Final thesis

Implementation of a Manycast Protocol in a Partitionable Mobile Ad hoc Network

by

Gustav Nykvist

LIU-IDA/LITH-EX-A--09/043--SE

2009-09-14
Final Thesis

Implementation of a Manycast Protocol in a Partitionable Mobile Ad hoc Network

by

Gustav Nykvist

LIU-IDALITH-EX-A--09/043--SE

2009-09-14

Supervisor: Mikael Asplund, IDA, Linköpings universitet
Examiner: Simin Nadim-Tehrani, IDA, Linköpings universitet
Implementation of a Manycast Protocol in a Partitionable Mobile Ad hoc Network

Gustav Nykvist

Wireless communication has grown very popular, and communication is the key to success in many situations. However, most of the common technologies today rely on infrastructure and in disaster situations infrastructure might be lost or get severely overloaded. This master thesis concerns intermittently connected mobile ad hoc networks. A network in which the devices may move freely in any direction and still be able to communicate. To be able to demonstrate a network protocol called random-walk gossip-based manycast (RWG) my assignment has been to implement this protocol using off-the-shelf hardware and software.

RWG is a multi-hop and partition-tolerant mobile ad hoc manycast network protocol. Multi-hop refers to information being able to hop between more than two nodes in a network and partition-tolerant means that the protocol works even though a network is partitioned. Manycast means that the information should be successfully delivered to K of all the potential nodes in the area. The RWG protocol makes use of four different packet types, request to forward (REQF), acknowledgement (ACK), ok to forward (OKTF) and be silent (BS). The actual data being sent is carried by REQF, and is referred to as messages. When a message is sent it takes what could be described as a random walk among the nodes in the network, hence the name.

The implementation of the RWG protocol resides in user-space and depends on the IEEE 802.11b standard and the raw socket that is specified in the BSD socket API. It is written in C and was developed on a machine running Ubuntu. It runs on systems that use Linux 2.6 kernels and it supports cross-compiling for ARM based devices such as the Nokia N810 internet tablet and the Android phone. To be able to demonstrate the protocol I developed my own client application. Moreover, an already existing application for Android, Portable Open Search and Identification Tool (POSIT), was successfully extended to run on top of the RWG implementation. The extension was developed by people in the POSIT project and tested in a physical experiment covering five devices.

The report covers the RWG protocol, the system choice, the implementation and the testing of the implementation.
Abstract

Wireless communication has grown very popular, and communication is the key to success in many situations. However, most of the common technologies today rely on infrastructure and in disaster situations infrastructure might be lost or get severely overloaded. This master thesis concerns intermittently connected mobile ad hoc networks. A network in which the devices may move freely in any direction and still be able to communicate. To be able to demonstrate a network protocol called random-walk gossip-based manycast (RWG) my assignment has been to implement this protocol using off-the-shelf hardware and software.

RWG is a multi-hop and partition-tolerant mobile ad hoc manycast network protocol. Multi-hop refers to information being able to hop between more than two nodes in a network and partition-tolerant means that the protocol works even though a network is partitioned. Manycast means that the information should be successfully delivered to K of all the potential nodes in the area. The RWG protocol makes use of four different packet types, request to forward (REQF), acknowledgement (ACK), ok to forward (OKTF) and be silent (BS). The actual data being sent is carried by REQFs, and is referred to as messages. When a message is sent it takes what could be described as a random walk among the nodes in the network, hence the name.

The implementation of the RWG protocol resides in user-space and depends on the IEEE 802.11b standard and the raw socket that is specified in the BSD socket API. It is written in C and was developed on a machine running Ubuntu. It runs on systems that use Linux 2.6 kernels and it supports cross-compiling for ARM based devices such as the Nokia N810 internet tablet and the Android dev phone 1. To be able to demonstrate the protocol I developed my own client application. Moreover, an already existing application for Android, Portable Open Search and Identification Tool (POSIT), was successfully extended to run on top of the RWG implementation. The extension was developed by people in the POSIT project and tested in a physical experiment covering five devices.

The report covers the RWG protocol, the system choice, the implementation and the testing of the implementation.

Sammanfattning

Trådlös kommunikation har blivit väldigt populärt, och kommunikation är nyckeln till framgång i många situationer. Men många av de teknologier som existerar idag bygger på infrastruktur. I en kontrollerad situation gör inte detta något, men i till exempel ett katastrofområde så kan infrastrukturen slås ut eller bli överbelastad.
Detta examensarbete handlar om *mobila spontana nätverk*. Dessa är sådana där noderna kan röra sig fritt i alla riktningar men ändå kommunicera med varandra. För att kunna demonstrera ett nätverksprotokoll som heter *random-walk gossip-based manycast* har min uppgift varit att implementera detta genom att använda hårdvara och mjukvara som går att finna i vanliga butiker.


Rapporten behandlar RWG-protokollet, mitt val av system, implementationen och testningen av implementationen.
Acknowledgments

First of all I would like to thank Simin, Mikael and the people in the POSIT project, it has been a great experience working with you. I would also like to thank my family and friends for the support that I have been given during my time at Linköping University.
Contents

1 Introduction .................................................. 1
   1.1 Background .................................................. 1
   1.2 Aim and Goal ................................................. 1
       1.2.1 Aim ..................................................... 2
       1.2.2 Goals ................................................... 2
       1.2.3 Guidelines .............................................. 2
   1.3 Analysis ..................................................... 3
   1.4 Limitations .................................................. 3
   1.5 Structure of the Report .................................... 3
   1.6 Related Work ................................................. 4

2 Background .................................................... 5
   2.1 Random-Walk Gossip-Based Manycast (RWG) ................. 5
   2.2 User-/Kernel-space ......................................... 7
   2.3 System Choices ............................................. 8
       2.3.1 Operating System ...................................... 8
       2.3.2 Technologies ........................................... 8
       2.3.3 Hardware ............................................... 9
   2.4 Programming Language .................................... 10
   2.5 IEEE 802.11 ................................................. 10
       2.5.1 Architecture and Protocols ......................... 11
       2.5.2 Common Problems ................................... 13
       2.5.3 IEEE 802.11b ......................................... 14
   2.6 Berkeley Sockets .......................................... 15

3 Implementation of RWG ....................................... 17
   3.1 Source Structure .......................................... 17
   3.2 Architecture .............................................. 18
   3.3 Design ...................................................... 18
       3.3.1 Threads ............................................... 18
       3.3.2 Internals ............................................... 19
   3.4 Protocol Interface ....................................... 22
       3.4.1 IPC .................................................... 22
       3.4.2 Startup ................................................ 23
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3 Control Messages</td>
<td>24</td>
</tr>
<tr>
<td>3.5 Handhelds</td>
<td>24</td>
</tr>
<tr>
<td><strong>4 Testing</strong></td>
<td><strong>25</strong></td>
</tr>
<tr>
<td>4.1 Client Applications</td>
<td>25</td>
</tr>
<tr>
<td>4.1.1 Tools</td>
<td>26</td>
</tr>
<tr>
<td>4.1.2 POSIT</td>
<td>26</td>
</tr>
<tr>
<td>4.2 Functional Testing</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1 Multi-hop</td>
<td>27</td>
</tr>
<tr>
<td>4.2.2 Partition Tolerance</td>
<td>29</td>
</tr>
<tr>
<td>4.2.3 Other</td>
<td>30</td>
</tr>
<tr>
<td>4.3 Performance Testing</td>
<td>32</td>
</tr>
<tr>
<td>4.3.1 CPU Load</td>
<td>32</td>
</tr>
<tr>
<td>4.3.2 Network Load</td>
<td>33</td>
</tr>
<tr>
<td>4.3.3 Memory Consumption</td>
<td>38</td>
</tr>
<tr>
<td>4.3.4 Range</td>
<td>38</td>
</tr>
<tr>
<td><strong>5 Discussion</strong></td>
<td><strong>39</strong></td>
</tr>
<tr>
<td><strong>6 Conclusions and Recommendations</strong></td>
<td><strong>43</strong></td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td><strong>45</strong></td>
</tr>
<tr>
<td>A Source Code, Implementation for Linux</td>
<td><strong>47</strong></td>
</tr>
<tr>
<td>B Source Code, Client Application</td>
<td><strong>83</strong></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This master thesis implements research prototypes in the area of mobile ad hoc networks (MANET). It is in partial fulfillment of a degree in Information Technology (30 credit points) carried out at Linköping university. To be able to demonstrate a network protocol called random-walk gossip-based manycast (RWG) my assignment has been to implement the fundamental functionalities of this protocol using off-the-shelf hardware and software. The report covers the RWG protocol, the system choice, the implementation and the testing of the implementation.

1.1 Background

Communication is the key to success in many situations. An example of such a situation is rescue operations, where it is important to be able to pass information among the participants in rescue teams efficiently. Wireless communication has grown very popular, and there exists a lot of different technologies which allows wireless communication. However, many of those technologies, for example GSM and 3G, rely on infrastructure. This is not a problem in a controlled situation, but when a disaster strikes there is a chance that infrastructure is lost or gets severely overloaded. Infrastructure-less communication has not come as far as infrastructure-based technologies, which makes this an interesting field of research.

Many of the devices you can find in stores today, such as laptops, handheld devices and cellphones contain technology which allows them to communicate over the wireless medium. This makes it possible to implement new network protocols that support wireless ad hoc communication and reach larger audience.

1.2 Aim and Goal

This section presents the aim and goal of the project to give an understanding of the purpose for this master thesis. It also covers the guidelines that were followed to achieve the goals.
1.2.1 Aim

The aim of this project is to successfully port the RWG protocol [2] to one or more operating systems. The reason for doing this is to see whether it is possible to run the protocol on laptops and handheld commodity devices, and also be able to demonstrate the protocol in a non-simulated environment. To provide visibility the project also includes extending a client application to work as a showcase for the protocol. The long term aim for the project, which subsumes the master thesis, is to provide greater heterogeneity by implementing the protocol on several different operating systems.

1.2.2 Goals

The following goals were identified, in an order of priority, for the master thesis,

1. Successfully porting the fundamental functionalities of the RWG protocol to laptops running a Linux based operating system.

2. Successfully porting the fundamental functionalities of the RWG protocol to a hand held commodity device (Nokia N810, internet tablet) running a Linux based operating system.

3. If time allows, porting the fundamental functionalities of the RWG protocol to other devices, for example Android phones, to provide some degree of heterogeneity.

4. If time allows, implement or extend a client application to demonstrate the protocol and provide visibility.

With successfully porting I refer to a functionally working implementation of the protocol, hence I do not consider other aspects for example security. With fundamental functionalities I refer to that I did not implement energy conserving functionalities that are specified in the RWG protocol description [2]. To verify that the implementation works according to the protocol different tests were performed.

1.2.3 Guidelines

The protocol was intended to be ported to hand held commodity devices running several different operating systems. However, since the complexity of this task at the start of the project was unknown, it was uncertain whether it was possible to port the protocol to all platforms given the time span of the master thesis. It was also uncertain whether it is possible to port the protocol to proprietary platforms since the implementation might require programming in the kernel space. Therefore the order of implementation was decided as follows, start with porting the protocol to a laptop that is running a Linux kernel. After this the work would focus on implementing the protocol to a hand held commodity device that is running a Linux based operating system. To provide visibility to the implementation the
next task was to demonstrate the work by implementing, or extending an already existing, client application that runs on top of the protocol.

1.3 Analysis

The work started with implementing the protocol to laptops running Linux (Ubuntu 8.10). The advantage with this is that Linux has an open source model, which allows me to implement kernel modules. Another advantage with starting with the implementation to Linux is that there hopefully exists other similar well documented implementations and tutorials. However, since programming in kernel space requires a lot of knowledge I started with examining methods that would allow me to accomplish the implementation in user-space. The idea was that an implementation in user-space, based on well known already existing technologies, could speed up the process and also make it easier to port to the hand held commodity devices. After this phase, when I had managed to implement and ported the protocol to a Linux based commodity device I focused on visibility.

1.4 Limitations

The implementation in this thesis project covers most of the functionalities of the RWG protocol as described in the published paper [2]. However, the mechanism for distinguishing active and inactive packets at each node is not implemented yet. This requires a fairly small adjustment to the implementation.

1.5 Structure of the Report

This section contains an overview of the report and its different sections and appendixes.

- Chapter 2, Background. This chapter addresses the RWG protocol. It will describe the concept and the fundamental functionalities. It will also describe the system I chose for the implementation and why I chose it. Issues concerning some of the technologies that are used and what impact they will have on the RWG implementation will also be addressed. Since some of the technologies mentioned in this section are rather complex there is not room for describing them in detail. The most fundamental parts will described. However since those are common technologies it is not hard to find more detailed information in textbooks or on the web.

- Chapter 3, Implementation. This chapter will give an overview of the architecture of the implementation to give the reader an idea how the implementation works and how to use it. I will also discuss why I chose to implement it as I did.

- Chapter 4, Testing. This chapter concerns the tests that have been performed to verify that the implementation works properly. This section will
also cover tests on CPU load, network load and memory consumption. Besides the tests this chapter also covers the client applications that have been used. One client application developed by me and one client application, POSIT, that was extended to work on top of the RWG implementation.

- Chapter 5, Discussion. This is where the discussion concerning the project is found. The discussion is mainly about the choices I made, the implementation itself and reflections on the RWG protocol.

- Chapter 6, Conclusions and Recommendations. Finally I will present the conclusions and recommendations. The conclusions and recommendations concerns the project in general.

- Appendix A, Source Code, Implementation for Linux. This is where the RWG implementation source code is found.

- Appendix B, Source Code, Application Client. This is where the source code for a client application is found.

1.6 Related Work

An interesting project that is similar to this one is picoNet II, a wireless ad hoc network for mobile handheld devices. That project was performed by Alex Song, and consisted of a kernel-space implementation of the dynamic source routing protocol (DSR) for TCP/IP on Linux, for both a PC and a handheld device, Compaq iPAQ. His implementation enabled the PC and the iPAQs to form a multi-hop ad hoc network with the help of 802.11b network interface cards. He also implemented a DSR to IP gateway which allowed the nodes in the ad hoc network to access external IP networks. The resulting implementation allowed existing unmodified TCP/IP applications to run on the picoNet II network [13].

Another interesting implementation of a MANET protocol is AODV-UU. This is an implementation of the ad hoc on-demand distance vector routing protocol (AODV), which was initially performed by Uppsala University, hence the postfix UU. It runs on Linux 2.4 and 2.6 kernels and in the ns-2 simulator and it also supports crosscompiling for ARM/Mips based devices such as the iPAQ. The protocol is implemented as an user-space daemon with kernel a component [10].

To the best of my knowledge there are no implementations of delay-tolerant protocols running on handheld devices for intermittently connected networks with frequent partitions.
Chapter 2

Background

In this chapter the RWG protocol is presented. It will also describe the system I chose for the implementation and why I chose it. To give the reader a greater technical understanding the most essential technologies that the RWG implementation relies on will also be addressed.

2.1 Random-Walk Gossip-Based Manycast (RWG)

RWG is a manycast protocol for intermittently connected MANETs. In this section the protocol will be explained to give the reader an understanding of how it works, first the different packet types will be introduced and after that the header fields will be explained. The underlying algorithms for the protocol will be discussed briefly. For a more in depth explanation I recommend reading the RWG protocol description [2]. As I mentioned in the previous chapter, the RWG implementation is partial and do not concern the energy conserving functionalities that are specified in the protocol description.

The four types of packets that are used by the RWG protocol are request to forward (REQF), acknowledgement (ACK), ok to forward (OKTF) and be silent (BS).

- request to forward, the REQF is the packet that carries the actual payload (the data) that is being sent over the network. In the report payload will be referred to as message. When a node sends a message it will be sent within a REQF to all neighboring nodes. The sender has no knowledge about nodes in its vicinity.

- acknowledgement, the ACK is used as a response packet. When a node receives a new message, carried by a REQF, it will respond with an ACK. This lets the sender of the message know which nodes have received it.

- ok to forward, the OKTF is used when one node wants to inform another node that it should take on the role of forwarder, custodian, for a message.
• be silent, the BS is used to silent nodes that are sending messages that should not be sent. RWG is a manycast protocol and the messages are supposed to reach a certain number of nodes. When a message has reached this number of nodes it is not supposed to be sent further.

<table>
<thead>
<tr>
<th>packetLength</th>
<th>type</th>
<th>hops</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sequenceNumber</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| origin | | |
| target | | |
| sender | | |
| timeToLive | | |
| informed | | |
| toAvoid | | |

Figure 2.1. The RWG header.

To get some more insight in how the protocol works it is essential to know about the fields in the header, the header is illustrated in figure 2.1. The first field, packetLength, is a two byte field that contains the total length of the packet. Field two, type, represents the type of the packet. The hops field contains the number of hops a packet has performed, that is how many times a message has been forwarded. The groupSize field contains the number of nodes a message should be delivered to. The following fields, origin, sender and target contains information that refers to which node that originally sent a packet, forwarded a packet and which node a packet is addressed to. However, here the protocol description [2] differs from the implementation, since it says that the fields should be four bytes long but in the implementation they are six bytes long. The reason for this is because I found it easy to use media access control (MAC) addresses since they are unique, and a MAC address is six bytes long.

To be able to tell how many and which nodes that a packet has reached the two fields informed and toAvoid are used. Those are bit-vectors and it is preferable that they are as long as the estimated number of nodes in the network, since the slots in the vectors represent the nodes in the network. A node knows which slot that represent it by hashing its MAC address. The algorithm works even when collisions in the bit-vectors occur.

Figure 2.2 illustrates the most fundamental part of the RWG algorithm. First a REQF is sent to all the neighboring nodes. The nodes that receive the REQF answer with an ACK. The node that originally sent the REQF waits for a short period of time and then choses one of the received ACKs at random and sends an OKTF addressed to the sender of that ACK. The node that receives the OKTF will send the REQF again. Just like the figure illustrates, a node will not respond with an ACK to the same REQF more than once. So the main idea behind RWG is that messages take random walks among the nodes in the MANET. With the help of the bit-vector informed, carried within the packet header, the nodes in a network can conclude whether a message has reached them before. The bit-vector
Figure 2.2. Example of four nodes communicating.

to avoid is used when a REQF reaches a certain hop limit, and in short, it allows for messages to sometimes visit nodes which they already have visited. For more information about this I recommend reading the protocol description [2].

Figure 2.3. 1) Station one sends a message, m1, which is received by station two. 2) Station two checks if it carries any messages that station one do not possess, m3 and m4, and sends them.

Another interesting functionality the RWG protocol has is wake. This functionality works as follows. If a node receives a message, lets call it message A, it will lookup whether it has messages that the sender of packet A do not have. This is possible thanks to the informed bit-vector. If the node finds some messages that the sender of message A do not have, it will start transmitting those messages, this is illustrated in figure 2.3. Since MANETs can be partitioned, this functionality is of great use.

2.2 User-/Kernel-space

One decision I had to make early in the project was whether to implement the protocol in kernel-space or user-space. Before making this choice I considered
some of the consequences the different approaches would have. The decision affects portability, CPU load, power efficiency and complexity.

Portability would be affected because if the implementation is written in kernel-space it might be hard to port the protocol to other platforms that use another type of kernel. First of all because another kernel might have another design and also because the kernel might be closed. If the kernel is closed you will not have access to the source code, and therefore implementing the protocol will not be possible. However, if you pursue an implementation in user-space there is a greater chance that you will be able to port the protocol. At least if you build the protocol on existing and common technologies.

It seems reasonable that the decision would affect the CPU load, because if the protocol resides in user-space all the information must be passed up from the kernel. From this follows a perhaps bigger concern, the CPU load will have a direct impact on power efficiency. Since the protocol is intended to run on mobile devices such as laptops and handhelds with limited battery time, good power efficiency is preferable.

