POLITECNICO DI MILANO

Master of Science in Computer Science and Engineering

Dipartimento di Elettronica, Informazione e Bioingegneria

A time deterministic communication stack for mesh networks

Relatore: William Fornaciari

Correlatore: Federico Terraneo

Tesi di Laurea di:

Federico Amedeo Izzo, 875328

Anno Accademico 2018-2019

Contents

1 Abstract **9**

2 Sommario **11**

3 Introduction **13**

3.1 An introduction to wireless networks 13

3.1.1 Case study: smart meters 14

3.1.2 Case study: industrial control 14

3.2 Goal of this work 15

3.3 Author's contribution 15

4 Literature review **16**

4.1 Low power wireless protocols 16

4.1.1 Low power listening 16

4.1.2 Channel access method 17

4.1.3 IEEE 802.15.4e 18

4.1.4 TSCH 18

4.1.5 DSME 19

4.1.6 LLDN 20

4.1.7 Comparison of TDMH with other protocols 20

4.2 Scheduling 21

4.2.1 Introduction to scheduling 21

4.2.2 TSCH scheduling 21

4.2.3 TASA 21

4.2.4 MODESA 22
4.2.5 Compatibility with TDMH 23
4.2.6 Considerations on spatial reuse of channel 24

5 Project overview **25**
5.1 Introduction to TDMH 25
5.1.1 Time synchronization 25
5.1.2 Master and Dynamic nodes 26
5.1.3 Protocol phases 27
5.1.4 Tiled frame 28
5.1.5 Downlink phase 28
5.1.6 Uplink phase 29
5.1.7 Data phase 29
5.2 Network stack design 30
5.2.1 From TDMH Mac to TDMH Stack 31
5.2.2 Streams and periods 32
5.2.3 Spatial and temporal redundancy 35
5.2.4 Spatial reuse of channels 36
5.2.5 Interference check in scheduler 37

6 Scheduling: Theory **38**
6.1 Problem statement 38
6.1.1 Routing and Scheduling 40
6.2 Formal definition of the problem 41
6.2.1 Scheduling problem 42
6.2.2 Routing problem 44

7 Scheduling: Algorithms **45**
7.1 Routing 45
7.1.1 Breadth-first search 46
7.1.2 Spatial and temporal redundancy 49
7.1.3 Limited depth-first search 50

7.2 Scheduling . . . 52
7.2.1 Greedy scheduler algorithm . . . 52

8 Schedule distribution 59
8.1 Implicit schedule idea . . . 60
8.1.1 Implicit schedule element . . . 60
8.1.2 Advantages of the implicit schedule . . . 60
8.2 Schedule packet format . . . 61
8.3 Dataphase schedule . . . 63
8.3.1 Schedule activation . . . 63
8.3.2 Dataphase schedule format . . . 63
8.3.3 Conversion algorithm . . . 64

9 Simulation 66
9.1 Python simulation . . . 66
9.2 OMNeT++ simulation . . . 70

10 Implementation 73
10.1 Object Oriented codebase . . . 73
10.2 Overview of TDMH modules . . . 74
10.3 Topology Collection . . . 75
10.4 Schedule computation . . . 76
10.4.1 Thread model . . . 76
10.4.2 Scheduler algorithm implementation . . . 78
10.4.3 Finding conflicts in implicit schedule . . . 79
10.5 Schedule distribution . . . 81
10.5.1 Finite state machine model . . . 81
10.6 Dataphase . . . 86
10.6.1 Schedule actions . . . 86
10.7 Stream manager . . . 88
10.7.1 Streams . . . 88
10.7.2 Servers . . . 88

11 Experiments **93**
11.1 First experiment: Simulation 93
11.1.1 Topology 93
11.1.2 Setup 94
11.1.3 Test procedure 94
11.1.4 Goal 95
11.1.5 Results 96
11.1.6 Conclusions 96
11.2 Second experiment: wired setup 97
11.2.1 Topology 97
11.2.2 Setup 97
11.2.3 Test procedure 99
11.2.4 Goal 100
11.2.5 Results 100
11.2.6 Conclusions 100
11.3 Third experiment: building floor setup 102
11.3.1 Topology 102
11.3.2 Setup 102
11.3.3 Test procedure 103
11.3.4 Goal 104
11.3.5 Results 104
11.3.6 Conclusions 104

12 Conclusions **106**

Bibliography **107**

List of Figures

5.1 Representation of TDMH superframe 28
5.2 ISO/OSI model chart 31
5.3 Comparison of two different stream periods 32

9.1 Network topology from RTSS paper 67
9.2 Schedule generated with the python simulator 70
9.3 Debug view of the OMNeT++ simulator 71
9.4 Simulation view of the OMNeT++ simulator 72

10.1 UML diagram showing polymorphism in the Uplink class . . . 74
10.2 Highlight of author's contribution in TDMH codebase 75
10.3 TDMH modules related to streams 76
10.4 FSM model of master schedule distribution 82
10.5 FSM model of master schedule application 83
10.6 FSM model of dynamic schedule distribution 84
10.7 FSM model of dynamic schedule application 85
10.8 FSM model of streams in dynamic nodes 89
10.9 FSM model of streams in master nodes 90
10.10FSM model of servers in dynamic nodes 91
10.11FSM model of servers in master nodes 92

11.1 Diamond topology shown in OMNeT++ 94
11.2 Diamond topology of the wired setup 97
11.3 Photo of the wired setup 98
11.4 Node placement from RTSS paper 102

11.5 WandStem nodes in their boxes 103

List of Tables

4.1 Comparison of TDMH with other IEEE 802.15.4 MACs 20

5.1 Possible choices for **period** with a `tile_duration` of 100ms . 33

11.1 First experiment results with redundancy disabled 96
11.2 First experiment results with triple redundancy 97
11.3 Second experiment results with redundancy disabled 101
11.4 Second experiment results with triple redundancy 101
11.5 Third experiment results, reliability with single, double and triple spatial redundancy 105

Chapter 1

Abstract

In these years **communication between devices** is becoming increasingly present in our lives. If we think about mobile phones or laptops, these devices couldn't even fullfill their purpose without a working wireless connection. Other fields where wireless networks are of paramount importance are for example the Internet of Things world and wireless data collection from sensors.

The most common wireless technologies can be divided roughly into **high performance** and **low power** technologies, two examples of high performance network technologies are WiFi and LTE while two low power network technologies are ZigBee or LoRa.

If we analyze the **end-to-end latency** of these technologies, we find that the latency of a WiFi network is around 1-5ms on a non congested network, but can quickly rise to 1s or more in presence of heavy traffic [1]. Meanwhile a ZigBee network have a normal latency of 50-350ms [2] which just like WiFi, can rise of several orders of magnitude in case of a congested network.

In this thesis we present **TDMH**, a wireless communication stack able to provide **bounded latency**, which means that the worst case network latency depends only on the network size and TDMH configuration and is **independent from the network load**.

The *bounded latency* makes TDMH suitable for **real time** applications like **industrial control**. TDMH is capable of managing **multi-hop** wireless **mesh networks** and its architecture allows to switch off the radio whenever a device doesn't need to transmit or receive packets, this enables **low power** applications such as being employed on battery powered devices.

The author's contribution to TDMH consists mainly in the scheduling and routing part of the network stack, which is the main focus of this thesis. The athor is responsible also for the design and implementation of several TDMH modules which are essential to the network stack operation.

You can find an overview of the main TDMH modules and the author's contribution in sections 3.3 and 10.2.

Chapter 2

Sommario

In questi anni la **comunicazione tra dispositivi** è diventata sempre piú presente nelle nostre vite. Ad esempio i telefoni cellulari o i computer portatili perderebbero di funzionalitá se non ci fosse a disposizione una connessione wireless. Altri campi in cui le reti wireless sono fondamentali sono il mondo dell'Internet of Things e la raccolta dati da sensori wireless.

Le tecnologie wireless piú diffuse si possono categorizzare in reti **ad alte prestazioni** e reti **low power**, due esempi di tecnologie ad alte prestazioni sono il WiFi e LTE, mentre due tecnologie low power sono ZigBee e LoRa.

Se analizziamo la latenza di rete di queste tecnologie, troviamo dei valori intorno a 1-5ms per una rete WiFi non congestionata, ma questi valori possono arrivare a 1s o piú in presenza di traffico elevato[1]. Mentre una rete ZigBee ha una latenza nominale di 50-350ms[2], che proprio come per il WiFi puó aumentare di diversi ordini di grandezza in una rete sotto carico.

In questa tesi viene presentato **TDMH**, uno stack di comunicazione wireless in grado di fornire una **latenza limitata**, ovvero una latenza di rete nel caso peggiore che dipende solo dalla dimensione della rete e dalla configurazione di TDMH e soprattutto è **indipendente dal carico di rete**.

La *latenza limitata* rende TDMH adatto per applicazioni **real time** come il **controllo industriale**.

TDMH è in grado di gestire reti wireless **mesh** con struttura **multi-hop**, inoltre TDMH permette di spegnere la radio quando un dispositivo non deve trasmettere o ricevere pacchetti, questo consente di risparmiare molta energia, aprendo possibilitá ad applicazioni **low power**, ad esempio su dispositivi alimentati a batteria.

Il contributo dell'autore a TDMH consiste principalmente nella parte di *scheduling* e *routing* dello stack di rete, questa tesi si concentrerá principalmente su questi due aspetti. L'autore è anche responsabile del design e dell'implementazione di alcuni moduli di TDMH che sono essenziali al funzionamento complessivo dello stack di rete.

Potete trovare una panoramica dei moduli principali di TDMH e del contributo dell' autore a questi nelle sezioni 3.3 e 10.2.

Chapter 3

Introduction

3.1 An introduction to wireless networks

The importance of **wireless networks** in the world we live in is greatly increasing, since this technology can provide various advantages like **mobility** and **flexibility**.

Mobility is really important in many of the devices we use, for example mobile phones would be pretty useless without wireless technology, another advantage is that wireless technology **doesn't require a heavy infrastructure** and can be used in different environments, on the contrary in some of these environments running cables for a wired network may not be possible.

3.1.1 Case study: smart meters

An example of a **low power** wireless network that comes with a **lightweight infrastructure** is the wireless network used to remotely collect utility measurements from **smart meters**, like gas meters or water meters.[3]

In this kind of application as in many other wireless network applications, the **low power consumption** is fundamental.

For example if we decide to replace all the gas meters in a city with smart meters, we want these to have a **battery life longer than two decades**, to avoid facing too soon the great effort of replacing the batteries of all the meters in the city.

The technology used in smart utility meters is called LPWAN, which stands for Low Power Wide Area Network. The LPWAN technology is fairly recent, we will do an overview of other low power network technologies in chapter 4.

3.1.2 Case study: industrial control

Another successful application of wireless networks is **industrial control**. In this environment, a wireless network could be used to read sensor data coming from various parts of a plant, or even to send commands to actuators. Wireless technology in this application has significant advantages over a wired network counterpart, these advantages include **lower costs**, **lower power consumption** and **higher flexibility**. The flexibility could mean adding new sensor devices without changing the current infrastructure.

The network stack we present includes these advantages, and in addition introduces **bounded latency**, which reduces dramatically the uncertainty in data latency, enabling high precision readings and faster and more reliable actuations.

3.2 Goal of this work

Low power networks comes generally with one important disadvantage: the **high latencies** that derive from power saving techniques.

The goal of the work behind this thesis is to build a **wireless network** that while being **low power**, is capable also of achieving **low latencies**, in the order of nanoseconds. This **low latency** capability allows **real-time** applications.

The network stack presented is called **TDMH**, which stands for Time Deterministic Multi Hop, an acronym that recalls the **deterministic** nature of the stack given by the bounded latency, and its ability to handle **multi-hop** networks.

3.3 Author's contribution

The author's contribution to the TDMH network stack consisted mainly in adding the missing **scheduling and routing** functionality to the stack, and additionally in designing, developing and testing several new modules of the network stack. The general goal of the work was to turn TDMH from being a **Medium Access Control** which included: time synchronization, network graph collection and packet sending, to a **full fledged network stack**, capable of opening and routing connections at runtime, and featuring several redundancy and data reliability options. This change is explained in detail in section 5.2.1.

Chapter 4

Literature review

4.1 Low power wireless protocols

As of note there is no wireless protocol that combines **low latency** with **low power consumption**, in fact high data rate protocols like WiFi (IEEE 802.11) have a notoriously high power consumption [4], while **low power** protocols like ZigBee (IEEE 802.15.4) save power at the cost of increasing latency.

4.1.1 Low power listening

One of the techniques used to reduce the power consumption of wireless protocols is the one adopted in IEEE 802.15.4 networks called **low power listening** [5], this technique is able to reduce the radio reception power consumption at the expense of the transmission-side power consumption.

This technique consists in the sender transmitting a long preamble (100ms) before every packet (2ms) and the receiver checking periodically whether the channel is free or a carrier-wave is present (*carrier sensing*), in the latter case he can tell that a packet is incoming.

The check gives a *statistical* or *deterministical* certainty about receiving a packet, depending on the period at which the checks are performed.

This approach has its **drawbacks**, for example a **collision** problem when we have more than one node that wants to trasmit, these collisions have the effect of generating **network delay**. Another downside is the fact that transmitting a long preamble drains the battery of the transmitting nodes, also the receiving node may use more power than necessary if it senses the channel in the middle of the preable, because it has to receive the remaining half of the preamble (50ms) and the packet being sent (2ms). The biggest disadvantage of low power listening is the inefficient use of the radio channel, which is mostly occupied with preambles.

4.1.2 Channel access method

An important notion for understanding wireless protocols is the **channel access method** used. If we think of two wireless radios in range and operating on the same frequency, we know that only one of them can transmit at a given time and the other can only receive, because if two radios transmit at the same time on the same frequency, the information can get lost because of the interference between the two transmissions. This is the reason why we need a **channel access method**, to organize the transmissions of the different radios and reduce or eliminate the possibility of interference.
The two most common channel access methods are **CSMA/CA** and **TDMA** [6].

CSMA/CA

CSMA/CA stands for *Carrier Sense Multiple Access with Collision Avoidance* and its principle is that a radio, before transmitting checks if the channel is free, if that is the case the radio can transmit, if otherwise there is another transmission happening, the radio waits for a random time interval, called **exponential backoff**. This type of channel access is called a **random channel access**, its advantage is that it doesn't require a centralized orchestration of the transmissions, so it adapts easily to a changing environment, a

drawback of CSMA/CA is that with heavy traffic the **latency** can become very high, and in general is not predictable.

TDMA

TDMA stands for *Time Division Multiple Access* and is a channel access method that works by dividing the time in a given number of slots called timeslots, and making sure that for every timeslot at most one radio unit is transmitting. This technique allows to avoid collisions between transmissions of the same network, since the transmissions are performed in different timeslots. The drawback of this technique is that it requires an *orchestration* of the transmissions, generally performed through a **scheduling**, we will see more of this technique because it is the one employed by TDMH. Among the advantages of TDMA, there is the **bound latency**, generally depending only on the propagation delay of the transmission and on the radio transmission and reception latency.

4.1.3 IEEE 802.15.4e

IEEE 802.15.4e [7] is an upgrade over existing standard IEEE 802.15.4, it introduces several new *physical* and *MAC level* standards, among which the most interesting ones are **TSCH** [8], **DSME** [9] and **LLDN** [10].

4.1.4 TSCH

It is the more developed standard, and the one most present in research. TSCH [8] is based on IEEE 802.15.4 **peer-to-peer architecture**, but employs a TDMA-like architecture instead of CSMA/CA proposed by the IEEE 802.15.4 standard. A TSCH network uses **beacons** sent by the "active" nodes of the network (IEEE 802.15.4 FFD, Fully Functioning Device) to synchronize the other nodes in the network. The "active" nodes in the network build a **schedule** using a distributed algorithm, the schedule is used to allocate **links** over **channels and slots**, the schedule effectively performs a

TDMA channel access mechanism.

Unfortunately, TSCH scheduling and schedule distribution are not defined in their *RFC*, in fact the design and development of these parts are left to the ones who implement the protocol. An example of TSCH implementation is the **6TiSCH** [11] protocol.

A possible limit of TSCH is that the protocol does not guarantee any boundaries on **delays** and **real-time periods**, the only guaranteed parameter is the **reliability** of packet reception.

4.1.5 DSME

DSME [9] is a *time-synchronized* MAC protocol capable of **multi-hop** transmissions, it provides some improvements over the IEEE 802.15.4 MAC specification, like the **multichannel** capability. DSME employs a TDMA channel access mechanism like TSCH, and uses also *channel diversity*, which is the simultaneous use of different radio channels.

The DSME MAC transmits frames based on a time structure called *multi-superframe*, this structure contains 16 superframe slots, each of them is divided into a *contention access period* (CAP), and a *contention free period* (CFP), the CAP portion is contended between the nodes of the DSME network, while the CFP portion is scheduled and allocated to the single nodes.

The CAP section of the DSME frame is the more power hungry, since it requires a node to listen for the whole superframe to assess if the channel is free. For this reason DSME employs a scheme called CAP reduction, that gives priority to the scheduled and more power saving CFP mode.

TSCH frames are managed and scheduled like DSME's CFP mode, and TSCH is able to get the power saving benefits of this mode.

