Implementing and Evaluating the Dynamic Manet On-demand Protocol in Wireless Sensor Networks

Romain Thouvenin

June 11, 2007

Supervisor : Lars Michael Kristensen
Abstract

According to MIT's Technology Review[1], wireless sensor networks are one of the ten emerging technologies that will change the world. Undoubtedly, this fact has been widely understood by the research community and explains the interest received by the research area of sensor networks. Many issues still need to be solved, one of them is enabling software to handle multi-hop routing among nodes. The best suitable protocol may vary depending on the hardware used and on the purpose of the network. That is the reason why a wide range of proposed protocols need to be evaluated, in order to find which protocol properties are appropriate for each kind of network. This is what this master's thesis aims at contributing to, by implementing and evaluating a particular routing protocol for mobile ad-hoc networks. The protocol considered is the Dynamic MANET On-demand (DYMO) protocol and is evaluated in the context of sensor networks.

The implementation was made in nesC on TinyOS, a programming language and an operating system specifically designed for distributed and embedded software limited by the hardware constraints of sensor networks. A simplified version of the DYMO protocol has been implemented with a more compact packet format adapted to these constraints.

The evaluation consists of experiments performed on real hardware to determine the applicability of the DYMO protocol for a typical sensor network application. The results confirm the strong relationship between the use of the radio emitter and the power consumption. They also provide an idea of the life expectancy of such a sensor network depending on the traffic rate of the application.
Acknowledgements

Because I enjoyed this work and was excited by its progress, I would like to thank all the people who helped me towards achieving this:

- My supervisor Lars Kristensen, who was always ready to provide feedback, carefully review my work, and help me improve it;
- Cécile Ritte, who provided me with advice to take the most of my stay in Denmark and to ease the preparation of my Master’s Thesis;
- Ian Chakeres, Charles Perkins, and the manet mailing list subscribers for their interest in my work and their availability to answer my questions about the DYMO protocol;
- The TinyOS team and its community for their interest in my work and their answers to my questions, as well as Philipp and Teja for their useful feedback;
- My uncle Jean-Louis and all the people who spent some time to proofread my writing;
- And finally, all the people involved in the process who allowed me to spend this year at the University of Aarhus. It was a very good experience.
## Contents

1 Introduction .................................................. 9
   1.1 Ad-hoc Sensor Networks and Routing Protocols ............ 9
   1.2 Goal and Contributions of this Thesis .................. 11
   1.3 Structure of this Thesis ................................ 11
   1.4 Reading Guide ......................................... 12

2 Distributed Sensor Networks ................................. 15
   2.1 History ................................................. 15
   2.2 Applications .......................................... 16
      2.2.1 Monitoring .......................................... 16
      2.2.2 Tracking ............................................ 17
      2.2.3 Mobile Networks .................................... 17
   2.3 Research Challenges .................................... 17
   2.4 Routing Protocols ...................................... 19
      2.4.1 Flat Routing Protocols ............................ 19
      2.4.2 Hierarchical Routing Protocols .................... 22
      2.4.3 Geographic Routing Protocols ..................... 22
   2.5 Conclusion ............................................. 22

3 Implementation Context ..................................... 25
   3.1 Motes .................................................. 25
      3.1.1 History ............................................. 25
      3.1.2 The TelosB Research Platform ..................... 26
      3.1.3 Software Requirements ............................ 27
   3.2 TinyOS ................................................ 28
      3.2.1 Component Model ................................... 29
      3.2.2 Execution Model ................................... 29
      3.2.3 Network Interface ................................ 29
   3.3 nesC ................................................... 30
      3.3.1 Component Model ................................... 30
      3.3.2 Concurrency ........................................ 34
### CONTENTS

3.3.3 Additional Features .............................................. 35  
3.4 Summary .......................................................... 35

4 The DYMO Routing Protocol ........................................ 37
  4.1 Overview ......................................................... 38
  4.2 Route Discovery .................................................. 38
    4.2.1 Routing Messages .......................................... 38
    4.2.2 Packet Processing and Forwarding ............................. 40
  4.3 Route Maintenance ................................................ 40
    4.3.1 Length and Freshness of Routes ............................. 41
    4.3.2 Link Monitoring ............................................. 41
    4.3.3 Route Errors ................................................ 41
  4.4 Packet Format .................................................... 42
    4.4.1 Original Format .............................................. 42
    4.4.2 Implemented Format ......................................... 44
  4.5 Conclusion ....................................................... 47

5 Implementation ...................................................... 49
  5.1 Integration in TinyOS ............................................. 49
    5.1.1 Link Layer .................................................... 49
    5.1.2 Network Layer ................................................. 53
  5.2 Architecture of the Implementation ............................. 54
    5.2.1 Overview ..................................................... 54
    5.2.2 The DYMO Service ........................................... 59
    5.2.3 The MH Service ............................................. 59
    5.2.4 The Routing Table ........................................... 61
  5.3 Routing Interfaces ............................................... 62
    5.3.1 Routing Table Interfaces .................................... 62
    5.3.2 Packets ..................................................... 63
    5.3.3 Route Selection ............................................. 66
  5.4 Implementation Issues .......................................... 66
  5.5 Conclusion ....................................................... 68

6 Tests and Experimentation .......................................... 71
  6.1 Functional Tests ................................................. 71
    6.1.1 TOSSIM ..................................................... 71
    6.1.2 Simulated Tests ............................................. 73
    6.1.3 Hardware Tests .............................................. 75
  6.2 RREQ Delay Evaluation ......................................... 75
    6.2.1 Experimental Process ....................................... 76
    6.2.2 Results ..................................................... 76
6.3 The Influence of DYMO on Power Consumption 77
  6.3.1 Goal 77
  6.3.2 Implementation 78
  6.3.3 Experimental Process 81
  6.3.4 Results 83
6.4 Conclusion 87

7 Conclusions and Futur Work 89
  7.1 Conclusion and Lessons Learned 89
  7.2 Futur Work 91
    7.2.1 Experimentation 91
    7.2.2 Implementation 91
    7.2.3 Design 92
Chapter 1

Introduction

This chapter deals with the background, aims and purpose of this thesis, and gives an outline of how the thesis is organized.

1.1 Ad-hoc Sensor Networks and Routing Protocols

Thanks to their low price and performance nowadays, wireless communication technologies are becoming widely used. They are integrated in numerous kinds of devices that can be mobile or stationary and used almost everywhere. Examples include laptops, PDAs, multimedia players, and mobile phones. These wireless devices have been so successful that large parts of the population own one in developed countries. There are also other less known devices used by armies or rescuing teams.

These devices provide the basis for many different applications. Each of them having its own strength and specificity, enabling them to communicate among them is a chance to find new uses. For example, having a multimedia player and a phone allows sending big files over a long distance, by enabling them to detect each other and communicate. A network of mobile phones could be set up out of range of a base station; soldiers can exchange data in real time while moving on a battle field; and communications can easily be established during fire or rescue operations.

For many of these applications the involved devices are rather small, possibly tiny. This implies some constraints on the software embedded on these devices such as scarce memory, limited lifetime, low computing power, and small bandwidth. Furthermore, the underlying network required to facilitate the communication has to operate in an ad-hoc fashion. Indeed, while mobile phones or laptops generally rely on a fixed infrastructure with access points, the kinds of applications described above do not have this requirement. Moreover, since small
CHAPTER 1. INTRODUCTION

devices usually have a low transmission power, they may well be out of direct range of the other devices. This is why nodes have to be able to communicate in a multi-hop fashion that is, relying on neighbors to forward data to nodes out of direct range (see figure 1.1). Such networks where potentially mobile nodes need to communicate peer to peer are called Mobile Ad-hoc NETworks, or MANETs[2].

![Figure 1.1: The radio of node S cannot reach node T. It therefore relies on other nodes to forward the communication to node T.](image)

Ad-hoc multi-hop routing among small devices, as well as many other challenges described later in this thesis, make such a network hard to set up and maintain. That is why robust and flexible routing protocols are needed for MANETs to be usable. Depending on which use is made of the network, the constraints are not the same and different trade-offs can be contemplated. Consequently, various approaches have been used for designing protocols for MANETs, and some specifications have been proposed for each approach.

One of these approaches is the so-called dynamic (or reactive) approach. Its main purpose is to minimize the traffic overhead caused by the protocol: nodes exchange routing information only when needed, before the real communication takes place. In this approach, adding communication latency is preferred to a constant overhead. Several protocols using this approach exist, including the experimental AODV (Ad-hoc On-demand Distance Vector) protocol[3]. As some challenges still exist to provide suitable protocols and since different variants are possible, research still continue within the field of dynamic protocols. Basing their work on previous experiments and researches, the MANET working group of the IETF designed another protocol that still has a draft status. This protocol is called DYMO, for DYNamic MANET On-demand[4].
1.2 Goal and Contributions of this Thesis

A particular and very promising application of MANETs is Distributed Sensor Networks (DSN)\(^\text{1}\). Though not always mobile, sensor networks share several characteristics with MANETs. They range from networks with a few, small, cheap sensor devices to networks with thousands of devices. They are used to monitor their environment, track targets or spread information; numerous applications have been foreseen and one can expect a lot of new applications to follow.

However, several issues raised by DSNs are difficult to solve and that is why it is currently an active research area. In this master’s thesis, we implement the DYMO protocol in the context of distributed sensor networks.

Given that the specifications of the DYMO protocol are still an IETF draft, there are only a few implementations available. The implementation developed in this thesis is thus an early implementation of DYMO. It allowed to provide feedback to the authors of the protocol about the simplicity of the protocol (simplicity was one of the main goals\(^\text{[4]}\)) and the clarity and correctness of the specifications. Several of our comments and reported mistakes have been included or fixed in the specification.

Moreover, we will see in chapter 5 that our implementation of the DYMO protocol is the first one to provide addressed unicast routing on TinyOS 2.0, the operating system of the implementation. We noticed that several people were interested in this feature, some of them even used our code at early stages of development. The implementation also provides various components that could be reused to implement other routing protocols on TinyOS.

The goal of the evaluation is to determine how well the DYMO protocol operates in a sensor network, especially the influence of the protocol on the power consumption of nodes. To achieve this, we performed experiments and measurements with a small number of real sensors.

1.3 Structure of this Thesis

The rest of the this thesis is structured as follows:

Chapter 2: Distributed Sensor Networks

Chapter 2 gives an introduction to the underlying concepts needed to understand the rest of the thesis. It introduces details about mobile networks and wireless sensor networks, as well as their routing protocols.

\(^{1}\)Sometimes also called Wireless Sensor Networks (WSN).
CHAPTER 1. INTRODUCTION

Chapter 3: Implementation Context

Chapter 3 first presents the hardware context, then the TinyOS platform, and finally the nesC language used to implement DYMO. It gives requirements imposed by hardware limitations and describes how these issues are tackled in TinyOS and nesC.

Chapter 4: The DYMO Protocol

Chapter 4 describes in detail the core functioning of DYMO and provides some examples. The format of the DYMO messages given in the specifications is also described, and the design of a modified format better suited for DSNs is presented. The motivations for this difference are presented in this chapter as well.

Chapter 5: Implementation

Chapter 5 describes the work done to implement the DYMO protocol on top of the TinyOS operating system. It documents the interface made available to applications, gives implementation details useful to know when using this interface, and explains the choices made during the implementation.

Chapter 6: Evaluation

Chapter 6 deals with the tests and experiments made with the DYMO implementation, and presents the results that have been obtained. This includes the introduction to the simulation environment and the description of the evaluation methods.

Chapter 7: Conclusion

Chapters 7 concludes the thesis with a summary of what has been achieved and learned, and it provides directions for future work.

1.4 Reading Guide

In order to take the most benefit from this master’s thesis, the reader should have a basic knowledge about network concepts and routing protocols. Understanding how technologies such as Ethernet or IP work should be sufficient. Details specific to distributed sensor networks are presented.

The chapters are ordered to progressively introduce all the concepts and issues, sometimes relying on what has been presented so far. As a consequence, readers
with sufficient background may not need to start from the first chapter: they can skip the parts they are familiar with but should read the rest in the proposed order.
Chapter 2

Distributed Sensor Networks

We saw in the introduction that the main motivation of this master’s thesis is the apparition and growth of a new type of network: the mobile ad-hoc networks (MANETs). Especially, Distributed Sensor Networks (DSN) represent an instance of the many possible applications offered by MANETs. DSNs are not always mobile networks but share many similarities with MANETs. This chapter introduces DSNs, including relevant details related with the thesis work. It also presents the main routing schemes and protocols for MANETs.

2.1 History

Research in sensor networks started just before the MANET technology, during the Cold War. It was conducted by the American army as a system of submarine sensors to track soviet submarines\[5\]. This was also the time when the United States and Canada deployed radars to protect themselves. Though the technology needed to meet the capabilities expected from the sensor networks was not yet available, research continued during the 1970s and 1980s. The focus was mainly on distributed algorithms and signal processing: sensors networks were expected to feature target detection and tracking as well as process sensed data without a central component in the network architecture.

Some good results were obtained given the state of the art. For example, Advanced Decision Systems developed a distributed algorithm able to track targets in a dense traffic or hard context\[6\]. MIT created hardware and software to track an aircraft with networked wireless acoustic sensors\[6\].

As both military and civil research began to see the numerous possible applications of DSNs, they also realized that it could even be better using a more dynamic approach. Sensor devices could be deployed on soldiers or mobile devices. Net-
works could also be made more flexible by allowing sensors to be moved without reconfiguring the network.

At the same time, in the 1990s, MANET routing schemes were becoming mature or at least they provided useful and interesting results in some fields. MANETs and DSNs naturally met to investigate how ad-hoc sensor networks could be used and improved. The recent apparition of personal devices like laptops, digital assistants, mobile phones, multimedia players, as well as the ability to build micro-electromechanical systems (MEMS)[7] tiny enough to fit in a cubic millimeter[8], and the development of fast wireless standards like IEEE 802.11, were all springboards to new applications of MANETs and DSNs. This generated much interest in this area.

2.2 Applications

With the technology that is or shall be available, applications of DSNs are only limited by imagination. A lot of ideas have already been proposed. One can separate them in three main categories, which are discussed in the following.