By complexity I am referring to the complexity of doing the actual implementation. Implementing a protocol in kernel-space is more complicated than in user-space, and therefore more time consuming. For example, if the implementation handles memory in an incorrect manner the consequences will be more severe if the implementation resides in the kernel. The reason for this is that in the kernel you have direct access to the memory, and handling memory incorrectly might lead to such a severe state that the operating system crashes. However, misusing memory in user-space will most often only lead to crashing the process that is running the implementation.

Finally, I chose to perform the implementation in user-space because I considered portability to be of great importance.

2.3 System Choices

Before starting with the actual implementation of the protocol I had to consider different system related choices. This section will cover the different choices I made and the consequences that the different choices have on the implementation.

2.3.1 Operating System

I chose to implement the protocol for Ubuntu 8.10 (Intrepid Ibex) an easy to use operating system that is running a Linux kernel. The main reason why I chose an operating system running a Linux kernel is because the handhelds, Nokia N810 and Android dev phone 1, also use Linux kernels. Using a Linux kernel also leaves open the possibility of a future porting of the protocol to kernel space.

2.3.2 Technologies

A wireless network depends on different and rather complex technologies. For example firmware and drivers for the hardware, and it would be unrealistic to
implement everything from scratch. Therefore the implementation must depend on already existing technologies. The most common family of standards for wireless local area networks (WLANs) today is IEEE 802.11. I chose to use this family of standards because it makes the implementation easier to port, compared to using a less common standard. This would also let me use a greater range of off-the-shelf hardware and software. However, the implementation can suffer from inherited issues. For example, 802.11 is far more developed for infrastructure-based WLANs, since those are more common and less complex. Also, drivers might deviate from the standard and not support infrastructure-less mode at all, which is necessary for the RWG implementation.

To be able to create and send my own types of packets over the wireless network I decided to use BSD sockets, a C-library for *inter-process communication* (IPC). IPC refers to sending information between processes and threads, even processes that run on different machines [6].

### 2.3.3 Hardware

The handhelds that have been used in the project are the Nokia N810 and the Android dev phone 1.

![Figure 2.4. Nokia N810 to the left and the Android dev phone 1 to the right.](image)

Besides the handhelds I also had access to three laptops, two IBMs and one Asus netbook. However, I chose not to use the network interface cards that came with the IBMs. Instead I used a USB dongle, *Linksys WUSB54GC*, and a *Netgear PCI adapter* which can be seen in figure 2.5. I chose those two because I knew that there exists well developed Linux drivers for them and also because I have some experience working with them.
2.4 Programming Language

Before I could start implementing the protocol I had to choose a programming language to work in. The language I chose was C and I did not really consider any other languages, even though it is possible to perform an implementation as such in other languages. The reason why I chose C was mainly because I thought it would be easier to port the protocol into the kernel, so I had that choice open. Another reason why I chose C is because I have done some network programming in it before.

2.5 IEEE 802.11

IEEE 802.11 is a family of standards for WLAN computer communication. It uses the 2.4, 3.6 and 5 GHz frequency bands. The family contains several different standards such as 802.11 (legacy), 802.11a, 802.11b, 802.11g and 802.11n. The standards are developed by the 802.11 Working Group under the auspices of the IEEE Project 802 LAN/MAN Standards Committee[1]. The IEEE 802.11 defines two different operational modes, infrastructure and infrastructureless. And at the moment IEEE 802.11 is the most common technology for infrastructure-based WLANs. However, the RWG implementation will rely upon the infrastructureless mode, which is not as mature as the infrastructure mode. Infrastructure-based WLANs uses access points, that for example helps the hosts that are connected to the WLAN with routing and clock synchronization. Infrastructureless WLANs do not use access points, so everything must be managed by the hosts them selves [3]. This approach is more complicated and complexity is one of the reasons why infrastructureless WLANs are less common. A basic service set (BSS) is the set of all hosts that can communicate with each other, and they are the building blocks of an IEEE 802.11 WLAN. The infrastructureless mode is often referred
to independent basic service set (IBSS) and requires a minimum of two hosts to work [12]. Figure 2.6 shows the differences between the two different operational modes.

Figure 2.6. A simple illustration of an infrastructure-based WLAN to the left, and an infrastructureless WLAN to the right.

IEEE 802.11 supports single-hop ad hoc networks, which requires all the nodes in the network to be within the same BSS. All nodes must be within each other’s transmission radius. When it comes to multi-hop ad hoc networks it is not as simple, a protocol such as RWG must exist so nodes that are not within the same BSS can communicate. The following subsections will address interesting parts of the IEEE 802.11 family of standards.

2.5.1 Architecture and Protocols

The IEEE 802.11 standards specify both the physical layer and the medium access control (MAC) layer. This section will focus on the MAC layer and will only concern the architectural solutions and protocols that will have substantial impact on the performance or behavior of the RWG implementation. IEEE 802.11 defines several functionalities for the MAC layer, the distributed coordination function (DCF), the point coordination function (PCF) and the hybrid coordination function (HCF). The functionality that is most interesting from an ad hoc perspective is the DCF. The PCF is an optional access method that is useful only in infrastructure networks and HCF is only useful in a quality of service (QoS) configuration [12].

Distributed Coordination Function

The IEEE 802.11 MAC layer uses a DCF called carrier sense multiple access with collision avoidance (CSMA/CA) as medium access method. It is implemented in all stations for use in both infrastructure-based and infrastructureless network configurations [12]. CSMA/CA was inspired by the success of Ethernet and its random access protocol, carrier sense multiple access with collision detection (CSMA/CD). CSMA, carrier sense multiple access, refers to that the stations must listen to the channel before sending on it, and if the channel is busy the station
will refrain from sending (to avoid collisions). However, there are differences between CSMA/CA and CSMA/CD. Since this section only concerns IEEE 802.11, CSMA/CD will not be explained. The main reason why the 802.11 MAC protocol does not implement CSMA/CD is because it is hard to detect collisions in a wireless network while it is rather easy in a wired network. Therefore the 802.11 MAC protocol instead pursues collision avoidance, that is avoiding collisions instead of detecting them.

Before discussing collision avoidance it is necessary to understand the 802.11 link-layer acknowledgement scheme. To deal with frames that do not reach their destination the 802.11 MAC protocol uses link-layer acknowledgements (ACKs). When a station receives a frame, it waits a short period of time and then responds with an ACK. This short period of time is called short inter-frame spacing (SIFS). If a station does not receive an ACK after it has sent a frame (within a specified period of time) it assumes that an error has occurred and retransmits the frame, using the CSMA/CA to get access to the channel. If the transmitting station does not receive an ACK within a specified number of retransmissions it gives up and discards the frame.

With some insight in link-layer acknowledgements, let’s continue with the 802.11 CSMA/CA protocol. Consider a wireless station that wants to transmit a frame. The transmitting station first checks if the channel on which it will transmit the frame is free. If it is free the station first waits a short period of time known as the distributed inter-frame space (DIFS) and then transmits the frame. However, if the channel is busy the station generates a random backoff value and counts down while the channel is free and while the channel is busy the backoff value is frozen. When the counter reaches zero the station transmits the whole frame and waits for an ACK. If the transmitting station receives an ACK it knows that the frame has been received by the receiving station correctly. However, if no ACK is received the station will continue with the same algorithm at the stage where it generates a random backoff value [7].

Time Synchronization Function

All stations in a basic service set (BSS) need to be synchronized to a common clock. The time synchronization function (TSF) helps the stations to maintain and synchronize a local TSF timer. When a WLAN is running an infrastructure configuration the access point will be the timing master. However, when running in an infrastructureless configuration all the nodes share the burden of keeping their clocks synchronized. This is done by a distributed algorithm that needs to be implemented on all the members of the IBSS. The stations in the WLAN send beacon frames and probe responses according to the distributed algorithm. All stations needs to adopt the TSF value in any received beacon frame or probe response if the value is greater than its own TSF timer and is from a station belonging to the same IBSS [12]. To get a more in depth understanding of how the TSF timer and the distributed algorithm works I recommend reading the IEEE 802.11 standard.
Active and Passive Scanning

There exists two different ways for a node to join (or create) an IBSS, these two will be explained briefly. First the station scans the wireless medium while the station receiver tunes in to different frequencies searching for special control frames. If no such control frames are found the station creates a new IBSS. The two different methods of scanning the wireless medium are passive scanning and active scanning. When a station performs passive scanning it just taps in to the wireless medium and listens for beacon frames. Beacon frames carry a complete set of IBSS parameters. This makes it possible for a station to join an already existing network without sending any information. Active scanning accesses the wireless medium with the DCF method, sends probe frames and listens for probe responses from other stations [3].

2.5.2 Common Problems

There are problems related to CSMA/CA, one of which is called the hidden terminal problem. Consider three stations in a wireless network, where only one of them is within the same transmission radius as the other two. This leads to two stations that cannot hear whether the channel is idle or not, since they can not hear the traffic from the terminal that is not within range. The algorithm in the CSMA/CA is build on the assumption that all nodes in a wireless network are within each other’s transmission radius, so the collision avoidance will not work properly in a WLAN where hidden terminal exists since collisions will occur [7].

Figure 2.7. Illustrates the hidden terminal problem.

To deal with the hidden terminal problem the IEEE 802.11 standard also defines a reservation scheme which helps to avoid collisions in networks where hidden terminals exists. However, this reservation scheme is optional and most commercial 802.11b cards have this feature turned off as default [11]. The idea behind this scheme is rather simple, it makes use of two kinds of control frames called ready
to send (RTS) and clear to send (CTS). When a station wants to transmit some data it first checks if the channel is free, and if it is, it waits one DIFS then sends a RTS. The station that receives the RTS waits one SIFS and than replies with a CTS. When the transmitting station receives the CTS it waits one SIFS and then sends the data, which is followed by an ACK from the receiving station if no error occurs. RTS and CTS frames can collide, but they are short and a collision will only last for the duration of the those frames. This is far better than having data packets colliding. You might wonder why this feature is optional, the answer to that is because most WLAN setups are rather simple and do not usually contain hidden terminals, and then RTS and CTS frames are only unnecessary overhead [7].

Exposed terminal is another common problem that might occur in WLANs, it is illustrated in figure 2.8. Assume there exists four nodes A, B, C and D. Assume further that node B is within the same transmission radius as A and that B is transmitting to A. However, B is also within the same transmission radius as C, but C is not within the same transmission radius as A. Finally, D is only within the same transmission radius as C. Since B is transmitting to A, it will not be possible for C to transmit to D at the same time even though collisions would not occur. This is due to the fact that the distributed coordination function, CSMA/CA, specifies that the node first has check that the medium is free before transmitting [3]. However, I do not believe that this problem has much impact on the performance of the RGW implementation. What could happen is that nodes sometimes refrain from sending, which might lead to some delay.

Figure 2.8. Illustrates the exposed terminal problem.

2.5.3 IEEE 802.11b

The actual IEEE 802.11 protocol that has been used during the implementation is 802.11b, which extends 802.11. The reason for this is rather simple, all the different NICs that I had access to supported 802.11b. This standard enables
transmissions at 5.5 Mbps and 11 Mbps in the 2.4 GHz frequency band. However, to support interoperability with 802.11, control frames and frames with multicast or broadcast destinations have to be transmitted at a rate belonging to the basic rate set. This set contains the data transfer rates that all stations must be able to use for receiving and transmitting [3].

### 2.6 Berkeley Sockets

Berkeley sockets is an application programming interface (API) for inter-process communication (IPC), usually used for computer networks. It was developed at the University of California, Berkely. Currently there exists different kinds of sockets that can be created with the help of the BSD Socket API, stream socket, datagram socket, sequenced packet socket and raw socket.

Stream sockets provide a reliable, bidirectional, sequenced and unduplicated data flow without record boundaries (the number of bytes written by the sender does not necessarily need to be the same as the number read by the receiver). The datagram socket provides a bidirectional data flow, but do not promise reliable, sequenced or unduplicated flow. The sequenced packet socket is like a stream socket, with the exception that record boundaries are preserved. The socket type that is most interesting for this master thesis is the raw socket. By using this it is possible to access the underlying layers in the Open Systems Interconnection (OSI) model. The raw socket type is not meant for general application writers, but for people interested in developing new kinds of communication protocols [8].
Chapter 3

Implementation of RWG

In this section the implementation of the RWG protocol will be described. The implementation will not be covered in great detail, but the concepts that have been used will be described to give the reader an idea about how the implementation works. However, for the curious reader the source code can be found in Appendix A.

3.1 Source Structure

Before describing some of the design concepts the structure of the source code will be presented. Since there are quite a lot of different functions the source code has been arranged in an order of functionality. The source code consists of the following files.

- **rwg.c and rwg.h**: Here you will find the main function, which is responsible for the startup of the protocol. Also functions for handling the threads which the protocol relies on are found here. In short, this is the part of the source where the main loops reside.

- **rwg_receives.c and rwg_receives.h**: In this part of the source functions concerned with handling incoming data, both from the network and processes on the same machine, are found. Also all the functions that perform bit-vector operations have been placed here.

- **rwg_send.c and rwg_send.h**: This is where you will find all functions concerned with sending data and creating REQF, OKTF, ACK and BS packets.

- **rwg_header.c and rwg_header.h**: In these files there exist functionalities for creating fundamental parts of the packets, such as the ethernet header. This is also where the definition of the RWG header data structure is placed. Another interesting data structure that is defined here is the packetBuffer, a structure for handling the most vital buffers.
• `rwg_socketssc` and `rwg_socketssh`. Finally, the functionalities found in these files concerns creating raw sockets and binding them to network interfaces.

### 3.2 Architecture

As mentioned in previous sections the implementation of the RWG protocol resides in user-space. The implementation is written in the common programming language C. It relies on the BSD socket API to provide the mechanism which allows it to forge its own packets and access the underlying layers to transmit them on the physical medium. Named pipes are used for inter-process communication on the same machine. Those work as an interface for the protocol so that other applications can use it, which is illustrated in figure 3.1. The standard that is used for accessing the wireless medium is 802.11b.

![Diagram of RWG Architecture](image)

**Figure 3.1.** An overview of where in the system the RWG implementation resides and how it communicates with other processes and the network.

### 3.3 Design

To understand how the implementation works some interesting design concepts will be presented.

#### 3.3.1 Threads

The implementation uses two active threads, one for sending packets and one for handling incoming packets. To synchronize the threads mutex locks are used, which is essential since the threads share resources. The thread that is handling
incoming packets is also responsible for the inter-process communication on the machine. There exists two named pipes, input and output. The input pipe is a fifo, which is used for providing the protocol with data. When the protocol finds new data on the input pipe it will create a packet carrying the data and pushes it down to the lower layers in the network stack, which will result in the packet getting sent over the physical medium. The output pipe, also a fifo, is used for passing data that has arrived over the network to other processes. The listening thread runs until there is something new on either the input pipe or the socket, and if there is, it will handle the incoming data. There will be for example checks of the packet type, if it already has the packet and so forth. After this, and if there is a packet that should be sent as a response to the incoming data, the listening thread will release the lock and inform the sending thread that it has to send a new packet. After the sending thread has sent the packet it will release the lock and standby till the listening thread informs it that there is something new that should be sent. Since threads share the same memory it is not hard to create a way of communication between them, the implementation use pointers to shared memory locations and flags. To get an idea of how this works figure 3.2 represents the design in the rough.

![Diagram of sending and listening threads](image)

**Figure 3.2.** A rough illustration of the sending thread and receiving thread.

### 3.3.2 Internals

Another interesting and vital part of the implementation is the buffer handling. This is probably the part that is the most complex to understand while reading the source code. First the reason for the buffers will be covered, then we will look at
the buffers designs. There exists four different main buffers for handling packets.

- The REQF buffer, has the assignment to hold pointers to all the REQF packets that have arrived and been sent. A REQF packet is only removed from the REQF buffer when its time to live expires or if it has visited enough nodes according to its group size. It is also vital that there does not exist any duplicates of pointers to the same REQF, since they are stored on the heap. This would of course lead to memory issues. The REQF buffer also contains other information about the REQFs that it carries, for example time stamps.

- The Waiting buffer, is responsible for the REQFs that are waiting for ACKs. It carries pointers to the elements in the REQF buffer. A REQF is never freed from the heap from the Waiting buffer. Time stamps are used to make sure that REQFs wait a certain amount of time for ACKs before responding with OKTFs.

- The ACK buffer, takes care of all the incoming ACKs. The buffer is necessary since a transmitted REQF might get many ACKs in response and because more than one REQF can be waiting for ACKs. The packets are stored on the heap and the ACK buffer contains the pointers to the packets. Whenever an ACK should be freed from the heap, it will be from the pointers in the ACK buffer.

- The Wake buffer, makes sure that packets that have woken on an incoming REQF get forwarded. The Wake buffer carries pointers to elements in the REQF buffer.

The designs of the buffers differs quite a lot. To get an understanding how they actually work they will be covered one by one. Figure 3.3 illustrates how the different buffers work.

The REQF buffer has an indicator which points to the next free position in the buffer. So whenever a new REQF arrives or when a new REQF is sent, the packet will be stored at the next free position. When the packet has been added the indicator is incremented, it will point to the next free position. When the indicator points to a position that precedes the size of the buffer new packets will not be added. Instead a forced check of all the REQFs, in the buffer, time to live and group size will be performed. Packets with expired TTL and packets that have reached enough nodes will be removed. However, this check of all packets is also performed at a given time interval. Since order in the buffer is necessary, because of possible fragmentation, the buffer will be sorted every time this check is performed. The sorting only makes sure that all the REQF pointers are stored on the left side of the indicator, and that the other positions are free.

The Waiting buffer works in a quite different manner compared to the REQF buffer. First of all it is a circular buffer, which means that the indicator will be modulo the size of the buffer and the rest will be the actual position in the buffer. The elements that are stored in the waiting buffer are pointers to the elements in the REQF buffer. Instead of having just one buffer indicator it has two. One indicator, the tail indicator, points to the oldest but yet valid packet. And one
Figure 3.3. Illustrates how the different buffers work, the arrows represents the buffer indicators. The REQF buffer contains pointers to the REQFs that are stored on the heap. The Waiting and Wake buffers contain pointers to the elements in the REQF buffer, and the ACK buffer contains pointers to the ACKs that are stored on the heap.

The ACK buffers design is rather straight forward. It is a circular buffer where the indicator points to the next free position in the buffer. This solution is rather practical since you always know that the oldest valid element in the buffer is the one to which the tail indicator points. So it is only necessary to check one time stamp at the time, instead of having to check all the time stamps of the messages that are waiting for ACKs. However, if the front indicator gets incremented till it points to the same position in the buffer as the tail indicator, then the tail indicator will be incremented. This case is not likely to occur, but if it does it might result in packets not being sent properly.

The Wake buffer design is rather similar to the design of the ACK buffer. It is a non-circular buffer with an indicator to the next free position in the buffer. The elements stored in the buffer are pointers to elements in the REQF buffer. When a REQF has been freed from the heap, its pointer is set to NULL, and before accessing a packet from the Wake buffer the pointer value is checked so it is not NULL. The indicator is decremented after a REQF has been sent from the
Wake buffer.

![Diagram of buffer relationship]

Figure 3.4. Shows the relationship between the different buffers.

Since it can be rather hard to understand how the different buffers relate to each other, figure 3.4 shows a simplified illustration of the relation concept. The Wake buffer contains pointers to the elements in the REQF buffer, which in turn contains pointers to REQFs that are stored on the heap. The Waiting buffer also contains pointers to the elements in the REQF buffer. The ACK buffer is independent from the other buffers, and carries pointers to ACKs that are stored on the heap.

### 3.4 Protocol Interface

To be able to use the protocol it is vital to know about the interfaces. This section covers how to start the protocol and the different flags that can be set during startup. It will also cover the control messages and how to communicate with the protocol over the named pipes.

