IMPLEMENTING A RELIABLE OVERLAY MULTICAST PROTOCOL ON WIRELESS SENSOR NODES

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Gabriel Martins Dias
2011

Leiter der Arbeit:
Professor Dr. Torsten Braun
Institut für Informatik und angewandte Mathematik

Contents

Contents **i**

List of Figures **iii**

List of Tables **vii**

1 Introduction **1**

1.1 Motivation 1
1.2 Goal 3
1.3 Structure of the Thesis 3

2 Related Work **5**

2.1 Wireless Sensor Networks 5
2.1.1 Wireless Sensor Nodes 5
2.1.2 Heterogeneous Wireless Sensor Networks 6
2.2 Communication Schemes 7
2.2.1 Communication Protocol - Internet Protocol 7
2.2.2 Transport Protocols 7
2.2.3 Routing Schemes 9
2.3 Hardware 14
2.4 Contiki Operating System 15
2.4.1 Characteristics 16
2.4.2 Protocol Stack 16
2.4.3 Transport Protocols 18

3 Sensor Nodes Overlay Multicast Communication (SNOMC) **21**

3.1 Node Roles 21
3.2 Definitions 22
3.3 Message Types 23
3.3.1 Default Messages 23
3.3.2 Notification Messages 23
3.3.3 Acknowledgements 24
3.4 Design Models 24
3.4.1 Receiver-driven vs. Source-driven Mode 25

3.4.2 Caching Scheme 25
3.4.3 Transmission Scheme 26
3.5 Data structures 27
3.6 The SNOMC Algorithm 28
3.6.1 Assumptions 28
3.6.2 Phases 28

4 Implementation 37
4.1 SNOMC Implementation in Contiki 37
4.1.1 Memory Allocation 38
4.1.2 Transmission Procedure 40
4.1.3 Receiving Procedure 43
4.1.4 Pre-defined Values 43
4.1.5 Additional Details 46
4.2 UDP and TCP Implementations in Contiki 48
4.2.1 Application using UDP 48
4.2.2 Comparing SNOMC and UDP algorithms 50
4.2.3 Application using TCP 51

5 Evaluation 53
5.1 Scenario 53
5.2 Test Configuration 54
5.3 Results 55
5.3.1 Transmission of a Configuration (20 bytes) 58
5.3.2 Transmission of a Code Update (1000 bytes) 60

6 Conclusions and Future Work 65
6.1 Conclusions 65
6.2 Future Work 66

Bibliography 67

List of Figures

1.1 Heterogeneous Network - WSN Management Scenario. 2

2.1 The architecture of a Wireless Sensor Node[1]. 6
2.2 Client-server interaction during a TCP connection. 8
2.3 The TCP header. 9
2.4 Client-server interaction during a UDP transmission. 9
2.5 The UDP header. 9
2.6 Many unicast transmissions. 10
2.7 A broadcast transmission. 11
2.8 A multicast transmission. 12
2.9 Tmote Sky module description. [2] 15
2.10 TelosB block diagram. [3] 16
2.11 The protocol stack in Contiki. 17
2.12 TCP and μIP headers of the Contiki's implementation. 18
2.13 UDP and μIP headers of the Contiki's implementation. 19

3.1 Nodes have different roles in SNOMC. 22
3.2 Transmission of *notification messages* using *positive acknowledgements*. 24
3.3 Message types in SNOMC. 27
(a) *Data* message type in SNOMC. 27
(b) *Start group* message type in SNOMC. 27
(c) *Finish group* message type in SNOMC. 27
(d) *Content acknowledgement* message type in SNOMC. 27
(e) *Positive acknowledgement* message type in SNOMC. 27
(f) *Start acknowledgement* message type in SNOMC. 27
3.4 Flowcharts of the SNOMC's *early phase*. 29
(a) Flowchart in the source node. 29
(b) Flowchart in other nodes. 29
3.5 Sequence diagram of the *early phase*. 29
3.6 Flowchart of the *transmission phase* in the *source node*. 30
3.7 Flowchart of the *transmission phase* in nodes that do not cache any *fragment*. . 31
3.8 Flowchart of the *transmission phase* caching the fragments and using the *first forward* approach. 31

3.9 Flowchart of the *transmission phase* caching the fragments and using the *first check* approach. 32
3.10 Flowchart of the *transmission phase* in the *receiving node*. 32
3.11 Sequence diagram of the *transmission phase* first forwarding the *fragments*. . . 33
3.12 Sequence diagram of the *transmission phase* first checking the *fragments*. . . . 33
3.13 Sequence diagram of the *transmission phase* first forwarding the *fragments*. . . 33
3.14 Sequence diagram of the *transmission phase* first checking the *fragments*. . . . 34
3.15 Flowcharts of the SNOMC's *final phase*. 36
(a) Flowchart in the source node. 36
(b) Flowchart in other nodes. 36
3.16 Sequence diagram of the *final phase*. 36

4.1 Protocol stack with SNOMC. 37
4.2 Snapshot of the memory consumption. 40
4.3 Flowchart of the UDP application's algorithms. 49
(a) Flowchart in the *source node*. 49
(b) Flowchart in *receiving nodes*. 49
4.4 Sequence diagram of the UDP implementation. 49
4.5 The scenario used to compare the algorithms. 50
4.6 Minimum number of transmissions used to transmit one fragment. 50
4.7 UDP implementation doing a transmission to one branch without packet losses. 51
4.8 Flowchart of the TCP application's algorithms. 52
(a) Flowchart in the *source node*. 52
(b) Flowchart in *receiving nodes*. 52
4.9 Sequence diagram of the TCP implementation. 52

5.1 Distribution of the nodes inside the building. 54
5.2 Time to transmit 20 bytes in the 2-nodes scenario. 57
(a) With TCP measurements. 57
(b) Without TCP measurements. 57
5.3 Numbers observed after transmitting 20 bytes in the 2-nodes scenario. 58
(a) Percentage of packet losses. 58
(b) Average number of transmissions. 58
5.4 Time to transmit 20 bytes in the 6-nodes scenario. 59
(a) With TCP measurements. 59
(b) Without TCP measurements. 59
5.5 Numbers observed after transmitting 20 bytes in the 6-nodes scenario. 59
(a) Percentage of packet losses. 59
(b) Average number of transmissions. 59
5.6 Time to transmit 1000 bytes in the 2-nodes scenario. 60
(a) With TCP measurements. 60
(b) Without TCP measurements. 60
5.7 Numbers observed after transmitting 1000 bytes in the 2-nodes scenario. 61
(a) Percentage of packet losses. 61

(b) Average number of transmissions. 61
5.8 Time to transmit 1000 bytes in the 6-nodes scenario. 62
5.9 Numbers observed after transmitting 1000 bytes in the 6-nodes scenario. 62
(a) Percentage of packet losses. 62
(b) Average number of transmissions. 62

List of Tables

4.1 Comparing UDP and TCP in Contiki. 38

5.1 Comparing SNOMC transmissions in the 2-nodes scenario (ideal case). 56
5.2 Comparing UDP transmissions in the 2-nodes scenario (ideal case). 56
5.3 Comparing SNOMC transmissions in the 6-nodes scenario (ideal case). 57
5.4 Comparing UDP transmissions in the 6-nodes scenario (ideal case). 57

Acknowledgment

First I would like to give my special thanks for all support I received during the development of this work.

Thank you Prof. Dr. Torsten Braun for the opportunity to be in the research group *Computer Networks and Distributed Systems*. I also would like to express my profound gratitude to Gerald Wagenknecht who supported me during the course of my diploma, helped me with technical and theoretical orientations, and sacrificed a lot of his spare-time proof-reading my thesis several times in order to achieve the best results. My special thanks go to Markus Anwander for all of the technical hints which helped me improve the implementation performance and to become familiar with Contiki.

Finally, I would like to thank my parents, Orlando and Maria Lucia Dias, for their unwavering support as well as all of my friends that encouraged me to write this thesis.

Chapter 1

Introduction

1.1 Motivation

Wireless sensor networks[4] (WSNs) are deployed in all kinds of environments. They are composed mostly of simple wireless sensor nodes, which offer very limited memory and may not have high energy consumption because their source power use to be common batteries rather than electric power. Since these nodes do not have high computational capacity, they are usually used to collect data from the environment and transmit them to a computer placed inside the network.

WSNs can be used for various applications in many areas such as monitoring building structures[5]. In these scenarios, the wireless sensor nodes are responsible for detecting vibrations, temperature and humidity, and transmitting the data to a central computer. Their software may work over a middleware in order to not be distracted by low-level abstractions, maintainability and code reuse. The software requirements of these applications combine two different challenges: they must run for long periods of time and transmit a large amount of collected data.

Even though these most common software requirements are also applicable to military applications[6], there are some new specific requirements in these environments such as security and quality of service to be supported by the WSNs. The different priorities in these environments include large-scale networks, self-configuration, network connectivity and maintenance, in addition to energy consumption. The WSNs may also work with well-defined topology and hierarchy strategies on its nodes. The communication between these wireless sensor nodes must save time as well as network bandwidth.

The cases above exemplify general uses of the WSNs with nodes deployed and distributed non-uniformly among a determined area. This configuration brings about a new issue: Occasionally the wireless sensor nodes may have their software maintained and updated in order to fix bugs or to change business models according to the collected data or environmental changes.

Compared to the transmission of the collected data, management tasks have more demanding requirements. For data collection, the data is transmitted from one or many sensor nodes to one base station, usually with a low data rate and not necessarily in a reliable way. Management of wireless sensor nodes requires reliable transmissions and has "bursty" traffic, because of the size of the updates. Therefore, the large number of extra messages transmitted by the nodes through the established connections can become a problem.

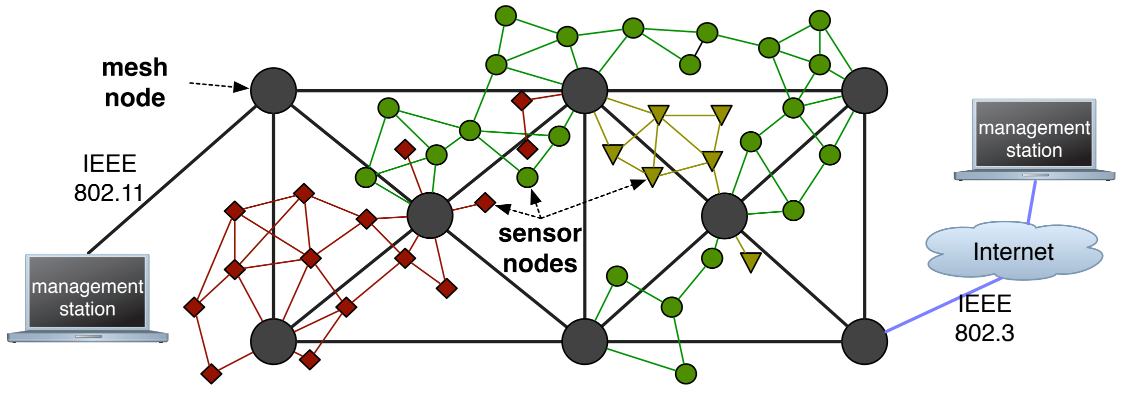


Figure 1.1: Heterogeneous Network - WSN Management Scenario.

Software updates can be done using either User Datagram Protocol[7] (UDP) or Transmission Control Protocol[8] (TCP) as transport protocols. The former produces less overload on the network and is unreliable, while the latter provides trustworthy connections with the disadvantage of a significant increase in the number of messages.

These protocols are standard solutions on the Internet and are capable of providing unicast and multicast communications between two or more points. The unicast option is not feasible in large wireless networks due to the many redundant connections during the transmission. On the other hand, multicast communication can be a solution so that one computer can communicate with a group of other nodes without transmitting the same message more than once. Unfortunately, in order to avoid problems like network flooding and denial-of-service attacks that can interrupt the network traffic, IP Multicast has not been widely deployed in the global Internet and is not really usable by end-users.

However, the multicast functionality can be implemented at the application level; this concept is called Application Level Multicast[9] (ALM) or just Overlay Multicast. In this approach, the nodes can transmit data to the others using groups based on the multicast communication but organized by the application. Thus, the routers do not interfere even if the messages go to different sub-networks, because it is based on the Peer-to-Peer[10] (P2P) paradigm.

Large networks may contain different hardware architectures integrating with each other as shown in Figure 1.1. Computers and also wireless mesh nodes can improve communication in order to avoid problems such as network congestion. The main focus is to increase network productivity by keeping nodes saving computational time to connect and to transmit data through the network.

This type of heterogeneous network requires a common way to transmit data over transparent connections. Internet Protocol[11] (IP) and IP-equivalent (such as μIP[12]) are the most used implementations to interconnect them. Since the nodes can easily communicate despite their hardware restrictions, they will transmit their data without worrying about the neighbours' technologies. As a result, the network will be scalable and will perform as well as the simpler configurations.

1.2 Goal

The main goals of this thesis are to implement, optimize, and evaluate the Sensor Nodes Overlay Multicast Communication[13] (SNOMC) protocol's performance in a real-world WSN.

SNOMC provides end-to-end reliable overlay multicast communication within a management station and its many nodes. The amount of transmitted data may vary from a few bytes (such as configuration changing) to large contents (in case of code updates, for example). For data transmission to be successful, redundant unicast connections must be avoided in order to prevent a big overload on the network during the transmission period. The protocol is reliable, which means that if a host starts to receive data, it will receive the whole content and the protocol must handle packet losses and out-of-order delivering problems that might occur during the transmission. It would be a problem if the receiver did not receive either new configurations or software updates transmitted by the management station.

Another important aspect is the scalability. The protocol must deliver good performance with a large number of participating nodes as well as with a smaller number of transmissions.

1.3 Structure of the Thesis

Chapter 2 contains an explanation of WSNs, including their environmental requirements and specifications. After this overview, the nodes' hardware and software characteristics are presented. Some research done in the past and previously used solutions are also described. Chapter 3 contains details about SNOMC with diagrams and figures for further explanation. The implementations of SNOMC and two other versions that use UDP and TCP as transport protocols are shown in Chapter 4. Chapter 5 describes test scenarios, and obtained results are evaluated. Finally, Chapter 6 presents conclusions and an outlook to future work.

Chapter 2

Related Work

2.1 Wireless Sensor Networks

A Wireless Sensor Network[4] (WSN) is composed of wireless sensor nodes and computers. Extra devices can be added in order to improve performance and reduce energy consumption in the nodes. The following sections describe the nodes and details about the WSNs' architectures.

2.1.1 Wireless Sensor Nodes

In recent years, small embedded systems have become popular in certain industries, military forces[6] and for environmental monitoring[5]. They are characterized by a low level of power consumption, portable size, and low cost. Their physical constraints do not prevent them from being self-sufficient and part of a larger network.

Usually, it is possible to configure and replace necessary components such as the flash memory and the radio processor according to the application requirements. The embedded microprocessor has some basic functionalities. To manage data collected by the sensors, perform power management algorithms, interface the sensor data to the physical radio layer and manage the radio network protocol are typical examples of these functionalities[14].

Figure 2.1 shows a wireless sensor node's architecture. Besides the processing units, it may be possible to see one or more sensors used to detect environmental changes like humidity, temperature, and even movements. Since these nodes have limited memory capacity, their main function is to collect data and forward it to either a real computer, a server, or a mesh node. Furthermore, one of the most important requirements for any wireless sensor node is to minimize the power consumption of the devices. In comparison with other basic hardware, a radio system consumes a large amount of battery power and requires good management algorithms to give the nodes a long lifetime.

The wireless sensor nodes present an event-driven architecture model and the algorithms have to handle the sensed events and process the corresponding collected data. The idea is to minimize the power consumed by the device and, to make this possible, the embedded microprocessor may be capable of managing radio power, sensors and any other hardware attached to the node.

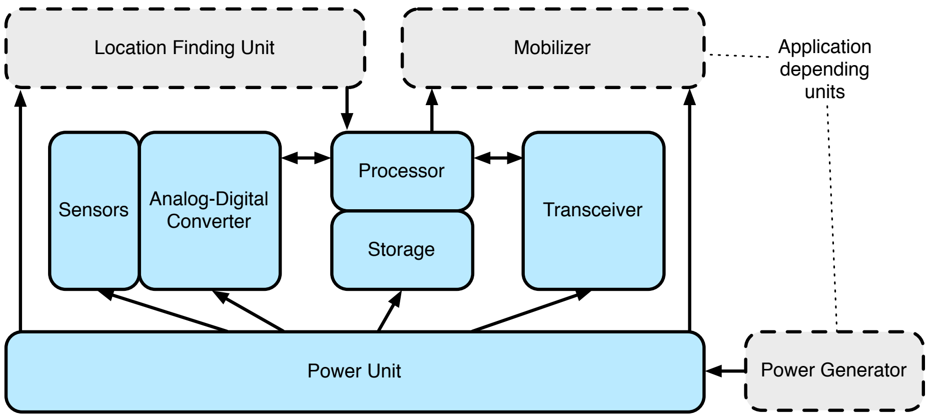


Figure 2.1: The architecture of a Wireless Sensor Node[1].

2.1.2 Heterogeneous Wireless Sensor Networks

Usually, wireless sensor nodes have limited transmission power and require multi-hop paths to establish a communication inside a WSN. As a consequence of this, many packets must be sent through the network, and problems such as collisions and interferences can cause many packet losses.

In order to improve these connections, wireless mesh nodes are used to divide large networks into smaller WSNs[15]. Thus, there are fewer hops inside the networks and less transmissions are required to establish connections. These networks with different node types are called heterogeneous networks and can be handled as simple WSNs from the programmer's perspective.

In some cases, wireless mesh nodes may be added and removed during the system operation, and no extra administrative procedure has to be executed in order to configure them. They only need a communication interface to realize this functionality and to increase the quality of the links, as well as their messages' throughput.

Figure 1.1 shows different device types deployed in the heterogeneous networks. Different wireless sensor node's architectures, wireless mesh nodes, and computers can be observed. Inside this system, the differences between the machines must be transparent at the application level. From the communication point of view, it is feasible by working over the TCP or UDP layers, but it also has to consider the limited resources from the simple nodes.

Eventually, the software of the nodes located in the heterogeneous networks has to be updated in order to fix defects, increment the functionalities or even change the behaviour of the nodes. In this scenario, such management tasks are controlled by the management station and the mesh nodes act as a communication gateway between the different sensor sub-networks. To establish a communication venue between the management station and the sensor nodes there are three main possibilities: unicast, broadcast, and multicast communication.

2.2 Communication Schemes

Wireless sensor nodes have limited options for establishing a communication channel between themselves or other devices. In this section, transport and communication protocols are described, as well as the positive and negative aspects of each option.

2.2.1 Communication Protocol - Internet Protocol

The Internet Protocol[11] (IP) is the most used communication protocol and the standard on the Internet Protocol Suite[16]. It is responsible for addressing hosts and routing sent packets from a source host to the destination host, even if they are on different IP networks.

IP layer provides best effort delivery. Therefore, there is no guarantee about the proper delivery of the datagrams and the reliability of the service. Thus, some errors such as data corruption, lost data packets, duplicate delivery, and out-of-order packet delivery may occur in this scenario.

2.2.2 Transport Protocols

Transmission Control Protocol (TCP)

As discussed above, IP does not provide any kind of reliability and TCP[8] was developed in order to cover this problem at the transport layer. To achieve this, TCP adapts to properties of the network (according to different locations, topologies and hierarchies) and is robust (using retransmissions, packet reordering and acknowledgement schemes) in the face of many kinds of failures that may occur at the lower layers.

The basic and ideal client-server interaction is shown in the Figure 2.2 and can be described as:

1. **Client** establishes a connection by sending a synchronize (SYN) packet.
2. **Server** answers the SYN packet with an acknowledgement (ACK) packet.
3. **Client** completes the three-way handshake with an ACK. Connection established.
4. **Client** sends the request. May be multiple packets.
5. **Client** finishes the request. It sends a final (FIN) packet to indicate that it is done sending.
6. **Server** acknowledges the request with an ACK packet and the FIN packet.
7. **Server** answers the request with a reply.
8. **Server** sends a FIN packet to indicate that it is done answering.
9. **Client** answers the received FIN packet and closes the connection.
10. **Server** closes the connection.

