ms)

	Messages count					
Nodes	5	10	20	30	40	50
5	774.91	1 570.78	3 383.16	5 052.23	6 759.78	8 038.79
10	1 636.25	3 789.29	6 375.33	10 678.00	12 537.07	21 407.66
20	3 455.85	7 121.86	15 662.50	23 170.68	32 196.94	38 802.32
30	5 557.82	12 161.15	24 834.30	37 743.08	52 264.37	55 606.29
40	7 291.13	14 987.34	33 006.21	54 017.31	59 454.87	79 383.53
50	9 955.75	18 727.36	40 775.20	64 636.29	85 388.45	96 762.51

Table A.3: Load test in spider network (time in ms)

	Messages count					
Nodes	5	10	20	30	40	50
5	449.08	879.30	1 673.71	2 465.54	3 403.05	4 621.36
10	702.91	1 286.19	2 583.22	4 156.72	5 119.25	5 628.70
20	1 335.83	2 443.32	5 698.82	8 496.59	11 644.52	14 343.81
30	2 085.96	3 663.97	7 762.44	11 470.17	15 007.17	16 854.41
40	1 290.56	1 976.94	4 811.29	7 304.15	9 735.35	11 902.04
50	926.00	1 997.19	3 794.13	6 189.43	7 213.05	11 435.77

Table A.4: Load test in sparse random network (time in ms)

	Messages count					
Nodes	5	10	20	30	40	50
5	464.65	801.40	1 698.52	2 915.97	3 756.20	5 045.64
10	416.51	847.98	1 736.56	2 420.67	3 661.56	5 548.96
20	411.74	797.75	1 581.62	3 144.31	3 942.15	5 638.12
30	371.04	837.08	1 743.69	2 529.69	4 429.35	4 977.77
40	482.86	870.82	1 682.84	2 702.47	3 254.15	4 853.75
50	424.34	764.85	1 609.20	2 498.20	4 077.27	4 954.96

Table A.5: Spike test to random node on chain network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	470.53	1 902.76	1 572.32	2 587.91	2 966.90
10	216.00	2 459.97	5 049.13	2 858.97	1 808.64
20	3 661.92	5 439.98	3 459.62	1 061.80	1 739.96
30	1 137.80	10 613.00	9 296.79	14 137.83	37 369.08
40	5 856.01	15 386.77	25 425.83	17 319.04	40 030.62
50	1 932.07	17 596.21	10 402.50	45 859.91	20 765.54

Table A.6: Spike test to random node on spider network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	420.88	992.33	1 537.02	1 965.11	3 242.56
10	452.89	913.76	1 380.87	1 872.89	3 301.98
20	382.23	778.21	1 370.06	2 108.60	3 398.44
30	416.91	806.74	1 477.04	1 933.06	3 028.07
40	426.21	861.81	1 359.06	2 030.23	3 715.26
50	438.39	856.36	1 112.72	1 901.42	3 180.94

Table A.7: Spike test to random node on random network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	435.07	1 019.82	1 611.12	2 439.25	6 110.96
10	464.72	1 392.40	2 229.32	2 555.95	5 995.88
20	1 271.66	877.52	1 685.63	3 559.32	6 190.59
30	1 193.38	4 017.76	2 991.32	4 547.24	6 206.21
40	1 089.23	2 698.51	3 054.69	4 957.76	8 480.96
50	1 123.25	1 491.02	3 780.12	4 533.02	8 976.86

Table A.8: Spike test to furthest node on chain network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	882.53	1 859.55	2 728.08	3 903.42	5 908.81
10	1 834.87	4 058.08	6 149.36	8 369.64	12 778.78
20	4 131.86	7 848.51	11 923.99	17 219.20	28 382.35
30	5 897.06	11 173.98	19 357.47	27 305.17	45 217.16
40	8 644.62	17 244.29	26 337.81	41 145.86	57 767.44
50	9 942.38	20 006.18	32 131.77	48 093.84	63 375.59

Table A.9: Spike test to furthest node on spider network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	428.94	835.11	1 348.06	1 778.93	3 044.48
10	453.19	878.13	1 440.95	1 976.24	2 926.91
20	444.22	944.12	1 465.98	1 870.90	3 104.64
30	423.26	969.55	1 452.84	1 975.75	3 435.17
40	424.99	915.58	1 272.10	2 048.87	3 212.02
50	445.57	916.48	1 470.66	2 206.34	3 173.43

Table A.10: Spike test to furthest node on random network (time in ms)