#### 3.4.1 IPC

As mentioned before, to be able to pass messages to the process running the protocol named pipes are used. The two existing pipes are called input and output. Data that is written on the input pipe (by another process on the same machine) will be read by the running protocol. The maximum transmission unit is 1024 bytes which includes the RWG header, so the maximum size of the payload is 964 bytes. However, if the data on the input pipe exceeds 964 bytes, the protocol will fragment the message and send it as several messages. Then the fragments needs
to be reassembled to recreate the original message, however this is not supported by the RWG algorithm. When the protocol reads from the input pipe it will try to read 964 bytes. This might lead to the protocol sending more than one message in a REQF packet. To avoid this it might be a good idea to check that the input pipe is empty before writing to it. However, this is only necessary when sending at rather high rates. The protocol puts all new incoming messages on the output pipe, in order to pass them to other processes. Messages that are written on the output pipe by the protocol always end with a special character, which makes it easier for other processes to separate messages when several have been written to the output pipe.

The code snippet below is an example in Java of how a client application reads from the named pipe output. In this code snippet the messages are read one char at the time from the input-stream. When the special character is reached the loop breaks and the new message is handled.

```java
/*Java, read from named pipe example*/
if (br.ready()){
    incoming = '';
    while (br.read(buff,0,1) > 0){
        if (buff[0] == '
'){
            break;
        }
        incoming = incoming + Character.toString(buff[0]);
    }
    messHandler.post(messUpdate);
}
```

This code snippet is also a Java example, it shows how a client application writes to the named pipe input. It is very straight forward, the messages are just written on to the output-stream.

```java
/*Java, write on named pipe example*/
... try{
    bw.write(newText);
    bw.flush();
} catch (Exception e){
    System.out.println(e.getClass().toString());
}
... 
```

3.4.2 Startup

The existing flags that can be attached during startup of the protocol are i, l, g, h, t, w, r and o. The flags have purposes such as starting the RWG implementation in different modes and setting different values.

- i is used for informing the protocol of which network interface that should be used e.g. wlan0, ath0, ra0.
- l is used for setting the TTL value, an integer value with a max of 3600 (seconds).
• $g$ is used for setting the group size value, an integer value with a max of 99.

• $h$ is used for setting the hop limit, an integer value with a max of 99.

• $t$ is for setting the protocol to trace mode, printing interesting information to the terminal.

• $w$ is for setting the protocol to write output mode. When it is set the sender of incoming REQFs will be written on the output pipe.

• $r$ is for setting the BS response mode. The protocol will answer with a BS packet instead of an ACK if an incoming message contains an inconsistent informed bit-vector.

• $o$ is for setting the time period a node waits between sending a REQF and an OKTF.

### 3.4.3 Control Messages

The control messages that exist are the following: -c-q, -c-c, -c-l, -c-g, -c-h, -c-t, -c-w, -c-r, -c-o, and those are used for changing different values while the protocol is running and also for ending the process. These messages are passed to the protocol through the input pipe.

• -c-q is used for ending the protocol process.

• -c-c is used for removing all messages from the REQF buffer.

• -c-l<integer value> is used for changing the TTL value.

• -c-g<integer value> is used for changing the group size value.

• -c-h<integer value> is used for changing the hop limit.

• -c-t is for turning trace mode on/off.

• -c-w is for turning the write output mode on/off.

• -c-r is for turning the response mode on/off.

• -c-o<integer value> is for setting a new time period as the interval a node waits between sending a REQF and an OKTF.

### 3.5 Handhelds

The implementation runs on both the Nokia N810 and the Android dev phone 1, there was no need to edit the original source code. Since the handhelds use ARM architecture cross-compiling was necessary. The cross-compilers that were used in this project are Scratchbox [9] and CodeSourcery [4]. The cross-compilers generate executable code that run on the two ARM-based devices.
Chapter 4

Testing

The testing that has been performed during the thesis work has been rather limited and mainly focused on testing functionality. One reason for this is because the purpose of the implementation was to provide a proof of concept. Another reason is because I only had access to so few nodes. However, some tests have been performed concerning CPU load, network load, memory consumption and range.

![Illustration of multi-hop packet flow](image)

**Figure 4.1.** Illustrates the expected multi-hop packet flow when station one sends a REQF in a network setup with three nodes.

4.1 Client Applications

Before looking at some of the tests that have been performed I would like to mention that I also developed some tools to ease the testing. Besides the tools that I developed, an application called POSIT was extended to be able to run on top of the RWG implementation. This application was developed by a team at Trinity college, Hartford, USA, and formed a basis of collaboration on free software
for disaster response.

### 4.1.1 Tools

First of all I wrote a program in Java which could be described as a chat client, the source code is found in Appendix B. This program allows the user to send messages over the network and read messages sent by other nodes. It also appends timestamps to the messages and allows the user to append a node number, just to make it easier to see the originator of the message and when it was sent. I also wrote a similar chat client for the Android dev phone 1. However, the chat client for the Android dev phone 1 does not time-stamp messages and the user does not have the possibility to choose a node number. The reason for this was lack of time.

Besides the chat client I also wrote a small program that generates unique messages and passes them to the RWG implementation. This was very helpful since it made it possible for me to walk around among the nodes and monitor the traces without having to generate the traffic myself.

### 4.1.2 POSIT

The RWG implementation was not only tested with the help of client applications, or tools, that I had implemented myself. It was also tested on Google Android phones running an application called Portable Open Search and Identification Tool (POSIT) at Trinity College. This application is produced within the Humanitarian Free and Open Source Software project (HFOSS), and was founded by Prasanna Gautam, Ralph Morelli and Trishan de Lanerolle. The main idea behind POSIT is being able to record information about finds and transmit this information to a central server. Finds can be different things, one example is survivors and victims in the aftermath of a natural disaster [5]. For the purpose of this collaborative testing the Trinity team implemented a distributed version of POSIT.

The programming necessary to get POSIT to work on top of the RWG implementation was performed by the developers in the POSIT project. Since the RWG implementation had not been documented at the time this collaboration started my assistance was necessary to provide the developers with vital information about the implementation. During this collaboration I also added some extra functionalities to the RWG implementation to ease the tests that were planned. For example appending the media access control (MAC) address of the last sender to the message.

This collaboration was interesting because RWG and POSIT have a similar purpose. Both the protocol and the client application are useful in disaster areas. It was also very interesting since it proved that the RWG implementation could be used with other client applications besides the ones I wrote myself. It also showed that the interface is rather easy to understand, since the programming to extend POSIT to work on top of the RWG implementation was not performed by me and also because it did not take a lot of time.

One of the field experiments that was performed by the POSIT developers at Trinity College contained five nodes. The idea behind this experiment was
that the nodes should report finds with help of the RWG implementation. Every time a node came across a find, it was reported. There also existed different communication points, places where the likelihood of meeting other nodes was higher than in the rest of the experiment area. Every node visited a communication point at least once during the experiment. At the end of the experiment all the nodes gathered at an end point. During this test a total of 13 finds were reported and all were successfully delivered to at least K of the nodes in the area.

4.2 Functional Testing

It is essential that the protocol behaves as it is expected to do, to verify the correctness of the behavior functionality tests were performed. First I will address tests concerning two main functionalities, multi-hop and partition tolerance. Following that, some other tests concerning for example TTL will be covered.

Figure 4.2. Illustrates four different network topologies, where the black dots are nodes and the gray circles represents the transmission range.

4.2.1 Multi-hop

To verify that the protocol could perform multi-hop I ran the protocol in different network topologies. In figure 4.2 some of the network topologies are illustrated. It is rather easy to verify that multi-hopping is performed, but it is a bit more tricky to verify that it is performed according to the protocol. The tools that I wrote made it possible for me to perform the tests. However, some network topologies
were rather hard to achieve, since the different nodes have different transmission ranges. For example setup 1 in figure 4.2 that was used for checking that the packets performed several hops. The expected packet flow in a network setup with three nodes can been seen in figure 4.1.

To give the reader some more insight in the multi-hop testing two tests and the results will be presented. The test to verify that packets could perform several hops was executed accordingly. Four nodes were used and positioned as illustrated in setup 1 in figure 4.2. In the first test I used the following protocol setup $hops = 7$, $groupSize = 20$ $TTL = 1$. The reason why I chose a very short $TTL$ was because I did not want the wake functionality to trigger. I wanted to make sure that packets could disperse through the network without the help of the wake functionality. After all nodes had been put in position, one of the end nodes ran a program which sent 50 messages with a 7 seconds interval between each message. The expected behavior for this test is that most of the messages sent by one of the two end nodes would perform three hops and than reach the other end node. The long time interval gave me a chance to study the traffic while the protocol ran and also move nodes if I noticed that the messages jumped through the network in an incorrect manner. By jumping in an incorrect manner I refer to messages that reach the end node but did not disperse through the network according to setup 1 in figure 4.2. I did this test three times, with the following results.

- **Multi-hop Test 1, round 1**, 36 of 50 messages reached the end node. Of the 36 messages that reached the end node 7 jumped through the network in an incorrect manner.

- **Multi-hop Test 1, round 2**, 44 of 50 messages reached the end node. Of the 44 messages that reached the end node 1 jumped through the network in an incorrect manner.

- **Multi-hop Test 1, round 3**, 48 of 50 messages reached the end node. Of the 48 messages that reached the end node 1 jumped through the network in an incorrect manner.

In the second test I used the following protocol setup $hops = 7$, $groupSize = 20$ $TTL = 100$. By using a larger $TTL$ value the wake functionality would trigger and retransmit missed REQFs. After all nodes had been put in position, one of the end nodes ran a program which sent 50 messages with a 1 second interval between each message. The reason why I chose a shorter time interval in the second test was mainly because I was less concerned about packets jumping in an incorrect manner and therefore I did not need to monitor the nodes as closely. The expected behavior for this test is that more messages should reach the end node than in Test 1, due to the fact that wake functionality will retransmit missed REQFs.

- **Multi-hop Test 2, round 1**, 50 of 50 messages reached the end node. Of the 50 messages that reached the end node all had jumped through the network in a correct manner.
- **Multi-hop Test 2, round 2**, 50 of 50 messages reached the end node. Of the 50 messages that reached the end node all had jumped through the network in a correct manner.

- **Multi-hop Test 2, round 3**, 50 of 50 messages reached the end node. Of the 50 messages that reached the end node all had jumped through the network in a correct manner.

While running the first round in Test 1 I noticed that one of the nodes, an Android dev phone 1 which was positioned in between the end nodes, did not seem to pass the incoming REQFs to the next node in the chain. By studying this a bit closer I noticed that the node did not manage to respond to all incoming REQFs with ACKs fast enough when it was in sleep mode (to save energy). In round 2 and round 3 this node was never allowed to sleep, which generated better results. I also managed to reduce the amount of messages that jumped through the network in an incorrect manner with the help of aluminum foil. I wrapped it around nodes to reduce the transmission radius. This was not as easy as one might think, because too much foil results in a greater chance of losing packets. Before discussing the results from Test 2 I would like to stress that Test 1 was mainly performed to prove that the protocol could perform multi-hop without the help of the "wake" functionality.

Test 2 was not as hard to perform as Test 1 because the network was more resilient to poor connections between the nodes, since the wake functionality retransmitted missed REQFs. Therefore I chose to use more aluminum foil to make sure the messages jumped through the network as planned. All the three rounds generated the same result, where all messages got through to the end node. This result was expected because of the large TTL value. I feel confident to conclude that the RWG implementation performs multi-hop with a better message throughput in network setups such as setup 1 illustrated in figure 4.2 when the "wake" functionality is being used.

Another, perhaps obvious, observation that I made while testing multi-hop, especially in network setup 1 in figure 4.2 is that the loss of one REQF, ACK or OKTF, will interrupt the network flow. In Test 1 this lead to messages not reaching the listening end node, and in Test 2 this lead to messages being delayed until they got retransmitted. However, a more dense network, for example setup 3 in figure 4.2, will not suffer as much because several nodes may have heard the REQF and answered with ACKs.

### 4.2.2 Partition Tolerance

Partition tolerance is also a key functionality in this protocol. To verify that it was functioning correctly I both emulated partitioned networks and tested the protocol in networks that were partitioned due to mobility.

The easiest way to emulate a partitioned network is to have one or more nodes deactivated while an active node produces some traffic, and after a while activating the deactivated nodes and exposing them to the network. After this it is easy to verify that the messages have spread to the newly introduced nodes in a correct
or incorrect manner. It is for example essential that all the messages reach all the nodes and that none of the nodes receive duplicates of any message. To give the reader some insight an actual test that were performed will be explained and the results presented. The network consisted of three nodes, A, B and C, where B and C were initially inactive. All the nodes were within each other’s transmission radius and had the same following protocol configuration, \( hops = 7 \), \( groupSize = 20 \), \( TTL = 200 \). The reason for the high \( TTL \) value is because it is easier to verify that all packets have been received correctly if they are not being removed from the REQF buffer. Then node A generated 100 messages carrying different information, the numbers 0 to 99, and after this nodes B and C were activated. The expected behavior for this test is that all the messages should be sent to B and C. I ran this test three times with the following result.

- **Partition Tolerance Test 1, round 1**, 100 of 100 messages were received by both node B and node C and none were duplicates.
- **Partition Tolerance Test 1, round 2**, 100 of 100 messages were received by both node B and node C and none were duplicates.
- **Partition Tolerance Test 1, round 3**, 100 of 100 messages were received by both node B and node C and none were duplicates.

This result was expected, and it shows that the *wake* functionality works as it should. The good results in Multi-hop Test 2 were also due to the *wake* functionality, however this test also shows that the nodes can discover each other. It is because the nodes send a random REQF from their REQF buffer if the network is silent for a given time interval, in this case about five seconds, that makes it possible for them to discover each other.

To test that the protocol was partition tolerant I also tested it in a non-emulated manner. Figure 4.3 illustrates how I tested it. First two nodes A and B shared some messages (figure 4.3 1), then B was moved out of range from A (figure 4.3 2) and was introduced to a new node (figure 4.3 3), C. The results came out the same as in the emulated test.

- **Partition Tolerance Test 2, round 1**, 100 of 100 messages were received by both node B and node C and none were duplicates.
- **Partition Tolerance Test 2, round 2**, 100 of 100 messages were received by both node B and node C and none were duplicates.
- **Partition Tolerance Test 2, round 3**, 100 of 100 messages were received by both node B and node C and none were duplicates.

### 4.2.3 Other

I also performed other tests for example that BS messages were sent properly and that messages were removed according to the TTL.
Figure 4.3. Shows one of the ways I tested partition tolerance. 1) Two nodes share some messages, 2) One node moves out of range, 3) The node that started moving comes in range with a third node and shares the messages that it carries.

Be Silent

When a message has visited the number of nodes according to the group size the message do not need to be forwarded any more. If a node receives a REQF (containing a message) that has reached the group size it will answer with a BS to silent the node that is sending the REQF. The group size is easily changed during startup of the protocol, so the testing of this functionality was rather straightforward. I just verified, by looking at the traces, that messages that had reached enough nodes were not sent further and that BSs were sent accordingly.

However, it should be mentioned that a peculiar situation can occur depending on how REQFs that reach the group size are handled. The problem that might arise is the occurrence of nodes receiving the same message more than once. This happens if REQFs that have reached the group size are removed from the REQF buffer. Any other node that is away from the area with nodes that have noticed K-delivery will have a bit-vector, informed, that can be out of date. Consider three nodes A, B and C, where A and B carries a message M. If B sends this message to C and it reaches the group size, C will respond with a BS. If A does not hear this BS it will not know that message M has reached the group size, hence carrying a REQF where the bit-vector informed is out of date. Let’s assume that B and C remove the REQF that contains the message M due to group size being reached. Now, if node A sends M and C hears the REQF carrying it, C will consider it as a new message, which results in the same message being received more than once.
This problem can be mitigated by only removing REQFs when their time to live (TTL) value expires. However, keeping REQFs in the REQF buffer even though they have reached the group size will have an impact on memory.

**Time To Live**

When a REQFs TTL value expires it must be removed to free memory. To test that this functionality worked properly I tested sending packets with different TTLs and then verified that they were removed, at all the nodes that carried them, when the TTL had expired. However, since there is no clock synchronization between nodes it is possible that the packets will not be removed at the exact same time. Another thing is that the TTL is only represented in seconds, which of course also has an impact on the accuracy on the removal of the packets. In small networks the impact is rather small, however, in large networks it will most likely be more noticeable. The reason for this is because the TTL value is only decreased if a whole second has passed. Consider two nodes A and B, assume that A is about to send a REQF to B and the TTL is 4. However, at the moment the REQF is sent 0.5 seconds have passed. But since the TTL value is represented in whole seconds, the TTL for the REQF sent will still be 4. So, the TTL for the REQF carried by node A will expire 0.5 seconds before the TTL for the REQF carried by node B.

**4.3 Performance Testing**

The focus for this project was not performance. However, some tests have been performed to get an idea of how fast the RWG implementation is, how it behaves and what impact it has on the devices running it.

**4.3.1 CPU Load**

To test the CPU load I used one of the tools I wrote, the program that automatically provides the protocol with messages to send. The fact that the RWG implementation performs busy waiting on the socket and the input pipe, will have an impact on the CPU load. However, the threads are forced to sleep at different states during the execution to reduce the CPU load and to avoid flooding the network.

Let’s have a look at the CPU load test. As mentioned, the test program (the tool) creates new messages at the end of a given time interval and writes the messages on the input pipe. The loads that have been used in this test are 0 messages/second, which works as a reference value, 1 messages/second, 2 messages/second, 4 messages/second, 8 messages/second, 16 messages/second and 32 messages/second. The network consisted of four nodes, where all the nodes were within each other’s transmission radius. All the nodes have the same protocol configuration, \textit{hops} = 7, \textit{groupSize} = 20, \textit{TTL} = 10. The main idea with this test is to see how big impact the RWG implementation will have on the CPU load. During this test it is the node that is running the program that generates new
messages that will be monitored. This node is an IBM t40 with an Intel Pentium M 1200 MHz processor.

- CPU load Test 1, round 1, (protocol not running), 2.0% CPU load.
- CPU load Test 1, round 2, (0 messages/second), 3.4% CPU load.
- CPU load Test 1, round 3, (1 messages/second), 5.2% CPU load.
- CPU load Test 1, round 4, (2 messages/second), 6.2% CPU load.
- CPU load Test 1, round 5, (4 messages/second), 8.3% CPU load.
- CPU load Test 1, round 6, (8 messages/second), 19.0% CPU load.
- CPU load Test 1, round 7, (16 messages/second), 19.3% CPU load.
- CPU load Test 1, round 8, (32 messages/second), 19.5% CPU load.

First of all I need to stress that the CPU load that is presented is not that accurately measured, the data have been collected with the help of the system monitor (a program that monitors the CPU load among other things). The CPU load is also for the whole system, and not only the process running the RWG implementation. However, it gives you an idea of what impact the RWG implementation has on the CPU load. The results shows a steady increase of the CPU utilization, which is expected. At the loads 8 messages/second and higher the CPU load does not increase much. The reason for this is because the REQF buffer gets congested and when this happens the process will sleep for a short period of time and when it wakes up it will look for messages to remove from the buffer. When the buffer is full both incoming and outgoing messages will be discarded.

### 4.3.2 Network Load

I consider the heavy buffer handling to be one of the major weaknesses in the RWG implementation, therefore a couple of tests have been focused on how the protocol behaves under different sending rates in terms of time per message. The two tests that have been performed are similar to the one concerning CPU load. The loads that were used in the tests are 1 messages/second, 2 messages/second, 4 messages/second, 8 messages/second, 16 messages/second. The network consisted of four nodes, A, B, C and D, and only node A generated traffic. All the nodes had the same protocol configuration, $hops = 7$, $groupSize = 20$, $TTL = 10$. However, the first network load test had the network setup that is illustrated in setup 3 in figure 4.2 and the second network load test had the setup that is illustrated in setup 2 in the same figure. What I am interested in with these tests is to see how many messages that reach each node. During each round (different sending rates) 256 messages were generated, and the following are the results for Test 1.

