POLITECNICO DI MILANO

Master's degree in Computer Science and Engineering

Dipartimento di Elettronica, Informazione e Bioingegneria

A time deterministic MAC protocol for low latency multi-hop wireless networks

Relatore: Prof. Alberto Leva
Correlatore: Terraneo Federico

Tesi di Laurea di:
Paolo Polidori, 838206

Anno Accademico 2016-2017

Contents

List of Figures

List of Tables

Chapter 1

Abstract - Italiano

Viene presentato un protocollo di Medium Access Control (MAC) a Time Division Multiple Access (TDMA) deterministico per Wireless Sensor Networks (WSN). Le sue caratteristiche sono: la minimizzazione della latenza, la possibilità di riservare un data rate minimo garantito per ogni comunicazione e il supporto di reti multi-hop. Vari protocolli multi-hop sono stati proposti durante gli anni, i quali implementano diversi paradigmi e hanno varie caratteristiche. Nessuno garantisce data rate o latenze tali da poterli considerare realmente deterministici, entrambi requisiti necessari per utilizzare il protocollo in sistemi real-time. Durante gli anni è stato ideato il protocollo Glossy, il quale realizza un'efficace primitiva di flooding. Esso è alle fondamenta di FLOPSYNC-2, il quale ottiene sincronizzazioni temporali inferiori al μs con consumi inferiori al μA. Questo concetto è stato sfruttato per ottenere un protocollo MAC deterministico per WSN il cui obiettivo principale è garantire latenza e un data rate minimo. Dato che FLOPSYNC-2 fornisce una primitiva di sincronizzazione nota per essere efficiente, esso viene usato per allineare i tempi nella rete, in una fase chiamata synchronization downlink. Il protocollo dispone anche di una fase di uplink il cui scopo è raccogliere informazioni sulla topologia di rete e sulle richieste di allocazione del canale (stream), i quali sono trasmessi fino al master un nodo alla volta. Successivamente, in maniera centralizzata, viene calcolata una schedule, che viene poi distribuita usando una fase di downlink. Infine, la fase di trasmissione dati viene eseguita secondo la schedule distribuita, facendo comu-

nicare i nodi deterministicamente. Partendo da riferimenti storici, questa tesi analizza i protocolli attualmente disponibili per le WSN. Poi il problema viene presentato nei suoi dettagli. Segue il design e relativa descrizione del protocollo. Viene poi realizzato un modello software del protocollo, illustrandolo nelle sue parti. Infine vengono presentate e analizzate delle simulazioni per effettuare una verifica dei concetti.

Chapter 2

Abstract - English

A deterministic Time Division Multiple Access (TDMA) Medium Access Control (MAC) protocol for Wireless Sensor Networks (WSN) based on time synchronization is realized. Its key features are the latency minimization, the possibility to reserve a granted minimum data rate for each communication and to cope with working in multi-hop networks. Many multi-hop protocols have been proposed during the years, implementing different paradigms and having different characteristics. However, no one deals with guaranteed data rates nor latencies enough to be considered truly deterministic. Though, These are necessary requirements in order to operate in real-time systems. Anyhow, during the years, a flooding protocol, called Glossy, came up with an efficient idea to transmit in WSNs. This is used as foundation for the FLOPSYNC-2 architecture, used to achieve sub-μs time synchronization and accuracy at a sub-μA consumption in WSNs. Therefore this concept has been leveraged to obtain a deterministic MAC protocol for WSNs whose main target and feature is to guarantee latency and minimum data rate. Since FLOPSYNC-2 provides a synchronization primitive, known to be efficient, it has been exploited to perform network time alignment, in a phase called time synchronization downlink. The protocol uses also an uplink phase to collect network topology and nodes bandwidth allocation requests (streams), which are transmitted node by node up to the master. After that, a schedule is computed, in a centralized way. Then it is distributed using another downlink phase. Finally a data phase is executed,

according to the distributed schedule, to make nodes communicate deterministically. Starting with historical references, the thesis analyzes the currently available protocols for WSNs. Then the problem is presented in its details. The protocol design in which all its parts are described follows. Then a software model to implement such protocol is illustrated. Simulations with the aim of proving the concept are then presented and analyzed.

Chapter 3

Introduction

General purpose wireless protocols often employ a simple statistical multiplexing MAC layer, based on some form of CSMA/CA, and on top of that build the rest of the protocol stack. However, statistical multiplexing suffers from the presence of collisions, which degrade the MAC time-determinism. Since all higher level network services are built atop of the MAC, this time uncertainty has far-reaching consequences. This is because, unlike other nonidealities of a protocol layer, such as missed packets that can be counteracted at higher layers with acknowledgment and eventual retransmissions, or bit errors that can be identified with error correcting codes, time determinism, once lost, cannot be compensated for at higher levels. In this thesis, we propose a different approach. We start with a clock synchronization scheme, FLOPSYNC-2, that performs synchronization with packet sending, receiving and retransmission interfacing directly with the physical level. This protocol has no dependencies from a MAC, and can be implemented fully without a working one. Then, we propose a MAC that builds atop of the aforementioned synchronization scheme, using the knowledge of time for a robust TDMA schedule, and the hop number assigned to each node through the synchronization packet flooding, to enable a topology collection that allows the node master to reconstruct the network and schedule communication flows. This chapter will provide an overview of the technologies, applied in different fields, considered for the development of the idea and for the realization of the relative protocol.

3.1 Internet of things

During the last decades of the previous century informatics grew up exponentially and with it, the number of devices. The devices are now very heterogeneous and are used for many different purposes. Therefore now we have a high quantity of different electronic devices with which we can interact. It's possible, for example, to use an app to remotely set up our house heating system while we are coming back from work. This concept, of having a lot of small devices of every kind and purpose interconnected, is called Internet of Things. It can be applied to different kind of devices from various domains. For instance, since wearable devices have come up, even simple everyday objects like a washing machine can now connect to the Internet. All these systems need to communicate and usually they don't have much computational power. Plus, some of them can't have a large battery installed, because they need to be small in size in order to be portable. These constraints create a whole new series of challenges in different fields of informatics, from hardware design to networking.

3.2 Wireless Sensor Networks

A particular kind of IoT devices are the Wireless Sensor Networks. It is a category of devices used in pervasive data systems for data sensing and at most performing actuation. These networks are usually composed of many devices, called nodes or motes, normally spread in an area like a building, on a volcano or in a vineyard. Their target is to collect and transmit data for different purposes. Their peculiarity is the battery duration. Batteries should be thought to last many years and long enough to avoid frequent physical mantainance of the nodes, which can be difficult in some environments. This adds another constraint to those presented by common IoT devices. This kind of node is often developed and realized using a hardware of limited dimensions (from that of coin up to that of a box of shoes). The hardware frequently contains a microcontroller, a radio transceiver, a battery for powering the device and sensors, directly attached or connected through connectors.

3.3 Networking

A network is a collection of two or more devices and links among them, which makes it possible to establish connections and allow them to exchange data. Networks can be seen in many different ways and in computer science they are modeled using a stack to represent the network at different abstraction levels. The two mostly used models are the TCP/IP model [1] and the ISO/OSI model [2], which divide the network in, respectively, 4 and 7 layers. In the former, the layer stack starts from network access; then passing by internet and transport, ending at the application. In the latter, the layers are: physical, data link, network, transport, session, presentation, application. In this thesis, just the first two layers of the ISO/OSI model, corresponding to the network access layer of the TCP/IP model, will be examined.

3.3.1 Physical layer

The first part to consider about the physical layer is the medium. The first commercial networks started being developed during the 70's using coaxial cables as medium. They worked with the Ethernet standard [3], which defines how to operate at the physical and data link layers. This system is still in use, but just on different mediums (twisted pairs, fiber cables, etc.). On the other hand, in Wireless Sensor Networks, the nodes can be moved or deployed in difficult environments, in which physically connecting them with a cable, either to supply them with current or to connect them to a network is not possible. Therefore, as the name suggests, the medium chosen in WSNs is wireless. The most widely known protocol for wireless networks is 802.11 [4]. It specifies how to implement wireless local area network communication for computers focusing on different ranges of frequencies. However, this standard has too high energy demands providing also a higher throughput, which is unneeded in the case of LR-WPANs (Low rate - wireless personal area networks). Instead, for such environments, 802.15.4 [5] was developed. The standard contains many physical layers at many different frequencies and modulation. Anyway we will adopt the most diffused one, the Offset quadrature phase-shift keying (O-QPSK) Direct Sequence Spread Spectrum (DSSS) modulation on the 2.4 GHz ISM band,

whose features will be better explained later.

3.3.2 Medium access control protocols

Ethernet, as already stated, is a historical reference for networking at the physical layer and the same holds for the data link layer. Its key concept is that when a device needs to access the media, it checks that no one else is transmitting, before starting. If, after starting, it senses a collision, they (all the nodes sending contemporarily) start jamming the channel and retry after a backoff period. This approach is called Carrier Sense Multiple Access with Collision Detection (CSMA/CD) [6].

In wireless networks the most common approach for this issue consists in avoiding its management. This is done by using a CSMA/CA algorithm (where CA stands for Collision Avoidance) to access the channel [7]. When a wireless connected device wants to transmit a frame, it senses the channel for a certain period, after which it starts transmitting in case the radio channel results still free. If not, the radio retries after a certain time, called Collision Window, randomly chosen within a defined interval. If the channel is free, the node will transmit a Request To Send (RTS), which is a frame shorter than a data frame. This process is thought in the case that two (or more) different nodes, both trying to access the channel, sense an available channel and contemporarily, start accessing it by sending the message. In such a situation the two frames would collide and get damaged, clogging the channel for a long period. The RTS, in case is correctly received, will be followed by a reply from the receiver called Clear To Send (CTS), advising all the nodes that he will be receiving a packet. In this way all the nodes sensing the signal transmitted by the source device will avoid transmitting and the same will happen with the devices which received the CTS from the destination device.

CSMA/CA is applied also in 802.15.4 [5], which is the MAC protocol standard at the base of ZigBee [8], that is a commercial protocol suite for WPANs. The main difference with 802.11 is that the device doesn't continue sensing during the Collision Window in order to avoid battery waste. It also avoids the RTS/CTS exchanges which would be too demanding due to the reduced physical protocol data unit dimensions (133 vs 2376 bytes for the PPDU). This protocol

received many amendments over the years in order to overcome its limitations such as low reliability, unlimited delays and the absence of protection against interference/fading, which makes it hard to use in real time systems or in applications with strict requirements about network reliability and latency. In its latest version, the 802.15.4 standard supports many new protocols designed for different application domains: Time Slotted Channel Hopping, Deterministic and Synchronous Multichannel Extension, Low Latency Deterministic Network, Asynchronous Multi Channel Adaptation, Radio frequency identification BLINK. These protocols provide a whole set of new features, improving significantly the previous standard. Their characteristics and features will be better explained in 4.3.

All these protocols can be classified into two types: the timeslotted/scheduled and the CSMA/CA ones. Most of them are actually an hybrid, as the first release of 802.15.4 is. The most relevant, though are trying to move towards a more timeslotted architecture, coping with the fact that the nodes are not time synchronized and they must try to keep the radio turned off as much as possible. So, in order to take this path, it is advisable to consider this aspect more in depth.

3.4 Time Synchronization

In networks, time synchronization didn't use to be a concern, since the majority of network protocols are historically contention-based and indeed, CSMA, under many variations, is extremely common. However, with the introduction of new kind of environments, in which a collisions and subsequent retransmissions from all the colliding transmitters has a non negligible energy cost, the development of networking protocols moved towards time determinism and time-slotted network access.

In particular, up to now, what is done to achieve synchronization in WSNs is either using a causal consistency, referring to the previously received message, or using control messages like beacons and ackowledgments to transmit the timestamp, that represents an absolute reference in the network. This means that the time synchronization is realized on top of layer 2 and suffers from its

weaknesses. Each protocol can have different delays based on the software, its implementation and the hardware on which it is deployed. Moreover, even the radio channel can introduce its delays or interferences. All of these characteristics can be non-linear and non-predictable, depending on various environmental parameters and conditions. This makes very hard to have an optimal reference, since the more abstractions layers we have, the more errors will be the introduced on the final measure. Therefore it is advisable to move the mechanism where the system's dynamic is more convenient, which has been proved to be when directly plugging it to the hardware radio. This approach will be illustrated better in 5.1.1.

This premise led to the realization of a MAC, which tries to be as deterministic as possible, making the most of this concept of time synchronization. This capabilities provide the possibility to build a protocol, whose delay, as well as the data rate, can be easily constrainted, making it desirable in process control networks and industry automation environments.

Chapter 4

Technological overview

4.1 Taxonomy and examples

WSN MAC protocols are very heterogeneous and each of them has a particular target and different priorities. Combining this with the already named constraints brings a wide range of solutions featuring different characteristics. [9] illustrates a taxonomy divided in four distinct branches: asynchronous, synchronous, frame-slotted and multichannel.

Asynchronous protocols trade off the synchronization with the cost of periodically waking up for checking the presence of incoming transmissions and the major cost of initiating a transmission. The first way to achieve this is duty-cycling nodes' wake up of a time as long the preamble, in order to avoid listening continuously. This is called preamble sampling. An improvement is Low Power Listening (LPL), which combines this technique with CCA before sending. Other solutions exist, in order to try to achieve synchronicity between transmitter and receiver a posteriori with as few energy as possible.

Synchronous protocols try instead to negotiate a wake up schedule to make every node know when it needs to listen for incoming packets or when it can try contacting a neighboring node. An example is S-MAC [10], for which all the nodes wake up periodically based on an interval called synchronization period. After waking up they contend the channel with their neighbors by trying to book it (with RTS and CTS), in order to communicate using it at the end of

the first synchronization period. Inactive nodes can sleep for the whole round. However it's noticeable that a multi-hop path needs more rounds to communicate; an issue that [11] tried addressing with adaptive listening. Another way to achieve synchronicity is with Future Request To Send. Its purpose is to make the node listen for a certain period for possible incoming transmissions instead of fixing the length of the active period and forwarding the request to the next hops, in order to make the packets travel through as many more hops as they can. Meanwhile, it lets inactive nodes sleep as much as possible, by adopting a flexible active part in the synchronization period. Staggered schedule protocols, like DMAC [12], offer a way to cope with data collection by dividing each node's time among its children and down to leaf nodes to achieve the lowest possible latency. A different approach to solve cases in which there are more populated routes is with adaptive duty cycling, which makes nodes on active paths spend more time sensing the channel for incoming packets, limiting the delays.

The family of frame-slotted protocols is instead formed around the TDMA concept, to achieve maximal channel utilization under circumstances of high contention. Anyhow, its cost is the need of global synchronization that may be achieved in a potentially difficult way. Although this leads to a low throughput and low channel utilization in case of low contention. An approach to solve this is with slot stealing, combining CSMA with TDMA in inactive channels. Adaptive assignment tries instead to switch between random access periods and TDMA periods and it assigns TDMA slots to nodes that need to send data, for which a representative protocol is TRAMA [13]. TreeMAC [14] tries to employ a different technique by looking for the maximization of the throughput instead of the spatial reuse of slots. Its strategy is about using a data gathering tree structure to collect data. They divide time in cycles, containing frames, each of whose is composed by 3 slots. A single slot can be assigned once every 3 hops, in order to reuse slots as much as possible without occurring in interferences. Frame assignment is distributed among nodes at the same depth, dividing the frame space assigned to the parent node. An improvement is PackMAC [15], which tries to reuse unassigned slots by applying a distributed free time-slot search algorithm after information collection, which is run in parallel with the TreeMAC algorithm. Only after them the network can start using such schedule.

Finally another kind of frame-slotted protocols is composed by those who assign slots to receivers instead of senders. This makes collision-free transmission no more longer guaranteed, but only nodes having data to deliver to a certain node have to wake up for the assigned slot.
To improve parallel transmission and network throughput, multichannel protocols have emerged. The first protocols were used to negotiate the channel to be adopted, but later a distributed alternative was realized, needing each node to choose a free channel within a 2 hop distance. In this last implementation, if a node needs to send a packet, it needs to snoop alternatively its own channel for data reception and listen to the channel where the destination node is listening in order to perform CCA. A different strategy can be adopted by grouping frequently communicating nodes in separate channels, dividing so the nodes which don't communicate much with each other. An example of this paradigm is GBCA [16] which uses a game theoretic approach to find a channel assignment that minimizes the total interference (shown to be NP-hard) with a suboptimal result. In other protocols the multichannel strategy is applied by node role, i.e. to send or to receive, notifying other nodes about the chosen channel. This can also be combined with channel hopping, meaning that nodes hop among the available channels after a predefined time using a pre-shared hopping sequence thus managing the hop assignment as a channel offset assignment.

4.2 WirelessHART and ISA-100.11a

WirelessHART[1] and ISA-100.11a[2] are two standard coming directly from industries, with Emerson as the current leading supplier for WirelessHART followed by Siemens, ABB, etc.; and Honeywell and Yokogawa for ISA-100.11a. As illustrated in [17] they both provide a TDMA protocol with frequency hopping, using the 2.4 GHz ISM band and employing a Direct Sequence Spread Spectrum (DSSS) technique. Both of them use a mesh topology and have technical limits of thousands of devices. Although when this number increases, even the energy consumption of nodes does, especially in those nodes residing in crucial points

[1]Other information about WirelessHART can be found at `https://webstore.iec.ch/publication/24433`

[2]Other information about ISA-100.11a can be found at `https://www.isa.org/store/products/product-detail/?productId=118261`

of the network. Moreover even the latency of packets may grow unpredictably. In ISA-100.11a three different hopping modes are defined. With slotted channel hopping, time is divided in frames composed by time slots, which can be assigned by the network manager to a sender and a receiver, with a channel offset. In slow channel hopping, instead a collection of contiguous time slots is grouped and the protocol is still subjected to channel hopping, but not to channel offset, leaving in fact access to the channel to CSMA/CA. Hybrid hopping can be also configured, resulting in a combination of the two solutions. WirelessHART does not specify how channel hopping can be realized, but states that it is handled by the network manager and distributed to the devices during the joining process. Security is another concrete concern and both the protocols support Cipher Block Chaining Message Authentication Code (CBC-MAC)[18] using Advanced Encryption Standard (AES)[19] with 128 bytes symmetric keys. As the reader will notice by some common characteristics, these protocols inspired the evolution of 802.15.4 during the years.

4.3 IEEE 802.15.4

The commercial reference for the MAC protocols, as stated in the introduction, is 802.15.4. This protocol was born taking the concept of CSMA/CA from 802.11 and adapting it to low power devices, by relaxing some rules and allowing the devices in the network to save energy for longer time. Also, the maximum frame size is limited, with 133 bytes against the 2304 bytes of WiFi. Its efficiency is measured in the equation (4.3). In 802.15.4, many physical layers are proposed in the standard and many were added with amendments, but we will focus on the O-QPSK 2.4 GHz ISM band, which is the most common and used for this protocol. In this band, 16 channels are available, each with a 2 MHz bandwidth, distributed with 5 MHz of distance. 32 consecutive radio symbols, translated with a DSSS map using a Pseudo Noise (PN) sequence of 64 bits per byte grants a data rate of 250 kbps. The time needed to send data of arbitrary dimension using this physical layer is calculated in the equation (4.1). Other physical layers use various modulations (BPSK, ASK, CSS, etc.) at different frequencies (868 MHz, 902 - 928 MHz, UWB, etc.).

d_{data} = Dimension of the data to be sent in bytes

T_{pkt} = Time to send an amount of data (CRC included)

T_{pkt}^{crc} = Time to send an amount of data (CRC excluded)

η_{pkt} = Spacial efficiency in sending a certain amount of data

$$T_{pkt}(d_{data}) = 31250 \left(6 \cdot \left\lceil \frac{d_{data}}{127} \right\rceil + d_{data} \right) \tag{4.1}$$

$$T_{pkt}^{crc}(d_{data}) = 31250 \left(6 \cdot \left\lceil \frac{d_{data}}{125} \right\rceil + d_{data} \right) \tag{4.2}$$

$$\eta_{pkt}(d_{data}) = \frac{d_{data}}{6 \cdot \left\lceil \frac{d_{data}}{127} \right\rceil + d_{data}} \tag{4.3}$$

The devices admitted in these protocols are of two kinds: Reduced Function Devices (RFD) and Full Function Devices (FFD). RFDs are only limited to send and/or receive. FFDs, on the contrary, are capable of both sending and receiving and also acting as coordinators.

This MAC allows two topologies: star and peer to peer. With the star topology a central FFD coordinates the PAN to which other devices can connect. The peer to peer topology allows instead many other topologies to be formed, but needs at least a FFD among them. This allows trees and mesh networks to be formed, with the constraint that most of the devices are FFDs.

The base time reference in this protocol is defined by a periodical unit called superframe. Each superframe starts with a beacon transmitted by the PAN coordinator. Every superframe contains an active part and, potentially, an inactive one, in which the PAN coordinator can stop listening to the network. The active part is then divided into CAP (Collision Access Part) and CFP (Contention Free Period). Each device can send data either in the CAP, with contention, or ask for the reservation of a GTS (Guaranteed Time Slot) in the CFP. To access the channel during the CAP and transmit, a device should wait for a random time $\in [0, 2^{BE-1}]$ backoff periods where BE is a backoff exponent usually initialized with a value of 3. After having waited, the node can start performing Clear Channel Assessment (CCA) for Contention Window (CW) multiplied by the backoff period. If the channel is found occupied BE is incremented (if less than a boundary value) and, if the algorithm has not been

iterated too many times, the procedure repeats from the start.
As previously stated, timing in 802.15.4 is based on beaconing. In the case of a tree based network, having so a peer to peer topology, beaconing is performed by every coordinator. Each of them has an incoming beacon and a transmitted beacon, which delimits the superframe structure. Superframe overlapping is not allowed, therefore the outgoing beacon must be sent during the inactive part of the incoming superframe and the outgoing superframe must not end after the next incoming superframe starts.
Another job accomplished by beaconing is time synchronization. In beacons the local timestamp of the source is indicated and the beacon is taken as a temporal reference in the network. The default window in which a node expects to receive packets from the network is of $\pm$ 2200 μs.
In newer versions of the standard, five new layer 2 protocols have been added, in order to serve different purposes. A survey describing these new protocols has been proposed in [20].