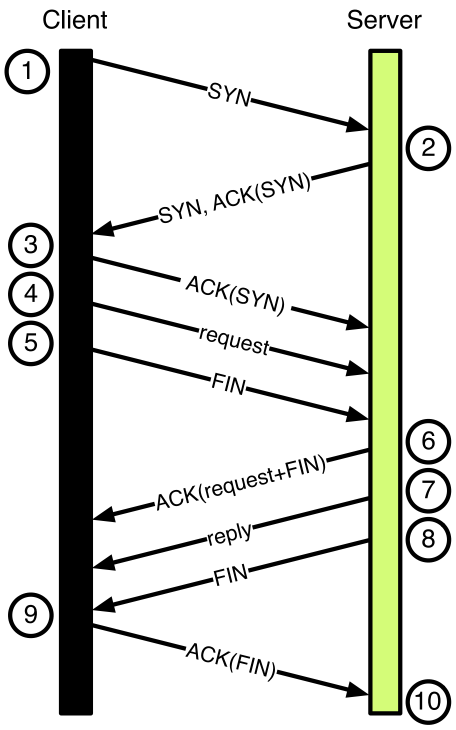


Figure 2.2: Client-server interaction during a TCP connection.

In order to have a reliable connection, server and client hosts have to store some information about the current connections such as the acknowledgement sequence number. This brings high memory costs to TCP against the limited memory capacity of these wireless sensor nodes.

According to the Internet standard, the TCP header has at least 20 bytes of length (see Figure 2.3). From those, some 32-bit words are required to send the basic information such as sequence and acknowledgement numbers. Besides this, all Internet hosts must accept TCP segments of 556 bytes[17]. These characteristics are good examples of the incompatibility of the pure-TCP with most of the wireless sensor nodes, which work with a 16-bit processor and may not send more than 128 bytes on each radio transmission.

User Datagram Protocol (UDP)

With UDP[7], each packet is unique and no information about it is stored in the node after its transmission. The protocol does not control if it has been already delivered to the application and IP problems (for example data corruption and lost data packets) may have an impact on the the application. Figure 2.4 shows that the connection is stateless and the hosts do not store any information about the packets after they are delivered to the upper layer.

UDP is a simple transport protocol, which basically adds a small header to the IP content in order to identify the ports used to do the communication, to define the length of the package and to insert a checksum validation, as shown in Figure 2.5. The goal is to deliver the message to

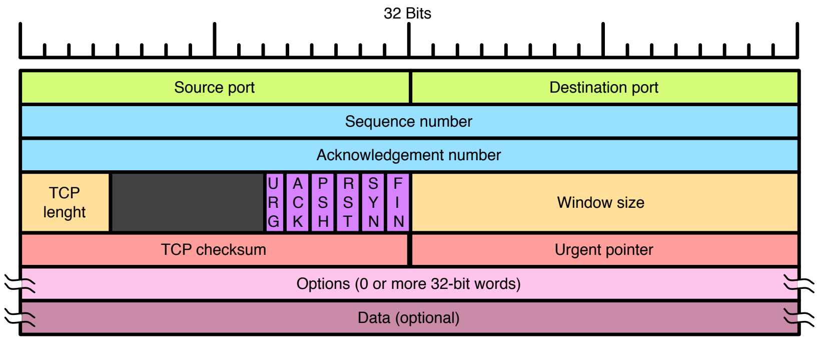


Figure 2.3: The TCP header.

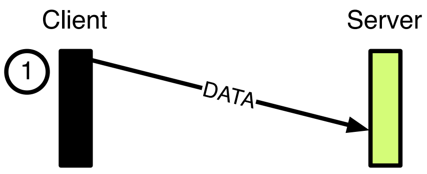


Figure 2.4: Client-server interaction during a UDP transmission.

the application that is listening to the destination port if no corruptions have occurred. Although this protocol is unreliable and does not provide any fragmentation mechanism, it only delivers the content to the application layer if there is no corrupted data in the datagram.

2.2.3 Routing Schemes

Unicast

The unicast connection is the simplest way to keep a private communication with any other device inside the network. Information about the flows (shown in Figure 2.6) is stored in packet headers, which makes it possible to identify the source and receiver nodes.

In a large network with numerous wireless sensor nodes, management tasks such as software updates can take a long time if done using unicast connections. Usually, in a software update routine, many nodes must receive the content. If unicast connections are used, this procedure creates one new connection to each receiver and the same content is sent many times through the network. However, this is very inefficient and consumes resources such as bandwidth and

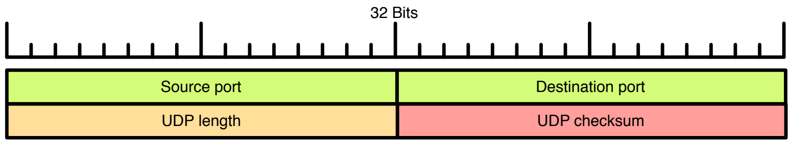


Figure 2.5: The UDP header.

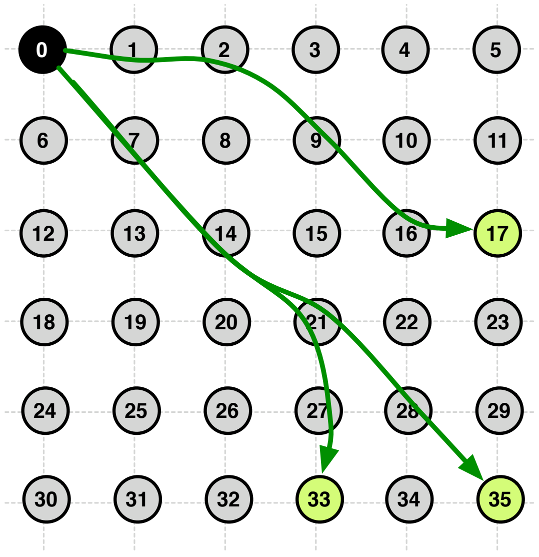


Figure 2.6: Many unicast transmissions.

energy. The complexity of a transmission using unicast is

$$O(receivers \times numberofhops \times messages)$$

It is even possible to save energy and avoid unnecessary transmissions by reducing this order of growth and, consequently, the power consumption.

Broadcast

Broadcast transmission is the simplest way of doing large scale transmissions. By using this, the sender distributes one data packet without defining a specific receiver and any device positioned in the transmission's range may get the content.

Many Ethernet networks support the broadcast inside a sub-network. This is required by some configuration protocols like Address Resolution Protocol[18] (ARP) and Dynamic Host Configuration Protocol[19] (DHCP). However, the routers do not allow the broadcast of messages from other networks to avoid network overload and Denial of Service[20] (DoS) attacks. This reduces using coverage of this scheme in such cases.

In general, there are three drawbacks to this scheme:

- In the management tasks, most of the sensor nodes might be out of the source's transmission range and will not receive the updates. This can be solved by using some kind of forwarding, which is done by the other nodes located inside the network.
- It is not possible to select which nodes are going to receive the content and the updates may be addressed for some specific nodes in the network. For example, if the system has to update only the software of nodes located in one part of the network. To handle this, a feature is needed to define which nodes will receive the data.

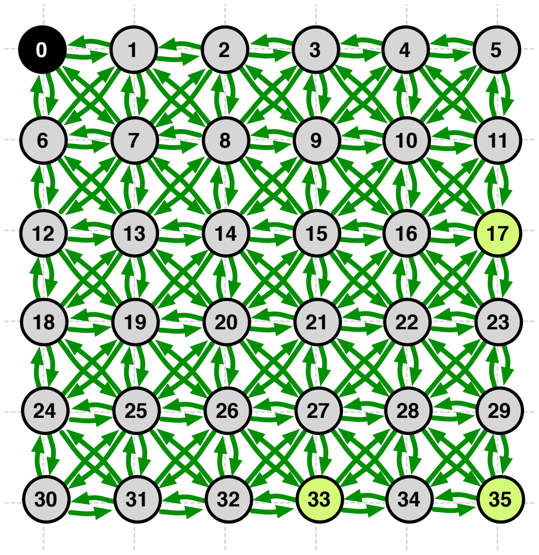


Figure 2.7: A broadcast transmission.

- In an ad hoc network, a host might rebroadcast the message upon receiving a broadcast message for the first time, if the network is not covered by other nodes. This problem or a bad behaviour of a node may arise on a high load of redundant broadcasts, heavy contention and large numbers of collisions. It is called broadcast storm[21] and can cause inoperability of the network. For instance, some hosts may experience starvation or in the case of wireless sensor nodes, run out of battery. There are some techniques to reduce this effect, but they can be unfeasible to the network due to nodes limitations and required storage space.

Multicast

Multicast is a communication scheme (shown in Figure 2.8) used to transmit data from one node to many recipients. This type of communication is an efficient way to disseminate data to a determined group of receivers interested in the transmission.

As well as broadcast communication, with multicast it is possible to transmit data from one to many receivers. On the other hand, unicast and multicast enable the sender to identify who is going to receive the messages.

On the Internet, the multicast paradigm has been implemented in the form of IP Multicast. Interested receivers send an Internet Group Management Protocol[22] (IGMP) group join message, the routers process these messages according to the IP Multicast Routing protocol used (PIM-SM[23], PIM-DM[24], etc.) and build the distribution tree among them. A sender now only sends an UDP Multicast packet to the group's address and the routers in the network then distribute the data according to the multicast tree, which has been set-up before.

Hence, multicast communication may reduce the number of transmitted packets, save energy and improve the management of Wireless Sensor Networks (WSNs).

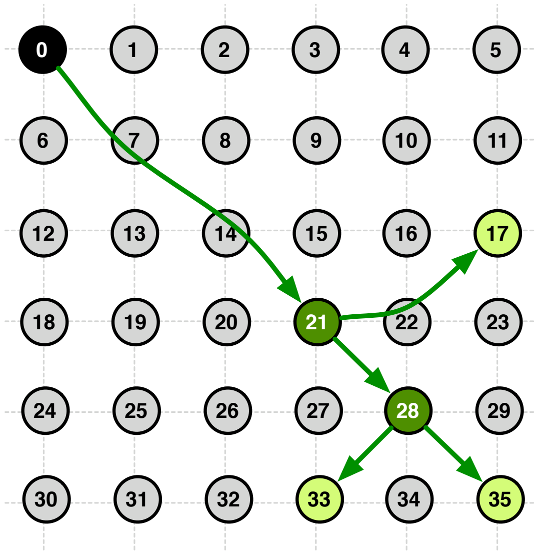


Figure 2.8: A multicast transmission.

Although IP Multicast has been available for a while on the Internet, it has not been widely deployed today due to different reasons (configuration, ISP agreements, etc). Besides this limitation in the Internet devices, the actual implementation of IP for embedded systems does not offer either support to send and receive joining messages (IGMP) or to receive non-local multicast packets.

Overlay Multicast

Overlay Multicast is nothing more than the multicast communication scheme implemented on the application layer. This kind of application enables the use of limited hardware without support to IP Multicast. These protocols can also use routing algorithms to define which nodes will forward the data before they are delivered to the receivers.

In this approach, the node application is responsible for the multicast functionalities. Because of this, the overlay multicast also depends on the application layer to provide an efficient overlay for data transmissions across networks.

In the end, the biggest advantage offered by these implementations is the delivery reliability. Other goals, such as low power consumption and absence of redundant messages, can also be achieved depending on the protocol.

Multicast Protocols for WSNs

There have been numerous research studies conducted regarding Multicast in WSNs. Some examples are:

Very Lightweight Mobile Multicast (VLM^2)

VLM^2[25] uses a data-driven approach and fits well in mobile networks.

The main advantage of this protocol is the small size of the code and the headers used in the packets' transmissions. It basically constructs a multicast tree according to the nodes position by using flooding messages. To achieve this, the nodes must send a periodic *SUBSCRIBE* message to join a group and receive the messages from it. Each transmitted message may travel through a different path because all nodes forward the received message.

The absence of the reliability is the main drawback of this protocol in the scenario of management tasks. This means it is not possible to send a code update because some of the nodes might not receive it and this could cause bad results later. The large amount of transmitted packets (due to the flooding scheme) and the necessary memory to cache information about the received packets are also relevant negative points to be observed.

Adaptive Demand-driven Multicast Routing (ADMR)

The original version of this protocol has been developed for Mobile Ad hoc Networks (MANET), which have mobile devices as nodes. The difference is the larger amount of available memory, lower bandwidth restrictions, and less energy consumption constraints provided by those devices. Due to these aspects, ADMR[26] supports the WSNs with some adaptations.

This algorithm has a route discovery phase, when the nodes transmit flooding messages and get the network status. After this, the nodes run an algorithm and determine the costs of each path according to some variables, for example, the link quality between two nodes. Later, the nodes prune routes and get the best paths to each node in the network.

The main advantage of this algorithm is to provide the best path between the source and the receivers, which may also consider weights assigned to links between the nodes. On the other hand, the algorithm requires large amounts of memory to store the link quality of all connections and at the preparing phases the nodes spend much time flooding the network and computing the best paths. These procedures may also be done periodically and might be an expensive overhead due to the WSN limitations. ADMR is not a reliable protocol. This means it does not provide any kind of confirmation to ensure the packets' reception.

Grid Multicast Protocol (GMP)

GMP works on the geographical position of the wireless sensor nodes. In order to be energy efficient, this approach routes the data from the source node to the multicast destination using the path with the shortest geographic distance. The main assumption is that the energy consumption is related to the number of hops between a pair of nodes.

A grid shape is used to define the nodes' positions inside the network. However, in real-world scenarios the nodes may be positioned in random places and a virtual grid is constructed over them.

Positive aspects of this algorithm are:

- The consideration of hop count in order to create less overload in the network during the multicast transmissions.

- The algorithm is local. The nodes do not need to know the whole network structure.
- It does not require maintenance of the multicast routing tables in the intermediate nodes.

On the other hand, this protocol does not specify any kind of reliability to the transmissions since the focus is mainly on how to define a routing to use the IP Multicast. Therefore, it does not provide reliability using UDP transmissions and because of this it cannot be a solution for management tasks if used with IP Multicast.

Geographic Multicast Routing (GMR)

As GMP, the GMR[27] protocol uses position information of the nodes to define routing tables during multicast transmissions. The constructed routes avoid broadcast flooding and reduce power consumption by using as few nodes as possible to forward the data.

To achieve these goals, the protocol uses the "new heuristic neighbour selection". This heuristic creates a relationship between the number of selected forwarding nodes, the number of possible forwarding (adjacent) nodes, the sum of distances from source to the destinations, and the sum of distances from forwarders to the destinations.

In comparison with GMP, the simulation of the GMR protocol implementation provided better results over a variety of networking scenarios. However, the algorithm is not local and requires the sending of extra messages through the network. In conclusion, the algorithm can provide good routes to forward the data, but it does not specify how the transmissions can be done. Furthermore, it works over IP Multicast without reliable transmission and does not fit with the management's requirements.

2.3 Hardware

Currently, there are different types of wireless sensor nodes able to provide the necessary hardware platform to join sensor sub-networks. They are able to consume minimal power, and have a high data rate while communicating with each other. The hardware differences are not visible for the application layer since the necessary features can be used by the software.

An operating system that can be installed on the different architectures is the solution to provide transparent access from the application layer to the sensors, USB ports, display connection, and so on. The nodes TelosB, MicaZ, MSB, and BTNodes have similar architectures and are all supported by powerful operating systems such as Contiki (will be described in Section 2.4). In this work, the model of the wireless sensor nodes is TelosB.

As shown in Figures 2.9 and 2.10 the Tmote Sky and TelosB are equipped with:

Wireless transceiver

A Chipcon CC2420 radio is used for the wireless transmissions. It has 250kbps data rate, and 2.4GHz radio. Moreover, it is IEEE 802.15.4 compliant and provides reliable wireless communication.

Microcontroller

The microcontroller is a MSP430 Texas Instruments with 8MHz clock, 10kB of RAM,

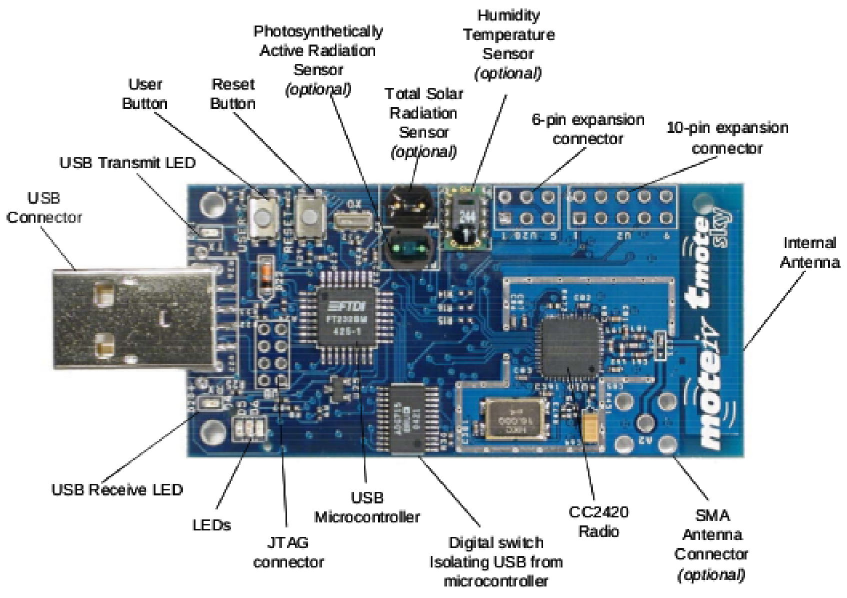

Photosynthetically Active Radiation Sensor (optional)
Humidity Temperature Sensor (optional)
User Button
Reset Button
Total Solar Radiation Sensor (optional)
6-pin expansion connector
10-pin expansion connector
USB Transmit LED
USB Connector
Internal Antenna
USB Receive LED
LEDs
JTAG connector
USB Microcontroller
Digital switch Isolating USB from microcontroller
CC2420 Radio
SMA Antenna Connector (optional)

Figure 2.9: Tmote Sky module description. [2]

and 48kB of flash memory. The 16-bit RISC processor features programming capabilities and extremely low active and sleep current consumption: It stays on sleep mode for the majority of the time, wakes up as fast as possible to process, then returns to sleep mode again.

Integrated on-board antenna

The internal antenna may attain 50-meter range indoors and 125-meter range outdoors.

Integrated humidity, temperature, and light sensors

The sensors are optional. If present, they may be directly mounted on the node module and have low power consumption.

USB connector

In order to provide a programming and data collection interface, there is an USB connector on the nodes.

2.4 Contiki Operating System

Contiki[28] is a small size operating system designed specially to fit with the wireless sensor nodes, better meeting the physical constraints and environment interaction requirements. Moreover, in order to integrate WSNs with IP networks, Contiki provides IP connectivity in a compact version of the Internet Protocol, called μIP[12].

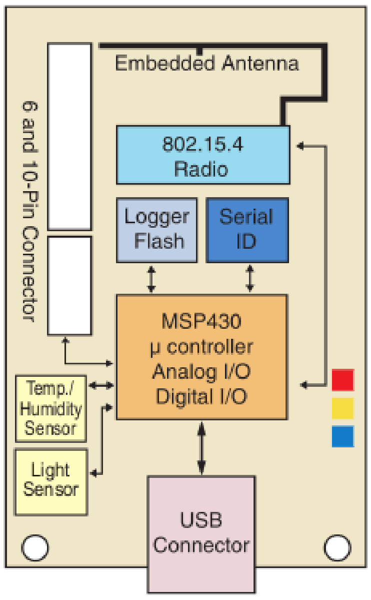


Figure 2.10: TelosB block diagram. [3]

2.4.1 Characteristics

The strengths can be summarized in:

- **Presence of multi-thread processing**, which is implemented using a hybrid model to handle the processes.
- **An event-driven kernel** where pre-emptive multi-threading uses an application library, which is optionally linked with programs that explicitly require it.
- **A set of libraries**, which can be loaded to the devices' memory according to the application requisites.
- **Only one stack to buffer the data** in the memory management

Computers can directly interact with Contiki nodes using a Web browser, UDP and TCP transmissions, or a text-based shell interface over serial line. The text-based shell provides special commands for wireless sensor network interaction and looks like a standard shell command. This module can be removed from the image in order to reduce memory consumption.