- Network load Test 1, round 1 (1 messages/second), node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.
• **Network load Test 1, round 2 (2 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 1, round 3 (4 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 1, round 4 (8 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 1, round 5 (16 messages/second)**, node B received 195 of 256 messages, node C received 194 of 256 messages, node D received 189 of 256 messages.

The main reason why messages are lost during round 5 (16 messages/second) is because the REQF buffer gets congested which will result in messages being discarded. It should also be mentioned that during this test all the nodes were rather close, this will result in less packets being dropped due to connection issues. However, during sending rates such as 16 messages/second other issues appear, such as several messages being sent in one REQF and nodes receiving duplicate messages. This will of course have an impact on the results in this test. The reason for several messages being sent in one REQF is because the process running the RWG implementation reads messages from the input pipe slower than the test program writes messages to the input pipe. For the IBM t40 with an Intel Pentium M 1200 MHz processor this problem starts appearing at 16 messages/second and increases with the sending rate. At rates such as 64 messages/second every other message consists of several messages. It should also be mentioned that this problem will have more impact on slow devices.

As mentioned, the second test uses another network setup. The reason for this is because it is interesting to see whether the RWG implementation manages to perform multi-hop under the different sending rates. 256 messages were generated during each round (different sending rate) with the following results.

• **Network load Test 2, round 1 (1 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 2, round 2 (2 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 2, round 3 (4 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.

• **Network load Test 2, round 4 (8 messages/second)**, node B received 256 of 256 messages, node C received 256 of 256 messages, node D received 256 of 256 messages.
• **Network load Test 2, round 5 (16 messages/second)**, node B received 208 of 256 messages, node C received 204 of 256 messages, node D received 204 of 256 messages.

It seems like the RWG implementation can handle multi-hop even under heavier loads. However most of the multi-hopping is thanks to the wake functionality. In this test one can also see that the REQF buffer gets congested during the sending rate of 16 messages/second which leads to messages being discarded.

![Diagram](image)

**Figure 4.4.** This is a trace from a network with two nodes, where node A receives 20 messages/second on the input pipe and sends them. The packets have been marked with numbers to show which REQF have triggered the ACK response and so forth.

It is also interesting to mention how the actual packet flow looks under higher sending rates. Figure 4.4 is a trace from a network consisting of two nodes. The test program works in the same manner as in the previous tests. In this case it generates 20 messages/second and three messages were sent. It differs from what one might consider to be the expected packet flow illustrated in figure 4.1. The reason for this behavior is because node A in figure 4.4 will wait for ACKs at least 0.1 seconds before sending an OKTF. And as a result it will be able to send two REQFs before sending the first OKTF. However, it works as illustrated in figure 4.1 if you consider a per packet packet flow.

When testing high sending rates in networks with more than two nodes I made an interesting observation. Before discussing this observation it is necessary for the
reader to understand that the issue that will be described occurs due to a slight deviation between the RWG implementation and the actual RWG protocol specified in the published paper [2]. As I mentioned in chapter 2, the RWG protocol specifies energy conserving functionalities which have not been implemented. However, since the phenomena that occurs is rather interesting it will be addressed.

Figure 4.5. This is a trace from a network with three nodes, where node A is sending messages at the rate 20 messages per second. This will trigger the wake functionality in node C because it does not contain the up to date bit-vectors for REQF(2).

Figure 4.5 illustrates a network setup with three nodes, where node A is the node running the test program that generates new messages. It generates 20 messages/second and writes them on the input pipe. This will result in unnecessary random walks. The reason for this is because node C receives a forwarded REQF from node B, REQF(1). When node C receives this REQF it will start looking for REQFs in its REQF buffer, to see whether it carries any messages that B do not have. This is done with the help of the bit-vector informed and the functionality is known as wake. Since the OKTF for REQF(2) has not been sent at this moment by node A the informed bit-vector in the REQF(2) header will not be up to date. This means that node C will respond with sending REQF(2), which is illustrated in step 7 in figure 4.5. At this point it is necessary to make sure that the informed bit-vector gets updated, and there exist different approaches to solve this.

One approach would be for the two nodes B and C to wait until node A sends the OKTF for REQF(2). When the OKTF arrives, B and C will update their informed bit-vector. However, the OKTF might get lost or perhaps node A moves out of range. This would leave the two nodes B and C in an unfortunate state, and the wake functionality will continue to trigger every time either of them sends a new REQF.
Another approach to solve this and avoid the unfortunate state would be for the node that receives the REQF which contains the out of date informed bit-vector to answer with an ACK. Considering how the RWG protocol works, I think it would be the most logical choice. However, in the case illustrated in figure 4.5 this would result in unnecessary traffic since node A is about to send the OKTF for REQF(2) and the expected packet flow will continue. The unnecessary traffic in this case would be one REQF from C, one ACK from B to C, one OKTF from C with target B and finally one REQF from B. So it will generate quite a lot of traffic.

I also tested an experimental solution for this. When a node receives a REQF and notices that it already has received it even though the informed bit-vector in the incoming REQF claims otherwise it will answer with a BS. The packet type BS, be silent, is usually used when a packet has reached the groupSize value, and should not be sent further. However, in the RWG implementation the BS actually only forces an update of the informed bit-vector. So, by sending a BS instead of an ACK the informed bit-vector will get updated and it will not trigger a response from the receiving node. This results in less unnecessary traffic in the case illustrated in figure 4.5, one REQF from C and one BS from B. Besides the already mentioned approaches there exist other ways to handle this situation that have not been tested. One example would be to introduce timers to make sure that wake does not trigger for recently received REQFs.

The logical ACK answer solution and the experimental BS answer solution were tested to get an idea of which one that is preferable. The sending rate that was used in the test was 16 messages/second. The network consisted of four nodes, A, B, C and D, and only node A generated traffic. All the nodes had the same protocol configuration, hops = 7, groupSize = 20, TTL = 6. The reason why a short TTL was used was because it would prevent the REQF buffer from getting congested. All nodes were within each other’s transmission radius. During each round 256 messages were generated. During this test the test program made sure that the input pipe was empty before writing to it, so there were no messages that contained several messages. The following are the results.

- **Network fast flow test ACK-response**, node B received 238 of 256 messages, node C received 238 of 256 messages, node D received 242 of 256 messages.

- **Network fast flow test ACK-response**, node B received 239 of 256 messages, node C received 233 of 256 messages, node D received 243 of 256 messages.

- **Network fast flow test BS-response**, node B received 217 of 256 messages, node C received 213 of 256 messages, node D received 218 of 256 messages.

- **Network fast flow test BS-response**, node B received 204 of 256 messages, node C received 215 of 256 messages, node D received 205 of 256 messages.

The results seems to point to the conclusion that the ACK response is the better choice. One reason why it is performing better in this test might be because it actually sends the messages more than the BS response mode does. As I mentioned before duplicate messages might occur which will have an impact on the result.
4.3.3 Memory Consumption

Memory consumption was mainly tested by sending and receiving a lot of packets while monitoring the protocols memory consumption in the system manager. In theory the memory consumption should correlate with the size of the buffers and the size of the packets. Since both the buffers and packets were rather small, the memory consumption should be rather low. This seemed to be correct since the system manager did not report any significant increased memory consumption while running the protocol.

Still memory consumption is an interesting topic since the protocol is supposed to run on handhelds which do not have much main memory. Consider the following scenario, a non-partitioned network with a 100 nodes. Each node sends 10 messages per second and the packet size is 1kB (including the header). This would result in roughly 1MB data sent every second, which needs to be stored in the memory. For example, a handheld could have about 200 MB RAM, which would mean that in about 200 seconds all the memory would be drained unless packets have been removed due to the time to live or the group size. However, this scenario is not applicable on my implementation because it would generate network load issues and because the REQF buffer, where the packets are stored, is rather small.

4.3.4 Range

The range has been tested, indoors, by moving a receiving node further and further away from a sending node. I made the following observations. First of all, the range indoors is short and depends a lot on the surroundings. Secondly, the range depends on the network interface card. I will not present any test data concerning the range, basically because the tests have not been extensive enough for this to be interesting. However, my understanding of the range indoors is about 70 meters (transmission radius) and less, depending on the network interface cards and the surroundings.
Chapter 5
Discussion

The discussion in this section will concern my ideas and thoughts about the RWG protocol in general, the way I chose to implement the protocol and my goals.

I believe that the RWG protocol is better applied in networks where the size of the messages are rather small. The reason for this is because fragmentation does not seem feasible, since you do not know which nodes that will receive a REQF when sending it. This means that the fragments of a message may end up at different nodes, which makes the message very hard to reassemble. The kind of information I believe is well suited for this type of protocol are short messages containing for example coordinates or text messages.

Another issue with the protocol is the fact that colliding hashes might occur in the informed bit-vector. The situation that can appear due to this phenomena is nodes that can not transmit messages to each other. A message that is sent between two nodes that generates the same hash value will appear to have already visited the receiving node. To reduce the likelihood for this to happen you can enlarge the size of the bit-vectors, informed and toAvoid. However, since every packet that is sent contains the bit-vectors, even ACKs, OKTFs and BSs, it will generate more overhead. The RWG protocol description [2] covers this problem, and presents a way on how to optimize the choice of the informed bit-vector size.

Another unwanted phenomena that might occur is that restarted nodes might generate packets with IDs that already exists. For example, consider a node that sends a message and then restarts. This will lead to the sequence number counter being reset. Since the sequence number and the origin are used to produce the unique packet ID, new packets will get the same ID as the ones sent before the node restarted. This will lead to lost data, since a REQF that does not contain an unique packet ID might be considered as an old already existing REQF and therefore not be properly handled. However, this is not likely to happen if the time to live for the REQFs is quite short. I think there are at least two approaches that could reduce the likelihood for this to happen even further. The first approach would be to set the sequence number to a random value during startup, instead of zero. Another approach would be to actively checking if the packet IDs are unique. However, the second approach does not actually reduce the chances for
this to happen, it introduces a way to handle it.

The protocol contains some interesting functionalities, for example wake. This functionality, explained in chapter 2, requires that all messages that are in the REQF buffer are compared to incoming REQFs. By doing this a node can figure out whether it holds some messages that another node does not have, and if so, send those messages. But to check the whole REQF buffer for every incoming REQF is a time consuming task and will have impact on the CPU load, especially if the network load is high and the REQF buffer is long. This is an interesting issue that would be interesting to investigate further.

In chapter 2 I described some pros and cons with the way I chose to implement the protocol based on assumptions I had early in the project. Now, after the RWG protocol has been implemented it is interesting to see whether the assumptions I had were valid.

The first decision I made was to implement the RWG protocol in user-space. The reason for this was greater portability and less complexity. I still assume that it is less complex to implement it in user-space, mainly because of better documentation and less severe bugs. However, since I never tried to implement it in kernel-space, it is still an assumption. I also had portability as a reason for the user-space implementation, and the protocol does run on different platforms for example Ubuntu and Android. However, both Ubuntu and Android use a Linux kernel. But I also ported the protocol partially to Leopard (Mac OS X 10.5) just to verify that the vital functionalities are supported, which they were. And Leopard is not a Linux based operating system. And the functionality that is most vital is the ability to open a raw socket, which allows you to create and send your own ethernet packets.

Another, but perhaps more obvious choice I made was to place the RWG protocol above the IEEE 802.11b standard. The argument I had for doing this was that the IEEE 802.11 standards are the most common when it comes to wireless LANs. This let me use off-the-shelf hardware and software, and I did use different network interface cards during the project. I did also argue that by using the 802.11 standards the RWG implementation would suffer from its limitations, this was of course also true. The IEEE 802.11b standards worst drawback, considering the purpose of the implementation was range. The short range will have a crucial impact on the usefulness of the RWG protocol.

I would also like to mention that the implementation was written somewhat on the fly. The reason for this was mainly lack of experience with larger programming tasks. This lead to the fact that I had to re-factor the code a couple of times, which is a quite time consuming task. It would of course been better if I had created a good design during the early phase of the thesis work. However, a good design might have been hard to achieve that early without the experience I have gained throughout the project.

During the planning stage I wrote a time table, in which I estimated how long all the different tasks that belonged to the project would take to solve. The time table was rather accurate, but there was one task that took more time than expected. That task was testing and searching for bugs. Still more testing is needed to see whether I missed any bugs, which I most likely did. It would also
be interesting to see the behavior of the protocol in a large network.

Let’s have a look at the goals I had with this master thesis. I feel confident to say that I have fulfilled both the first goal, Successfully porting the fundamental functionalities of the RWG protocol to laptops running a Linux based operating system, and the second goal, Successfully porting the fundamental functionalities of the RWG protocol to a hand held commodity device (Nokia N810, internet tablet) running a Linux based operating system. I assume there exists bugs in the source and that the RWG implementation can be optimized, but I do consider the RWG implementation to be successful. The third goal, If time allows, porting the fundamental functionalities of the RWG protocol to other devices, for example Android phones, to provide some degree of heterogeneity, was pursued to a certain point, the RWG implementation runs on Android. The forth goal, If time allows, implement or extend a client application to demonstrate the protocol and provide visibility was also pursued. There was a collaboration with a project, POSIT [5]. An interesting and successful collaboration, which is still ongoing. I also implemented my own client application.
Chapter 6

Conclusions and Recommendations

In this section I will present some of the conclusions I have come to during the thesis work. I will also mention some of my thoughts on the subject of implementation and planning.

I have learned more than just C during this project. One of the greatest lessons that I have come to learn is that a good plan is very useful. I do not say that the one that I had was bad, but I believe that if I had more experience with large programming projects and more knowledge about wireless networks it would have been even better. As I have mentioned, the part of the plan that was least accurate was the testing. Testing is far more time consuming than I had imagined. Another very good thing with planning are deadlines. I believe that it is easier to get things finished if there is a deadline, whether it is a deadline where something must be handed in or one where you just have promised yourself that something must be finished.

I would also like to address the approach of my implementation. Even though my first intention, which is not mentioned in the report, was to implement the protocol in kernel-space I am rather satisfied with the user-space implementation. However, I do miss the experience that I would of gained if I had implemented the protocol in kernel-space. Since I never pursued the kernel-space implementation it is hard to say how much better, in terms of performance, that implementation would have been. Considering modern computers I think the performance difference is negligible in small networks. I also believe that the user-space implementation serves the purpose of being able to demonstrate the RWG protocol well.

As I have mentioned in the discussion there exists several drawbacks with the RWG implementation. One of those drawbacks is the fact that the implementation relies upon the IEEE 802.11 standards. The problem is range, it is rather short. However, since the project aims for heterogeneity I do not know any other good alternative besides IEEE 802.11. So in the end perhaps IEEE 802.11 is the way to go. The next step to take, in the development of the RWG implementation that I
would recommend, is improving the buffer handling. I am certain that there exists more efficient ways of handling the buffers. I also think porting the protocol to kernel-space would be interesting.

I found this thesis work to be a really educating and good experience. I have learned a lot about programming, wireless networks, Linux and so on. During this project I have also come to understand how complex an efficient multi-hop ad hoc network is and how useful one can be. Hopefully my work will be of use when demonstrating the RWG protocol, and also encourage people to implement experimental network protocols.
Bibliography


Appendix A

Source Code, Implementation for Linux

```c
/*
 * rwg.h
 */

#ifndef __rwg
#define __rwg

/* Necessary headers*/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <net/ether.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>

#include "rwg_send.h"
#include "rwg_header.h"
#include "rwg_receive.h"
#include "rwg_sockets.h"

/* Some constants*/
#define RWGEther_Type 0x1111 // shows that an eth frame carries a rwg packet as payload
#define Broadcast_Addr "ff:ff:ff:ff:ff:ff"
#define MTU 1024 // maximum transmission unit (bytes)

/*FLAGS*/
int SEND_ACK;
int SEND_REQF_N; // create new from input
int SEND_REQF_F; // forward from buffer
int SEND_REQF_R; // send random when network is silent
int SEND_OKTF;
int SEND_DS;
```
```c
int LISTEN_ACK;
int RETRANSMIT;
int SEND_RETRANSMIT;
int TRACE; // print trace
int WO_MODE; // decides what will be written on the output pipe
int LOOP; // loop flag for the threads
int BS_RESPONSE; // answer with BS when REQF exist in buffer even though incoming REQF says otherwise

/*RWG sender*/
unsigned char sender[6];

/*RWG hashed sender values*/
int hashedAddr;

/*RWG target*/
unsigned char target[6];

/*RWG broadcast*/
unsigned char broadcast[6];

/*RWG groupSize*/
unsigned short int groupSize;

/*RWG hops*/
unsigned short int hops;

/*sequence number for REQF*/
unsigned short int sequenceNumber;

/*Time to live*/
unsigned short int TTL;

/*how long to wait before answering with OKTF*/
unsigned int oktf_interval;

/*FUNC: Called to exit protocol*/
void exit();

/*FUNC: Remove all packets in the reqf buffer*/
void rwg_clean_buffer(packetBuffer *packetBuff);

#endif

/*
 * rwg.c
 *
*/

#include "rwg.h"

/*to serialize the threads*/
pthread_cond_t cond_send = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

/*FUNC: Sets the loop flag LOOP to zero, makes the threads return*/
void rwg_exit() {
    LOOP = 0;
}

/*FUNC: Remove all packets in the reqf buffer*/ // have to handle the active reqf
void rwg_clean_buffer(packetBuffer *packetBuff) {
    int reqf_c;
```
for(reqf_c = 0; reqf_c < packetBuff->reqf_counter; reqf_c++){
    //To make sure that there are no pointers pointing to the freed addr
    //at the heap
    if((packetBuff->waiting[packetBuff->reqf[reqf_c].wait_pos] == &
        packetBuff->reqf[reqf_c])){  
    }
    //To make sure that there are no pointers pointing to the freed addr
    //at the heap
    if(packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] == &
        packetBuff->reqf[reqf_c]){  
        packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] = NULL;
    }
    // remove the packet, since TTL < 0 and the REQF is not in the
    // waiting buffer or the wake buffer
    free(packetBuff->reqf[reqf_c].reqf);
    packetBuff->reqf[reqf_c].reqf = NULL;
    packetBuff->reqf[reqf_c].arrived_at.seconds = 0;
    packetBuff->reqf[reqf_c].arrived_at.u_seconds = 0;
    packetBuff->reqf[reqf_c].wake = 0;
    printf("\"reqf_c, %d\"", reqf_c);
}
packetBuff->reqf_counter = 0;