4.3.1 TSCH

Time Slotted Channel Hopping (TSCH) uses a TDMA approach to manage how the nodes access the network combined with multichannel support, for better allocating transmissions in multi-hop networks. It is the most developed among the new MAC behavior modes. In TSCH time is divided in slotframes, which continuously repeat. Slotframes are then composed of timeslots, slices of time in which a transmission and an acknowledge can occur. The Absolute Slot Number (ASN) is the count of how many slotframes has past since the network starts up, thus making a reference timeline. It is maintained by the network and shared among the nodes using Enhanced Beacons (EBs). The main characteristic of the protocol is channel hopping and it is used to switch channel based on the ASN. Each link will be then represented as a communication in a timeslot with a channel offset called cell. A link can be dedicated, meaning that two nodes only use a link, or shared, meaning that more communications can occur in it. In shared links, transmissions use a simplified CSMA/CA algorithm with a backoff exponent in case of collision but without CCA. Either to grant network coexistence or avoid the choice of noisy or bad quality channels, blacklisting should

be also possible. This is achieved by making the scheduler avoid such channels when formulating timeslot assignments. Anyway, the scheduling function is not included in TSCH, but the standard leaves instead the opportunities open for designs and implementations. Although most of the existent schedulers are not suitable for TSCH because they do not allow per-packet scheduling, channel spatial reuse is not always considered, devices' memory and computational power is usually too limited.

Two different families of scheduling algorithms have been proposed: centralized and distributed. Both of them try to solve the same problem, starting however from different points of view and considering further contexts. Among them Traffic Aware Scheduling Algorithm (TASA) [21] proposes a centralized approach starting with a statically configured topology represented in a graph structure, whose scheduling is extracted with a technique that combines matching and vertex coloring. [22] is another centralized algorithm which tries to maximize throughput using a Hungarian algorithm and minimizes delays using a branch and bound algorithm. In both of them, no strategy of distribution is proposed. A distributed version of TASA, namely DeTAS [23], claims to cope with building an efficient and fair scheduling with few information exchanged with each node's neighbors, achieving better results than its centralized version. Another distributed but simpler solution was proposed in [24]. It tries to allocate slots in random cells and performs *"housekeeping"* operations to distribute the load among the available cells based on packet delivery ratio. DIVA [25] is a decentralized scheduling solution in which nodes try not to concentrate traffic towards the root node, preferring peripheral paths, as main target for the scheduling result. Its solution is based on connection management packets, exchanged between neighbors, without considering multi-hop contexts. Nodes listen to connection request packets while they are in idle state with a defined probability. If a connection request packet is received, they reply with an ack and switch to the requested channel, expecting data to arrive. Time synchronization in TSCH can either happen when receiving beacons as previously stated or at the reception of a data or a control frame from the node's time source neighbor. Effort was put in improving TSCH synchronization by [26]. Their work is based on the concept of adaptive synchronization, which uses clock drift

estimation to decide when performing synchronization is needed (i.e. the estimated skew goes beyond a threshold), avoiding battery wastes. Results showed that in a networks composed of 3 hops, nodes desynchronize of a maximum of 76 μs and need to transmit an average of 83% less synchronization packets then in a network that doesn't use adaptive synchronization. As showed by [27], a combination of HF and LF crystals can be used to improve the resolution and the results previously presented to under $2mu$s per-hop.
A commercial version of this protocol, called SmartMesh IP® is brought by Linear Technologies [28]. They support up to 100 motes in a network if using their manager's embedded version, while capabilities over 1000 motes are achievable if using VManager, another manager version running on a Linux VM.

4.3.2 DSME

Deterministic and Synchronous Multi-Channel Extension (DSME) can be considered as a modification of the original 802.15.4 protocol, trying to expand the Collision Free Part (CFP) to have a more reliable protocol by avoiding packet collisions and energy wasting. CFP was already present in the original standard, but was very limited and actually usable only by nodes close to the PAN coordinator. This brings more Guaranteed Time Slots (GTS), whose use in the original standard is actually possible only for devices situated in the proximities of the PAN coordinator [29]. Another feature consists in allowing devices to operate in multiple channels in two different ways: channel hopping and channel adaptation. Channel hopping mode is used to periodically switch channel, based on a pre-shared hopping sequence, in order to avoid to dealing with unusable links due to interference or multipath fading. Each node, will then have its channel offset, which is summed to the base channel (given by the hopping sequence combined with the current sequence number) and every communication will occur at the receiver's chosen channel. Channel adaptation mode is instead a mechanism used to allow the nodes to negotiate a channel to communicate with each other, using any free available channel and taking into account the channel quality estimated by the nodes. This results in a solution for multi-hop mesh networks with a more deterministic latency. Its beacon scheduling and slot allocation are not performed by a central entity but they

are instead managed in a completely distributed way. This means that each node can autonomously allocate or deallocate GTS slots, resulting in a different architecture with respect to TSCH. As shown previously, in TSCH it's not always possible to reconfigure a network, recompute a schedule or distribute it at runtime, making DSME more adapt to dynamic networks.

4.3.3 LLDN

Another operational mode proposed in the 802.15.4 amendments is Low Latency Deterministic Network (LLDN). It is a single-hop (star topology) single-channel protocol for low latency applications, suitable for a single topology only. Its design target is sensor sampling and collection every 10ms from 20 different sensors. In this protocol the time is seen as a sequence of superframes, which repeat seamlessly. Each superframe is divided among the nodes transmitting their data in short MAC frames with just 1 byte header.

4.3.4 AMCA

Asynchronus Multi-Channel Adaptation (AMCA) is an additional operation mode for 802.15.4 designed for Smart Utility Networks (SUN), infrastructure monitoring networks and process control networks. In this protocol each node picks up a channel with the best local quality and every device that needs to transmit switches to the channel the message receiver has chosen. Information about nodes' channels can be obtained using a special packet or by asking the network coordinator to transmit a beacon.

4.3.5 BLINK

The last mode proposed in the amendment is Radio Frequency Identification Blink (BLINK). It is a protocol intended for item and people identification, location and tracking. It works by making the nodes just communicate their IDs to other nodes without any association nor acknowledgement using an ALOHA-like protocol.

Chapter 5

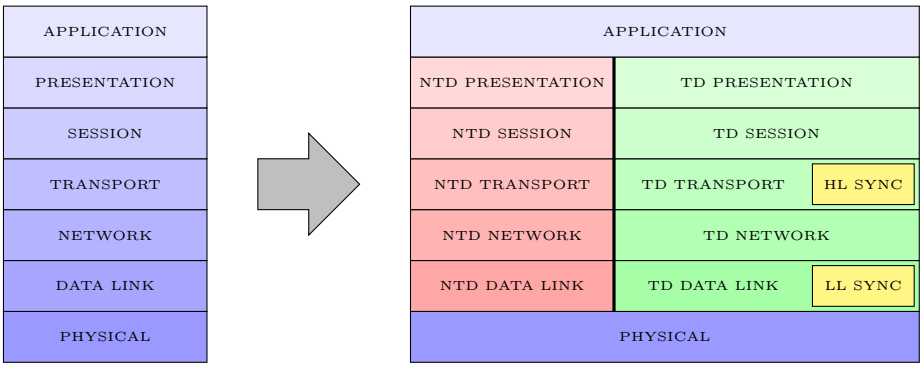
Problem statement

The problem that this thesis tries to solve is the lack of a deterministic multi-hop MAC protocol capable of providing limited latency and a guaranteed minimum data rate in WSNs. The existing protocols that try to assign channel utilization use different kinds of synchronizations and mechanism to perform such assignments. However, better approaches exist and are discussed in section 7.2.1. Also, the WSN world is very heterogeneous and a very wide range of devices is available, therefore an analysis of what to target the protocol for is performed in section 5.2. Anyway the protocol tests would require a large number of devices and configurations. So, a simulator was used for sketching and testing the protocol model, being able to stress it under various conditions. It is illustrated in section 5.3.

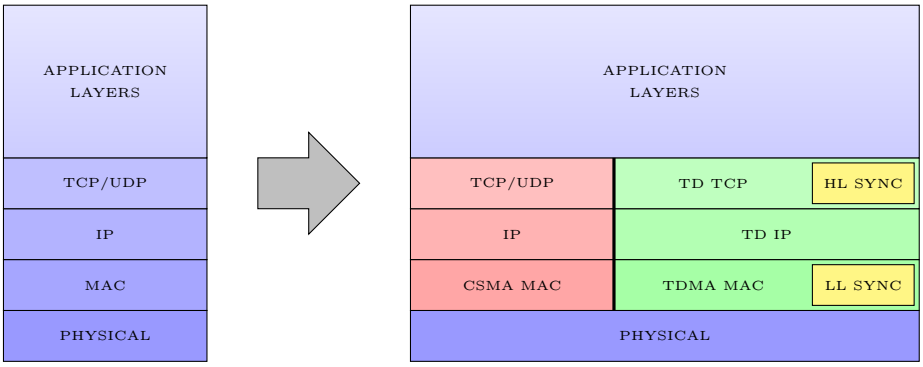
5.1 Time synchronization

Time synchronization among devices is a longstanding issue. It has been tried to find a solution in different ways and with various approaches, either using hardware or with network protocols, at different levels. In networking, the original approach to time synchronization was to delegate the problem to the same layer in which it originated, or to work around it by using a non-time deterministic solution (NTD protocols). This, for a protocol of layer 2, means sensing the media for a predefined period, after which a sender can start transmitting its packets. This concept is the base of CSMA, on which rely most of the networks

currently used around the world. However, CSMA protocols need to sense the channel in order to avoid the occurrence of a collision. Plus, even after sensing, collisions are keen to occur, especially in case of overcrowded networks. This happens due to the presence of the hidden node problem [30]. Those are the reasons why such protocols are called *"best effort"*. On the other hand, recent developments led to a more conscious energy consumption which in certain environments is also a strict constraint, due to the difficulties of energy harvesting or delivery. Direct consequence of this concept is that the previously considered methods for achieving time synchronization became less suitable for the context, making a well synchronized network more interesting. However, the problem in this kind of solutions is the non-deterministic cost of collisions, whereas keeping network times synchronized has a fixed cost. Therefore this trade-off needs to be considered and analyzed on the basis of the context and the cost of synchronization. They depend on various factors, but can be divided into those really making a trade off and those which worsen the conditions in both the techniques. The presence of interfering devices, like Wi-Fi and Bluetooth ones, sharing the frequency spectrum, can hinder communications making the risk of collisions worse in any case. Though, internal interferences, i.e. concurrent transmissions, can make the best effort case worse than the synchronized one if the energy loss is higher than the one used to achieve synchronization, due to undelivered/colliding messages. In fact, every kind of synchronization based on the protocol level at which it is implemented has its own purpose and its own approximation. However, low level synchronization protocols (LL SYNC) made its cost affordable and thus interesting to be used in place of synchronization techniques using highers layers of the protocol stacks (HL SYNC). Therefore, since the goal was to develop a MAC protocol targeting environments in which there are strong determinism and latency requirements, the choice made was to employ FLOPSYNC-2. It is a low level synchronization protocol, whose characteristics are better explained in the next section, making possible to obtain a time-deterministic protocol stack. The differences between an asynchronous environment, versus LL-SYNC and HL-SYNC is illustrated in figures 5.1a for the ISO/OSI protocol stack and in 5.1b for the TCP/IP protocol stack. A comparison on orders of magnitude precision and the purpose of time synchronization



(a)



(b)

Figure 5.1: Time determinism and synchronization inside the ISO/OSI protocol stack (a) and the TCP/IP suite (b).

at different layers is displayed in table 5.1.

5.1.1 FLOPSYNC-2

Clock synchronization is a two front battle where it is necessary to compensate for the nonidealities of both the local clock and the channel used for time dissemination.

For what concerns the sources of error in the local clock, other than the initial offset and frequency error, a clock suffers from quantization, as the resolution of any clock is finite, as well as various sources of nonlinearity that cause the clock frequency to change, differing by their timescale. Among these, we can consider jitter, a random variation of the period of each tick, that is thus as fast as the clock itself. Wander is a random walk in clock frequency occurring over a timescale of milliseconds to seconds, typical of systems employing PLLs

Layer	Physical	Data link	Network, Transport
Synchronisation purpose	Physical medium operation	MAC	Internetworking
Technology	Hardware (PLL,...)	Hardware/software	Software
Examples	Bit-level synchronisation (e.g., CAN) Clock recovery (Ethernet)	Random exponential backoff (Ethernet)	TCP/IP
Main error sources	Physical (jitter, skew,...)	Collisions	Routing,...
Error order of magnitude	ns–μs	$\sim 10\mu s$–ms	$\sim 10ms$

Table 5.1: Typical synchronisation error magnitude at the various ISO-OSI layers.

for clock generation. At a greater timescale, seconds to minutes, the temperature dependence of quartz crystal oscillators becomes significant, while for even longer timescales, aging effects arise.

FLOPSYNC-2 is the clock synchronization scheme used in this thesis. It is a low-level scheme that starts from a model of the local clock and corrects it with a controller specifically designed to counteract its nonidealities. It is capable of being highly precise, below the μs, with a power consumption of less than $2.1\mu W$. Moreover, it provides a monotonic and continuous clock by only performing rate corrections except when resynchronizing after the network connection is lost.

FLOPSYNC-2 also integrates seamlessly with the Virtual High-resolution Time (VHT) [32] approach,which is a technique used to achieve a high resolution timebase without requiring an always-active (thus power hungry) high frequency oscillator. It is realized by employing two different clocks, i.e. two different crystal oscillators, a slow and low-power one combined with an high frequency and high consumption one. Using the high frequency crystal only when the node is processing and using resynchronizing it with the low frequency one during sleep times makes a good consumption compromise as well as it makes it possible to have an high resolution timer when needed. An enhanced jitter-compensated version of VHT has been proposed [33]. This solves an accuracy issue which is present in the original VHT algorithm since it didn't consider the intrinsic jitter of the crystals.

For what concerns the communication channel, the most significant source of error is variable network delays, followed by propagation delays. FLOPSYNC-2

relies on Glossy [31] to disseminate the clock synchronization packets in a time-deterministic way, and uses a custom scheme to compensate for propagation delays.

Glossy is an innovative idea for network flooding in WSNs. It exploits concurrent transmissions for taking profitably flooding WSNs. As stated before, 802.15.4 uses an O-QPSK modulation at 2.4 GHz with a bandwidth of 2 MHz. Data, before being transmitted is translated to chips and the inverse operation is performed by a receiving radio. The radio correlator performing such operation always tries capturing signals of every intensity and compares them to the well known DSSS PN sequences. If a signal has been received before others and can be correlated corectly, it will. This effect is called *delay capture*. Another similar effect is the *power capture* for which when a signal is stronger than others, the correlator will correctly rebuild the stronger one. Known that, Glossy built a mechanism called flooding for which a node transmit a packet and all the nodes of the first hop receive it. After a well-determined time interval all the nodes of the first hop retransmit the packet and so on and so forth. Adopting this mechanism all the nodes will have a common time reference, deducing it from the packet arrival time. It has been shown that if two nodes transmit the same packet within 500 ns, which correspond exactly to a chip period, they will have a very high probability to interfere constructively ($> 75\%$). A graphical representation of this is presented in figure 5.2. In the leftmost part, the master node transmits its packet, which is received by nodes 1, 2 and 3. Then, after a Δt these nodes retransmit the packets as well. As it can be evinced in the central part of the figure, node 4 will receive the packet transmitted by 2 and 3. In this case the power capture will make the message to be correctly reassembled. Then, as shown in the rightmost part of the illustration, also the packet transmitted by node 3 starts to be received, however its signal strength will make it be discarded by the correlator.

Reverse Flooding [35] is an extension to Glossy to estimate the propagation delay, having so precise timestamping at a precision higher than $1\mu s$ by cancelling them. This is achieved by asking to the previous hop's nodes in sight the measurement of the roundtrip time. They will reply after a predefined time delta with a packet in which, using the bar graph encoding presented to depict

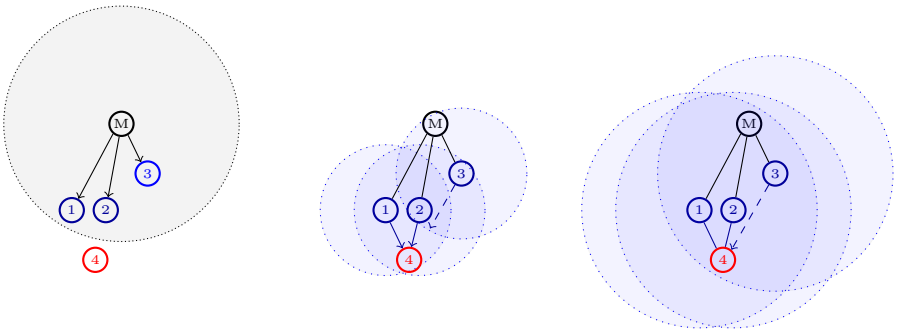


Figure 5.2: Flooding mechanism step-by-step

each node's cumulated delay to the master. By summing the obtained roundtrip time value with the response data, a good estimation of the propagation delay to the master is obtained, achieving a great timestamp accuracy by cancelling 95% of the relative error induced by propagation delays. The illustrated solutions will be used as protocol primitives in order to have a concrete time awareness at the protocol base.

5.2 WSN platform

In WSN and IOT applications the device environment is very heterogeneous and a wide range of devices has been developed. However most of them are custom made by each single company for their clients and only few try selling mass products like Linear Technologies SmartMesh IP®. The approach chosen to manage time synchronization and the lack of an off the shelf hardware capable of supporting it led to the development of a custom hardware based on publicly available components called WandStem and whose design is available as open hardware[1]. The WandStem components include an ARM Cortex M3 microcontroller that uses 32 bits instructions with 128 MB of RAM and an internal flash memory equipped with a 48 MHz oscillator crystal. It is also attached to a 32768 Hz oscillator for using VHT. The radio transceiver connected is a TI CC2520 clocked at 32 kHz. To achieve accurate packet timestamping, its arrival time is registered from the microcontroller using 48 MHz frequency, i.e. a $21ns$ resolution. Other more modern solution involve a System on Chip (SoC)

[1]Device details and schematic files are available at `http://miosix.org/wandstem.html`

architecture, with the drawback of being less customizable and not usable for implementing FLOPSYNC-2.
The operating system used with this device is Miosix [36], which has an embedded kernel written in C++ whose code is publicly available[2] under GNU GPL license.
The protocol will however be usable in many other devices, even if a fast clock will be needed for the timestamping resolution. Also the radio transceiver must be able to manage packet timestamps with such a resolution. Memory and other requirements elicitation are left to the implementer, since they depend on the protocol configuration and the network dimension, as will be explained in 6.

5.3 Testbed architecture

The strategy used for developing and testing the protocol was to develop a simulation in OMNeT++[37]. OMNeT++ is a widely used discrete event simulator to design networks and protocols. This approach made it possible to test the protocol without needing to have multiple devices physically distributed to form the desired topologies and deploy firmwares over and over, making debug and testing slow and difficult. Moreover, this choice has been made to make the protocol available to the community, to work with it, performing improvements or personalizations and comfortably simulating them without the need to realize circuits. This obviously required the development of a Miosix-friendly interface to wrap the simulator in order to have a cross-compatible software.
In Miosix, each component, either hardware or software, is mapped, as the object oriented programming (OOP) principles suggest, to a class. These classes, when representing unique entities, such hardware peripherals, use the singleton design pattern, which is particularly thought for these cases when only one instance of such class can and must exist. The drawback of this approach has been found while miming the Miosix system API, because in OMNeT++ all the nodes share the same application space, instead of having that (physically) separated, making unfeasible to use the standard singleton pattern. However, the adaptation was possible by using an API to retrieve the running simulation,

[2]Miosix code: `https://github.com/fedetft/miosix-kernel`

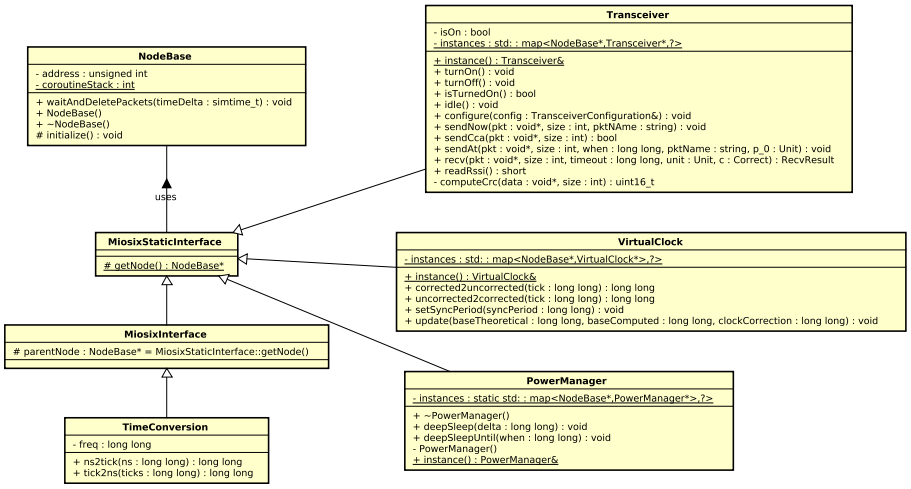


Figure 5.3: Miosix-OMNeT++ interface Class Diagram

which has a method to obtain the currently active device and thus keep a device to instance map making the adaptation possible. The proposed architecture is shown in the class diagram in figure 5.3, in which Transceiver, VirtualClock and PowerManager are examples of Miosix singletons.