2.2.1 Monitoring

This category is concerned with sensor networks that are generally not mobile, monitoring targets that are not mobile either. This is useful when a large number of sensors is needed, or when the target is in an harmful or inaccessible area for humans. Environment and industrial monitoring are two good examples.

In order to monitor volcanic activity in an area or to sense climatic data, a number of sensors can be dropped from an aircraft over a potentially large area. The network has to configure itself, sense required data, and may decide which data should be kept or discarded. Data is either sent continually to a satellite for instance or only on human request. Data can also be directly analyzed and warnings would be sent when needed.

Another instance of inaccessible areas is locations inside industrial machinery. Measuring noise, vibration, oil, speed levels can be useful to be aware of machine performance or deterioration. However sensors may need to be placed in inaccessible regions to measure such data. The problem could be solved by putting standard sensors able to communicate with the existing network in the factory into the machine at construction time.
2.2.2 Tracking

This type of DSN is about possibly mobile but generally stationary sensors used to detect and track mobile targets. A main application is of course enemy tracking in military fields. With acoustic, temperature, pressure, infrared sensors, there are a lot of detection possibilities. A combination of these possibilities can improve accuracy, avoid false alarms, or provide more information.

However, detection and tracking can be useful in other domains also. For example, there are other highly secured but civil areas such as nuclear plants and banks. Sensors can also detect terrorist, biological or chemical threats in crowded areas. Habitat and traffic monitoring have interests too. Some sensors are already in use to count vehicles or animals and provide information about current activity, enabling to draw a real-time map[9]. Interesting results recently appeared for interfaces also. A small number of acoustic sensors fixed on any surface can be used to locate contacts with the surface[10]. This can transform any surface into a tactile user interface, allowing for instance a sheet of paper with a keyboard drawing to behave as a real keyboard.

2.2.3 Mobile Networks

Even though it might seem natural that sensor networks are non-mobile, interesting applications have been proposed where sensors are mobile. For example, military applications are possible. Soldiers equipped with sensors providing augmented reality and detecting threats can be seen as the nodes of a mobile network: information is spread to surrounding soldiers and propagated over the battlefield. It would improve the range of available information and facilitate coordination.

Traffic control is also concerned, but as a help for drivers. Sensors embedded on vehicles could send summaries about the traffic situation to any reachable vehicle. Information would propagate from vehicle to vehicle within a relevant range, helping users to avoid, e.g., traffic jams.

2.3 Research Challenges

To be feasible, all previously discussed applications pose technical challenges. In this section, we list all the important and common challenges in order to know what to focus on during the implementation and evaluation of a routing protocol for such networks.

- **Network discovery and control**
  
  Since in many applications, nodes are wireless and the network topology is not fixed *a priori*, nodes need to discover the identity of others. They may
also need to know their geographic location, either using Global Positioning System (GPS) or relative positioning algorithms. Moreover, in the context of mobile networks, nodes usually move and links can change, appear or disappear at any time. So the discovered network has to be controlled for nodes to be aware of these changes, and nodes should not rely on the presence of other mobile nodes.

- **Multi-hop routing**
  In the context of wireless networks, nodes need to be linked in a multi-hop fashion. It means that, given the limited range of radio transmitters, nodes need to rely on other nodes to forward messages to their final destination when the destination is not directly reachable. As a consequence, any communication has to find a (preferably short) path through the mobile nodes.

- **Hardware constraints**
  We have seen that software will be embedded on small or even tiny sensor devices. Such devices have obviously limited features, in that power, processing speed, memory and bandwidth cannot be expected to be as high as in conventional computers. Algorithms have to be aware of these limitations in order to change their behavior e.g., as power decreases.

- **Collaborative processing**
  As a solution to hardware constraints, but also because ad-hoc networks are not centralized, it is important to rely on distributed algorithms. A single node may not be able to handle all the sensed data and process it. However, distributed computing is a difficult task, and an active research area. It is either possible to spread data so that each node runs a part of the process, or to spread the code to be executed so that each node executes it on its own data[11].

- **Heterogeneity**
  A single network will include different kind of devices. For instance, devices may be different for each type of sensor, e.g. temperature or sound. The devices of a Personal Area Network (PAN) will certainly have various purposes (camera, phone, player or PDA). Therefore, nodes should not expect other nodes to have similar abilities.

- **Security**
  Since DSNs are a technology with many military applications, security and reliability are important characteristics. Communications should not be easily detected and should resist against intrusion. Encrypting information
might be quite hard given the drastic constraints of sensor devices. This applies even more to wireless networks where packets are easy to catch.

- **Querying**
  Most sensor networks will be autonomous and unstructured, they will measure data and process it without human intervention for long periods of time. For the information to be available on request, the possibility of querying the network is needed. It means that the network should be usable as a distributed database, where a user can request and retrieve data at any given location and time.

### 2.4 Routing Protocols

We saw in the previous section that a number of challenges have to be faced in the context of DSNs. Various protocols for routing in MANETs have been designed to meet these challenges, each of them having their own way of choosing how and when to run neighbor discoveries and compute shortest paths. This section presents main approaches currently used or implemented[12].

#### 2.4.1 Flat Routing Protocols

In contrary to the Internet with its huge number of nodes and its highly hierarchical structure, MANETs sometimes have a rather small number of nodes, and thus do not need any routing hierarchy. Routing protocols without any hierarchy that keeps knowledge of the whole network or internetwork are called flat protocols.

The main advantage of flat protocols is that they are simpler to design and implement than hierarchical protocols. Two common paradigms can be found among flat protocols, and a third one tries to take advantage of both paradigms, by combining them.

**Proactive Protocols**

Proactive protocols use the same approach as routing in the Internet. Nodes regularly exchange routing information and use it to build data structures. When a node has to send or forward a packet, it only has to read its routing table to know through which interface and to which address the packet should be sent. It is then easy and fast to send or forward a message, but the price to be paid is a prohibitive bandwidth overhead in order to continuously exchange routing information.

The exchanged data can take two forms, both used in the Internet: Distance Vector and Link State. The first one is based on having information about the
CHAPTER 2. DISTRIBUTED SENSOR NETWORKS

distance to the different nodes and is used for example in Destination-Sequenced Distance-Vector (DSDV) routing[13]. The second one is based on having information about the topology which is then used by the nodes to compute shortest paths. It is used in the Topology dissemination Based on Reverse-Path Forwarding (TBRPF)[14] and Optimized Link-State Routing (OLSR)[15] protocols for instance.

Figure 2.1 shows an example of a distance-vector protocol. At step 1, regularly and/or because of a topology change, node F broadcasts its routing table. When node C receives it, it updates its routing table accordingly, and may propagate the message, depending on the protocol (step 2). When a node wants to send a message, it reads the next hop in its routing table, and it is an error if it does not know the route.

Reactive Protocols

In order to reduce network traffic allocated to routing information, reactive protocols compute the path to a destination only when needed. It also relies on flooding: when a node wants to send a packet and does not have fresh information about its destination, it broadcasts a request for a path to this destination. The request is flooded and keeps track of the network path followed. The destination and nodes that have a fresh path to this destination respond to the request. The source of the request chooses the best response and caches it.

This is illustrated on figure 2.2. As long as no node wants to communicate, no message is sent. When node F wants to send a packet to node A, it broadcasts a route request targeted to A. When A receives it, it sends a route reply. How its
path is determined depends on the protocol. When F receives the route reply, it knows a route for A and can send its packet. The route will be deleted when it is not used anymore.

The flooding process operates only on demand, avoiding bandwidth overhead. It does not take care of inactive nor unused nodes. On the other hand, this incurs a latency that may not be acceptable in some contexts. Ad-hoc On-demand Distance Vector (AODV)[3] and Dynamic Source Routing (DSR)[16] protocols are well-known examples of reactive protocols, and DYMO, the protocol we implement and evaluate here, is also a reactive protocol.

**Hybrid Protocols**

The idea of hybrid protocols is using both proactive and reactive approaches, each one with a different scope. The network is divided into smaller groups (or clusters). Then, a proactive paradigm is used to collect information about nodes within the cluster, while a reactive paradigm is used for communications with nodes in distant clusters.

Sending a packet within a cluster (which is supposed to occur more often) is fast, and exchanged routing information is still rather small. Sending a packet out of a cluster will probably take longer, but this should happen not as often, and the prohibitive bandwidth overhead is avoided. The other side of the coin is a much more complicated design and implementation of such a protocol. The difficult part is deciding how the clusters are formed and how to handle changes in the topology. The Zone Routing Protocol (ZRP)[17] and the Hazy Sighted Link State (HSLS)[18] protocol are two good examples in this category.


2.4.2 Hierarchical Routing Protocols

Hierarchical protocols use the idea of clustering previously discussed and generalize it to a multilevel hierarchy. Each node belongs to the lowest level. They are divided into clusters that may overlap as in hybrid protocols. But here, nodes within a cluster elect a leader that belongs also to the upper level of the hierarchy, the set of leaders is also divided in clusters with their own leader, and so forth. Again, choosing the limits of clusters and their leader may be intricate.

When a packet is sent, it is forwarded from level to level until it reaches a cluster where a node is a (possibly indirect) leader of the destination. The packet then travels down in the hierarchy until it reaches a level-0 cluster to which the destination belongs. An example of such a protocol is the Hierarchical State Routing (HSR)[19] protocol.

2.4.3 Geographic Routing Protocols

Geographic protocols can be used when nodes have knowledge of the location of a given address. Thus they do not need to perform flooding, instead they forward the packet to only one node (or a set of nodes) on the path towards the destination. Simplest protocols use a greedy algorithm to do so: packets are forwarded to the directly reachable node that is closest to the destination. If the destination cannot be reached with this path, backtracking is used to find another path. The Greedy Perimeter Stateless Routing (GPSR)[20] protocol is an example of geographic protocol.

2.5 Conclusion

When research in distributed sensor networks started, it was more or less a vision, because both software and hardware challenges were far from being solved. But now the research community has tackled many of these challenges, at least individually. Some applications have already been realized with promising results. However, combining solutions to solve the different challenges is rather difficult, and still an active research area.

In this thesis, we will only consider a subset of these challenges, those implied by ad-hoc routing. Network discovery and multi-hop routing are of course the main ones, but hardware constraints are also involved. Indeed, algorithms and their implementation have to be conservative with the use of of data storage, packet transmissions, and data processing in order to save limited resources.
2.5. CONCLUSION

Though security is also a concern, we will not take security issues into consideration during the implementation. The reason is that DYMO is not designed for that purpose (but a variant of DYMO called SDYMO provides security features[21]).

Routing schemes presented here are only the most common ones, and each of them is used by several protocols. Though MANETs introduce new challenges, many of these protocols are based on ideas for stationary networks, especially peer-to-peer networks. Indeed, these kinds of networks have some similar concerns, but at an upper layer in the protocol stack. Before discussing the implementation and evaluation, we will review the implementation context and the details of the DYMO protocol in the next chapters.
Chapter 3

Implementation Context

This chapter deals with the hardware and software on top of which we have implemented the DYMO protocol. It illustrates physical limits of sensor devices with some figures and describes the primitives provided by the operating system and the programming language used for the implementation.

3.1 Motes

The recent advances in technology making it possible to produce cheap and small sensor devices led to a new computing concept: motes\(^1\). Named after their aimed size, these new devices have numerous applications as we saw in section 2.2. This section sums up the birth of motes and presents their main characteristics.

3.1.1 History

Motes were first introduced by the University of Berkeley in 1998, and the first generation was called *RF motes*\(^{22}\). Soon followed improved revisions called *miniMotes* (smaller) and *weC motes* (remotely programmable). They were already rather small, but with very limited resources: 8 KB of program memory, 512 bytes of RAM, a 4 MHz CPU, and a 10 Kbps radio with a 20 meter range.

During the following years, new platforms were designed: *René* in 1999, *Dot* in 2000 and *Mica*\(^{23}\) in 2001. The latter offered much better capabilities than its predecessors. It features 128 KB of program memory, 4 KB of RAM, a 4 MHz CPU and a 40 Kbps radio with a 30 meter range - while consuming only half the power of a weC mote. This second generation became consequently popular in the research community, and was manufactured and commercialised by

\(^{1}\)The name “smart dust” is also used, but generally refers to the smallest devices.
Crossbow\cite{24} in the United States. They come in two different forms (with the same capabilities): rectangular (58 x 32 x 7 mm) or circular (25 x 7 mm).

The University of Berkeley also developed a project called Smart Dust\cite{8}, a mote with a size of no more than a few cubic millimeters. It uses a solar cell as power supply, and tiny mirrors and a laser diode to communicate.

In the meanwhile (2001), UC Berkeley also developed an operating system specifically designed for motes named TinyOS, described in section 3.2. The weC platform has been the first to use it. TinyOS evolved along with the birth of new platforms to support them and offer new features.

In 2003, Moteiv\cite{25} created the Telos platform. Its purpose was to focus on power consumption rather than memory and CPU capabilities. The result is a mote with half the memory of a Mica mote, but with a drastically improved power consumption and bandwidth: it uses 3 mW in active power mode (8 mW for the best mica platform) and 6 $\mu$W in sleep mode (75 $\mu$W for Mica platforms); and it can transmit data at up to 250 Kbps.

\subsection*{3.1.2 The TelosB Research Platform}

In response to the great interest in this field from the research community, a revision of the Telos platform called TelosB was made for research purpose. It features a USB connector to program the mote and retrieve data from a conventional computer. Since USB is a widely used standard present on any modern computer, it is handy to make research and development with TelosB motes. This is the platform used for the evaluation presented in this master’s thesis, as well as micaZ (derived from Mica, with the radio chip of Telos platforms) for simulations.
3.1. MOTES

Figure 3.2 shows the main characteristics of the TelosB platform and a photo of a TelosB mote[26].