/*FUNC: Removes old packets from the reqf buffer*/
void refresh_reqf_buffer(packetBuffer *packetBuff){

t_stamp stamp;
set_time_stamp(&stamp);
int reqf_c;
int ttl_check;
packetBuff->temp_reqf_counter = 0;

for(reqf_c = 0; reqf_c < packetBuff->reqf_counter; reqf_c++){
    ttl_check = ((rwg_header *)packetBuff->reqf[reqf_c].reqf)->TTL -
        (stamp.seconds - packetBuff->reqf[reqf_c].arrived_at.seconds);
    if(((rwg_header *)packetBuff->reqf[reqf_c].reqf != NULL) &
        (ttl_check < 0) | | ((rwg_header *)packetBuff->reqf[reqf_c].reqf)
        ->groupSize <= rwg_bitvector_count((rwg_header *)packetBuff->
        reqf[reqf_c].reqf)->visited)){
        //To make sure that there are no pointers pointing to the freed addr
        //at the heap
        if((packetBuff->waiting[packetBuff->reqf[reqf_c].wait_pos] == &
            packetBuff->reqf[reqf_c])){
        }
        //To make sure that there are no pointers pointing to the freed addr
        //at the heap
        if(packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] == &
            packetBuff->reqf[reqf_c]){  
            packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] = NULL;
        }
        // remove the packet, since TTL < 0 and the REQF is not in the
        // waiting buffer or the wake buffer
        free(packetBuff->reqf[reqf_c].reqf);
        packetBuff->reqf[reqf_c].reqf = NULL;
        packetBuff->reqf[reqf_c].arrived_at.seconds = 0;
        packetBuff->reqf[reqf_c].arrived_at.u_seconds = 0;
    }
}
packetBuff->reqf[reqf_c].wake = 0;
packetBuff->reqf[reqf_c].wait = 0;
}
else if((!(rwg_header *)packetBuff->reqf[reqf_c].reqf != NULL)) {
    // move to temp buffer, set new timestamp, change the pointers in the
    // wake/ waiting buffer to the correct addresses
    packetBuff->temp_reqf[packetBuff->temp_reqf_counter] = packetBuff->
        reqf[reqf_c];
    packetBuff->temp_reqf[packetBuff->temp_reqf_counter].arrived_at =
        stamp;
    packetBuff->temp_reqf[packetBuff->temp_reqf_counter].reqf_pos =
        packetBuff->temp_reqf_counter;
}

if((packetBuff->waiting[packetBuff->reqf[reqf_c].wait_pos] == &
    packetBuff->reqf[reqf_c])) {
    packetBuff->waiting[packetBuff->reqf[reqf_c].wait_pos] = &
        packetBuff->temp_reqf_counter;
}

if(packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] == &
    packetBuff->reqf[reqf_c]) {
    packetBuff->wake[packetBuff->reqf[reqf_c].wake_pos] = &packetBuff->
        reqf[packetBuff->temp_reqf_counter];
}

if(ttl_check >= 0) {
    // Just in case, since the TTL is an unsigned int, it will behave
    // very odd if it will be assigned a neg value
    (rwg_header *)packetBuff->reqf[reqf_c].reqf)->TTL = ttl_check;
} else{
    (rwg_header *)packetBuff->reqf[reqf_c].reqf)->TTL = 0;
}
packetBuff->temp_reqf_counter++;

/* Just replace the real REQF buffer with the temporary reqf buffer
packetBuff->reqf_counter = packetBuff->temp_reqf_counter;
if((packetBuff->reqf_counter != 0)) {
    memcpy(packetBuff->reqf, packetBuff->temp_reqf,(packetBuff->
       temp_reqf_counter)*sizeof(reqf_info));
}
*/

/*FUNC: handles thread one, is responsible for sending new packets and
forwarding*/
void *handleThreadO(void ** args){
    pthread_mutex_lock(&mutex);

    /*shared resources*/
    int socket = (int)args[0];
    packetBuffer *packetBuff = (packetBuffer *)args[1];
    //int inPipe = (int)args[2];
    unsigned char* macaddr = (unsigned char*)args[3];

    /*The main loop for thread one*/
    while(LOOP){
        pthread_cond_wait(&cond_send,&mutex);
        rwg_send_routine(socket,packetBuff);
        packetBuff->active_reqf.reqf = NULL; // Just testing this
        pthread_cond_signal(&cond_send);
    }

    // release the lock, before returning
    pthread_cond_signal(&cond_send);
    pthread_mutex_unlock(&mutex);
}
*/FUNC: handles thread two, it will be responsible for listening, to new
input that will be sent n' incoming traffic such as REQF, OKTF...*/

void *handleThread2(void **args)
{
    pthread_mutex_lock(&mutex);
    /* those are shared between threads*/
    int socket = (int)args[0];
    packetBuffer *packetBuff = (packetBuffer *)args[1];
    int inPipe = (int)args[2];
    unsigned char* macaddr = (unsigned char*)args[3];
    unsigned int silentTimer;
    unsigned int refreshReqTimer;
    int outPipe = (int)args[4];
    int retransmit_c;
    int retransmit_stop;
    t_stamp stamp;

    /* the main loop for thread two*/
    if (TRACE) printf("TRACE, inside T2, waiting
    ");
    silentTimer = (unsigned int)time(NULL);
    refreshReqTimer = (unsigned int)time(NULL);

    while (LOOP) {
        set_time_stamp(&stamp);

        /* make sure the REQF buffer is not full, if it is sleep a short period
         and refresh the buffer every 7 seconds, to remove reqfs with
         Time To Live < 0*/
        if (packetBuff->reqf_counter >= (sizeof(packetBuff->reqf)/sizeof(REQF))){
            if (TRACE) printf("REQF buffer is full, messages arriving will be
discarded\n");
            rwg_listen(socket, inPipe, outPipe, packetBuff); // checks for
            ctrlpackets
            refresh_reqf_buffer(packetBuff);
            usleep(5000);
            /* Will refresh the reqf buffer every 7 seconds, to remove reqfs with
             Time To Live < 0s*/
        } else if (((refreshReqTimer+7)<(unsigned int)time(NULL))){
            if (TRACE) printf("Refresh the REQF buffer by checking TTL (must be
done to avoid overflow)\n");
            refresh_reqf_buffer(packetBuff);
            refreshReqTimer = (unsigned int)time(NULL);

            /* Checks if there are any reqf packets in the waiting buffer (waiting for
             ACKs),
             also checks the timestamp (waits 0.1s). Sets SEND_OKTF and signals
             the other thread*/
        } else if (packetBuff->w_tail != packetBuff->w_front &&
            packetBuff->waiting [packetBuff->w_tail % (sizeof(*packetBuff)
                . waiting)/4] != NULL &&
            (check_time_stamp(packetBuff->waiting [packetBuff->w_tail % (sizeof(*packetBuff)
                . waiting)/4])->w_stamp, stamp, oktf_interval))
            SEND_OKTF = 1;
            pthread_cond_signal(&cond_send);
            pthread_cond_wait(&cond_send,&mutex);
            silentTimer = (unsigned int)time(NULL);
            /* If the pointer have been set to NULL, the stack pointer must be
             increased*/
        } else if (packetBuff->w_tail != packetBuff->w_front &&
            packetBuff->waiting [packetBuff->w_tail % (sizeof(*packetBuff)
                . waiting)/4] == NULL){
            packetBuff->w_tail++;
            /* If the network have been silent for 5 seconds send a random REQFs*/
        } else if ((silentTimer+5)<(unsigned int)time(NULL)){
    }
}
if (TRACE) { printf("Network is silent, remove old reqfs (TTL < 0) and send random REQFs
");
refresh_reqf_buffer(packetBuff);
SEND_REQF_R = 1;
pthread_cond_signal(&cond_send);
pthread_cond_wait(&cond_send,&mutex);
silentTimer = (unsigned int)time(NULL);

// Check if there is any packets on the socket or the named pipe*
}
else if (rwe_listen(socket_inPipe, outPipe, packetBuff)) {
if (TRACE) { printf("HandleThread2, All the counters for the mod buffers; ack_c: %i w_front: %i w_tail: %i\n", packetBuff->ack_counter, packetBuff->w_front, packetBuff->w_tail);
}
if (TRACE) { printf("Time... %i : %i\n", stamp.seconds, stamp.u_seconds);
}
silentTimer = (unsigned int)time(NULL);
pthread_cond_signal(&cond_send);
pthread_cond_wait(&cond_send,&mutex);

/* If nothing is happening let the CPU rest a little*/
}
else{
  usleep(5000);
}

// release the lock, before returning
pthread_cond_signal(&cond_send);
pthread_mutex_unlock(&mutex);


/**************************** MAIN ****************************/
*
int main(int argc, char **argv)
{
  pthread_t thread1;
  pthread_t thread2;
  pthread_t thread3;
  char *dev;
  void *arguments[5];
  int socket;
  unsigned char macaddr[6];

  /* Actually named pipes*/
  int outPipe;
  int inPipe;

  /* set default values*/
  TRACE = 0;
  groupSize = 20;
  hops = 7;
  TTL = 10;
  W_MODE = 0;
  BS_RESPONSE = 0;
  oktf_interval = 100000;

  /* Fetch the device name and check flags*/
  if (argc < 2 || 10 < argc) {
    printf("Usage: ./rweexec -t -h <hops> -l <TTL> -g <groupSize> -i <interface> \n");
    return 0;
  }
  else{
    int i = 1;
    char arg[2];
    arg[2] = '\0';
    for(i;i<argc;i++){
      strncpy(arg,argv[i],2);
      if (!strcmp(arg,"-t")){
TRACE = 1;
printf("Setting trace mode\n");
}
else if(!strcmp(arg, "-r")){
    BS_RESPONSE = 1;
    printf("Setting BS response mode\n");
}
else if(!strcmp(arg, "-w")){
    WO_MODE = 1;
    printf("Setting write output mode\n");
}
else if(!strcmp(arg, "-h")){
    hops = atoi(argv[++i]);
    if(0 > hops || hops > 99){
        printf("hops value to low/high\n");
        return 0;
    }
}
else if(!strcmp(arg, "-g")){
    groupSize = atoi(argv[++i]);
    if(0 > groupSize || groupSize > 99){
        printf("groupSize value to low/high\n");
        return 0;
    }
}
else if(!strcmp(arg, "-i")){
    if((dev = argv[++i]) == NULL){
        printf("Incorrect device\n");
        return 0;
    }
}
else if(!strcmp(arg, "-l")){
    TTL = atoi(argv[++i]);
    if(0 > TTL || TTL > 3600){
        printf("TTL value to low/high\n");
        return 0;
    }
}
else if(!strcmp(arg, "-o")){
    oktf_interval = atoi(argv[++i]);
    if(0 > oktf_interval || oktf_interval > 1000000){
        printf("oktf_interval value to low/high\n");
        return 0;
    }
}
else{
    printf("Usage: ./rwgexec -t -w -r -o <oktf_interval> -h <hops> -g <groupSize> -l <TTL> -i <interface> \n");
    return 0;
}
}

/*open streams, named pipes for communicating with other processes*/
inPipe = open("input", O_RDWR|O_NONBLOCK);
outPipe = open("output", O_RDWR|O_NONBLOCK);
if(outPipe == -1 || inPipe == -1){
    perror("Failed to open pipe\n");
}
if (TRACE) {
    printf("hops, \%i\n", hops);
    printf("groupSize, \%i\n", groupSize);
    printf("TTL, \%i\n", TTL);
    printf("oktf_interval \%i\n", oktf_interval);
}

/* this method is later used to get timestamps */
if (TRACE) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    double s = (double)tv.tv_sec;
    double u_s = (double)tv.tv_usec;
    double stamp;
    u_s = u_s * 0.000001;
    stamp = s + u_s;
    printf("Time check: \%f\n", stamp);
}

/* creates the packetBuffer. A struct that holds buffers of different types */
packetBuffer pbuff;
pbuff.reqf_counter = 0;
pbuff.reqf_temp_counter = 0;
pbuff.ack_counter = 0;
pbuff.wake_counter = 0;
pbuff.w_front = 0;
pbuff.w_tail = 0;
int i = 0;

for (; i < (sizeof (pbuff.ack) / 4); i++){
    pbuff.ack[i] = NULL;
}

for (i = 0; i < (sizeof (pbuff.waiting) / 4); i++){
    pbuff.waiting[i] = NULL;
}

for (i = 0; i < (sizeof (pbuff.reqf) / sizeof (reqf_info)); i++){
    pbuff.reqf[i].reqf = NULL;
    pbuff.reqf[i].arrived_at.seconds = 0;
    pbuff.reqf[i].arrived_at.u_seconds = 0;
    pbuff.reqf[i].wake = 0;
    pbuff.reqf[i].wake_pos = 0;
    pbuff.reqf[i].wait_pos = 0;
    pbuff.reqf[i].w_stamp.seconds = 0;
    pbuff.reqf[i].w_stamp.u_seconds = 0;
    pbuff.reqf[i].reqf = NULL;
    pbuff.reqf[i].arrived_at.seconds = 0;
    pbuff.reqf[i].arrived_at.u_seconds = 0;
    pbuff.reqf[i].wake = 0;
    pbuff.reqf[i].wake_pos = 0;
    pbuff.reqf[i].wait_pos = 0;
    pbuff.reqf[i].w_stamp.seconds = 0;
    pbuff.reqf[i].w_stamp.u_seconds = 0;
}

packetBuffer *packetBuff = &pbuff;

/* create a raw socket */
socket = rwg_create_socket(ETH_P_ALL);

/* binds socket to interface */
if (!rwg_bind_socket(dev, socket, ETH_P_ALL))
    { perror("Error: could not bind raw socket to interface"); return 0;}

/* sets O_NONBLOCK flag, so it wont block when buffer is empty */
rwg_nonblock_socket(socket, 1);
/*get the mac address from the network device*/
rwg_get_macaddr(macaddr, socket, dev);

/*prints the macaddr*/
if (TRACE){
    int k = 0;
    printf('macaddr: ');
    for (; k<6; k++){
        printf('%.2x', macaddr[k]);
    }
    printf('
');
}

/*sets sender*/
memcpy(sender, macaddr, sizeof(unsigned char)*6);

/*hash the sender addr n's the hashed value*/
unsigned int hashInput;
memcpy(&hashInput, sender+2, sizeof(int));
hashedAddr = rwg_hash(hashInput);
if (TRACE){printf('hashed sender addr: %i\n', hashedAddr);}

/*sets temporary target*/
memset(target, '0', sizeof(unsigned char)*6);

/*sets broadcast*/
memset(broadcast, 0xff, sizeof(unsigned char)*6);

/*set sequenceNumber*/
sequenceNumber = 0;

/*arguments for the threads, shared resources.. such as packet buffers*/
arguments[0] = (void*)socket;
arguments[1] = (void*)packetBuff;
arguments[2] = (void*)inPipe;
arguments[3] = (void*)macaddr;
arguments[4] = (void*)outPipe;

/*set LOOP flag to 1*/
LOOP = 1;

/*create the threads, sending, listening, handle acks*/
if (0 != pthread_create(&thread1, NULL, (void*)handleThread1, arguments))
{
    perror("Error: failed to create thread\n");
}
if (0 != pthread_create(&thread2, NULL, (void*)handleThread2, arguments))
{
    perror("Error: failed to create thread\n");
}
if (TRACE){printf('TRACE, inside MAIN\n');}

/*waits for the threads to finish before exit*/
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

/*clean up the memory*/
for (i=0; i<pbuff.reqf_counter; i++){
    free(pbuff.reqf[i].reqf);
}

/*close stream*/
close(inPipe);
close(outPipe);

/*close the socket*/
close(socket);

printf('finished\n');

return 1;
/* receive.h, Handles incoming packets */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>
#include <net/ether.h>
#include <netinet/ether.h>
#include <sys/time.h>
#include "rwg_header.h"
#include "rwg_send.h"
#include "rwg.h"

/* FUNC: Just for printing out packets in hex to terminal, is used in trace mode */
void printPacket(unsigned char* buffer, int length);

/* FUNC: handles control messages */
int rwg_handle_control(char* ctrlMess, int length, packetBuffer *packetBuff);

/* FUNC: takes a pointer to a time stamp and sets the current time */
void set_time_stamp(t_stamp* t);

/* FUNC: takes two stamps and an interval(us) and checks if stamp1+interval < stamp2*/
int check_time_stamp(t_stamp stamp1, t_stamp stamp2, unsigned int interval);

/* FUNC: takes payload from received REQF and writes it to the output buffer*/
int rwg_write_output(unsigned char* outputBuffer, int outPipe, int length);

/* FUNC: It’s a hash function, used for the vectors visited n’ recentVisited*/
int rwg_hash(unsigned int address);

/* FUNC: looks up a position in the bit vector, returns 0 if 0 and 1 if 1*/
int rwg_bitvector_lookup(unsigned short int *bitVector, unsigned short int pos);

/* FUNC: sets a position in the bit vector to 1 */
void rwg_bitvector_setbit(unsigned short int *bitVector, unsigned short int pos);

/* FUNC: takes two bit vectors and performs or*/
void rwg_bitvector_update(unsigned short int *bitVector1, unsigned short int *bitVector2);

/* FUNC: counts the 1’s in the bit vector*/
int rwg_bitvector_count(unsigned short int* bitVector);

/* FUNC: Looks up all messages that a sender of an incoming REQF do not possess*/
int rwg_wake(unsigned char incomingSender[6], packetBuffer *packetBuff);

/* FUNC: Handles incoming packets*/
#include "rwg_receive.h"

/* receive.c, Handles incoming packets */

void rwg_read() {
    void rwg_read()
    {
        //must make the packetBuffer public ...

        //check if reqf at packetBuffer->reqf_counter −1 != NULL,

        //take the last packets payload and returns it
    }
}

/* just for printing out packets in hex to terminal, is used in trace mode*/
void printPacket(unsigned char * buffer, int length) {
    int i = 0;
    printf("PACKET: ");
    for(i=0; i<length; i++){
        printf("%.2x ", buffer[i]);
    }
    printf("\n");
}

/* handles control messages*/
int rwg_handle_control(char* ctrlMess, int length, packetBuffer * packetBuff) {
    char type;
    char value[length];

    if(length <4)
        return 0;

    type = ctrlMess[3];