For what concerns the radio, it was needed to simulate its characteristics, staying as close to the reality as possible, but keeping in mind that the development of a model was needed, which by definition can't simulate perfectly environmental characteristics and non-ideals. OMNeT++ provides a channel model useful to develop its specific characteristics. However, the easiest solution was keeping tight to the ideal channel model and use the Transceiver class developed for compatibility to model also the real channel behaviors. This is because, other than the data rate, yet developed in the simulator in a dedicated model, there was the need to model the interference. However, it depends on physical characteristics of the media and on which values does the correlator produce from sensing it. Once it finds a preamble and a SFD it starts correlating the signal present in its radio range for reconstructing a packet, making it possible for the radio to receive also jammed packets. This is an achievable result, but the theory behind Glossy has already been proved, both in simulation and in hardware in [31], [34] and [35]. Therefore, the implemented algorithm doesn't take into account all the actual physical characteristics, like propagation delays, and, as described in Glossy, it just considers any packet's arrival until

the current packet is completely received. If other packets arrive within $500ns$, a possible constructive interference is considered, elsehow the packets interfere destructively, returning a wrong CRC, if enabled, or random data if not. In case of interference within the theoretical time limit of $500ns$, random bytes from the colliding packets are picked.

Chapter 6

Protocol design

Summarizing, the needed MAC protocol finds its roots in:

Synchronization for deterministic low latencies and transmissions.

Flooding a mechanism which Glossy proved to be efficient for massively diffuse a message. Can be used to broadcast control messages from the master node or to transmit multiple packets with the same content contemporarily to the whole network.

Hop knowledge by running FLOPSYNC-2, nodes become aware of their hop, transmitted in the packet content.

802.15.4 PHY 2.4 GHz O-QPSK DSSS, since FLOPSYNC-2 is currently implemented on it.

To obtain a deterministic environment, the choice of a TDMA architecture was pretty straightforward. However, to obtain this target, every node must be able to acquire data on its exact timings. Moreover, these timings need to be scalable at each node's necessities in terms of throughput and to obtain the most efficient protocol in terms of energy wasting, meaning the highest effective throughput with the least activity time. To manage this, two different approaches are possible. Either the nodes always maintain the same necessities in terms of communication with their neighbors, or they can change their necessities over time. Considering that a MAC protocol that can't scale its devices throughput when the necessities change, leads to the need of reconfiguring the

whole network, the decision was to prefer having a more flexible structure.
A similar approach was taken for the management of the network topology, since in many environments, things can change and devices can be added or removed, or condition in the surrounding environment can be modified by external factors, making some paths more difficult to be considered reliable enough for use. Therefore the chosen approach was permitting to have a dynamic topology, capable of coping with device removal, addition and even movement. The protocol primitives which needed to be designed are therefore intended to deal with an unknown topology and unknown scheduling management, enabling every node to communicate with another and maintaining a limited latency.
However, in order to communicate, nodes have the necessity to identify themselves at least within the network they are currently connected to. Consequently, address management needs to be handled. Nodes should be able to understand to which other nodes they can talk, and at a level, sooner or later it will need to be statically defined, either at the application level with a DNS, or at network level with a static network address like an IP address, or at the data link level, with a MAC address. The translation function obviously requires some effort, that is stacked on top of the one made to achieve dynamical address assignment. Analyzed this trade-off, the choice made was to implement static addressing, without anyway excluding the possibility to expand the protocol features to enable dynamic address assignment in the future.
Finally, the elicited requirements are made for a protocol with a reasonable number of devices having a statically assigned address, dynamic topology, variable bandwidth request and being able to communicate with a TDMA protocol based on FLOPSYNC-2.
Once surveyed the desired characteristics, the protocol is ready to be designed. In the protocol there is a base time unit, a fixed interval of time to which each part will refer, to enable every node to identify each protocol phase and to deduce when to transmit and when to receive easily. This part, for uniformity and ease of interpretation with respect to similar protocols, will be called slotframe. Each slotframe can be identified by a FLOPSYNC-2 flooding, occurring as the first action.
As with every other protocol, an initial taxonomy about packets devise two types

of them: control and data. Its control part can be accomplished in two different ways: by making every node negotiate with neighbors on how to manage the space around them, which comprise the so called distributed protocols, or in a centralized fashion, by collecting nodes' necessities and knowledge, calculating centrally how the network time will be managed among the nodes and then distributing information about how and when to access the radio. Other protocols tried using distributed solutions to perform network management ((some TSCH scheduler implementations do, as shown in 4.3.1). But, since the architecture under examination has the possibility to have fast distribution of information thanks to Glossy flooding, the choice was to use a centralized way to manage the network. This led to the nodes having to perform less operations, leaving to the master node the job of taking care of the network phases organization. This made necessary the development of ways to make the master node obtain:

Data paths which represent data flows, called streams, within the network and the related required data rate.

Network topology which can be seen as a list of adjacencies, in order to form a graph of the connections between nodes in the network.

It is also necessary to realize a way to obtain and distribute the network schedule. It is so evident, that a second taxonomical distinction is present for the control packets. Some of them are produced by all the nodes and need to reach the master node, whilst other are sent by the master and need to reach every node of the network. They will be respectively called uplink and downlink phases. The details about them will be stated and the presented issues analyzed in the following sections. A first base formula of the protocol time division can be calculated, as illustrated in equation (6.1), to obtain the most macroscopic time division and thus a related spatial efficiency measure (equation (6.2)).

$$
\begin{aligned}
T_{sf} &= \text{Total duration of a slotframe} \\
T_{up} &= \text{Duration of the uplink phase} \\
T_{down} &= \text{Total duration of the downlink phases} \\
T_{d} &= \text{Total duration of the data timeslots} \\
T_{sf} &= T_{up} + T_{down} + T_{d}
\end{aligned}
\tag{6.1}
$$

$$\eta_{space} = \eta_{pkt} \cdot \frac{T_d}{T_{sf}} \tag{6.2}$$

6.1 Uplink

Each uplink packet will need to travel to a defined node of the previous hop, towards the master. This would obviously take some time and different steps, based on where the node is positioned. Therefore, given that each node's uplink data transmission requires t_{up} time, the best case is a star topology, in which this phase requirement is shown in equation (6.4) while the worst case is a line topology (i.e. each node is connected with two nodes, but the master and the last are connected to one node only), in which the required time is displayed in equation (6.5). This is because each message needs to travel all hops separating it from the master, number that goes from 1 to the maximum hops number. An illustration of this is displayed in figure 6.1. So, this information flow can't obviously happen contiguously, generating a trade off between control overhead and responsiveness of the network to topology changes. A complete uplink phase, in the sense that every node's information reaches the master node, will be spanned over multiple slotframes. The proposed structure makes every node transmit a defined number of packets per slotframe. The information is delivered to a well chosen node of the previous hop, which will be in charge of forwarding the information up to the master node. The duration of the uplink phase is calculated in equation (6.3). As evinced in equation (6.6), the less uplink packets are used when calculating the spatial efficiency of the uplink phase, the higher is the spacial efficiency ratio. Moreover, this leaves more space to manage data,

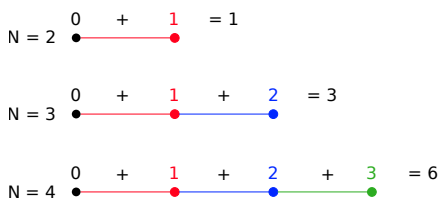


Figure 6.1: Worst case evaluation of the total number of hops that messages should pass in order to reach the master, given that one message is sent by each node

making data spatial efficiency higher by diminishing the protocol control part, which is already big since it's a fully deterministic protocol. Thus, keeping the uplink packets count per node to one unless really necessary is advisable.

N = Number of maximum addressable nodes

t_{proc} = Time for processing the packet

t_{start} = Time for making the first node prepare its packet

$\#_{up}$ = Number of uplink packets sent by each node in each slotframe

$$T_{up} = t_{start} + N \left(\sum_{i=1}^{\#_{up}} \left(T_{pkt}(d^i_{data}) + t_{proc} \right) \right) \tag{6.3}$$

$$T^{min}_{up} = t_{up}(n_{nodes} - 1) \tag{6.4}$$

$$T^{max}_{up} = t_{up} \frac{n_{nodes} \cdot (n_{nodes} - 1)}{2} \tag{6.5}$$

$$\eta_{up} = \eta_{pkt} \left(\sum_{i=1}^{\#_{up}} \left(d^i_{data} \right) \right) \tag{6.6}$$

How to forward each uplink content data is left to each field. So, each packet will contain the hop to which the node belongs, the assignee, which is the node of the previous hop that will be in charge of forwarding the contained data, the topology data and the stream management information. Anyway, which content will be forwarded is up to the uplink management, to assure fairness among different nodes and different kinds of data to forward. Fairness is obtained through the usage of a FIFO queue with allocation optimization, meaning that, the first information arrived is also forwarded. There is however the possibility that the first topology information in queue can't fit the available space, while another behind it does. In this case an exception is made, adding the latter to the message, in order to optimize channel utilization and throughput. Its dimension is calculated in equation (6.7) and a graphical representation is shown in figure 6.2.

H = Number of maximum manageable hops

D_{nd} = Dimension of neighbor discovery data in bits

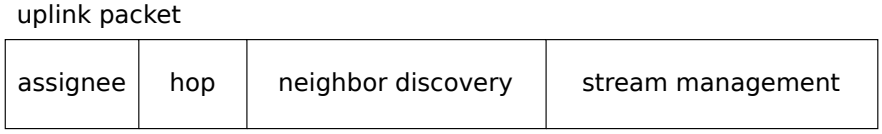


Figure 6.2: Content of an uplink packet

D_{sm} = Dimension of stream management data in bits

$$D_{up} = log_2 N + log_2 H + D_{nd} + D_{sm} \tag{6.7}$$

Uplink transmissions happen in an ordered way, starting from the last node in the address space and arriving to the first. This choice was thought to deal with networks in which the master has address 0, the nodes belonging to the first hop have low addresses, and, while the number increases, the distance from the master node, measured in hops, increases too. This is not a rule in the sense that it limits the network functioning in case it is not enforced, but a rule of thumb for fast network convergence (when a node with an higher address connects to the network as a leaf, it is added as soon as it shows up during an uplink phase, otherwise it would take more slotframes). Each node needs to listen to every other's node transmission. However, since it has been achieved a very accurate synchronization, it can listen up to the time of expected arrival to which a receiving window's tolerance is summed. The resulting minimum interval the transceiver awaits a packet is highlighted in cyan in figure 6.3. It contains not only the preamble and the Start Frame Delimited (SFD), but also a receiving window, computed by the FLOPSYNC-2 algorithm. The error, which is used during the time synchronization downlink for the algorithm functioning, is highlighted as the difference between the expected end of the SFD line, in red, and the actual end of the SFD, in black. At the hardware level, the SFD is taken into consideration as receiving time reference, making the appropriate calculations, since it's the time in which the correlator can determine if a packet is arriving or not. In the case in which a In this way, a node can go into deepsleep mode for most of the time, a noticeable amount of energy can be saved. Notice that, for functionality purpose, an upper bound of acceptance for the receiving

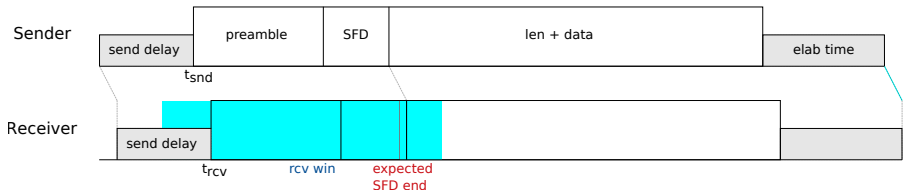


Figure 6.3: Example of node sending an uplink message

window should be configured, making the nodes achieve a minimum level of synchronization before operating the MAC, but also keeping tolerance times low to achieve a higher throughput.

6.1.1 Topology collection

The topology collection aim is to make the master node aware of the actual network composition, of the way nodes are linked and data can travel among them. Then these data need to be transmitted and carried over to the master. Each node, in its uplink turn, will transmit its known topology. By hypothesis, every node will receive an uplink packet from at least a node belonging to the previous hop, called predecessor. However, it is possible that a node has more than one predecessor, requiring each node to choose wisely to which predecessor the uplink message will be assigned. This is accomplished by collecting the Received Signal Strength Indication (RSSI) of the neighbors of the previous hop.

Said so, the topology information transmitted up to the master node will have to contain only the information that is really needed to be known by the master, avoiding information overhead. This, anyway, depends on what topology is desired. Two topologies will be provided, for giving to the user more flexibility about trade-offs.

Mesh

Given a root node, or master node using the protocol terminology, in a mesh topology there can be more different paths to the master, while in a tree topology the path can be only one. In this case the master needs to know each node's adjacencies, by collecting each one's neighbors. A neighbor is any node whose

uplink message, during its relative turn, can be received by the node in object. Also, this information is useful to collect knowledge about nodes closer to the master and elect the one to which the uplink message will be assigned. The method to transmit them easily is by providing each node's neighbor list. The chosen representation is a bitmask $[node_0 \dots node_{m-1}]$ in which every value will be 0 but the bits representing neighboring nodes. The dimension of a node's adjacencies that will need to travel to the master is displayed in equation (6.8).

d_{nd}^m = Dimension of mesh neighbor discovery node's data in bits

$$d_{nd}^m = N \tag{6.8}$$

By receiving this information from all the nodes, the root node will be able to rebuild the complete network graph. To do so, forwarding must be implemented. The complete topology packet contains the sender's adjacencies and the forwarded topologies with their count. Its count is necessary for the receiver to reconstruct the list of forwarded topologies coherently, avoiding the possibility to misinterpret the content. Each forwarded topology is composed of its original sender and the adjacencies list. The complete mesh topology information transmitted dimension is calculated in equation (6.9). The topology part of the uplink packet will be composed as of figure 6.4. A representation of this topology is illustrated in figure 6.6.

D_{nd}^m = Dimension of mesh neighbor discovery data in bits

f_t = Actual number of forwaded

F_t = Maximum number of forwadable topologies

$$D_{nd}^m = d_{nd}^m + log_2 F_t + f_t(log_2 N + d_{nd}^m) \tag{6.9}$$

Tree

A simpler alternative to the mesh topology is a tree topology. This topology can be easily collected by making the master node aware of the path each node needs to follow to reach the master. However, is not the full path that needs

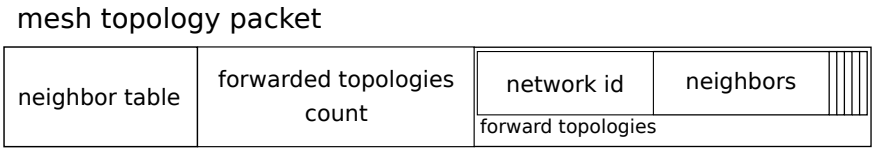


Figure 6.4: Mesh topology part content of the uplink message

to be transmitted up to the master, but just the node to node links represented by a couple of addresses. Its dimension is shown in equation (6.10).

d^t_{nd} = dimension of tree neighbor discovery node's data in bits

$$d^t_{nd} = 2log_2 N \tag{6.10}$$

In this way, it results easier for nodes to forward other nodes' information, which translates to a faster converging network, especially when crowded with devices. However, this method lacks of redundancy, which in critical environments can be mandatory. Moreover, the path each packet needs to follow to reach a node located in the last hop is statically decided by the node itself, leaving to the chosen predecessor all the load of delivering its packets. This makes it use more energy and therefore discharge his batteries faster. Anyhow, in the IOT environments where nodes can be moved or conditions around them can change or when there are few interconnections among nodes making redundancy paths useless, this topology can result more effective. The actual packet will contain just the forwarded links and their count, for parsing purposes. The complete dimension, including the forwarded data can be observed in equation (6.11). A graphical representation of the packet is displayed in figure 6.5. An illustration of this topology is illustrated in figure 6.6.

D^t_{nd} = Dimension of tree neighbor discovery data in bits

$$D^t_{nd} = log_2 F_t + f_t d^t_{nd} \tag{6.11}$$

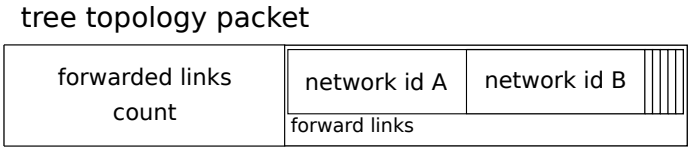


Figure 6.5: Tree topology part content of the uplink message

Dimensioning

Based on the network dimensions, the topology part of the message has different sizes. Obviously, the number of streams that are to be opened or closed is not predictable. Therefore, in this dimensional analysis, it's considered that streams management is achieved in another packet and 2 uplink packets will be sent by each node per slotframe. With these hypothesis, an analysis has been performed. The equations regulating occupancy in terms of forwardable topologies and remaning space depend only on N and H, as evinced in equations (6.12) and (6.15). The equations are evaluated in bits, however this choice does not grant any noticeable space efficiency. An example calculation is performed in table 6.1 and 6.2. However, operating with bitwise requires a noticeably higher number of operations, making space efficiency very costly if compared to calculation throughput. Moreover, the remaining space can be used to allocate the stream management part, hence the use of the byte-aligned version is recommended. Choices marked with an *'*'* should use less forwardable slots in order to accomodate slots management in the same uplink packet. This consideration about byte-aligned fields has been performed for the topology collection part of the uplink message, but is also valid in general, even if in certain cases the trade off may be more advantageous for the choice of the unaligned version.
Also, there is a noticeable spatial convenience in using the tree topology, so the adoption choice will be left to the implementation, based on the actual context and necessities.

$$\begin{aligned} \widehat{D^t_{nd}} &= 1000 - (log_2 H + log_2 N) \\ \widehat{f^m_t} &= \left\lfloor \frac{\widehat{D^t_{nd}} - N - log_2 \widehat{f^m_t}}{log_2 N + N} \right\rfloor \end{aligned} \tag{6.12}$$

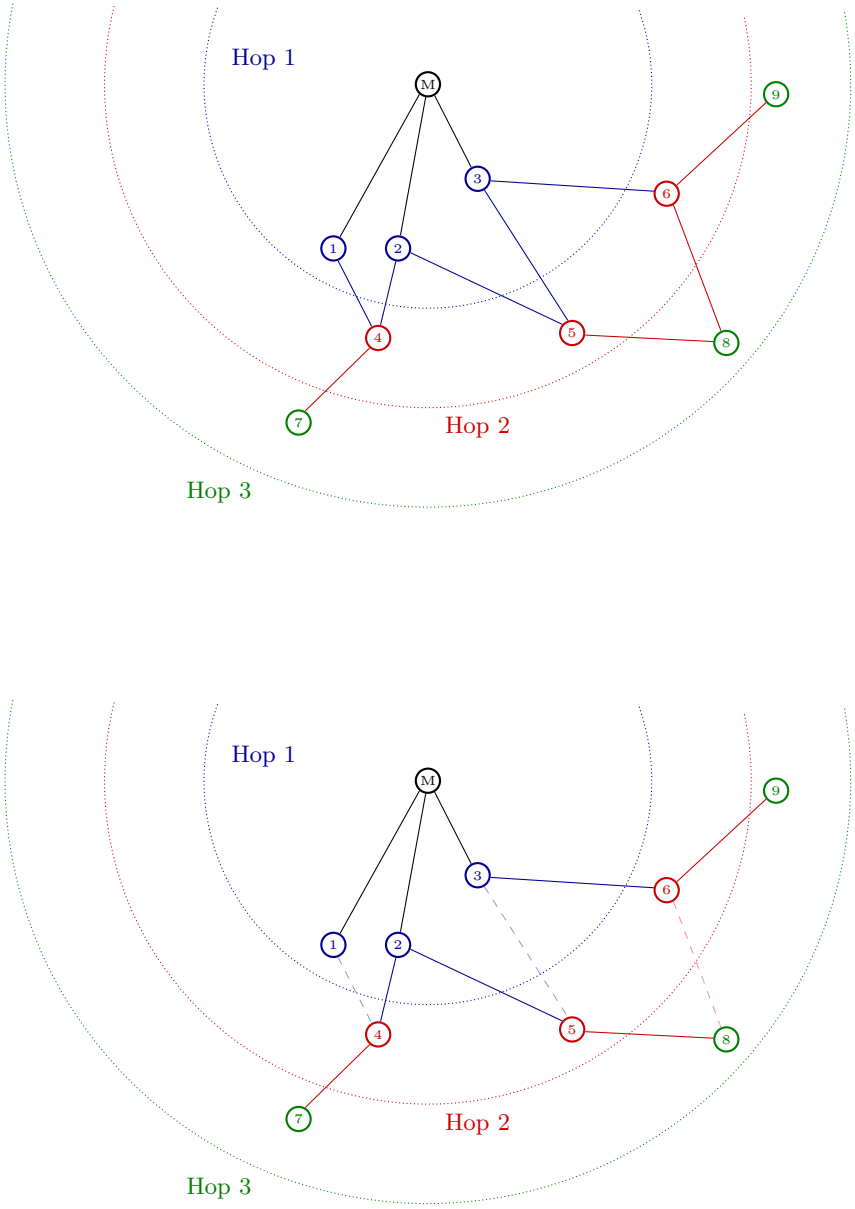


Figure 6.6: The mesh (top) and the tree (bottom) topologies obtained from the same nodes layout

Max hops	Max nodes	Byte aligned			
		Mesh		Tree	
		Fw.able	Rem. bits	Fw.able	Rem. bits
7	8	6˚	872	6	880
15	16	14˚	632	14	752
31	32	23	24*	30	496
63	64	12	48*	61	0*
127	128	6	32*	61	0*
255	256	2	192	61	0*
256	512	NA	NA	30	16*
256	1024	NA	NA	30	16*
256	2048	NA	NA	30	16*
256	4096	NA	NA	30	16*