4.1.6 LLDN

LLDN [10] is another MAC presented in the IEEE 802.15.4e standard. The LLDN MAC is limited to single hop topologies (star networks) and uses a single channel. It is aimed at data collection from sensors and is able to collect data every 10ms from 20 different sensors. The time structure of LLDN is composed by superframes and the nodes access the slots of the superframe by using CSMA/CA.

4.1.7 Comparison of TDMH with other protocols

See the table 4.1.

Protocol	TDMH	TSCH	DSME	LLDN
multi-hop	yes	yes	yes	no
guaranteed period	yes	no	no	yes
topology type	mesh	cluster-tree	cluster-tree	star
spatial redundancy	yes	no	no	no
temporal redundancy	yes	feasible	feasible	feasible
channel spatial reuse	yes	yes (MODESA)	no	no
management	central.	central./distr.	central.	central.

Table 4.1: Comparison of TDMH with other IEEE 802.15.4 MACs

4.2 Scheduling

4.2.1 Introduction to scheduling

Scheduling is the operation of allocating of a set of resources to a number of agents requiring these resources. You can find a more in depth explaination of the problem in chapter 6. In the context of wireless sensor networks, and in particular of TDMA based networks, the resource available is a number of free timeslots, and the agents are the nodes of the network that want to send data through these timeslots.

Since the solution to this problem is not trivial, in TDMA based MACs we find a software module called **scheduler** that is able to solve this problem.

4.2.2 TSCH scheduling

The most similar scheduling algorithms found in literature are two algorithms proposed for the IEEE 802.15.4e TSCH MAC, called **TASA** [12] and **MODESA** [13].

4.2.3 TASA

TASA [12] stands for Traffic Aware Scheduling Algorithm, and is a proposed scheduling scheme for TSCH. It uses a **centralized** scheduling algorithm and requires that the master node has a complete topology information of the network.

The network topology information is composed of:

- Logical network graph
- Physical connectivity graph

The TASA scheduling algorithm works as a combination of the *matching* and *vertex coloring* problems, which are solved by two algorithms. The two algorithms are applied in an *iterative* way, such that for every data slot the suitable links are selected step by step.

4.2.4 MODESA

MODESA [13] stands for Multichannel Optimized DElay time Slot Assignment, like TASA is a scheduling scheme proposed for TSCH. In MODESA, the nodes compete for a time slot if and only if they have something to transmit. Every node in a network scheduled with MODESA has a **dynamic priority**, which is calculated as $remPckt * parentRcv$, where $remPckt$ is equal to the number of packets present in the buffer, and $parentRcv$ is the total number that the parent of the node has to receive.

The schedule is calculated by selecting for any timeslot the node that has the highest priority among all the nodes having data to transmit. A node can be scheduled in the current timeslot and channel only if it doesn't interfere with nodes already scheduled on the same timeslot and channel, otherwise the node is scheduled on a different channel. This allows the **spatial reuse of channels**, since we can have multiple simultaneous transmissions on the same channel and timeslot, the check for interferences is performed on the *connectivity graph*, since the logical network graph does not use all the possible physical links.

4.2.5 Compatibility with TDMH

The TASA and MODESA algorithms cannot be applied directly to the TDMH scheduling problem because of two core differences between TSCH and TDMH:

- TSCH employs **channel hopping** while TDMH uses a **single channel**
- TSCH uses a **tree topology**, while TDMH uses a **mesh topology**

The topology difference is the more important, since in a **tree network** all packets in the network are routed through the **network coordinator**, which is at the root of the network tree, this architecture simplifies a lot the scheduling problem, but reduces the overall network efficiency, since the distance between a node of the network and the network coordinator may be higher that the distance from the source node to the destination node in a mesh network.

Convergecast scheduling

There is another reason of incompatibility between TASA, MODESA and TDMH: TASA and MODESA are created for a **convergecast** network, while TDMH is not convergecast.

A **convergecast** network is a network where packets can only flow from a number of *source* nodes to a single *sink* node, that corresponds usually with the network coordinator [14].

This model is particularly suitable for to the usecase of collecting data from wireless sensors, without the need to send any data to the sensors itself.

TDMH is not a convergecast network, since it allows **bidirectional communication** between any pair of nodes of the network, so the scheduler should follow this requirement.

4.2.6 Considerations on spatial reuse of channel

The protocol we analyzed in this literature review have different approaches to **spatial reuse of channel**.

For example **WirelessHART** avoids on purpose the spatial reuse of channels when scheduling, this is done to avoid transmission failure due to interference between simultaneous transmissions [15].

On the contrary **TSCH**, in particular with the **MODESA** scheduler takes advantage of spatial reuse of channel by using a **connectivity graph** that is a superset of the **network graph**, containing all the link in radio range between nodes. Spatial reuse of channel is enabled for links without interference while links with interference are scheduled different channels [13].

Chapter 5

Project overview

5.1 Introduction to TDMH

TDMH is a **real-time** network stack capable of **low power** consumption. It is based on the IEEE 802.15.4 Radio standard and its network structure and algorithms are **centralized**. Regarding the network organization, TDMH is structured as a **mesh network**, and is capable of **multi-hop** transmissions.

TDMH relies heavily on **time-synchronization**, thanks to which all the nodes in the network have their internal clocks synchronized, this allows doing a TDMA (see section 4.1.2) access to the radio, without the need for **carrier sensing**, which we have seen in the literature review (chapter 4) having a great disadvantages in the form of latency and power consumption.

5.1.1 Time synchronization

The **time synchronization** part of TDMH is performed by employing the FLOPSYNC-2 [16] scheme, which is particularly effective in compensating the various sources of clock error and desynchronization, like jitter, temperature drift or clock errors caused by the PLL.

FLOPSYNC-2 consists in a **model** of the local clock, coupled with a

controller able to correct its non-idealities. This scheme is capable of high precision, below the μs, with a power consumption of less than $2.1\mu W$. Other benefits of FLOPSYNC-2 include a **monotonic** and **continuous** clock, except for when the nodes resynchronize after losing connection to the network.

5.1.2 Master and Dynamic nodes

As hinted before, TDMH is a centralized protocol, which means that there is a special node in every network which orchestrates the entire network, this node is called the **Master node** and its tasks include keeping the main clock at which the other nodes of the network synchronize theirs to. The master node is also in charge of gathering the network topology and connection requests, and evaluate these two to grant or deny the opening of a Stream, this happens during the Scheduling phase, of which we will talk in the next chapters.

The other nodes of the TDMH network apart from the only Master node are called **Dynamic nodes**, this to indicate the fact that they might change their position in space, while the Master node is more likely to have a fixed position.

Dynamic nodes as opposed to the Master node, cooperate and forward each others' transmission to perform three main tasks:

- Synchronizing their **clock** with the clock of the Master node
- Collecting the **network topology** graph
- Collecting information about **connection requests** or Streams
- Send the **application data** using Streams

We have cited in this section the concept of Stream, which is specific to TDMH. A Stream is a connection between two arbitrary nodes of the network, and can be thought as the omologous of TCP/IP sockets.

TDMH Streams have some peculiarities with respect to Sockets, and we will talk in detail about these in the section 5.2.2 and following.

5.1.3 Protocol phases

As we said, TDMH is a TDMA protocol, and as such all the nodes have a notion of the **network time**, which is sychronized among the whole network. The network time is used to divide the time in different **phases**, which are used to carry out the various tasks of the network stack.

The phases are three, and can be distinguished based on the direction of information in of each of these.

The three network phases are:

- The **Downlink** phase in which the Master node distributes information to the rest of the network.
- The **Uplink** phase in which the Dynamic nodes gather information about the network and forward it to the Master node.
- The **Data** phase in which the Master and Dynamic nodes send application data to other nodes of the same network.

5.1.4 Tiled frame

The network phases we listed are organized such that they can alternate and repeat in a sequence that is known to all the nodes of the network, as the exact number and sequence of the phases is part of the TDMH configuration.

In particular the time is divided into **frames**, which are time periods with a default lenght of $100ms$, which are used to measure the time passed in the network stack.

The phases repetition is done by creating two types of frames called **tiles**, each of these tiles can be of two types, based on the TDMH phase is contains. The tiles alternate and repeat always in the same sequence, as we said before, this sequence if part of TDMH configuration. You can see an example of TDMH tiles in figure 5.1.

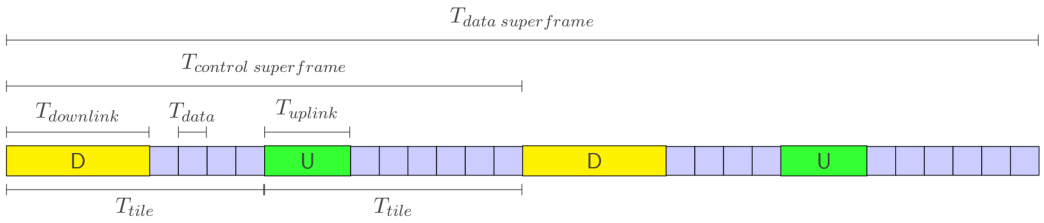

Figure 5.1: Representation of TDMH superframe

5.1.5 Downlink phase

The **Downlink** phase has two functions: its used to distribute information from the Master node to the other nodes in the network, but it is also used to keep the clocks of the Dynamic nodes synchronized with the one of the Master node. The part of the Downlink phase which is related to clock synchronization is called **Timesync**.

The information which is distributed consists in:

- **Clock sychronization packets** in the Timesync Downlink part
- **Network schedule**, in the Schedule Distribution part
- **Signaling messages** called Info Elements

Flooding

Packets in the Downlink phase are propagated in the network using a **flooding** technique, which takes advantage of the **constructive interference**. Constructive interference is the phenomenon that occurs when packets with the same content are transmitted at the same time by all the nodes of a certain hop distance, and the single resulting packet is received by all the nodes at the next hop. The technique used for flooding packets using constructive interference is a modified version of Glossy [17].

5.1.6 Uplink phase

The **Uplink** phase is used to collect information from the Dynamic nodes towards the Master node, the information collected is the following:

- **Network Topology** which is used to synchronize the clocks
- **Stream Management Elements** which represent a request for opening a Stream.

5.1.7 Data phase

The **Data** phase is used by the Master and Dynamic nodes of the network to communicate and transfer application payloads over the network.

The Data phase works by performing the playback of a TDMA schedule that is computed by the Master node, and then distributed to the Dynamic nodes of the network in the Downlink phase.

5.2 Network stack design

The goal of the work behind this thesis was to **expand and improve TDMH** to make it able to be used to send data between two arbitrary nodes, without knowing *a priori* the details of the *stream* of data being sent in the network.

The **main features** of TDMH that were developed during this work are:

- **Real-time streams** to transmit data
  with **constant period** and **bounded latency**
- **Spatial and temporal redundancy** of transmitted data
  for improved **reliability** and **resilience**
- **An automatic scheduler** for scheduling streams, taking count of the **network topology** and **avoiding conflicts**
- **Spatial reuse of channels** of the scheduled streams
- **A schedule distribution** system

The developement of these features needed a **thorough design** before the actual implementation, in this section we will present the design decisions behind the features of TDMH listed above.

5.2.1 From TDMH Mac to TDMH Stack

During the developement we realized that we were **expanding the scope of TDMH** from being a **Medium Access Control** to being a **proper Network Stack**.

This decision was not taken lightly, as we would have preferred to adhere to a **layered network model** like ISO/OSI [18]. However this was not possible for lack of support for crucial features of TDMH Streams in the layer **4 - Transport** and upper. The unsupported features are **periodic real-time streams** and **redundancy**.

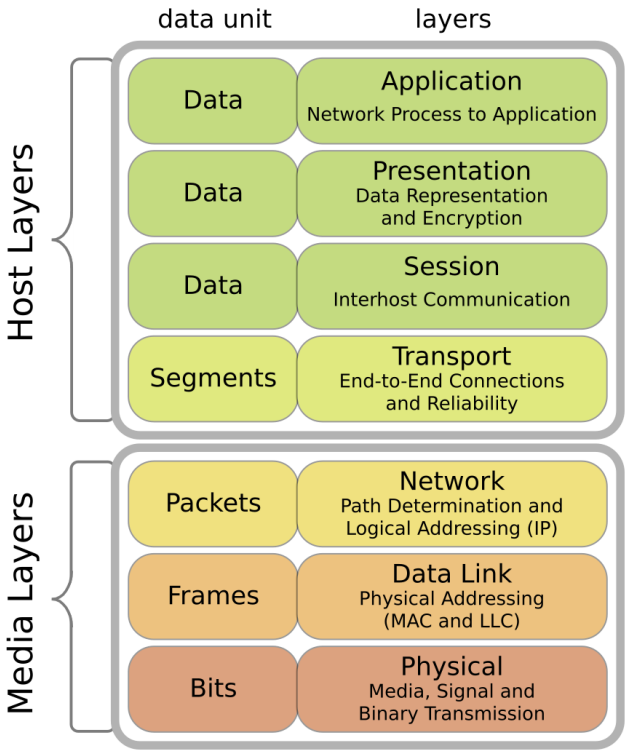

Figure 5.2: ISO/OSI model chart

5.2.2 Streams and periods

The ability to send and receive data through **real-time Streams** is one of the core features of TDMH. The two main characteristics of TDMH Streams are **constant period** and **bounded latency**.

Period

The main feature of a **real-time** stream is its **Period**. The period determines what is the **distance in time** from one transmission of the stream and the following transmission of the same stream. You can see in figure 5.3 a comparison between a stream of Period 1 and another stream of Period 2.

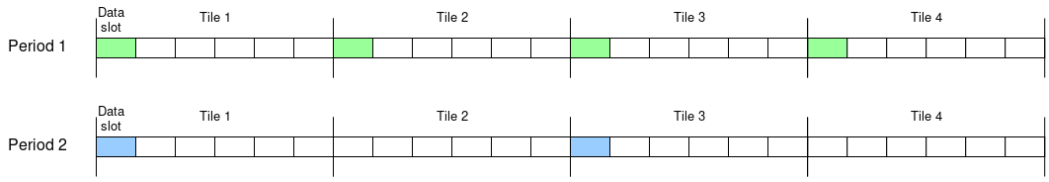

Figure 5.3: Comparison of two different stream periods

Which period to choose depends largely on the **needs of the application** that will make use of the TDMH stream, in particular the choice must be based on the **frequency** at which the data needs to be sent or received. We know that the period can be calculated as the inverse of the frequency:

$$P = \frac{1}{f}$$

For example a **control loop** running at a frequency of 1 Hz may specify a period of 1 s between one transmission and the next one, while an application reading data from a sensor at 10 Hz, needs to specify a period of 100 ms.

The period for a stream cannot be chosen in an arbitrary way, in fact there is a list of **available periods**, which are calculated in the following way:

K	Period
1	100ms
2	200ms
5	500ms
10	1s
20	2s
50	5s
100	10s
200	20s
500	50s
1000	100s
2000	200s
5000	500s
10000	1000s

Table 5.1: Possible choices for **period** with a `tile_duration` of 100ms

$$P(K) = K * tile\_duration$$

$$K \in N, K[1, N]$$

Where `tile_duration` is the time lenght of the TDMH tile (see chapter 5.1.4) and the K parameter indicates what multiple of `tile_duration` we want to use as period.

With the `tile_duration` default value of 100 ms, the possible periods are shown in table 5.1. Predefined period values will be denoted with PK, $K \in [1, N]$ for example P1 corresponds to 100ms, P50 to 5s and so on.

You may have noticed that only certain values of K are being shown on the table. In fact TDMH supports for K only integers with 1,2 or 5 as the *leftmost* figure, and a number of 0s for the other figures.

This is not a coincidence but is a solution to the problem explained below.

Length of a schedule

If we think of a schedule as a sequence of TDMA time slots, grouped in blocks of size equal to a TDMH tile. A schedule for a given stream must have some of the slots indicating the activity of the nodes of that stream, and is generally long a number of tiles equal to the K factor. If we denote the length of a schedule in number of tiles as *schedule_tiles* we have that:

$$schedule\_tiles(P) = K$$

When we have more than a stream, for example with two streams, the schedule length becomes the *least common multiple* of the periods of the two streams, this because the schedule needs to specify not only the slots used by a stream in its period, but all the possible combination of the slots used by the two streams, and the combinations of the slots are cyclic with a cycle long $lcm(P1, P2)$. So for a number N of streams, the lenght of a schedule is:

$$schedule\_tiles(P1, P2, ..., PN) = lcm(P1, P2, ..., PN)$$

The choice of numbers starting with either 1,2 or 5 comes from the Frobenius coin problem, in fact these three numbers can be used to obtain any higher number, and at the same time their *lcm* remains low [19].

Data rate

Choosing a period for a stream among the available ones means also **determining the data rate** of the stream, since the corresponding data rate for a given period can be calculated in the following way

$$Data\ rate = \frac{payload\_size}{Period}$$

Where *payload_size* is the number of packets sent per transmission, and the $Period$ is expressed in seconds.

5.2.3 Spatial and temporal redundancy

Reliability in receiving data over networks is fundamental, because **data loss** can happen due to packet loss on wired network and interference or signal loss in wireless networks. Generally we want a way to mitigate the effect of data loss. In case of TCP/IP networks this is done by employing the TCP protocol itself, that makes sure that all data sent is also received, and if some data is lost along the way, it is retransmitted until it is received.