2.4.2 Protocol Stack

Since the requisites of the WSNs and the hardware architecture of the wireless sensor nodes are unlike those in typical networks, the protocol stack of the nodes usually does not have the same structure. Figure 2.11 shows the Contiki's implementation stack protocol, where the lowest

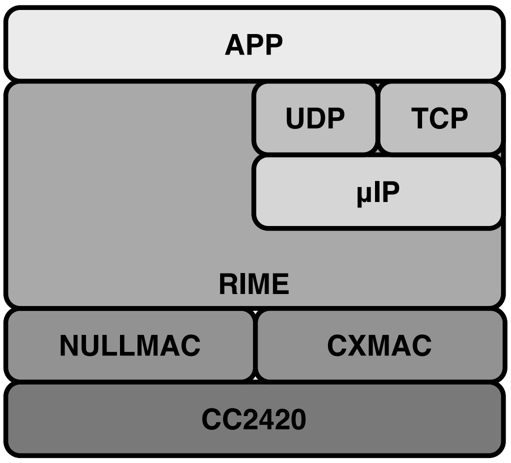


Figure 2.11: The protocol stack in Contiki.

levels are responsible for handling the radio connections, and due to the hardware limitations, a simplified version of the Internet Protocol called μIP is used.

The μIP layer implements the basic operations and supports the most common transport protocols: *UDP* and *TCP*. Furthermore, with some small changes it is possible to extend the implementation according to the needs of the application.

The communication is done by the Rime[29] stack. Rime is a radio networking stack implemented in Contiki and it is responsible for providing the necessary connection between two sensor nodes. At this layer level, Contiki is able to handle protocols such as reliable data collection, best-effort network flooding, multi-hop bulk data transfer and data dissemination.

The μIP packets rely on the Rime implementation and are tunnelled over multi-hop routing.

μIP[12] is an embedded version of the IP and can be used with 8- and 16-bit microcontrollers, such as TelosB. Its architecture was specially developed to handle the hardware constraints of the wireless sensor nodes. Thus, it has small code size and low memory consumption. The low available memory space also forced the code to be tightly coupled, removing the clear separation between two different layers.

Some Internet protocols are available on μIP such as ARP[18], SLIP[30], IP[11], UDP[7], ICMP[31] and TCP[8]. Despite the low memory availability, the TCP/IP implementation attends all RFC requirements affected by host-to-host communication and supports flow control, fragment reassembly, and retransmission time-out estimation.

By having use of this set of protocols, useful applications such as web servers, web clients, SMTP clients, Telnet servers and DNS hostname resolvers can be deployed on the wireless sensor nodes. Hence, common devices can communicate with them because of the IP compatibility provided by μIP.

Finally, the implementation supports broadcast communication as well as transmitting multicast packets with some restrictions: It is neither possible to send *join messages* to multicast groups (IGMP) nor to receive non-local multicast packets.

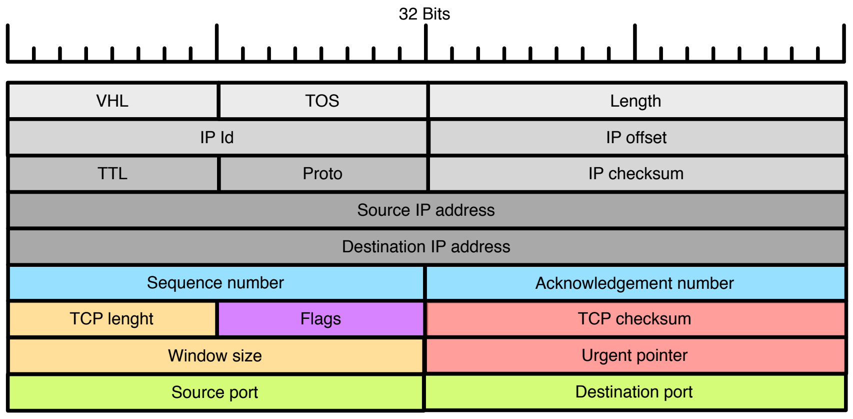


Figure 2.12: TCP and μIP headers of the Contiki's implementation.

2.4.3 Transport Protocols

TCP

Contiki offers an implementation of the Transmission Control Protocol (TCP) over the μIP (IP-equivalent) layer. The main feature of the TCP connections is to be reliable. This brings some extra costs to the application because of the number of control messages and extra bytes located on their headers to check the integrity of the data, for example.

As shown in Figure 2.12, there are 16 control bytes and they are used as a congestion controller, to identify the participants of the connection, to check the fragmentation and to detect transmission errors. On the other hand, there are some limitations due to the constraints of available space and some mechanisms to deliver the packets to the application are not implemented, for instance, the soft error reporting mechanism and dynamically configurable type-of-service bits for TCP connections[16]. After some research, the authors from Contiki discovered that usually these functionalities are not used by the applications and this would not cause a big impact in the real world[12].

UDP

The simplicity of the UDP[7] turns it into a good option on embedded systems. The UDP implementation in Contiki has basically the same header of the Internet standards as shown in Figure 2.13: It includes the source and destination ports, the checksum and 16 bits to store the package length.

As discussed before, UDP is connectionless and does not transmit extra packets as TCP does. Thus, UDP was chosen to develop this project because of the following reasons:

- The reliability and fragmentation controls are done by SNOMC on the application layer, it is not necessarily a redundant reliability in the lower layers.
- Its header is smaller. Hence, there is more space to transmit the content and it is possible to avoid unnecessary traffic by appending more bytes of content on each transmission.

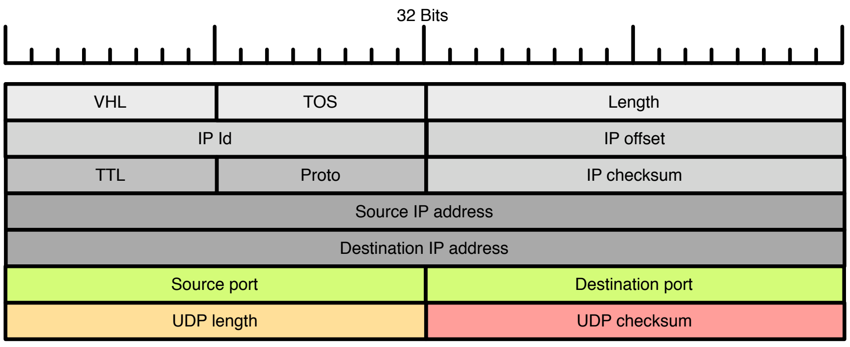


Figure 2.13: UDP and μIP headers of the Contiki's implementation.

Chapter 3

Sensor Nodes Overlay Multicast Communication (SNOMC)

As discussed previously, reliability and low energy consumption are the most important requirements on management tasks such as code updates. In these scenarios, multicast communication appears to be the solution that best fits to the WSNs' constraints. Sensor Nodes Overlay Multicast Communication[13] (SNOMC) protocol proposes a solution for these problems. The main objective is to provide a reliable multicast communication based on Overlay Multicast instead of IP Multicast.

In the subsequent sections details concerning this protocol are explained and discussed.

3.1 Node Roles

In order to accomplish the requirements, nodes have different roles inside a SNOMC transmission (see Figure 3.1). In this section each role is described along with the nodes' behaviour during the transmission.

The *source node* is unique in a group. It is responsible for all administrative tasks: to start the group, cache the whole content, transmit the data and finish the group after the user request.

Usually, more than one node receives the data and transmits it to the application by SNOMC. These are called *receiving nodes*. They must have enough space to receive the data so it is available for the application after the end of the transmission.

Forwarding nodes are responsible for receiving data and forwarding them to their next hop in the group. They operate as a bridge between two nodes and can also cache data to reduce the number of end-to-end retransmissions in case of packet losses. This mechanism depends on the implemented model (see Section 3.4).

According to the routing table, a node may have more than one neighbour, which will forward the content. These type of nodes are called *branching nodes* and they can replicate the received data and transmit them to two or more nodes.

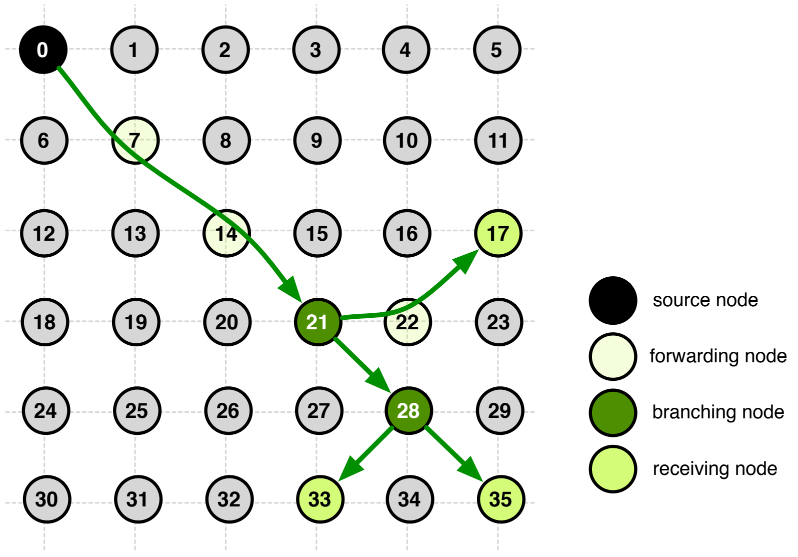


Figure 3.1: Nodes have different roles in SNOMC.

3.2 Definitions

SNOMC uses specific definitions in its description. Detailed explanations of these terms are given in this section.

A *group* is composed by one *source node* and at least one *receiving node*. Each *group* has a unique number identification and the *source node* has a list with the identification of the nodes that will receive the transmissions.

The smallest pieces of data transmitted to the nodes are the *fragments*. They may not contain many bytes due to the size limit of the transmissions done by the nodes. For example, each radio transmission supports a maximum data payload of 128 bytes and some of these bytes are used in the header sections of the lower layers protocols.

A *content* is a set of data without fixed length that is going to be transmitted to the *receiving nodes*. If the *content* has a large amount of data, this sequence will be split into small *fragments* to be transmitted in a single UDP packet. During the transmission phase (see Section 3.6.2), the *source node* may transmit one or more *contents* to the nodes but each *content* transmission cannot start before the successful delivery of older transmissions in the same *group*.

SNOMC does not use positive acknowledgments for each transmitted fragment (see Section 3.3). The detection of packet losses is done as follows: After receiving a *fragment*, the node starts a timer. This timer expires after an interval greater than the time required to receive all transmitted *fragments*. Thus, if there is one or more missing *fragments* in the transmission, the node will detect them after a while, before transmitting a *negative acknowledgment* to the source. This waiting interval is called *negative time-out*.

If all *fragments* from a *content* are missed, the node cannot take notice concerning the transmission. Consequently, the source will never receive a response notifying it of the losses. To avoid this problem, sender nodes start a timer called *content time-out*, after the last *fragment* transmission.

The interval of the *content time-out* has to be larger than the time elapsed while the system transmits the whole content to all nodes, including the *fragments* retransmission. Failing this, unnecessary and expensive retransmissions will occur.

If the *content time-out* expires, the whole *content* is retransmitted. As soon as the node has received feedbacks (either a *negative acknowledgment* or a *content acknowledgment*) from each neighbour, this timer can be stopped.

3.3 Message Types

In order to provide reliability and fragmentation, SNOMC uses special types of messages to control the data flow over the nodes. The subsequent sections detail the range of message types used by SNOMC.

3.3.1 Default Messages

Default messages are used with the standard operations, to start and finish a group and to carry the content, for example. The direction of their transmission is always from the source to the receivers.

The *start group* message is used to notify the nodes about the group existence. In the source-driven mode, this is the most complex message type because it contains identification from every node reachable by the node that is receiving it. Hence, the sender must generate different *start group* messages to each neighbour. The content is filled after the node checks the routing table and is sure about the reception of the message by one neighbour.

On the other hand, in the receiver-driven mode, the *start group* message is very simple and has only the group number and the source identification to notify the nodes about the transmission. The nodes that want to receive the transmission must join the group afterwards by sending a *join group* message with its identification and the group number to the *source node*.

Each *data* message carries one *fragment* from the *content*. In its header some information such as content identification, number of fragments in the whole content and fragment number are described. All *fragments* are transmitted using the same message format even when a *fragment* is being retransmitted.

At the end of the content transmission, the *source node* may close the group with a single message. The *finish group* message will notify all participants and they will not receive any further data from the respective transmission.

3.3.2 Notification Messages

Usually, notifications are transmitted in the opposite direction from the data. They are used to control whether the nodes have to retransmit packets or not.

A *content acknowledgment* is used to notify the sender about the successful delivery of the content to the neighbours in one direction. Since the node has received a *content acknowledgment* from each neighbour, a new one is generated and transmitted to the relative source node.

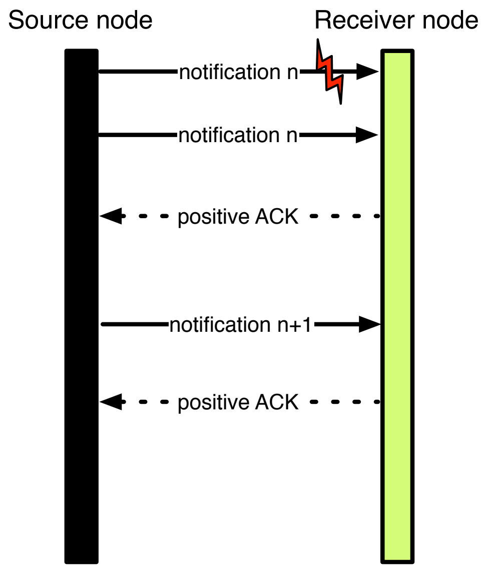


Figure 3.2: Transmission of *notification messages* using *positive acknowledgements*.

If a *receiving node* has no neighbours to which to forward the data, a *content acknowledgement* is generated at the end of the content reception. After the transmission of the *content acknowledgement*, the memory is released, if the node has any data on its cache.

Start group acknowledgment is very similar to the *content acknowledgment* but it is used to notify the nodes about the end of the *early phase* (see Section 3.6.2). Every *receiving node* must transmit a *start group acknowledgment*. Since the source node has received *start group acknowledgments* from all *receiving nodes*, it can be sure that all nodes were notified about the transmission and then start the data transmission.

The *notification messages* are retransmitted until the node receives a *position acknowledgement* from its neighbours, as shown in Figure 3.2.

3.3.3 Acknowledgements

As described before, the *positive acknowledgments* are not sent to confirm every received *fragment*. They are only used to inform about the reception of *notification messages*. As they are crucial to keeping the system working, these messages have higher priority and are transmitted before any other message type placed in the transmission queue. Since the node has received a *positive acknowledgment* from its neighbour, it stops to retransmit the *notification message*.

SNOMC uses *negative acknowledgments* to notify the nodes about packet losses. It carries the missing fragment numbers, in order to avoid the retransmission of the entire content.

3.4 Design Models

As multicast protocols can be used in numerous scenarios, the options of design models depend on the resources availability, environment, data relevance and others. These decisions must be done according to the applications functionalities and hardware constraints.

This section describes different design models, which can be combined according to the transmission requirements.

3.4.1 Receiver-driven vs. Source-driven Mode

There are two approaches to designing multicast communication. One of them is called receiver-driven because during the start up phase the *receiving nodes* must inform the *source node* about the desire for receiving the transmitted data. To do this, nodes use control messages to inform about group joining and data availability.

The second approach is called source-driven. In this approach, the *source node* must have extra storage space to load a list with all *receiving nodes*. This list can be set before the beginning of the transmission with an user interaction or in a configuration file, for example. Although the *source node* stores which neighbours must receive its transmissions in both approaches, only in the source-driven mode all *receiving nodes* must be known before the start of the transmission.

So far, the source-driven approach has been chosen in order to fulfil the use cases requirements. Both in software updates and in configuration tasks, the *source node* has knowledge about which nodes will receive the *contents* as well as the control about adding and removing new receivers. For future works, the receiver-driven approach can be considered.

3.4.2 Caching Scheme

Depending on the network characteristics, nodes may have less space to cache data. This difference depends mainly on the size of the WSN and on the physical limitations of the nodes such as memory size.

Caching only in the Source Node

This is the simplest implementation of the algorithm and uses less resources in the non-receiving nodes. The data is cached only in the *source node* until it has been successfully delivered to all *receiving nodes*. Moreover, the primary objective is to avoid unnecessary memory allocation at the intermediate nodes. The cache policy works as follows:

- **In the source node** all data is cached until it has been successfully delivered to all *receiving nodes*, because this node is responsible for retransmitting them as many times as needed.

- **In forwarding nodes** each packet is only forwarded. Hence, only one packet can be buffered during a short time interval between its reception and the respective forwarding.

- **In branching nodes**, similar to the rule above, packets are only forwarded and not cached. The difference is the time interval size while the packet stays at the node. It is larger because these nodes have to forward the data to multiple nodes. After the transmission, the packet is not stored in the memory any more.

- **In receiving nodes** received data is cached. It requires a buffer equal or greater than the transmitted *content*. With this, SNOMC assures the delivery of the whole *content* to the application with no fragmentation.

Forwarding and *branching nodes* do not consume space to cache the whole transmitted data. Consequently, packet losses have high costs to the system because retransmitted messages must travel among the whole path between the *source* and *receiving nodes*.

Caching in Every Node

This approach caches all the data in every intermediate node. Memory is released as soon as the node has received confirmation from its neighbours, which have received the data successfully. One drawback of this approach is that it buffers the content in every node used in the transmission, even if they are not participating in that group. However, because the *fragments* are cached on every node, a packet loss can be detected earlier than in other cases, resulting in a faster transmission.

Caching in Branching Nodes

Caching in branching nodes is an intermediate solution between the two prior policies. The idea is to cache the data only in *branching nodes* because they must forward the data to more than one neighbour and packet losses have a bigger impact on the network. Furthermore, there are at least two nodes to notify a *branching node* about packet losses and this can overload the network and consume more battery than the necessary.

3.4.3 Transmission Scheme

Using the *caching in every node* policy, there are two ways to transmit the data through the intermediate nodes. The main difference is that in one approach only the *receiving nodes* may transmit *negative acknowledgements* and in the other one any node can do it.

Moreover, both schemes require each node to know the number of hops to the farthest one.

First Forward

In this approach, after receiving a *fragment* a node caches it and forwards to its neighbours. Thus, the intermediate nodes do not detect packet losses and consequently they never generate *negative acknowledgements*. It could result in a large number of messages being transmitted at the same time through the nodes.

All *negative acknowledgements* are generated by the *receiving nodes*. After receiving a *negative acknowledgement*, an intermediate node first checks if the *fragment* is cached in it. If yes, the node retransmits the fragment. Otherwise, it forwards the *negative acknowledgement*.

The main drawback is the occasional propagation of the packet loss to all descendant nodes.

type	time_stamp	group_id	
1 byte	2 byte	1 byte	
frag_no	frag_length	no_of_frags	content_id
1 byte	1 byte	1 byte	2 byte
data			
SNOMC_PAYLOAD_SIZE			

(a) *Data* message type in SNOMC.

type	time_stamp	group_id
1 byte	2 byte	1 byte
source_id	deep_back_size	receiver_list_size
2 byte	1 byte	1 byte
receiver_list		
LIST_LENGTH		

(b) *Start group* message type in SNOMC.

type	time_stamp	group_id
1 byte	2 byte	1 byte
source_id		
2 byte		

(c) *Finish group* message type in SNOMC.

type	time_stamp	group_id
1 byte	2 byte	1 byte
source_id	content_id	
2 byte	2 byte	

(d) *Content acknowledgement* message type in SNOMC.

type	time_stamp	group_id
1 byte	2 byte	1 byte
time_stamp_ack		
2 byte		

(e) *Positive acknowledgement* message type in SNOMC.

type	time_stamp	group_id
1 byte	2 byte	1 byte
group_id	deep_size	
1 byte	1 byte	
combine_pack	timestamp_ack	
1 byte	2 byte	

(f) *Start acknowledgement* message type in SNOMC.

Figure 3.3: Message types in SNOMC.