Table 6.1: Topology collection space occupancy in byte-aligned packet management for both the topologies. *˚ maximum number of forwardable topologies achieved. * not enough space left for stream management.*

$$\widehat{r_t^m} = \widehat{D_{nd}^t} - \widehat{f_t^m}(log_2 N + N) \tag{6.13}$$

$$\widehat{f_t^t} = \left\lfloor \frac{\widehat{D_{nd}^t} - log_2 \widehat{f_t^t}}{2log_2 N} \right\rfloor \tag{6.14}$$

$$\widehat{r_t^m} = \widehat{D_{nd}^t} - \widehat{f_t^m}(2log_2 N) \tag{6.15}$$

6.1.2 Stream management

In the stream management part of the uplink control messages, streams can be opened, closed, or their minimum required data rate modified. Requirements about the data rate value dimensions and meaning are left to the scheduler implementation. Its only constraint is that a zero-valued element will have the meaning of a request to close the stream. Another constraint is that the streams are unidirectional, so to open a symmetric stream, two stream management elements (SMEs) must be sent. A SME will contain the address of the source node, the address of the destination node and the already named data rate. The dimension of each element is calculated in equation (6.16) and a graphical representation of it is presented in figure 6.7. The stream management part, which is the last part of the uplink message, will be made up by an enumeration of SME. The count of transmitted elements would be redundant, since its length

Max hops	Max nodes	Byte aligned			
		Mesh		Tree	
		Fw.able	Rem. bits	Fw.able	Rem. bits
2	8	6˚	919	6˚	957
2	16	14˚	695	14˚	879
2	32	25	32*	30˚	689
2	64	13	15*	62˚	243
2	128	6	51*	70	5*
2	256	2	206	61	9*
2	512	NA	NA	54	12*
2	1024	NA	NA	49	3*
2	2048	NA	NA	44	14*
2	4096	NA	NA	40	21*
4	8	6˚	918	6˚	956
4	16	14˚	694	14˚	878
4	32	25	31*	30˚	688
4	64	13	14*	62˚	242
4	128	6	50*	70	4*
4	256	2	205	61	8*
4	512	NA	NA	54	11*
4	1024	NA	NA	49	2*
4	2048	NA	NA	44	13*
4	4096	NA	NA	40	20*
8	8	6˚	917	6˚	955
8	16	14˚	693	14˚	877
8	32	25	30*	30˚	687
8	64	13	13*	62˚	241
8	128	6	49*	70	3*
8	256	2	204	61	7*
8	512	NA	NA	54	10*
8	1024	NA	NA	49	1*
8	2048	NA	NA	44	12*

8	4096	NA	NA	40	19*
16	16	14˚	692	14˚	876
16	32	25	29*	30˚	686
16	64	13	12*	62˚	240
16	128	6	48*	70	2*
16	256	2	203	61	6*
16	512	NA	NA	54	9*
16	1024	NA	NA	49	0*
16	2048	NA	NA	44	11*
16	4096	NA	NA	40	18*
32	32	25	28*	30˚	685
32	64	13	11*	62˚	239
32	128	6	47*	70	1*
32	256	2	202	61	5*
32	512	NA	NA	54	8*
32	1024	NA	NA	48	19*
32	2048	NA	NA	44	10*
32	4096	NA	NA	40	17*
64	64	13	10*	62˚	238
64	128	6	46*	70	0*
64	256	2	201	61	4*
64	512	NA	NA	54	7*
64	1024	NA	NA	48	18*
64	2048	NA	NA	44	9*
64	4096	NA	NA	40	16*
128	128	NA	NA	69	13*
128	256	NA	NA	61	3*
128	512	NA	NA	54	6*
128	1024	NA	NA	48	17*
128	2048	NA	NA	44	8*
128	4096	NA	NA	40	15*

256	256	NA	NA	61	2*
256	512	NA	NA	54	5*
256	1024	NA	NA	48	16*
256	2048	NA	NA	44	7*
256	4096	NA	NA	40	14*
512	512	NA	NA	54	4*
512	1024	NA	NA	48	15*
512	2048	NA	NA	44	6*
512	4096	NA	NA	40	13*

Table 6.2: Topology collection space occupacy in unaligned packet management for both the topologies. ˚ *maximum number of forwardable topologies achieved.* * *not enough space left for stream management.*

can be deduced by the receiver as shown in equation (6.17). SMEs are managed using an updatable FIFO queue, which keeps the packets FIFO timeliness but updates the content if a packet from the same sender regarding the same stream ID is received, to achieve fairness and minimize delays. SMEs transmission starts from the source node. What it does, is enqueuing its message to its local stream management queue and wait until it's sent, repeating the operation in every slotframe until the effect is noticed in the schedule downlink message, whose structure and function is enunciated in 6.2.2. Otherwise, when nodes receive uplink messages from others, every received SME is added to stream management queue, respecting its internal mechanisms previously illustrated.

d_s = Size of a stream management element in bits

f_s = Sent stream elements

d_{dr} = Dimension in bits of the data rate value

L = length byte in the PHY header

$$d_s = 2log_2N + d_{dr} \tag{6.16}$$

$$f_s = \frac{8L - (log_2N + log_2H + D_{nd})}{d_s} \tag{6.17}$$

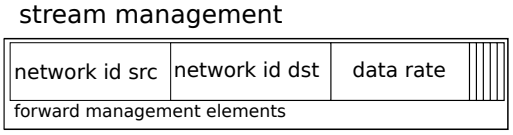


Figure 6.7: Stream management part content of the uplink message

6.2 Downlink

Each downlink packet will be flooded, using the constructive interference mechanism explained in 5.1.1. This makes possible to reach quickly all the network nodes, making the entire network aware of the scheduling that needs to be used. This process takes T_{down} as calculated in equation (6.18), having a $t_{retransm}$ retransmission interval among the hops, that must be tuned considering the packet receiving time and the operative time for switching the radio operational mode. There are more kinds of contents that can be transmitted using the downlink and are explained in the sections to follow. It must be kept in mind, though, that downlink packets need to have all the same payloads for being capable of leveraging the capture effects in a constructive way. In the protocol, two types of downlink packets are implemented: time synchronization and schedule distribution. An example of downlink message flowing through 3 hops, with the synchronization error explicited is shown in figure 6.8.

$$
\begin{aligned}
t_{down} &= \text{Duration of a downlink phase} \\
t_{retransm} &= \text{Time needed for retransmitting a packet} \\
t_{down}(d_{data}) &= t_{start} + H\left(T_{pkt}(d_{data}) + t_{retransm}\right) \\
T_{down} &= t_{down}^{timesync} + t_{down}^{sched}
\end{aligned}
\tag{6.18}
$$

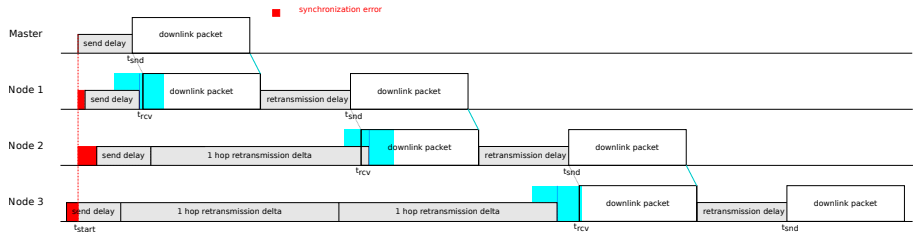


Figure 6.8: Example of downlink message flooding

6.2.1 Time synchronization

This kind of downlink packet is used to implement FLOPSYNC-2 making it possible to achieve synchronization and thus being able to save energy. Synchronization leads to energy saving by making it possible to have a very small receiving window and low risk to loss an expected packet or to transmit in some other node's round, as pointed out in 5.1.1. This packet is the most similar to a 802.15.4 beacon and, to make it understandable for other devices, which would understand that this channel is already occupied by devices using another protocol, it starts with a 802.15.4 compatible header. This is also handy because it makes the nodes using this protocol aware that it is a flooding packet. It starts with a 2 bytes frame control sequence stating that this is a reserved type intra-pan frame, with no source addressing and destination addressing only. The next byte, which is by standard the sequence counter, is used for the source's hop count, which will be used by nodes joining the network to obtain mathematically the actual time at which the current slotframe started and hence calculate all the time points correctly. Then the 2 bytes of pan ID are left unused and the destination address of 2 bytes set to broadcast (65536). The space from byte 10 to byte 26 contains the current timestamp of the master. Finally, the last $\#_{rtt} \cdot log_2 N$ packet bits, are reserved to state which will be the nodes enabled to perform the reverse flooding algorithm in the sudden part of the phase. Its complete duration is therefore calculated in equation (6.20). $\#_{rtt}$ must be defined a-priori while configuring the network and shared among all the nodes.
So, the nodes that can be reverse by flooding initiators for the slotframe are chosen by the master. It needs to choose them accurately, preventing two nodes from asking the roundtrip calculation to the same node, leading to an unde-

fined behavior. Also, a node that performs reverse flooding calculation that has another node doing the same in its predecessor set would not obtain an accurate estimation. These two cases need to be avoided by the master node when choosing. After being chosen, nodes will be able to calculate their roundtrip time to the master node, in order to better align its timer and achieve a more accurate timestamping during the current slotframe. This gives to the nodes the possibility to sleep in order to save energy, since their neighbors in the successive hop are not listed in the time synchronization downlink message. Attention must be payed in this part, because in a tree topology, a node should listen not only to its direct children, but to every node receiving its flooded messages, i.e. the nodes whose uplink message is received. In case a node hasn't already measured its RTT to the master, it will not be able to reply with a correct estimation, so it won't be listening for any RTT request. A part of the slotframe is reserved to the nodes willing to perform the roundtrip estimation after the time synchronization downlink. This phase is displayed in figure 6.9. The

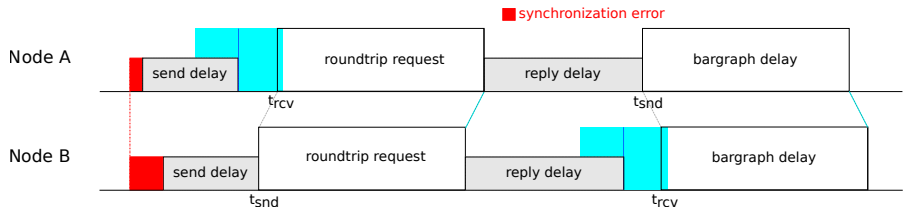


Figure 6.9: Example of roundtrip request and response

packet composition is of 3 bytes, the first 2 to identify, that is, to ask for the roundtrip calculation, and the last for the hop count. If such a packet is not received within the expected receiving time and the receiving window calculated by FLOPSYNC-2, the receiving nodes goes into sleeping mode to save energy. Otherwise, a reply bargraph-encoded packet, as explained in [35], is sent back. The duration of a roundtrip estimation with reverse flooding is calculated in equation (6.19). A graphical representation of this process with the constructive interference is displayed in figure 6.10. The flooding is represented in blue. Then, supposing that N4 is listed in the last bytes of the flooding packet and it is interested in measuring its RTT to the master, in the dedicated part of the slotframe, it sends the RTT measurement request, colored in red. Then, all

the nodes of the previous hop within its radio range reply with the bargraph encoded delay to the master, in green. Hence this makes N4 able to estimate itself the delay to the master.

$$T_{rf} = \text{Duration of a reverse flooding}$$

$$T_{down}^{timesync} = \text{Duration of the downlink time synchronization phase}$$

$$T_{rf} = t_{start} + T_{pkt}(3) + t_{retransm} + T_{pkt}(127) \tag{6.19}$$

$$T_{down}^{timesync} = t_d \left(9 + \left\lceil \#_{rtt} \cdot \frac{log_2 N}{8} \right\rceil\right) + T_{rf} \tag{6.20}$$

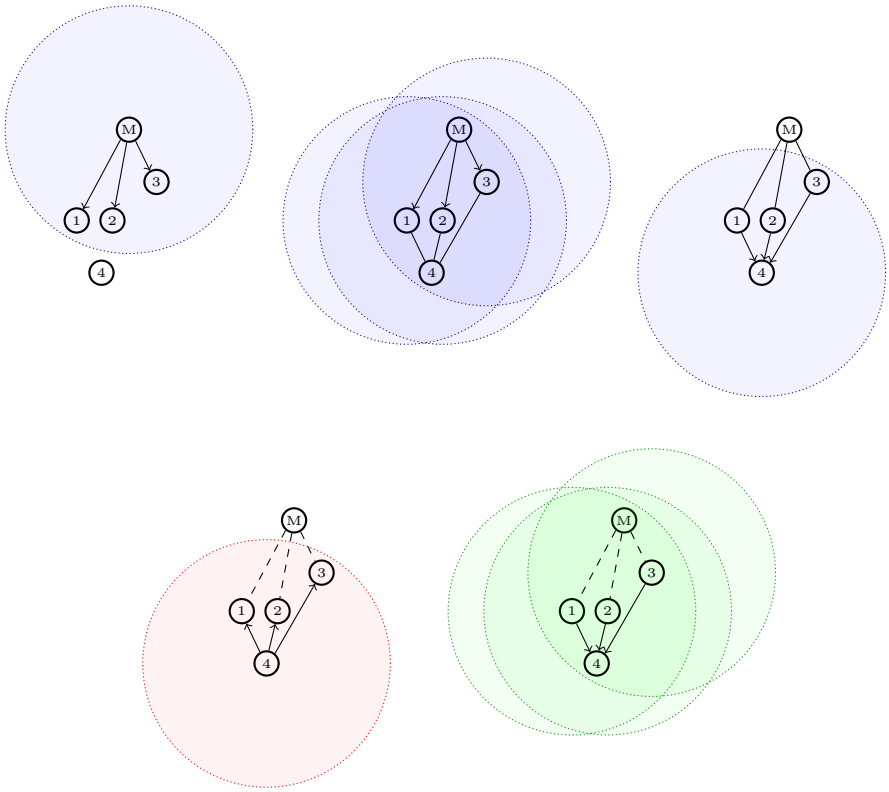


Figure 6.10: Time synchronization process

6.2.2 Schedule distribution

The schedule, once is computed by the master node, needs to be distributed. Since a fast downlink process is available, the central scheduler option has been

chosen and therefore the schedule will be broadcasted with flooding. However, the choice and design of the scheduler and how the schedule is computed is not treated in this thesis. Instead, it is managed as a black box, taking the topology map and the required streams as inputs, producing a schedule in terms of nodes and timeslots as output. Then, the distribution of such output among the nodes is analyzed. Supposing to have an implementation of the protocol with the reasonable values of $10ms$ per data slot and $10s$ for each slotframe, excluding the control part, the available data slots are 1000. Supposing that a fixed number of transmissions in each data slot is allowed entails a result in bits calculated in equation (6.21), needed for the distribution of a complete schedule, for each data slot and each transmission in it. Of course, this value is not manageable since 4kB of schedule (32 packets) would need to be transmitted, in order to send a schedule in a network with 256 nodes capacity and 65536 manageable slots, requiring $134ms$, per-sending, according to equation (4.2). In a network of 8 hops, it would need more than $1s$ meaning more than 10% of the slotframe, considering that other control parts were not included in the calculation. This also excludes the possibility for two nodes to transmit contemporarily, leading to a poor network throughput. Moreover, before the schedule is able to change completely, all the schedule, so all the 32 packets, need to be flooded.

Another possibility is to send the schedule enumerating it by stream and thus having a variable length schedule based on the number of currently active streams. This makes the streams, unless a stream ID is managed, *"anonymous"*, but every node, in receiving a stream's schedule can understand the correspondence by looking at its source and destination nodes. The constraint added is that two streams with the same source and same destinations can't exist. However, this is applied only at MAC level, meaning that at the transport level, if two sockets need to be opened, the data rate can be adapted. The adaptation to be done depends on the scheduler adopted and the meaning that it will give to the data rate value. This can lead to the allocation of slots that will not be used by the node, though. This happens, because the requested value must be strictly met to keep the data rate fixed among every two successive transmissions of the upper layer stream, leading to the impossibility of simply summing the values. Although the overhead reduces when the rates are

equals or have a common divisor different from 1 (i.e. they have a common base frequency and each one has its own multiplier). However, a significant advantage of this distribution function with respect to the previous is that no limit is imposed on the number of transmission in each data slot. In this case the needed bits are shown in equation (6.22).
It is pointless to enumerate also the unallocated timeslots in the schedules. Therefore, instead of listing all the timeslots for every opened stream, it is just necessary to state the stream source and the number of hops to travel, and list at what time each node in the path must expect it (i.e. the node that received it previously needs to retransmit it). The schedule dimension calculation is accomplished in equation (6.23). Also, it should be noticeable that $h(s) \leqslant 2 \cdot H$, in the worst case in which a node of the last hop needs to reach another node of the last hop without any parents of any level in common but the master node.
A simpler alternative, that reduces the schedule space allocation, fixes the global minimum data rate and considers data rate increments only as its multiples. It can be represented using tuples containing the stream source's address, the count of traveled hops, the data rate multiplier (represented by simplification with N_s, since it is an upper bound) and the couples receiver - instant, similar to the previous structure. An initial value specifying the minimum data rate is also needed. Its space occupation, measured in bits, is illustrated in equation (6.24).
By comparing all the equations, deductions can be made.

$\#_{ds}$ = Data slots in each slotframe

$\#_{tps}$ = Transmissions per data slot

S = Maximum number of managed streams

S_o = Number of opened streams

$h(s)$ = Hops traversed by the stream

$\overline{h}_s$ = Average hops traversed by a stream

$n(s)$ = Number of times the stream is executed in the current schedule

$\overline{n}_s$ = Average number of times the stream is executed in the current schedule

N_s = Number of times a stream can travel from the source to the destination in a schedule

$$D_{sched1} = \#_{ds} \cdot \#_{tps}(2log_2N + log_2S) \tag{6.21}$$

$$D_{sched2} = S_o \cdot \#_{ds} \cdot 2log_2N \tag{6.22}$$

$$D_{sched3} = \sum_{s=0}^{S_o<S} (log_2N + log_22H + n(s)h(s)\,(log_2N + log_2\#_{ds})) \tag{6.23}$$

$$D_{sched4} = log_2N_s + \sum_{s=0}^{S_o<S} (log_2N + log_22H + log_2N_s + h(s)\,(log_2N + log_2\#_{ds})) \tag{6.24}$$

$$D_{sched2} < D_{sched1} \iff S_o < \#_{tps}\left(1 + \frac{log_2S}{2log_2N}\right) \tag{6.25}$$

$$D_{sched3} < D_{sched2} \iff log_22H + \overline{h}_s \cdot \overline{n}_s \cdot log_2 \cdot \#_{ds} < log_2N\left(2 \cdot \#_{ds} - \left(1 + \overline{h}_s \cdot \overline{n}_s\right)\right) \tag{6.26}$$

$$D_{sched4} < D_{sched3} \iff log_2N_s\left(1 + \frac{1}{S_o}\right) < \overline{h}_s\,(\overline{n}_s - 1)\,(log_2N + log_2\#_{ds}) \tag{6.27}$$

In equation (6.25) it is shown under which conditions the second format can be potentially better than the first. By knowing that usually $S \geqslant N$, since a node should be able to open at least a stream, the more are the non-opened streams, the better is the second format. Also, as it could be easily imagined, more operable streams in the same data slots lead to a higher length of the first schedule format. So, the second format is more convenient, unless there are few slots per slotframe and not many streams per node. Moreover, the first one has no flexibility about the number of allocable streams. By comparing the third proposed schedule with the second, as proposed in equation (6.26), it can be noticed that the former is less greedy of space when there are more data slots per slotframe and when the average transmissions per data slot over opened streams ratio is high, so when there is more parallelism. The modification adopted in equation (6.27) is quickly analyzed since there is an improvement only if the number of bits to represent the base data rate and each stream's multiplier is lower than the number of bits needed to represent the average schedule repetitions in

terms of data slot and destination address. This is therefore advantageous in case of schedules sharing a common base frequency, repeated many times during a slotframe. At this point, the values considered in the first example turn being meaningless, since the presented schedule distribution functions become more and more parametric and hence their length variable. The best way to use these values, finding the best combination in different configurations of the protocol, evaluating the drawbacks, is just a matter of a choice, which can be performed by picking the most appropriate function case by case and plugging it into the protocol.

A completely different approach is transmitting a delta-encoded schedule. In this case, every time a SME is received or when the topology changes significantly, a new schedule is computed. The differences between the new and the old schedules will be transmitted using a tuple which contain an addition/removal bit, the stream number and the entire path, with its modified parts. This last part is represented as a count of the hops followed by the addresses of the path and the related dataslots. This solution would be efficient in cases where the scheduler tries to minimize the number of modified paths. However, it needs to retransmit the whole path for each modified schedule. Other solutions transmitting only the modified parts and correctly changing only such parts can be compared and used. Anyway, since the trade-offs depend mostly on the scheduler, this aspect has been no further developed.

When the schedule is received by the node, it is rebroadcasted and the data is analyzed. Therefore, pending SMEs are checked and, if the current schedule results satisfactory, they stop being transmitted; instead, they are kept in the queue and hence transmitted when possible. Vice-versa, the master node, analyzing uplink's SMEs, checks them with those considered in the active schedule, avoiding to allocate the same stream several times.

Throughout the analysis a single-channel protocol is considered. If a multiple channel implementation of the data part is desired, it must be considered that it will be needed to specify the channel for each communication that will occur in the schedule. Moreover, this analysis is scheduler-agnostic. Based on the needs and the constraints of the scheduler, a deeper exploration can be performed thus achieving better and more suitable results to efficiently distribute the schedule.

6.3 Data transmission

As previously stated, the data part of the protocol, is divided in data slots, which, for simplicity, have a fixed duration. Per-stream timeslot management can be surveyed and considered, since different application types should be capable to use the same protocol at the same time with the most accurate timeslot duration. The time part dedicated to this is calculated in equation (6.28) and thus the efficiency of channel utilization can be deduced in terms of the amount of data that has been treated by the protocol with respect to the amount of data that has been transmitted by it in the channel (equation (6.2)).

N_{slots} = Number of data slots in a slotframe

t_{slot} = Duration of a single slot

$$T_d = T_{sf} - T_{up} - T_{down} \tag{6.28}$$

$$N_{slots} = \frac{T_d}{t_{slot}} \tag{6.29}$$

Except for their duration t_{slot}, timeslots can feature different behavioral modes. An interesting difference is found between acknowledged and non-acknowledged timeslots. In the latter, the connection reliability, if needed, must be addressed in the upper layers. Whereas, in the former, acknowledgments are managed at the MAC layer. This can be seen as a limit to the real-time characteristic of this protocol. However, for certain mission-critical applications it can be useful for an application to know exactly whether a datum has been correctly delivered or not, and hence needs to be transmitted again at the next available timeslot. This, as a side-effect impacts on the slots duration, therefore on the number of slots as can be noticed in equation (6.29) and on the amount of energy spent to send a message.