TCP's solution of **error detection** is not suitable to TDMH, because the packet loss effects are mitigated by resending data, and in a real-time network we don't want to retransmit data based on the success of the previous transmission, because doing that would create a non predictable increase of the latency. In fact in certain environments keeping the latency low is crucial, and a data that has been resent becomes old and possibly useless with respect to data delivered in time.

Our mitigation to the problem of data loss over radio interfaces comes in the form of **transmission redundancy**, in pratice data is sent two or three times within a period, to increase the probability of receiving at least one of the two or three copies of the data.

In particular TDMH supports two kind of redundancy:

- **spatial redundancy**: the two copies are sent over two different paths on the network (possibly without common intermediate nodes)
- **temporal redundancy**: the two copies are sent over the same path

Clearly **spatial redundancy** is not always applicable in fact we may not have two paths without common intermediate nodes, in this case the scheduler will **downgrade the redundancy type** from spatial to temporal.

Finally, it is important that the two paths do not have common intermediate nodes, because these nodes would become the **single point of failure**

of the redundant stream.

5.2.4 Spatial reuse of channels

We wanted to apply another optimization to TDMH, since it is a TDMA network in which the master node knows and decides exactly in which instant every node of the network is going to send or receive messages on the radio.

We decided to develop an **interference model** based on the network graph, to enable the **spatial reuse of channels**, that is the possibility for two or more nodes to send and receive packets simultaneously.

The simultaneous transmissions are allowed by the **scheduler** once it verifies that an interference cannot happen.

Summarizing, in a TDMA without spatial reuse, only one node in the network can transmit at a given time

Without spatial reuse of channels

1	2	3	4	5	6
T1	T2	T3	T2	T4	

With spatial reuse of channels

1	2	3	4	5	6
T1	T3	T2			
T2		T4			

As previously said, the **spatial reuse of channel** is possible thanks to the **interference check** performed in the scheduler, which will be explained in the following section.

5.2.5 Interference check in scheduler

To model the **radio interference** between two nodes transmitting and receiving at the same time, we start from the following assumptions:

- Two nodes in radio range are connected in the topology map
- Two nodes not connected in the topology map do not interfere with each other

The case in which two nodes are too distant to communicate properly but close enough to interfere each other is considered unlikely, because of the **power capture** and **delay capture** effects of the O-QPSK modulation employed [17].

Interference hypotesis

We have an **interference** if and only if we have one of these two situations:

- A node is **trasmitting** and at least one of its neighbors is **receiving** for a different transmission.
- A node is **receiving** and at least one of its neighbors is **transmitting** for a different transmission.

The four possible cases are summarized in this truth table:

Node A	Node B	Interference
TX	RX	YES
TX	TX	NO
RX	TX	YES
RX	RX	NO

Chapter 6

Scheduling: Theory

6.1 Problem statement

Scheduling in Computer Science generally means finding the best mapping between a set of jobs and the resources that can fullfill these jobs. This mapping is usually done by following a certain criterion, with the aim of maximizing a particular metric, for example fairness of use or quality of service. An example of scheduling is usually found in Operating Systems Architecture, in which the so called scheduler manages the allocation CPU time to Processes. In this case the metric can be from keeping the latency of a given process low to ensuring the fair use of the CPU among Tasks.

In the case of TDMH, being a TDMA network stack, the resources we are talking about are the slots of time, and the jobs to allocate are the transmissions from the nodes of the network.

More in detail: When an application running on a TDMH enabled node wants to send data to another node in the same network, it requests the TDMH stack running on the node to open a **Stream** to that other node. We can think of a TDMH Stream as omologous to a TCP/IP socket.

This request gets forwarded by the network to the master node, the special node that coordinates the network, the master node can evaluate the stream request, and basing its decision on the current **network topology**, other opened streams and network configuration, decide if it is possible to open the requested stream and which TDMA time slots to allocate it to.

A **Stream request** is composed by the following information:

- Source node ID
- Destination node ID
- Source Port
- Destination Port
- Period
- Size of the payload (in packets)
- Redundancy level

The **Network Topology** of TDMH is defined as a **partial mesh**, or a generic graph that is *not fully connected*, in which:

- Graph **nodes** represent the physical nodes forming the network
- Graph **edges** mean that the two nodes are in radio range

The **Network Configuration** of TDMH contains information about how the *superframe* is structured, which slots of the TDMA frame are available for data transmission and which other are needed for the other TDMH phases and cannot be used for data.

6.1.1 Routing and Scheduling

An important distinction to make is between the **Routing** and **Scheduling** problems, both of them are required to be solved in the scheduling of a generic Stream, but they are entirely different.

Scheduling

We previously said that the **Scheduling** problem consists in placing a requested stream in the suitable TDMA time slots.

This placement can be made by directly running the **Scheduler algorithm** only for Streams connecting two nodes in radio range, this situation can be seen in the topology as a **graph edge** connecting the two nodes in the Network graph.

The type of stream we just described is called **single-hop**, but they are not the only type of stream admitted by TDMH, in fact TDMH also supports **multi-hop** streams.

Routing

Multi-hop streams are data communication between two nodes that are not in direct radio range, but are still *connected* in the network graph, so the data can be sent by the source, forwarded by one or more intermediate nodes in the network and finally be received by its destination.

The **Routing problem** is the task of finding the best sequence of connected intermediate nodes between source of destination to **route** the multi-hop stream through the network.

The TDMH Router accepts as input a single multi-hop stream, and produces as output a **path on the graph**, then converted to a list of two or more single-hop *Transmissions*. These transmissions, when properly scheduled make the multi-hop stream work.

Summarizing, the **Scheduler** fills the TDMA slots with chosen Streams, but accepts as input only **single-hop** Streams, while **multi-hop** Streams needs to be processed first by the **Router**, that converts them into a list of **single-hop** transmissions that the Scheduler can handle, these transmissions are then scheduled in place of the original multi-hop Stream.

6.2 Formal definition of the problem

The inputs of the **scheduling and routing problems** are:

- **TDMH configuration**
- **Network topology** graph
- List of **Stream requests**

While the output is:

- **Schedule**: list of Schedule Elements, each element containing an action for every node of the network, per every dataslot of the superframe

An **action** can be Sleep, Transmit or Receive.

The **schedule** is obtained by running the **Router** on the stream requests, to find a path for multi-hop requests and break them down to single-hop transmissions. The routed streams obtained by the Router are then fed into the **Scheduler** that tries to allocate them in the dataslots, obtaining the final schedule.

The **TDMH configuration** is:

- N = Maximum number of nodes of the network
- M = Maximum number of hops admitted in the network
- D = Number of available dataslots per slotframe

The **Network topology** graph is defined as:

- Node $n_i,\ i \in [0, N]$
- Edge $e_i = (n_i, n_j),\ i, j \in [0, N]$
- Graph $G:\ e_i \in G \iff n_i$ and n_j are in radio range (can receive each others' packets)

A generic **Stream** request $s_i,\ i \in [1, S]$ is defined as:

- $s_i = (n_i, n_j, p, h),\ i \in [1, S]$
- $S =$ Number of stream requests
- $n_i \in N$ Source node
- $n_j \in N$ Destination node
- $per(s_i) = p$ Stream period
- $red(s_i) = h$ Desired redundancy level

6.2.1 Scheduling problem

The **Scheduler** operates on routed streams $r_i,\ i \in [1, S]$, containing only single-hop transmissions. The goal of the scheduler is to place the routed streams in the dataslots, satisfying the constraints and avoiding possible conflicts.

Constraints

- **Period** constraint: The period between one stream transmission and the following must be maintained
- **Causality** constraint: An intermediate node can retransmit a packet only after receiving it.

Conflicts

Two transmissions belonging to different streams can conflict if their nodes are scheduled to the same data slot and one of the following condition is true:

- **Unicity** conflict: One node transmits AND receives on the same dataslot
- **Interference** conflict: One node transmits while a neighbor node that is not the recipient of the transmission is listening for a different transmission. For a more detailed explaination see chapter 5.2.4

Scheduler definition

- $k = (n_i, d_i, r)$ Schedule Element
- $n_i \in [0, N]$ Sender/Receiver node
- $d_i \in [0, D]$ Data slot
- $r = \{TX, RX\}$ Radio activity
- $SK = \{k_0, k_1, ..., k_n\}$ Schedule
- $\forall r_i = (T_i, p, h),\ i \in [1, S], \forall t \in T_i :$
  $\exists k_i(tx(t), d_i, TX), k_j(rx(t), d_j, RX), d_i = d_j$
- $\forall k_i = (n_i, d_i, r_i) :$
  $\nexists k_c = (n_c, d_c, r_c), d_c = d_i \wedge n_c = n_i \wedge$
  $\nexists k_d = (n_d, d_d, r_d), d_d = d_i \wedge r \neq r_d, e_k(n_d, n_i) \in G$
- $\forall t \in T_i, : \nexists k_e = (n_e, d_e, r_e), k_f(n_f, d_f, r_f),$
  $n_e = tx(t) \wedge n_f = rx(t) \wedge d_e > d_f$

6.2.2 Routing problem

The **Routing problem** accepts as input a set of generic streams $s_i,\ i \in [1, S]$ If a stream is single-hop, it doesn't need routing, if it is multi-hop and a path on the graph exists, the router calculates the corresponding routed stream $r_i,\ i \in [1, S]$, composed of single-hop transmissions.

- $s_h = (n_i, n_j, p, h),\ h \in [1, S],\ i, j \in [0, N]$ Stream
- $t_k = (n_i, n_j),\ i, j \in [0, N]$ Transmission
- $T = \{t_1, t_2, ..., t_n\},\ n \in [1, M]$ List of transmissions
- $\forall t_k \in T,\ t_k = (n_i, n_j) : tx(t_k) = n_i,\ rx(t_k) = n_j$
- $\forall s_i \in S,\ \exists r_i = (T(n_i, n_j), p, h), e_h = (n_i, n_j) \in G\ \vee$
  $\exists r_i = (T, p, h), \forall t_k = (n_i, n_j) \in T,\ \exists e_h = (n_i, n_j) \in G$
  $\wedge\ tx(t_1) = src(s_i) \wedge\ rx(t_n) = dst(s_i)$
  $\wedge\ \forall t_h, t_k, k = h + 1, rx(t_h) = tx(t_k)$

A **routed Stream** $r_i,\ i \in [1, S]$ is defined as a list of one or more transmissions. The number of hops of the stream corresponds to the transmission set cardinality.

- $r_i = (T_i, p, h),\ i \in [1, S]$
- $T_i = \{t_1, t_2, t_3, ..., t_n\},\ i \in [1, S],\ n \in [1, M]$
- $hop(r_i) = |T_i|,\ hop(r_i) \in [1, M]$
- $t_i = (n_i, n_j),\ t_i \in G$

Chapter 7

Scheduling: Algorithms

7.1 Routing

The **Router** is the TDMH module responsible for finding a suitable path between the source and destination nodes on the network graph for every multi-hop stream request. The path found is used for breaking down the **multi-hop** stream in two or more **single-hop** transmissions, so that the scheduler can schedule them.

The network graph we are talking about is an **undirected, unweighted** and **partially connected** graph, and in such a graph there may be more than one path between two nodes. Our goal in case of multiple paths being available is choosing the **shortest one** (in number of hops), this to **minimize the transmission time** between source and destination.

The problem of finding the shortest path in a generic graph or **shortest path problem** is well known in literature, to solve this problem we employed the **breadth-first search** (Moore, 1959; Zuse, 1972), which is **optimal** [20] for unweighted and undirected graphs.

7.1.1 Breadth-first search

The breadth first search algorithm is a graph search that starts from the source node in the graph and visits all the neighbor nodes at distance 1, when all the nodes at distance 1 are visited, the algorithm visits the neighbors of neighbors (distance 2) and so on until the destination node is found.

This algorithm has temporal complexity $O(E + V)$ with E = number of edges and V = number of vertices, and it's the best algorithm known for undirected unweighted graphs.

Also the breadth-first search is **guaranteed to find the shortest solution first** because the paths are found with the distance from root strictly increasing. After the first solution is found, the algorithm could possibily explore further the graph and find every other solution. However we are not interested in other solutions longer than the first one found, so we are going to stop the algorithm once the first and shortest solution is found, this saves computation time.

Python code

Here is the Python code for breadth-first search, taken from the Python simulation of the scheduler (see chapter 9).
Note that these Python simulations are used only as a pseudocode, and TDMH runs a C++ implementation of the scheduler and router algorithm.

```
def breadth_first_search(topology,stream):
    # Breadth-First Search for topology graph
    # Data structures
    open_set = queue.Queue()
    visited = set()
    parent_of = dict()  # key:node value:parent node
    src, dst = stream;  # Stream tuple unpacking
    root = src
    parent_of[root] = None
    open_set.put(root)
    while not open_set.empty():
        subtree_root = open_set.get()
        if subtree_root == dst:
            return construct_path(subtree_root,\
                                  parent_of)
        for child in adjacence(topology, subtree_root):
            if child in visited:
                continue;
            if child not in list(open_set.queue)
                parent_of[child] = subtree_root
                open_set.put(child)
        visited.add(subtree_root)
```

```
def construct_path(node, parent_of):
    path = list()
    # Continue until you reach the root (parent=None)
    # Append destination node to avoid losing it
    path.append(node)
    while parent_of[node] is not None:
        # Skip the destination node saved above
        node = parent_of[node]
        path.append(node)
    path.reverse()
    return path
```

Sample output

```
Starting Breadth-First search
bfs open set: [6]
parent_of: {6: None}
bfs open set: [8, 2, 4]
parent_of: {6: None, 8: 6, 2: 6, 4: 6}
bfs open set: [2, 4, 5, 7]
parent_of: {6: None, 8: 6, 2: 6, 4: 6, 5: 8, 7: 8}
bfs open set: [4, 5, 7]
parent_of: {6: None, 8: 6, 2: 6, 4: 6, 5: 8, 7: 8}
bfs open set: [5, 7]
parent_of: {6: None, 8: 6, 2: 6, 4: 6, 5: 8, 7: 8}
bfs open set: [7, 0, 1, 3]
parent_of: {6: None, 8: 6, 2: 6, 4: 6, 5: 8, 7: 8, 0: 5, 1: 5, 3: 5}
bfs open set: [0, 1, 3]
parent_of: {6: None, 8: 6, 2: 6, 4: 6, 5: 8, 7: 8, 0: 5, 1: 5, 3: 5}
Primary Path (BFS): [6, 8, 5, 0]
Routing (6, 0) as [(6, 8), (8, 5), (5, 0)]
```

From the last line of the output we can see that the multi-hop stream $(6, 0)$ has been routed to the three single-hop transmissions $(6, 8)$, $(8, 5)$ and $(5, 0)$. this **transmission block** will be **scheduled** making sure that it remains **atomic and sequential**.

7.1.2 Spatial and temporal redundancy

Temporal redundancy

Enabling **temporal redundancy** for a stream means that the stream data will be sent twice or three times over the network. This is useful to improve the **reliability** of the said stream, in fact in case one of the transmissions is not received correctly (e.g. because of an interference), the recipient can still get the data during one of the redundant transmissions.

The **temporal redundancy** can be obtained easily in the router, to do so we duplicate or triplicate (depending on the redundancy level required) the **transmission block** we obtain from the breadth first search. Obviously the duplicated or triplicated data must be handled by at least another component of TDMH to avoid the recipient application getting unwanted copies of the same data. This task is performed by the **Dataphase** while the schedule is executed by the nodes, see chapter 10.6 for details.

Spatial redundancy

The **spatial redundancy** for a stream consists in sending two copies of the data over two **distinct paths** on the network, two distinct paths are paths in which the intermediate nodes of the **primary path** do not appear in the intermediate nodes of the **secondary path**.

To achieve the spatial redudancy we need to obtain a **secondary path** without common intermediate nodes with the primary path.

A complete search of the graph to find a secondary path could be expensive in terms of computation time, and may retrieve paths that are **much longer** than the primary path, these paths are not so useful because they could introduce an unwanted delay in the reception of the streams with spatial redundancy.

7.1.3 Limited depth-first search

With the goal of finding secondary paths with the same lenght or slightly longer than the primary path, we adopted the idea of doing a **depth limited search**, in particular the algorithm we chose to implement is the **limited depth-first search**, which we **run to the completion** with the length limit equal to the lenght of the primary path plus a configurable parameter, subsequently called `more_hops`.

The **limited depth-first search** temporal complexity of $O(l)$ with l being the limit equal to the primary solution lenght plus `more_hops`. This algorithm is optimal like breadth-first search but uses less memory because it doesn't need to store the `parent_of` relations of all visited nodes.

The algorithm starts at the source node of the stream and explores the graph by following a path and avoiding visiting already visited nodes. This allows the algorithm to visit the longest paths from the start node, until the limit is reached, then it starts again from the source node and chooses a different path, still avoiding choosing already visited nodes.

When the destination node is found, the algorithm *backtracks* and reconstruct the path from the destination, up to the source node.