First Check

In order to improve the scheme described above, the nodes can check the fragments' reception before forwarding them. With this, a packet loss can be detected before the content is transmitted to the next nodes and the respective *negative acknowledgements* are generated even by the intermediate nodes.

A node starts to forward a *content* only if all *fragments* are cached in it. This reduces the number of transmitted messages through the network because there is no error propagation. On the other hand, this approach may increase the time to transmit all *fragments* if no packet is lost.

3.5 Data structures

Basically, SNOMC uses seven different message types as described in Section 3.3. The implemented structures share a common header with the type of the message, a timestamp and the group identification. The message types are as shown in Figure 3.3.

In the *receiving nodes*, *start acknowledgements* are transmitted after the *start group* message has been positively acknowledged. Moreover, each *receiving node* does not have any information about other nodes that will transmit and receive messages from the same sender. Because of this, the probability of collision caused by two transmissions at the same time is very high. Thus, in order to avoid packet losses, the implementation can combine two message types and reduce the network load to produce better results.

3.6 The SNOMC Algorithm

This section describes the SNOMC algorithm behaviour. Expected configurations and working phases are detailed with diagrams illustrating possible states and interactions between the nodes in a group.

3.6.1 Assumptions

Some configurations and constants are meant to be available such as the groups' topology, routing tables and connections with the neighbours.

In the source-driven approach, the *source node* must know which nodes will be part of a group before it starts the transmission. As soon as it has started this group, no more participants can join the transmission.

SNOMC is not aware of the routing algorithm. However, a routing table must be defined before the beginning of the transmission and it does not change until the group transmissions are finished. This means the routes are static and the nodes can store their relative positions inside the group; The *source node* is the one which is transmitting data to the group. The group members are relative on each node: They are the neighbours that are going to receive the transmission from that node. Occasionally, a node may receive the data only to forward to the *receiving nodes*.

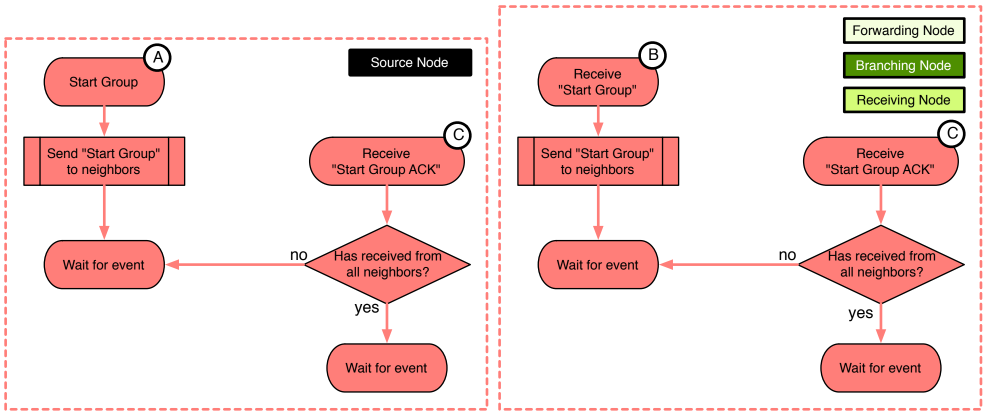
If one or more nodes are not reachable before the beginning of the transmission, to provide a reliable transmission the nodes will continue attempting to transmit until the other nodes confirm delivery.

3.6.2 Phases

SNOMC is composed of three phases: *early phase*, *transmission phase* and *final phase*. Each one is clearly defined by a set of possible states, defined as pre and post conditions and are described here. In order to illustrate the procedures letters are used in the processes' descriptions and mapped into the flowcharts.

Early Phase

The *early phase* (see Figure 3.5, Figure 3.4) happens whenever the user decides to build a group and notifies the *source node*. In this phase no content is transmitted, but the participant nodes are notified about the group and their respective function.


Source Node
Start Group
A
Send "Start Group" to neighbors
Wait for event
Receive "Start Group ACK"
C
Has received from all neighbors?
no
yes
Wait for event
Forwarding Node
Branching Node
Receiving Node
Receive "Start Group"
B
Send "Start Group" to neighbors
Wait for event
Receive "Start Group ACK"
C
Has received from all neighbors?
no
yes
Wait for event

(a) Flowchart in the source node. (b) Flowchart in other nodes.

Figure 3.4: Flowcharts of the SNOMC's *early phase*.

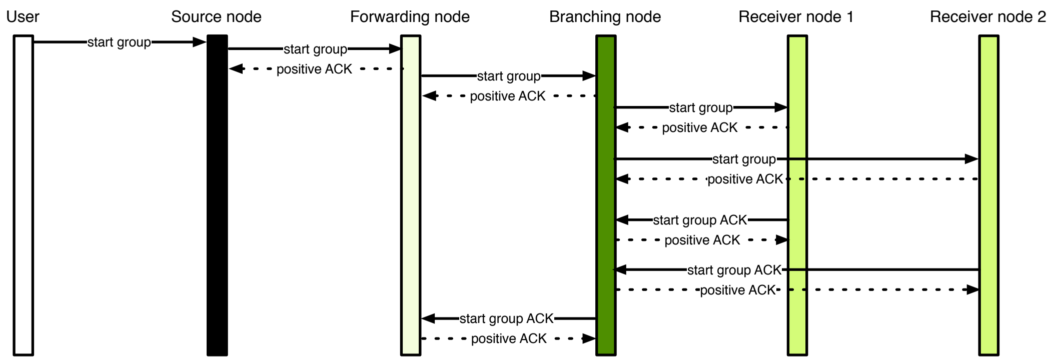


Figure 3.5: Sequence diagram of the *early phase*.

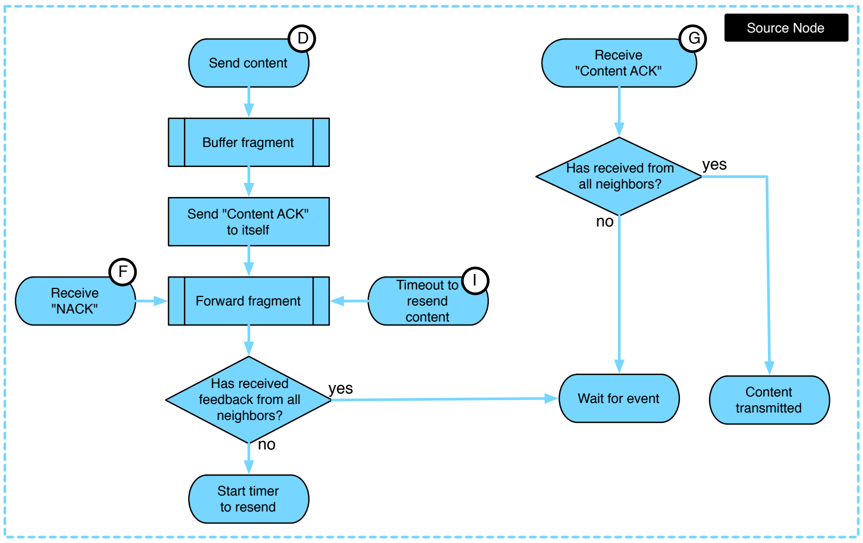


Figure 3.6: Flowchart of the *transmission phase* in the *source node*.

Source Node

After receiving the user request [**A**], it will prepare the *start messages* and transmit to the respective neighbours. Each neighbour must confirm the delivery with a *positive acknowledgment*, read the content and transmit new *start group* messages to its neighbours, if applicable.

Other Nodes

If a node receives the *start group* message [**B**] and does not have to forward it to any other node, it transmits a *start group acknowledgment* back to the sender. Otherwise, the node will wait to receive confirmation from each neighbour [**C**] and transmit only one *start group acknowledgment* back to the relative sender in order to confirm that all nodes in that direction have already been notified about the transmission.

This procedure goes forward until the *source node* receives confirmation from each neighbour and finishes the *early phase*. Then it will be ready for the transmission of the content.

Transmission Phase

In the *transmission phase*, the *source node* transmits the *content* to the nodes, which have already received a *start message* during the previous phase and are ready to receive the data (see Figures 3.6, 3.7, 3.8 and 3.9).

Finally, the *transmission phase* can be used to transmit as many *contents* as the user wants. Since a *content* has been successfully transmitted, the source may start transmitting a new one.

Source Node

After receiving the *content* from the application [**D**] (see Figure 3.6), the *source node* will

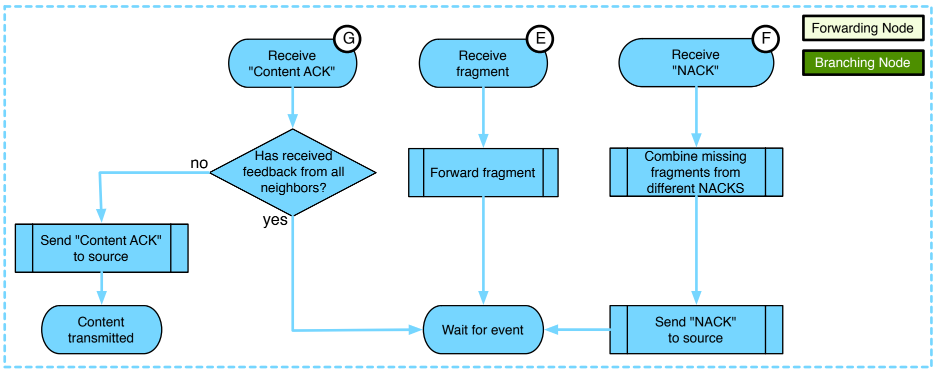
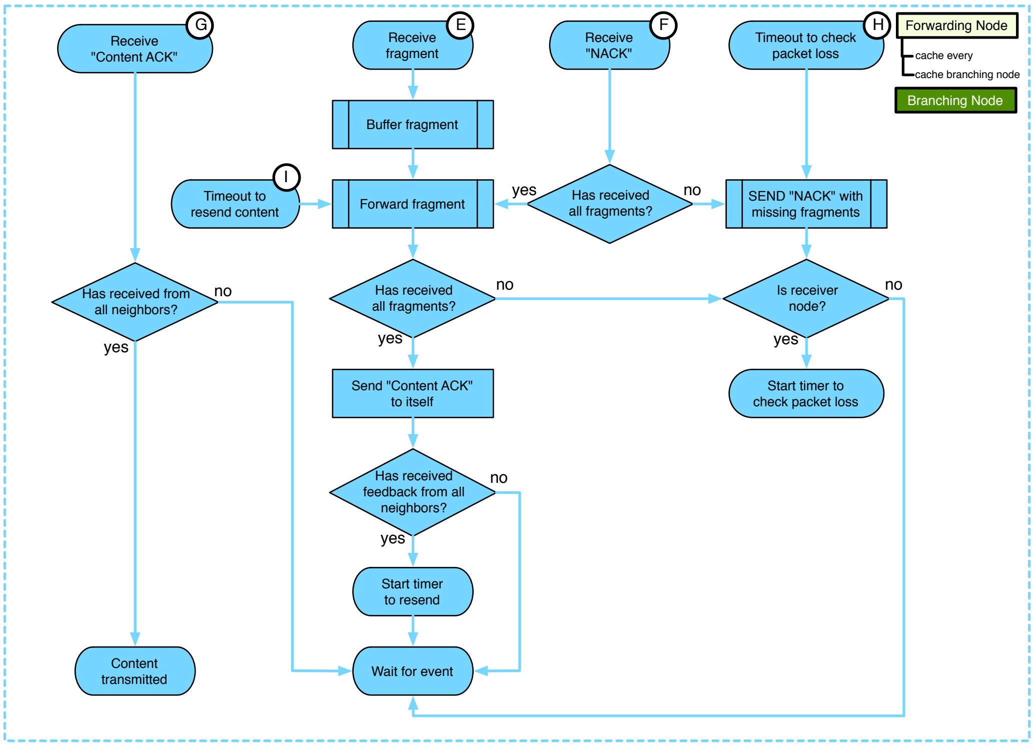


Figure 3.7: Flowchart of the *transmission phase* in nodes that do not cache any *fragment*.



Receive "Content ACK"
G
Receive fragment
E
Receive "NACK"
F
Timeout to check packet loss
H
Forwarding Node
cache every
cache branching node
Branching Node
Buffer fragment
Timeout to resend content
I
Forward fragment
yes
Has received all fragments?
no
SEND "NACK" with missing fragments
Has received from all neighbors?
no
yes
Has received all fragments?
no
yes
Is receiver node?
no
yes
Send "Content ACK" to itself
Start timer to check packet loss
Has received feedback from all neighbors?
no
yes
Start timer to resend
Content transmitted
Wait for event

Figure 3.8: Flowchart of the *transmission phase* caching the fragments and using the *first forward* approach.

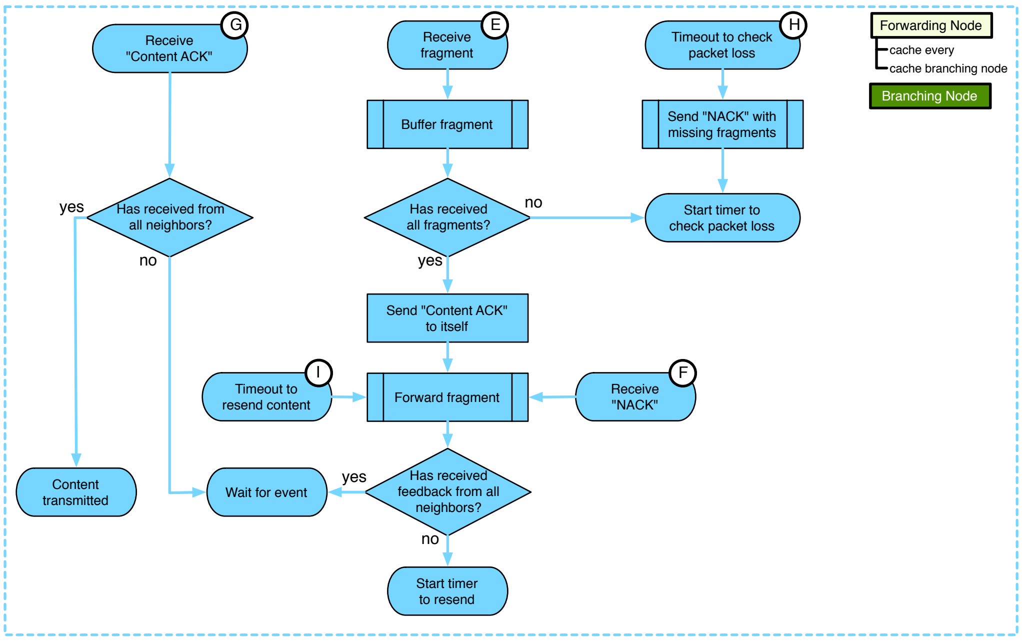

Receive "Content ACK"
G
Receive fragment
E
Timeout to check packet loss
H
Forwarding Node
cache every
cache branching node
Branching Node
Buffer fragment
Send "NACK" with missing fragments
yes
Has received from all neighbors?
no
Has received all fragments?
no
Start timer to check packet loss
yes
Send "Content ACK" to itself
Timeout to resend content
I
Forward fragment
Receive "NACK"
F
Content transmitted
Wait for event
yes
Has received feedback from all neighbors?
no
Start timer to resend

Figure 3.9: Flowchart of the *transmission phase* caching the fragments and using the *first check* approach.

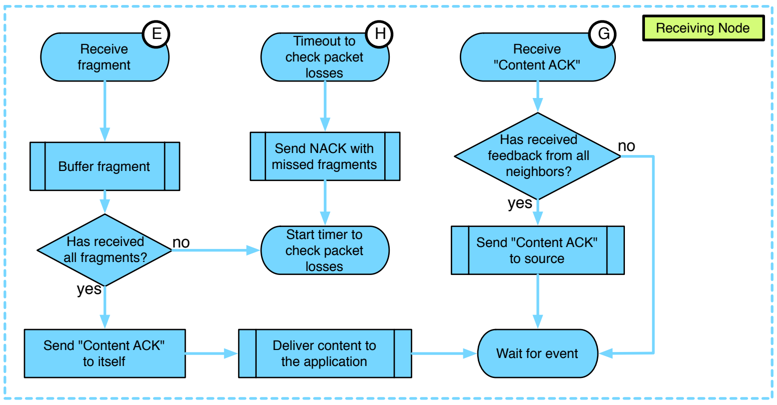

Receiving Node
Receive fragment
E
Timeout to check packet losses
H
Receive "Content ACK"
G
Buffer fragment
Send NACK with missed fragments
Has received feedback from all neighbors?
no
yes
Has received all fragments?
no
Start timer to check packet losses
Send "Content ACK" to source
yes
Send "Content ACK" to itself
Deliver content to the application
Wait for event

Figure 3.10: Flowchart of the *transmission phase* in the *receiving node*.

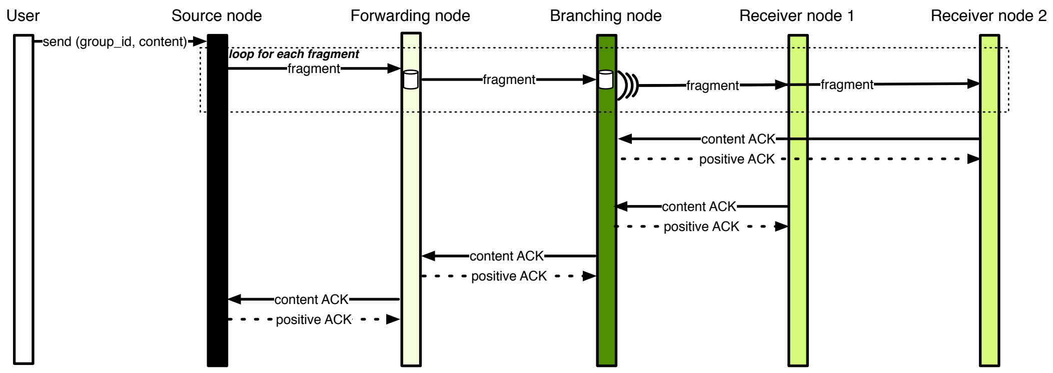


Figure 3.11: Sequence diagram of the *transmission phase* first forwarding the *fragments*.

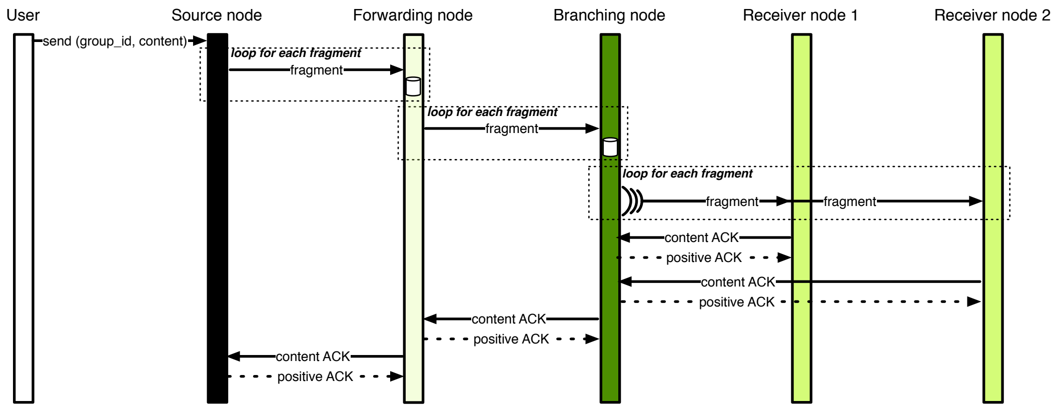


Figure 3.12: Sequence diagram of the *transmission phase* first checking the *fragments*.