Another feature influencing the slot duration is the length of the data to be sent. In some environments, few data may need to be transmitted, like sensor values of few bytes. In this case, t_{slot} would obviously decrease, the number of slots available increase and thus granting the possibility to perform frequent transmission decreasing the average delay to the destination, while diminishing

although the η_{space}.
However, in other applications, greater amounts of data may need to be transmitted, for instance when having few nodes for a great number of sensors due to the difficulty to deploy more nodes. On the contrary of the previous case, the delay per packet would be increased, although allowing to reach higher throughputs.

6.4 Interleaving function

In order to be able to grant a constant delay, downlink and uplink packets will need to be interleaved with data timeslots. This is performed by a dedicated interleaving function, whose job is to map uniquely a timestamp with the phase to be executed and vice-versa. The timestamps used by the protocol are called logic, since the actual timestamps, as seen from the point of view of the radio channel, are named physical. Logical timestamping is managed by convention as follows:

1. Time synchronization downlink;
2. Uplink complete phase;
3. Schedule distribution downlink;
4. Data transmission.

The interleaving function needs to be shared among all the nodes in the network and all of them must employ the same one, in order to communicate with each other. The allocation algorithm behind the interleaving function needs to be known by the scheduler too, in order to compute schedules with a consistent delay evaluation. The simplest way to perform this job is by knowing the following configuration parameters:

- T_{sf} the duration of the entire slotframe;
- $T_{down}^{timesync}$ how long does the timesync phase last;
- $t_{down}(d_{sched})$ for each packet, for a total dimension of D_{sched};
- T_{up} and $\#_{up}$ to allocate each node's uplink packet;

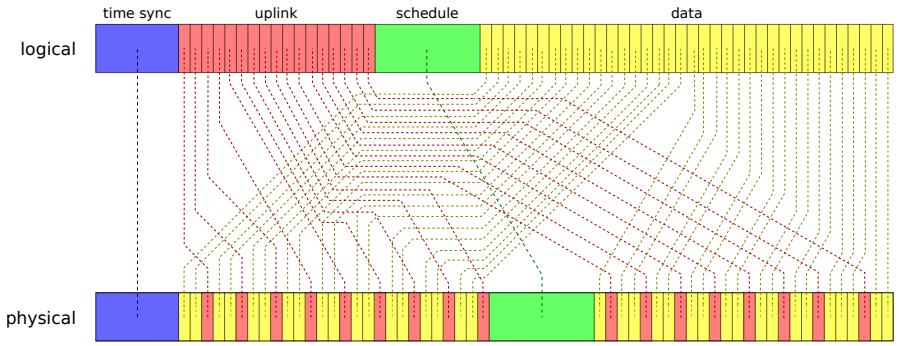


Figure 6.11: Representation of the interleaving function with a 17 uplink slots, single slot schedule downlink and 36 data slots

- t_{slot} the duration of a single data slot.

Phases, for being allocated, need to be divided in slots. Each slot is composed by an elementary radio activity, identified by one or more transmission which can be effortlessly separated from other transmissions of such phase. In uplink phases, each slot is identified by a node to node transmission. Instead, in downlink phases, a single packet flooding from the master to the last hop's nodes is considered as a single slot. In this way, by knowing in proportion how much time of each phase needs to be allocated and the duration of each phase slot, it is possible to build a fair interleaved schedule. This is needed to keep the allocated proportion as close to the final ones as possible while allocating each phase's slot in sequential order. This metric should be used also to choose the next slot to be allocated while building the timeline. Once statically computed, the resulting map should be used as a configuration parameter to be distributed among the nodes. A graphical representation, not considering the effective durations is shown in 6.11.

6.5 Complexive structure

Summarizing the main protocol characteristics, it can be represented as shown in figure 6.12 for what concerns dynamic nodes, and in figure 6.13 from the operational point of view of the master node. From a top-down point of view, the upper layers interface with it using the stream management function and

enqueuing or dequeuing data from their streams. Stream allocation is managed by the dedicated part of the uplink message. Those messages also contain information about the topology that the master node needs to acquire. It also manages other nodes' topologies and Stream Management Elements that have been delegated to it. In order to achieve synchronization and performing an accurate TDMA access, a time synchronization downstream phase is managed. Another downstream phase is used to deal with the schedule for network access in the data slots. All this phases are orchestrated over time by an interleaving function, common to all nodes, which aims to minimize the average delay among each data timeslot. Finally, despite being present only in the master node, the scheduling function is also an important part of the protocol. It receives the interleaving function map, the opened streams information and the collected topologies as input parameters, producing the schedule to be flooded using the dedicated downlink packet.

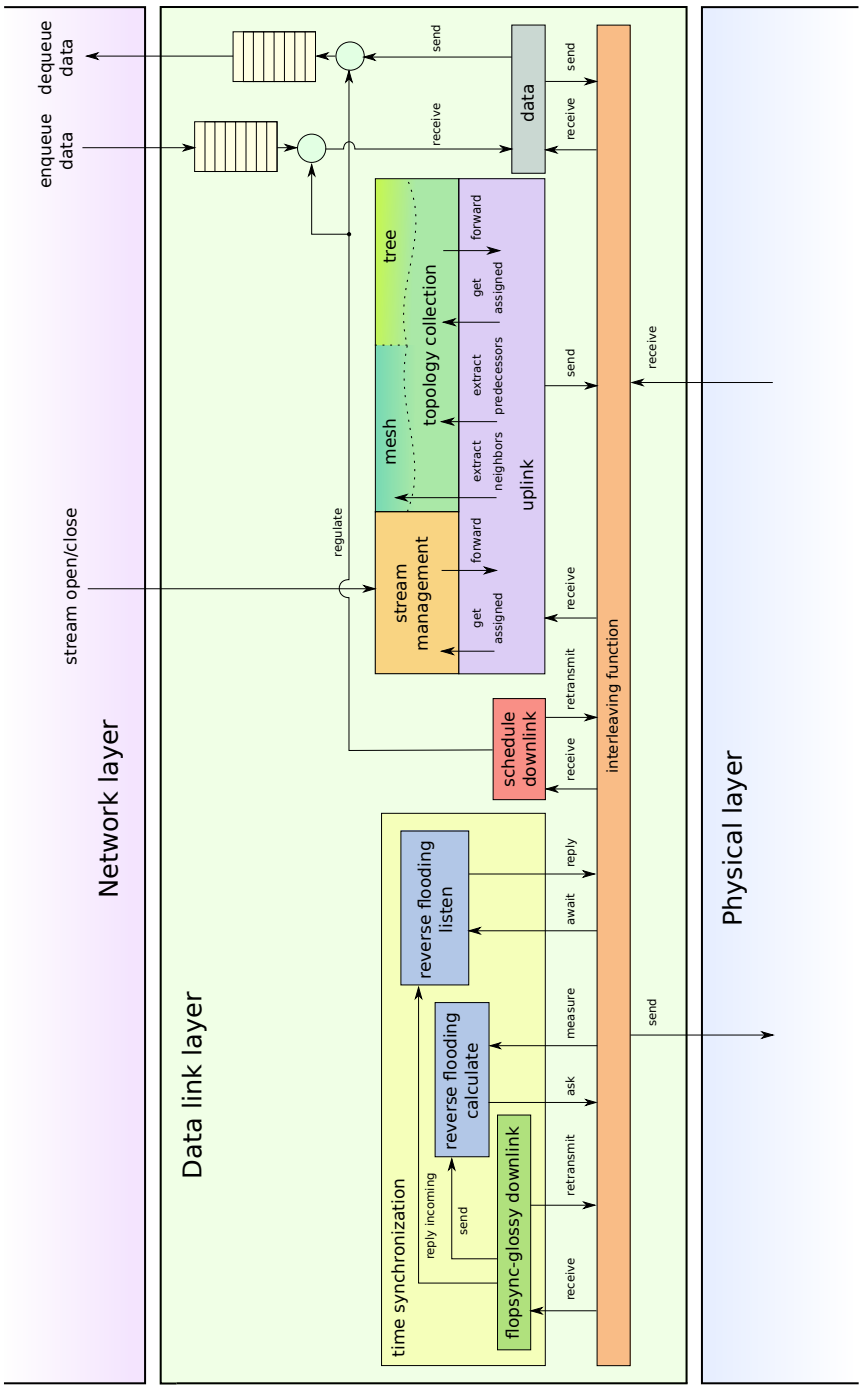


Figure 6.12: Protocol stack components diagram

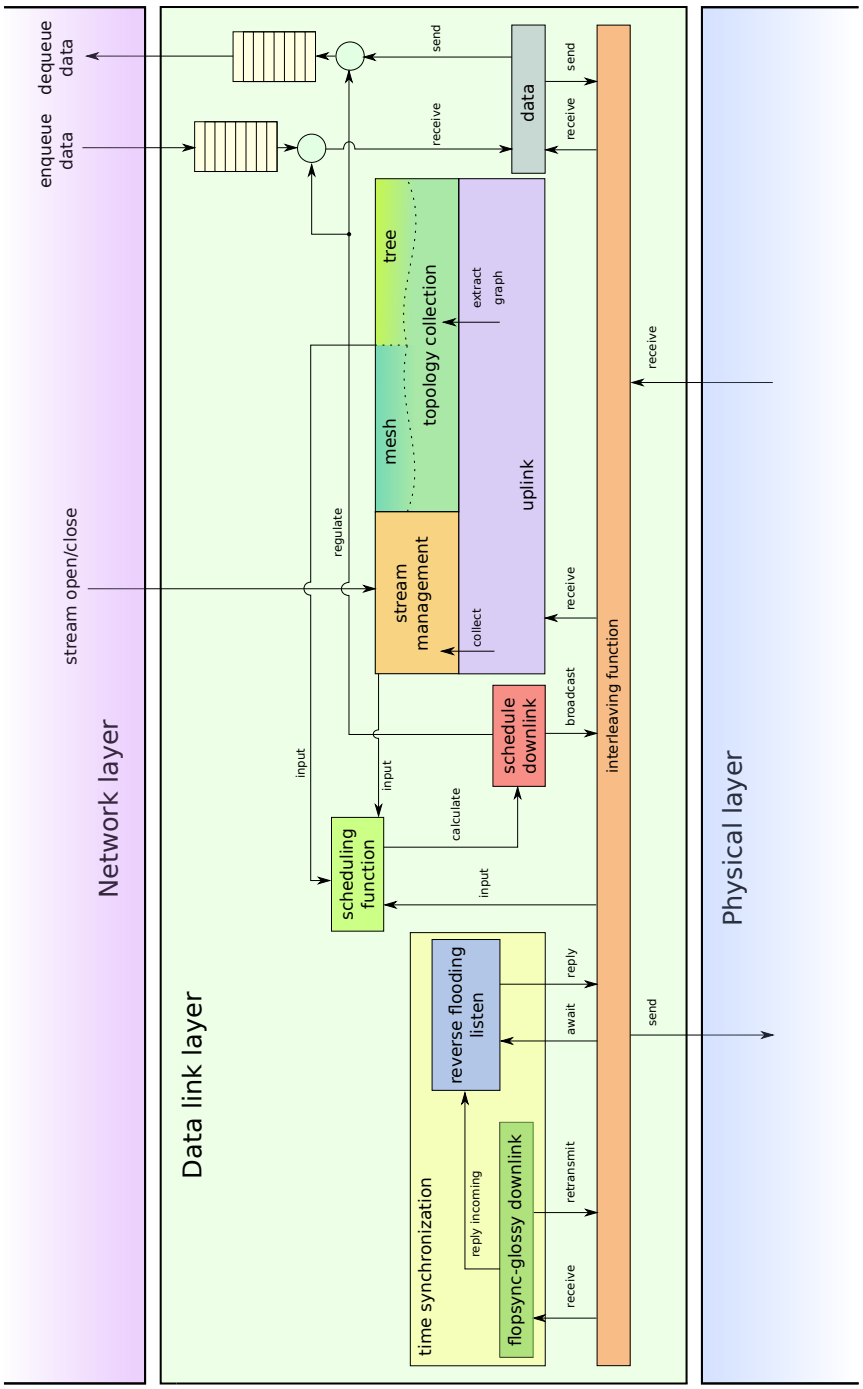


Figure 6.13: Master's protocol stack components diagram

Chapter 7

Software project

For the implementation of the protocol, a software engineering project has been performed. Both the Miosix kernel and OMNeT++ are based on C++, therefore it has been chosen as target programming language. Since its main supported paradigm is the object oriented, the related structuring approach has been used to synthesize the project. So, knowing the protocol structure and its functionalities, the first task to be performed is the outlining of a class diagram, starting from the structure obtained in the figures Upper layers API From the point of view of the upper layers, the MAC protocol needs to expose an API to activate it, with a given *NetworkConfiguration*, containing the parameters to configure the protocol as explained in detail in 6, and a *Transceiver* (ref. 5.3), to be bound to, representing the lower layer of the protocol stack. It also provides methods for opening, closing and managing the data rate of an *OutgoingStream* besides those to accept *IncomingStream* connections, returning also the number of slots per slotframe assigned in the scheduling. These operations can be accomplished in a synchronous blocking way, by creating a SME that will be sent during the stream management phase and awaiting for the schedule to adapt, or asynchronously, by sumbitting a callback that will be called when the schedule will adapt. This is resumed in the *MediumAccessController* class, defining all these behaviors. It does not implement the singleton pattern, since it is desirable to create more instances of this class in case it will be used on a node with more Transceivers.

The just cited *OutgoingStream* and *IncomingStream* are subclasses of *DataStream*, which is another class of our model. Its main job is to contain packets and thus it includes a queue and methods to manage it by obtaining its size and to flush all the packets contained, in the case of a clogged stream. It also detains a getter accessor for the queue which can be called only once making the queue to be managed by the protocol internally only. The upper layers will be able to access the stream's queue only by its already cited children classes which provide methods for sending and receiving data both synchronously (busy waiting) and asynchronously (with a callback). This is displayed in the class diagram of figure 7.1.

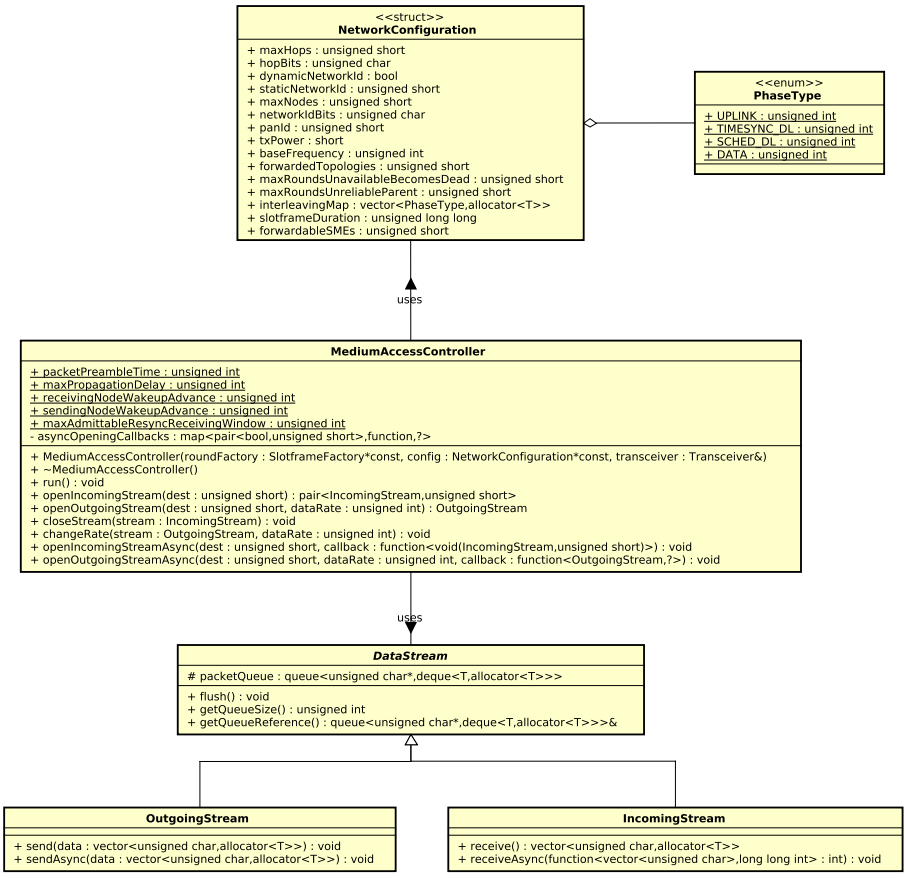


Figure 7.1: Upper layers interface class diagram

7.1 Internal statuses

The *MACContext* class is the main container to keep the status necessary for the protocol and its components to run. It also stores structures to manage the slotframes and the phases of the protocol, whose content will be explained in the next section. Let's analyze now the class content, for what concerns the status management part, as represented in figure 7.2:

1. for back referencing the *MediumAccessController* object;
2. the hop, as will be received during the time synchronization's flooding;
3. networkId, as loaded from the *NetworkConfiguration* at the moment, but considering that in a future, dynamic address assignment can be implemented;
4. syncStatus, to manage the status of the FLOPSYNC-2 time synchronization inside a dedicated class, *SyncStatus*;
5. scheduleContext, representing the current schedule, used both during the schedule downlink phase, for its update, and the data phase, to know what to do slot by slot;
6. topologyContext, which is an abstract class whose derivated class manage, by elaborating the received uplink messages, the content to be sent in the topology part of the uplink message;
7. networkConfiguration, for containing the configuration as passed during the initialization of the *MediumAccessController*;
8. streamContext, which operates over the stream management part of the uplink phase;
9. transceiverConfig, for keeping a base configuration for the transceiver, to be adapted as needed.

Every component's content will be described in the appropriate part, by connecting it with the actual functionalities. Other functional parts of this core class are better explained in the next section.

Figure 7.2: Class diagram for all the classes representing the internal states of each part of the protocol

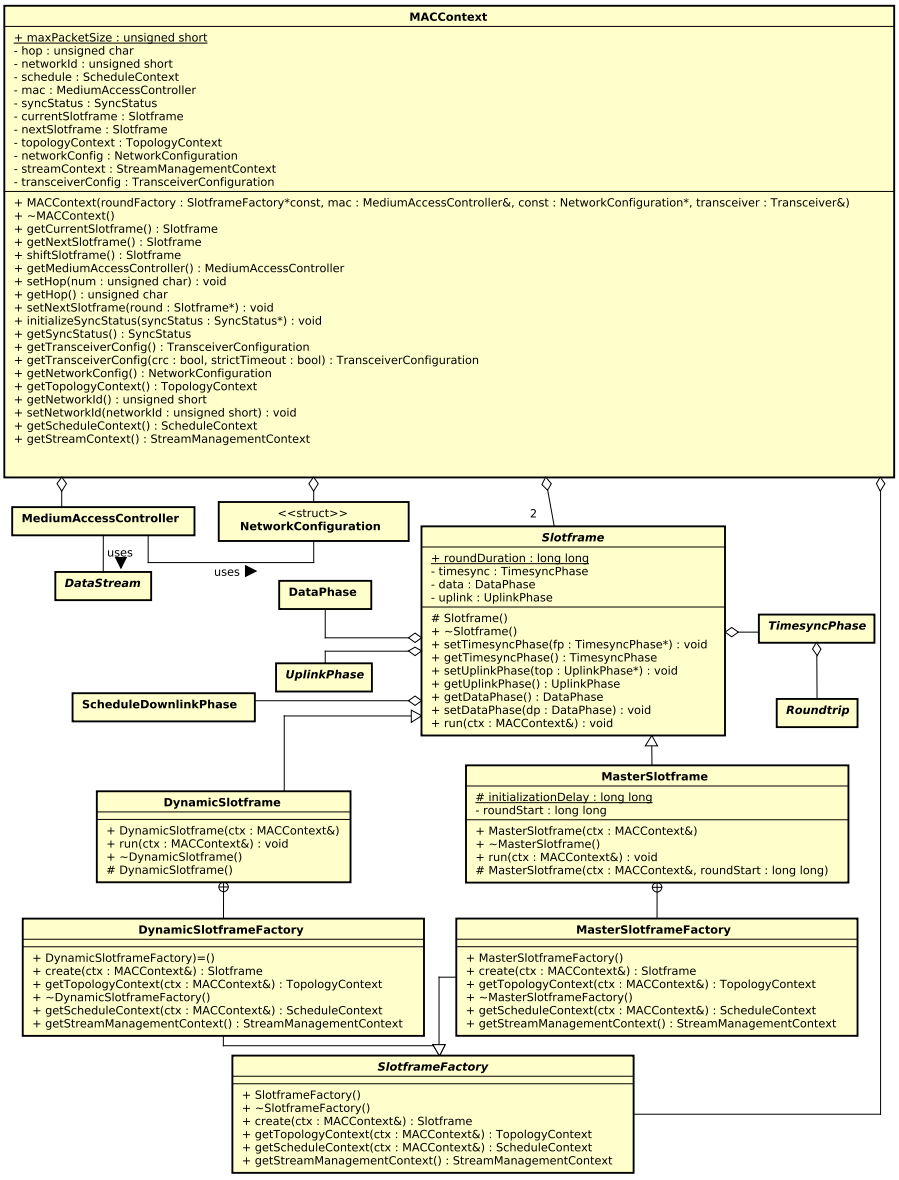


Figure 7.3: Class diagram representing all the classes between the MAC interface and the Slotframe class

7.2 Protocol phases

Proceeding with a top-down approach, the *MediumAccessController* class contains the already known *MACContext* class, detaining the protocol status, its configuration and context. It has been separated to give internal classes a set of operations, which would be confusing and can lead to dangerous effect if exposed to the user. It contains different status-keeping objects and attributes already briefly discussed. However, from the point of view of the operative structure, it contains the current and next *Slotframe*, used to manage what happens in the current and the next slotframes. As the current slotframe finishes, it is switched with the next, and this last one gets reset. This objects are first created when the protocol is started, and with them also the node-type specific status objects. The *Slotframe* contains an instance for each phase, each of them represented by a different class: *TimesyncPhase*, which contains in its turn the *RoundtripPhase*; *DataPhase*, to send and receive data when needed; *UplinkPhase* to manage topology collection and stream management; *ScheduleDownlinkPhase* to receive the schedule, retransmit it and then parse it by suddenly modifying the relatex context. Details and specializations of these classes will be explained in their relative sections to follow.