<table>
<thead>
<tr>
<th>Processor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>16-bit RISC</td>
</tr>
<tr>
<td>Clock rate</td>
<td>8 Mhz</td>
</tr>
<tr>
<td>Program Flash memory</td>
<td>48 kB</td>
</tr>
<tr>
<td>Measurement serial Flash</td>
<td>1024 kB</td>
</tr>
<tr>
<td>RAM</td>
<td>10 kB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Radio Transceiver</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency band</td>
<td>2.4 to 2.483 GHz</td>
</tr>
<tr>
<td>Data rate</td>
<td>250 kbps</td>
</tr>
<tr>
<td>Outdoor range</td>
<td>75 to 100 m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Battery</td>
<td>2 AA batteries</td>
</tr>
<tr>
<td>Size</td>
<td>85 x 31 x 6 mm</td>
</tr>
</tbody>
</table>

Figure 3.2: TelosB platform characteristics and photo[26]

3.1.3 Software Requirements

The hardware constraints presented in section 2.3 and illustrated in figure 3.2 incur several requirements on the software to run on motes. The two main ones are energy and memory consumption.

Energy Saving

Nowadays, motes achieve a very low energy consumption, but one should keep in mind that motes may rely on their battery for a very long time. For instance, a sensor network monitoring a habitat has to last several years, a sensor in an industrial machine has to live longer than the machine. Also, motes are often so miniaturized that it is impossible to replace the battery. As a result, such motes need to be replaced when the life of their battery reaches its end.

Consequently, software should be designed to save energy. Monitoring networks that sense data only on a regular basis should use a power saving mode between measurements, though it may be difficult to predict when to wake up. In many applications, decreasing performances of the network along with the power of the motes would be better than suddenly cutting communications. For instance, motes tracking targets in a critical context should progressively decrease the amount of information processed if they begin to run out of energy, rather than merely stop during the tracking process. As the radio transceiver is a big energy consumer, the amount of communications should also be adapted to the battery level.
CHAPTER 3. IMPLEMENTATION CONTEXT

Memory Saving

We have also seen that Flash and RAM memory is limited. During the implementation of DYMO, it should be kept in mind that this is only an extension of the operating system. The memory used by the OS and the protocol implementation should allow the application to be the main memory consumer. Therefore, data structures should be chosen and used judiciously, and the compiler has to optimize the code size if possible.

Reactive Concurrency

Lastly, motes need to support a reactive concurrency. Indeed, they will not run sequential code as usual computers do. Instead, they will be driven by events, generally environmental data or messages receipt, which both need real-time responses. Moreover, these events may happen at any time and can thus be concurrent. As a consequence, software applications have to be able to concurrently run usual processing and event processing.

All these requirements need a strong cooperation between hardware and embedded software. For that purpose, the University of Berkeley also developed an operating system called TinyOS and a C dialect called NesC.

3.2 TinyOS

TinyOS\cite{27} is an open-source\footnote{The source code is available on Sourceforge.} operating system (OS) specifically designed for networked embedded sensors. The aim of the design team was to tackle challenges inherent in sensor networks, as presented in section 2.3. It is also designed to be open to contributions and improvements, because this field is still novel and actively researched. It is tiny (the core kernel need only 40 bytes of RAM), modular, flexible, and it features the software requirements previously discussed\cite{28}. Since TinyOS 1.0, NesC (discussed in the following section) is the language used for implementing TinyOS and providing its programming model. In November 2006, the developers released TinyOS 2.0. It introduces significant improvements in the model and implementation, but is not compatible with TinyOS 1.x. Our implementation of DYMO is made on TinyOS 2.0.
3.2. TINYOS

3.2.1 Component Model

TinyOS applications are assemblies of components, each of them being allocated to a specific task. The programming model in NesC allows components to share the same specifications and have different implementations. It is therefore possible to encapsulate drivers in components, so that programmers do not have to change the code to use their applications on some other hardware. Moreover, we will see that a design of perfectly independent components allows to include only the required ones when the application is built.

3.2.2 Execution Model

TinyOS’s execution model features reactive concurrency as required by the nature of sensor networks. To avoid a memory overhead, TinyOS does not provide a thread system because keeping the state of each thread is expensive. Instead, a system of tasks is used, which provides a deferred computation mechanism. Tasks are procedures that a program can post when time considerations are not critical for this computation. The posting act immediately returns, and tasks are executed later. They do not interfere with each other: when a task is posted, it must wait for the completion of the previous ones before running. Tasks are managed by a scheduler, a part of the TinyOS kernel, that typically handle tasks with a FIFO policy (but other policies are available).

Another concurrency source is introduced by events. Practically, events represent either a hardware interrupt or the end of a task, and can be triggered at any time. It means that they may preempt an ongoing computation (a task or even another event), contrarily to tasks.

TinyOS’s programming style combines these two concurrency mechanisms to divide programs in split-phase operations. Any non-trivial operation is encapsulated in one or several tasks, and they are executed according to the scheduler’s policy. The completion of a task is signaled by an event. Furthermore, events are triggered upon message or data receipt. This simple concurrency model allows motes to be reactive to their environment while having a low CPU and memory usage.

3.2.3 Network Interface

TelosB and MicaZ (see section 3.1.2) use an IEEE 802.15.4 radio, which implies that addresses are 16 bits long. TinyOS provides the components that allow sending and receiving one-hop messages through the radio (this is discussed in details in section 5.1.1). However, TinyOS 2 distribution does not come with any network
layer implementation. As a result, DYMO will be implemented on top of the link layer and we will provide a network layer to use it.

3.3 nesC

Soon after TinyOS’s development started, a programming language called NesC was designed to embody TinyOS concepts[29]. It is derived from C, with straightforward extensions, and a few restrictions to improve optimization and debugging.

This section uses an example to illustrate all the characteristics of nesC. The example is a very simple application in which a LED of the mote blinks every second. It uses three components provided by TinyOS to manage the hardware: the Main component (which implements Boot, the equivalent of the main() C function); a Timer; and the LED manager (see figure 3.3).

![Figure 3.3: Interactions between the components of our example.](image)

3.3.1 Component Model

NesC provides the syntax to organize the code in independent components. The specifications of a component must be provided in one or several interfaces to be used by other components. Interfaces are bidirectional: they provide a number of commands (called by the user), but can require their users to provide some event handlers (called by the component).

In our example, the Timer component triggers an event regularly and can be started or stopped by the user component. Figure 3.4 shows the declaration of the interface specifying the (simplified) features of the Timer component.

These interfaces can then be implemented by one or more components, which can be of two kinds: modules and configurations.

Modules contain the code implementing one or several interfaces. Figure 3.5 shows an example. In the first part, it declares which interfaces are implemented with the keyword provides. If the module relies on other components, this is also
interface Timer {
    /* Set the period and start the timer */
    command void startPeriodic(uint32_t period);

    /* Cancel the timer */
    command void stop();

    /* Signal the end of a period */
    event void fired();
}

Figure 3.4: A code example to declare an interface in NesC.

declared in this part with the keyword uses. In a second part called implementation, the module implements the commands of the provided interfaces and the event handlers of the used interfaces. Events of the provided interfaces can be signalled to the user with the keyword signal, and commands of the used interfaces can be called with the keyword call. This part can declare and use local variables and functions too.

In a nutshell, NesC applications are organized in components, each of them providing (except the top-level module) and possibly using interfaces as illustrated in Figure 3.6. Relying on interfaces instead of components lets the programmer choose the implementation adapted to his needs and hardware while keeping a flexible way to change the dependencies without altering the code. As a consequence, the programmer needs a way of linking the components together to build the complete application.

This is done by the other kind of components: the configurations. Configurations are used to link interfaces and components (modules or configurations) in order to specify which implementation is used for each interface. This process of linking components and interfaces is called wiring. Configurations are components because they can use and provide interfaces too. But instead of implementing the commands and event handlers, they rely on components providing and using these interfaces. In the implementation part, the components used for the wiring are declared, and they are linked together with arrows. “A -> B” means “A uses the implementation of B”, whereas “A < B” means “A provides the implementation to B”; using one syntax or the other is a matter of choice. A “=” is used to wire a component to an interface provided by the configuration. This is illustrated in figure 3.7.

Figure 3.8 shows the graph specified by the configuration. This style of dia-
module TimerM {
    provides Timer;
    uses Alarm;
}

implementation {
    /* Local variables */
    uint32_t p;

    /* The module must implement the commands */
    /* of the provided interfaces */
    command void startPeriodic(uint32_t period) {
        p = period;
        call Alarm.start(call Alarm.getNow(), p);
    }

    command void stop() {
        call Alarm.stop();
    }

    /* The module must implement the events of the used interfaces */
    event void Alarm.fired() {
        call Alarm.start(call Alarm.getNow(), p);
        signal Timer.fired();
    }
}

Figure 3.5: A module implementing Timer and using Alarm, a non-periodic timer.

Figure 3.6: The components of our example and the associated interfaces.
configuration TimerC {
  provides interface Timer;
}
implementation {
  components TimerM, AlarmC;
  Timer = TimerM.Timer;
  AlarmC.Alarm <- TimerM.Alarm;
}

class Example {} //Top-level configuration
implementation {
  components AppliM, MainC, TimerC, LedC;
  AppliM.Boot -> MainC.Boot;
  //It is generally optional to name the interface twice
  AppliM.Timer -> TimerC;
  AppliM.Led -> LedC;
}

Figure 3.7: Two wiring examples.

The wiring process allows the compiler to know exactly which code is used or not in the application, and include only the required code in the final binary file. This reduces the code size as much as possible, resulting in an application-specific operating system.

We have only covered the basic features of nesC here. It is for instance possible to parameterize modules and interfaces. Parameterized modules let users choose for instance the maximum size of a data structure or the precision of some results according to their needs. Parameterized interfaces allow a single component to provide several independent instances of the same interface, they are identified in the implementation and the wiring with these parameters (generally integers).

Another example is the possibility to wire several command calls to a single implementation (“fan-in”), and to wire a single command call to several implementations (“fan-out”), provided that a function to combine the return values is defined.
Figure 3.8: The component diagram of our example (dependencies of MainC, TimerC and LedC are not represented).

### 3.3.2 Concurrency

NesC implements the execution model of events and tasks described in section 3.2.2. Declaring a task is as straightforward as declaring an event or a command: the keyword `task` is placed before the declaration of a function, which must have the return type and parameters `void`. Then to post the task, a regular function call preceded by the keyword `post` is enough. Figure 3.9 shows an example of use: it is the implementation of a send interface. The command posts a task that will effectively send the data and immediately returns. The sender will know that the data are sent thanks to the sendDone event.

We saw in section 3.2.2 that though tasks do not interfere with each other, events can preempt running code at any moment. As a consequence, data races can occur if an event shares data with a task or an other event. To avoid data races, nesC provides an atomic environment, in which hardware interrupts are disabled to prevent events from being signaled. The nesC compiler is able to distinguish asynchronous code (reachable from a hardware interrupt) from synchronous code (only reachable from tasks), and requires a race-free invariant: If a variable $x$ is accessed from asynchronous code, then any access of $x$ outside of an atomic statement is a compile-time error. This enables the compiler to find most of the potential data races.
3.3.3 Additional Features

In addition to the component model and concurrency features, nesC alters the C language to address DSN challenges. First, nesC does not allow separate compilation of components: the whole application must be wired to be compiled. This lets the compiler analyze the whole program, which is useful for data-race detection and memory optimization.

Also, the expressive power of C is reduced by forbidding dynamic memory allocation and function pointers. As a result, it is possible to know at compile time the amount of memory used and the complete call-graph. This is again to facilitate the whole program analysis.

These differences from C can sound prohibitive, but the limited size of memory prevents programs from being big and requiring dynamic memory allocation.

3.4 Summary

The first attempts to dramatically reduce the size of sensor devices to embed them in what is called motes are only ten years old. Yet, the involved technology is already close to what was aimed at, allowing to make tiny and power saving motes. The results are promising amazing applications, that could be reached by 2010[30].

Because the goal is to reduce the size of motes rather than increasing their
capabilities, the software embedded in motes face various challenges to address limited physical resources and characteristics of distributed sensor networks. To tackle this challenges, UC Berkeley designed TinyOS, a free, community-oriented, operating system for motes.

As an OS, TinyOS is linked to the hardware, but need to be flexible enough to be used on the various mote architectures. Every driver is encapsulated using TinyOS’s component and event model, so that programmers can communicate with the hardware transparently, without adapting the code to the hardware.

For the code size to be as small as possible, TinyOS does not provide a ready-to-use OS. It is rather a framework for building an application-specific OS. TinyOS distribution comes with a set of components that developers can use and wire to their applications. At build time, only the used components and functionalities are included in the binary.

TinyOS’ programming model is provided by NesC, a dialect of C designed to embody TinyOS concepts. It features a component model and concurrency mechanisms in order to be adapted to distributed and embedded sensing applications. It also restricts some functionalities of C so that the compiler has a better understanding of the program and can optimize it.
Chapter 4

The DYMO Routing Protocol

This chapter presents the details of the DYnamic Manet On-demand (DYMO) routing protocol[4]. It is a reactive routing protocol mainly based on ideas from AODV[3]. It determines multi-hop unicast routes on-demand in a dynamic network topology.

We will use a running example in this chapter to illustrate the operation of the protocol. Figure 4.1 shows a representation of this example network (neighbor nodes are connected with lines). We assume that node A wants to communicate with node H. To avoid confusion, we use a different vocabulary for communications at the link layer and the network layer. At the link layer, packets are exchanged from a sender to a receiver, which are neighbors. At the network layer, packets are sent from an originator to a target.

Figure 4.1: A small instance of MANET, where A wants to communicate with H.
4.1 Overview

As a reactive protocol, DYMO does not explicitly store the network topology. Instead, nodes compute a unicast route towards the desired destination only when needed. As a result, little routing information is exchanged, which reduces network traffic overhead and thus saves bandwidth and power. Also, since little routing state is stored, DYMO is applicable to memory constrained devices like motes.

When a node needs a route, it disseminates a Route Request (RREQ), which is a packet asking for a route between an originator and a target node. The packet is flooded to the entire network or within a number of hops from the originator (see figure 4.2). When the packet reaches its target (or a node that has a fresh route towards the target), the node replies with a Route Reply (RREP). A route reply packet is very similar to a route request, but it follows a unicast route and no reply is triggered when the target is reached (see figure 4.3).

When nodes receive a RREQ or a RREP, they cache information about the sender and the originator, so that they know a route to the originator that can be used later (if it is fresh enough) without sending a RREQ. The nodes have the possibility to accumulate the path followed by the packet in the packet itself. So, when nodes disseminate a RREQ or RREP, a lot of information can actually be obtained from the packet, much more than a route between two nodes.