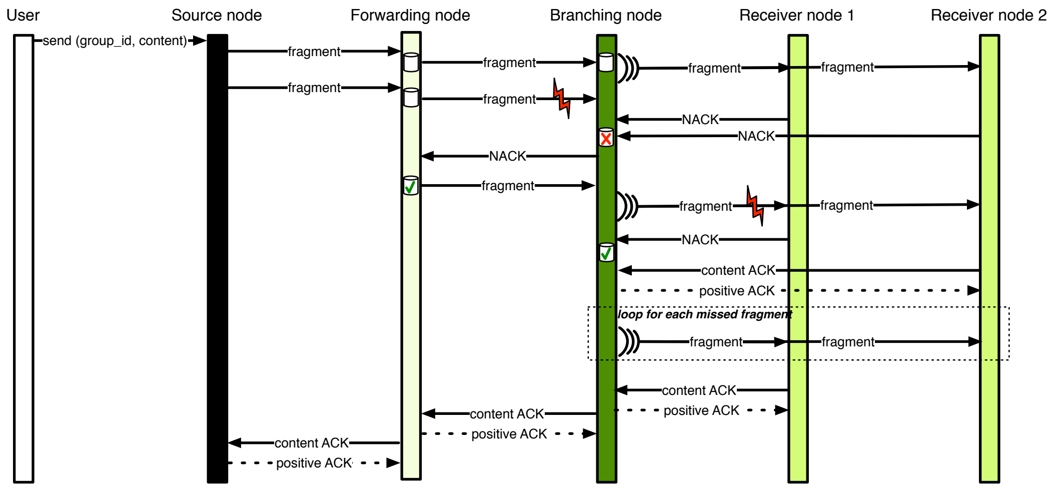


Figure 3.13: Sequence diagram of the *transmission phase* first forwarding the *fragments*.

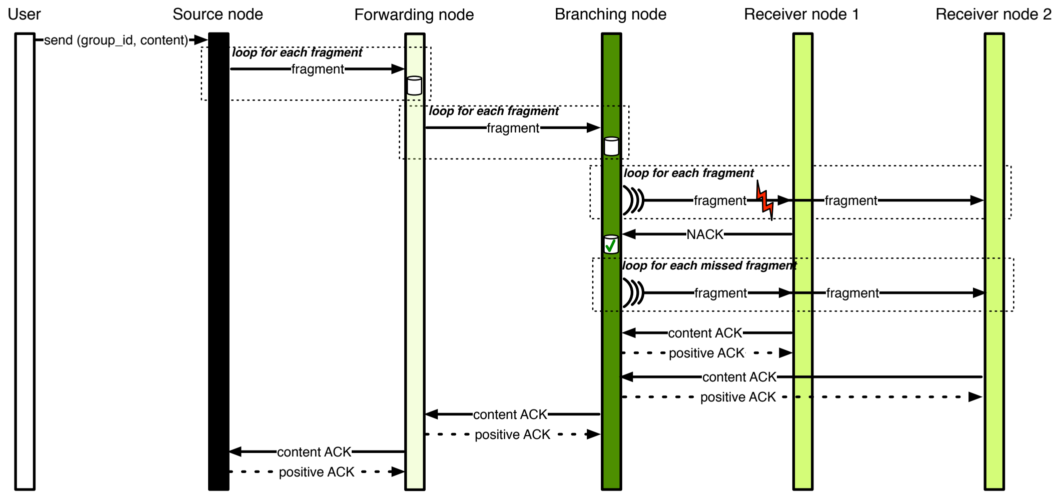


Figure 3.14: Sequence diagram of the *transmission phase* first checking the *fragments*.

split it into small *fragments* and transmit them. The transmission procedure works as follows: after fetching the next *fragment*, the *source node* transmits it to all neighbours with a delay between two transmissions (see Section 4.1.4). This procedure repeats for all *fragments* of the *content* and the interval depends on the used cache policy and transmission scheme.

After the transmission, it waits either for a *negative acknowledgment* to retransmit missing fragments or a *content acknowledgment* to finish the transmission. If none of these are received the source retransmits the whole content again after the *content time-out* has expired [**I**].

When an application tries to transmit data, SNOMC may return a negative answer considering the available space to cache the data.

Forwarding Nodes

The *forwarding nodes* have different behaviours according to the implemented approach:

- *Cache in all nodes* with *first check* approach - a node does not start forwarding the data before it is sure about the successful reception of all fragments transmitted so far [**E**], which implies the use of *negative acknowledgements* to detect packet losses (see Figures 3.9, 3.12 and 3.14).
- Other approaches - as soon as the node has received the *fragment* [**E**], it caches the data (if necessary) and forwards it to the neighbours. Thus, this node does not generate *negative acknowledgements*. It combines the received *negative acknowledgements* and forwards only one message to the source, if the missing fragments cannot be found in its cache (see Figures 3.8, 3.11 and 3.13).

If the *forwarding node* caches the data (according to the caching policy):

- It caches every *fragment* from a *content* until a *content acknowledgement* has been received from all neighbours.
- It has to ensure that its neighbour will receive the content. Thus, after the transmission of the last *fragment*, it waits for a feedback from the node which has received the data (it can be either a *negative acknowledgment* or a *content acknowledgment*) to finish the transmission and send a *content acknowledgement* to the sender. If none of these are received, the node retransmits the *content* again after the *content time-out* has expired [**I**].

Occasionally, the *forwarding node* can have no available space to store the info about the *fragment*. Thus, it drops the packet and the sender will detect it as a packet loss and will do its retransmission afterwards.

Branching Nodes

As well as the *forwarding nodes*, the behaviour of the *branching nodes* depends on the implementation:

- *Cache in all nodes* with *first forward* approach and *cache only in the source node* - the node receives every fragment [**E**]. If necessary, it caches them before forwarding to the neighbours. It does not generate *negative acknowledgements* to notify the source about packet losses. Therefore, it combines received notifications and forwards them to the source (see Figures 3.8, 3.11 and 3.13).
- Other approaches - the node does not forward the *fragments* before checking if the whole content has been received [**E**]. Then, if there are missing fragments it will generate *negative acknowledgements* to notify the source about the packet losses and retransmit the whole content to the neighbours if no feedback is received (see Figures 3.9, 3.12 and 3.14).

If the *branching node* has to cache the data:

- Every *fragment* from a *content* is cached until a *content acknowledgement* has been received from all neighbours.
- The node waits for either a *negative acknowledgment* or a *content acknowledgment* from its neighbours to finish the transmission and send a *content acknowledgement* to the sender. If no feedback has been received, the node retransmits all *fragments* after the *content time-out* has expired [**I**].

An improvement in the *branching nodes* is to use broadcast messages to transmit the fragments to their neighbours. Since the nodes already know about the group's existence, they can filter the messages that they want to receive. This simple change can avoid many transmissions over the network during the procedure. The packet retransmission after a *negative acknowledgment* can also use broadcast communication in order to avoid the transmission of the same fragments to different nodes.

Occasionally, as in the *forwarding nodes*, a *branching node* can have no available space to store the info about the *fragment*. Thus, it drops the packet and the sender will detect it as a packet loss and will do its retransmission afterwards.

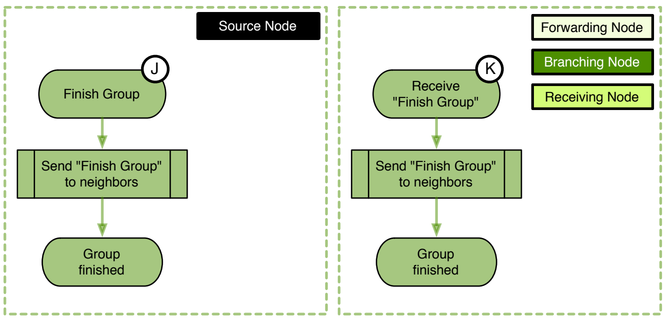
Receiving Nodes

The *receiving nodes* cache the whole content until all fragments have been received. This prevents the application from receiving the *fragments* out of order (see Figure 3.10).

After receiving all *fragments* of a *content*, *receiving nodes* only transmit a *content acknowledgment* to the source and wait for new messages from it. If one or more fragments are missing, a *negative acknowledgement* is generated by the *receiving nodes* after the *negative time-out* has expired [**H**].

If the *receiving node* does not have space to cache a received fragment, it drops the packet and waits for the retransmission later.

Final Phase



(a) Flowchart in the source node. (b) Flowchart in other nodes.

Figure 3.15: Flowcharts of the SNOMC's *final phase*.

Figure 3.15 shows the *final phase*. After the *early phase*, the *source node* will receive the request to finish the group [**J**] and if a content is still being transmitted, it marks the connection to finish later. Alternatively, it transmits a *finish group* message to its neighbours [**K**].

SNOMC may not notify the application about the success of this operation.

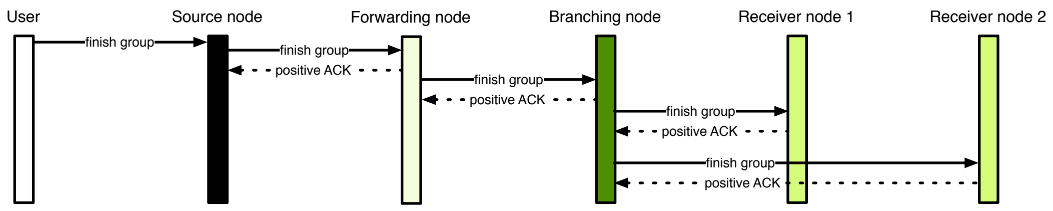


Figure 3.16: Sequence diagram of the *final phase*.

Chapter 4

Implementation

The main goal of this thesis is to implement and compare the SNOMC protocol with other solutions in a real-world environment. In order to provide essential data for this, some algorithms were implemented and tested. In this chapter, the implementations are described step-by-step starting with an improved version of SNOMC, an algorithm using UDP as transport protocol and the last one, using pure TCP from the source to the receivers.

4.1 SNOMC Implementation in Contiki

After initial observations on the first version of SNOMC[13], some improvements could be integrated according to the necessary changes. In this final implementation the protocol works on the application layer and can be used by any other application.

Figure 4.1 shows the protocol stack of this implementation, with SNOMC working on top of UDP, on the application layer. Table 4.1 compares UDP and TCP implementations in Contiki and shows what the worst problem with TCP is: In order to provide some features such as reliability and fragmentation, its header consumes more space than the UDP header (see Figures 2.12 and 2.13). Because these features are implemented in SNOMC, UDP appears to be the best

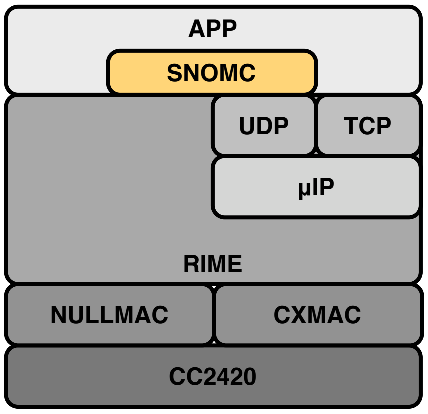


Figure 4.1: Protocol stack with SNOMC.

	UDP	**TCP**
Header length	8 bytes	16 bytes
Header length (with IP)	28 bytes	36 bytes
Reliability	no	yes
Fragmentation	no	yes
Broadcast	yes	no
Payload length	100 bytes	92 bytes

Table 4.1: Comparing UDP and TCP in Contiki.

choice: It has more space to transmit the data in addition to providing broadcast communication, which improves the scalability of the implementation.

4.1.1 Memory Allocation

Contiki provides a way to dynamically allocate memory using a fixed number of pre-defined memory blocks, but this implementation does not use it in order to avoid out of memory problem during high load periods.

Memory is allocated during the application start up and the size of the caches are pre-defined using constants and might be changed before its compilation.

Groups - The groups' users and IDs are stored in the *source node*. The implementation has a pre-defined number of available places to store this information and the *source node* loads the groups to transmit data later. The following information is used for both active and non-active groups:

- **Group ID** - 1 byte with the group identification.
- **Source ID** - 1 byte with the identification of the group owner.
- **Participants' list** - pre-defined number of bytes with the participating nodes.
- **Size** - 1 byte with the number of participants of this group.

The active groups have some extra information to be stored:

- **Is branching?** - 1 byte to identify if this is a branching node.
- **Is participant?** - 1 byte to identify if this node is a participant of this group.
- **To finish?** - 1 byte to flag the end of the group's transmissions.
- **Last content ID** - 2 bytes with the identification of the last content received.
- **Sent content ID** - 2 bytes with the identification of the last content successfully transmitted.
- **Start acknowledgements to receive** - 1 byte with the number of missing *start acknowledgements*.

- **Received start acknowledgements' list** - list of nodes that have already transmitted a *start acknowledgement* to this one. The maximum length is the number of participants in a group.
- **Clock time** - 2 bytes with the time at the beginning of the transmission. Used for statistical purposes.
- **Distance from source** - 1 byte with the distance to the *source node*.

The first 3 bytes can be combined in one flag with 1 byte in order to reduce memory consumption.

Contents - Each *content* has its own aggregated information, which is used during the transmission to control received *content acknowledgements*, for example. The implementation has two available positions. This means that two groups can transmit a message to their nodes at the same time using nodes that participate on both of them.

- **Is used?** - 1 byte to identify if this space is occupied.
- **Content ID** - 2 bytes with the identification of the *content*.
- **Group ID** - 1 byte with the identification of the group.
- **Clock time** - 2 bytes with the time at the beginning of the transmission. Used for statistical purposes.
- **Total of fragments** - 1 byte with the total number of fragments in this *content*.
- **Received fragments** - an array structure to identify which *fragments* were already received. The maximum length is the size of the fragments cache.
- **Pointer to the fragment** - 1 byte with a pointer to the *fragment* in memory.
- **Nodes list** - an array structure to store the nodes that will receive this *content*. The maximum size is the number of group participants.
- **Number of nodes** - 1 byte with the number of nodes to receive the data.
- **Negative acknowledgement's timer** - 10 bytes with a timer to transmit a *negative acknowledgement* to the *source node*.
- **Received content acknowledgements' list** - nodes that have already transmitted a *start acknowledgement* to this one.
- **Number of received content acknowledgements** - 1 byte.
- **Content timer** - 10 bytes with a timer to retransmit the data to the neighbours.
- **Has delivered to the application?** - 1 byte to identify if this *content* was already delivered to the application.
- **Is forwarded?** - 1 byte to identify if this *content* is cached on this node or just forwarded.

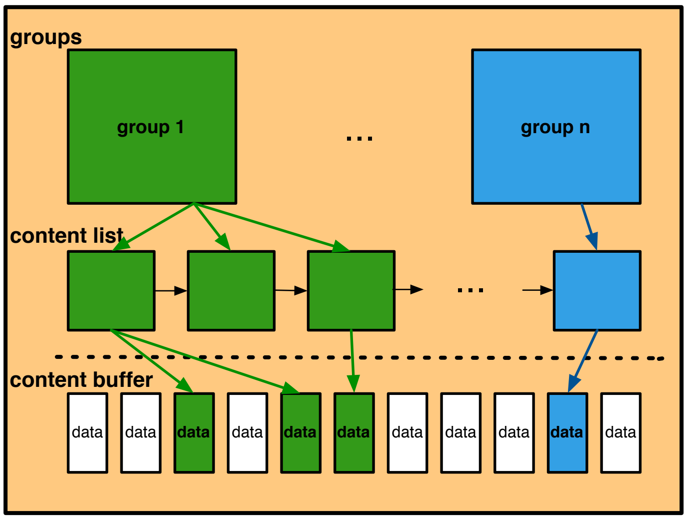


Figure 4.2: Snapshot of the memory consumption.

Data - There is an array to store all *fragments* received by SNOMC. Therefore, a *content* may occupy as many free spaces as it needs. For example, if the array has a total of 10 free spaces, the node can buffer one *content* with up to 10 *fragments*, or two *contents* that together have 10 or less *fragments*. It may be a content with 9 *fragments* and the other with only one. By doing this, a *content* may be larger than the other and both will be cached at the same time, with less risk of running out of memory. Another positive aspect of this approach is that the *fragments* do not have to be duplicated before being transmitted because any function can directly read from this cache. The size of this cache may change and the structure of each element is as follows:

- **Is used?** - 1 byte to identify if this space is occupied.
- **Length** - 1 byte to store the length of the *fragment*.
- **Content** - the *fragment*.

Figure 4.2 shows how SNOMC allocates the memory. Basically, a group can have zero or more contents to be transmitted. On each *content* it is possible to find data fragments of the message. In the case where the node does not have to cache the data, the program will have only the part above the dashed line.

4.1.2 Transmission Procedure

Section 2.4 explains that Contiki was specially developed for wireless sensor nodes. One of the improvements of this operating system is that it has an event-driven kernel. Taking advantage of this feature, the implementation uses timers and events (such as packet reception) to compute and transmit to the nodes.

Caches

The transmission procedure uses 3 different caches: one for the notification messages, another for the received *negative acknowledgements*, and a third one for the data. If the node receives a message that requires it to cache any data, the status of caches will be verified and a *positive acknowledgement* is transmitted only if there is enough space to compute a response.

For notification messages, the cache works as a list where new items are added at the end. Having done this, if two nodes transmit notification messages (such as a *content acknowledgement* and a *finish group* message) at the same time to each other a deadlock could happen. Thus, in order to avoid this problem, *positive acknowledgements* are pushed onto the beginning of the list, providing higher priority to them.

Each notification cache item has the following structure:

- **Content type** - 1 byte to set the type of the notification.
- **Recipients list** - the recipients' identifications.
- **Recipients list size** - 1 byte with the list length.
- **Next recipient** - 1 byte to flag who is the next recipient.
- **Content** - the complete notification message.
- **Was transmitted?** - 1 byte to identify if this message was already transmitted.
- **Is used?** - 1 byte to identify if this space is occupied.

The last 2 bytes can be combined in one flag with 1 byte in order to reduce memory consumption.

The nodes have a pre-defined number of spaces to cache notification messages. The larger these caches are, the fewer packets will be dropped at the reception and the network overload can be avoided.

The data cache has space to store a data packet and its content. Therefore, it also has necessary information to transmit one or more fragments to the nodes. The cache has only one element with the following structure:

- **Content ID** - 2 bytes.
- **Data Packet** - the complete data packet.
- **Recipient** - 2 bytes with the intermediate recipient, if unicast.
- **Fragments** - an array structure with the number of all fragments to be transmitted.
- **Number of Fragments** - 1 byte.
- **Next fragment** - 1 byte with the position of the next fragment.
- **Is used?** - 1 byte to identify if this space is occupied.

- **Is broadcast?** - 1 byte to identify if the transmission will use broadcast or not.

The last 2 bytes can be combined in one flag with 1 byte in order to reduce memory consumption.

When SNOMC fills this cache to transmit a *fragment*, it also sets the data packet with the next *fragment* to be transmitted. After this, the transmission procedure prepares the next *fragment* (if applicable) and waits for the respective time-out to transmit it.

Even if the node does not cache the data, it will require extra cache space for the received *negative acknowledgements*. The node stores them in a First In First Out (FIFO) list and computes them just before transmitting the content again. When the node has to read the *negative acknowledgement* and retransmit the missing fragments, it fills the data cache according to the receiver information and transmits the requested fragments using the data cache describe above.

Priorities

Each node has only one procedure that transmits the packets and a timer to control it. In order to make sure that the interval between two consecutive transmissions is large enough to avoid collisions and network congestion, the procedure is only called when the timer is expired.

The nodes transmit the messages according to the following priority order:

1. *Positive Acknowledgements* - If the node receives a *notification message*, it must transmit a *positive acknowledgement* to avoid network congestion. If a node is transmitting a *notification message* and starts receiving another *notification message* (that is not a *positive acknowledgement*) from the receiver, both will be subject to starvation because they will never receive a *positive acknowledgement* and stop transmitting the current notifications.

2. *Notification messages* - As described in Section 3.3.2, *notification messages* are used to control the network and avoid high load on the nodes.

3. *Fragments*

4. *Negative Acknowledgements* - They are the last because missing fragments may be received after its creation and before its transmission.

Timers

As one of the requirements for wireless sensor nodes is to avoid high energy consumption, it would not make sense to use infinite loops to transmit the data. Instead of this, the transmitting procedure is activated by timers, which are set on demand. These intervals are chosen according to the type of the last transmitted message.

If the node transmits a *fragment*, no *positive acknowledgement* is required before the next one is sent. This does not happen when a *notification message* is transmitted, as shown in Figure 3.2. Thus, after transmitting the *notification message*, the node must wait for a larger interval before doing the next transmission because it requires instant confirmation.