    switch(type) {
        //quit the protocol
case 'q':
    rwg_exit();
    return 1;
    // remove all packets from the reqf buffer
  case 'c':
    rwg_clean_buffer(packetBuff);
    return 1;
    // change trace mode
  case 't':
    if (TRACE) {
        TRACE = 0;
    }
    else {
        TRACE = 1;
    }
    return 1;
    // change WQ_MODE
  case 'w':
    if (WQ_MODE) {
        WQ_MODE = 0;
    }
    else {
        WQ_MODE = 1;
    }
    return 1;
    // change response mode
  case 'p':
    if (BS_RESPONSE) {
        BS_RESPONSE = 0;
    }
    else {
        BS_RESPONSE = 1;
    }
    // change TTL
    return 1;
  case 'l':
    memcpy(value, ctrlMess + 4, length - 2);
    value[length - 2] = '\0';
    if (0 > atoi(value) || atoi(value) > 3600) {
        if (TRACE) {
            printf("rwg_handle_control, TTL value to low/high (0–3600)\n");
        }
        return 1;
    }
    TTL = atoi(value);
    if (TRACE) {
        printf("rwg_handle_control, New TTL is %d\n", TTL);
    }
    return 1;
    // change hops
  case 'h':
    memcpy(value, ctrlMess + 4, length - 2);
    value[length - 2] = '\0';
    if (0 > atoi(value) || atoi(value) > 99) {
        if (TRACE) {
            printf("rwg_handle_control, hops value to low/high (0–99)\n");
        }
        return 1;
    }
    hops = atoi(value);
    if (TRACE) {
        printf("rwg_handle_control, New hops is %d\n", hops);
    }
    return 1;
    // change groupSize
  case 'g':
    memcpy(value, ctrlMess + 4, length - 2);
    value[length - 2] = '\0';
    if (0 > atoi(value) || atoi(value) > 99) {
        if (TRACE) {
            printf("rwg_handle_control, groupSize to low/high (0–99)\n");
        }
        return 1;
    }
    groupSize = atoi(value);
    if (TRACE) {
        printf("rwg_handle_control, New groupSize is %d\n", groupSize);
    }
    return 1;
    // change oktf_interval
  case 'o':
    memcpy(value, ctrlMess + 4, length - 2);
    value[length - 2] = '\0';

```c
if (0 > atoi(value) || atoi(value) > 1000000)
    if (TRACE) { printf("rwg_handle_control, oktf_interval to high/low
        (0-1000000)\n"); }
    return 1;
oktf_interval = atoi(value);
if (TRACE) { printf("rwg_handle_control, New oktf_interval is %i\n", 
    oktf_interval); }
return 1;

default: return 0;
}

/*FNC: takes a pointer to a time stamp and sets the current time*/
void set_time_stamp(t_stamp *t)
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    t->seconds = tv.tv_sec;
    t->u_seconds = tv.tv_usec;
}

/*FNC: takes two stamps and an interval(us) and checks if stamp1+interval
< stamp2*/
int check_time_stamp(t_stamp stamp1, t_stamp stamp2, unsigned int interval)
{
    if (((stamp1.u_seconds + interval) > 999999) {
        stamp1.seconds ++;
        stamp1.u_seconds = (interval - (999999 - stamp1.u_seconds));
    } else {
        stamp1.u_seconds = stamp1.u_seconds + interval;
    }
    if (stamp1.seconds < stamp2.seconds)
        return 1;
    if (stamp1.seconds == stamp2.seconds && stamp1.u_seconds <= stamp2.
        u_seconds)
        return 1;
    return 0;
}

/*FNC: takes payload from received REQ and writes it to the output buffer*/
int rwg_write_output(unsigned char* rwgPacket, int outPipe, int length)
{
    int write_len;
    unsigned char sender[20];
    rwg_header *rwghdr = (rwg_header *)rwgPacket;
    unsigned char out[length - sizeof(rwg_header) + sizeof(sender)];
    sprintf(sender, "%%x%:%%x%:%x%:%x%:%x%:%x%:%x%:\", rwghdr->sender[0], 
        rwghdr->sender[1], rwghdr->sender[2], rwghdr->sender[3], rwghdr->sender 
        [4], rwghdr->sender[5]);

    switch(WO_MODE)
        //adds the sender of the packet to the message
    case 0:
        memcpy(out, sender, sizeof(sender)-1);
        memcpy(out+sizeof(sender)-1, rwgPacket+sizeof(rwg_header), length-
            sizeof(rwg_header));
        out[sizeof(out)-1] = '\r';
        write_len = sizeof(out);
        break;
    ```
case 1 :
    memcpy(out, rwgPacket+sizeof(rwg_header), length−sizeof(rwg_header));
    write_len = length−sizeof(rwg_header) + 1;
    out[length−sizeof(rwg_header)] = '\r';
    break;
default :
    break;
}

if (TRACE){
    int i = 0;
    char temp[length];
    memcpy(temp, out, write_len);
    printf("\rwg_write_output, sizeof(out): %i\n", sizeof(out));
    printf("\rwg_write_output(%i): ", write_len);
    for(;i<write_len;i++){
        printf("%c",temp[i]);
    }
    printf("\n");
}
write(outPipe, out, write_len);
return 1;
}

/*FUNC: It’s a hash function, used for the vectors visited n’ recentVisited*/
int rwg_hash(unsigned int address)
{
    unsigned int k = address;
    int p = 7;
    unsigned int pOnes = 0;
    int i;
    for(i = 0;i<p;i++){
        pOnes = pOnes << 1;
        pOnes = pOnes | 1;
    }
    unsigned int A = 2654435769ull;
    unsigned int Ak = A*k;
    int result = (Ak>>(32−p)) & pOnes;
    return result;
}

/*FUNC: looks up a position in the bit vector, returns 0 if 0 and 1 if 1*/
int rwg_bitvector_lookup(unsigned short int *bitVector, unsigned short int pos)
{
    /*check if a position in the bit vector is 1 or 0*/
    unsigned short int rest;
    unsigned short int byte;
    unsigned short int checkpos = 0x1;
    byte = pos/sizeof(unsigned short int)*8); // find the correct byte pair
    rest = pos % (sizeof(unsigned short int)*8); // which position in the
    byte pair
    checkpos = checkpos << rest; // shifts the bit to the correct position
    (−1 since it starts with 0b1)
    if((checkpos & bitVector[byte]) != 0){
        return 1;
    } else{
        return 0;
    }
/* sets a position in the bit vector to 1 */
void rwg_bitvector_setbit(unsigned short int *bitVector, unsigned short int pos) {
    unsigned short int rest;
    unsigned short int byte;
    unsigned short int setpos = 0x1;
    byte = pos / (sizeof(unsigned short int) * 8); // find the correct byte pair
    rest = pos % (sizeof(unsigned short int) * 8); // which position in the byte pair
    setpos = setpos << rest; // shifts the bit to the correct position (-1 since it starts with 0b1)
    bitVector[byte] = bitVector[byte] | setpos; // just or the value
}

/* takes two bit vectors and performs or */
void rwg_bitvector_update(unsigned short int *bitVector1, unsigned short int *bitVector2) {
    int i = 0;
    rwg_header tmphdr;
    for (; i < (sizeof(tmphdr.recentVisited) / 2); i++) {
        bitVector1[i] = bitVector1[i] | bitVector2[i];
    }
}

/* counts the 1's in the bit vector */
int rwg_bitvector_count(unsigned short int *bitVector) {
    int result = 0;
    unsigned short int pos;
    rwg_header tmphdr;
    int i = 0;
    int j = 0;
    for (; i < (sizeof(tmphdr.recentVisited) / 2); i++) {
        j = 0;
        pos = 0x1;
        for (; j < 16; j++) {
            if ((bitVector[i] & pos) > 0) {
                result++;
            }
        }
        pos = pos << 1;
    }
    return result;
}

/* Looks up all messages that a sender of an incoming REQF do not possess */
int rwg_wake(unsigned char incomingSender[6], packetBuffer *packetBuff) {
    /* hashes the sender of a message */
    unsigned int hashInput;
    memcpy(&hashInput, incomingSender + 2, sizeof(int));
    unsigned short int pos = rwg_hash(hashInput);
    int reqf_c = packetBuff->reqf_counter - 1;
    for (; -1 < reqf_c; reqf_c--){
        // this is for copying pointers to the wake buffer
if (!rwg_bitvector_lookup((rwg_header *)packetBuff->reqf[reqf_c].reqf) 
    && !rwg_bitvector_lookup(((rwg_header *)packetBuff->reqf[reqf_c].reqf)->recentVisited, pos))

    // Check if the packet already exists in the wake/waiting buffer. 
    // Also check that the buffer is not full.
    if (packetBuff->reqf[reqf_c].wake != 1 && 
        packetBuff->wake_count < (sizeof(packetBuff->wake) / 4) && 
        packetBuff->reqf[reqf_c]
    )
    {
        packetBuff->wake[packetBuff->wake_counter] = &packetBuff->reqf[reqf_c];
        packetBuff->wake[packetBuff->wake_counter] = wake = 1;
        packetBuff->reqf[reqf_c].wake_pos = packetBuff->wake_counter; // Will be used when the reqf buffer is refreshed
        if (TRACE) printf("THE \WAKE\_COUNT\ER: %i\n", packetBuff-> 
            wake_counter);
    }

    packetBuff->wake_counter++;

    return 0;

/*FUNCTION: Handles incoming packets*/
int rwg_handle_incoming(unsigned char *packet, packetBuffer *packetBuff, 
    int outPipe, int length)
{

    int type;
    unsigned char *prot_type;
    unsigned char* srcaddr;
    //unsigned char* packet;
    rwg_header *rwghdr;
    struct ether_header* eptr;
    eptr = (struct ether_header *) packet;

    if (eptr->ether_type == RWG\_ETHER\_TYPE) { // ethernet header kastas om (bytevis)
        if (TRACE) {
            printf("\rwg\_handle\_incoming: Ethernet packet contains RWG packet!\n"); 
            printPacket(packet, length);
        }

    /* checking the type of the rwg packet*/
    if (length >= 14) 
        rwghdr = (rwg_header *) (packet+14);
        type = (int)rwghdr->type;
    else{
        if (TRACE){ printf(' \wrg\_handle\_incoming: It’s just an ethernet header\n');
            return 0;
    }

    /*Switch Case over rwg protocol types : REQF (1), ACK(2), OKTF(3), BS 
    (4)*/
    switch(type)
    {
        case 1:
            if (rwg_handle_reqf(rwghdr, packetBuff, outPipe))
                return 1;
        else
break;

case 2:
    if (packetBuff->w_tail != packetBuff->w_front) {
        if (TRACE) { printf("rwg_handle_incoming: Type ACK has arrived\n") ;}
        rwg_handle_ack(rwghdr, packetBuff);
    } else {
        if (TRACE) { printf("rwg_handle_incoming: Type ACK, but not waiting for one\n");}
    }
    break;

case 3: if (rwg_handle_oktf(rwghdr, packetBuff))
    return 1;
    else
    break;

case 4: rwg_handle_bs(rwghdr, packetBuff); break;

default: if (TRACE) { printf("rwg_handle_incoming: Incorrect RWG protocol type\n");
    break;
}
    else{
        if (TRACE) { printf("rwg_handle_incoming: Ethernet packet does not contain RWG (type 1111): %i\n", eptr->ether_type);
    }
    return 0;
}
*/Funct: listen on the raw socket, the named pipe 'input' and check if there are messages in the wake buffer*/
int rwg_listen(int socket, int inPipe, int outPipe, packetBuffer * packetBuff)
{
    unsigned char buffer[ETH_FRAME_LEN];
    unsigned char pipeBuff[MTU-sizeof(rwg_header)];
    int length = 0;

    // check if the buffer is full, only listen for ctrl packets
    if (packetBuff->reql_counter >= (sizeof(packetBuff->reqf)/sizeof( reqf_info))){
        length = read(inPipe, pipeBuff,MTU-sizeof(rwg_header));
            rwg_handle_control(pipeBuff, length, packetBuff);
        }
        return 0;
    }

    // check if there is any data on the socket (data received by the NIC)
    else if ( (length = recvfrom(socket, buffer, ETH_FRAME_LEN, 0, NULL, NULL)) > 0 && rwg_handle_incoming(buffer, packetBuff, outPipe, length))
    {
        return 1;
    }

    // check if there is any data in the input pipe, data that will be sent as a new REQQ
    else if ( (length = read(inPipe, pipeBuff,MTU-sizeof(rwg_header))) > 0) {

        // checks if it is a control message for the protocol
            if (rwg_handle_control(pipeBuff, length, packetBuff))
                return 0;
        }
// sets the new REQF to active_reqf in the packet buffer, will be handled by rwg_send_reqf_n.
// Will set the packet length here so I don't have to pass that value.
unsigned char *reqf;
rwg_header rwghdr;
reqf = (unsigned char *)malloc(sizeof(rwg_header)+length);
rwghdr.packetLength = length+sizeof(rwg_header);
memcpy(reqf,&rwghdr,sizeof(rwg_header));
memcpy(reqf+sizeof(rwg_header),pipeBuff,length);
packetBuff->active_reqf.reqf = reqf;
packetBuff->active_reqf.reqf_pos = packetBuff->reqf_counter;
SEND_REQF_N = 1;

if (TRACE){
  printf("rwg_listened, payload length: %i\n", length);
  printf("SEND_REQF_N is set\n");
}
return 1;

// check if there is data in the wake buffer
}
else if (packetBuff->wake_counter > 0){
  // Needs to check that the pointer have not been NULLed when REQF buffer is refreshed
  if (packetBuff->wake[packetBuff->wake_counter - 1] == NULL){
    packetBuff->wake_counter--;
    return 0;
  }
  // There is data in the wake buffer, set active and the flag SEND_REQF_F reqf and return
  packetBuff->active_reqf.reqf = packetBuff->wake[packetBuff->wake_counter - 1]->reqf;
  packetBuff->active_reqf.reqf_pos = packetBuff->wake[packetBuff->wake_counter - 1]->reqf_pos;
  usleep(25000); // since the other node will wait for ACKS, you don't want to flood the network
  SEND_REQF_F = 1;
  if (TRACE){printf("rwg_listened, data in wake buffer\n");}
  return 1;
}
return 0;

// FUNC: handles incoming REQF:
int rwg_handle_reqf(rwg_header *rwghdr, packetBuffer *packetBuff, int outPipe)
{
  if (TRACE){printf("rwg_handle_reqf: RWG packet is of type REQF\n");}
  // will be used for checking if incoming REQF matches REQF in buffer
  unsigned char packet_id[8];
  int match;
  memcpy(packet_id,rwghdr->origin,sizeof(unsigned char)*6);
  memcpy(packet_id+sizeof(unsigned char)*6,&rwghdr->sequenceNumber,sizeof(unsigned short int));

  unsigned short int *rVisited = rwghdr->recentVisited;
  unsigned short int *visited = rwghdr->visited;
t_stamp stamp;

  // Checks if this packet already exists in the REQF buffer, and updates the visited list.
  // Also sets this node to the visited list (probably redundant)
  if (match >= 0){
    if (TRACE){printPacket((unsigned char *)packetBuff->reqf[match].reqf,((...}}
rwg_header *packetBuff->reqf[match].reqf->packetLength);
    rwg_bitvector_update(((rwg_header *)packetBuff->reqf[match].reqf)->visited,rwghdr->visited);
    rwg_bitvector_update(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited,rwghdr->recentVisited);
}

// checks if this node have messages that the sender of the incoming REQF do not have
unsigned rwg_bitvector_update(rwghdr->recentVisited, rwghdr->sender, sizeof(unsigned char)*6);
    rwg_bitvector_update(rwghdr->recentVisited, rwghdr->sender, sizeof(unsigned char)*6);

    rwg_bitvector_setbit(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, rwghdr->sender);
    packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
    packetBuff->active_reqf.reqf_pos = match;
    SEND_ACK = 1;
    return 1;
}
else{
    if (TRACE) { printf("rwg_handle_reqf: REQF does not exist in buffer (perhaps restarted node)\n");}
}

if (TRACE) { printf("recentVisited when hop Limit)\nif(rwg_bitvector_lookup(rvisited, hashedAddr) && !rwg_bitvector_lookup(rvisited, hashedAddr)){
    if (TRACE) { printf("rwg_handle_reqf: Is already visited but not recentVisited\n");}
    if (match == 0) {
        // changes to the new recentVisited vector and changes the sender of the stored packet (so the ACK will be sent to)
        // the correct node
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, rwghdr->recentVisited, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited));
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->sender, rwghdr->sender, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->sender));
        rwg_bitvector_setbit(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, hashedAddr);
        packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
        packetBuff->active_reqf.reqf_pos = match;
        SEND_ACK = 1;
        return 1;
    }
else{
    if (TRACE) { printf("rwg_handle_reqf: REQF does not exist in buffer (perhaps restarted node)\n");}
}

if (TRACE) { printf("recentVisited when hop Limit)\nif(rwg_bitvector_lookup(rvisited, hashedAddr) && !rwg_bitvector_lookup(rvisited, hashedAddr)){
    if (TRACE) { printf("rwg_handle_reqf: Is already visited but not recentVisited\n");}
    if (match == 0) {
        // changes to the new recentVisited vector and changes the sender of the stored packet (so the ACK will be sent to)
        // the correct node
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, rwghdr->recentVisited, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited));
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->sender, rwghdr->sender, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->sender));
        rwg_bitvector_setbit(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, hashedAddr);
        packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
        packetBuff->active_reqf.reqf_pos = match;
        SEND_ACK = 1;
        return 1;
    }
else{
    if (TRACE) { printf("rwg_handle_reqf: REQF does not exist in buffer (perhaps restarted node)\n");}
}

if (TRACE) { printf("recentVisited when hop Limit)\nif(rwg_bitvector_lookup(rvisited, hashedAddr) && !rwg_bitvector_lookup(rvisited, hashedAddr)){
    if (TRACE) { printf("rwg_handle_reqf: Is already visited but not recentVisited\n");}
    if (match == 0) {
        // changes to the new recentVisited vector and changes the sender of the stored packet (so the ACK will be sent to)
        // the correct node
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, rwghdr->recentVisited, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited));
        memcpy(((rwg_header *)packetBuff->reqf[match].reqf)->sender, rwghdr->sender, sizeof(((rwg_header *)packetBuff->reqf[match].reqf)->sender));
        rwg_bitvector_setbit(((rwg_header *)packetBuff->reqf[match].reqf)->recentVisited, hashedAddr);
        packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
        packetBuff->active_reqf.reqf_pos = match;
        SEND_ACK = 1;
        return 1;
    }
else{
    if (TRACE) { printf("rwg_handle_reqf: REQF does not exist in buffer (perhaps restarted node)\n");}
}
// writes the payload to the output buffer
// if (rwg_write_output((unsigned char*)rwghdr + sizeof(rwg_header), outPipe, rwghdr->packetLength - sizeof(rwg_header))) perror("rwg_write_output failed to write to buffer\n");
if (!rwg_write_output((unsigned char*)rwghdr, outPipe, rwghdr->packetLength))
  perror("rwg_write_output failed to write to buffer\n");

// copy packet and add a pointer of the copy to (reqf) packetBuffer buffer, add length n’ increase counter
unsigned char *reqfCopy = (unsigned char*)malloc(rwghdr->packetLength);
memcpy(reqfCopy, rwghdr, rwghdr->packetLength);
packetBuff->reqf[packetBuff->reqf_counter].reqf = reqfCopy;
packetBuff->reqf[packetBuff->reqf_counter].wake = 0;
// set the time stamp arrived_at in buffer
set_time_stamp(&stamp);
packetBuff->reqf[packetBuff->reqf_counter].arrived_at.u_seconds = stamp.u_seconds;
packetBuff->reqf[packetBuff->reqf_counter].arrived_at.n_seconds = stamp.n_seconds;
packetBuff->reqf[packetBuff->reqf_counter].reqf_pos = packetBuff->reqf_counter;
packetBuff->active_reqf.reqf = packetBuff->reqf[packetBuff->reqf_counter].reqf;
packetBuff->active_reqf.reqf_pos = packetBuff->reqf_counter;
packetBuff->active_reqf.reqf_counter++;
// check if the groupSize have been reached, if so send a BS
if (!reqf_counter)
  if (rwg_bitvector_count((rwg_header*)rwghdr->visited) >= ((rwg_header*)rwghdr)->groupSize)
    SEND_BS = 1;
  return 1;
}
else
  if (TRACE) printf("rwg_handle_reqf: REQF exists in buffer but not recent Visited/visited, updates recent Visited\n");
  // rwg_bitvector_setbit((rwg_header*)packetBuff->reqf[match].reqf)->recent Visited, hashedAddr);
  rwg_bitvector_setbit((rwg_header*)packetBuff->reqf[match].reqf)->visited, hashedAddr);
  memcpy((reqfCopy, packetBuff->reqf[match].reqf)->sender, rwghdr->sender, sizeof(char)*6);
  packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
  packetBuff->active_reqf.reqf_pos = match;
  // Sends out a BS so if the other node will update its bitvectors
  if (BS_RESPONSE)
  SEND_BS = 1;
  return 1;
}

// schedule a send ACK
SEND_ACK = 1;
return 1;
}

if (TRACE) printf("rwg_handle_reqf, recent visited REQF\n");
return 0;

// FUNC: handles incoming OKTF: ss/
int rwg_handle_oktf(rwg_header *rwghdr, packetBuffer *packetBuff)
{
  if (TRACE) printf("rwg_handle_oktf: This packet is of type: OKTF\n");
  int match;
  unsigned char packet_id[8];
  rwg_header tmp;
memcpy(packet_id, rwghdr->origin, sizeof(rwghdr->origin));
memcpy(packet_id + sizeof(rwghdr->origin), &rwghdr->sequenceNumber, sizeof(rwghdr->sequenceNumber));
match = rwg_match_packetid(packet_id, packetBuff);

// update the visited and recent visited vector if the REQF exists in the REQF buffer
if (match >= 0){
    rwg_bitvector_update(((rwg_header*)packetBuff)->reqf[match].reqf)->visited, rwghdr->visited);
    rwg_bitvector_update(((rwg_header*)packetBuff)->reqf[match].reqf)->recentVisited, rwghdr->recentVisited);
} else{
    if (TRACE){printf("rwg_handle_oktf: OKTF no match in reqf buffer\n");
    return 0;
}

// check oktf target
if (memcmp(rwghdr->target, sender, sizeof(tmp.target)) == 0){
    if (TRACE){
        printf("rwg_handle_oktf: Target match\n");
        printPacket((unsigned char*)rwghdr, rwghdr->packetLength);
    }
    packetBuff->active_reqf.reqf = packetBuff->reqf[match].reqf;
    packetBuff->active_reqf.reqf_pos = match;
} else{
    if (TRACE){printf("rwg_handle_oktf: OKTF target does not match, visited /recentVisited updated\n");
    return 0;
}

// check if the visited if the visited list >= groupSize
if (((rwg_bitvector_count(((rwg_header*)packetBuff)->reqf[match].reqf)->visited) >= (((rwg_header*)packetBuff)->reqf[match].reqf)->groupSize))
    if (TRACE){printf("rwg_handle_oktf, will not forward\n");
    return 0;
}

// set flag to forward REQF
SEND_REQF_F = 1;
return 1;
}

//FUNC: handles incoming BS:s, returns 1 if there is a match 0 if not*
int rwg_handle_bs(rwg_header *rwghdr, packetBuffer *packetBuff)
{
    if (TRACE){ printf("rwg_handle_bs: This packet is of type: BS\n");
        // looks at packet_id (origin n’ sequence number), find relating reqf and update visited
        unsigned char packet_id[8];
        int match;
        memcpy(packet_id, rwghdr->origin, sizeof(unsigned char)*6);
        memcpy(packet_id + sizeof(unsigned char)*6, &rwghdr->sequenceNumber, sizeof(unsigned short int));
        match = rwg_match_packetid(packet_id, packetBuff);
        if (match >= 0){
            /* update visited*/
            rwg_bitvector_update(((rwg_header*)packetBuff)->reqf[match].reqf)->visited, rwghdr->visited);
            rwg_bitvector_update(((rwg_header*)packetBuff)->reqf[match].reqf)->recentVisited, rwghdr->recentVisited);
            return 1;
        }
        return 0;
    }

//FUNC: handle ACKs/
int rwg_handle_ack(rwg_header *rwghdr, packetBuffer *packetBuff)
{
    if (TRACE) printf("rwg_handle_ack, this packet is of type: ACK\n");
    //check if the ACK target matches this node's MAC
    if (memcmp(rwghdr->target, sender, sizeof(sender)) == 0) {
        if (TRACE) printf("rwg_handle_ack, ACK target matches MAC addr\n");
        //save a copy of the ACK
        unsigned char *ackCopy = (unsigned char*)malloc(rwghdr->packetLength);
        memcpy(ackCopy, rwghdr, rwghdr->packetLength);
        packetBuff->ack_counter++;
        //free old ACK, if there is one at the new position of the buffer pointer
            if (TRACE) printf("rwg_handle_ack, ack_counter: %i\n", packetBuff->ack_counter % (sizeof(*packetBuff).ack) / 4);
            free(packetBuff->ack[packetBuff->ack_counter % (sizeof(*packetBuff).ack) / 4]);
        }
        return 1;
    } else {
        if (TRACE) printf("rwg_handle_ack, ACK target DOES NOT MATCH MAC addr\n");
    } return 0;
}

/*
 * rwg_send.h
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <netinet/if_ether.h>
#include <net/ethernet.h>
#include <netinet/ether.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <net/if.h>

#include "rwg_sockets.h"
#include "rwg_header.h"
#include "rwg.h"
#include "rwg_header.h"

/*FUNC: The function that is sending all the raw packets*/
int rwg_send_raw(int rawsock, unsigned char *pkt, int pkt_len);

/*FUNC: matches against packets in the REQF packet buffer, returns a
position in the buffer
if there is a match, if there is no match or buffer is empty it returns
-1 */
int rwg_match_packetid(unsigned char packet_id[8], packetBuffer *packetBuff);

/*FUNC: creates new packet with type REQF*/
unsigned char* rwg_create_reqf_n(packetBuffer *packetBuff);
```c
//FUNC: forwards old REQF after getting an OKTF or sending a REQF from the
//wake buffer*/
unsigned char* rwg_createreqf_f(packetBuffer *packetBuff);