When routes have not been used for a long time, they are deleted. If a node is requested to forward a packet through a deleted route, it generates a Route Error (RERR) message to warn the originating node (and other nodes) that this route is no longer available (see figure 4.4).

As another route maintenance mechanism, DYMO uses sequence numbers and hop counts to determine the usefulness and quality of a route. We explain the operation of DYMO in more detail in the following sections.

4.2 Route Discovery

4.2.1 Routing Messages

The messages exchanged during the process of route discovery, that is RREQs and RREPs, are called routing messages. They always include the target and originator addresses, as well as a hop limit and a hop count which prevent a routing message from being forwarded several times.

Since hop counts are used to determine the quality of a route, a mechanism to ensure loop freedom and avoid the count-to-infinity problem is needed. DYMO
4.2. ROUTE DISCOVERY

Figure 4.2: Node A wants a route to node H, it broadcasts a RREQ.

Figure 4.3: When the RREQ reaches H, it knows it can reach A through G, G knows it can reach A through F (for instance), and so on. A RREP is issued along this path. When it reaches A, a bidirectional route has been established between A and H and they can communicate.

Figure 4.4: When F realizes that G has moved, it broadcasts a RERR announcing that G and H cannot be reached via F anymore. B and C do not forward the RERR because they know a route through C and D respectively.
uses sequence numbers, as introduced in the distance-vector protocols. So, all nodes have a sequence number and include it in the DYMO packets they send. Nodes increment their sequence number whenever they estimate that the routing information of the other nodes is too old.

This applies to both RREQs and RREPs, they are therefore very similar. The differences are only algorithmic: the way nodes decide if they should increment their sequence number before sending the packet, as well as how the packets are forwarded (since RREQs are broadcast and RREPs are unicast).

4.2.2 Packet Processing and Forwarding

When nodes receive a routing message, they look at all the included routing information. Based on sequence numbers and distances counted in hops, they decide whether the routing information is better than what they already know, and update their routing tables consequently (the decision is explained in section 4.3.1).

If a piece of routing information about a node is judged useful, it is assumed that the node can be reached through the sender of the packet (at the link layer), and the routing table is updated accordingly. Therefore, links are assumed to be bidirectional. Also, during this process, the content of the message is updated. Hop counts are incremented and routing information that was not considered useful is removed from the message so that it is not propagated further.

After having judiciously updated their routing tables, nodes can append additional routing information to a routing message before forwarding it. They can add information about themselves or about other nodes, whenever they believe it will be useful for surrounding nodes. This may decrease the number of RREQs and enable quicker RREPs. Indeed, when a node receive a RREQ with a target for which it knows a fresh route, it can send an intermediate RREP instead of forwarding the RREQ. As a result, the originator of the RREQ receives the RREP sooner, and the RREQ is not propagated further, which reduces the traffic overhead. Appending routing information to routing messages increases the chances that other nodes will send intermediate RREPs at the expense of bigger packets.

4.3 Route Maintenance

Since DYMO applies to a context with a highly dynamic network topology, routes need to be actively monitored after having been established. The protocol does not impose a monitoring mechanism, but specifies how this can be done with route timers.
4.3. ROUTE MAINTENANCE

4.3.1 Length and Freshness of Routes

A part of the route maintenance is keeping routes fresh and as short as possible. We saw that this is achieved by systematically inspecting received packets.

To compare pieces of routing information, nodes use sequence numbers to check that incoming information is fresh enough. Not only fresh information is better, it also ensures loop freedom. When the information is fresh and loop-free, only the shortest path available is kept. This is determined by comparing hop counts, which is the distance from the considered node counted in hops. For instance, on figure 4.2, nodes F and G forward the RREQ only once, and they do not update their routing table with the second RREQ, because it does not have a better hop count.

4.3.2 Link Monitoring

Each time a node creates or updates a route in its routing table, it can monitor the route with associated timers. To ensure that nodes can rely on the information they receive in RREPs, nodes are expected to keep their routes for a minimum amount of time. Routes also have a maximum age, because keeping a route for a long time in a dynamic context is not safe and can lead to forwarding loops, and also because we do not want to spend memory for a route that is not actively used.

Between these minimum and maximum ages, routes are kept as long as they are used. Each time a packet is forwarded through a route, the timer for this route is updated. When the timer expires, the route can be deleted. Nodes may also use other methods to monitor links and routes, for instance a neighbor discovery protocol or a link-layer feedback.

4.3.3 Route Errors

When the route monitoring process detects a broken route, a broken flag is set for the corresponding route entry. If a node tries to use this route, a route error (RERR) message is flooded in the network. The RERR contains information about the unreachable node (node G on figure 4.4 for instance), and may also contain information about nodes (such as node H) previously reachable through this node. A RERR warns other nodes that some nodes are no longer available through the sender of the RERR.

Upon receiving a RERR, nodes that do not have superior information about unreachable nodes set a broken flag for the relevant route entries. Unless the routing information included in the RERR is considered too old, the RERR is forwarded to all neighbors.
4.4 Packet Format

This section describes the packet format as specified by the DYMO protocol. However, for several reasons discussed here, we have implemented a more compact format, which will also be presented below.

4.4.1 Original Format

The DYMO message format conforms to packetbb[31]. Packetbb is an attempt to provide a generalized packet and message format for routing protocols in MANETs. Packetbb messages are made up of a header, an overall Type-Length-Value (TLV) block (which is empty for DYMO messages), some address blocks, and possibly TLV blocks associated to each address block.

A TLV is a generic format to transport data. It is made of three consecutive fields: the type of data, the length of the data, and the data. A TLV block is a set of consecutives TLVs.

In packetbb, every message contains a TLV block that applies to the whole message. It is not used in DYMO. Then, several lists of addresses may follow and each address block can have its own TLV block, where the data only apply to a range of addresses from the address block. It is used in DYMO to attach routing information to addresses.

Routing Messages

DYMO messages do not include originating and target addresses in the message header, because this would not allow to attach routing information to these addresses. Instead, the protocol specifies that the target and originator addresses are included in an address block placed immediately after the (empty) TLV block.

When nodes want to append additional information to the message, they add the addresses after the first two addresses or in additional address blocks. If they are known, the sequence number and the hop count of each address can be included in associated TLV blocks.

Figure 4.5 shows the format of a RREQ message, as it would be received by node F in the example of figure 4.2 (provided nodes B and E append their routing information to the packet). It has been adapted to the 16-bit addresses used in the implementation (see section 3.2.3). The second byte, the message semantics, means that the hop limit and hop count values are included, but not the originator address. This byte always has this value for DYMO messages. The Hop Limit, the Hop Count and all the included hop counts are updated each time the packet is processed. Node A set the hop limit to 10 for example, that is why it has a
value of 8 when the packet arrives to node F. After the empty TLV block, there is the address block, with four addresses. Its semantics means that the addresses share a common head. This allows to save one byte for each address. Since the addresses are 2-byte long, the length of the head is useless, but it is included because packetbb was originally designed for IPv4 and IPv6. Then a address TLV block is attached to the address block. The semantics of each TLV block mean that they contain per-address values, and only for a range of addresses specified by the fields named start and end.

```
<table>
<thead>
<tr>
<th>RREQ</th>
<th>0.0.0.0.0.0.1</th>
<th>Message size = 37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hop Limit = 8</td>
<td>Hop Count = 2</td>
<td>TLV Block Size = 0</td>
</tr>
<tr>
<td>NumAddr = 4</td>
<td>0.0.0.0.0.0.1.0</td>
<td>Head Length = 1</td>
</tr>
<tr>
<td>H Tail</td>
<td>A Tail</td>
<td>B Tail</td>
</tr>
<tr>
<td>TLV Block Size = 21</td>
<td>SeqNum TLV Type</td>
<td>0.0.0.0.0.0.0</td>
</tr>
<tr>
<td>TLV length = 6</td>
<td>start = 1</td>
<td>end = 3</td>
</tr>
<tr>
<td>(com)</td>
<td>B SeqNum</td>
<td>E SeqNum</td>
</tr>
<tr>
<td>(com)</td>
<td>HopCtrl TLV Type</td>
<td>start = 1</td>
</tr>
<tr>
<td>0.0.0.0.0.0.0</td>
<td>TLV Length = 3</td>
<td>A Hop Count = 2</td>
</tr>
<tr>
<td>E Hop Count = 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 4.5: An instance of RREQ message, with two additional addresses and associated routing information. A RREP would be similar.

**Error Messages**

Error messages do not include the originator address, since including an address in a error message means this address is unreachable. For the same reason as routing messages, the message header do not contain any address. Instead, all addresses are part of an address block, and corresponding routing information is in associated TLV blocks. The first address of the message is the address of the node responsible of the broken link. Addresses of unreachable nodes because of the broken link may follow.

The format is quite similar to routing messages since they both conform to the packetbb format. Figure 4.6 shows the RERR message that could be received by node B on figure 4.4, also adapted to 16-bit addresses.
CHAPTER 4. THE DYMO ROUTING PROTOCOL

Figure 4.6: An instance of RERR message, with two addresses and their sequence number.

4.4.2 Implemented Format

Motivation

Using the packetbb format has two disadvantages in the context of DSNs.

The main disadvantage of the packetbb format is that messages cannot be processed linearly. Indeed, since addresses and associated data are separated in different blocks, the forwarding process need to read the whole address block before reading the TLV block and processing the first address. A separate data block enables to associate a single value to a range of addresses, but only the multi-value option is used in DYMO.

The other disadvantage of using packetbb for DYMO is a small waste of bytes because some parts of packetbb are never used but are imposed by packetbb specifications. These parts are the semantics of the message header (because it always has the same value) and the TLV block size (which is always nil). Moreover, several bits of the semantics of the block headers are never used, and only a few TLV types are needed. This is a detail, but one should pay a lot of attention to memory and bandwidth usage in the context of motes.

As a consequence, even if the algorithms of our implementation conform to the DYMO specification, the format has been slightly redesigned in order to exploit the advantages of packetbb, but avoid the disadvantages. Of course, this will prevent this implementation to interoperate with others. But the goal of this master’s thesis is not to provide a ready-to-use implementation, but rather to evaluate DYMO and see how it can be used in sensor networks.

For the packets to be processed linearly, any routing information associated

---

1One could use a single value when several addresses share the same sequence number or hop count, but it would be much more complicated to generate and process messages.
4.4. PACKET FORMAT

with an address directly follows the address. With such a format, once an address
is read, there is no need to access it again and the associated information can be
immediately processed.

However, the routing information associated with an address is not always the
same. It can contain a sequence number, a hop count and a maximum age, each of
them being optional. Therefore, addresses with the same amount of information
are gathered in an address block, which has a header specifying which options are
included after each address. This header also specifies whether plain addresses or
head and tails are used, since this feature can reduce the size of packets.

To save bytes, useless bytes were removed. Furthermore, the format gather
the packetbb address blocks, and their associated address TLV blocks into one
block, which we call an address block. Consequently, all the semantics fields of
the previous blocks are gathered into a single byte.

Specifications

The rationale above led to the following format. We describe it in a similar manner
as packetbb specifications[31], using regular expressions where “?” indicates an
optional element, “+” indicates an element present one or more times. These
operators only apply to the preceding element.

- `<message> = <message-header><address-block>+`

- `<message-header> = <type><size><hop-limit><hop-count>`
  where: `<type>`, `<size>`, `<hop-limit>` and `<hop-count>` are described in
  packetbb specifications.

- `<address-block> = <semantics><num-addr><head>?<addr-info>+`
  where: `<semantics>` is described below;
  `<num-addr>` is an 8-bit integer, the number of `<addr-info>` blocks;
  `<head>` is an 8-bit prefix common to all the addresses of this address block.

- `<addr-info> = <address><seqnum>?<hop-count>?<max-age>?`
  where: `<address>` is either an 8-bit suffix or a 16-bit address, depending
  on the presence of `<head>`;
  `<seqnum>` is a 16-bit integer, the sequence number of the described node;
  `<hop-count>` is an 8-bit integer, the distance in hops from the processing
  node to the described node;
  `<max-age>` is a 16-bit integer, the maximum number of milliseconds that
  the associated information can be kept.

- `<semantics>` is an 8-bit field where:
– bit 0 (head) is set to 1 if <head> is present. In that case, the addresses of the nodes in this address block are <head><address>, otherwise they are <address>;
– bit 1 (seqnum) is set to 1 if <seqnum> is present;
– bit 2 (hop-count) is set to 1 if <hop-count> is present;
– bit 3 (max-age) is set to 1 if <max-age> is present;
– bits 4-8 are all set to 0.

Figures 4.7 and 4.8 show two instances of the format. The messages contain the same information as the instances of figures 4.5 and 4.6.

<table>
<thead>
<tr>
<th>RREQ type</th>
<th>Size = 24</th>
<th>Hop limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hop Count</td>
<td>0.0.0.0.0.0.0.0.0.1</td>
<td>NumAddr = 1</td>
</tr>
<tr>
<td></td>
<td>0.0.0.0.0.1.1.1</td>
<td>NumAddr = 3</td>
</tr>
<tr>
<td>A Tail</td>
<td>A SeqNum</td>
<td>A HopCnt = 2</td>
</tr>
<tr>
<td>B Tail</td>
<td>B SeqNum</td>
<td>B HopCnt = 1</td>
</tr>
<tr>
<td>E Tail</td>
<td>E SeqNum</td>
<td>E HopCnt = 0</td>
</tr>
</tbody>
</table>

Figure 4.7: An instance of RREQ message with our simplified format.

<table>
<thead>
<tr>
<th>RERR type</th>
<th>Size = 14</th>
<th>Hop limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hop Count</td>
<td>0.0.0.0.0.0.0.1.1</td>
<td>NumAddr = 2</td>
</tr>
<tr>
<td>G Tail</td>
<td>G SeqNum</td>
<td>Head</td>
</tr>
<tr>
<td>H SeqNum</td>
<td></td>
<td>H Tail</td>
</tr>
</tbody>
</table>

Figure 4.8: An instance of RERR message with our simplified format.

One may notice that the format include a 16-bit size field, though TinyOS packets are limited to 255 bytes, and often have a much smaller limit. Bigger packets are useful for networks that need a path accumulation of more than about 50 addresses, which is a quite high topology diameter. These networks would need a fragmentation support to be used on TinyOS.
4.5 Conclusion

What stands out from DYMO specifications is its simplicity. There are only three message types, and their processing is similar: judging included information with a common algorithm, then updating the routing table if necessary. Generating the messages and updating the routing table are simple algorithms, even more simple to implement with our simplified format. Due to its reactive operation, DYMO does not require any storage of complex data structures such as graphs, which alleviates intricate data manipulation and exploring. We can thus already say that DYMO has achieved one of its main goals.
Chapter 5

Implementation

This chapter presents the components that we have implemented for the DYMO protocol, in order to provide a multi-hop network service. The first of the three sections introduces more details about the existing communication features of TinyOS, whereas the two other sections describe our implementation and its functioning.

5.1 Integration in TinyOS

Networking is a critical part of sensor networks and of TinyOS. We detail here the networking principles of TinyOS as well as the existing components that TinyOS provides.

5.1.1 Link Layer

The TinyOS link layer relies on Active Messages (AM). Active Messages are packets that specify a handler ID in their header. They are called active messages because they trigger the invocation of the associated handler upon receipt, preempting any ongoing computation. They provide an unreliable one-hop datagram service. The implementation is provided by a set of interfaces and associated components described in the following. For programmers to share common conventions about network interfaces, the TinyOS documentation describes the choices made to design the network and AM interfaces, and how upper layers should be designed in order to meet these conventions[32].

The basic interface is Packet (see figure 5.1), which provides access to the payload field that is common to all datagram protocols. Protocols must provide a specific interface in order to give access to the relevant fields of their format. To avoid copy of data and memory consumption, the components of the different pro-
interface Packet {
  /* Completely clears the packet */
  command void clear(message_t* msg);

  /* Returns the value of the length field */
  command uint8_t payloadLength(message_t* msg);

  /* Sets the value of the length field */
  command void setPayloadLength(message_t* msg, uint8_t len);

  /* Returns the maximum length of the data payload */
  command uint8_t maxPayloadLength();

  /* Returns a pointer to the data payload */
  command void* getPayload(message_t* msg, uint8_t* len);
}

Figure 5.1: The Packet interface.

tocol layers share the same packet buffer implemented via a message_t pointer. They rely on the underlying packet interfaces to know where their data begins.

The AMPacket interface provides accessors to three additional fields: destination, source and type (see figure 5.2). The type of an active message is a one-byte integer to identify the handler of the message. Though the AMPacket interface also provides commands to set these fields, this is solely for specific purposes beyond the scope of this document. The fields will be set during the send process by the send component, specified by the AMSend interface (see figure 5.3). We can notice the addr parameter (a 2-byte integer) of the send command, but there is no type parameter. It is because AMSend implementers provide a parameterized interface (see section 3.3.1), with the type as a parameter. This lets co-existing protocols share a common sending queue which is then managed by the sender component.

Since all layers share the same packet buffer and all rely on a specific interface to manipulate messages, a single interface to receive packets is needed. It is named Receive and shown in figure 5.4. The commands associated with the payload are only here for convenience: this lets the user access to the payload of a message without being wired to a Packet interface.

One can notice that the receive handler is required to return a message_t
5.1. INTEGRATION IN TINYOS

interface AMPacket {
    /* Returns the address of the local node */
    command am_addr_t address();

    /* Returns the destination address of the AM message */
    command am_addr_t destination(message_t* amsg);

    /* Returns the source address of the AM message */
    command am_addr_t source(message_t* amsg);

    /* Returns the AM type (the handler ID) of the message */
    command am_id_t type(message_t* amsg);

    /* Sets the destination address */
    command void setDestination(message_t* amsg, am_addr_t addr);

    /* Sets the source address */
    command void setSource(message_t* amsg, am_addr_t addr);

    /* Sets the AM type */
    command void setType(message_t* amsg, am_id_t t);

    /* True if the message is destined to this node's address */
    /* or if it is a broadcast message */
    command bool isForMe(message_t* amsg);
}

Figure 5.2: The AMPacket interface.

buffer when a packet is received. This is done to avoid memory leaks. Indeed, the receive component has to know when the packet has been processed and the buffer available, so that it can reuse the buffer for another received packet. If the receive handler had to process several packets, it could keep all the buffers (which are statically allocated, see section 3.3.3). Therefore, the receive component could run out of memory, and this would prevent other components from receiving messages. By returning a buffer, the receive handler provides to the receive component a receive buffer that can be used for the next packet received. The receive handler can either process the packet and return the given buffer, or post a processing task and return another buffer.
interface AMSend {
    /* Sends a message to addr, with len bytes of data */
    command error_t send(am_addr_t addr, message_t* msg, uint8_t len);

    /* Cancels the sending of a message */
    command error_t cancel(message_t* msg);

    /* The sending of msg is finished */
    event void sendDone(message_t* msg, error_t error);

    /* Returns the maximum amount of data, in bytes */
    command uint8_t maxPayloadLength();

    /* Returns a pointer to the data payload */
    command void* getPayload(message_t* msg);
}

interface Receive {
    /* A message has been received with len bytes of data at payload */
    event message_t* receive(message_t* msg, void* payload, uint8_t len);

    /* Returns a pointer to the payload, fills len with its length */
    command void* getPayload(message_t* msg, uint8_t* len);

    /* Returns the amount of data in bytes */
    command uint8_t payloadLength(message_t* msg);
}

Figure 5.3: The AMSend interface.

Figure 5.4: The Receive interface.
In order to save energy, the radio is not started during the boot process on the mote. This allows an application to start and stop the radio when needed if it is not used often. Consequently, the radio need to be started before any communication can take place, this is done via a SplitControl interface (see figure 5.5). Contrarily to the StdControl interface which provides two simple commands to start and stop a component, the SplitControl interface provides these two tasks as split-phase operations: two commands to invoke the starting and stopping, and two events to signal their completion.