Moreover, each *content* has two different timers. The first one is used to control *negative acknowledgement* transmissions and the other one fires the retransmission procedure. It is important to remember that according to the cache policy, some fragment transmissions will take longer than a one-hop transmission.

The timer to retransmit the content depends on the number of hops from the farthest node achieved by the node's neighbours and has its countdown started as soon as the last fragment has been transmitted. This timer is disabled whenever the node receives any feedback (either a negative acknowledgement or a content acknowledgement) from each neighbour.

Both *content acknowledgements* and *negative acknowledgements* are enough to make sure that the neighbour was notified about the transmission and will ask for lost packets, if necessary.

As described above, every node that caches data does not start to retransmit the fragments before the whole content is received. Thus, only one of these two timers is active at a time in a node.

4.1.3 Receiving Procedure

Every received packet has its timestamp checked in a local cache. In this cache, the node stores the last received timestamp as well as the source identification. If the node has already received the packet, it is discarded. Except *data* messages and *negative acknowledgements*, all other messages are positively acknowledged after their reception.

Data messages are received and, according to the cache policy, the node presents different behaviours.

Notification messages, with the exception of *content acknowledgements*, are transmitted immediately after their creation, without any delay. The transmission interval between two notifications is set large enough to receive a response before the next message is transmitted, and this is only achieved if the source node does not add any delay before transmitting a message back.

4.1.4 Pre-defined Values

The implementation uses constant values set according to the hardware constraints and the transmission requirements. These values are:

Routing table size
: The routing table has at least the number of nodes in the network less one. As in the tests 6 nodes were used, the implementation requires a 5-position array in each node.

Number of groups
: The number of groups. The tests used 2 groups, then the value was set to 2.

Maximum group size
: This value depends on the number of nodes to receive the data. As in the tests 6 nodes were used, the implementation requires a 5-position array in each node.

Notification cache size
: This value has to be set according to the number of nodes. There is no minimum value,

but with a small size many packets might be dropped and the network will be overloaded. The implementation uses 5.

Received negative acknowledgements cache size
This value depends on the group structure and it is under the same rule as the notification cache. There is no minimum value, but with a small size many packets might be dropped and the network will be overloaded. The implementation uses 3.

Received timestamps cache size
The number of received timestamps depends on the number of nodes. It does not have a minimum value, because the implementation uses the least recently used (LRU) algorithm if the number of spaces is not enough.

Payload size
Length of the data fragments. This value may change if other protocols are chosen, because they may not have the same header length. The implementation uses 56 bytes.

Size of contents cache
The larger this value is, the more simultaneous transmissions are supported. If only one group transmission is done at a time, it can be 1. The value was set to 1 in the implementation.

Size of fragments cache
The number of fragments may depend on the transmission. It was set to 19.

String buffer length
This is used as a support to transmit the message to the application. The implementation uses 1045 bytes.

The MAC protocol used in the implementation was Null MAC. According to Contiki's documentation, this is a MAC protocol implementation that does not do anything. By default, it is configured to not provide collision avoidance, collision detection or channel access control. On the other hand, interval times between two transmissions are defined by the application and do not have to consider big variations due to MAC layer computation. Therefore, transmissions might be faster.

The smallest time unit in Contiki is a clock tick, generated by the processor. According to Contiki's source code, in the MSP430 microcontroller each second takes 128 ticks.

Because of this, the *transmission procedure* (explained on Section 4.1.2) and *receiving procedure* (explained on Section 4.1.3) are totally dependent on the timers used in the application. To set these values some experiments had to be done first and analysing the debugging logs (see Section 4.1.5), it was possible to infer that each transmission takes around 3 ticks.

As the data packets do not require any instant acknowledgement, the transmission interval between two data packets can be set based directly on this value. Thus, for the notification messages that require *positive acknowledgements* (see Figure 3.2), the interval between two transmissions has to be larger than 9 ticks (3 ticks to transmit, 3 ticks for computation time at the receiver, 3 ticks to receive a feedback).

In order to avoid collisions and keep the implementation scalable, some timers need to have a random delay added to them. For example, *content acknowledgements* could be transmitted at the same time by two different nodes and cause collisions.

Other timer values were derived from these and some of them can be calculated *on the fly*:

Maximum delay between two fragment transmissions

Depending on the adopted approach, the delay between two fragments may be different. If the neighbours are going to check the fragments before forwarding, than the interval between two transmissions can be only 3 ticks as described above. In other approaches, if the neighbours do not check the whole content before forwarding, the delay may be

$$t \ = \ 3 \times \# \, of \ hops \, to \, the \, farthest \, node$$

Maximum delay between two notification transmissions

Collisions between *positive acknowledgements* and *notification messages* might happen if the interval is the same at the two nodes. This delay reduces this problem during the transmissions. The value was set to 5 ticks.

Maximum delay to *content acknowledgement* transmissions

Because of the broadcast transmissions, some nodes may receive the last content fragment at the same time. After it has been received, a *content acknowledgement* is transmitted and many collisions might occur if no delay is used. This delay is bigger than the delay usually added to the transmission procedure if the next packet in the queue is a notification message. The maximum delay to *content acknowledgement* transmissions in this implementation is 17 ticks.

Delay before retransmitting the whole content

If a node does not receive any fragment from a content, it cannot transmit a *negative acknowledgement* to the source node. Thus, it is not possible to detect the packet losses and the content cannot be delivered. To avoid this scenario, the source node retransmits the content after an interval if no feedback has been received.

For this timer, the transmissions require an interval proportional to the number of hops between the node and the farthest node accessible from one of its neighbours. The distances in hops from each node to the farthest node is retrieved during *early phase* and stored with the group information.

If the fragments are cached on every node: Assuming that packet losses can occur during the transmission and they will be retransmitted after a *negative acknowledgement*, the value used is 3 times the *notification interval* (10 clock ticks). The result (30 clock ticks) is multiplied by the distance explained above and by the number of content fragments.

$$t \ = \ 3 \times notification \ interval \ \times \ \# \, of \ hops \, to \, the \, farthest \, node$$

If some of the nodes do not cache the fragments: The difference is that the nodes do not have to wait while the whole content is forwarded but only one fragment because the nodes without cache just forward the fragments (as described in Section 3.6.2).

Delay before transmitting a *negative acknowledgement*

As described before, *negative acknowledgements* should wait until the last fragment has been transmitted, even with eventual packet losses. The value used on each node is proportional to the number of hops to the nearest node with data cache, which is retrieved during the *early phase*. In order to optimize it, the number of fragments that the node may still receive is also used.

$$t = K \times \# \, of \; hops \; to \; the \; nearest \; caching \; node \; \times \; \# \, of \; fragments \; to \; receive$$

The fixed constant (K) is 30 ticks.

After the first *negative acknowledgement*, this value is set using:

$$t = J \times \# \, of \; hops \; to \; the \; last \; cache$$

And the fixed constant (J) is 40 ticks.

4.1.5 Additional Details

Debugging

In order to track the behaviour of the algorithm and check its correctness, a macro was created to print debug messages. Each debug line contains the local timestamp, the node identification, and a string. Different messages were set according to the place where it was used.

Due to memory restrictions, it is not possible to turn on all debug messages at the same time. However, once the algorithm was running as expected, the debug messages could be turned off, because the nodes used to waste relevant time to print them in the screen.

A positive aspect of the debugging facility is that by using the SLIP[30] tool provided by the nodes, it was possible to merge all messages, generate logs and check the transmissions. This helped to adjust timers and delays as well as improve the mechanisms of SNOMC.

Interface with Computer

As described in Section 2.3, Tmote Sky and TelosB nodes provide an USB interface to transmit the software images from a computer. Section 2.4 explains that Contiki has a module with a shell command that provides an interface between the computer and the node. This module was deactivated in order to reduce the image size and leave space to other modules such as μIP.

Instead of the shell interface, the computer interacts with the nodes using a TCP connection on the port 2802 (generic). The developed interface provides complete access to the nodes' resources and functionalities such as to start and finish groups, transmit contents, reset nodes and fetch their log data.

Using a SLIP connection, the source node prints on the screen the time spent to start groups and transmit *contents*. These times are measured as follows: When the source node receives the command to start a group or transmit a *content*, it stores the actual timestamp. When it receives the last *start acknowledgement* or the last *content acknowledgement* it prints the time difference (in ticks).

Transmission Logging

Every node stores a data structure with the number of received and transmitted packets. The counters are increased according to the packet type and can be either fetched from a TCP connection or printed on the screen using the “user button” in the nodes.

The output generated by this tool looks like the following example:

```
+- Received log from 0.1        --------+
|       Type: RECEIVED packets          |
+---------------------------------------+
|       START_GROUP messages:0          |
|       FINISH_GROUP messages: 0        |
|       DATA messages: 0                |
|       PACK messages: 96               |
|       NACK messages: 19               |
|       JOIN messages: 0                |
|       UNJOIN messages: 0              |
|       CONTENT_ACK messages:50         |
|       START_ACK messages: 51          |
+---------------------------------------+
|       Total: 216                      |
+---------------------------------------+
|       DATA for this node: 0           |
|                0 bytes                |
+---------------------------------------+

+- Received log from 0.1        --------+
|       Type: SENT packets              |
+---------------------------------------+
|       START_GROUP messages:55         |
|       FINISH_GROUP messages: 56       |
|       DATA messages: 934              |
|       PACK messages: 101              |
|       NACK messages: 0                |
|       JOIN messages: 0                |
|       UNJOIN messages: 0              |
|       CONTENT_ACK messages:0          |
|       START_ACK messages: 0           |
+---------------------------------------+
|       Total: 1146                     |
+---------------------------------------+
```

Using the logs from each node it is possible to combine them and calculate relevant information such as percentage of packet losses, number of messages during the transmission, number of retransmissions and others.

Changes in Contiki

In the default implementation of Contiki, the broadcast messages are not transmitted to all nodes but only to the gateway. The Rime [29] implementation redirects the broadcast packets to the gateway node and it is responsible for handling these packets.

Since Contiki is open source and allows changes to its contents, the broadcast functionality was changed to work as described in Section 2.2.3: Transmitted messages are received by all neighbours and the application drops the packets that came from an unexpected node, which is not responsible to transmit content to it. The nodes do not rebroadcast the received messages, avoiding the broadcast storm [21] problem.

4.2 UDP and TCP Implementations in Contiki

Since the idea of this work is to compare SNOMC with existing solutions, applications using UDP and TCP as transport layer were implemented too. With them, the same tests could be run and results compared.

UDP and TCP protocol implementations are placed over the μIP layer and the routing algorithm is placed in a lower layer. Therefore, a static routing table is configured in the μIP layer and makes the packets to use always the same route from the source to the receivers without any change on application level.

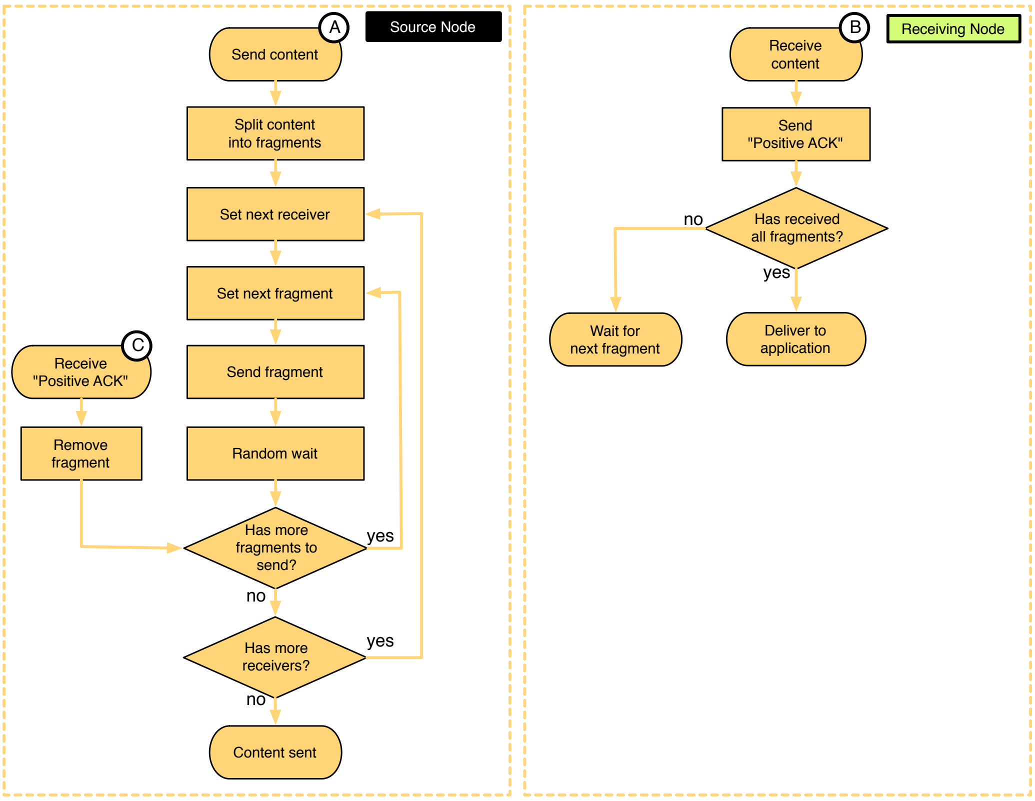
4.2.1 Application using UDP

The idea of this implementation is to create a simple solution using UDP that does the data transmission in a reliable way. Figure 4.3 shows a flowchart with the sequence of steps in the application at *source* and *receiving nodes*. The interaction between the nodes during the transmissions is shown in Figure 4.4.

This application works basically by transmitting the data for one receiver at a time. The *source node* receives the message from the application [**A**]. If the content is bigger than the maximum payload size, the application splits it into small fragments and transmits them one by one. For each received packet [**B**], the *receiving node* transmits an acknowledgement and waits for the next one, if existing. Only after transmitting all data to a receiver, the *source node* starts to transmit the same content to the next one in the list.

In order to provide fragmentation and reliability, the application inserts a header on each message. This header contains 1 byte to identify if the message is a positive acknowledgement or a content fragment, 2 bytes for the content identification, 1 byte with the number of fragments to be sent and 1 byte with the actual fragment number. Each fragment's length is 60 bytes.

In this implementation, the *receiving nodes* transmit an acknowledgement for each fragment received, different from SNOMC (as explained in Section 3.3.2). Thus, the source node transmits a fragment and waits for an interval. This interval is adaptive and is set as the time spent to receive the last acknowledgement plus a small delay (1 clock tick). The results in Chapter 5



(a) Flowchart in the *source node*. (b) Flowchart in *receiving nodes*.

Figure 4.3: Flowchart of the UDP application's algorithms.

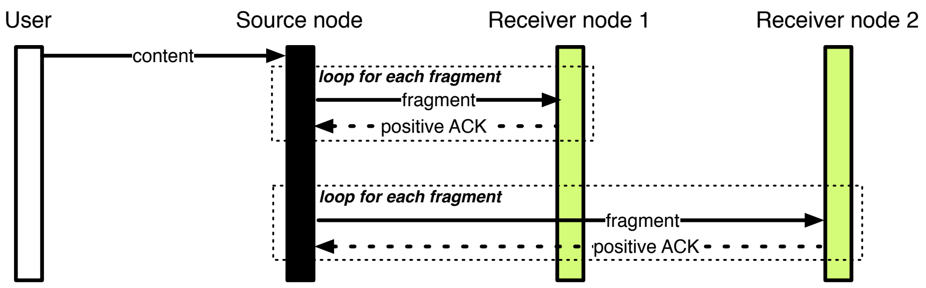


Figure 4.4: Sequence diagram of the UDP implementation.

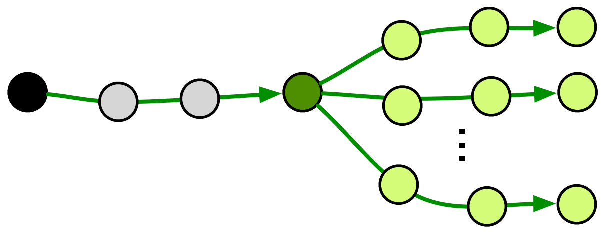


Figure 4.5: The scenario used to compare the algorithms.

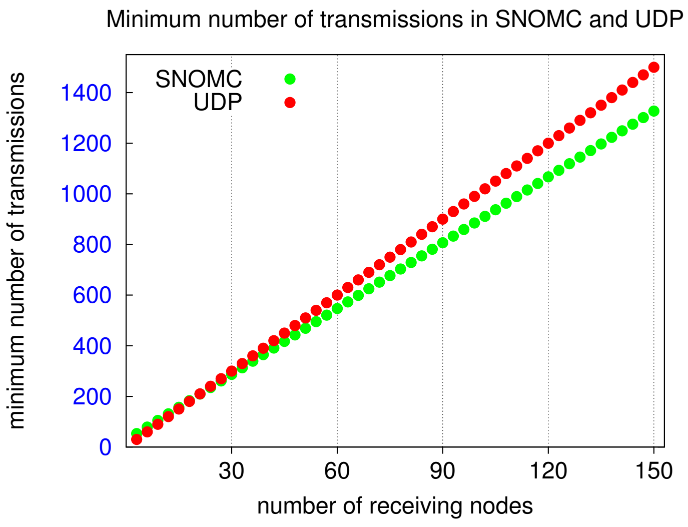


Figure 4.6: Minimum number of transmissions used to transmit one fragment.

show that there are almost no packet losses. It means that the intervals were enough to receive the acknowledgements when the fragment has been successfully delivered.

If the *source node* receives an acknowledgement before the end of this interval [**C**], it transmits the next fragment instantly. Otherwise, it retransmits the same fragment and waits for a new time interval once more. Whenever the last acknowledgement is received, the time is printed out as well as the number of transmitted messages. For each *receiving node* in the group, all fragments from a content must be transmitted through the nodes.

This implementation uses Null MAC protocol as well as the SNOMC's one (already described in Section 4.1.4).

4.2.2 Comparing SNOMC and UDP algorithms

Figure 4.5 shows a fixed structure with the source node, two *forwarding nodes* and a *branching node*. The diagram in Figure 4.6 shows the minimum number of transmissions used to transmit a small *content* (with one *fragment*).

From the left to right hand side in the axis, each point represents the minimum number of transmissions after adding a new branch with three *receiving nodes* to the structure.

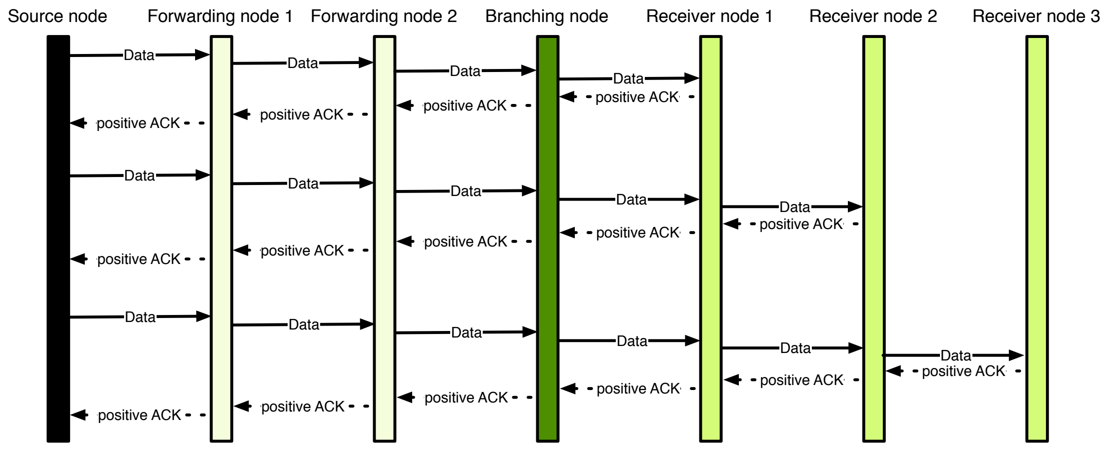


Figure 4.7: UDP implementation doing a transmission to one branch without packet losses.

For each new branch with three nodes, at least 26 new transmissions are done in SNOMC (11 during the *early phase*, 9 during the *transmission phase* and 6 during the *final phase*). In contrast, UDP requires a minimum of 30 new transmissions for each added branch, as shown in Figure 4.7.