//FUNC: forwards a random REQF, when network is silent*/
unsigned char* rwg_createreqf_r(packetBuffer *packetBuff);

//FUNC: creates packets with type ACK*/
unsigned char* rwg_createack(packetBuffer *packetBuff);

//FUNC: creates packets with type OKTF*/
unsigned char* rwg_createoktf(packetBuffer *packetBuff);

//FUNC: creates packets with type BS*/
unsigned char* rwg_createbs(packetBuffer *packetBuff);

//FUNC: handles the send routines*/
int rwg_send_routine(int socket, packetBuffer *packetBuff);

#include "rwg_send.h"

//FUNC: Takes a string, and the its length and passes the information to
//the protocols/
int rwg_write(char* input, int length){
    if(input[length-1] != '\0') return 0;
    // put this in some buffer
    return 1;
}

//FUNC: The function that is sending all the raw packets*/
int rwg_send_raw(int rawsock, unsigned char *pkt, int pkt_len){
    int sent = 0;
    /* A simple write on the socket .. thats all it takes! */
    if((sent = write(rawsock, pkt, pkt_len)) != pkt_len)
        if(TRACE) printf("\rwg_send_raw, Could only send %d bytes of packet of
\nlength %d\n", sent, pkt_len);
    return 0;
}

//FUNC: matches against packets in the REQF packet buffer, returns a
//position in the buffer
//if there is a match, if there is no match buffer is empty it returns
//-1 */
int rwg_match_packetid(unsigned char packet_id[8], packetBuffer *packetBuff){
    int rc;
    if(packetBuff->reqf_counter > 0){
        rc = packetBuff->reqf_counter - 1;
    } else{
        return -1;
    }
    return 1;
}
```
for (rc >= 0; rc--){
    if (memcmp((rwg_header *)packetBuff->reqf[rc].reqf)->origin, packet_id, sizeof((rwg_header *)packetBuff->reqf[rc].reqf)->origin) == 0 &&
        memcmp(&(rwg_header *)packetBuff->reqf[rc].reqf)->sequenceNumber, packet_id+6, sizeof((rwg_header *)packetBuff->reqf[rc].reqf)->sequenceNumber) == 0){
        if (TRACE){printf("rwg_match_packetid, rc: %i \n", rc);}
        return rc;
    }
return -1;
}

//FUNC: creates new packet with type REQF*
unsigned char* rwg_create_reqf_n(packetBuffer *packetBuff){

    rwg_header *rwghdr;
    rwghdr = (rwg_header *)packetBuff->active_reqf.reqf;

    //sets the fields in the rwg header, packetLength is set in rwg_listens/
    rwghdr->type = 0x01;
    rwghdr->hops = 0x01;
    rwghdr->groupSize = groupSize;
    rwghdr->TTL = TTL;
    rwghdr->sequenceNumber = sequenceNumber+4;
    memcpy(rwghdr->origin, sender, sizeof(unsigned char)*6);
    rwghdr->target[0] = 0xff; rwghdr->target[1] = 0xff; rwghdr->target[2] = 0xff;
    memcpy(rwghdr->sender, sender, sizeof(unsigned char)*6);

    //int i = 0;

    //clean the memory*
    memset(rwghdr->visited, '0', sizeof(rwghdr->visited));
    memset(rwghdr->recentVisited, '0', sizeof(rwghdr->recentVisited));
    rwg_bitvector_setbit(rwghdr->recentVisited, hashedAddr);
    rwg_bitvector_setbit(rwghdr->visited, hashedAddr);

    //stores the pointer to the reqf_info in the reqf buffers*/
    w_front % (sizeof(*packetBuff).waiting)/4);

    //fetch a new time stamp*/
    t_stamp stamp;
    set_time_stamp(&stamp);

    //sets the waiting timestamp in the reqf_info*/
    packetBuff->reqf[packetBuff->reqf_counter].w_stamp = stamp;

    //set wait flag*/
    //packetBuff->reqf[packetBuff->reqf_counter].wait = 1;
    packetBuff->w_front++;
/* make sure front wont reach the tail (when the buffers get full) */
if (packetBuff->w_front % (sizeof(*packetBuff)/4) == packetBuff
 ->w_tail % (sizeof(*packetBuff)/4) &&
 packetBuff->w_front != packetBuff->w_tail)
 packetBuff->w_tail = (unsigned int)(&packetBuff + 1);
 packetBuff->w_tail = (unsigned int)(&packetBuff + 1);
}

/* stores the pointer to the reqf in the reqf buffer, and saves the time
 stamp */
packetBuff->reqf[packetBuff->reqf_counter].reqf = (unsigned char *)
 rwg hdr;
packetBuff->reqf[packetBuff->reqf_counter].arrived_at = stamp;
packetBuff->reqf[packetBuff->reqf_counter].reqf_pos = packetBuff->
 reqf_counter;
packetBuff->reqf_counter++;
if (TRACE) printf("%d\n", packetBuff->
 reqf_counter);

/* makes a copy to send */
unsigned char *rwgPacket;
rwgPacket = (unsigned char *)malloc(rwg hdr->packetLength);
memcpy(rwgPacket, rwg hdr, rwg hdr->packetLength);
return rwgPacket;

/* FUNC: forwards old REQF after getting an OKTF or sending a REQF from the
 wake buffers */
unsigned char *rwg_create_reqf_f(packetBuffer *packetBuff){
 if (TRACE) printf("Forwards old reqf\n");
 unsigned char *rwgPacket;
/* check if active_reqf is set */
if (packetBuff->active_reqf reqf != NULL &&
 packetBuff->reqf_counter >
 packetBuff->active_reqf reqf pos)
 if (TRACE)
 printf("%d\n", packetBuff->reqf_counter);
 printf("%d\n", packetBuff->active_reqf reqf pos);
 printf("Is forwarded: %d\n",
 (rwg create reqf f(packetBuffer *packetBuff) ==
 packetBuff->active_reqf reqf pos));

/* increase hop counter, change sender address and update visited list */
 rwg hdr = (rwg hdr) packetBuff->active_reqf reqf;
 memcpy(rwg hdr->sender, sender, sizeof(unsigned char)*6);
 rwg bitvector_update(rwg hdr->visited, rwg hdr->recent Visited);
/* check if the REQF was from the wake buffer, */
if (packetBuff->wake_counter > 0 &&
 packetBuff->wake[packetBuff->wake_counter - 1] != NULL &&
 packetBuff->active_reqf reqf ==
 packetBuff->wake[packetBuff->
 wake_counter - 1]->reqf)
 if (TRACE) printf("Forwards from wake buffer\n");
 if (TRACE) printf("%d\n", packetBuff->
 wake_counter - 1);
packetBuff->wake[packetBuff->wake_counter - 1]->wake = 0;
packetBuff->wake[packetBuff->wake_counter-1] = NULL;
packetBuff->wake_counter =
 /* Fetch a new time stamp */
t_stamp stamp;
set_time_stamp(&stamp);

printf("%u\n", reqf_pos);

if((int)(rwghdr->TTL - (stamp.seconds - packetBuff->reqf[packetBuff->active_reqf.reqf_pos].arrived_at.seconds)) < 0){
    //packetBuff->active_reqf.reqf = NULL;
    return NULL;
}

rwghdr->TTL = (rwghdr->TTL - (stamp.seconds - packetBuff->reqf[packetBuff->active_reqf.reqf_pos].arrived_at.seconds)); //decrease TTL

packetBuff->reqf[packetBuff->active_reqf.reqf_pos].arrived_at = stamp;
// update the arrived at value

/*stores the pointer to the reqf info in the waiting buffer*/

/*make sure front wont reach the tail (when the buffers get full)*/
if((packetBuff->w_front % (sizeof(*packetBuff).waiting)/4) ==
    packetBuff->w_tail % (sizeof(*packetBuff).waiting)/4) &
    packetBuff->w_front != packetBuff->w_tail{
    packetBuff->w_tail++;
}

/*make a copy to send*/
rwgPacket = (unsigned char*)malloc(((rwg_header*)packetBuff->active_reqf.reqf)->packetLength);
memcpy(rwgPacket, packetBuff->active_reqf.reqf, ((rwg_header*)packetBuff->active_reqf.reqf)->packetLength);
return rwgPacket;

else{
    if (TRACE) printf("rwg_create_reqf_r: reqf_counter: %d\n", packetBuff->reqf_counter);
    return NULL;
}

/*FUNC: forwards a random REQF, when network is silents*/
unsigned char *rwg_create_reqf_r(packetBuffer *packetBuff)
{
    unsigned char *rwgPacket;

    if (TRACE) printf("rwg_create_reqf_r: reqf_counter: %d\n", packetBuff->reqf_counter);
/\* choose the reqf with least marks in the visited list */
int reqf_c = packetBuff->reqf_counter - 1;
if (reqf_c > -1) {
    int matches;
    int matchesPrevious = rwg_bitvector_count(((rwg_header *)packetBuff->reqf[reqf_c].reqf)->visited);
    int reqfPrevious = reqf_c;
    if (TRACE) { printf("MATCHES(Visited): "); }
    for (; -1 < reqf_c; reqf_c--) {
        matches = rwg_bitvector_count(((rwg_header *)packetBuff->reqf[reqf_c].reqf)->visited);
        if (TRACE) { printf(" %d", matches); }
        if (matchesPrevious > matches) {
            matchesPrevious = matches;
            reqfPrevious = reqf_c;
        }
        if (TRACE) { printf(" rwg_create_reqf_r, matchesPrevious: %d reqfPrevious: %d\n", matchesPrevious, reqfPrevious); }
    }
    if (TRACE) { printf("\n"); }
    /* copy, return, and set active reqf if visited < groupSize, and change the sender of the packet */
    if (((rwg_header *)packetBuff->reqf[reqfPrevious].reqf)->groupSize > matchesPrevious) {
        /* Fetch time, used for timestamps */
        t_stamp stamp;
        set_time_stamp(&stamp);
        rwg_header *rwghdr;
        rwghdr = (rwg_header *)packetBuff->reqf[reqfPrevious].reqf;
        /* make sure the TTL have not expired before sending this packet, and update TTL */
        if ((int)(rwghdr->TTL - (stamp.seconds - packetBuff->reqf[reqfPrevious].arrived_at.seconds)) < 0) {
            return NULL;
        }
        memcpy(rwghdr->sender, sender, sizeof(char) * 6);
        rwghdr->TTL = rwghdr->TTL - (stamp.seconds - packetBuff->reqf[reqfPrevious].arrived_at.seconds); // decrease TTL
        packetBuff->reqf[reqfPrevious].arrived_at = stamp; // update the arrived_at value
        rwgPacket = (unsigned char *)malloc(sizeof(rwgPacket));
        memcpy(rwgPacket, packetBuff->reqfPrevious, sizeof(packetBuff->reqfPrevious));
        /\* stores the pointer to the reqf in the waiting buffer (reqfs that are waiting for acks) */
        /\* make sure front wont reach the tail (when the buffers get full) */
        if (packetBuff->w_front % sizeof(*packetBuff).waiting) / 4) ==
            packetBuff->w_tail % sizeof(*packetBuff).waiting) / 4) &&
            packetBuff->w_front != packetBuff->w_tail {
            }
packetBuff->w_tail++;
}

if (TRACE){ printf("%s", rwghdr); return rwgPacket; }

return NULL;

/*FUNC: creates packets with type ACK*/
unsigned char* rwg_create_ack(packetBuffer *packetBuff) {

    if (TRACE){ printf("%s", rwg_create_ack: Creates new ACK\n");}

    rwg_header *rwghdr;
    rwghdr = rwg_create_rwghdr();
    rwghdr->packetLength = sizeof(rwg_header);
    rwghdr->type = 0x02;
    rwghdr->hops = 0x00;
    rwghdr->TTL = 0x80;
    rwghdr->groupSize = 0;
    rwghdr->sequenceNumber = ((rwg_header *)packetBuff->active_reqf.reqf)->sequenceNumber;
    memcpy(rwghdr->origin ,((rwg_header *)packetBuff->active_reqf.reqf)->origin ,sizeof(char)*6);
    memcpy(rwghdr->target ,((rwg_header *)packetBuff->active_reqf.reqf)->sender ,sizeof(char)*6);
    memcpy(rwghdr->sender ,sender ,6);
    memcpy(rwghdr->recentVisited ,((rwg_header *)packetBuff->active_reqf.reqf)->recentVisited ,sizeof(rwghdr->recentVisited));
    memcpy(rwghdr->visited ,((rwg_header *)packetBuff->active_reqf.reqf)->visited ,sizeof(rwghdr->visited));

    /*copy the sender of the reqf to target, will be used when creating the
eq header to reduce network load*/
    memcpy(target ,rwghdr->target ,sizeof(unsinged char)*6);

    return (unsigned char*)rwghdr;

    }

/*FUNC: creates packets with type OKTF*/
unsigned char* rwg_create_oktf(packetBuffer *packetBuff) {

    if (TRACE){ printf("%s", rwg_create_oktf: Creates new OKTF\n");}

    rwg_header *rwghdr;

    /*checks that the reqf actually exists*/
    if (packetBuff->w_tail == packetBuff->w_front) {
        if (TRACE){ printf("%s", rwg_create_oktf, waiting is empty\n");}
        return NULL;
        if (TRACE){ printf("%s", rwg_create_oktf, packetBuff->waiting[packetBuff->
        w_tail mod sizeof(*packetBuff).waiting)/4] is NULL \n");}
        packetBuff->w_tail++;
        return NULL;
    } else{
        packetBuff->active_reqf.reqf = packetBuff->waiting[packetBuff->w_tail
        /*find the ACKs belonging to the last reqf in the waiting buffer*/

}
unsigned char ack_id[8];
unsigned char reqf_id[8];
int ack_c = 0;
unsigned char *temp_acks[sizeof(*(packetBuff).ack) / 4]; // a temporary buffer for storing pointers to acks
int temp_ack_c = 0;

memcpy(reqf_id, ((rwg_header *)packetBuff)->waiting[packetBuff->w_tail % (sizeof(*packetBuff).waiting) / 4] -> reqf) -> origin, sizeof(unsigned char) * 6);
memcpy(reqf_id + sizeof(unsigned char) * 6, ((rwg_header *)packetBuff) -> waiting[packetBuff->w_tail % (sizeof(*packetBuff).waiting) / 4] -> reqf) -> sequenceNumber, sizeof(unsigned short int));

/* loops through the whole ack buffer to find acks matching the reqf*/
for (; ack_c < sizeof(*(packetBuff).ack) / 4; ack_c++) {
  if (packetBuff->ack[ack_c] == NULL) {
    memcpy(ack_id, ((rwg_header *)packetBuff) -> ack[ack_c] -> origin, sizeof(unsigned char) * 6);
    memcpy(ack_id + sizeof(unsigned char) * 6, ((rwg_header *)packetBuff) -> ack[ack_c] -> sequenceNumber, sizeof(unsigned short int));
    /* if match add the ack to the temp buffer...*/
    if (memcmp(ack_id, reqf_id, sizeof(ack_id)) == 0) {
      temp_acks[temp_ack_c] = packetBuff->ack[ack_c];
      packetBuff->ack[ack_c] = NULL; // the ack will be freed later
      temp_ack_c++;
    }
  }
}

if (TRACE) { printf("rwg_create_oktf, temp_ack_c: %i\n", temp_ack_c);
/* if there are no matching ACKS, set old pointers to null. increase the counter and return*/
if (temp_ack_c == 0) {
  if (TRACE) { printf("rwg_create_oktf, there are no matching ACKS in the buffer... \n\n");
    packetBuff->w_tail++;
    //packetBuff->active_reqf.reqf = NULL;
    return NULL;
  }
}
/* chose random ack*/
srand((int)time(NULL));
int ackno = rand();
if (TRACE) { printf("rwg_create_oktf, ackno before mod: %i\n", ackno);
  ackno = ackno % temp_ack_c;
if (TRACE) { printf("rwg_create_oktf, ackno: %i\n", ackno);

/* sets the values in the OKTF*/
rwghdr = rwg_create_rwghdr();
rwghdr->packetLength = sizeof(rwg_header);
rwghdr->type = 0x03;
rwghdr->hop = ((rwg_header *)packetBuff->active_reqf.reqf) -> hops;
rwghdr->ttl = 0x00;
rwghdr->groupSize = ((rwg_header *)packetBuff->active_reqf.reqf) -> groupSize;
rwghdr->sequenceNumber = ((rwg_header *)packetBuff->active_reqf.reqf) -> sequenceNumber;
memcpy(rwghdr->orig, ((rwg_header *)packetBuff->active_reqf.reqf) -> origin, sizeof(unsigned char) * 6);
memcpy(rwghdr->target, ((rwg_header *)temp_acks[ackno]) -> sender, sizeof(unsigned char) * 6);
memcpy(rwghdr->sender, sender, sizeof(unsigned char) * 6);

/* set the values to zero (so there is nothing bad in mem)*/
int i = 0;
/* for(; i < sizeof(rwghdr->visited) / 2; i++){
    rwghdr->visited[i] = 0x00;
    rwghdr->recentVisited[i] = 0x00;
*/

    memset(rwghdr->visited, '\0', sizeof(rwghdr->visited));
    memset(rwghdr->recentVisited, '\0', sizeof(rwghdr->recentVisited));

    /* update recentVisited and visited: with the incoming ACKs, for the
     * oktf and the reqf saved in the buffer, also free all the acks.
     * since there is no more use for them */
    temp_ack_c = temp_ack_c - 1;
    for (; i < temp_ack_c; temp_ack_c--) {
        rwg_bitvector_update(rwghdr->recentVisited, ((rwg_header *)temp_acks[temp_ack_c])->recentVisited, ((rwg_header *)packetBuff->active_reqf.reqf)->recentVisited, ((rwg_header *)temp_acks[temp_ack_c])->visited);
        rwg_bitvector_update(rwghdr->visited, ((rwg_header *)temp_acks[temp_ack_c])->visited);
        rwg_bitvector_update(((rwg_header *)packetBuff->active_reqf.reqf)->visited, ((rwg_header *)temp_acks[temp_ack_c])->visited);
        free(temp_acks[temp_ack_c]);
    }
    temp_acks[temp_ack_c] = NULL;
}

    rwg_bitvector_update(rwghdr->visited, rwghdr->recentVisited);
    rwg_bitvector_update(((rwg_header *)packetBuff->active_reqf.reqf)->visited, ((rwg_header *)packetBuff->active_reqf.reqf)->recentVisited);

    /* check the hop limit; update visited if it exceeds the limit and zero
     * out recent visited and hops */
    if (rwghdr->hops > hops) {
        rwghdr->hops = 0;
        ((rwg_header *)packetBuff->active_reqf.reqf)->hops = 0;
        /* clean recent visited */
        memset(rwghdr->recentVisited, '\0', sizeof(rwghdr->recentVisited));
        memset(((rwg_header *) packetBuff->active_reqf.reqf)->recentVisited, '\0', sizeof(rwghdr->recentVisited));
    }

    /* set old pointers to NULL and increase the w_tail*/
    packetBuff->w_tail++;
    //packetBuff->active_reqf.reqf = NULL;
    return (unsigned char *)rwghdr;
}

/* FUNC: creates packets with type BS*/
unsigned char *rwg_create_bs(packetBuffer *packetBuff) {
    if (TRACE) printf("rwg_create_bs: Creates new BS\n");

    unsigned char *rwgPacket;

    /* To avoid segmentation fault, even though this case should not happen*/
    if (((rwg_header *)packetBuff->active_reqf.reqf != NULL)) {
        /* find the correct packet to send BS on, match packet id (origin
         * sequence number)*/
        unsigned short int sequenceNumber;
        unsigned char packet_id[8];
        //send BS, change sender and type from existing REQF, and send without
        //payload/
        rwgPacket = (unsigned char *)malloc(sizeof(rwg_header));
        memcpy(rwgPacket, packetBuff->active_reqf.reqf, sizeof(rwg_header));
        /* change sender address, type, packetSize*/
rwg_header *rwghdr = (rwg_header *)rwgPacket;
    rwghdr->packetLength = (unsigned short int) sizeof(rwg_header);
    memcpy(rwghdr->sender, sender, sizeof(unsigned char) * 6);
    rwghdr->type = 0x4;
} else {
    if (TRACE) printf("rwg_create_bs, active_reqf is NULL\n");
    return NULL;
}

/* set active_reqf to NULL and return*/
// packetBuffer->active_reqf reqf = NULL;
return rwgPacket;
}

/*FUNC: handles the send routines*/
int rwg_send_routing(int socket, packetBuffer *packetBuff)
{
    unsigned char *packet;
    int ehdr_len;
    int rwghdr_len;
    unsigned char *ehdr;
    rwg_header *rwghdr;
    unsigned char *rwgPacket;

    if (SEND_ACK) {
        rwgPacket = rwg_create_ack(packetBuff);
        SEND_ACK = 0;
        if (rwgPacket == NULL) return 0;
        // set specific target, to reduce network load
        ehdr = rwg_create_ethhdr(sender, target, RWGEther_TYPE);
    } else if (SEND_REQF_R) {
        rwgPacket = rwg_create_reqf_r(packetBuff);
        SEND_REQF_R = 0;
        if (rwgPacket == NULL) return 0;
        if (TRACE) printf("rwg_send_routing, LISTEN_ACK set\n");
        ehdr = rwg_create_ethhdr(sender, broadcast, RWGEther_TYPE);
    } else if (SEND_REQF_F) {
        rwgPacket = rwg_create_reqf_f(packetBuff);
        SEND_REQF_F = 0;
        if (rwgPacket == NULL) return 0;
        if (TRACE) printf("rwg_send_routing, LISTEN_ACK set\n");
        ehdr = rwg_create_ethhdr(sender, broadcast, RWGEther_TYPE);
    } else if (SEND_REQF_N) {
        rwgPacket = rwg_create_reqf_n(packetBuff);
        SEND_REQF_N = 0;
        if (rwgPacket == NULL) return 0;
        ehdr = rwg_create_ethhdr(sender, broadcast, RWGEther_TYPE);
    } else if (SEND_OKTF) {
        rwgPacket = rwg_create_oktf(packetBuff);
        SEND_OKTF = 0;
        if (rwgPacket == NULL) return 0;
        ehdr = rwg_create_ethhdr(sender, broadcast, RWGEther_TYPE);
    } else if (SEND_BS) {
        rwgPacket = rwg_create_bs(packetBuff);
        SEND_BS = 0;
        if (rwgPacket == NULL) return 0;
        ehdr = rwg_create_ethhdr(sender, broadcast, RWGEther_TYPE);
    } else {
        if (TRACE) printf("SHOULD NOT GET HERE (besides during exit)\n");
        return 0;
    }

    /* creates a ethernet header*/
    // ehdr = rwg_create_ethhdr(sender, BROADCAST_ADDR, RWGEther_TYPE);
rwghdr_len = sizeof(rwg_header);
ethdr_len = sizeof(struct ethhdr);

// adds ethernet headers
rwghdr = (rwg_header *)rwgPacket;
packet = (unsigned char *)malloc(rwghdr->packetLength+ethdr_len);
memcpy(packet, ethr, ethdr_len);
memcpy(packet+ethdr_len, rwgPacket, rwghdr->packetLength);

if (TRACE) {printPacket(packet, rwghdr->packetLength+ethdr_len);

// Send the packet
if (!rwg_send_raw(socket, packet, ethdr_len + rwghdr->packetLength)) {
    perror("Error sending packet");
    free(rwgPacket);
    free(packet);
    free(ethr);
    return 0;
}

if (TRACE) {printf("rwg_send_routine: Packet sent!\n");

// Free the allocated memory
free(rwgPacket);
free(packet);
free(ethr);
return 1;
}

 RootState
0 — 15 16 — 31
| packetLength | type | hops |
| groupSize | sequenceNumber |
| origin | mac |
| target | mac |
| sender | mac |
| visited | |
| ... | 256 bitv |
| ... | |
| recentVisited | 256 bitv |
| ... | |

packetLength: length of packet in bytes
* type: REQF — Request to forward, ACK — Acknowledgement, OKTF — Ok to forward,
* BF — be silent
* hops: number of hops made by the message
* groupSize: to know how many nodes the message should be delivered to
* sequenceNumber + origin: unique identifier of a message
* target: used when a specific node node is intended to receive the message
* sender: node that sent the message
* visited: bitvector(256) which is used to indicate which nodes that have seen the message
* recentVisited: bitvector(256) which indicates recently infected nodes
*/

#ifndef __
#define __

#include <stdlib.h>
#include <linux/if_ether.h>
#include <netinet/if_ether.h>
#include <net/ethernet.h>
#include <netinet/ether.h>
#include <stdio.h>
#include <string.h>

/*the rwg headers*/
typedef struct {
    unsigned short int packetLength;
    unsigned char type;
    unsigned char hops;
    unsigned short int TTL;
    unsigned short int groupSize;
    unsigned short int sequenceNumber;
    unsigned char origin [6];
    unsigned char target [6];
    unsigned char sender [6];
    unsigned short int visited [16];
    unsigned short int recentVisited [16];
} rwg_header;

/*time_stamp*/
typedef struct {
    unsigned int seconds;
    unsigned int u_seconds;
} t_stamp;

/*used in the reqf buffs*/
typedef struct {
    unsigned char *reqf;
    t_stamp arrived_at;
    t_stamp w_stamp;
    int wake;  // I if reqf is in wake buffer
    int wait;  // I if reqf is waiting on ACK
    int wake_pos;
    int wait_pos;
    int reqf_pos;
} reqf_info;

typedef struct {
    unsigned char *reqf;
    int reqf_pos;
} a_reqf;
typedef struct {
  reqf_info *wake[128]; // keeps track of REQFs that should be sent (because of wake)
  unsigned int wake_counter;
  unsigned char *ack[20]; // keeps track of incoming ACKs
  unsigned int ack_counter;
  reqf_info *waiting[20]; // points at the next free slot in waiting/stamp
  unsigned int w_front; // points at first element
  unsigned int reqf_counter;
  reqf_info reqf[128]; // keeps track of all messages (REQFs)
  unsigned int reqf_counter;
  reqf_info temp_reqf[128]; // used while cleaning the real buffer...
  unsigned int temp_reqf_counter;
  a_reqf active_reqf; // the reqf that is being processed at the moment
} packetBuffer;

/* FUNC: creates a pointer to rwg_header, size 256*/
rwg_header * rwg_create_rwlockdr();

/* FUNC: creates a new ethernet header*/
unsigned char * rwg_create_ethhdr(char *src_mac, char *dst_mac, int protocol);
#endif

/*
 * rwg_header.c
 *
 */
#include "rwg_header.h"

/* FUNC: creates a pointer to rwg_header*/
rwg_header * rwg_create_rwlockdr(){
  rwg_header *header;
  header = (rwg_header *)malloc(sizeof(rwg_header));
  return header;
}

/* FUNC: creates a new ethernet header*/
unsigned char * rwg_create_ethhdr(char *src_mac, char *dst_mac, int protocol){
  struct ethhdr *ethernet_header;
  ethernet_header = (struct ethhdr *)malloc(sizeof(struct ethhdr));
  /* copy the Src mac addr */
  memcpy(ethernet_header->h_source, src_mac, 6);
  /* copy the Dst mac addr */
  memcpy(ethernet_header->h_dest, dst_mac, 6);
  /* copy the protocol */
  ethernet_header->h_proto = htons(protocol);
  /* done ... send the header back */
  return ((unsigned char *)ethernet_header);
}

/* rwg_sockets.h */
```c
/*
 #include <stdio.h>
 #include <string.h>
 #include <stdlib.h>
 #include <errno.h>
 #include <sys/socket.h>
 #include <netinet/in.h>
 #include <arpa/inet.h>
 #include <netinet/if_ether.h>
 #include <netinet/ether.h>
 #include <features.h>
 #include <linux/if_packet.h>
 #include <linux/if_ether.h>
 #include <stdlib.h>
 #include <netinet/if_ether.h>
 #include <linux/if_ether.h>
 #include <fcntl.h>