```c
interface SplitControl {

    /* Starts this component and all of its subcomponents. */
    command error_t start();

    /* Notifies caller that the component has been started. */
    event void startDone(error_t error);

    /* Stops this component and all of its subcomponents. */
    command error_t stop();

    /* Notifies caller that the component has been stopped. */
    event void stopDone(error_t error);

}
```

Figure 5.5: The SplitControl interface.

### 5.1.2 Network Layer

Since TinyOS 2.0 is quite recent (released on November 6th, 2006), the default distribution does not provide many generic components to implement routing protocols. So far, it only provides the implementation of Dissemination, a protocol to send a packet to every node, and Collection, which is an address-free protocol to transport small packets in an tree topology.

However, interesting work had been done for TinyOS 1.x. A modular network layer[33] was designed to ease implementation of new protocols and code reuse. Some parts of this layer were implemented and provided with the TinyOS distribution. This earlier work gives a good basis for our design, therefore we adapted it to TinyOS 2 and our needs. This is discussed in next section.
5.2 Architecture of the Implementation

This section describes the components involved in the implementation of DYMO, and how they are linked together. We will first review the general architecture of the components, and give more details about each of them afterwards. More details about the interfaces that link these components together are given in the next section.

5.2.1 Overview

Before describing the design of our network layer, we present the work on which we have based ours. The paper[33] describes a modular network layer for sensor-nets, and its goal is to “ease the implementation of new protocols, by increasing code reuse, and enable co-existing protocols to share and reduce code and resources consumed at run-time”.

Original layout[33]

To achieve the goal stated above, a representative set of various protocols for sensor networks was examined in order to identify their common parts. This made it possible to divide the protocols into several functions, some of which can be shared by all or some of the protocols. This was then used to design a general layout of components that provides a framework for implementing routing protocols.

The layout is divided into two parts: the data plane and the control plane. Implementing the control plane is not surprisingly much more complicated, since it implements the routing algorithms. The functioning of this layout is illustrated in figure 5.6.

The Dispatcher examines the header of the packets coming from the lower or upper layer in order to determine the protocol to which the packet belongs, and passes the packet to the appropriate protocol service. The latter is a set composed of a Forwarding Engine, a Routing Engine and a Topology Engine.

Though the Forwarding Engine is part of a protocol service, it is not aware of the protocol format and algorithms. It simply requests the Routing Engine to fill the routing header of a packet before forwarding it, or deliver the packet to the upper layer when the packet has reached its destination. The reason why the Forwarding Engine belongs to the protocol service is that it may perform packet aggregation or scheduling, and these tasks depend on the protocol.

The Routing Engine and the Topology Engine are the core components of a protocol: while the Routing Engine generates and processes control packets,
5.2. ARCHITECTURE OF THE IMPLEMENTATION

The Topology Engine computes and stores the necessary information about the network topology, according to the data reported by the Routing Engine.

Finally, the Output Queue handles the packets to be sent from all the protocols running on the node. Since all packets must go through this component to be sent, the Output Queue can schedule them according to the node policy.

This earlier work provided us not only a good starting point to implement the DYMO protocol, but also general guidelines to ensure that our work is generic enough to be reused by the research community. Indeed, some parts of our implementation are not related to DYMO, and were implemented only because TinyOS does not provide them yet. Our aim is that these parts will be useful for other protocol implementers.

**Implemented layout**

The goal of the implementation is to provide a component to an application in order to transparently send and receive data in a multi-hop network. We have called this component `DymoNetworkC`, which is a configuration. The wiring provided by this configuration is illustrated in figure 5.7. Since DYMO is a routing protocol,
Figure 5.7: General layout of the components. To avoid complicating the diagram further, the Packet interfaces are not represented. They would have appeared each time another type of packet is used or provided.
the configuration must include a transport protocol to transport data on multi-hop routes. It can be any transport protocol using the same address format as DYMO, a 16-bit address in this case. In this document, we refer to the transport protocol as MH (for Multi-Hop). The configuration also provides the application with the possibility to inspect all the multi-hop data packets that travel through this node, and to decide if they should be forwarded. This is done via the Intercept interface.

To be used, the network layer need to be started with the SplitControl interface. This is implemented by a dedicated module, NetControlM, which waits for all other components to start before letting the application use the network layer (ActiveMessageC implements the link layer). The application can then send and receive MH packets, which can be manipulated with the MHServiceC component.

The DymoServiceC and MHServiceC components are the protocol services: they are responsible for all the processing and packet manipulations related to their respective protocol. Both of them have their own sending and receiving queue, an instance of AMSenderC and AMReceiverC. This does not break the single Output Queue principle we have seen above: though it is not represented on the diagram for simplicity reasons, these queues rely on ActiveMessageC to exchange packets with the radio chip. It is this component that actually plays the role of the Output Queue, and it uses the parameterized wiring feature of NesC (see section 3.3.1) to deal and gather the packets to the AMReceiverC and from the AMSenderC components. This is also why there is no need for a Dispatcher component. The ActiveMessageC component provides the link-layer feedback as well: it is possible to request hardware acknowledgements for each packet sent and thus determine if the neighbor received the packet.

To make things clearer, the sequence diagram in figure 5.8 shows the interactions between components when the application wants to send a packet to an “unknown” node. The application is unaware of the routing operations, it thus sends the packet as if it was a single-hop packet. The packet is given to the MHServiceC component which does not know how the routing protocol operates, but is aware that obtaining a route may not be immediate. Therefore, when the routing table (which is shared by both of the protocol services) signals that the route is not available yet, the send command returns and the MH service will retry regularly to send the packet. In the meanwhile, the routing table signals to the DYMO service that a route is needed, and a route request is issued. When the route reply arrives, the routing table is updated, so that the next try from the MH service will be successful. The data packet is eventually sent to the next hop on the route, and the sendDone event is signaled to the application, so that it can reuse the packet buffer.
Figure 5.8: Sequence diagram for the sending of a packet triggering a RREQ. DymoTableC is shared by DymoServeC and MHServiceC.
5.2.2 The DYMO Service

The DymoServiceC (figure 5.9) does not exactly follow the modular layout presented in section 5.2.1. Indeed, it has a Routing Engine (the DymoEngineM and DymoPacketM components) and a Topology Engine (the DymoTableC component), but no Forwarding Engine. The main reason is that it would have added useless complexity. Since upper layers are not interested in DYMO packets, the delivering functionality of the Forwarding Engine is not needed. Furthermore, the DymoEngine is the only component that sends DYMO packets, therefore the Forwarding Engine would not need to request the DymoEngine to select a route, since it would have already been selected by the DymoEngine.

As a consequence, the DymoEngine is directly connected to the AMSend and Receive interfaces, and it handles the received packets. Since processing a packet can take a long time, it is implemented as a split-phase operation (see section 3.2.2), illustrated in figure 5.10. When a DYMO packet is received, it is given to the DymoPacketM module, which returns immediately and posts a task to read the packet. Each piece of information found in the packet is given to DymoEngineM via an appropriate event. The event handler uses the routing table to judge the usefulness of the information, and decides accordingly if the information should be propagated. It returns its decision to the DymoPacketM module, which in parallel constructs the packet to be forwarded.

5.2.3 The MH Service

The routes determined by the DYMO protocol need a multi-hop transport protocol to be used. Though we did not need such a protocol to implement DYMO, we need one to test and evaluate the implementation. Since no such protocol was available in the TinyOS 2.0 distribution, we implemented a very simple one. Implementing such a protocol also allowed to provide a directly usable multi-hop network layer to applications.

The protocol actually implements the Active Message interfaces (see section 5.1.1) on top of the existing Active Message stack.

Contrarily to the the DYMO service, the MH service (figure 5.11) does have a Forwarding Engine, which is actually more complicated than the control plane. When a MH packet is received from the AM layer or sent by the application, the Forwarding Engine requests MHEngineM to fill the AM fields (and the MH fields if necessary) in order to put the packet on the route toward its target. Given that the route may be unknown and that we are working with a reactive routing protocol, the Forwarding Engine does not discard the packet if no route is available. Instead,
CHAPTER 5. IMPLEMENTATION

Figure 5.9: Layout of the DYMO service component.

Figure 5.10: Sequence diagram of the processing of a DYMO packet.
5.2. ARCHITECTURE OF THE IMPLEMENTATION

Figure 5.11: Layout of the MH service component.

it puts it in a waiting queue and regularly retries to request the route. If the RREQ issued by the DYMO service is successful before a certain timeout, the packet is finally given to the sending queue. Since it does not have any functionality specific to the MH protocol, the Forwarding Engine was made as generic as possible and does not rely on any MH-specific interface. It may therefore be used by other protocol services.

The MHEngineM module is almost trivial. Unless the packet has reached its target, it requests the routing table for a route. If one is available, the packet header is updated and the Forwarding Engine can send it, otherwise the Routing Engine tells the Forwarding Engine to wait.

5.2.4 The Routing Table

The routing table is implemented by the DymoTableC component. Though it appears in the wiring of DymoNetworkC, DymoServiceC and MHServiceC, it is of course the same instance. The DymoTableC stores known routes, that is mainly a target address, a next hop, a sequence number and a hop count. Each routing entry is attached to several timers as suggested by the DYMO specifications to monitor the routes (see section 4.3.2).

Routing information is retrieved from the table via the RoutingTable interface, a generic interface for routing tables (described in section 5.3.1). The
DymoEngineM module has more control thanks to the DymoTable interface, to update the table and know when a route is needed, so that a route request can be issued.

5.3 Routing Interfaces

To compose the network layer provided by our implementation and let components communicate with each other, a number of new interfaces were needed in addition to those provided by the TinyOS distribution. This section presents these interfaces.

5.3.1 Routing Table Interfaces

Due to the fact that the routing table is shared by two protocols with different purposes, two different interfaces to manipulate the routing table were needed.

RoutingTable

The first interface (figure 5.13) is a generic interface that could be used for other routing tables. It provides access to the information stored in the routing table through the getRoute or getForwardingRoute commands. The first one is called to send a packet while the second one is called to forward it. Two different commands are needed because some protocols take different decisions depending on whether the packet is sent or forwarded. DYMO is one of them: when a route is unknown, a RREQ is generated if the packet is to be sent, but a RERR is generated if the packet is to be forwarded.

Since being too generic would also mean too much complexity, the interface only applies to unicast routes. As a result, these commands only take an address as a parameter, in addition to the memory address of where to store the result of the command. They return a code to specify if the route exists, if it will soon (i.e., if a route request is pending), or if it is broken.

A user of the routing table can also be informed when a route is deleted from the table (in case it is relying on this route). This can happen when the route was replaced by a new one because the table was full, when the route become too old, or when a broken link is detected. This information is obtained via the evicted event.

Routing information is represented via the rt_info_t structure (see figure 5.12). It is therefore up to the implementation of the routing table to define this type, as well as the reason_t type. In our implementation, route entries may contain any
5.3. **ROUTING INTERFACES**

piece of routing information that a DYMO packet can transport, plus the next hop on the route.

**DymoTable**

The second interface (see figure 5.14) is specific to DYMO, and provides more information and control to the users. The two goals of this interface are to fill and update the routing table and to be aware of the needed routes.

The first goal is achieved with the `update` command. According to the parameters and the content of the table, the command decides if the information is better than what is available and updates the table accordingly. It is therefore a command called each time a piece of routing information is found in a DYMO packet.

The second goal is achieved with the two other commands: `routeNeeded` and `brokenRouteNeeded`. The first one is called whenever a route needs to be discovered, that is when a node wants to send a packet to an unknown route; while the second one signals that a route was expected but it is broken or absent, thus requiring a RERR. Since the DYMO engine relies on the routing table to find routes, it does not need to determine if a RREQ or a RERR is needed: relevant signals will be triggered by the routing table via the `DymoTable` interface.

5.3.2 **Packets**

For either DYMO or MH, only one module knows how to manipulate a packet at the bit level. Each time another component wants to read or write into a packet, it must rely on the module. They are two such modules in our implementation, `DymoPacketM` and `MHPacketM`, and both of them are used via a dedicated interface.

**DymoPacket**

This interface (figure 5.15) is to be provided by a component that knows all the internals of the DYMO packet format.

In order to create or alter a DYMO packet, a component has two commands at its disposal: `createRM` and `addInfo`. The first one creates a DYMO message with the minimum amount of information, which is the message header, the target and originator nodes for a routing message, or the first unreachable node for an error message (in which case `origin` is not specified). Then, if the creator or a forwarding node wants to append additional information to the message, it can use the `addInfo` command. This command does not specify where (that is, in which block) the piece of information should be added, it is up to the implementer to
typedef struct {
    addr_t address;
    addr_t nexthop;
    seqnum_t seqnum;
    bool has_hopcnt;
    uint8_t hopcnt;
} rt_info_t;

typedef enum {
    REASON_FULL,
    REASON_OLD,
    REASON_UNREACHABLE
} reason_t;

Figure 5.12: Types associated with routing tables.

interface RoutingTable {
    command error_t getRoute(addr_t address, rt_info_t * info);
    command error_t getForwardingRoute(addr_t address, rt_info_t * info);
    event void evicted(const rt_info_t * route_info, reason_t r);
}

Figure 5.13: The RoutingTable interface.

interface DymoTable {
    command void update(rt_info_t * route_info);
    event void routeNeeded(addr_t destination);