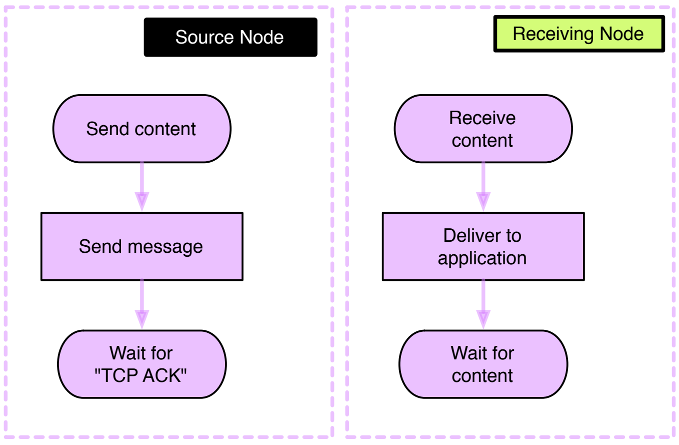
According to the diagram shown in Figure 4.6, UDP requires less transmissions than SNOMC if the WSN has less than 15 nodes. However, the number of transmitted messages increases faster if more nodes are added to the network.

4.2.3 Application using TCP

As the TCP protocol implementation provides fragmentation and reliability, the application can strictly transmit the content and the control will be given back to it only after the TCP acknowledgements have been received.

Figures 4.8 and 4.9 show the simplicity of this algorithm. The application just pays attention to the messages to be transmitted. The reliability is guaranteed by the transmission protocol and exchanged messages between the nodes at the application level can be identified in Figure 4.4.

This implementation does not require any kind of header in the application messages because of the TCP features. Every message is transmitted over the socket connection and no extra adjusts such as timers have to be done.



(a) Flowchart in the *source node*. (b) Flowchart in *receiving nodes*.

Figure 4.8: Flowchart of the TCP application's algorithms.

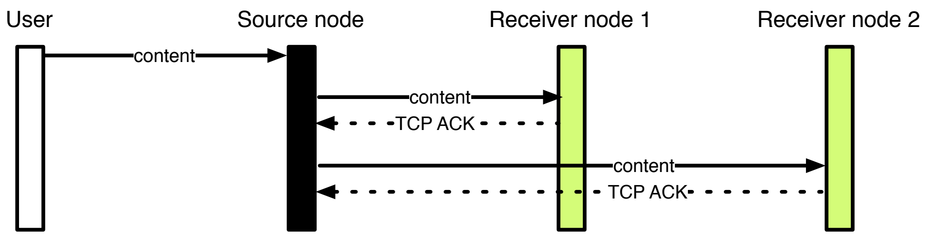


Figure 4.9: Sequence diagram of the TCP implementation.

Chapter 5

Evaluation

This chapter describes the tests done with the implementations presented before. The idea of the tests was to create a real-world WSN to simulate management tasks such as transmission of small configurations (20 bytes) and software updates (1000 bytes).

5.1 Scenario

Tests were done in the building of the Institute of Computer Science and Applied Mathematics in the University of Bern. In this building, many wireless networks are available as well as a test-bed with wireless sensor nodes and wireless mesh nodes establishing wireless transmissions. Moreover, walls, closed doors, and GSM networks are part of this environment. These aspects are relevant because parallel transmissions can cause interference and packet collisions in the transmission.

As discussed before, there are many possible scenarios in the real world where the WSNs are deployed, from military applications to environment monitoring. To create a real-world environment, six nodes were distributed inside the building with a distance between 4 and 8 meters to each direct neighbour. They were distributed at the same floor in the building as shown in Figure 5.1.

One of these nodes (the *source node*) was connected to a desktop computer with a Virtual Machine with Linux and each node was identified with a number and each line represents connectivity between two of them. The node number *1* is always the *source node* connected to the computer and transmitting data to the nodes using the interface.

The tests are performed in two scenarios: One with two nodes and other one with six nodes.

1. With two nodes:
   - node *1* acts as *source node*
   - node *2* acts as *receiver node*
2. With six nodes:
   - node *1* acts as *source node*

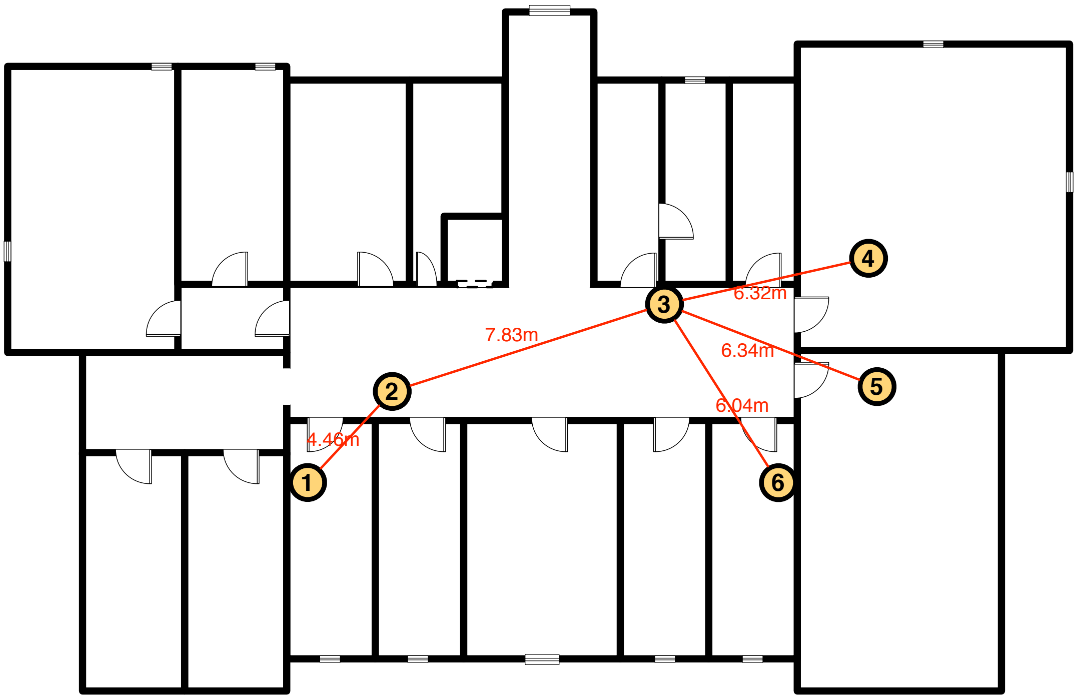


Figure 5.1: Distribution of the nodes inside the building.

- node *2* acts as *forwarding node*
- node *3* acts as *branching node*
- nodes *4*, *5* and *6* act as *receiver node*

To transmit the data to 3 *receiving nodes*, SNOMC uses *branching nodes* and *forwarding nodes*. With them, the 3 different caching schemes can be implemented, tested and the results are also compared in the next section.

5.2 Test Configuration

As SNOMC was created to be used in management tasks, the tests simulate typical management operations: configuration set up and software update.

In real-world applications, nodes collect data and transmit only the meaningful information to the sink node. After a while, the meaning of the data can change and new thresholds may be set. The tests simulate this transmission with 20 bytes to change the configuration of the *receiving nodes*.

Software updates can be done for some different reasons such as bug fixing, changes in the business models, etc.. An update may have around 700 bytes for a software with around 45KB [32]. For the tests, the computer simulated this update transmitting 1000 bytes to all *receiving nodes*.

5.3 Results

This section presents the results of the experiments described above. First, the tests with 20 bytes (simulation of a configuration changing) are shown and later the simulation of a transmission of a code update, with 1000 bytes.

Each test has been performed 50 times. The 50 observed values were sampled and presented in a boxplot in two different ways. The first one considers only the time spent to transmit the content to the nodes and the other one considers the time spent to start the group before transmitting the content (*early phase*) too. With these, it is possible to have a concrete idea about the variation of the observed times.

In order to have an idea about the energy consumption in the nodes, for each implementation a diagram with the total of transmissions (including packet retransmissions) and the percentage of packet losses are also shown. The percentage of packet losses was calculated as the relation between the total of done transmissions (including retransmissions) and the number of received messages:

$$\%\, of\ packet\ losses\ =\ \frac{\#\ of\ lost\ packets}{\#\ of\ lost\ packets + \#\ of\ received\ packets}$$

Each column in the diagrams represents a different combination of design models (as discussed in Section 3.4). Some columns do not consider all phases of the algorithm. The labels are described below:

SNOMC
SNOMC with no *forwarding nodes* and *branching nodes*, there is no cache policy in this case. Here, only the *transmission phase* (see Section 3.6.2) is considered.

SNOMC with early phase
SNOMC with no *forwarding nodes* and *branching nodes* as the previous one, but considering the *early phase* (see Section 3.6.2).

SNOMC Every
SNOMC with cache in every node (see Section 3.4.2). It uses the *first forward* approach (see Section 3.4.3). Here only the *transmission phase* is considered.

SNOMC Every with early phase
SNOMC with cache in every node. It uses the *first forward* approach. The results presented consider also the *early phase*.

SNOMC Every first check
SNOMC with cache in every node, using the *first check* approach (see Section 3.4.3). In this case, only the *transmission phase* is considered.

SNOMC Every first check with early phase
SNOMC with cache in every node, using the *first check* approach. The results presented consider also the *early phase*.

		20 bytes	**1000 bytes**
Early phase			
	Start group messages	1	1
	Start group acknowledgements	1	1
	Positive acknowledgements	1	1
Transmission phase			
	Data messages	1	18
	Content acknowledgements	1	1
	Positive acknowledgements	1	1
Final phase			
	Finish group messages	1	1
	Positive acknowledgements	1	1
Total		8	25

Table 5.1: Comparing SNOMC transmissions in the 2-nodes scenario (ideal case).

	20 bytes	**1000 bytes**
Data messages	1	17
Positive acknowledgements	1	17
Total	2	34

Table 5.2: Comparing UDP transmissions in the 2-nodes scenario (ideal case).

SNOMC Branch
SNOMC with cache in *branching nodes* (see Section 3.4.2). In this case, only the *transmission phase* is considered.

SNOMC Branch with early phase
SNOMC with cache in *branching nodes*, considering the *early* and the *transmission phases*.

SNOMC Source
SNOMC with caching only in the *source node* (see Section 3.4.2). In this case, only the *transmission phase* is considered.

SNOMC Source with early phase
SNOMC with caching only in the *source node* considering the *early* and the *transmission phases*.

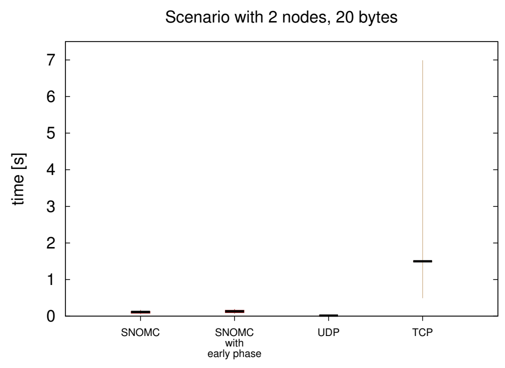
Tables 5.1, 5.2, 5.3, and 5.4 compare the minimum number of transmissions of the implementations. To calculate these values, eventual packet losses were not considered.

		20 bytes	**1000 bytes**
Early phase			
	Start group messages	5	5
	Start group acknowledgements	5	5
	Positive acknowledgements	7	7
Transmission phase			
	Data messages	3	54
	Content acknowledgements	5	5
	Positive acknowledgements	5	5
Final phase			
	Finish group messages	5	5
	Positive acknowledgements	5	5
Total		40	91

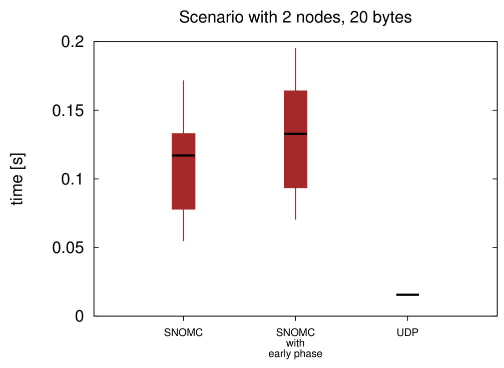
Table 5.3: Comparing SNOMC transmissions in the 6-nodes scenario (ideal case).

	20 bytes	**1000 bytes**
Data messages	9	174
Positive acknowledgements	9	174
Total	18	348

Table 5.4: Comparing UDP transmissions in the 6-nodes scenario (ideal case).

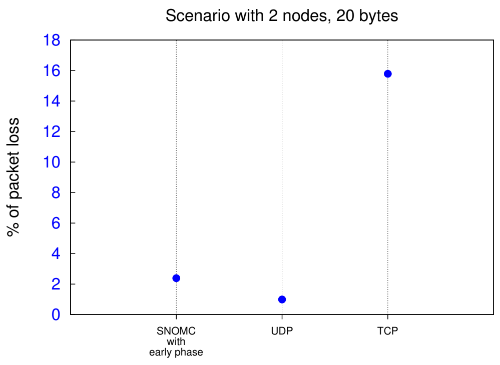


(a) With TCP measurements.

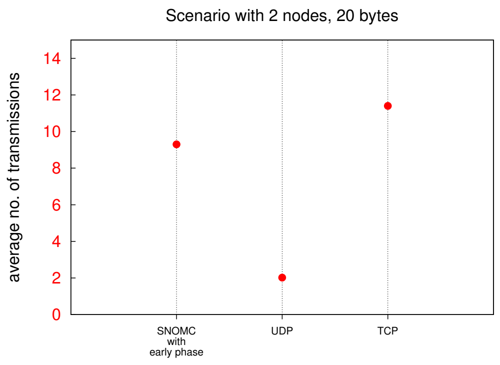


(b) Without TCP measurements.

Figure 5.2: Time to transmit 20 bytes in the 2-nodes scenario.



(a) Percentage of packet losses.



(b) Average number of transmissions.

Figure 5.3: Numbers observed after transmitting 20 bytes in the 2-nodes scenario.

5.3.1 Transmission of a Configuration (20 bytes)

Scenario with 2 Nodes

In the first test, the *source node* transmits 20 bytes to one node. Figure 5.2(a) shows results comparing SNOMC, UDP and TCP.

Figure 5.3(b) shows a large variance between the observed transmission times using the TCP implementation: The lowest time is 0.49 seconds and the highest is 7 seconds. The big difference between the times can be explained by the high percentage of packet losses (see Figure 5.3(a)), which represents the high number of retransmissions done in order to provide reliability (see Figure 5.3(b)).

As TCP has a result much worse than the others, Figure 5.2(b) shows the results of SNOMC and UDP in a new diagram with new intervals better fitting to them.

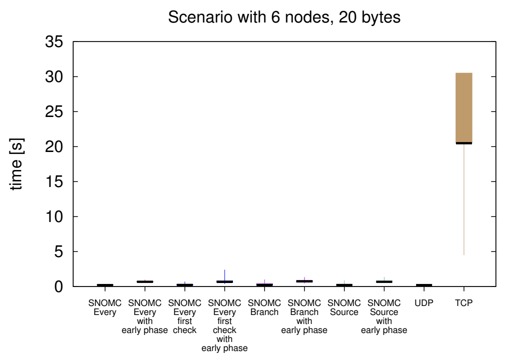
According to these diagrams, SNOMC is a better solution when compared to TCP, but it is worse than UDP. SNOMC has a small number of packet losses (see Figure 5.3(a)) but the average number of transmissions is around 20% larger than the number of messages used by the UDP implementation. This difference was predicted in Tables 5.1 and 5.2 and happens because of the messages transmitted during the *early* and *final phases*.

Scenario with 6 Nodes

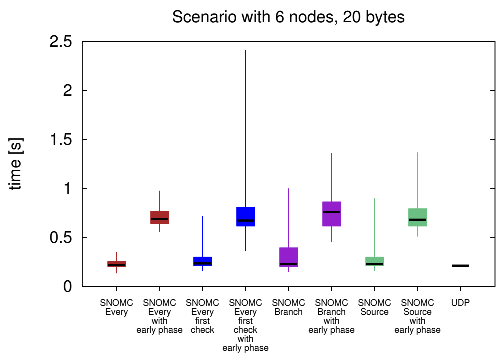
The transmission of 20 bytes to 3 receivers in the scenario with 6 nodes was done in this test. In this case, TCP and UDP have to transmit the same content three times.

Figure 5.4(a) shows the four caching approaches of SNOMC, UDP, and TCP results. In these tests, the average number of messages transmitted in TCP is larger than in UDP and SNOMC implementations as in the 2-nodes scenario. The difference now is the larger percentage of packet losses of the TCP implementation, which explains the big variance in the Figure 5.4(a).

Figure 5.4(b) shows the results comparing SNOMC approaches with UDP in a new diagram with new intervals fitting better with them.

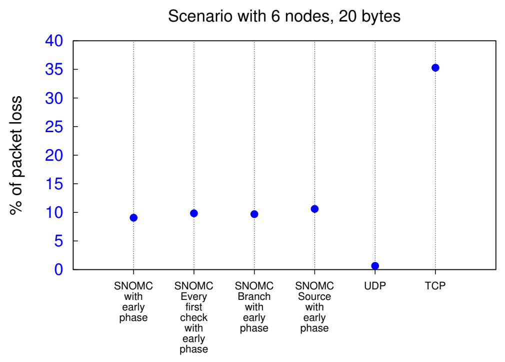


(a) With TCP measurements.

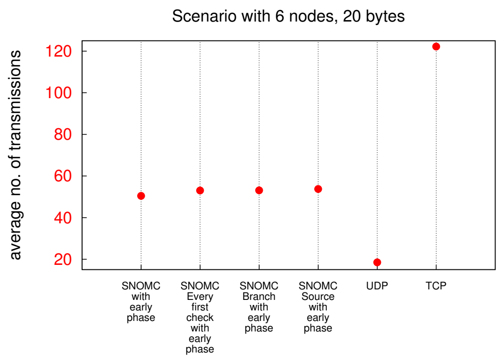


(b) Without TCP measurements.

Figure 5.4: Time to transmit 20 bytes in the 6-nodes scenario.



(a) Percentage of packet losses.



(b) Average number of transmissions.

Figure 5.5: Numbers observed after transmitting 20 bytes in the 6-nodes scenario.

According to these diagrams, SNOMC is a better solution when compared with TCP, but not if compared with UDP.

For this kind of small amount of data and low number of *receiving nodes*, SNOMC does not seem to be the best solution. This happens mainly because the *early phase* takes relatively long time in comparison with the *transmission phase*. Using any SNOMC implementation, it is expected a minimum of 40 transmissions (see Table 5.3) and using the UDP implementation, it is expected a minimum of 18 transmissions (see Table 5.4).

Comparing SNOMC with UDP, Figure 5.5(a) shows the percentage of packet losses in the implementations. UDP has a lower percentage of packet losses (0.65%) than SNOMC implementations (9.06%, 9.81%, 9.70%, and 10.64%). This also explains why the average number of transmitted messages (see Figure 5.5(b)) is smaller than in SNOMC implementations: UDP has transmitted around 18.5 messages per content, while SNOMC has used more than 50 transmissions for each content.

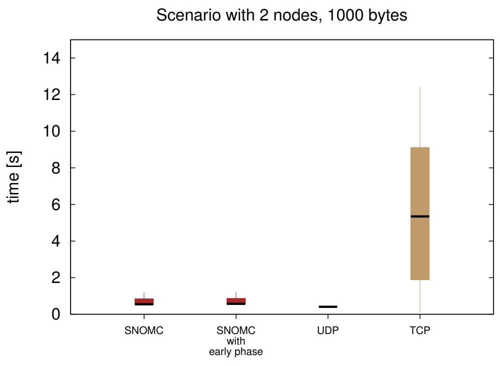
However, according to the diagrams in Section 4.2.2, these results were expected with small number of *receiving nodes*. According to them, SNOMC will probably provide better results if more nodes are added to the WSN.

Considering only the SNOMC implementations, the one using cache in every node and the *first check* approach is the best. This can be explained because of the small number of fragments and packet losses, which do not require many retransmissions. Analysing the timing results for *early* and *transmission phases*, *SNOMC Every* has a difference of 0.42 seconds from the highest to the lowest values. The variance in *SNOMC Every first check* is 2.05 seconds, in *SNOMC Branch* it is 0.91 seconds, and in *SNOMC Source* it is 0.86 seconds.