Python code

The Python code for depth-first search applied to path finding, taken from the Python simulation (see chapter 9)

```
def dfs_paths(graph,start,target,limit,path=None):
    if path is None:
        path = [start]
    if start == target:
        yield path
    for next in set(adjacence(graph,start))-set(path):
        if limit == 0:
            continue
        limit -= 1
        yield from dfs_paths(graph,next,target,limit,\
                                path+[next])
```

Sample output

```
Primary Path (BFS): [6, 8, 5, 0]
Routing (6, 0) as [(6, 8), (8, 5), (5, 0)]
Primary path length: 4
Searching secondary path of max length: 4+2= 6
DFS solutions: [[6, 8, 2, 4, 5, 0], [6, 8, 2, 4, 7, 0],
[6, 8, 2, 7, 0], [6, 8, 4, 2, 7, 0], [6, 8, 4, 5, 0],
[6, 8, 5, 0], [6, 8, 7, 0], [6, 2, 8, 4, 5, 0],
[6, 2, 8, 5, 0], [6, 2, 7, 0]]
Middle nodes [8, 5]
Found indipendent path
Secondary Path (limited-DFS): [6, 2, 7, 0]
Routing (6, 0) as [(6, 2), (2, 7), (7, 0)]
```

7.2 Scheduling

After running the router on our initial list of streams, we should have a new list, containing both the single-hop streams that didn't need routing and the multi-hop streams that have been routed through their **shortest path** into a block of **single-hop transmissions**. Among these transmission-blocks, some have been duplicated or triplicated for **redundancy**, these redundant blocks are scheduled normally because the scheduler itself has no particular role in the redundancy, other that checking for conflicts as for any other stream.

Summarizing, the scheduler work is to place the already routed streams and transmission blocks into the TDMA slots available for data transfer, this fullfilling the constraints and avoiding the conflicts as explained in chapter 6.2.1.

7.2.1 Greedy scheduler algorithm

To complete this task we designed a custom **greedy algorithm** that:

- **Sort the streams** to have the highest-period streams first
- **Iterates over the streams** and tries to allocate them to the first available timeslot
- **Checks for conflicts** and makes sure that the constraints are satisfied
- If there are conflicts, the scheduler **tries the next available timeslot**
- If on the next timeslot there are no conflicts, the **stream is scheduled**
- If there are no more timeslots available, the **stream is not scheduled**.

Below you can find the Python code for the greedy scheduler algorithm, taken from the Python scheduler simulation (see chapter 9)

Greedy scheduler algorithm

```
def scheduler(topology,req_streams,data_slots):
    schedule = []
    for stream_block in req_streams:
        # last_ts guarantees sequentiality in blocks
        # and avoids conflicts between two consecutive
        # streams in a block.
        last_ts = 0
        err_block = False;
        num_streams_in_block = 0;
        for stream in stream_block:
            # If a stream in a block cannot be
            # scheduled, undo the whole block,
            # then break block cycle
            if err_block:
                for i in range(num_streams_in_block):
                    schedule.pop();
                    schedule.pop();
                break;
            for timeslot in range(last_ts,data_slots):
                # Stream tuple unpacking
                src, dst = stream;
                conflict = False;
                err_unreachable = False;
                ## Connectivity check:
                # edge between src and dst nodes
                if not is_onehop(topology,stream):
                    err_block = True;
```

```
            err_unreachable = True;
            break; #Cannot schedule transmission
        ## Conflict checks
        # Unicity check:
        # no TX and RX for the same node
        # on the same timeslot
        conflict |= check_unicity_conflict(
                    schedule,timeslot,stream)
        # Interference check:
        # no TX and RX for nodes at 1-hop
        # distance in the same timeslot
        # Check TX node for RX neighbors
        conflict |= check_interference_conflict(
                    schedule,topology,timeslot,
                    src,'RX')
        # Check RX node for TX neighbors
        conflict |= check_interference_conflict(
                    schedule,topology,timeslot,
                    dst,'TX')
        # Checks evaluation
        if conflict:
            # Try to schedule in next timeslot
            continue;
        else:
            last_ts = timeslot
            num_streams_in_block += 1
            # Adding stream to schedule
            schedule.append((timeslot,src,
                             'TX'));
            schedule.append((timeslot,dst,
                             'RX'));
```

```
                    # Successfully scheduled
                    # transmission, break
                    # timeslot cycle
                    break;
                ## Next transmission in block should start
                ## from next timeslot to satisfy
                ## the causality constraint
                last_ts += 1
                # If we are in the last-timeslot
                # and have a conflict
                # the stream is not schedulable
                if timeslot == data_slots - 1:
                    err_block = True;
    return schedule;
```

Sample output

```
Final stream list: [[(3, 0)], [(6, 8), (8, 5), (5, 0)],
[(6, 2), (2, 7), (7, 0)], [(4, 5), (5, 0)], [(4, 7), (7, 0)]]
Checking stream (3, 0) on timeslot 0
Scheduled stream (3, 0) on timeslot 0
Checking stream (6, 8) on timeslot 0
Scheduled stream (6, 8) on timeslot 0
Checking stream (8, 5) on timeslot 1
Scheduled stream (8, 5) on timeslot 1
Checking stream (5, 0) on timeslot 2
Scheduled stream (5, 0) on timeslot 2
Checking stream (6, 2) on timeslot 0
Conflict Detected! src or dst node are busy on timeslot 0
Conflict Detected! TX-RX conflict between node 6 and node 8
Conflict Detected! TX-RX conflict between node 2 and node 6
```

```
Cannot schedule stream (6, 2) on timeslot 0
Checking stream (6, 2) on timeslot 1
Conflict Detected! TX-RX conflict between node 2 and node 8
Cannot schedule stream (6, 2) on timeslot 1
Checking stream (6, 2) on timeslot 2
Scheduled stream (6, 2) on timeslot 2
Checking stream (2, 7) on timeslot 3
Scheduled stream (2, 7) on timeslot 3
Checking stream (7, 0) on timeslot 4
Scheduled stream (7, 0) on timeslot 4
Checking stream (4, 5) on timeslot 0
Conflict Detected! TX-RX conflict between node 4 and node 8
Conflict Detected! TX-RX conflict between node 5 and node 3
Cannot schedule stream (4, 5) on timeslot 0
Checking stream (4, 5) on timeslot 1
Conflict Detected! src or dst node are busy on timeslot 1
Conflict Detected! TX-RX conflict between node 4 and node 5
Conflict Detected! TX-RX conflict between node 5 and node 8
Cannot schedule stream (4, 5) on timeslot 1
Checking stream (4, 5) on timeslot 2
Conflict Detected! src or dst node are busy on timeslot 2
Conflict Detected! TX-RX conflict between node 4 and node 2
Cannot schedule stream (4, 5) on timeslot 2
Checking stream (4, 5) on timeslot 3
Conflict Detected! TX-RX conflict between node 4 and node 7
Cannot schedule stream (4, 5) on timeslot 3
Checking stream (4, 5) on timeslot 4
Conflict Detected! TX-RX conflict between node 5 and node 7
Cannot schedule stream (4, 5) on timeslot 4
Checking stream (4, 5) on timeslot 5
Scheduled stream (4, 5) on timeslot 5
```

```
Checking stream (5, 0) on timeslot 6
Scheduled stream (5, 0) on timeslot 6
Checking stream (4, 7) on timeslot 0
Conflict Detected! TX-RX conflict between node 4 and node 8
Cannot schedule stream (4, 7) on timeslot 0
Checking stream (4, 7) on timeslot 1
Conflict Detected! TX-RX conflict between node 4 and node 5
Conflict Detected! TX-RX conflict between node 7 and node 8
Cannot schedule stream (4, 7) on timeslot 1
Checking stream (4, 7) on timeslot 2
Conflict Detected! TX-RX conflict between node 4 and node 2
Conflict Detected! TX-RX conflict between node 7 and node 5
Cannot schedule stream (4, 7) on timeslot 2
Checking stream (4, 7) on timeslot 3
Conflict Detected! src or dst node are busy on timeslot 3
Conflict Detected! TX-RX conflict between node 4 and node 7
Conflict Detected! TX-RX conflict between node 7 and node 2
Cannot schedule stream (4, 7) on timeslot 3
Checking stream (4, 7) on timeslot 4
Conflict Detected! src or dst node are busy on timeslot 4
Cannot schedule stream (4, 7) on timeslot 4
Checking stream (4, 7) on timeslot 5
Conflict Detected! src or dst node are busy on timeslot 5
Conflict Detected! TX-RX conflict between node 4 and node 5
Conflict Detected! TX-RX conflict between node 7 and node 4
Cannot schedule stream (4, 7) on timeslot 5
Checking stream (4, 7) on timeslot 6
Conflict Detected! TX-RX conflict between node 7 and node 5
Cannot schedule stream (4, 7) on timeslot 6
Checking stream (4, 7) on timeslot 7
Scheduled stream (4, 7) on timeslot 7
```

```
Checking stream (7, 0) on timeslot 8
Scheduled stream (7, 0) on timeslot 8

Resulting schedule
Time   Src, Dst
 0:     3    0
 0:     6    8
 1:     8    5
 2:     5    0
 2:     6    2
 3:     2    7
 4:     7    0
 5:     4    5
 6:     5    0
 7:     4    7
 8:     7    0
```

Chapter 8

Schedule distribution

Until now we thought of the **schedule** as a **list of schedule elements**, one for each timeslot, with every schedule element containing a timeslot, an action (TX,RX) and which node has to do it. From now on we will call this schedule encoding **explicit schedule**:

`explicitScheduleElement` $= (timeslot, action, node)$

When thinking about **distributing the schedule** from the master node to the other nodes of the network, we realized that the explicit schedule was really large, since it scales **linearly with the number of timeslots**, which can become big very quickly, especially for streams with *coprime* periods.

For example in case we have a first stream with period 2 and a second stream with period 5, the schedule size in periods would be $lcm(2, 5) = 10$, which in timeslots is $10 * $`tile_size`, assuming a `tile_size` of 16, the number of `explicitScheduleElement`s to send on the network would be 160.

An option could be **compressing the schedule** to reduce the size of the elements without an action, but this measure would not be effective for a heavily loaded network (few timeslots without action).

8.1 Implicit schedule idea

The information saved in the explicit schedule is **heavily correlated**, in particular all the schedule elements of a stream are repeating at distance equal to the stream period, for the complete lenght of the explicit schedule, and this is true for all the streams in the schedule.

Knowing the distance between a particular schedule element and the corresponding schedule element of the next period isn't enough to reconstruct the explicit schedule. The other information we need is the slot in which the first schedule element of that stream is placed, we will call this value **offset**.

8.1.1 Implicit schedule element

An explicit schedule can be obtained without uncertainties from a more compact form that we will call **implicit schedule**, which elements are composed by a **node**, an **action**, the **period** and the **offset** of the first transmission.

$$\texttt{implicitScheduleElement} = (node, action, period, offset)$$

We just defined the schedule element **offset**, while the stream **period** was introduced in section 5.2.2.

8.1.2 Advantages of the implicit schedule

The implicit schedule format has the clear advantage of using less memory space than its explicit counterpart, in fact an implicit schedule scales in size **linearly with the number of streams**, while the explicit schedule scales linearly with the number of timeslots in the schedule, which is generally much bigger than the number of streams. In fact the implicit schedule is much smaller and easy to send over the network. Given this advantage, we decided to employ the implicit schedule right from its initial computation in the **scheduler** and during the **schedule distribution**.

However the **dataphase** requires an explicit schedule to operate more efficiently, since it allows to perform a lookup on the current timeslot to get the corresponding action. The explicit schedule is converted from its implicit counterpart on the node itself, after the schedule distribution; this allows an important optimization: when the explicit schedule is derived, every node extracts only the actions in which he is involved, resulting in a simpler explicit schedule containing only streams related to a given node.

The conversion from implicit schedule to explicit will be explained in section. 8.3

8.2 Schedule packet format

Here we explain the format used to **distribute the implicit schedule** from the master node to all the other nodes of the network.

The **schedule distribution** happens in the downlink phase of TDMH (see section 5.1.3), in this phase the master creates a **schedule packet** by encoding the newest implicit schedule available and sends it to its neighbors, who retransmit the packet after a predefined retransmission time and keep the packet content to build a local version of the schedule that is being distributed. The schedule packet is distributed to all the nodes in the network.

The **schedule packet** is composed of the following:

- a `scheduleHeader` containing information on the current packet
- one or more `scheduleElement`, corresponding to an element of the **implicit schedule**.

The **scheduleHeader** *class* contains the following data:

- `totalPacket`: number of schedule packets needed to distribute the schedule
- `currentPacket`: current packet of the total
- `scheduleID`: unique ID of the schedule distributed
- `activationTile`: tile number in which the schedule becomes active
- `scheduleTiles`: length of explicit schedule in tiles
- `repetition`: current repetition of the schedule distribution

The **scheduleElement** *class* contains the following data:

- `src`: source node of the stream
- `dst`: destination node of the stream
- `srcPort`: source port of the stream
- `dstPort`: destination port of the stream
- `tx`: node transmitting this transmission
- `rx`: node receiving this transmission
- `redundancy`: redundancy level
- `period`: period of the stream (in tiles)
- `payloadSize`: size of the data to send (in packets)
- `direction`: used to open bidirectional streams
- `offset`: offset to calculate the explicit schedule

8.3 Dataphase schedule

Once a *dynamic node* of the network **receives all the schedule packets** forming a schedule, he can easily reconstruct the content of the most recent implicit schedule by unpacking the `scheduleElement`s contained in the `schedulePacket`, since they have a direct correspondence with the implicit schedule elements. The node can also get some metadata about the schedule from the `scheduleHeader` for example the schedule ID, size, and **activation tile**.

8.3.1 Schedule activation

All the nodes in the network keep track of the current tile number from the start of TDMH. When the current tile becomes equal to the activation tile of the received schedule, the implicit schedule can be converted to an explicit schedule, and the last one can be applied in the dataphase for playback.

8.3.2 Dataphase schedule format

The Dataphase employs an explicit schedule, but its format is different from the explicit schedule we mentioned before. In fact the Dataphase does not need a list of generic `explicitScheduleElement`, but needs a vector of elements containing the operation that the specific node needs to perform on each of the timeslots.

`dataphaseScheduleElement` $= (timeslot, dataphaseAction)$

The **dataphaseAction** represents one of the five operations that the Dataphase can perform:

- **Sleep**: save power when no other action is needed
- **Send from stream**: send packet from the application level
- **Receive to stream**: receive packet to the application level

- **Send from buffer**: send packet from the packet buffer
- **Receive to buffer**: receive packet to the packet buffer

Note that every node has a buffer able to store **one packet**, used for retransmit packets for multi-hop transmission.

It is an implicit constraint of the scheduler that a node needs to store only one packet at a time, this keeps the multi-hop transmissions in order, the memory use of the nodes low and avoids **buffering** of packets in the network that will worsen the overall latency.

8.3.3 Conversion algorithm

the conversion from implicit schedule to Dataphase schedule is done with the following algorithm in C++, taken from the C++ implementation (see chapter 10):

```
std::vector<DataphaseScheduleElement>
 ScheduleDownlinkPhase::expandSchedule(
                          unsigned char nodeID) {
    // New explicitSchedule to return
    std::vector<DataphaseScheduleElement> result;
    // Resize new explicitSchedule
    // and fill with default value (sleep)
    auto slotsInTile = ctx.getSlotsInTileCount();
    auto scheduleSlots = header.getScheduleTiles() *
                         slotsInTile;
    result.resize(scheduleSlots,
                  DataphaseScheduleElement());
    // Scan implicit schedule and derive
    // the corresponding action
    for(auto e : schedule) {
        // Period is normally expressed in tiles,
        // get period in slots
```

```
        auto periodSlots = toInt(e.getPeriod()) *
                           slotsInTile;
        Action action = Action::SLEEP;
        // Send from stream case
        if(e.getSrc() == nodeID && e.getTx() == nodeID)
            action = Action::SENDSTREAM;
        // Receive to stream case
        if(e.getDst() == nodeID && e.getRx() == nodeID)
            action = Action::RECVSTREAM;
        // Send from buffer case
        // (send saved multi-hop packet)
        if(e.getSrc() != nodeID && e.getTx() == nodeID)
            action = Action::SENDBUFFER;
        // Receive to buffer case
        // (receive and save multi-hop packet)
        if(e.getDst() != nodeID && e.getRx() == nodeID)
            action = Action::RECVBUFFER;
        // Apply action to the right slots
        for(auto slot = e.getOffset();
            slot < scheduleSlots;
            slot += periodSlots) {
            result[slot] = DataphaseScheduleElement(
                           action,
                           e.getStreamInfo());
        }
    }
    return result;
}
```

Chapter 9

Simulation

The codebase of TDMH runs on the MIOSIX [21] operating system, and it's written in C++. Before working on this project I was not very proficient in C++, and I hesitated at the idea of implementing and testing the algorithms directly in a language I was not confident with.

Because of this I decided together with my supervisor to implement a **proof-of-concept** simulation of the scheduling and routing algorithms explained in the previous chapters in a self-contained program written in Python, a language I was already proficient at.

9.1 Python simulation

The results of this simulation written in Python were **very positive**, in fact I was able to test the correctness of the algorithms in a small program that given the basic inputs of the scheduling and routing problem:

- A network **topology graph**
- A list of **stream requests**
- The number of available **time slots**

Could run the algorithms and produce as output an **explicit schedule** composed by:

- A list of tuples containing **(time slot, node, activity)**

Another advantage of the Python simulation was that we had a working implementation of the algorithms to be implemented in TDMH, so this made the implementation in C++ easier, but not without issues at all.