7.2.1 Timesync downlink

The timesync part has as main job the synchronization of the network and the estimation of the roundtrip to the master. Each protocol phase is represented by a *MACPhase*, a class to represent all the phases of the protocol, exposing an execute method, which accepts the global context and the time at which the next action will be executed. A *MACPhase* lasts for the duration of the *Slotframe* and can repeat its execution once, as for the downlink phases (typically), or vice-versa it can be executed more times to perform slightly different jobs, like the *DataPhase* and the *UplinkPhase*. It also offers a method to know when it ends, so the next *MACPhase* will be temporally following it.

The phase representing the timesync context is the *TimesyncPhase*, which manages the phase execution, for both the downlink phase and the *Roundtrip*. Anyway, three different behaviors can be identified during this phase. The first

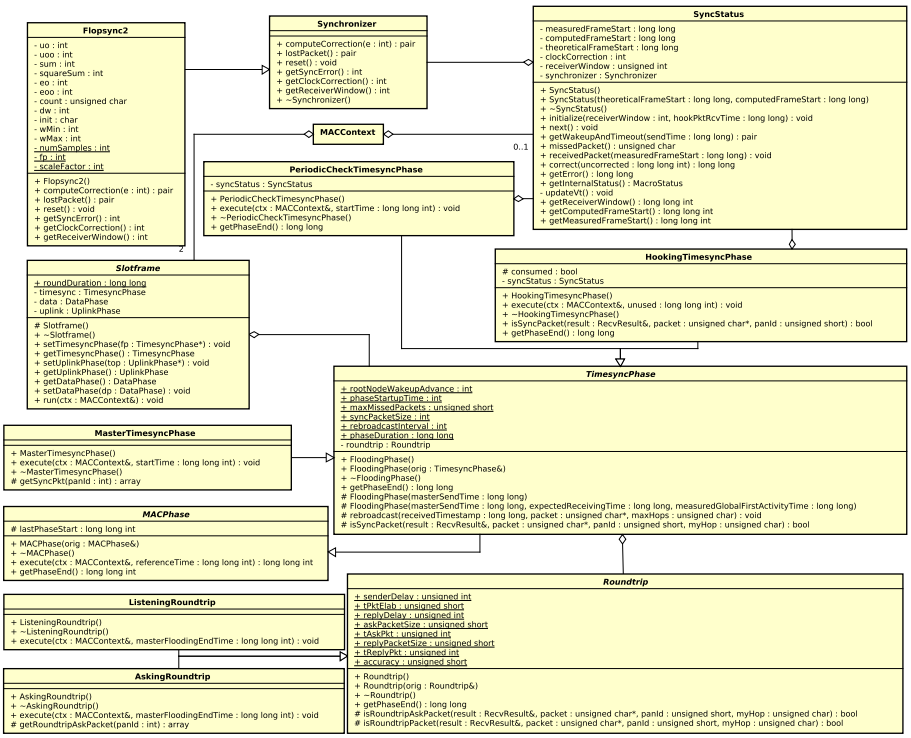


Figure 7.4: Class diagram for the Timesync part of the protocol

to be described is the *MasterTimesyncPhase*, whose job is to start the flooding to synchronize all the nodes. Then, the other two regarding the other nodes. These are the *HookingFloodingPhase* and the *PeriodicCheckFloodingPhase*. The *HookingFloodingPhase* is used when the synchronization to the master is lost, so it senses the channel looking for time synchronization flooding packets. The latter is instead intended to keep the synchronization by receiving the broadcasted packet and retransmitting it to the next hop.

These last two phases rely on the *SyncStatus* to store the synchronization state. It also provides an API to set the synchronization timestamps and obtain those of the next turn, provided by the *Flopsync2* controller. Another important method is getWakeupAndTimeout, which returns the maximum wake up time and the timeout, based on a given timestamp, using the information available after the synchronization has been performed.

After the flooding, the node could be involved in a *Roundtrip* calculation, performed in the last part of the execution of the *TimesyncPhase*. It can either

use the *ListeningRoundtrip* class or the *AskingRoundtrip* class, based on its role during the phase. They execute the roundtrip calculation as explained in the Reverse Flooding algorithm [35].

7.2.2 Uplink

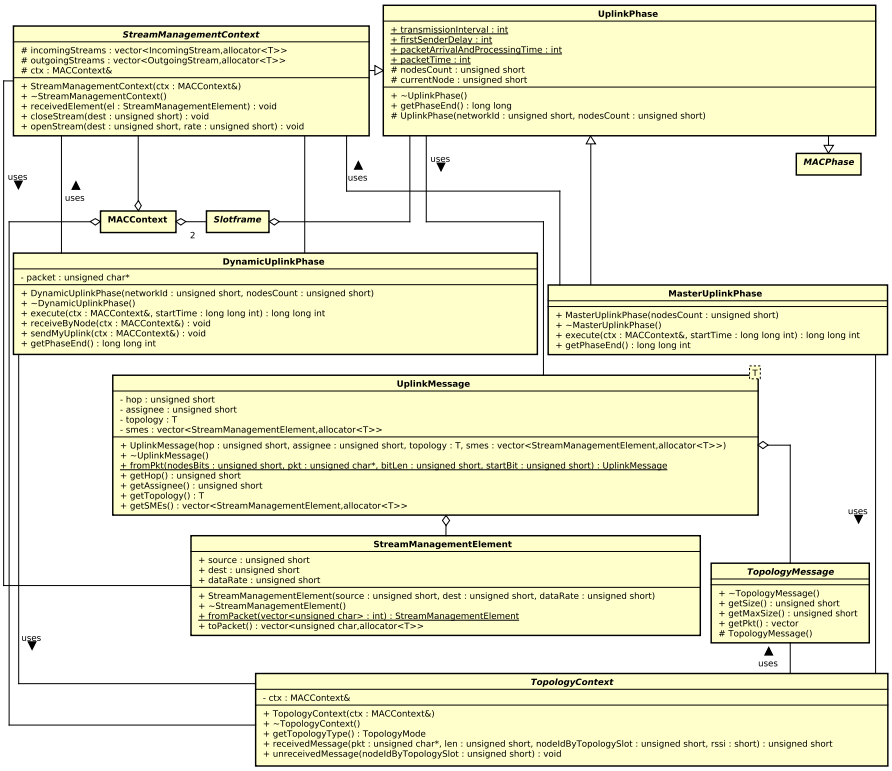


Figure 7.5: Principal components for the uplink phase.

Another phase of the protocol is the uplink phase, represented by the *UplinkPhase* class in the diagram. Every node during this phase needs to listen to every node's uplink phase, collecting their uplinks using the *UplinkMessage*. It then extracts the *TopologyMessage* to populate the *TopologyContext* as explained in the following section. Also, it extracts a list of *StreamManagementElement* objects that will be used to compute the next schedule, if it is the master node, otherwise it will be forwarded to the chosen predecessor. Then, when the node's turn comes, the *TopologyMessage* is formed and the SMEs to be sent are chosen. This is accomplished by the *DynamicUplinkPhase* and the

MasterUplinkPhase, interfacing with the *TopologyContext* and *StreamManagementContext* objects, whose function will be elucidated later. The components responsible for this mechanism are illustrated in figure 7.5.

Uplink - Topology

For what concerns the topology part, it is managed within the *TopologyContext*, which is an abstract class. It comes in two flavors: *MasterTopologyContext* and *DynamicTopologyContext*, based on the node's role. The former is used to collect the topologies within a *TopologyMap*, a class representing a generic undirected graph. This is also used by the *Scheduler*, whose function will be better explained later, to compute the schedule to be distributed. Also, if the master node notices the disconnection of a node, it closes its opened streams by acting on the *MasterStreamManagementContext*, that will be explainded in the dedicated section. Dynamic nodes instead employ the *DynamicTopologyContext* class whose specialization is keeping track of the known neighbors, together with the amount of times they didn't shown up during the uplink phase. This is accomplished in order to avoid having an unstable topology. It also keeps track of the messages to forward in a *UpdatablePriorityQueue*, which contains the messages in a FIFO queue, but enables to update its content if it changes (i.e. a node's neighbors addition/removal while it's still in queue). However, before storing the message, the context class needs to parse it, using the related class, subclass of *TopologyMessage*, depending on the topology in use. These messages are passed to the relative *TopologyContext* subclass, based on the topology type and the node role. They will either deduce neighbors, predecessors and gather topologies to forward if they are dynamic nodes or collect them into a *TopologyMap* if the receiving node is a master node. Then, in the former case, during the node's round, the information about the self topology and other nodes' topologies are assembled in an appropriate *TopologyMessage* and sent over the network in an *UplinkMessage*. The topology messages can be of two types, either a *MeshTopologyMessage* or a *TreeTopologyMessage*. The former contains the node address and the neighbors bitmask, as the protocol specifies, while the latter just contains the chosen parent of the previous hop address and the node's address. All these processes are described in figure 7.6.

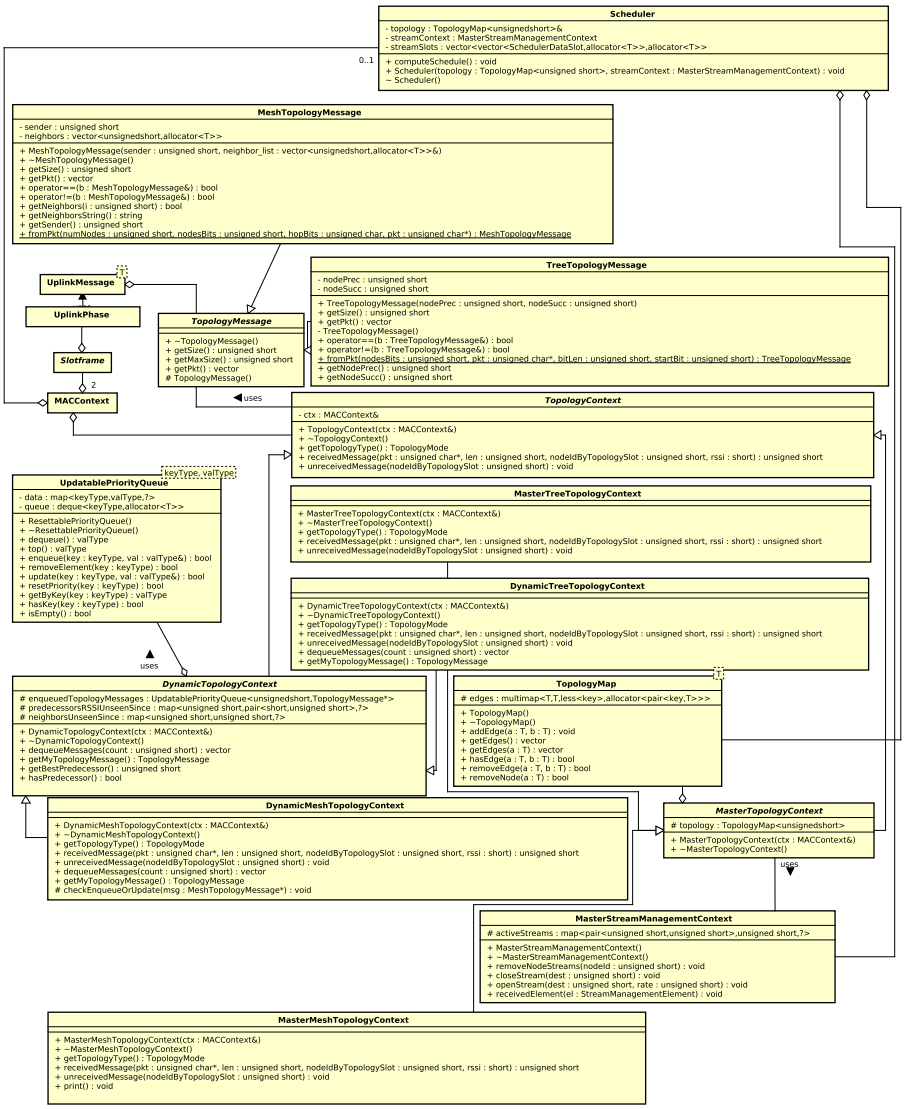


Figure 7.6: Classes for the topology collection part of the uplink phase.

Uplink - Stream management

The uplink message also contains information about the opening and closing streams. They are parsed from the *UplinkPhase* as *StreamManagementElement*, when they are received. If the receiving node is a master, those will be used to change the activeStreams property of the *MasterStreamManagementContext*. Those will be accessed also by the *Scheduler*, to compute a new schedule keeping track of them. In the case the receiving node is a dynamic node, they will be

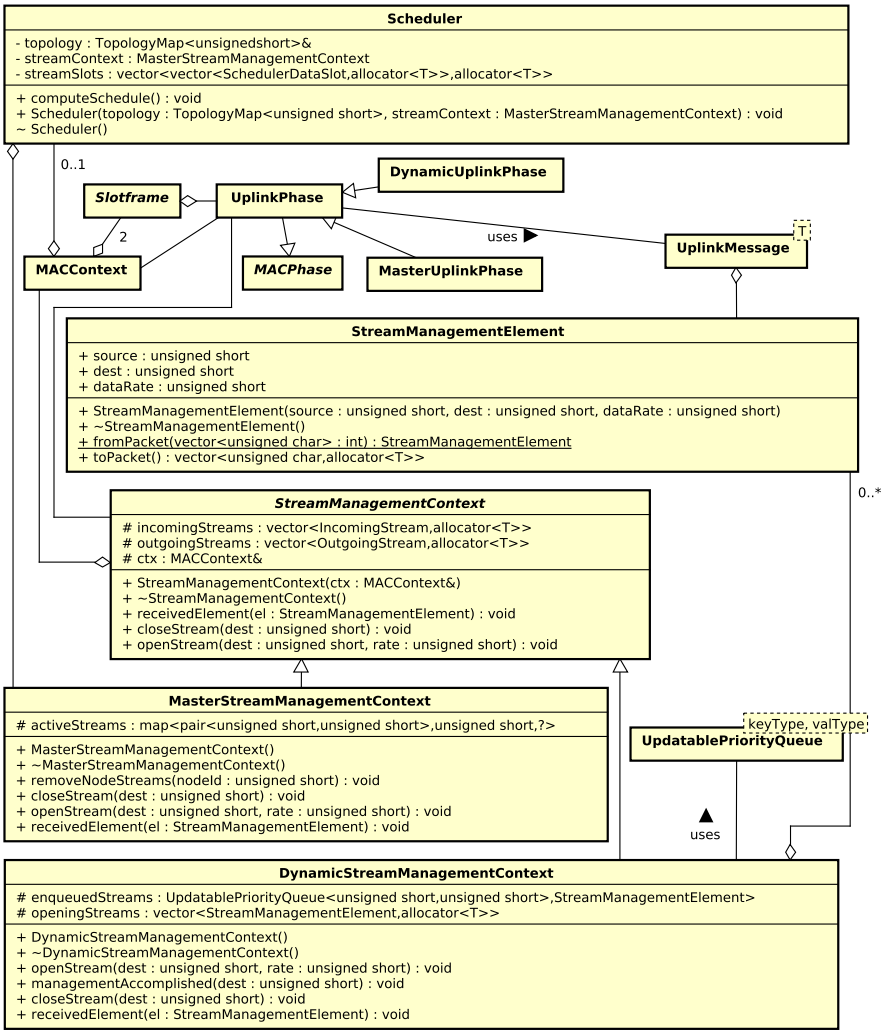


Figure 7.7: Classes for the stream management part of the uplink topology.

enqueued in the *DynamicStreamManagementContext*, to be sent to the master node repeatedly, until the schedule has changed to fulfill the required stream. When, instead, these requests are made directly from the node, i.e. it wants to open an outgoing stream, the outcome depends on the node's role. They all have a common API, openStream() and closeStream() in the *StreamManagementContext*. If it is a master, it directly allocates or deallocates it and will schedule it to the other nodes of the network during the next schedule downlink phase. Instead, in case of a dynamic node, this is managed by enqueuing it every time it sends out an uplink message and, when the related schedule is

received in the downlink, it stops sending the SME. This part is displayed in figure 7.7.

7.2.3 Schedule distribution

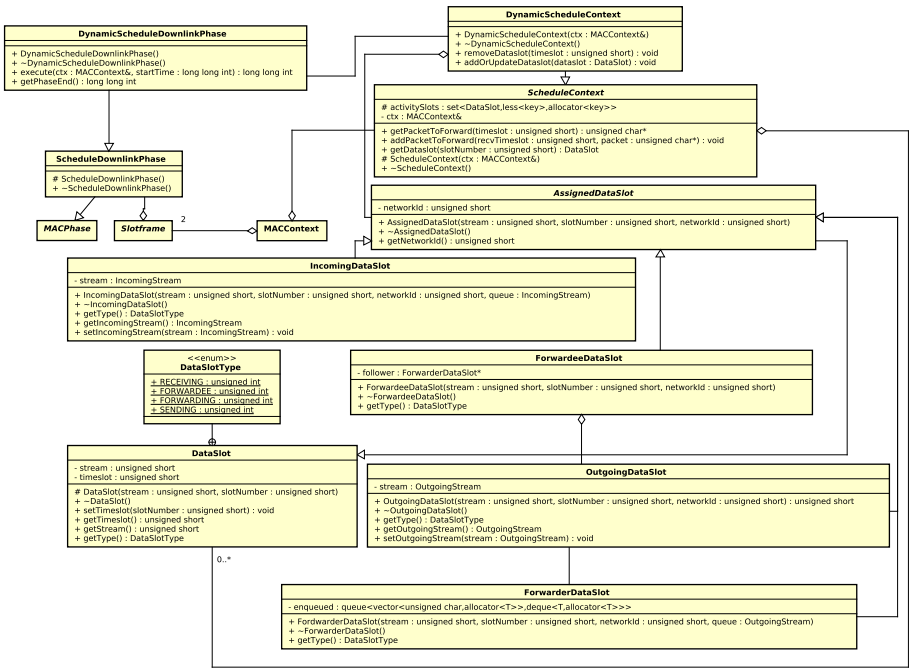


Figure 7.8: Classes for the schedule downlink phase for the dynamic nodes.

The schedule distribution phase starts from the master node. The *Scheduler* uses the collected topology and the opened streams to determine the schedule to be distributed and adopted. It is represented as an abstract class only, since it doesn't have an actual implementation. The computed schedule will be inserted into the *MasterScheduleContext*, which will also calculate the delta to distribute. Then, when the *ScheduleDownlinkPhase* starts, the *MasterScheduleDownlinkPhase* initiates the flooding by fetching the schedule delta from the context and flooding it. In the meanwhile, receiving nodes have started their *DynamicScheduleDownlinkPhase*, in which they will receive the schedule from the previous hop, parsing it into *DataSlot* of the appropriate type, if the node is contained in it. This is based on the presence and role of the node in the stream, inferred by the known information and the received schedule delta. Those will

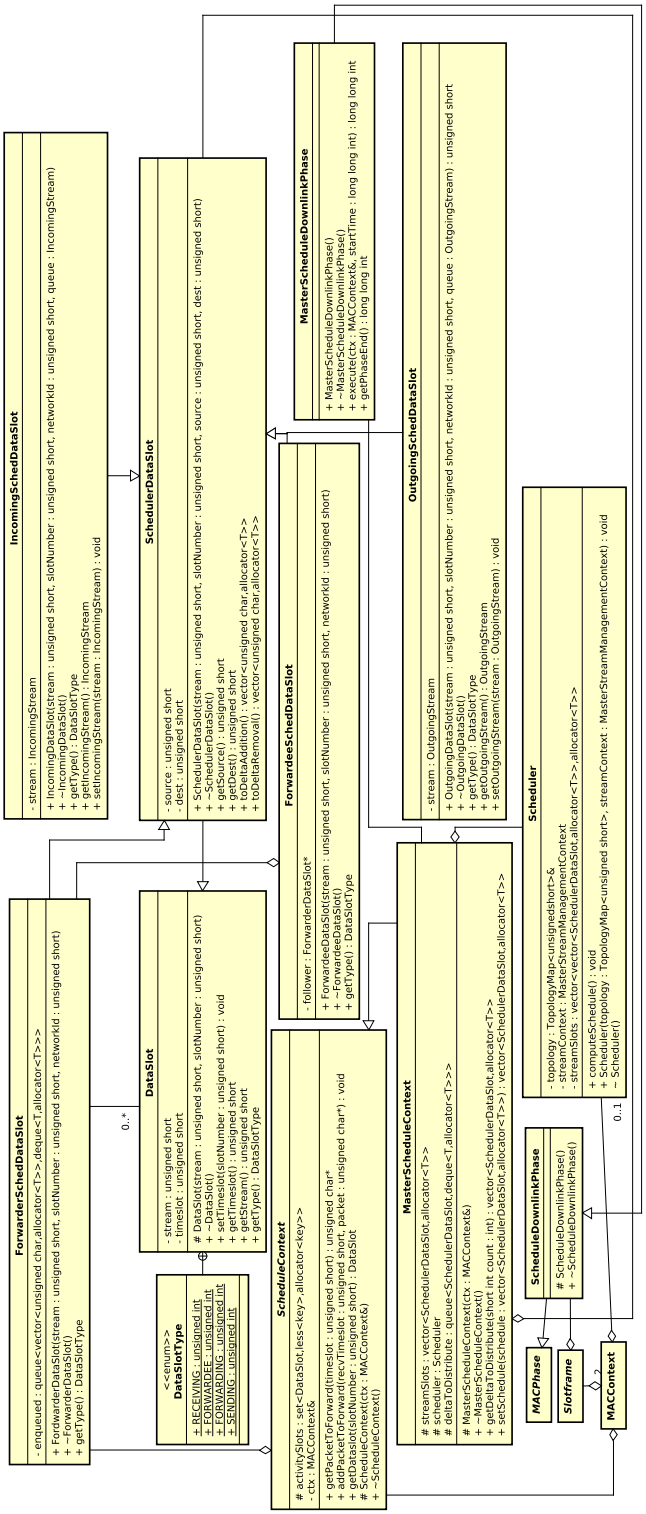


Figure 7.9: Classes for the schedule downlink phase for the master node.

be then parsed in *OutgoingDataSlot*, *IncomingDataSlot*, *ForwarderDataSlot* or *ForwardeeDataSlot*. Similar classes are created by the master for itself, based on the *DataSlotType*, in case it is a sender, a receiver or it needs to forward data from a slot to another. The related class diagram is shown in figure 7.9 and 7.8 for what concerns the master node in the former and for the dynamic node in the latter.