	Messages count				
Nodes	5	10	15	20	30
5	485.96	954.92	1 545.93	2 408.73	5 619.27
10	682.99	1 420.72	2 360.47	3 797.08	8 483.34
20	1 816.46	3 265.55	5 234.71	8 324.88	18 406.30
30	1 940.31	4 346.74	6 945.03	9 372.48	22 200.96
40	1 284.77	3 059.96	4 399.58	6 512.52	15 309.61
50	1 101.56	2 300.72	3 811.33	6 215.49	13 671.11

Table A.11: Harsh environment testing with 50% probability of retransmission (time in ms)

	Messages count			
Nodes	5	10	15	20
5	728.04	1 428.13	2 123.05	2 913.48
10	1 154.69	2 262.72	3 447.43	4 551.22
20	2 897.69	5 819.81	8 570.71	11 812.35
30	3 712.79	7 712.22	11 479.36	15 124.80
40	2 411.70	4 969.35	7 535.47	9 893.62
50	1 979.51	4 106.48	5 867.48	7 851.98

Table A.12: Harsh environment testing with 75% probability of retransmission (time in ms)

	Messages count			
Nodes	5	10	15	20
5	869.47	1 773.25	2 629.32	3 443.92
10	1 548.50	3 118.16	4 593.50	6 251.12
20	5 316.66	10 241.07	15 432.29	20 242.58
30	7 300.23	14 686.32	21 966.39	29 194.77
40	4 093.62	8 334.08	12 505.87	16 620.05
50	3 227.85	6 487.17	9 595.44	12 830.99

Table A.13: Harsh environment testing with 40% probability of retransmission 100ms after start (time in ms)

	Messages count			
Nodes	5	10	15	20
5	677.81	1 370.94	2 068.59	2 841.22
10	1 046.37	2 100.59	3 218.27	4 286.53
20	2 551.89	5 127.02	7 756.25	10 436.26
30	3 289.39	6 650.91	10 043.47	13 593.55
40	2 146.43	4 330.82	6 585.47	8 566.67
50	1 824.53	3 555.47	5 501.65	7 274.88

Bibliography

[ADS86] Bowen Alpern, Alan J Demers, and Fred B Schneider. Safety without stuttering. *Information Processing Letters*, 23(4):177–180, 1986.

[Asl15] Aslak Hellesoy and Cucumber developers team. Cucumber – making bdd fun. [on-line] `http://cukes.info`, 2015.

[Bar98] Michael Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, 47(2):263–263, 1998.

[BRG99] Meenakshi Bansal, Rachna Rajput, and Gaurav Gupta. Mobile ad hoc networking (MANET): Routing protocol performance issues and evaluation considerations. *The Internet Society*, 1999.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer, 2011.

[CMPC04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic algorithms for reliable content-based publish-subscribe: An evaluation. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 552–561. IEEE, 2004.

[Dic11] American Heritage Dictionaries. *The American Heritage Dictionary of the English Language, Fifth Edition.* Houghton Mifflin Harcourt Trade, 2011.

[Dig11a] Digi International Inc. Xbee-pro 868 datasheet. [on-line] `http://www.digi.com/pdf/ds_xbeepro868.pdf`, 2011.

[Dig11b] Digi International Inc. Xbib-u-dev reference guide. [on-line] `http://ftp1.digi.com/support/documentation/xbibudev_referenceguide.pdf`, 2011.

[Dig14a] Digi International Inc. Xbee rf modules - digi international. [on-line] `http://www.digi.com/products/wireless-wired-embedded-solutions/zigbee-rf-modules/zigbee-mesh-module/`, 2014.

[Dig14b] Digi International Inc. Xbee®/xbee-pro®868 rf modules. [on-line] `http://ftp1.digi.com/support/documentation/90001020_E.pdf`, 2014.

[Dig15] Digi International Inc. About digi international. [on-line] `http://www.digi.com/aboutus/`, 2015.

[Dij59] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[DR91] Joan Daemen and Vincent Rijmen. The design of Rijndael: AES — the Advanced Encryption Standard. *Journal of Cryptology*, 4(1):3–72, 1991.

[EFGK03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[Eur09] European Union Parliament. Directive of the european parliament and of the council amending council directive 87/372/eec on the frequency bands to be reserved for the coordinated introduction of public pan-european cellular digital land-based mobile communications in the community. [on-line] `http://www.europarl.europa.eu/sides/getDoc.do?pubRef=-//EP//TEXT+REPORT+A6-2009-0276+0+DOC+XML+V0//EN`, 2009.

[Eur11] European Telecommunications Standards Institute. Electromagnetic compatibility and radio spectrum matters (erm); radio frequency identification equipment operating in the band 865 mhz to 868 mhz with power levels up to 2 w;. [on-line] `http://www.etsi.org/deliver/etsi_en/302200_302299/30220801/01.04.01_40/en_30220801v010401o.pdf`, 2011.