In fact the main difference between the Python simulation and the actual implementation is that the simulation worked with **explicit schedules**, because it was written before we decided to adopt the **implicit schedule**. The adoption of the implicit scheduler brought its own difficulties mainly in **finding conflicts** between two implicit form streams, this has to be done by just using their period and offsets. For more details see section 10.4.3 of the implementation chapter.

Below you can find the execution output of the Python simulation running with the input data (topology, stream requests and number of time slots) taken from the RTSS article "TDMH-MAC: Real-Time and Multi-hop in the Same Wireless MAC" [22]

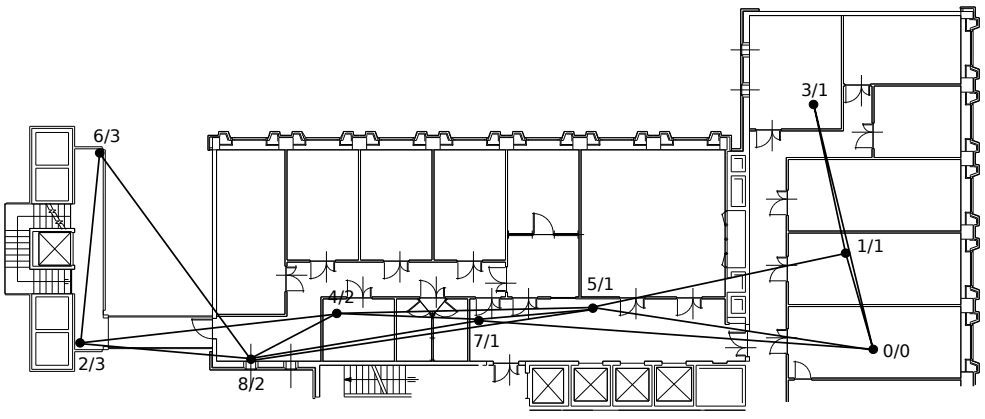

Figure 9.1: Network topology from RTSS paper

Below is the input data for the simulation:

```
topology    =   [(0, 1), (0, 3), (0, 5), (0, 7),
                 (1, 3), (1, 5), (1, 7),
                 (2, 4), (2, 6), (2, 7), (2, 8),
                 (3, 5),
                 (4, 5), (4, 6), (4, 7), (4, 8),
                 (5, 7), (5, 8),
                 (6, 8),
                 (7, 8)]
req_streams = [(3, 0),
                 (6, 0),
                 (4, 0)]
time_slots = 10
```

Python simulation output

```
$ ./scheduler_sym.py run
Primary Path (BFS): [6, 8, 5, 0]
Routing (6, 0) as [(6, 8), (8, 5), (5, 0)]
Primary path length: 4
Searching secondary path of max length: 4+2= 6
DFS solutions: [[6, 8, 2, 4, 5, 0], [6, 8, 2, 4, 7, 0],
[6, 8, 2, 7, 0], [6, 8, 4, 2, 7, 0], [6, 8, 4, 5, 0],
[6, 8, 5, 0], [6, 8, 7, 0], [6, 2, 8, 4, 5, 0],
[6, 2, 8, 5, 0], [6, 2, 7, 0]]
Middle nodes [8, 5]
Found indipendent path
Secondary Path (limited-DFS): [6, 2, 7, 0]
Routing (6, 0) as [(6, 2), (2, 7), (7, 0)]
Primary Path (BFS): [4, 5, 0]
Routing (4, 0) as [(4, 5), (5, 0)]
```

```
Primary path length: 3
Searching secondary path of max length: 3+2= 5
DFS solutions: [[4, 2, 8, 5, 0], [4, 2, 7, 0], [4, 5, 0],
[4, 5, 1, 0], [4, 7, 0]]
Middle nodes [5]
Found indipendent path
Secondary Path (limited-DFS): [4, 7, 0]
Routing (4, 0) as [(4, 7), (7, 0)]
Final stream list: [[(3, 0)], [(6, 8), (8, 5), (5, 0)],
[(6, 2), (2, 7), (7, 0)], [(4, 5), (5, 0)], [(4, 7), (7, 0)]]

Resulting schedule
Time   Src, Dst
 0:     3    0
 0:     6    8
 1:     8    5
 2:     5    0
 2:     6    2
 3:     2    7
 4:     7    0
 5:     4    5
 6:     5    0
 7:     4    7
 8:     7    0
```

In figure 9.2 there is a graphical representation of the schedule generated with the python simulator. Looking at the schedule, we note that it seems reasonable and efficient, and it has no conflicts.

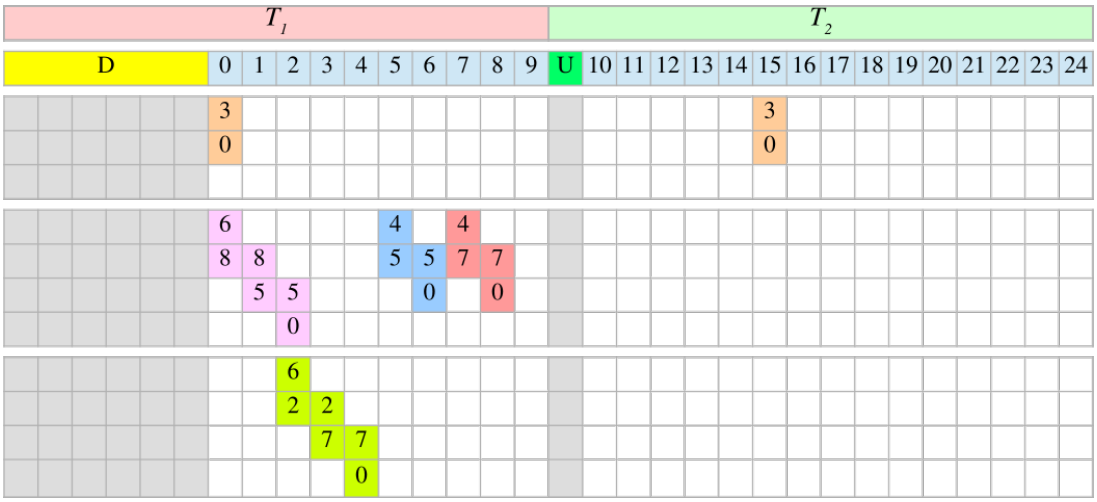

Figure 9.2: Schedule generated with the python simulator

9.2 OMNeT++ simulation

After the Python simulation was ready and tested, we began implenting the algorithms in the actual TDMH codebase.

TDMH is designed to be run on wireless nodes running the Miosix [21] Operating System, in particular on the WandStem [23] nodes, because of the **hardware timestamping** needed to make **FLOPSYNC-2** work.

Implementing and testing the scheduling algorithm directly on real hardware has its own set of problems, in particular the difficulty in troubleshooting and the time required to make tests with more than one node.

For this reason Paolo Polidori, the previous student doing its master thesis [24] on TDMH, developed an interface to run TDMH in OMNeT++ [25], a well known **network simulator** based on *discrete event simulation*.

The **integration** was done by developing an **interface** compatible with MIOSIX APIs for radio, timers and logging, that will call OMNeT++ corresponding functions instead of the MIOSIX kernel ones. Doing so we have a unique codebase that can run both in simulation and on real hardware.

Running TDMH in simulation allowed developing and testing the new components of the code in a simulation environment, with the ease of running the code and trying different topologies that comes with it.

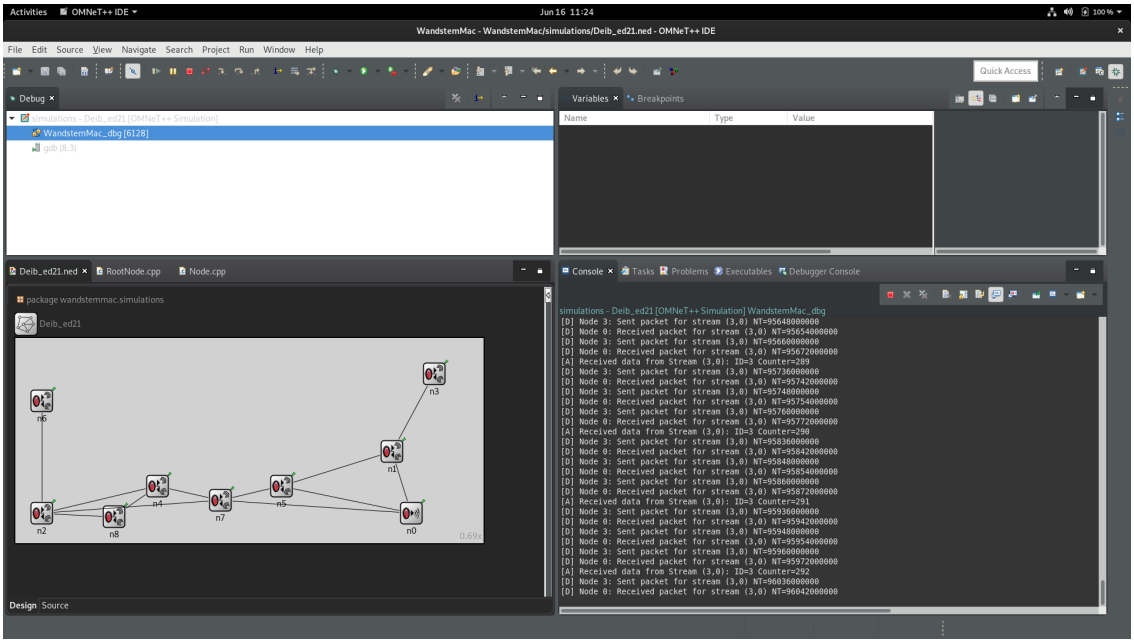

Figure 9.3: Debug view of the OMNeT++ simulator

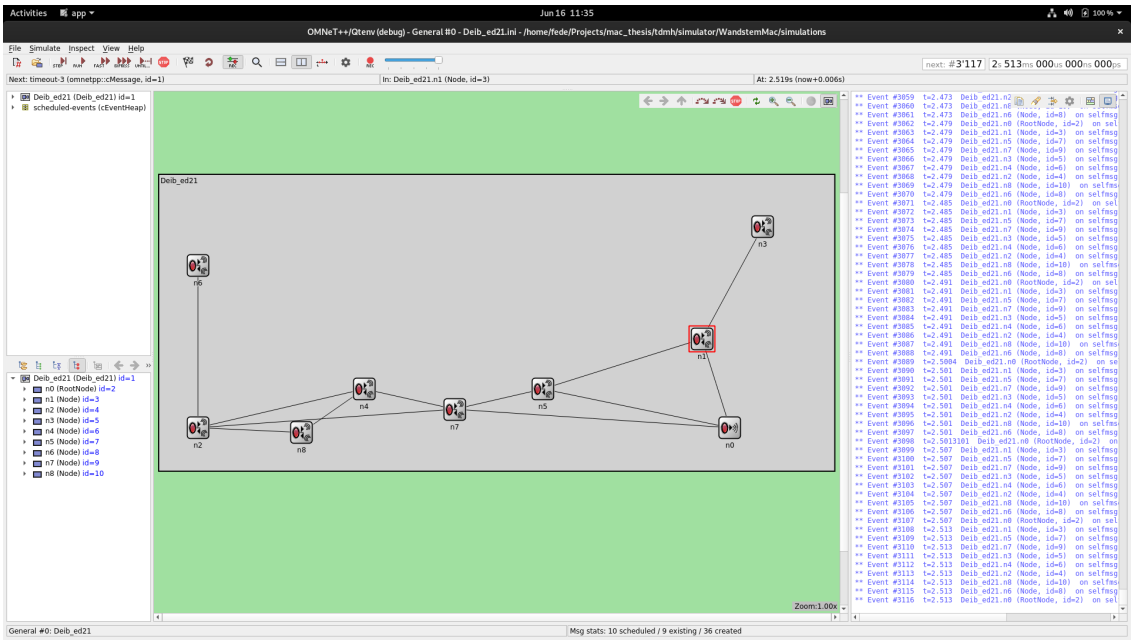

Figure 9.4: Simulation view of the OMNeT++ simulator

Chapter 10

Implementation

In this chapter we will explain some details about the **implementation** of the Scheduler and Router algorithms presented in the previous chapters into the **existing C++ codebase of TDMH**. We will also present the design and implementation of the other components of the network stack that have been built in the process of making the TDMH stack work in **simulation and on real hardware**.

10.1 Object Oriented codebase

The codebase on which this thesis is built is the work of post. Doc. professor Federico Terraneo (author of MIOSIX [21]) and previous Ms. thesis student Paolo Polidori. The TDMH source code is released under the Open Source license GPLv2 [26] and it is currently hosted on GitHub[27].

This codebase has a strongly **Object Oriented** style, which means that every self contained portion of the code is kept in a separate class, and there is a heavy use of **inheritance** within these classes.

The most notable example of inheritance is that every class that should have a different behaviour in the master node with respect to dynamic nodes is structured as a generic parent class implementing the common methods between master and dynamic nodes, and two different classes with prefix 'Master' and 'Dynamic' that implement the behaviour specific to the two type of nodes.
You can see an example of this pattern in figure 10.1.

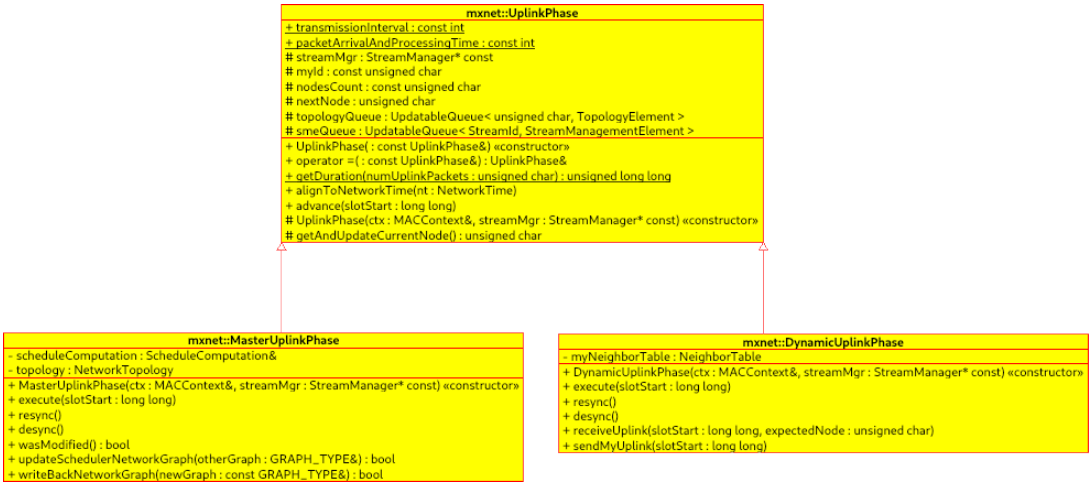

Figure 10.1: UML diagram showing polymorphism in the Uplink class

10.2 Overview of TDMH modules

In figure 10.2 we can see that a relevant part of the current TDMH functionality has been implemented as part of this thesis work.
In the following sections you can find the general design and some implementation details of the TDMH modules on which this work had its focus.

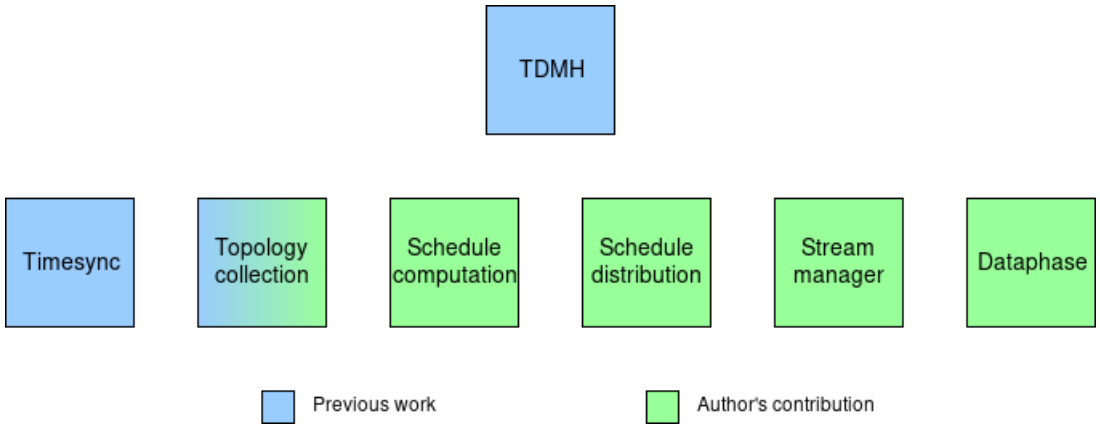

Figure 10.2: Highlight of author's contribution in TDMH codebase

10.3 Topology Collection

The **Topology Collection** module is the one responsible for collecting information from the *Dynamic nodes* of the network, towards the *Master node*. The information collected is mainly the map of the network, but also the requests for opening a Stream or a Server.