7.2.4 Data transmission

Data transmission phase is based on the received schedule and the role. For each data slot, either the *MasterDataPhase* or the *DynamicDataPhase*, based on the node's role is executed. To know what will be done in the data slot, *ScheduleContext* will be used to fetch the *DataSlot* for the current slot number. The actions depend on what will be fetched:

- **nullptr**, no action will be done, and the node will sleep saving energy;
- ***IncomingDataSlot***, the node will receive data from another node and will send it to the upper layers using the *IncomingStream*;
- ***OutgoingDataSlot***, if the *OutgoingStream* contained in the object is holding data, it will be sent on its route;
- ***ForwardeeDataSlot***, the node will listen for data and store it into the related *ForwarderDataSlot* queue, in order to forward it to the successive node
- ***ForwarderDataSlot***, the data enqueued, if present, will be forwarded, otherwise the node will sleep.

A variant of these *DataSlot* classes is present for the master node, which will use those generated by the scheduler directly, containing more elements in order to be flooded, as stated in the previous section. Figure 7.10 displays the class structure for the data phase for the master node. Figure 7.11 displays instead the structure for the dynamic nodes.
The proposed software structure represents the protocol in the most general cases, however it uses many classes and structures, which are thought for scalability and maintainability and not to achieve the best performance. Therefore,

Figure 7.10: Class diagram for the data phase for the master node.

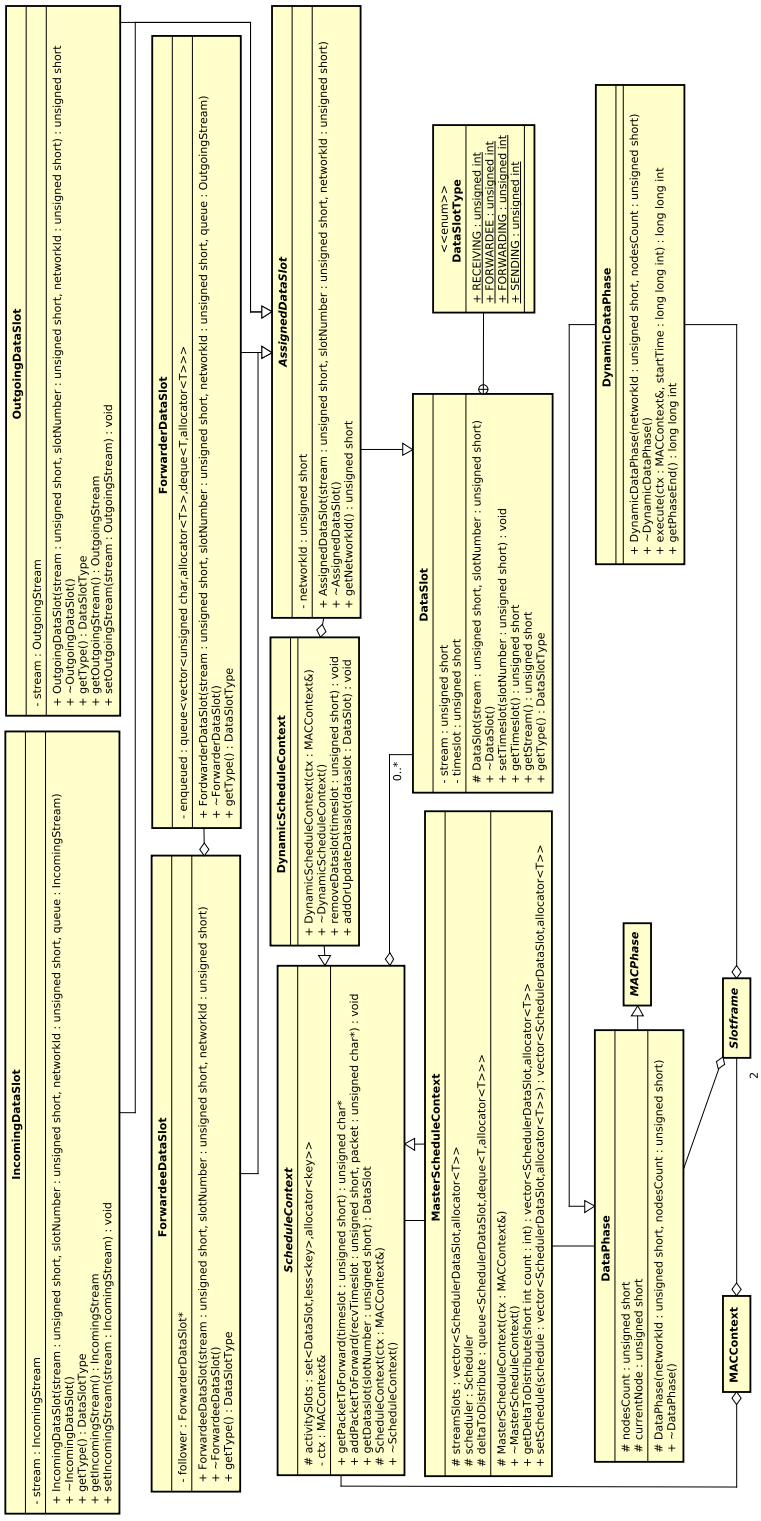


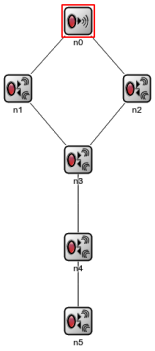
Figure 7.11: Class diagram for the data phase for the dynamic nodes.

when implementing the protocol, especially in low memory, slow devices, this aspect must be addressed and an analysis of structures to be used (i.e. linear search on lists versus logarithmic lookup on maps) should be accomplished. Moreover, many classes try to be as immutable as possible, to comply with OOP principles in the best possible way, causing however trashing and continuous memory reallocation. The use of a lower number of more mutable classes is suggested to achieve higher performances.

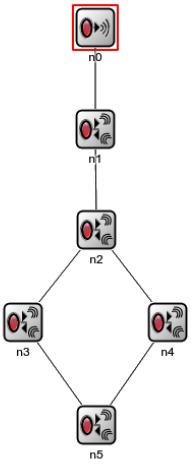
Chapter 8

Proof of concept

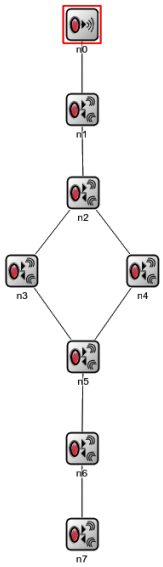
An implementation of the protocol has been realized in C++ within the OMNeT++ environment, as anticipated in 5.3. Its purpose is to prove the protocol to be working, therefore it is not a complete implementation. It models a network of 256 nodes maximum with a limit of 16 hops, in different configurations. The chosen topology model is the mesh topology, thus being able to use all the actual connections made in the simulator. Since no scheduler has been developed, the schedule used by the nodes was statically hardcoded to be sent out at a predefined time. Also, since no upper layer was present, the data to be sent out consist of *"Hello"* messages generated at runtime. Then, the interleaving function is a 1:1 map, not distributing the phases. The topologies implemented and for which the implementation has been tested are shown in figure 8.1. The three kites dispositions and the diamond topology are adopted because they help illustrate some particularities of the protocol, like the capture effects and how the uplink phase is run under certain conditions. The partial mesh is the most realistic topology. Star and full mesh topologies represent instead contexts in which all the nodes are respectively far from and close to the master. In the sections to follow the execution of the protocol is analyzed in particular cases and in some topologies, for which different characteristics can emerge. All the times displayed are expresses in nanoseconds.



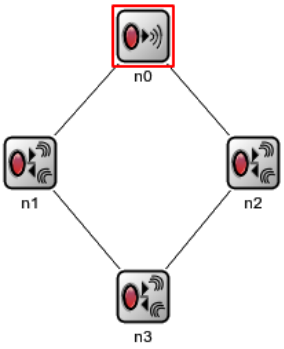
(a) Kite



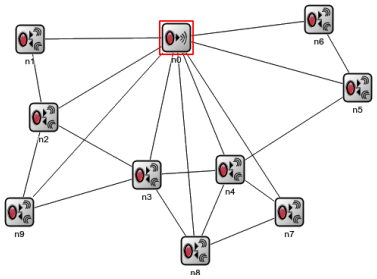
(b) Upside-down kite



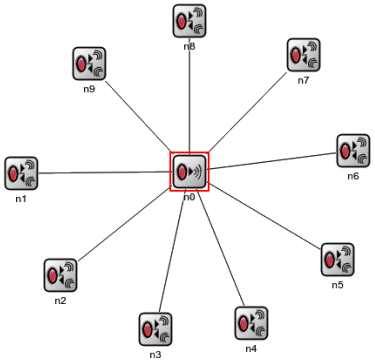
(c) Double tailed kite



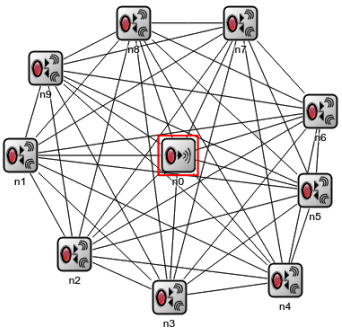
(d) Diamond



(e) Partial mesh



(f) Star



(g) Full mesh

Figure 8.1: Topologies simulated

8.1 Time synchronization

The time synchronization downlink phase is implemented in the protocol as explained in section 7.2.1. In the log, in listing 8.1 it is shown the time synchronization phase output for the diamond topology, already known to be representative for this topic. All the dynamic nodes start by resynchronizing their time to the master's, awaiting the Glossy flooding to happen. Then, when the master node sends the flooding, the nodes receive it, calculating the *Measured Gobal First Activity Time* (MGFAT) and rebroadcasting it to the next hop. Then also node 2 will receive such a packet. The *wake-up* (WU) and *timeout* (TO) times for the next synchronization phase are calculated. Notice, in row 10, that the node 2 knows it will receive the packet after a predefined forwarding interval from the previous hop, performing the whole calculation from the hop count included in the packet and its receiving time. At the next iteration the nodes will receive the synchronization packet, with error 0 since the simulation happens in an ideal discrete event simulator and no jitter nor any other particular physical linear or non-linear effect has been simulated. The FLOPSYNC-2 controller starts with a maximum window of $6ms$, however, after a reasonable number of iterations, the nodes achieve a better alignment with the master's clock thus being able to reduce it and activate their MAC protocol, respecting a maximum synchronization window constraint. No roundtrip time calculation is performed, since in simulation it would be 0.

Listing 8.1: Time synchronization phase in the diamond topology

```
1  n1: [F] Resync
2  n2: [F] Resync
3  n3: [F] Resync
4  n0: [F] ST=1000000
5  n1: [F] MGFAT 1000000
6  n1: [F] WU=9994550000 TO=10007160083
7  n2: [F] MGFAT 1000000
8  n2: [F] WU=9994550000 TO=10007160083
9  n3: [F] MGFAT 1000000
10 n3: [F] WU=9995566000 TO=10008176083
11 ------------------------------------------------
12 n0: [F] ST=10001000000
13 n1: [F] RT=10001000000
14 n1: e=0 u=0 w=6000000 rssi=5
15 n1: [F] MGFAT 10001000000
16 n1: [F] WU=19994550000 TO=20007160083
17 n2: [F] RT=10001000000
18 n2: e=0 u=0 w=6000000 rssi=5
19 n2: [F] MGFAT 10001000000
```

```
20 n2: [F] WU=19994550000 TO=20007160083
21 n3: [F] RT=10002016000
22 n3: e=0 u=0 w=6000000 rssi=5
23 n3: [F] MGFAT 10001000000
24 n3: [F] WU=19995566000 TO=20008176083
```

8.2 Uplink

The uplink phase, as described in 7.2.2, is a phase in which every node sends a packet (or more than one, if configured) containing its information to make the master gather knowledge about the network topology and elements to manage the streams related to a node, i.e. opening, closing or modifying a stream whose source is the sender node. First of all, a more verbose example of topology collection without stream management for a diamond topology is displayed in listing 8.2. At the first iteration, no node will know anything about its neighbors. Anyway, node 1 and 2 belong to the first hop, therefore they know about it, and can uniquely identify it as their predecessor. Hence they will send their uplink message during their turns (lines 5 and 9). The nodes within the radio range, after the appropriate time to receive the message, will receive their messages (lines 10-14, 17-19, 23-24). The time for expecting the packets is calculated on the basis of the node's address and all the nodes but the sender will listen to it using an appropriately calculated receiving window, based on their synchronization status. The master node will build the topology map in the meanwhile, collecting information about the network (lines 20-22). At the next iteration, the process will repeat. However, since the notion of neighbors changed, the uplink message content reflects it (lines 26, 30, 34, 40, 42, 49, 56). Therefore node 3 will be able to choose a predecessor, in this case node 1 (lines 41-42), which will enqueue its information and forward them as soon as possible respecting a FIFO queue. In this case, the message reaches the master node during the same round, since no other nodes belong to hop 2. Now, the master has a complete knowledge about the topology (lines 51-55).

Listing 8.2: Diamond topology collection

```
n3: neighbors:
n1: [T] expect 3 from 91530966068 to 91536966068
n2: [T] expect 3 from 91530966068 to 91536966068
n2: neighbors:
n2: [T] N=2 -> @91542706000
n3: [T] expect 2 from 91530966068 to 91542966068
n1: [T] expect 2 from 91536966068 to 91542966068
n1: neighbors:
n1: [T] N=1 -> @91548706000
n0: Received topology :[2/1->0]:0
n0: Topology added 2 -> 0
n0: [T] <- N=2 @91542706000
n3: received: [2/1] : 0
n3: [T] <- N=2 @91542706000
n2: [T] expect 1 from 91544114000 to 91548966068
n3: [T] expect 1 from 91544178000 to 91548966068
n0: Received topology :[1/1->0]:0
n0: Topology added 1 -> 0
n0: [T] <- N=1 @91548706000
n0: [T] Current topology:
n0: [0 - 2]
n0: [0 - 1]
n3: Received topology :[1/1->0]:0
n3: [T] <- N=1 @91548706000
-----------------------------------------------
n3: neighbors: 1, 2,
n3: [T] N=3 -> @101536706000
n1: [T] expect 3 from 101530966068 to 101536966068
n2: [T] expect 3 from 101530966068 to 101536966068
n2: received: [3/2] : 1, 2
n2: [T] <- N=3 @101536706000
n2: neighbors: 3,
n2: [T] N=2 -> @101542706000
n1: received: [3/2] : 1, 2
n1: [T] <- N=3 @101536706000
n3: [T] expect 2 from 101538114000 to 101542966068
n1: [T] expect 2 from 101538178000 to 101542966068
n1: neighbors: 3,
n1: [T] N=1 -> @101548706000
n0: Received topology :[2/1->0]:0, 3
n0: Topology added 2 -> 3
n0: Received topology :[3/2->2]:1, 2
n0: Topology added 3 -> 1
n0: [T] <- N=2 @101542706000
n3: received: [2/1] : 0, 3
n3: [T] <- N=2 @101542706000
n2: [T] expect 1 from 101545234000 to 101548966068
n3: [T] expect 1 from 101545298000 to 101548966068
n0: Received topology :[1/1->0]:0, 3
n0: [T] <- N=1 @101548706000
n0: [T] Current topology:
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 3]
n0: [2 - 3]
n3: Received topology :[1/1->0]:0, 3
n3: [T] <- N=1 @101548706000
```

In certain topologies, this process is accomplished faster, even if the connection number is higher. For instance in the complete mesh topology all the nodes can communicate to the master their topology directly, since they belong to hop 1. Moreover, they also listen to the uplink messages sent before theirs. Therefore they are able to build their knowledge about the neighbors iteratively, as the uplink phase goes on, and the same holds for the master node. Hence, as displayed in 8.3, after just one iteration, all the node's topology information will be collected into the master node's topology map.

Listing 8.3: Complete mesh topology collection

```
n9: neighbors:
n0: Received topology :[9/1->0]:0
n0: Topology added 9 -> 0
n2: received: [9/1] : 0
n3: received: [9/1] : 0
n1: received: [9/1] : 0
n8: received: [9/1] : 0
n8: neighbors: 9,
n7: received: [9/1] : 0
n4: received: [9/1] : 0
n6: received: [9/1] : 0
n5: received: [9/1] : 0
n0: Received topology :[8/1->0]:0, 9
n0: Topology added 8 -> 0
n0: Topology added 8 -> 9
n3: received: [8/1] : 0, 9
n4: received: [8/1] : 0, 9
n7: received: [8/1] : 0, 9
n7: neighbors: 8, 9,
n9: received: [8/1] : 0, 9
n1: received: [8/1] : 0, 9
n6: received: [8/1] : 0, 9
n5: received: [8/1] : 0, 9
n2: received: [8/1] : 0, 9
n0: Received topology :[7/1->0]:0, 8, 9
n0: Topology added 7 -> 0
n0: Topology added 7 -> 8
n0: Topology added 7 -> 9
n4: received: [7/1] : 0, 8, 9
n8: received: [7/1] : 0, 8, 9
n6: received: [7/1] : 0, 8, 9
n6: neighbors: 7, 8, 9,
n9: received: [7/1] : 0, 8, 9
n5: received: [7/1] : 0, 8, 9
n3: received: [7/1] : 0, 8, 9
n2: received: [7/1] : 0, 8, 9
n1: received: [7/1] : 0, 8, 9
n0: Received topology :[6/1->0]:0, 7, 8, 9
n0: Topology added 6 -> 0
n0: Topology added 6 -> 7
n0: Topology added 6 -> 8
n0: Topology added 6 -> 9
n5: received: [6/1] : 0, 7, 8, 9
```

```
n5: neighbors: 6, 7, 8, 9,
n7: received: [6/1] : 0, 7, 8, 9
n8: received: [6/1] : 0, 7, 8, 9
n4: received: [6/1] : 0, 7, 8, 9
n3: received: [6/1] : 0, 7, 8, 9
n2: received: [6/1] : 0, 7, 8, 9
n1: received: [6/1] : 0, 7, 8, 9
n9: received: [6/1] : 0, 7, 8, 9
n0: Received topology :[5/1->0]:0, 6, 7, 8, 9
n0: Topology added 5 -> 0
n0: Topology added 5 -> 6
n0: Topology added 5 -> 7
n0: Topology added 5 -> 8
n0: Topology added 5 -> 9
n4: received: [5/1] : 0, 6, 7, 8, 9
n4: neighbors: 5, 6, 7, 8, 9,
n6: received: [5/1] : 0, 6, 7, 8, 9
n7: received: [5/1] : 0, 6, 7, 8, 9
n3: received: [5/1] : 0, 6, 7, 8, 9
n2: received: [5/1] : 0, 6, 7, 8, 9
n1: received: [5/1] : 0, 6, 7, 8, 9
n8: received: [5/1] : 0, 6, 7, 8, 9
n9: received: [5/1] : 0, 6, 7, 8, 9
n0: Received topology :[4/1->0]:0, 5, 6, 7, 8, 9
n0: Topology added 4 -> 0
n0: Topology added 4 -> 5
n0: Topology added 4 -> 6
n0: Topology added 4 -> 7
n0: Topology added 4 -> 8
n0: Topology added 4 -> 9
n8: received: [4/1] : 0, 5, 6, 7, 8, 9
n3: received: [4/1] : 0, 5, 6, 7, 8, 9
n3: neighbors: 4, 5, 6, 7, 8, 9,
n7: received: [4/1] : 0, 5, 6, 7, 8, 9
n5: received: [4/1] : 0, 5, 6, 7, 8, 9
n6: received: [4/1] : 0, 5, 6, 7, 8, 9
n9: received: [4/1] : 0, 5, 6, 7, 8, 9
n1: received: [4/1] : 0, 5, 6, 7, 8, 9
n2: received: [4/1] : 0, 5, 6, 7, 8, 9
n0: Received topology :[3/1->0]:0, 4, 5, 6, 7, 8, 9
n0: Topology added 3 -> 0
n0: Topology added 3 -> 4
n0: Topology added 3 -> 5
n0: Topology added 3 -> 6
n0: Topology added 3 -> 7
n0: Topology added 3 -> 8
n0: Topology added 3 -> 9
n2: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n2: neighbors: 3, 4, 5, 6, 7, 8, 9,
n9: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n8: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n4: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n5: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n1: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n7: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n6: received: [3/1] : 0, 4, 5, 6, 7, 8, 9
n0: Received topology :[2/1->0]:0, 3, 4, 5, 6, 7, 8, 9
n0: Topology added 2 -> 0
n0: Topology added 2 -> 3
n0: Topology added 2 -> 4
n0: Topology added 2 -> 5
n0: Topology added 2 -> 6
```

```
n0: Topology added 2 -> 7
n0: Topology added 2 -> 8
n0: Topology added 2 -> 9
n3: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n9: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n1: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n1: neighbors: 2, 3, 4, 5, 6, 7, 8, 9,
n4: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n5: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n6: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n7: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n8: received: [2/1] : 0, 3, 4, 5, 6, 7, 8, 9
n0: Received topology :[1/1->0]:0, 2, 3, 4, 5, 6, 7, 8, 9
n0: Topology added 1 -> 0
n0: Topology added 1 -> 2
n0: Topology added 1 -> 3
n0: Topology added 1 -> 4
n0: Topology added 1 -> 5
n0: Topology added 1 -> 6
n0: Topology added 1 -> 7
n0: Topology added 1 -> 8
n0: Topology added 1 -> 9
n0: [T] Current topology:
n0: [0 - 9]
n0: [0 - 8]
n0: [0 - 7]
n0: [0 - 6]
n0: [0 - 5]
n0: [0 - 4]
n0: [0 - 3]
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 2]
n0: [1 - 3]
n0: [1 - 4]
n0: [1 - 5]
n0: [1 - 6]
n0: [1 - 7]
n0: [1 - 8]
n0: [1 - 9]
n0: [2 - 3]
n0: [2 - 4]
n0: [2 - 5]
n0: [2 - 6]
n0: [2 - 7]
n0: [2 - 8]
n0: [2 - 9]
n0: [3 - 4]
n0: [3 - 5]
n0: [3 - 6]
n0: [3 - 7]
n0: [3 - 8]
n0: [3 - 9]
n0: [4 - 5]
n0: [4 - 6]
n0: [4 - 7]
n0: [4 - 8]
n0: [4 - 9]
n0: [5 - 6]
n0: [5 - 7]
n0: [5 - 8]
n0: [5 - 9]
```

```
n0: [6 - 7]
n0: [6 - 8]
n0: [6 - 9]
n0: [7 - 8]
n0: [7 - 9]
n0: [8 - 9]
```

A simpler example is with the partial mesh, in which not all the connections are made, although all the nodes belong to the first hop, so the topology map will be built at the first iteration, as well as the nodes' knowledge of their neighbors. This is illustrated in listing 8.4.