    event void brokenRouteNeeded(const rt_info_t * route_info);
}

Figure 5.14: The DymoTable interface.
typedef enum {
    ACTION_KEEP,
    ACTION_DISCARD,
    ACTION_DISCARD_MSG
} proc_action_t;

interface DymoPacket {
    /* Returns DYMO_RREQ, DYMO_RREP or DYMO_RERR */
    command dymo_msg_t getType(message_t * msg);

    /* Returns the size of the message */
    command uint16_t getSize(message_t * msg);

    /* Creates a DYMO message with its heading routing information */
    command void createRM(message_t * msg, dymo_msg_t msg_type,
                           const rt_info_t * origin, const rt_info_t * target);

    /* Adds a piece of routing information to a message */
    command error_t addInfo(message_t * msg, const rt_info_t * info);

    /* Processes msg and fills newmsg with the message to forward */
    command void startProcessing(message_t * msg, message_t * newmsg);

    /* The hop values have been read */
    event proc_action_t hopsProcessed(message_t * msg,
                                       uint8_t hop_limit, uint8_t hop_count);

    /* A new piece of routing information has been extracted */
    event proc_action_t infoProcessed(message_t * msg, rt_info_t * info);

    /* The message processing is finished */
    event void messageProcessed(message_t * msg);
}
choose a good place so that the packet size is minimized. This command can fail if the packet has reached its maximum size.

Reading a packet is completely different. The goal was to be able to easily go through the list of pieces of routing information included in the message, since it was an important purpose of our simplified packet format (see section 4.4.2). As a consequence, accessing a piece of information by its position would not be suitable, because the complexity of processing a message would not be linear. Returning a table with all the information would imply copying a large amount of data, and there is no such thing as *iterators* in nesC, as proposed by high-level languages. We thus decided to let the DymoPacketM module iterate through the packet and report each piece of information to the user component via appropriate events. This is also illustrated on the sequence diagram 5.10.

During the packet processing, the DymoPacketM module also builds the message that may be forwarded. For each event reported during the packet processing, the user (that is the DYMO engine) specifies if this information should be kept in the forwarded message. Also, when it has enough information about the processed message, it can request DymoPacketM to stop building the forwarded message if it is useless.

### 5.3.3 Route Selection

To let the routing engine of the MH service decide what should be done with a packet, the forwarding engine uses the RouteSelect interface (figure 5.16). It features a single command that will select a route towards a target (using the routing table), and fill the header of the message appropriately. The command modifies the message instead of simply returning the next hop address, so that the forwarding engine does not have to be aware of the packet format and routing options. The returned value specifies what the forwarding engine should do with the packet: sending, dropping, or giving it to the upper layer.

### 5.4 Implementation Issues

This section presents the main issues encountered to design and program the implementation of DYMO.

**Architectural design**

The main goals of separating the code into several components are to have a clear vision of how the different parts of the protocol interact with each other, to ease
5.4. IMPLEMENTATION ISSUES

typedef enum {
FW_SEND,       //Put the message in the sending queue
FW_RECEIVE,    //Give the message to the upper layer
FW_WAIT,       //Retry later
FW_DISCARD,    //Discard the message
} fw_action_t;

interface RouteSelect {

/**
 * Ask the routing engine to fill a message with routing
 * information, in order to send it to target.
 */
command fw_action_t selectRoute(message_t * msg, addr_t target);

}

Figure 5.16: The RouteSelect interface.

the implementation of each part, and to have the ability to change one part without altering the rest of the code. The modular network layer for sensornets presented in section 5.2.1 was a useful help for that purpose. However, in practise, the specifications of the DYMO protocol did not always match with this modular network layer.

For example, comparing two pieces of routing information is an algorithm used in several occasions, it was not easy to decide which component should implement it, which and how components should have access to it. We saw also (in section 5.2.2) that the DYMO service does not include a forwarding engine. Though it was clear that the functioning of DYMO is slightly different from the proposed model, it was difficult to decide how the model should be altered to allow the easiest and most efficient implementation.

As a result and as usual, we can say it was a good idea to spend a lot of time on the architectural design.

The packet format

Exchanging and parsing DYMO messages is probably the task that occurs the most often during DYMO operations. As a consequence, we tried to ensure that building and reading a packet was as efficient as possible. This involved design-
ing a new packet format more compact and easier to implement. But this would have been useless without a properly written packet interface since it would not have been possible to read the packet linearly, only once, without compromising the efficiency or the memory consumption. To achieve this, we began to implement different schemes to compare how much code and memory they would need. This led to the choices described in section 5.3.2 and the associated DymoPacket interface.

**DYMO operations**

Though the DYMO protocol is quite simple, it still is a generic routing protocol. As such, it can operate in numerous different situations, especially when we consider the mobility of nodes. Therefore, while implementing such a protocol, one has to keep in mind the broad variety of situations the algorithms can go through, and ensure the code will behave properly for each of these situations. Furthermore, one goal of this implementation was to evaluate the protocol. Thus it was also important to look critically at the algorithms before implementing them, and possibly discuss about them with the authors. As a result, it sometimes required a lot of thinking to implement just a tiny part of the protocol or add a single line of code.

For example, the algorithm used to judge the usefulness and freshness of routing information has been largely discussed. Though this part might seem to be a detail in the specification, it is one of the core functions: it is used each time a routing message is to be forwarded and each time the routing table is updated. Small modifications of the algorithm influence the bandwidth overhead and the performance of route requests, the two main characteristics of a reactive routing protocol. It was thus worth spending some time on improving the details of the algorithm. Moreover, discussing the issue forced to examine all the details of the protocol and imagine various examples of operations, which was useful for the rest of the implementation.

### 5.5 Conclusion

The TinyOS communication model relies on *Active Messages*, a protocol with a simple message format to exchange one-hop datagrams through the radio. On top of this link-layer, we have implemented a network layer that provides an unreliable multi-hop protocol where packets use the same packet format as the link layer. Multi-hop routes of this network are built with the DYMO protocol and a more compact protocol format. The implementation design is based on A Modular Network Layer for SensorNets[33], first proposed to ease the implementation of
routing protocols on TinyOS.

As the memory footprint is an important characteristic of the implementation, we calculated its value for the final binary. The values are given for the telosb platform and do not take into account the two optional features of our network layer that can be excluded at compile-time. These features are the possibility to use a local network loop for two applications to communicate with MH messages, and the ability of the MH service to use packet acknowledgments to detect broken links.

The whole network layer has a total code size of 19,916 bytes and needs 940 bytes of RAM. That is roughly 40% of the available ROM size and 10% of the available RAM on Telosb motes. The RAM footprint does not take into account the dynamically allocated stack memory used by the function calls. The memory usage is divided according to the following table where the values are specified in bytes:

<table>
<thead>
<tr>
<th>Component</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYMO service modules</td>
<td>3,740</td>
<td>216</td>
</tr>
<tr>
<td>MH service modules (without the table)</td>
<td>746</td>
<td>72</td>
</tr>
<tr>
<td>NetControlM</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>Other TinyOS modules</td>
<td>15,386</td>
<td>651</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19,916</strong></td>
<td><strong>940</strong></td>
</tr>
</tbody>
</table>

The values are given for a routing table with five entries, each entry requiring nine bytes. Though the code size could be a problem for large applications, both values are acceptable as most of the available RAM and ROM is left to the application. Since the network layer uses many components from TinyOS, most likely some of them will also be used by the application and the code size will not grow too much.
Chapter 6

Tests and Experimentation

Soon after the implementation began, the code was tested in a simulated environment to ensure that it performs correctly and is compliant with the DYMO specifications. When the implementation became almost complete, it was also possible to examine how well DYMO operates in simple test cases and to test the implementation using real experiments. In addition to providing a network layer for TinyOS, the idea behind implementing DYMO was to evaluate the protocol. Rather than writing various simulation scripts, we preferred to conduct concrete experiments. This chapter describes the tests and experiments that were conducted.

6.1 Functional Tests

Each time a component of the implementation was created, its functionalities were checked with the TOSSIM simulator, described in the following. The whole network layer has also been tested with TOSSIM when possible, and with small real networks afterwards.

6.1.1 TOSSIM

TOSSIM (TinyOS SIMulator) is a simulator provided with the TinyOS distribution. Its main advantage is that it runs code written in nesC. It is thus possible to benefit from the advantages of a simulator without having to rewrite the application in a specific language. TOSSIM is implemented as a library: it provides an interface to the execution of the TinyOS application, and the user writes a program in C, C++ or Python to control the execution of the simulation. The advantages of using TOSSIM to test and debug a TinyOS application are numerous:
• Print statements: it is possible to include in the nesC code preprocessor macros that print messages on a chosen output in a manner similar to the printf function of C. The code of these calls is kept only when the application is compiled for TOSSIM.

• Variable inspection: TOSSIM provides functions to retrieve the value of any variable of the application.

• Control of the execution: the basic function of TOSSIM is runNextEvent, which runs a single task of the TinyOS scheduler. When used with Python, it is thus easy to pause and resume the simulation to examine what is happening.

• Fast execution: the code is run with the processing speed of the computer, that is hundred times faster than on a real mote. Since TinyOS timers rely on the processor speed of the mote to compute time, they are also faster during the simulation. The only problem that this fast execution could result in is the difference between the real propagation time and the simulated one. We did not notice this issue though.

• Time awareness: TOSSIM knows the processing speed of the simulated platform, so that it can deduce the simulated time and display it. It is also possible to control at which rate instructions are run to slow down the simulation.

• Network simulation: several motes can be simulated simultaneously (provided they run the same application). TOSSIM provides a model for wireless communications and functions to set links between nodes. It is also possible to inject packets in the network at a given time.

TinyOS however currently provides simulation components only for the micaZ platform (see section 3.1.2). Thus one has to write the simulation code to reproduce a more realistic behaviour for other hardware, which implies knowing how to emulate all the chips of the needed platform. Furthermore, it is not possible to specify on a per-mote basis the application to run. TOSSIM only runs one application and duplicates it to simulate a network. As a consequence, to simulate a sensor network with different applications, one has to build a large application that decides at run-time which code to run depending on the mote ID. Finally, TOSSIM for TinyOS 2.0 is still recent and does not provide simulation code for all the components of TinyOS. As a consequence of all these facts, there are some tests that cannot be performed with TOSSIM.

Another reason to perform real tests instead of simulations is that the simulator does not have the hardware constraints of the mote. For example, if a call
graph becomes too big for the memory stack of the mote, or if a bad packet overflows a buffer, the simulation could succeed while the real execution would fail or behave differently. This difference can lead to bugs and problems that would not be detected by the simulation.

6.1.2 Simulated Tests

The method to test a component was the same for all components. A testing component that provides and uses all the interfaces that the tested component uses and provides was written and wired to the tested component (see figure 6.1). The implementation of these interfaces was either empty, or a simple well-defined behaviour that serves the purpose of the test. The features of TOSSIM were then used to provide as much information as possible about the execution and to check that the component behaves as expected. As the tested components became more and more advanced and the test cases more and more complicated, the testing component sometimes relied on TinyOS components or well tested components to provide their real behaviour. Another helpful solution for complicated test cases involving communications was the packet injection functionality. We give below three examples of such tests.

![Diagram](image)

Figure 6.1: The generic wiring of a test program.

DymoPacketM

Two different parts of the module had to be tested: creating a packet and processing a packet. To test all the possibilities, we made a test for the two types of packet (routing message and route error), and for each type, we tried with all the possible kinds of information format: with/without the head prefix, the sequence number field, the hop count and the maximum age (see section 4.4.2).
First, a packet was created and different pieces of routing information were appended to the packet. Between each command call, the content of the packet was displayed to check that it contained the appropriate values at the appropriate place. Then the packet was processed, the reported information was printed on screen and compared to the original information. This module did not raise any problem, as writing it was mostly a matter of dereferencing or copying at the appropriate address.

DymoTableM

Making a comprehensive list of the test cases was more complicated for the routing table, because route entries can be altered by command calls and by timers. Route entries can be in different states which affect how they are updated, the result of a look up, and the events that are signaled (route needs or evictions). Furthermore, adding a route entry has different consequences depending on the number and the state of the other entries.

Writing all the test cases would have taken a lot of time. Therefore we only ensured to write unit tests for all the functions and commands, and functional tests for test cases that cover big parts of the module code. To verify the absence of data race, the testing component implemented fake timers so that we had a fine control on when the different timeouts were triggered. However this is not a foolproof method since the execution order of tasks and events is eventually determined by the TinyOS scheduler, which we did not try to control.

Network Services

We used two different techniques to test the DYMO and MH services. The first one involved only one node and was similar to the techniques described above: a testing component wrapped the whole service in order to control all the incoming and outgoing calls. For example, the testing component creates a MH packet, passes it to the MH service via its AMSend interface or its underlying Receive interface. The packet is then processed by the MH service, during which various information about the process is displayed. The packet is eventually given to the upper or lower layer via the appropriate command or event, which are both implemented by the testing component so that it can check and display the content of the packet.