The Topology Collection works during the Uplink phase (see section 5.1.6) and is essential for the correct functioning of the other modules, because it provides the base information on which the schedule is computed, and is also responsible for the stream and server commands.

10.4 Schedule computation

The Scheduler and Router algorithms are contained respectively in the `ScheduleComputation` and `Router` classes, in these classes are implemented the algorithms explained in chapter 7. These algorithms perform the translation from the Stream opening requests received by the nodes of the network to a TDMA schedule, ready to be distributed and finally executed in the Dataphase. See figure 10.3 for a graphic representation of this process.

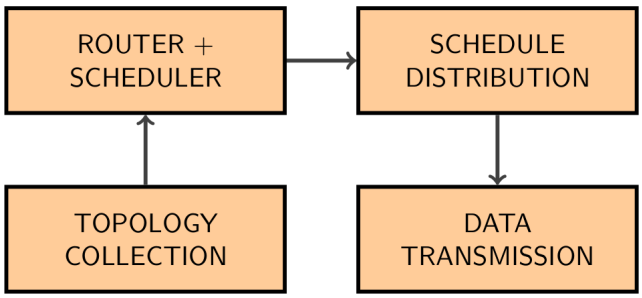

Figure 10.3: TDMH modules related to streams

The scheduling and routing algorithms are computationally intensive because they employ multiple graph searches. Since the timing is crucial in the TDMH stack, the execution of these algorithms might delay the network phases. For this reason, all the computation for the scheduling and routing is done in an **offline** fashion, by running the scheduler and router code on a separate thread, that is opportunely synchronized with the network stack thread.

10.4.1 Thread model

The synchronization between the *scheduler thread* and the *network stack thread* is not trivial. In fact the scheduler thread needs to wait until there are one or more suitable streams to schedule, and this wait should not con-

sume CPU resources.

To fullfill this requirement, we employed a **producer consumer** pattern for **thread synchronization**. This was implemented using a **mutex** with a **condition variable**, the implementation is shown below:

```
void ScheduleComputation::run() {
    ...
    // Mutex lock to access stream list.
    {
#ifdef _MIOSIX
    miosix::Lock<miosix::Mutex> lck(sched_mutex);
#else
    std::unique_lock<std::mutex> lck(sched_mutex);
#endif
    // Wait until topology or stream list changes
    while(!topology_ctx.wasModified() &&
          !stream_mgmt.wasModified()) {
        sched_cv.wait(lck);
        }
    // Begin scheduling
    ...
}
```

The `notify` for this condition variable is called in the main loop of TDMH, this is done to activate the scheduler at each TDMH cycle. The function that sends the `notify` is called `beginScheduling`

```
void MACContext::run() {
    ...
    for(running = true; running; ) {
    ...
    else {
        if(!scheduleDistribution->distributingSchedule())
            beginScheduling();
        scheduleDistribution->run(currentNextDeadline);
    }
    ...
}
```

10.4.2 Scheduler algorithm implementation

The `ScheduleComputation` class implements the **greedy scheduler algorithm** presented in chapter 7.2.1, that we report here for convenience:

Greedy scheduler algorithm:

- **Sort the streams** to have the highest-period streams first
- **Iterates over the streams** and tries to allocate them to the first available timeslot
- **Checks for conflicts** and makes sure that the constraints are satisfied
- If there are conflicts, the scheduler **tries the next available timeslot**
- If on the next timeslot there are no conflicts, the **stream is scheduled**
- If there are no more timeslots available, the **stream is not scheduled**.

The TDMH implementation of this algorithm differs from the **Python simulation** found in chapter 9, because in the TDMH implementation we employed the concept of **implicit schedule** (see section 8.1).

The main difference that arises from the adoption of the implicit schedule, is that for every stream, we **iterate over the available offsets**, and not over the timeslots as we did on the Python simulator.

An important step of the greedy scheduler algorithm is finding potential conflicts between the stream currently being scheduled and the already scheduled streams.

The **conflict checks** that we perform in the scheduler algorithm are the following:

- **Unicity check**: A node in the network cannot transmit and receive in the same timeslot
- **Interference check**: The current transmission must not interfere with other transmissions happening in the same timeslot (see section 5.2.5)

The complication introduced by the adoption of the implicit schedule arises from the fact that we do not know directly the timeslot used by every stream, in fact every stream is characterized only by a **period** and an **offset**.

10.4.3 Finding conflicts in implicit schedule

This problem was solved by leveraging the correlation between the **period** and **offset** of a stream, and the **dataslots** resulting from these.

In fact we can derive all the dataslots occupied by a stream with the following formula: given `tile_size` the lenght in dataslots of a single Period and `per` a value ranging from 1 to `Period`

$$Dataslot = offset + per * \text{tile\_size}$$

From this we know that we can simply iterate over the offsets and check the conflicts only with the streams have the calculated **dataslots in common**.

The function that calculates if two streams have dataslots in common is called `checkSlotConflict` and has the following implementation.

```
bool ScheduleComputation::checkSlotConflict(...) {
    // Calculate slots used by the two transmissions
    // and see if there is at least a common value
    unsigned period_a = toInt(newtransm.getPeriod());
    unsigned period_b = toInt(oldtransm.getPeriod());
    unsigned periodslots_a = period_a * tile_size;
    unsigned periodslots_b = period_b * tile_size;
    unsigned schedule_slots = lcm(period_a, period_b) *
        tile_size;

    for(unsigned slot_a=offset_a;
        slot_a < schedule_slots;
        slot_a += periodslots_a) {
      for(unsigned slot_b=oldtransm.getOffset();
          slot_b < schedule_slots;
          slot_b += periodslots_b) {
            if(slot_a == slot_b)
                return true;
        }
    }
    return false;
}
```

10.5 Schedule distribution

The **schedule distribution** module's goal is to distribute to the entire network the schedule computed by the master node, and to make sure that the new schedule is applied in the whole network at the same time.

10.5.1 Finite state machine model

We decided to model the schedule distribution module with the **finite state machine** paradigm, the advantages of this type of model is better clarity and better solidity. The behaviour of the schedule distribution module can be described with two FSMs (Finite state machines) for the master node and two FSMs for the dynamic nodes, for a total of four FSMs.

- master schedule distribution
- master schedule application
- dynamic schedule distribution
- dynamic schedule application

We will show below the model of the four FSM, together with a description

Master schedule distribution

The **master schedule distribution** FSM is in charge of retrieving a new schedule and computing the tile number at which the new schedule will be activated (`activationTile`), after this it will send the **Schedule Packets** for a predefined number of repetitions, in this case three.
You can find a representation of this FSM in figure 10.4.

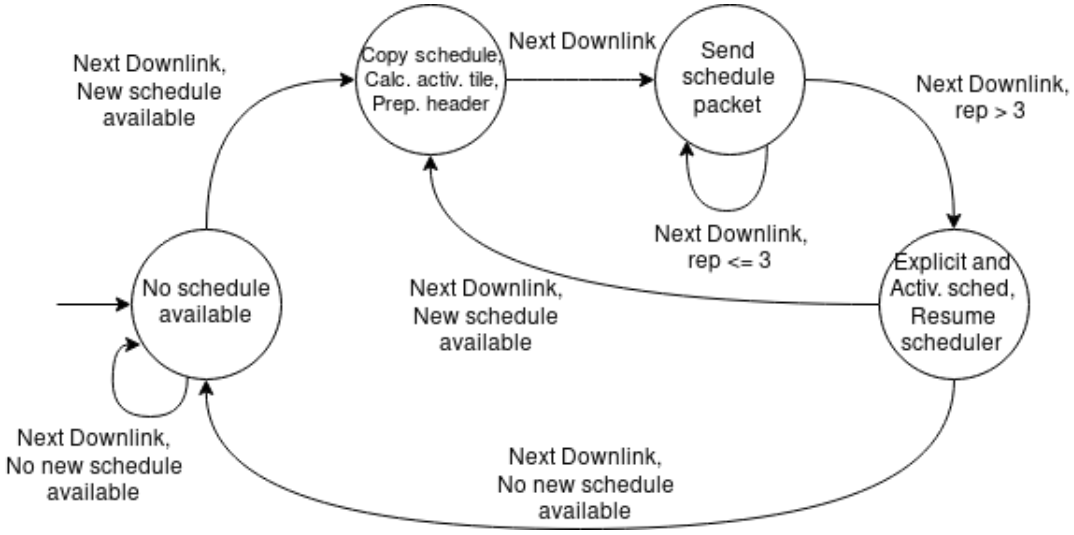

Figure 10.4: FSM model of master schedule distribution

Master schedule application

The **master schedule application**, when a new schedule is present waits until it has been distributed and then compares the current tile number, with the saved `activationTile`. When these two numbers are equal, the schedule is applied. **A given schedule can be applied only after it has been distributed**, otherwise the nodes that still haven't received the new schedule cannot apply it, thus leading to a non uniformity of the schedule over the network. This requirement is handled by employing a boolean flag called `distributing`.

You can find a representation of the master schedule application FSM in figure 10.5.

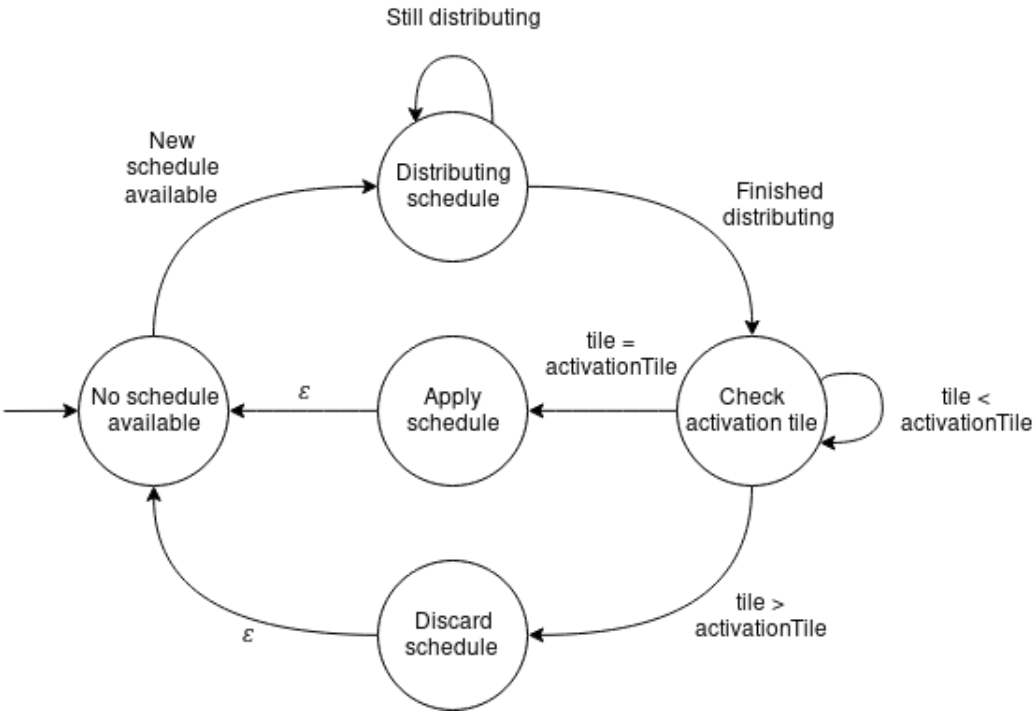

Figure 10.5: FSM model of master schedule application

Dynamic schedule distribution

The **dynamic schedule distribution** FSM runs on the dynamic nodes and listens for incoming Schedule Packets, when all the packets composing a schedule are received, the complete schedule is reconstructed.

You can find a representation of the dynamic schedule distribution FSM in figure 10.6.

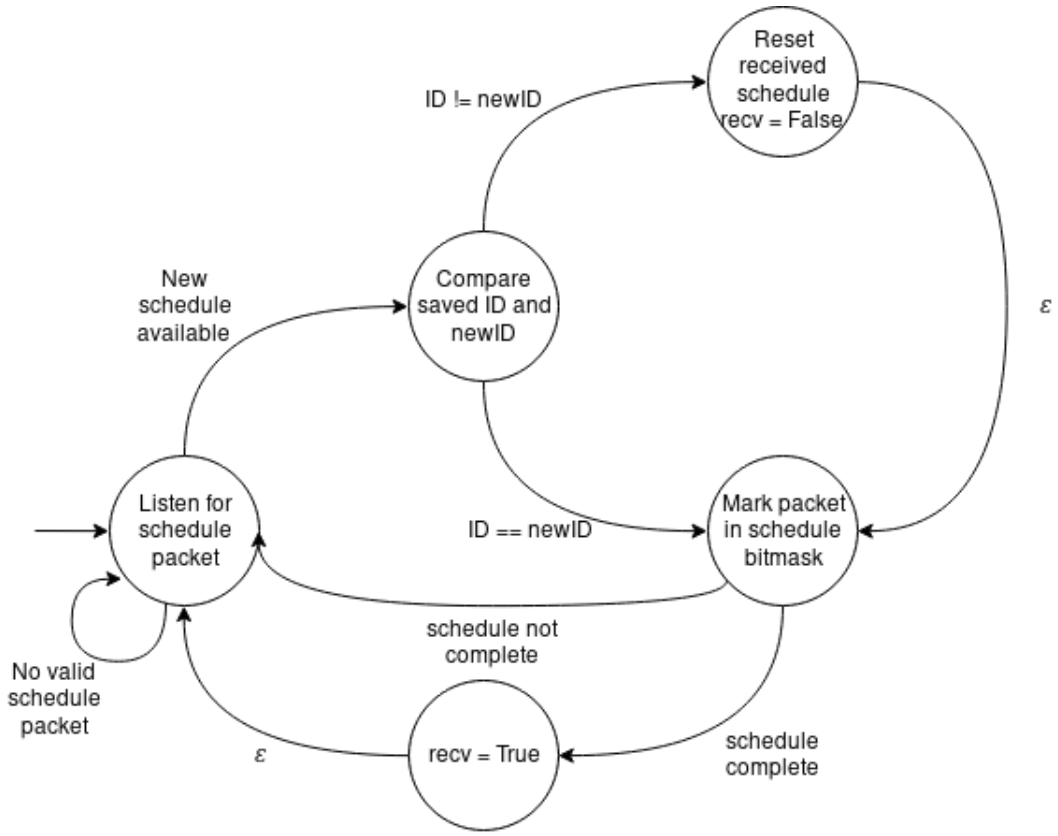

Figure 10.6: FSM model of dynamic schedule distribution

Dynamic schedule application

The **dynamic schedule application** FSM runs on the dynamic nodes and compares the current tile number with the schedule `activationTile` received with the schedule, applying the schedule if the two tile numbers correspond.

You can find a representation of the dynamic schedule application FSM in figure 10.7.

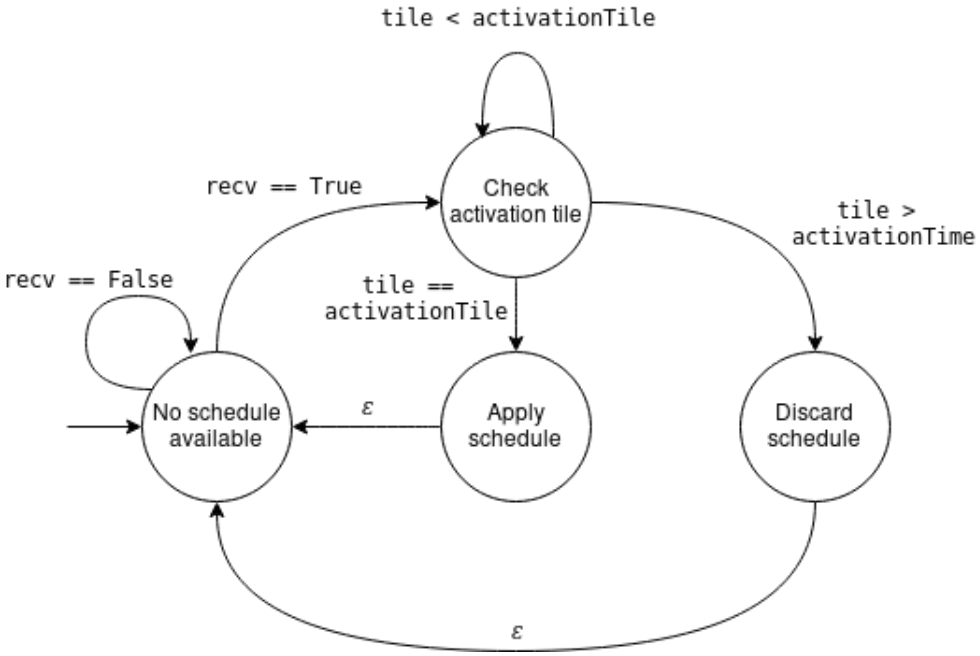

Figure 10.7: FSM model of dynamic schedule application

10.6 Dataphase

The purpose of the Dataphase module is to playback the latest available schedule, executing the operations specified for every timeslot of the slotframe. The Dataphase module is essential to the functioning of the Streams, as it's the one sending or receiving the actual application data from the network. The Dataphase uses a schedule in its **explicit form**, this schedule is converted after its distribution as explained in section 8.3.

As we have seen before, an explicit schedule defines an action for every timeslot. The explicit schedule is specific to a given node, since it contains only actions related to that node.