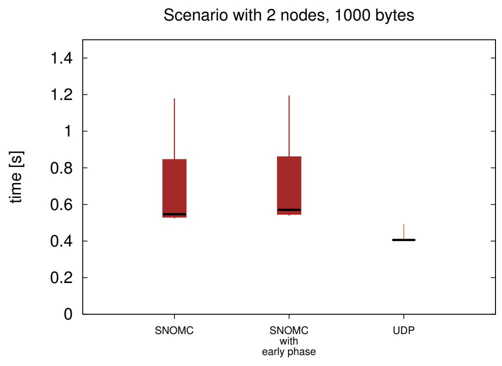
These results show that SNOMC implementation takes almost the same time than UDP to transmit the content, but there is an overhead to notify the groups about the group transmission (*early phase*), which turns it into an expensive choice for this configuration.

5.3.2 Transmission of a Code Update (1000 bytes)

Scenario with 2 Nodes

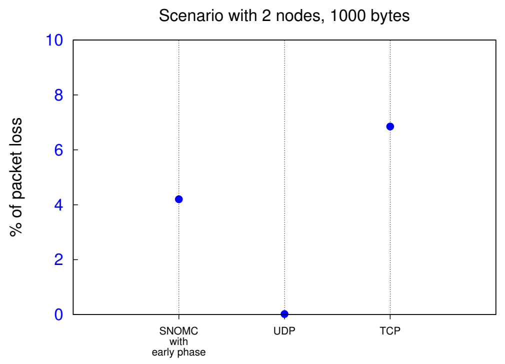


(a) With TCP measurements.

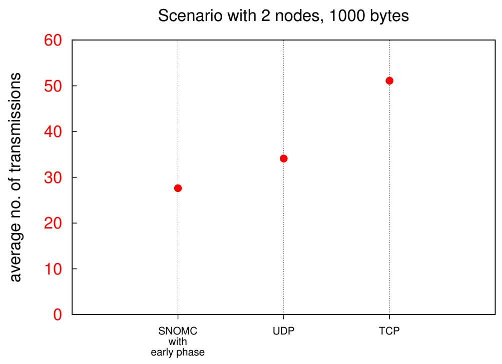


(b) Without TCP measurements.

Figure 5.6: Time to transmit 1000 bytes in the 2-nodes scenario.



(a) Percentage of packet losses.



(b) Average number of transmissions.

Figure 5.7: Numbers observed after transmitting 1000 bytes in the 2-nodes scenario.

The large amount of data requires many transmissions from the *source node* to the *receiver node*. Figure 5.6(a) shows the high number of transmissions due to the reliability offered by TCP.

The best result using TCP takes around 4 times more time than the worst results from SNOMC and UDP implementations. The TCP implementation has more packet losses (6.85%) than the other implementations (UDP has 0.02% and SNOMC has 4.2%), as shown in Figure 5.7(a). This large amount of packet losses implies many retransmissions and a large number of messages (see Figure 5.7(b)). The result is a big variance between the observed times: The highest value is 122 times bigger than the lowest one.

Even using UDP transmissions in SNOMC, the percentage of packet losses is bigger than in the UDP implementation. This difference can be explained because some transmissions can be delayed by environment interferences and the timers used by SNOMC had not been set large enough to avoid few unnecessary retransmissions, which generates some packet collisions.

Removing TCP from the diagram, it is possible to see the difference between SNOMC and UDP in Figure 5.6(b).

In this case, the relation between the *early phase* and the *transmission phase* in SNOMC is not as big as in last tests. Moreover, the longest transmission in SNOMC took only 2.2 times more than the fastest one and most of the transmissions were not more than 1.6 times worse than the best one.

UDP is a faster solution for the 20-bytes transmission case, but SNOMC scales better and requires less transmissions in this scenario. This can be seen in Figure 5.7(b): SNOMC transmitted less messages than the UDP implementation.

Scenario with 6 Nodes

The TCP implementation was not able to finish the whole transmission. After some attempts, the content with 1000 bytes was never delivered to the receivers. Thus, it was not possible to display the respective data in the diagram.

According to the Figure 5.8, the *early phase* is relatively short in comparison with the time

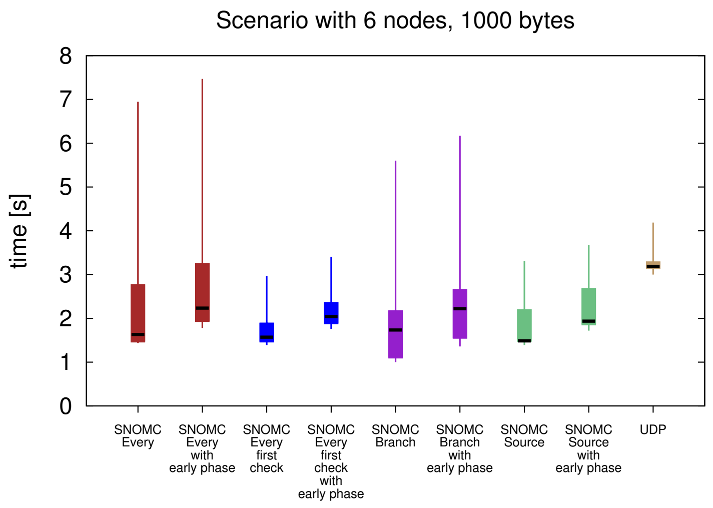
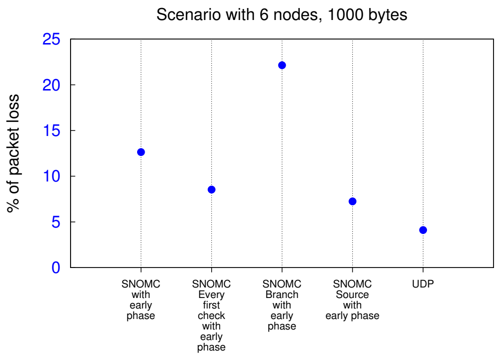
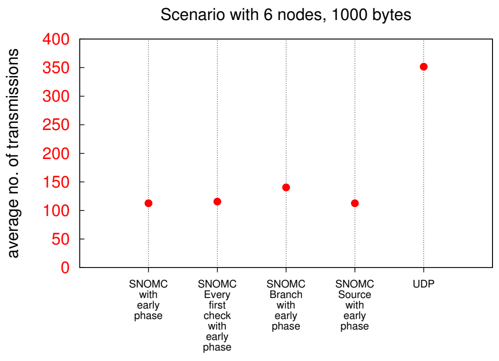


Figure 5.8: Time to transmit 1000 bytes in the 6-nodes scenario.



(a) Percentage of packet losses.



(b) Average number of transmissions.

Figure 5.9: Numbers observed after transmitting 1000 bytes in the 6-nodes scenario.

used to transmit the content. Even with them, SNOMC presents results around 1.5 times better than UDP.

Tables 5.3 and 5.4 show the minimum number of transmissions in SNOMC and UDP. After checking the values, it is possible to predict that SNOMC implementations may have larger percentage of packet losses than the UDP implementation without doing as many transmissions as it does. It depends on how many transmissions are required to handle a packet loss.

SNOMC implementations transmit around 3 times less messages than UDP. The main reason for this is because SNOMC uses unicast and also broadcast transmissions, which avoids many redundant transmissions. In the actual scenario, SNOMC can combine 3 transmissions in one using broadcast.

This big difference in the number of transmissions is relevant to the transmission time and may have an impact on the nodes' energy consumption.

The percentage of packet losses is very low in UDP (4.11%), as shown in Figure 5.9(a). Therefore, although SNOMC has a larger number of packet losses in SNOMC, it is not enough to make more transmissions than in UDP, as shown in Figure 5.9(b).

Moreover, even though each SNOMC implementation uses UDP transmissions, there are relevant differences between their percentage of packet losses.

SNOMC implementation with cache in the *branching nodes* has 22.14% of packet losses, which represents up to three times more than the other SNOMC approaches. In the *forwarding nodes*, the *fragments* are received and forwarded, because have not been cached. Whenever a *fragment* is received by a *branching node*, it waits to receive the whole content before starting to forward it (as described in Section 3.4.2). By doing this, it is harder for the *source node* to predict how long the transmission will take if some packets are lost. Therefore, timers used in the *source node* may have different consequences:

- Timers large enough to avoid unnecessary packet retransmissions, but this would increase the time to retransmit a *content*, if a fragment was lost.
- Unnecessary retransmissions may cause packet collisions and increase the number of packet losses.

In these tests, the timers are set to detect packet losses earlier. As result of this, the nodes make some unnecessary packet retransmissions generating a larger percentage of packet losses and a big variance between the fastest and the slowest content transmissions, as shown in Figure 5.8. Moreover, Figure 5.9(b) shows that the average number of transmissions in the *SNOMC Branch* implementation is larger than in the others.

Figure 5.9(a) shows that the SNOMC implementation using cache only in the *source node* has less packet losses (7.25%) than the other SNOMC approaches (12.64%, 8.54%, and 22.14%). This happens because it is easier to detect packet losses and avoid packet collisions caused by unnecessary retransmissions. Only the *source node* is responsible for making retransmissions in a group, which requires more packet transmissions to be done. This is why Figure 5.9(b) shows that this approach still has as many transmissions as the other SNOMC implementations.

Differences between the four implementations of SNOMC cannot be distinguished using only the times used to transmit the *contents*. If either *branching* or *forwarding nodes* do not have space to cache the *fragments*, the cache policy can be changed without a big performance decrease.

Number of messages can be considered, but the implementations do not present big differences. *SNOMC Source* has the lowest number of transmissions: Around 112.6 messages. *SNOMC Branch* has the highest number of transmissions and has transmitted around 140.36 messages (25% more). This represents 150% less messages than the UDP implementation.

Supposing *branching* and the *forwarding nodes* have space to cache the data, there are two SNOMC implementations with similar results to be used: *cache in all nodes* with *first check* and *cache in source node*. However, the small difference between the shortest and the longest times illustrates the stability and scalability of the *cache in all nodes* approach.

Chapter 6

Conclusions and Future Work

This chapter explains the conclusions about the observed results, the relevance of the improvements and some points which can be studied in future work.

6.1 Conclusions

The chosen scenarios provided a real-world impact over the application, with concurrent transmissions and much interference, causing packet losses and requiring the reliability's functionality of the applications.

For transmissions only between two nodes, SNOMC spent more time than UDP. The number of transmitted messages is larger if the content has only 20 bytes. Thus, it is possible to predict that the number of messages increases exponentially if more nodes are added to the WSN. This value increases according to the number of hops between the *source node* and the *receiving nodes*. Despite this occurrence, SNOMC has been able to add new receivers without network overload because of the use of broadcast transmissions and caching strategies. Based upon these results, it is possible to suggest that SNOMC is also a good option for the transmission of small amounts of data to more than three nodes.

If a large amount of data has to be sent, SNOMC is the best option. It always uses less transmissions than the UDP and TCP implementations, and in the 6-nodes scenario the transmissions are also faster than UDP.

After the tests' results and analysis, SNOMC can be considered a good solution to transmit large amounts of data to groups of nodes in a WSN. As predicted in the description section, SNOMC may avoid many connections, energy consumption and messages transmissions.

A comparison between SNOMC and TCP resulted in much better results with SNOMC. TCP provides the same functionalities as SNOMC such as fragmentation and reliability, but the high number of control messages turns it into a non-feasible option in this scenario.

SNOMC is a self-adaptive implementation, because the timer values used by the nodes to transmit the packets depend upon the distance between two caching nodes (as described in Section 4.1.2). This quality is also explicitly shown when the nodes use the same implementation to transmit 20 and 1000 bytes, which is not feasible using UDP and TCP applications.

6.2 Future Work

SNOMC brings up some points which may be the focus of future work. This section presents some ideas which may improve the results obtained thus far and are offered as ideas for the next steps.

- The actual version consumes too much memory space from the nodes and does not allow a big application to be deployed. Due to this size, some extra functionalities could not be used such as the shell provided by Contiki. Without the shell utility, it was not possible to deploy the implementation in TARWIS [33] and use its command interface to test the implementation. The clean interface of TARWIS, may make it possible to easily test scenarios with more than six nodes. Future work may consider to make changes in the code in order to generate a compact image and do the tests in TARWIS.

- Another point to be observed in SNOMC is the header size. For example, the *data message* type consumes 9 bytes and has only 56 bytes free for the data content. A smaller header would imply larger data fragments and lower number of messages used in a group transmission.

- On the medium access layer, the option used was the Null MAC. For future works, other MAC protocols like X-MAC or BEAM [34] may work better without having to depend upon the pre-defined timers' values to avoid congestion and collisions between two transmissions. These MAC protocols allows the calculation of how much energy can be saved using SNOMC.

- So far, the biggest restriction of SNOMC is the absence of procedures to create routes between *source node* and *receiving nodes*. Future works may consider the idea to combine SNOMC with one of the algorithms presented in Section 2.2.3 (VLM^2, ADMR, GMP, GMR) to discover the routes used by SNOMC.

- Finally, SNOMC may be compared to other implementations such as Flooding, Directed Diffusion [35] (DD) and Multi-point relay [36] (MPR). These approaches are used to disseminate the data to many receivers and are often used in real-world implementations.

Bibliography

[1] A. Munir and A. Gordon-ross, “Optimization approaches in wireless sensor networks,” in *Sustainable Wireless Sensor Networks*. InTech, 2010, pp. 313–338.

[2] *Tmote Sky Datasheet*, Nov. 2006. [Online]. Available: http://www.sentilla.com/files/pdf/eol/tmote-sky-datasheet.pdf

[3] *TelosB Datasheet*. [Online]. Available: www.willow.co.uk/TelosB_Datasheet.pdf

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(01)00302-4

[5] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Gunǎ, M. Corrà, M. Pozzi, D. Zonta, and P. Zanon, “Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment,” in *In Proc. of the 8th ACM/IEEE Int. Conf. on Information Processing in Sensor Networks (IPSN). Best Paper Award*, 2009.

[6] S. H. Lee, S. Lee, H. Song, and H. S. Lee, “Wireless sensor network design for tactical military applications: remote large-scale environments,” in *Proceedings of the 28th IEEE conference on Military communications*, ser. MILCOM’09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 911–917. [Online]. Available: http://portal.acm.org/citation.cfm?id=1856821.1856955

[7] J. Postel, “User Datagram Protocol,” RFC 768 (Standard), Internet Engineering Task Force, August 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[8] V. Cerf, Y. Dalal, and C. Sunshine, “Specification of Internet Transmission Control Program,” RFC 675, Internet Engineering Task Force, December 1974. [Online]. Available: http://www.ietf.org/rfc/rfc675.txt

[9] M. Hosseini, D. Ahmed, S. Shirmohammadi, and N. Georganas, “A Survey of Application-Layer Multicast Protocols,” *IEEE Communications Surveys & Tutorials*, vol. 9, no. 3, pp. 58–74, 2007. [Online]. Available: http://dx.doi.org/10.1109/COMST.2007.4317616

[10] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, “A survey and comparison of peer-to-peer overlay network schemes,” *IEEE Communications Surveys and Tutorials*, vol. 7, pp. 72–93, 2005.

[11] “Rfc 791 internet protocol - darpa inernet programm, protocol specification,” Internet Engineering Task Force, September 1981.

[12] A. Dunkels, “The uIP Embedded TCP/IP Stack The uIP 1.0 Reference Manual,” 2006.

[13] G. Wagenknecht, M. Anwander, M. Brogle, and T. Braun, “Reliable multicast in wireless sensor networks,” in *FGSNâ08*, 2008, pp. 69–72.

[14] J. S. Wilson, *Sensor Technology Handbook*, J. S. Wilson, Ed. Newnes, 2005.

[15] G. Wagenknecht, M. Anwander, T. Braun, T. Staub, J. Matheka, and S. Morgenthaler, “MARWIS: A Management Platform for Heterogeneous Wireless Sensor Networks,” 2008, pp. 177–188. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68807-5_15

[16] R. Braden, “Rfc 1122 requirements for internet hosts - communication layers,” Internet Engineering Task Force, October 1989. [Online]. Available: http://tools.ietf.org/html/rfc1122

[17] A. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall Professional Technical Reference, 2002.

[18] D. Plummer, “Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware,” RFC 826 (Standard), Internet Engineering Task Force, Nov. 1982, updated by RFCs 5227, 5494. [Online]. Available: http://www.ietf.org/rfc/rfc826.txt

[19] R. Droms, “Dynamic Host Configuration Protocol,” RFC 2131 (Draft Standard), Internet Engineering Task Force, Mar. 1997, updated by RFCs 3396, 4361, 5494. [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt

[20] M. Handley, E. Rescorla, and IAB, “Internet Denial-of-Service Considerations,” RFC 4732 (Informational), Internet Engineering Task Force, Dec. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4732.txt

[21] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu, “The broadcast storm problem in a mobile ad hoc network,” *Wirel. Netw.*, vol. 8, no. 2/3, pp. 153–167, 2002.

[22] H. Holbrook, B. Cain, and B. Haberman, “Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast,” RFC 4604 (Proposed Standard), Internet Engineering Task Force, August 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4604.txt

[23] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas, “Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised),” RFC 4601 (Proposed Standard), Internet Engineering Task Force, Aug. 2006, updated by RFCs 5059, 5796. [Online]. Available: http://www.ietf.org/rfc/rfc4601.txt

[24] A. Adams, J. Nicholas, and W. Siadak, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)," RFC 3973 (Experimental), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3973.txt

[25] A. Sheth, B. Shucker, and R. Han, "Vlm2: A very lightweight mobile multicast system for wireless sensor networks," in *IEEE Wireless Communications and Networking Conference (WCNC) 2003*, 2003, pp. 1936–1941.

[26] B. rong Chen, K. kumar Muniswamy-reddy, and M. Welsh, "Ad-hoc multicast routing on resource-limited sensor nodes," in *in Proceedings of the 2nd International Workshop on Multi-hop Ad*. ACM Press, 2006, pp. 87–94.

[27] J. A. Sanchez, P. M. Ruiz, and I. Stojmenovic, "GMR: Geographic multicast routing for wireless sensor networks," in *Proceedings of the 3rd Annual IEEE Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, vol. 1, 2006, pp. 20–29.

[28] A. Dunkels, "Contiki: Bringing ip to sensor networks." *ERCIM News*, vol. 2009, no. 76, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/ercim/ercim2009.html#Dunkels09

[29] ——, "Rime — a lightweight layered communication stack for sensor networks," in *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007. [Online]. Available: http://www.sics.se/~adam/dunkels07rime.pdf

[30] J. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP," RFC 1055 (Standard), Internet Engineering Task Force, June 1988. [Online]. Available: http://www.ietf.org/rfc/rfc1055.txt

[31] J. Postel, "Internet Control Message Protocol," RFC 792 (Standard), Internet Engineering Task Force, Sept. 1981, updated by RFCs 950, 4884. [Online]. Available: http://www.ietf.org/rfc/rfc792.txt

[32] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," Los Angeles, CA, USA, Tech. Rep., 2003. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.4326

[33] P. Hurni, G. Wagenknecht, M. Anwander, and T. Braun, "A testbed management architecture for wireless sensor network testbeds (tarwis)," in *7th European Conference on Wireless Sensor Networks (EWSN)*, Feb. 2010, pp. 33–35.

[34] M. Anwander, G. Wagenknecht, T. Braun, and K. Dolfus, "Beam: A burst-aware energy-efficient adaptive mac protocol for wireless sensor networks," in *7th International Conference on Networked Sensing Systems*, June 2010, pp. 195–202.

[35] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Trans. Netw.*, vol. 11, pp. 2–16, February 2003. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.808417

[36] Y. Faheem and S. Boudjit, “Sn-mpr: A multi-point relay based routing protocol for wireless sensor networks,” in *Proceedings of the 2010 IEEE/ACM Int’l Conference on Green Computing and Communications & Int’l Conference on Cyber, Physical and Social Computing*, ser. GREENCOM-CPSCOM ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 761–767. [Online]. Available: http://dx.doi.org/10.1109/GreenCom-CPSCom.2010.139