The previous technique allowed to ensure that packets were processed correctly. To check that the services are still working in a networking context involving concurrency, multiple nodes and multi-hop routes, we used a different technique. The testing component was simpler, it was a small application that regularly sends messages to pre-determined nodes. The topology was known in
6.2. **RREQ DELAY EVALUATION**

advance and the involved addresses were hard coded in the testing component, it was thus easy to check that messages were forwarded and received by the expected nodes. The network was then deployed in TOSSIM and each node displayed the messages it sent, forwarded and received.

### 6.1.3 Hardware Tests

To compare with the simulations, or when there was no other choice, tests on real motes were also performed. The possibility of setting the power of the radio allowed to set up small networks within a room or even on a desk.

The big difference with a simulation is the available information. The standard display capacities of Telosb motes are only three LEDs. It is thus very difficult to know what is happening in the code without trying to make a LED blink at different places in the code. Since each try involve recompiling the application and reprogramming the mote, such tests can be quite painful. We only performed them when really needed, or as a quick check to ensure that the code performs correctly with concrete test cases. We did not test individual components as with TOSSIM, only the global DymoNetworkC configuration. For that purpose, we just reused the tests described above, slightly modified: instead of printing information about the exchanged messages, the LEDs are toggled accordingly.

### 6.2 RREQ Delay Evaluation

The RREQ delay is the time elapsed between the signal from the routing table that a route is needed and the insertion of the route in the table. This is not the only possible definition, for example it can be defined as the time elapsed between sending a RREQ and receiving the corresponding RREP, or as the time elapsed between requesting to send a data packet and actually sending the packet. The definition we chose was the most interesting for us because of the way we implemented the MH protocol. We saw in section 5.2.3 that the forwarding engine regularly tries to get a route when none is available at first. The delay between each try has to be carefully set so that the engine does not retry before the RREP arrives, and not too long after, in order to send the data packet as quickly as possible. This delay corresponds to the definition of the RREQ delay that we chose.

We describe in the following the experiments to evaluate this delay and the results we obtained.
6.2.1 Experimental Process

In this experimentation, we set up a network of eight motes with a line topology. Each mote is in the direct range of the previous and the following mote. The first mote is programmed with an application that sends periodically an empty data packet to another mote, with a period slightly above the maximum age of a route, so that a route request will be issued for each sending. It begins with sending 100 messages to its neighbor, then 100 messages to the third mote, and so forth until the eighth mote.

Each time a route is discovered, the mote sends the corresponding delay through the USB port and the delay is logged on the computer. The other nodes only perform DYMO routing operations: forwarding routing messages and replying to route request. We ensured that only the target of the route request replies by disabling intermediate route replies. Therefore with this experiment, we should obtain an average RREQ delay and be able to relate it to the length of the discovered route.

6.2.2 Results

Figure 6.2.2 shows a table and an histogram of the results given in milliseconds. Overall, as one could expect, the average RREQ delay increases with the length of the route. However, these figures reveal a quite high standard deviation and do not show an affine combination between the discovery delay and the number of hops. This may be explained by two different causes.

First, given the standard deviation we obtained, 100 measurements for each length may be too low to provide reliable results. The best solution would have been to make the same experimentation with more measurements, but unfortunately we did not have the time to do so. The cause may also be a technical issue. For convenience, the experiment was established within an office, with the radio power of the motes set at its minimum. At this range, the variation of range can be important because of the variation of the battery voltage. As a consequence, it is possible that a mote cannot reach its neighbor during a try or on the contrary, that a mote skip its neighbor.

The results give anyway a good idea of the discovery delay, useful to configure the forwarding engine. One should notice that these measurements took place without intermediate route replies and path accumulation. By enabling these two features, route request can be replied faster and the routing messages can carry more information that can be used by motes to reply to route requests. We would expect lower delays with these features enabled.


6.3. **THE INFLUENCE OF DYMO ON POWER CONSUMPTION**

<table>
<thead>
<tr>
<th>Number of hops</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average delay</td>
<td>33,94</td>
<td>40,25</td>
<td>41,4</td>
<td>59,87</td>
<td>67,81</td>
<td>70,57</td>
<td>67,86</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>5,48</td>
<td>11,04</td>
<td>8,44</td>
<td>8,06</td>
<td>12,1</td>
<td>9,58</td>
<td>9,29</td>
</tr>
<tr>
<td>Minimum</td>
<td>23</td>
<td>21</td>
<td>23</td>
<td>34</td>
<td>47</td>
<td>54</td>
<td>52</td>
</tr>
<tr>
<td>Maximum</td>
<td>50</td>
<td>74</td>
<td>62</td>
<td>85</td>
<td>103</td>
<td>93</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 6.2: Results of the RREQ delay evaluation. Delays are in milliseconds.

### 6.3 The Influence of DYMO on Power Consumption

In order to have a more realistic and detailed insight of how a DYMO-routed sensor network behaves, we designed and implemented a sensing application coupled to a monitoring Java application and established small networks to conduct experiments. This section describes these experiments and their results.

#### 6.3.1 Goal

The goal of the experimentation is to provide experimental data to another project, which aims at setting up a sensor network in buildings to measure humidity. The main unknown is the life expectancy of such a network, and how to maximize it. We thus wanted to determine with our experimentation an estimation of how long this kind of network can live, and how the parameters of the application and the DYMO protocol affect this value.

Though the Telosb motes at our disposal are capable of sensing humidity, we used their light sensor because it is much easier to control the ambient amount of light and check the data.
6.3.2 Implementation

For this experimentation to take place, four different parts were needed: one central base station; a central application hosted on a computer; a sensor application embedded on the motes; and another embedded application to monitor DYMO operations. These four elements are described in the following, the resulting network is illustrated on figure 6.3.

![Figure 6.3: The sensor network used in our experimentation.](image)

**The Base Station**

The base station is a mote connected to a computer. Its sole role is to provide a gateway service between the central application run by the computer and the sensor network. For each message received over the radio by the base station, it extracts the payload and sends it through the USB port. It also does exactly the same the reverse way: each message coming from the USB port is injected in the DYMO-routed network. As it is a bottleneck of the network traffic, the base station uses a queueing system for each forwarding direction.

This kind of topology where all the motes send their data to a central base station (also called a *sink*) is common for sensor networks. This is why the TinyOS
distribution comes with an implementation of the base station, and we did not have to develop it. However, we had to slightly modify it to work with our multi-hop network layer. The provided base station application only sends the payload to the central application, with a `AMSend.send` command (see section 5.1.1) parametrized with the AM type. As a result, the central application does not know the originator of the message. This has been solved by sending the whole MH-Packet to the central application.

The Sensor Application and Protocol

Depending on the use of a sensing application, the period at which the data is sensed and sent as well as the amount of data sent may vary. These parameters also affect the power decrease since the radio transmitter is a main energy consumer. Therefore, we designed the application running on the motes so that it can change the parameters upon reception of a command from the central application. This allows different experiments to be conducted without reprogramming the motes and reconfiguring the network. We also wanted the sensors to be able to send their current parameters to the central application in order to check that they are correct.

The protocol enabling these features is very simple. Its format is a simple Type-Length-Value block, where the type is a set of flags (it is not the value of the type that is important, but the values of each bit of the type). Since the sensors always send their messages to the central application (and of course the central application always sends its messages to the sensors), it is possible to assign different semantics depending on the receiver of a message. In other words, the central application do not understand the messages the same way the motes do, even if the content is the same.

When a message is sent from the central application, the first flag specifies if this is a “set” or a “get” command. The subsequent flags specify to which parameter the command applies, they can be:

- The latest value(s) read by the sensor (for a get command);
- The period at which the sensor reads;
- The threshold: values read by the sensor are sent only if they are above this threshold;
- The number of values to buffer before sending them.

When the message is sent from a mote, the first flag specifies if the message is sent in return of a command or if it is periodic. The second flag specifies if
the content of the message is sensed data or the value of some of the sensor parameters, in which case the following flags specify which parameters. Figure 6.4 shows two examples of messages that could be sent or received by a mote. As we did not implemented a flooding mechanism in the MH protocol or via a separate protocol, it is not possible to send a command to all the motes from the central application. But since it was for small experiments with less than ten nodes, it was not a big problem. When needed, we sent the command node by node using unicast.

![Figure 6.4: (1) A command sent by the central application that requests the mote to set its sensing period to 10 seconds and to buffer 10 readings before sending a message. (2) The beginning of a data message of 10 readings (each is 2-byte long) sent by a mote.](image)

The embedded application simply senses on the motes, buffers, and sends data periodically according to its parameters, and update its parameters according to the received commands.

**The Monitoring Application and Protocol**

In addition to the sensor application, we needed an application to store the different values we are interested in for our experimentation. The main one is the battery level, and we also wanted to relate the value of some variables (e.g. the delay of RREQs or the number of forwarded packets) with the parameters of DYMO.

This application is similar to the sensor application: it stores a number of variables, and uses a TLV-formatted message to send their value to the central application periodically or in return to a command. Since some of these values are only known by the modules of the DYMO service, we made a simple interface so that applications can have read access to these variables. However, it introduces more code, uses more memory and is useful only for specific purposes, we thus let the developer disable this feature with a preprocessor parameter.

**The Central Application**

The central application is in charge of processing the messages received from the motes via the USB port and logging the necessary information in order to exam-
6.3. THE INFLUENCE OF DYMO ON POWER CONSUMPTION

ine it afterwards. Moreover, to provide real-time information about the sensor network, the application has a graphical user interface that can displays the network topology and the values of the different variables of each node, and can send commands to the motes. The network topology can be computed by requesting motes to send the content of their routing table.

The TinyOS distribution comes with a library that among other things provides helpful classes to process the packets coming from the base station: a generic class to represent a message, and a class to listen to the message coming from the USB port (among other sources). A command-line tool called mig, also provided by TinyOS, allows to generate a Java (or python) class from a nesC structure describing a packet format. Given an array of raw data, this class can construct an instance that stores the content of the packet and gives access to each of its fields. There was thus no need to write the parsing of the different packet formats. Therefore, the application was written in Java: it allowed a fast development thanks to the Swing API and to the TinyOS library.

A screenshot of this application is provided on figure 6.5. It represents a network of seven nodes, their location is specified via a configuration file. The color of nodes shows their battery level. A bright green indicates full or unknown batteries, the color turns into red as the power decreases. A grey node is an inactive node, it has not sent any message yet. Nodes can send the content of their routing table, which allows to represent the topology. A black link is a neighbor link whereas a blue link is a multi-hop route. Two links are drawn if the route is bidirectional. The right part of the screenshot shows the available information about the selected node, and provides buttons to request additional information or send commands.

6.3.3 Experimental Process

The experiments had to be long enough so that a significant decrease of the battery level could be observed. But in the beginning, we did not know the amount time this requirement represents. So we first wrote a simple application that sends a message every second and we set up a network of three motes that communicate with this application. The goal was not to exhaust the battery, but to get a rough idea of how fast the power decreases. The network was deployed within a small area and the radio power was set at its minimum. We let the application run for 24 hours, measured the difference of battery level, and it appeared that about four percent of the battery was consumed. As we knew that the radio would consume much more at full power, but messages would not be sent so often during the experimentation, we decided to keep the duration of the experiments between two and three days.
Then, we designed various sets of parameter values and planned experiments for each of them. There are four kinds of such sets:

- **Logging**: The goal is to keep a log of all the readings in a central location. There is no time constraint, the only requirement is that all the data will be stored eventually. The period is thus set as needed for the time precision, the threshold is null, and motes buffer as many readings as possible to take the most advantage of the packet size.

- **Security logging**: The goal is to keep track of possible problems but only to examine them afterwards, there is no time constraints. The parameter values are similar to the previous ones, except that a threshold value is set. The messages should be less frequent.

- **Monitoring**: The goal is to send the data in real-time to an agent or a computer to process it. It requires the same parameters as the logging application, but with lower values so that the central station is often updated.

- **Alarm**: The goal is to detect a problem as fast as possible in order to react
quickly. A threshold is set, the sensing period needs to be low, and motes send a reading as soon as it is obtained and is higher than the threshold.

Of course, the first one is probably the only one that is useful for measuring the humidity in a building. We wanted to make the experiments for different kinds of applications to compare with the application we are interested in and see if using DYMO would be more relevant for other kinds of application.