10.6.1 Schedule actions

The actions that the Dataphase can perform based on its explicit schedule are five, here is the prototype of the methods implementing the various action, taken from `dataphase.h`

```
  /* Five possible actions, as described by
  the explicit schedule */
void sleep(long long slotStart);
void sendFromStream(long long slotStart, StreamId id);
void receiveToStream(long long slotStart, StreamId id);
void sendFromBuffer(long long slotStart);
void receiveToBuffer(long long slotStart);
```

Sleep

The **sleep** action puts the microcontroller in a low power state until the start of the next timeslot.

Send from stream

The **send from stream** action checks if a packet is available from the upper layer (stream) and sends it on the network, if it's not available it sleeps until the start of the next timeslot.

Receive to stream

The **receive to stream** action receives a packet from the network, and if it is valid, it is forwarded to the upper layer (stream).

Send from buffer

The **send from buffer** action checks if a valid packet is present in the buffer and sends it on the network, if it's not available it sleeps until the start of the next timeslot.

Receive to buffer

The **receive to buffer** action receives a packet from the network, and if it is valid, it is stored on the buffer and the buffer is marked as valid.

10.7 Stream manager

The Stream Manager is the TDMH module responsible for managing the Streams and Servers. It contains informations about all the Streams and Servers regarding the current node. The Stream Manager presents an interface to the application level, with all the available functions to operate on the Streams. The Stream Manager is also in charge of modifying the Streams according to the information it receives from the network, in the form of a new schedule or a list of Info elements.

The Streams and Servers themselves are implemented with a finite state machine model. The Streams and Servers current status is kept respectively in the instances of the `Stream` and `Server` classes inside the Stream Manager. The Stream Manager is the one providing events that change the status of the finite state machines, these events can either come from the user (Stream API) or from the Network (Schedule, Info element or disconnection).

10.7.1 Streams

You can find a representation of the stream state machine in the dynamic nodes in figure 10.8, and the related state machine in the master node in figure 10.9.

10.7.2 Servers

You can find a representation of the server state machine in the dynamic nodes in figure 10.10, and the related state machine in the master node in figure 10.11.

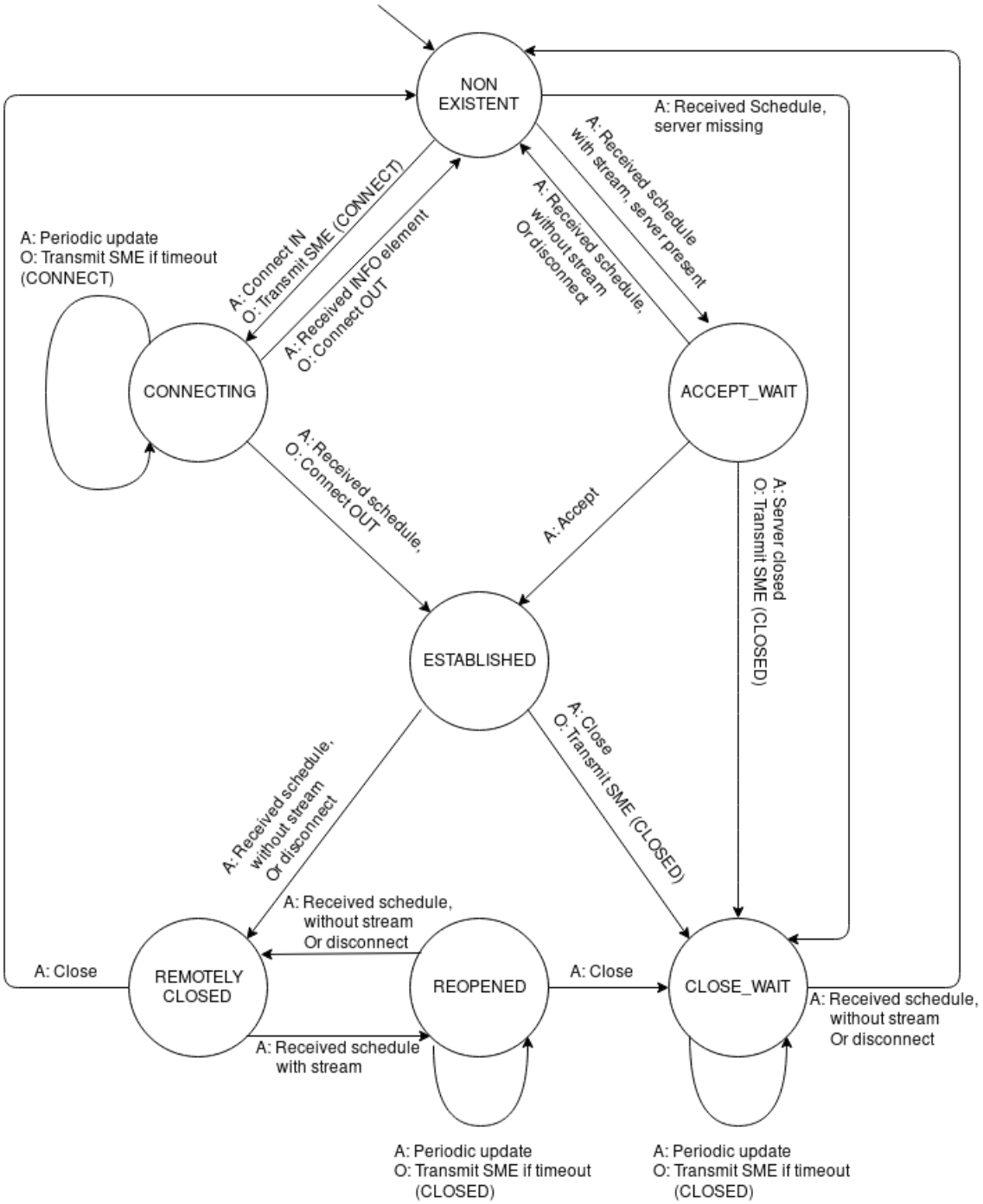

Figure 10.8: FSM model of streams in dynamic nodes

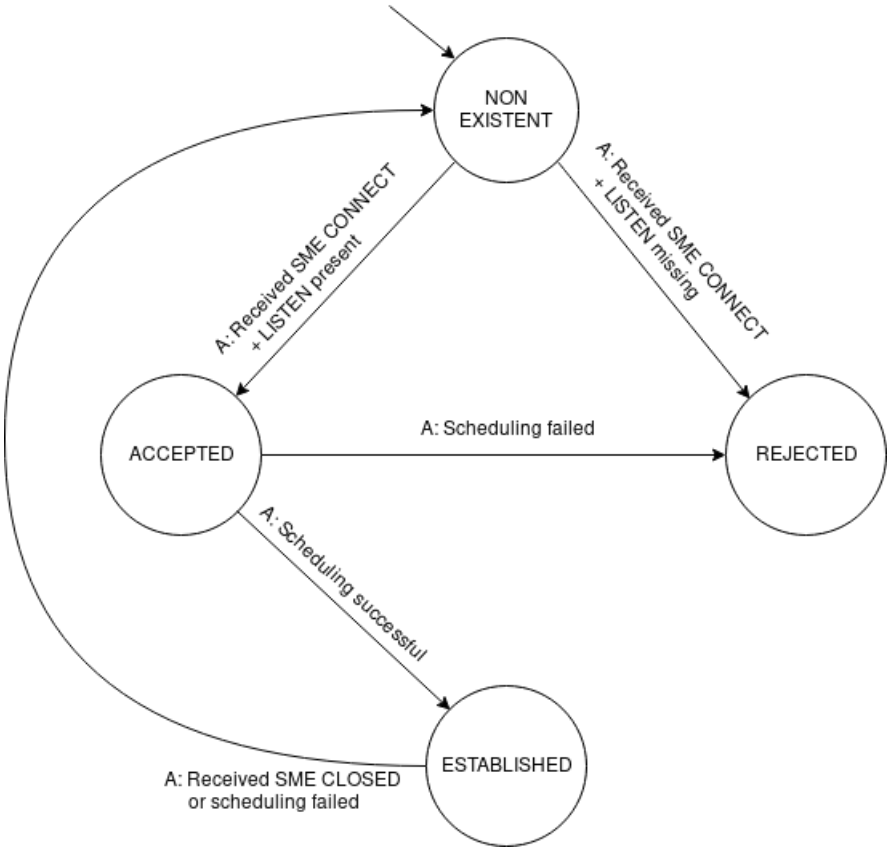

Figure 10.9: FSM model of streams in master nodes

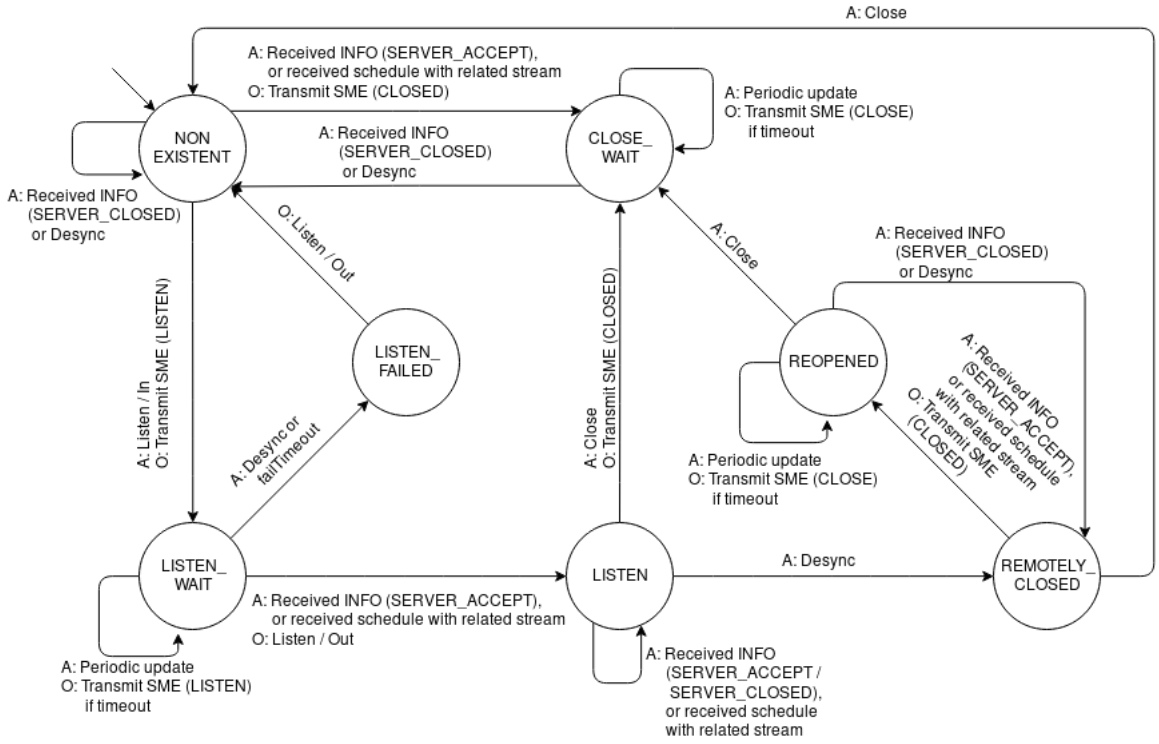

Figure 10.10: FSM model of servers in dynamic nodes

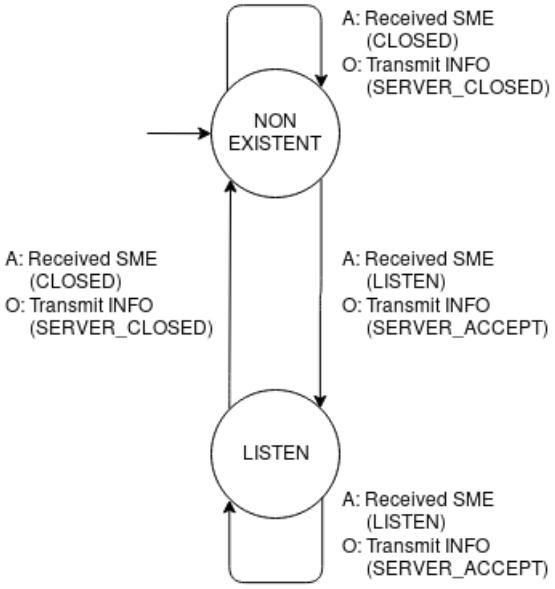

Figure 10.11: FSM model of servers in master nodes

Chapter 11

Experiments

The goal of the experiments presented in this chapter is to validate the general functionalities of TDMH and in particular to prove the effectiveness of the **redundancy measures** in the transmission of data.

We propose **three experiments**, consisting in a simulation and several real-world experiments:

- 1: validation of the stream redundancy in **simulation**
- 2: validation of the stream redundancy using a **wired setup**
- 3: validation of the stream redundancy with nodes **placed over a building floor**

11.1 First experiment: Simulation

11.1.1 Topology

The **network topology** chosen for the first experiment is the so called **diamond topology**, consisting in four nodes connected in a rhombus geometry. See figure 11.2

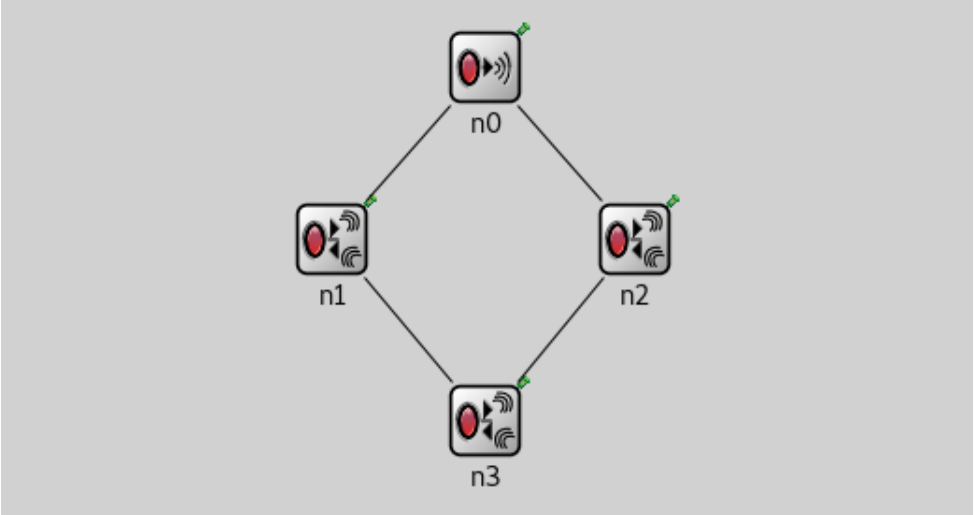

Figure 11.1: Diamond topology shown in OMNeT++

This topology is simple but allows to take advantage of the **spatial redundancy**, thanks to the two paths without common intermediate nodes between nodes 0 and 3: $0 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 3$.

11.1.2 Setup

This experiment is run on the OMNeT++ simulator, the simulator allows to shutdown one of the simulated nodes at a given time, this function is used to simulate the fault.

11.1.3 Test procedure

First run

In the first run a stream is opened from node 3 to node 0, with Period::P1 and without redundancy. The simulated time is 10 minutes, and at 5 minutes from the start we simulate a fault, turning off the intermediate node of the path $0 \rightarrow 3$ (which could be 1 or 2 depending on the schedule).

- The network is started and left running for 10 minutes
- After 5 minutes, node 1 or 2 is shut down
- The network is left running for additional 5 minutes
- Total test time: 10 minutes

Second run

In the second run a stream is opened from node 3 to node 0, with Period::P1 and with **triple spatial redundancy**. The simulated time is 10 minutes, and at 5 minutes from the start we simulate a fault, turning off one of the two intermediate nodes of the path $0 \rightarrow 3$ (node 1 or node 2).

- The network is started and left running for 10 minutes
- After 5 minutes, node 1 or 2 is shut down
- The network is left running for additional 5 minutes
- Total test time: 10 minutes

11.1.4 Goal

First run

The expected result of turning off the node is that there is a **small packet loss**, until the network realizes that the node is not available anymore, at this point the master node should produce a new schedule that avoids the missing node, restoring the normal stream functionality.

Second run

The expected result of turning off the node is that there is **no packet loss**, because thanks to the spatial redundancy, the packet of the stream follow two distinct path, and are not affected by the fault. Also in this case the

master node should realize that one of the nodes is not available anymore, and should produce a new schedule that avoids the missing node.

11.1.5 Results

The results of this experiment are listed in tables 11.1, 11.2,

11.1.6 Conclusions

First run

This test case shows the **self diagnosing** and **self-healing** capabilities of TDMH. In fact when a node of the network becomes unavailable, the master node detects the change in the network topology and produces a new schedule with a new path for the stream, if there is one available.
The result is that even with **no redundancy**, in case a node becomes unreachable we have a small data loss, but the affected stream is soon restored and able again to send packets.

Second run

This test case shows the **effectiveness of the spatial redundancy** in avoiding data loss in case of temporary or permanent failure of a node in the network, this can be seen in practice by the fact that the stream (3,0) with **spatial redundancy** keeps receiving packets even after node 2, that is on one of the possible paths is shut down.