[Fac15] Antenna Factor. Ant-868-cw-hwr-rps datasheet - specifications: Frequency: 868mhz (853mhz - 883mhz). [on-line] `http://www.digchip.com/datasheets/parts/datasheet/1359/ANT-868-CW-HWR-RPS.php`, 2015.

[Gil59] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, pages 1141–1144, 1959.

[Goo15] Google developers team. googletest, google c++ testing framework. [on-line] `https://code.google.com/p/googletest/wiki/Primer`, 2015.

[HBC01] Jean-Pierre Hubaux, Levente Buttyán, and Srdan Capkun. The quest for security in mobile ad hoc networks. In *Proceedings of the 2nd ACM international symposium on Mobile ad hoc networking & computing*, pages 146–155. ACM, 2001.

[JJV08] Jorma Jormakka, Henryka Jormakka, and Janne Väre. A lightweight management system for a military ad hoc network. In *Information Networking. Towards Ubiquitous Networking and Services*, pages 533–543. Springer, 2008.

[JS03] Jangeun Jun and Mihail L Sichitiu. The nominal capacity of wireless mesh networks. *Wireless Communications, IEEE*, 10(5):8–14, 2003.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *Software Engineering, IEEE Transactions on*, (2):125–143, 1977.

[LK01] Hyojun Lim and Chongkwon Kim. Flooding in wireless ad hoc networks. *Computer Communications*, 24(3):353–363, 2001.

[Lon14] Roy Longbottom. Roy longbottom's raspberry pi benchmarks. [on-line] `http://www.roylongbottom.org.uk/Raspberry%20Pi%20Benchmarks.htm`, 2014.

[Per08] Charles E Perkins. *Ad hoc networking*. Addison-Wesley Professional, 2008.

[PM14] R. Pressman and B. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2014.

[Poo00] Robert D Poor. Self-organizing network, February 22 2000. US Patent 6,028,857.

[Ras15] Raspberry Pi Foundation. What is a raspberry pi? [on-line] `http://www.raspberrypi.org/help/what-is-a-raspberry-pi/`, 2015.

[Red15] Redis developers team. Pub/sub redis. [on-line] `http://redis.io/topics/pubsub`, 2015.

[RTB+13] Daniel G Reina, Sergio L Toral, Federico Barrero, Nik Bessis, and Eleana Asimakopoulou. The Role of Ad Hoc Networks in the Internet of Things: A Case Scenario for Smart Environments. In *Internet of Things and Inter-cooperative Computational Technologies for Collective Intelligence*, pages 89–113. Springer, 2013.

[Rub13] Ruby development team. pack (array) - apidock. [on-line] `http://apidock.com/ruby/Array/pack`, 2013.

[SNS03] Keng Siau, Fiona Nah, and Hong Sheng. Values of mobile applications to end-users. *European Research Consortium for Informatics and Mathematics (ERCIM) News*, 54:50–1, 2003.

[Spa13] SparkFun Electronics. Sparkfun xbee explorer dongle - wrl-11697 - sparkfun electronics. [on-line] `https://www.sparkfun.com/products/11697`, 2013.

[SS83] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, 1(3):222–238, 1983.

[Sub06] B. M. Subraya. *Integrated Approach to Web Performance Testing: A Practitioner's Guide*. IGI Global, 2006.

[Tij12] Henk Tijms. *Understanding probability*. Cambridge University Press, 2012.

[UH12] Eben Upton and Gareth Halfacree. *Meet the Raspberry Pi*. John Wiley & Sons, 2012.

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[Wan06] Roy Want. An introduction to rfid technology. *Pervasive Computing, IEEE*, 5(1):25–33, 2006.

[YAM06] YAML.org. Yaml ain't markup language. [on-line] `http://www.yaml.org/refcard.html`, 2006.

[Ye13] W. Ye. *Instant Cucumber BDD How-to*. Packt Publishing, 2013.

Poznań University of Technology
Faculty of Computing Science
Institute of Computing Science

Typeset using LaTeX in Computer Modern.

BibTeX:

```
@mastersthesis{ key,
    author = "Amadeusz Juskowiak \and Tomasz Kuczma \and Mateusz Rybarski \and Maciej Żurad",
    title = "{Design and implementation of communication protocols for self-organizing multi-hop
ad hoc networks using XBeePro-868 platform}",
    school = "Pozna{\'n} University of Technology",
    address = "Pozna{\'n}, Poland",
    year = "2015",
}
```