The main DYMO parameter we were interested in is the route timeout. Indeed, the longer the timeout is, the fewer route requests there are, and the less power is consumed. We thus planned to experiment each of the four kinds of application with a short timeout and with a longer one. A timeout is short if it is unlikely that a route will be used several times before being deleted, while with a long timeout motes send several messages with a single route request.

Another interesting DYMO parameter is the path accumulation feature. The DYMO protocol allows nodes to append routing information to routing messages before forwarding them (see section 4.2.2). Enabling the path accumulation can dramatically increase the size of packets, especially on the longest routes, and would as a consequence increase the power consumption.

6.3.4 Results

As they involve more motes, messages and situations than the functional tests we made, the experiments made several unexpected bugs appear. Quite often, several hours could pass before the bug happened (so it could happen during the night), and it was not always possible to reproduce it on demand. Consequently, fixing such bugs was difficult and required a lot of time, it also obliged to restart the same experiment several times. Furthermore, we faced technical issues with the communications between the base station and the central application. Several times, an error in the JavaComm library cut the USB communications, resulting in the loss of all the data subsequently sent to the computer to be logged.

As a consequence, we have been able to perform only a small subset of the planned experiments, and they were not as long as wanted. Figures 6.6, 6.7 and 6.8 show the decrease of the battery level for three different experiments. For all the experiments, the sensor reads a value every two seconds. The first one is the logging kind of application: the mote stores all the reading, as many as allowed by the maximum packet size (that is ten readings). The second one is the monitoring kind of application: the mote immediately sends the readings. The third one is the alarm kind of application: the mote immediately sends the readings that are above a threshold. During the first twelve hours of the experiments the threshold
CHAPTER 6. TESTS AND EXPERIMENTATION

Figure 6.6: A 35 bytes message every 20 seconds.

Figure 6.7: A 17 bytes message every 20 seconds.

Figure 6.8: A 17 bytes message every 3 minutes about.
6.3. *THE INFLUENCE OF DYMO ON POWER CONSUMPTION*

is rarely reached, but becomes regularly reached for the motes 3 and 6 afterward. The overall message rate is about 20 messages per hour.

In the following, we will not take into account Node6 in the first experiment and Node2 in the second and third experiment. During the first two experiments, the respective nodes ran out of batteries, and as the graphs show the voltage seems to decrease faster when the power becomes too low (these motes need 1.7V to work). As for the third experiment, the behavior of Node6 is unexplained. The corresponding node did not send more or bigger messages, and the voltage was high enough. The best explanation we found is that the batteries were deteriorated.

**Life Expectancy**

One may find that the power decreases very fast according to the figures. The last graph, which has the slowest decrease, shows that the battery would exhaust in about 10 days: the motes lost about 75mV in 12 hours, and 1300mV are available because the motes need at least 1.7V. But for these experiments, we did so that the power decreases quickly for the experiments to be shorter. First, the motes sense and send data at quite a high rate. Monitoring humidity would not require to send the values so often. Next, the route timeout is short, route requests are thus send quite often. In a real context, routes can be kept for a long time because the motes are stationary. Since route requests are flooded, using a short route timeout results in a lot of dymo messages, and therefore a fast power decrease. Finally, as the motes would not have sent messages so often in a real context, they could have entered in a sleep mode when they are idle, or at least shut down the radio. As a result, the life expectancy of the network would have been much longer with more realistic settings.

**Relation between Radio Emission and Power Consumption**

In the first experiment sent messages are smaller than in the second, and in the third experiment fewer messages are sent than in the second. So the first fact that one would expect to see on the graphics is that the power decrease is always slower than during the previous experiment. Looking at the voltage variation for a given duration confirms this: the variation is between 250mV and 300mV on the first graph (which is 4.5 hours long), between 100mV and 200mV on the second graph for the first 4.5 hours, and between 25mV and 50mV on the third graph for the first 4.5 hours (during which the message rate is really low).

It would be interesting to see how the voltage variation relates to the number of messages or bytes sent. But this needs to take the DYMO routing messages into account, and the issues mentioned above prevented us to acquire much data about it. The table below shows such data for 3 motes acquired over different periods of
time. We can see that the relations are positive as expected, but certainly not linear. However, the data corresponds for different experiments and different durations, and only concerns 3 motes, they cannot be relied on.

<table>
<thead>
<tr>
<th>messages</th>
<th>bytes</th>
<th>∆ Voltage</th>
<th>∆ Voltage/messages</th>
<th>∆ Voltage/bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10718</td>
<td>255070</td>
<td>459</td>
<td>0.043</td>
<td>0.00180</td>
</tr>
<tr>
<td>251</td>
<td>5995</td>
<td>26</td>
<td>0.104</td>
<td>0.00434</td>
</tr>
<tr>
<td>5843</td>
<td>132163</td>
<td>413</td>
<td>0.071</td>
<td>0.00312</td>
</tr>
</tbody>
</table>

Initial and Final Decrease

We can see that a number of nodes see their battery voltage decrease faster at the beginning of the experiment, for about half an hour. This does not happen for all motes and only for the first two experiments, during which messages are sent periodically. However this can hardly be explained by a difference between the experiments. Indeed, the three experiments share a similar start (a command sent to all motes), and the initial messages (the command and first route discoveries) are exchanged during the first minutes, not the first 30 minutes. A possible explanation could be that this behaviour happens when the mote boots.

The graphics also shows special plots when the mote has new batteries or on the contrary when they become too weak. On the second graph, the voltage of Node4 stays at its maximum value for almost 90 minutes. Most likely, it is because the batteries have a higher voltage than 3 volts but the voltage sensor is unable to read such a high value. The actual value returned by the sensor is not in millivolts, but in a linear combination of millivolts, with a maximum value of 4095 (which is $2^{12} - 1$). So one could easily imagine that the sensor stores the value on 11 bits and returns the maximum value when the reading overflows. The other special plots are for Node6 on the first graph and Node2 on the second graph. The mote seems to demand more and more current as the batteries begin to be too weak to power all the hardware elements. Nonetheless, the experiments were not designed to examine these facts, further experiments would be necessary to confirm them and explain them properly.

The Base Station Singularity

Data for the base station is not represented here because it was powered through the USB port, so the available power was not affected by the operations of the mote. It does not send many messages, but it does receive a lot of them if the motes have not set a threshold. As a result, the relatively rare routes that the base station needs are not used several times, and the routes it obtained via received
route requests are generally not used to sent messages back. So one should set DYMO parameters specifically for the base station, which should prevent routes from being stored more than a few seconds and disturbing the insertion of new routes.

6.4 Conclusion

Testing and experimenting was the hardest part of the work. Even though TOSSIM provides useful features to debug an application, it is sometimes not easy to find the cause of a problem in a distributed application, where the ongoing computation can be preempted at any time by events. As for real hardware testing, it is even more true due to the low amount of available information about the execution.

But it was also the most teaching part. The issues we faced gave us counter-examples of designing and programming techniques for the implementation of a protocol. We realized that deploying a real size network can lead to difficulties, as one may not be able to observe and control several nodes at the same time. It is also difficult to gather data or facts quickly for some applications, because quite a long time may elapse between interesting events. We can therefore say that this part of the work gave us a good experience about experimentation techniques and problems for distributed sensor networks.
Chapter 7

Conclusions and Future Work

This chapter summarizes the thesis, accounts for the lessons learned, and introduces possible directions for future work.

7.1 Conclusion and Lessons Learned

The goals of the DYMO protocol were to provide a simple but yet generic and efficient routing protocol for Mobile Ad-hoc Networks, capable of adapting to highly dynamic topologies. In this thesis, we explored the protocol with a rather different idea in mind. We wanted to see if the constraints of Mobile Ad-hoc Networks and Distributed Sensor Networks can justify the use of the DYMO protocol with sensor networks, even if they are stationary.

Summary of the Results

We consider the generic and extensible packet format provided by PacketBB to have a too high size overhead to be used by sensor networks. In a context where the typical size of a packet is 35 bytes, the use of each byte should be carefully considered. As a consequence, we designed a DYMO-specific packet format that cannot be extended but which is simpler to parse and more compact. Though it could be useful, we did not provide in this first implementation the internetwork routing features of the DYMO protocol in order to keep things as simple as possible.

Even with this simplified version of DYMO, implementing the protocol on TinyOS was not a simple task, notably because we tried to keep two other goals in mind. First, the DYMO specifications being an IETF internet draft, we wanted to carefully examine the DYMO operations as well as the writing of their description in order to provide useful feedback to the authors of the specifications before they
are submitted to the IESG. This resulted in several discussions and modifications of the specifications that took place before we actually implemented the related functionalities. Second, we wanted not only to provide the DYMO implementation to the TinyOS community, but also to provide code and components that can be reused to implement other routing protocols. In this respect, we tried to make a clean component layout that separates all the main functionalities of a routing protocol by following published guidelines to make a network layer for sensor networks.

The resulting implementation has been run with various tests and experiments which revealed many flaws and bugs, but we believe it is now functional and should operate without trouble with small networks. The performance and scalability have not been evaluated and compared to other routing protocols, we can only say that our experience did not reveal any obvious problem in that regard.

The main evaluation we performed was about the power consumption in the context of a specific but typical sensor network application. Technical issues and time constraints did not allow to perform all the planned experiments, which reduces the available results. The results still give an idea of what we were looking for, the factors of the life expectancy of a sensor network, and provide a starting point for further or additional experimentation.

Lessons Learned

This project is our first experience of embedded programming, and as such taught us all the basic concepts. We have been introduced to the numerous challenges of embedded programming and faced the corresponding difficulties. Some of these were really hard to tackle, especially when they also involved the issues related to networking. The project was also another experience of a protocol implementation, and even though we did not design the protocol, we carefully reviewed it to participate in its improvement. As a result, we gained an insight of how all these issues can interact and be solved. This includes the manipulation of simulation tools, real experimentation, and how they relate.

More generally, we understood the challenges of routing and programming for mobile networks and sensor networks, we understood why there is so much interest in this area from the research community. We saw the work that was done and still needs to be done in the close future. The software and the documents made by the TinyOS community gave a good and interesting overview of this past and future work. It also introduced us to a successful example of how the research challenges can be tackled. We hope our work will be helpful for that purpose.
7.2 Futur Work

This section concludes by presenting different ideas that can be followed to continue, extend or improve our work.

7.2.1 Experimentation

As we did not complete the experimentation we planned, continuing the experiments is an obvious way of developing our project further. It involves performing experiments with the sets of parameters we did not have the time to realize, and possibly other sets according to the results of the experiments and the interrogations they raise. Even the existing results should be confirmed with longer and larger experiments for a better reliability.

The experiments could also involve other evaluations of characteristics such as performance, scalability and mobility handling. This kind of experiments where the software is pushed at its limits generally require to be simulated, which would involve writing simulation programs. Our simple Python scripts would not be enough for that purpose as the use of an interpreted language dramatically decreases the performances of the simulation. Furthermore, mobility simulation is not convenient in TOSSIM 2, a layer on top of it or another framework would be necessary, similarly to what has been done for TinyOS 1.x.

These experiments represent therefore a large amount of analysis and programming work.

7.2.2 Implementation

We saw that our implementation of the DYMO protocol is not a comprehensive implementation of the specifications. Some points have been simplified, mainly the packet format and internetworking features. Since large sensor network applications can take advantage of internetworks to ease routing, it would be useful to add the feature to our implementation. It would also be useful, and not a very large task, to implement the original PacketBB format to compare how DYMO operates in terms of performance and power consumption depending on the packet format. This would especially tell if redesigning the packet format was worthy.

Moreover, the implementation can certainly be improved. It was our first experience of embedded programming and TinyOS concepts, it is thus likely that a lot of details can be reviewed to improve the performance, the memory consumption and the concurrency support of our code. The possible experiments described above would be helpful to determine which points need to be corrected.
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.2.3 Design

At the moment of the writing, there are ongoing discussions to include our work in the TinyOS distribution. This means that the community is interested in such a network layer for TinyOS, but also that most likely, some people will adapt the work to their needs or reuse some part of it. We also saw that providing an implementation framework for routing protocols was one of our ideas. If this inclusion happens, reviewing the design of the implementation would be useful for two reasons.

First, the main disadvantage of the compact format is that it cannot be extended and still interoperate with the existing format. There may be some solutions to allow the extension of the format while keeping it compact, this needs further investigation. Next, to provide more reusable code, the component layout of the implementation could be redesigned to be more fine-grained so that even different kinds of routing protocol could benefit from it. The current layout was designed with an end-to-end unicast routing protocol in mind.

In other words, though we believe this work partly fulfills all of its goals, there are a lot of possibilities to develop it and we hope we will be able to realize some of it.
Bibliography


94

BIBLIOGRAPHY