Node ID	Total packets	Lost packets	Reliability
3	5770	60	98.96%

Table 11.1: First experiment results with redundancy disabled

Node ID	Total packets	Lost packets	Reliability
3	5770	0	100%

Table 11.2: First experiment results with triple redundancy

11.2 Second experiment: wired setup

This experiments is meant to test the two situations presented in the first experiment, but on real hardware, with a wired setup 11.2.2, that allows to have a constant RSSI and to avoid external interferences.

11.2.1 Topology

For the second experiment we chose the same **network topology** we used for the first experiment (diamond topology).

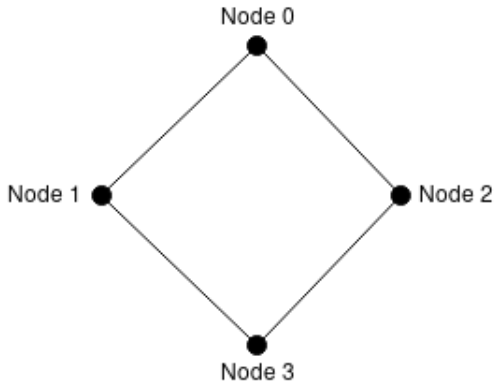

Figure 11.2: Diamond topology of the wired setup

11.2.2 Setup

The **wired setup** consists in connecting the WandStem [23] nodes in a wired RF circuit. This has the great advantage of eliminating the problems of low RSSI, interferences with other protocols or noise in the 2.4GHz spectrum.

The RF circuit is composed of the following elements:

- n.8 Coaxial cables
- n.4 3-way RF splitters
- n.4 30dB attenuators

In figure 11.3 you can see a photo of the final setup.

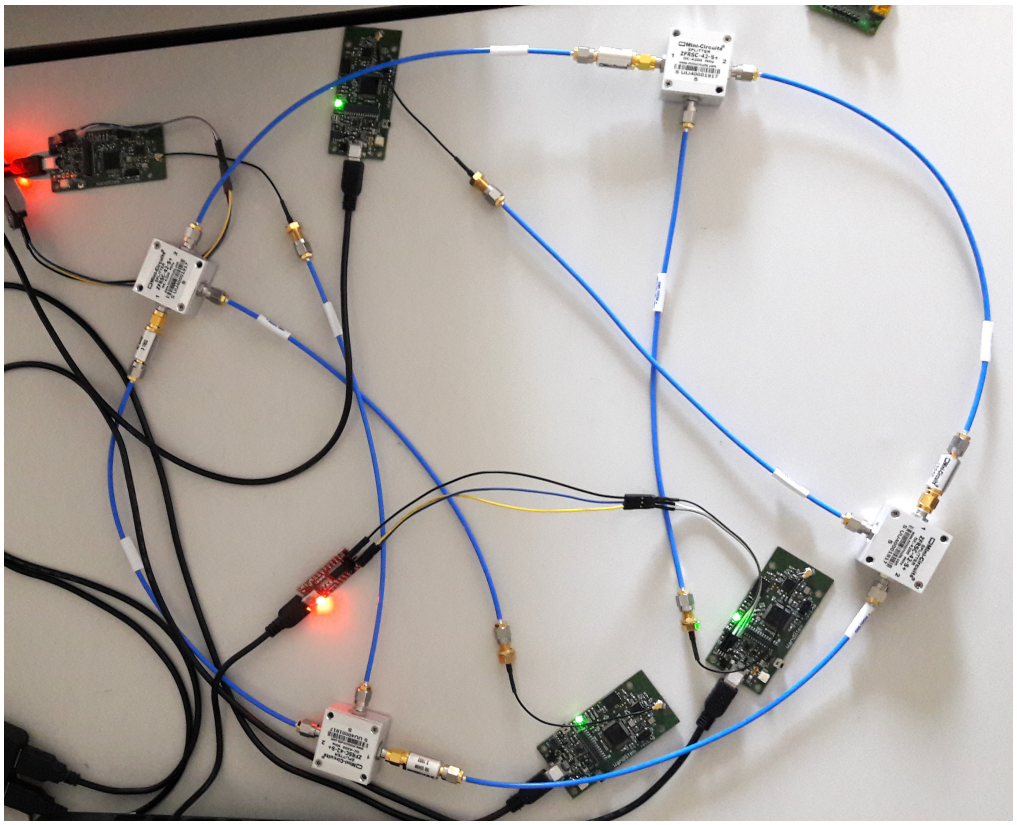

Figure 11.3: Photo of the wired setup

11.2.3 Test procedure

First run

In the first run a stream is opened from node 3 to node 0, with Period::P1 and without redundancy. The simulated time is 10 minutes, and at 5 minutes from the start we simulate a fault, turning off the intermediate node of the path $0 \rightarrow 3$ (which could be 1 or 2 depending on the schedule).

- The network is started and left running for 10 minutes
- After 5 minutes, node 1 or 2 is shut down
- The network is left running for additional 5 minutes
- Total test time: 10 minutes

Second run

In the second run a stream is opened from node 3 to node 0, with Period::P1 and with **triple spatial redundancy**. The simulated time is 10 minutes, and at 5 minutes from the start we simulate a fault, turning off one of the two intermediate nodes of the path $0 \rightarrow 3$ (node 1 or node 2).

- The network is started and left running for 10 minutes
- After 5 minutes, node 1 or 2 is shut down
- The network is left running for additional 5 minutes
- Total test time: 10 minutes

11.2.4 Goal

First run

The expected result of turning off the node is that there is a **small packet loss**, until the network realizes that the node is not available anymore, at this point the master node should produce a new schedule that avoids the missing node, restoring the normal stream functionality.

Second run

The expected result of turning off the node is that there is **no packet loss**, because thanks to the spatial redundancy, the packet of the stream follow two distinct path, and are not affected by the fault. Also in this case the master node realizes that one of the nodes is not available anymore, and it should produce a new schedule that avoids the missing node.

11.2.5 Results

The results of this experiment are listed in tables 11.3, 11.4,

11.2.6 Conclusions

First run

As the omologous simulation experiment in section 11.1, this test case shows the **self diagnosing** and **self-healing** capabilities of TDMH. In fact when a node of the network becomes unavailable, the master node detects the change in the network topology and produces a new schedule with a new path for the stream, if there is one available.
The result is that even with **no redundancy**, in case a node becomes unreachable we have a small data loss, but the affected stream is soon restored and able again to send packets.
This behaviour is confirmed even on **real hardware**.

Node ID	Total packets	Lost packets	Reliability
3	6027	57	99.05%

Table 11.3: Second experiment results with redundancy disabled

Node ID	Total packets	Lost packets	Reliability
3	6023	1	99.98%

Table 11.4: Second experiment results with triple redundancy

Second run

As the omologous simulation experiment in section 11.1, this test case shows the **effectiveness of the spatial redundancy** in avoiding data loss in case of temporary or permanent failure of a node in the network, this can be seen in practice by the fact that the stream (3,0) with **spatial redundancy** keeps receiving packets even after node 2, that is on one of the possible paths is shut down. This behaviour is confirmed even on **real hardware**.

11.3 Third experiment: building floor setup

11.3.1 Topology

For the third experiment the topology was collected by the TDMH network itself, as part of its normal functionality. For this experiment, the WandStem [23] nodes were deployed over the first floor of the Building 21 of Politecnico di Milano, reproducing the node placement seen in the paper presented by Terraneo et al. at the RTSS conference [22]. This choice was done to be able to compare the results to the one presented in the said paper with a previous version of TDMH.
You can see the node placement in figure 11.4.

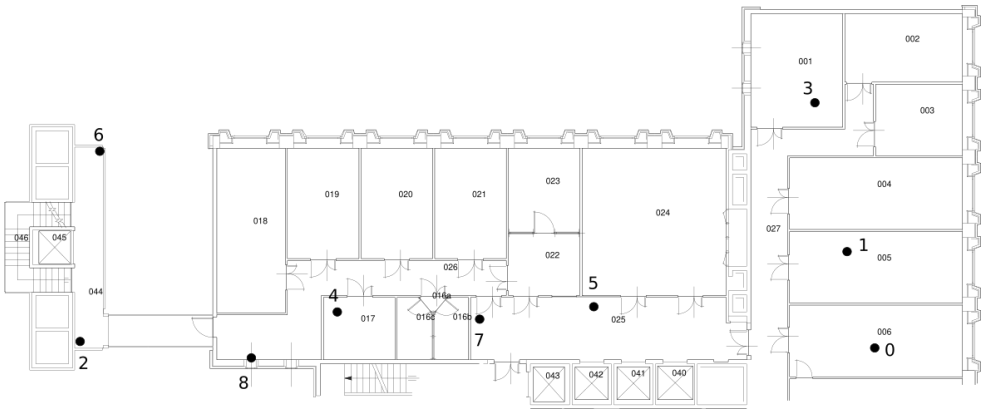

Figure 11.4: Node placement from RTSS paper

11.3.2 Setup

The WandStem nodes were placed in appropriate boxes and powered by a pair of AA alcaline batteries for each node. The master node was powered by the USB port of a computer, and its output logged from a serial connection.

This experiments is meant to test the TDMH general reliability in a real-world setup, employing real hardware and radio communication in a **complex environment**. Another purpose of this test was evaluating the

Figure 11.5: WandStem nodes in their boxes

effectiveness of the **redundancy** settings. The spatial reuse of channels was disabled for this experiment.

11.3.3 Test procedure

The network is composed of **9 nodes** (8 dynamic nodes and 1 master node). All the dynamic nodes (ID:1,8) open a stream to the master node with Period::P10 (corresponding to 1 second interval between packets), and triple spatial redundancy. The master node logs every packet received or missed, and prints the content of the packets.

The experiment is being run with the redundancy set to its highest value (triple spatial). However when analyzing logs, we can extract the reliability corresponding to redundancy disabled or double redundancy, by considering only the first or the first two repetitions of a transmission. This way we can get statistics about the three types of redundancy within the same experiment.

- The experiment is started at 18:00 and left running until the 9:30 of the next day.
- Total test time: 15 hours, 30minutes
- Number of packet sent: ∼55000 per stream

11.3.4 Goal

We expect that the **Redundancy** setting is able to increase the reliability of the streams, at the expense of using more data slots (thus more bandwidth). So we expect to have a stream reliability for every level of redundancy that is higher than the lower level.

11.3.5 Results

The results of this experiment are listed in table 11.5.

11.3.6 Conclusions

From the results of this experiment we can see that the reliability is good since it is generally higher than 95%. We can see that rising the redundancy settings results in strictly higher reliability values, that confirm the effectiveness of this measure.

Node ID	Reliability (single)	Reliability (double)	Reliability (triple)
1	99.90%	99.93%	99.93%
2	96.08%	99.44%	99.95%
3	97.19%	99.67%	99.87%
4	95.90%	99.65%	99.95%
5	95.99%	99.60%	99.88%
6	95.99%	99.38%	99.85%
7	96.28%	99.62%	99.82%
8	96.13%	99.39%	99.83%

Table 11.5: Third experiment results, reliability with single, double and triple spatial redundancy

Chapter 12

Conclusions

In this thesis we presented **TDMH**, a wireless communication stack capable of **multi-hop** mesh networks, with **bounded latency** and **low power consumption**.

The focus was put in the scheduling and routing problems, algorithms and implementations that were the main task of this thesis' work. We presented also the design decisions and some implementation details of several other TDMH components that were created within the same work.

Summarizing, the two major tasks behind this thesis were

- Solve the **scheduling and routing** problem of TDMH and implement the resulting algorithm.
- Design and develop the modules needed to turn TDMH into a **full network stack**.

These two tasks were both completed with a rigorous approach, starting from the problem formalization, to the creation and refinement of a model for the software and finally to the implementation of a software following the model. In this thesis you found details about these three aspects.

TDMH represents a novelty in research thanks to its real-time capabilities, but there is also room for future improvements, thanks to its modular design.

Bibliography

[1] K. Sui, M. Zhou, D. Liu, M. Ma, D. Pei, Y. Zhao, Z. Li, and T. Moscibroda, "Characterizing and improving wifi latency in large-scale operational networks", in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16, Singapore, Singapore: ACM, 2016, pp. 347–360, ISBN: 978-1-4503-4269-8. DOI: 10.1145/2906388.2906393. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906393.

[2] Silabs, *An1138: Zigbee mesh network performance*, 2018.

[3] L. Singh, A. Paliwal, and R. Shekhawat, "Extending battery life of smart meters by optimizing communication subsystem", in *2014 IEEE Students' Conference on Electrical, Electronics and Computer Science*, 2014, pp. 1–7. DOI: 10.1109/SCEECS.2014.6804470.

[4] R. Friedman, A. Kogan, and Y. Krivolapov, "On power and throughput tradeoffs of wifi and bluetooth in smartphones", *IEEE Transactions on Mobile Computing*, vol. 12, no. 7, pp. 1363–1376, 2013, ISSN: 1536-1233. DOI: 10.1109/TMC.2012.117.

[5] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks", in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04, Baltimore, MD, USA: ACM, 2004, pp. 95–107, ISBN: 1-58113-879-2. DOI: 10.1145/1031495.1031508. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031508.

[6] Q. Wang, K. Jaffrès-Runser, Y. Xu, J. Scharbarg, Z. An, and C. Fraboul, "Tdma versus csma/ca for wireless multi-hop communications: A comparison for soft real-time networking", in *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, 2016, pp. 1–4. DOI: 10.1109/WFCS.2016.7496512.

[7] "IEEE standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans) amendment 1: Mac sublayer", *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pp. 1–225, 2012. DOI: 10.1109/IEEESTD.2012.6185525.

[8] Seong-Soon Joo, Bong-Soo Kim, Jong-Arm Jun, and Cheol-Sig Pyo, "Enhanced mac for the bounded access delay", in *2010 International Conference on Information and Communication Technology Convergence (ICTC)*, 2010, pp. 423–424. DOI: 10.1109/ICTC.2010.5674810.

[9] Wun-Cheol Jeong and Junhee Lee, "Performance evaluation of IEEE 802.15.4e dsme mac protocol for wireless sensor networks", in *2012 The First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT)*, 2012, pp. 7–12. DOI: 10.1109/ETSIoT.2012.6311258.

[10] D. De Guglielmo, S. Brienza, and G. Anastasi, "IEEE 802.15.4e: A survey", *Computer Communications*, vol. 88, May 2016. DOI: 10.1016/j.comcom.2016.05.004.

[11] *An architecture for ipv6 over the tsch mode of IEEE 802.15.4*, 1st ed., IETF, Internet Engineering Task Force, 2019-03.

[12] M. R. Palattella, N. Accettura, M. Dohler, L. A. Grieco, and G. Boggia, "Traffic aware scheduling algorithm for reliable low-power multi-hop IEEE 802.15.4e networks", in *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC)*, 2012, pp. 327–332. DOI: 10.1109/PIMRC.2012.6362805.

[13] R. Soua, P. Minet, and E. Livolant, "Modesa: An optimized multi-channel slot assignment for raw data convergecast in wireless sensor networks", in *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, 2012, pp. 91–100. DOI: 10.1109/PCCC.2012.6407742.

[14] V. Annamalai, S. K. S. Gupta, and L. Schwiebert, "On tree-based convergecasting in wireless sensor networks", in *2003 IEEE Wireless Communications and Networking, 2003. WCNC 2003.*, vol. 3, 2003, 1942–1947 vol.3. DOI: 10.1109/WCNC.2003.1200684.

[15] A. Saifullah, Y. Xu, C. Lu, and Y. Chen, "Real-time scheduling for wirelesshart networks", in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 150–159. DOI: 10.1109/RTSS.2010.41.

[16] F. Terraneo, L. Rinaldi, M. Maggio, A. V. Papadopoulos, and A. Leva, "Flopsync-2: Efficient monotonic clock synchronisation", in *2014 IEEE Real-Time Systems Symposium*, 2014, pp. 11–20. DOI: 10.1109/RTSS.2014.14.

[17] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with glossy", in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2011, pp. 73–84.

[18] *Iso/iec 7498-1:1994 information technology – open systems interconnection – basic reference model: The basic model*, 1st ed., ISO, International Organization for Standardization, 1994-11.

[19] D. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon, "Faster algorithms for frobenius numbers.", *Electr. J. Comb.*, vol. 12, Jun. 2005.

[20] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search", *Artificial Intelligence*, vol. 27, no. 1, pp. 97 –109, 1985, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370285900840.

[21] Federico Terraneo. (2019-03). Miosix kernel, [Online]. Available: https://miosix.org/.

[22] F. Terraneo, P. Polidori, A. Leva, and W. Fornaciari, "Tdmh-mac: Real-time and multi-hop in the same wireless mac", *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 277–287, 2018.

[23] Federico Terraneo. (2019-03). Wandstem: The next generation low-power sensor node, [Online]. Available: https://miosix.org/wandstem.html.

[24] Paolo Polidori, "A time deterministic mac protocol for low latency multi-hop wireless networks", Master's thesis, Politecnico di Milano, 2017. [Online]. Available: http://hdl.handle.net/10589/140120.

[25] OpenSim Ltd. (2019-03). Omnet++, [Online]. Available: https://www.omnetpp.org/.

[26] GNU Project. (2017-09). Gnu general public license, version 2, [Online]. Available: https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

[27] Federico Terraneo. (2019). Tdmh: A time deterministic wireless network stack, [Online]. Available: https://github.com/fedetft/tdmh.