 int rwg_nonblock_socket(int socket, int value);

 int rwg_create_socket(int protocol_to_sniff); 

 int rwg_bind_socket(char *device, int rawsock, int protocol);

 int rwg_get_macaddr(unsigned char* addr, int s, unsigned char* dev); 
*/

/* rwg_sockets.c Functions for creating sockets */
*/

#include "rwg_sockets.h"

/* FUNC: sets socket in non blocking mode, wont block if buffer is empty*/
int rwg_nonblock_socket(int socket, int value)
{
    int oldflags = fcntl(socket, F_GETFL, 0);
    if (oldflags < 0)
        return oldflags;
    if (value != 0)
        oldflags |= O_NONBLOCK;
    else
        oldflags &= -O_NONBLOCK;
    return fcntl(socket, F_SETFL, oldflags);
}

/* FUNC: creates a new raw socket*/
int rwg_create_socket(int protocol_to_sniff)
{
    int rawsock;
    if ((rawsock = socket(PF_PACKET, SOCK_RAW, htons(protocol_to_sniff))) == -1)
    {
        perror("Error creating raw socket: ");
        exit(-1);
    }
    return rawsock;
} 
```
/*FUNC: Binds a socket to the interface...*/
int rwg_bind_socket(char *device, int rawsock, int protocol)
{
    struct sockaddr_ll sll;
    struct ifreq ifr;
    bzero(&sll, sizeof(sll));
    bzero(&ifr, sizeof(ifr));
    /* First Get the Interface Index */
    strncpy((char *)&ifr.ifr_name, device, IFNAMSIZ);
    if((ioctl(rawsock, SIOCGIFINDEX, &ifr)) == -1){
        printf("Error getting Interface index !\n");
        exit(-1);
    }
    /* Bind our raw socket to this interface */
    sll.sll_family = AF_PACKET;
    sll.sll_ifindex = ifr.ifr_ifindex;
    sll.sll_protocol = htons(protocol);
    if((bind(rawsock, (struct sockaddr *)&sll, sizeof(sll))) == -1){
        perror("Error binding raw socket to interface\n");
        exit(-1);
    }
    return 1;
}

/*FUNC: Gets the mac addr from the network device*/
int rwg_get_macaddr(unsigned char* addr, int s, unsigned char* dev){
    struct ifreq ifr;
    strncpy(ifr.ifr_name, dev, sizeof(ifr.ifr_name)-1);
    if(-1 == ioctl(s, SIOCGIFHWADDR, &ifr)){
        perror("ioctl(SIOCGIFHWADDR)\n");
        return 0;
    }
    memcpy(addr, ifr.ifr_hwaddr.sa_data, sizeof(unsigned char)*6);
    return 1;
}
import java.io.*;
import java.nio.channels.*;

public class Client {

    public static void main(String[] args) {
        System.out.println("Starting client...");
        boolean correctPath = true;

        do {
            try {
                PathPanel p = new PathPanel();
                String path = null;

                if (!correctPath){
                    p.setInfo("Incorrect path");
                }
                correctPath = true;

                while (path == null){
                    path = p.getPath();
                    Thread.sleep(350);
                }

                p.setVisible(false);

                RWGpanel rwgp = new RWGpanel(path);
                FileInputStream fis = new FileInputStream(path.replaceAll("/output", "\output");
                InputStreamReader isr = new InputStreamReader(fis, "ASCII");
                BufferedReader br = new BufferedReader(isr);
                String incoming;
            }
        }
    }
}
/*Reads data on the output pipe, data received from the protocol*/

import java.awt.Container;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Calendar;
import javax.swing.JButton;
import javax.swing.JButton;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.JTextFieldConstants;

public class PathPanel {
    private JFrame f;
    private Container content;
    private JTextField pathText;
    private JTextArea infoText;
    private String path;
    private String info;

    public PathPanel() throws Exception{
        f = new JFrame("Set the path");
        content = new JPanel();
        pathText = new JTextField(20);
        pathText.setText("/home/gusny326/Desktop/RWgmanet/");
        JButton pathButton = new JButton("Set Path");

        infoText = new JTextArea(6,34);
        infoText.setWrapStyleWord(true);
        infoText.setBorder(BorderFactory.createEmptyBorder(8,8,8,8));
        infoText.setEditable(false);
        info = "Set the path to the RWGfolder which contains the named pipes (input/output). Start the protocol before the client, the pipes are blocking (or activate the pipes through the terminal, cat < input, echo";
something > output). Cheers!*/
infoText.setText(info);
content.add(pathText);
content.add(pathButton);
content.add(infoText);

ActionListener setPath = new SendListener();
pathText.addActionListener(setPath);
pathButton.addActionListener(setPath);

f.setLocation(100, 100);
f.setSize(new Dimension(3, 2));
f.setSize(450, 200);
f.setContentType(content);
f.setVisible(true);
f.validate();
}

public void setInfo(String in){
    infoText.setText(in);
}

public String getPath(){
    return path;
}

public void setVisible(boolean bool){
    f.setVisible(bool);
}

/*@ Listens to the send button, reads the data and writes on the input pipe */
private class SendListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        path = pathText.getText();
    }
}

} /*try{
catch (Exception err){
System.out.println("Failed to write to pipe");
}*/

import java.awt.Container;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JScrollPaneConstants;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.InputStreamReader;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.Date;
import java.util.Calendar;
import javax.swing JComboBox;

public class RWGpanel extends JPanel {
    private JFrame f;
    private Container content;
    private JTextField inputText;
    private JTextArea outputText;
    //private Container outputText;
    private Container messContent;
    private JScrollPane opScroll;
    private String oldText;
    private PrintWriter pw;
    private BufferedWriter bw;
    private Calendar cal;
    private JComboBox nodeList;
    private String nodeNumber;

    public RWGpanel(String path) throws Exception{
        System.out.println("This pipe blocks. Open pipe, cat < pipe");
        /* open input pipe */
        FileInputStream fos = new FileInputStream(path + "/input.dat");
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        bw = new BufferedWriter(bos);
        pw = new PrintWriter(bw);

        f = new JFrame("RANDOM-WALK GOSSIP-BASED MANYCAST");
        // some content
        content = new JPanel();
        outputText = new JTextArea(12,35);
        outputText.setWrapStyleWord(true);
        outputText.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));
        outputText.setEditable(false);
        inputText = new JTextField(20);
        JButton sendButton = new JButton("Send");

        opScroll = new JScrollPane(outputText);
        /*opScroll = new JScrollPane(123,123);
         opScroll.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));*/
        opScroll.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

        // drop down list for choosing node
        String[] nodes = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};
        nodeList = new JComboBox(nodes);
        //nodeList.setSelectedIndex(9);
        nodeNumber = "1"; // The node number is 1 as default

        JTextArea as = new JTextArea(12,35);
        opScroll.add(as);

        content.add(opScroll);
        content.add(inputText);
        content.add(sendButton);
        content.add(nodeList);
    }
}
// add an action listener to the sendButton and inputText
ActionListener doSend = new SendListener();
sendButton.addActionListener(doSend);
inputText.addActionListener(doSend);

// add an action listener to the drop down menu
ActionListener setNode = new SetListener();
nodeList.addActionListener(setNode);

// Some layout stuff
f.setLocation(100, 100);
f.setSize(new Dimension(320, 300));
f.setSize(450, 300);
f.setContentType(content);
f.setVisible(true);
f.validate();

/* Prints the incoming message on the text field, and scrolls down */
void setIncomingText(String input)
{
   outputText.setText(outputText.getText() + "Received: " + input + "\n");
   outputText.setCaretPosition(outputText.getText().length());
}

/* Listens to the send button, reads the data and writes on the
   input pipe */
private class SendListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      /* Gets new instance, with time that is up to date */
      Calendar cal = Calendar.getInstance();
      String newText = "" + nodeNumber + " " + cal.getTime().toString().substring(10, 19) +" " + inputText.getText();
      oldText = outputText.getText().concat("sent: " + newText + "\n") ;
      inputText.setText("" + newText);
      outputText.setText(oldText);
      try{
         bw.write(newText);
         bw.flush();
      } catch (Exception err) {
         System.out.println("Failed to write to pipe");
      }
   }
}

private class SetListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      JComboBox cb = (JComboBox)e.getSource();
      nodeNumber = (String)cb.getSelectedItem();
   }
}