Listing 8.4: Partial mesh topology collection

```
n9: neighbors:
n8: neighbors:
n0: Received topology :[9/1->0]:0
n0: Topology added 9 -> 0
n2: received: [9/1] : 0
n3: received: [9/1] : 0
n0: Received topology :[8/1->0]:0
n0: Topology added 8 -> 0
n3: received: [8/1] : 0
n4: received: [8/1] : 0
n7: received: [8/1] : 0
n7: neighbors: 8,
n6: neighbors:
n0: Received topology :[7/1->0]:0, 8
n0: Topology added 7 -> 0
n0: Topology added 7 -> 8
n4: received: [7/1] : 0, 8
n8: received: [7/1] : 0, 8
n0: Received topology :[6/1->0]:0
n0: Topology added 6 -> 0
n5: received: [6/1] : 0
n5: neighbors: 6,
n0: Received topology :[5/1->0]:0, 6
n0: Topology added 5 -> 0
n0: Topology added 5 -> 6
n4: received: [5/1] : 0, 6
n4: neighbors: 5, 7, 8,
n6: received: [5/1] : 0, 6
n0: Received topology :[4/1->0]:0, 5, 7, 8
n0: Topology added 4 -> 0
n0: Topology added 4 -> 5
n0: Topology added 4 -> 7
n0: Topology added 4 -> 8
n8: received: [4/1] : 0, 5, 7, 8
n3: received: [4/1] : 0, 5, 7, 8
n3: neighbors: 4, 8, 9,
n7: received: [4/1] : 0, 5, 7, 8
n5: received: [4/1] : 0, 5, 7, 8
n0: Received topology :[3/1->0]:0, 4, 8, 9
```

```
n0: Topology added 3 -> 0
n0: Topology added 3 -> 4
n0: Topology added 3 -> 8
n0: Topology added 3 -> 9
n2: received: [3/1] : 0, 4, 8, 9
n2: neighbors: 3, 9,
n9: received: [3/1] : 0, 4, 8, 9
n8: received: [3/1] : 0, 4, 8, 9
n4: received: [3/1] : 0, 4, 8, 9
n0: Received topology :[2/1->0]:0, 3, 9
n0: Topology added 2 -> 0
n0: Topology added 2 -> 3
n0: Topology added 2 -> 9
n3: received: [2/1] : 0, 3, 9
n9: received: [2/1] : 0, 3, 9
n1: received: [2/1] : 0, 3, 9
n1: neighbors: 2,
n0: Received topology :[1/1->0]:0, 2
n0: Topology added 1 -> 0
n0: Topology added 1 -> 2
n0: [T] Current topology:
n0: [0 - 9]
n0: [0 - 8]
n0: [0 - 7]
n0: [0 - 6]
n0: [0 - 5]
n0: [0 - 4]
n0: [0 - 3]
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 9]
n0: [3 - 4]
n0: [3 - 8]
n0: [3 - 9]
n0: [4 - 5]
n0: [4 - 7]
n0: [4 - 8]
n0: [5 - 6]
n0: [7 - 8]
```

In both the kite and the upside-down kite topologies, the topology map needs instead three rounds to be built, since there are four hops and the information about the fourth hop is provided by the topology data of previous hop's nodes. Such information arrives to the master after a number of iterations equal to those node's hop count ($\#_{hop} - 1$), in case that no uplink message is oversaturated. The logs of such collections is displayed in listings listings 8.5 to 8.6.

Listing 8.5: Topology collection of a kite configured network

```
n5: neighbors:
n4: neighbors:
n3: neighbors: 1, 2,
n2: received: [3/2] : 1, 2
n2: neighbors: 3,
n1: received: [3/2] : 1, 2
n4: received: [3/2] : 1, 2
n1: neighbors: 3,
n0: Received topology :[2/1->0]:0, 3
n0: Topology added 2 -> 3
n0: Received topology :[3/2->2]:1, 2
n0: Topology added 3 -> 1
n3: received: [2/1] : 0, 3
n0: Received topology :[1/1->0]:0, 3
n0: [T] Current topology:
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 3]
n0: [2 - 3]
-----------------------------------------------
n5: neighbors:
n4: neighbors: 3,
n3: received: [4/3] : 3
n3: neighbors: 1, 2, 4,
n5: received: [4/3] : 3
n2: received: [3/2] : 1, 2, 4
n2: received: [4/3] : 3
n2: neighbors: 3,
n1: received: [3/2] : 1, 2, 4
n4: received: [3/2] : 1, 2, 4
n1: neighbors: 3,
n0: Received topology :[2/1->0]:0, 3
n0: Received topology :[3/2->2]:1, 2, 4
n0: Topology added 3 -> 4
n0: Received topology :[4/3->3]:3
n3: received: [2/1] : 0, 3
n0: Received topology :[1/1->0]:0, 3
n0: [T] Current topology:
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 3]
n0: [2 - 3]
n0: [3 - 4]
-----------------------------------------------
n5: neighbors: 4,
n4: received: [5/4] : 4
n4: neighbors: 3, 5,
n3: received: [4/3] : 3, 5
n3: received: [5/4] : 4
n3: neighbors: 1, 2, 4,
n5: received: [4/3] : 3, 5
n2: received: [3/2] : 1, 2, 4
n2: received: [4/3] : 3, 5
n2: received: [5/4] : 4
n2: neighbors: 3,
n1: received: [3/2] : 1, 2, 4
n4: received: [3/2] : 1, 2, 4
n1: neighbors: 3,
n0: Received topology :[2/1->0]:0, 3
n0: Received topology :[3/2->2]:1, 2, 4
n0: Received topology :[4/3->3]:3, 5
```

```
n0: Topology added 4 -> 5
n3: received: [2/1] : 0, 3
n0: Received topology :[1/1->0]:0, 3
n0: [T] Current topology:
n0: [0 - 2]
n0: [0 - 1]
n0: [1 - 3]
n0: [2 - 3]
n0: [3 - 4]
n0: [4 - 5]
```

Listing 8.6: Topology collection in an upside-down kite topology

```
n5: neighbors:
n4: neighbors:
n3: neighbors:
n2: neighbors:
n1: neighbors:
n0: Received topology :[1/1->0]:0
n0: Topology added 1 -> 0
n0: [T] Current topology:
n0: [0 - 1]
n0: 1 -[1]-> 0
-----------------------------------
n5: neighbors:
n4: neighbors: 2,
n3: neighbors: 2,
n2: received: [4/3] : 2
n5: received: [4/3] : 2
n2: received: [3/3] : 2
n2: neighbors: 1, 3, 4,
n5: received: [3/3] : 2
n3: received: [2/2] : 1, 3, 4
n4: received: [2/2] : 1, 3, 4
n1: received: [2/2] : 1, 3, 4
n1: received: [4/3] : 2
n1: received: [3/3] : 2
n1: neighbors: 2,
n0: Received topology :[1/1->0]:0, 2
n0: Received topology :[2/2->1]:1, 3, 4
n0: Topology added 2 -> 3
n0: Topology added 2 -> 4
n0: Received topology :[4/3->2]:2
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 4]
-----------------------------------
n5: neighbors: 3, 4,
n4: received: [5/4] : 3, 4
n4: neighbors: 2, 5,
n3: received: [5/4] : 3, 4
n3: neighbors: 2, 5,
n2: received: [4/3] : 2, 5
n2: received: [5/4] : 3, 4
n5: received: [4/3] : 2, 5
n2: received: [3/3] : 2, 5
```

```
46 n2: neighbors: 1, 3, 4,
47 n5: received: [3/3] : 2, 5
48 n3: received: [2/2] : 1, 3, 4
49 n4: received: [2/2] : 1, 3, 4
50 n1: received: [2/2] : 1, 3, 4
51 n1: received: [4/3] : 2, 5
52 n1: received: [5/4] : 3, 4
53 n1: neighbors: 2,
54 n0: Received topology :[1/1->0]:0, 2
55 n0: Received topology :[3/3->2]:2
56 n0: Received topology :[2/2->1]:1, 3, 4
57 n0: [T] Current topology:
58 n0: [0 - 1]
59 n0: [1 - 2]
60 n0: [2 - 3]
61 n0: [2 - 4]
62 -----------------------------------
63 n5: neighbors: 3, 4,
64 n4: received: [5/4] : 3, 4
65 n4: neighbors: 2, 5,
66 n3: received: [5/4] : 3, 4
67 n3: neighbors: 2, 5,
68 n2: received: [4/3] : 2, 5
69 n2: received: [5/4] : 3, 4
70 n5: received: [4/3] : 2, 5
71 n2: received: [3/3] : 2, 5
72 n2: neighbors: 1, 3, 4,
73 n5: received: [3/3] : 2, 5
74 n3: received: [2/2] : 1, 3, 4
75 n4: received: [2/2] : 1, 3, 4
76 n1: received: [2/2] : 1, 3, 4
77 n1: received: [3/3] : 2, 5
78 n1: received: [4/3] : 2, 5
79 n1: neighbors: 2,
80 n0: Received topology :[1/1->0]:0, 2
81 n0: Received topology :[4/3->2]:2, 5
82 n0: Topology added 4 -> 5
83 n0: Received topology :[5/4->4]:3, 4
84 n0: Topology added 5 -> 3
85 n0: [T] Current topology:
86 n0: [0 - 1]
87 n0: [1 - 2]
88 n0: [2 - 3]
89 n0: [2 - 4]
90 n0: [3 - 5]
91 n0: [4 - 5]
```

Another example of topology collection, is realized in a the simple star topology, in which also streams are opened during the uplink phase. The example is about a real application, in which the network is used to perform data collection with a low frequency sampling. The master node achieves knowledge about other node's information in one slotframe only, as expected. Stream requests are displayed in lines 4, 9, 14, 19, 24, 29, 34, 39 of listing 8.7. The output of

node 0 shows that the streams were opened in lines 7, 12, 17, 22, 27, 32, 37, 42, 45, and then the whole information collected by node 0 about the open streams from line 56 to line 65.

Listing 8.7: Topology collection and stream management of a star topology

```
1  n9: neighbors:
2  n9: [S] Opening stream to 0 with rate 1
3  n8: neighbors:
4  n8: [S] Opening stream to 0 with rate 1
5  n0: Received topology :[9/1->0]:0
6  n0: Topology added 9 -> 0
7  n0: [S] Opened stream 9 -[1]-> 0
8  n7: neighbors:
9  n7: [S] Opening stream to 0 with rate 1
10 n0: Received topology :[8/1->0]:0
11 n0: Topology added 8 -> 0
12 n0: [S] Opened stream 8 -[1]-> 0
13 n6: neighbors:
14 n6: [S] Opening stream to 0 with rate 1
15 n0: Received topology :[7/1->0]:0
16 n0: Topology added 7 -> 0
17 n0: [S] Opened stream 7 -[1]-> 0
18 n5: neighbors:
19 n5: [S] Opening stream to 0 with rate 1
20 n0: Received topology :[6/1->0]:0
21 n0: Topology added 6 -> 0
22 n0: [S] Opened stream 6 -[1]-> 0
23 n4: neighbors:
24 n4: [S] Opening stream to 0 with rate 1
25 n0: Received topology :[5/1->0]:0
26 n0: Topology added 5 -> 0
27 n0: [S] Opened stream 5 -[1]-> 0
28 n3: neighbors:
29 n3: [S] Opening stream to 0 with rate 1
30 n0: Received topology :[4/1->0]:0
31 n0: Topology added 4 -> 0
32 n0: [S] Opened stream 4 -[1]-> 0
33 n2: neighbors:
34 n2: [S] Opening stream to 0 with rate 1
35 n0: Received topology :[3/1->0]:0
36 n0: Topology added 3 -> 0
37 n0: [S] Opened stream 3 -[1]-> 0
38 n1: neighbors:
39 n1: [S] Opening stream to 0 with rate 1
40 n0: Received topology :[2/1->0]:0
41 n0: Topology added 2 -> 0
42 n0: [S] Opened stream 2 -[1]-> 0
43 n0: Received topology :[1/1->0]:0
44 n0: Topology added 1 -> 0
45 n0: [S] Opened stream 1 -[1]-> 0
46 n0: [T] Current topology:
47 n0: [0 - 9]
48 n0: [0 - 8]
49 n0: [0 - 7]
50 n0: [0 - 6]
51 n0: [0 - 5]
52 n0: [0 - 4]
53 n0: [0 - 3]
```

```
n0: [0 - 2]
n0: [0 - 1]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 2 -[1]-> 0
n0: 3 -[1]-> 0
n0: 4 -[1]-> 0
n0: 5 -[1]-> 0
n0: 6 -[1]-> 0
n0: 7 -[1]-> 0
n0: 8 -[1]-> 0
n0: 9 -[1]-> 0
```

In the following case, instead, a double tailed configuration was used. It it the nodes open streams to the master with data rate 1 and to another node, with data rate 255. In this case, however, all the SMEs can fit in the uplink message and are therefore forwarded as well as the topology information. This is illustrated in listing 8.8.

Listing 8.8: Double tail kite topology collectionn and stream management

```
n7: neighbors:
n6: neighbors:
n5: neighbors:
n4: neighbors:
n3: neighbors:
n2: neighbors:
n1: neighbors:
n1: [S] Opening stream to 0 with rate 1
n1: [S] Opening stream to 6 with rate 255
n1: [S] Sending SME: SND 1 -[1]-> 0
n1: [S] Sending SME: SND 1 -[255]-> 6
n0: Received topology :[1/1->0]:0
n0: Topology added 1 -> 0
n0: [S] Opened stream 1 -[1]-> 0
n0: [S] Opened stream 1 -[255]-> 6
n0: [T] Current topology:
n0: [0 - 1]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n2: received: [1/1] : 0
-----------------------------------------------
n7: neighbors:
n6: neighbors:
n5: neighbors:
n4: neighbors:
n3: neighbors:
n2: neighbors: 1,
n2: [S] Opening stream to 0 with rate 1
```

```
n2: [S] Opening stream to 7 with rate 255
n2: [S] Sending SME: SND 2 -[1]-> 0
n2: [S] Sending SME: SND 2 -[255]-> 7
n3: received: [2/2] : 1
n4: received: [2/2] : 1
n1: received: [2/2] : 1
n1: [S] enqueued SME: SND 2 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 2 -[255]-> 7 RCV
n1: neighbors: 2,
n1: [S] Forwarding SME: SND 2 -[1]-> 0
n1: [S] Forwarding SME: SND 2 -[255]-> 7
n0: Received topology :[1/1->0]:0, 2
n0: Topology added 1 -> 2
n0: Received topology :[2/2->1]:1
n0: [S] Opened stream 2 -[1]-> 0
n0: [S] Opened stream 2 -[255]-> 7
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n0: 2 -[1]-> 0
n0: 2 -[255]-> 7
n2: received: [1/1] : 0, 2
-----------------------------------------------
n7: neighbors:
n6: neighbors:
n5: neighbors:
n4: neighbors: 2,
n4: [S] Opening stream to 0 with rate 1
n4: [S] Opening stream to 1 with rate 255
n4: [S] Sending SME: SND 4 -[1]-> 0
n4: [S] Sending SME: SND 4 -[255]-> 1
n3: neighbors: 2,
n3: [S] Opening stream to 0 with rate 1
n3: [S] Opening stream to 8 with rate 255
n3: [S] Sending SME: SND 3 -[1]-> 0
n3: [S] Sending SME: SND 3 -[255]-> 8
n2: received: [4/3] : 2
n2: [S] enqueued SME: SND 4 -[1]-> 0 RCV
n2: [S] enqueued SME: SND 4 -[255]-> 1 RCV
n5: received: [4/3] : 2
n2: received: [3/3] : 2
n2: [S] enqueued SME: SND 3 -[1]-> 0 RCV
n2: [S] enqueued SME: SND 3 -[255]-> 8 RCV
n2: neighbors: 1, 3, 4,
n2: [S] Forwarding SME: SND 4 -[1]-> 0
n2: [S] Forwarding SME: SND 4 -[255]-> 1
n2: [S] Forwarding SME: SND 3 -[1]-> 0
n2: [S] Forwarding SME: SND 3 -[255]-> 8
n5: received: [3/3] : 2
n3: received: [2/2] : 1, 3, 4
n4: received: [2/2] : 1, 3, 4
n1: received: [2/2] : 1, 3, 4
n1: received: [4/3] : 2
n1: received: [3/3] : 2
n1: [S] enqueued SME: SND 4 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 4 -[255]-> 1 RCV
n1: [S] enqueued SME: SND 3 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 3 -[255]-> 8 RCV
n1: neighbors: 2,
```

```
n1: [S] Forwarding SME: SND 4 -[1]-> 0
n1: [S] Forwarding SME: SND 4 -[255]-> 1
n1: [S] Forwarding SME: SND 3 -[1]-> 0
n1: [S] Forwarding SME: SND 3 -[255]-> 8
n0: Received topology :[1/1->0]:0, 2
n0: Received topology :[2/2->1]:1, 3, 4
n0: Topology added 2 -> 3
n0: Topology added 2 -> 4
n0: Received topology :[4/3->2]:2
n0: [S] Opened stream 4 -[1]-> 0
n0: [S] Opened stream 4 -[255]-> 1
n0: [S] Opened stream 3 -[1]-> 0
n0: [S] Opened stream 3 -[255]-> 8
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 4]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n0: 2 -[1]-> 0
n0: 2 -[255]-> 7
n0: 3 -[1]-> 0
n0: 3 -[255]-> 8
n0: 4 -[1]-> 0
n0: 4 -[255]-> 1
n2: received: [1/1] : 0, 2
------------------------------------------------
n7: neighbors:
n6: neighbors:
n5: neighbors: 3, 4,
n5: [S] Opening stream to 0 with rate 1
n5: [S] Opening stream to 2 with rate 255
n5: [S] Sending SME: SND 5 -[1]-> 0
n5: [S] Sending SME: SND 5 -[255]-> 2
n4: received: [5/4] : 3, 4
n4: [S] enqueued SME: SND 5 -[1]-> 0 RCV
n4: [S] enqueued SME: SND 5 -[255]-> 2 RCV
n4: neighbors: 2, 5,
n4: [S] Forwarding SME: SND 5 -[1]-> 0
n4: [S] Forwarding SME: SND 5 -[255]-> 2
n3: received: [5/4] : 3, 4
n6: received: [5/4] : 3, 4
n3: neighbors: 2, 5,
n2: received: [4/3] : 2, 5
n2: received: [5/4] : 3, 4
n2: [S] enqueued SME: SND 5 -[1]-> 0 RCV
n2: [S] enqueued SME: SND 5 -[255]-> 2 RCV
n5: received: [4/3] : 2, 5
n2: received: [3/3] : 2, 5
n2: neighbors: 1, 3, 4,
n2: [S] Forwarding SME: SND 5 -[1]-> 0
n2: [S] Forwarding SME: SND 5 -[255]-> 2
n5: received: [3/3] : 2, 5
n3: received: [2/2] : 1, 3, 4
n4: received: [2/2] : 1, 3, 4
n1: received: [2/2] : 1, 3, 4
n1: received: [4/3] : 2, 5
n1: received: [5/4] : 3, 4
n1: [S] enqueued SME: SND 5 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 5 -[255]-> 2 RCV
```

```
n1: neighbors: 2,
n1: [S] Forwarding SME: SND 5 -[1]-> 0
n1: [S] Forwarding SME: SND 5 -[255]-> 2
n0: Received topology :[1/1->0]:0, 2
n0: Received topology :[3/3->2]:2
n0: Received topology :[2/2->1]:1, 3, 4
n0: [S] Opened stream 5 -[1]-> 0
n0: [S] Opened stream 5 -[255]-> 2
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 4]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n0: 2 -[1]-> 0
n0: 2 -[255]-> 7
n0: 3 -[1]-> 0
n0: 3 -[255]-> 8
n0: 4 -[1]-> 0
n0: 4 -[255]-> 1
n0: 5 -[1]-> 0
n0: 5 -[255]-> 2
n2: received: [1/1] : 0, 2
-----------------------------------------------
n7: neighbors:
n6: neighbors: 5,
n6: [S] Opening stream to 0 with rate 1
n6: [S] Opening stream to 3 with rate 255
n6: [S] Sending SME: SND 6 -[1]-> 0
n6: [S] Sending SME: SND 6 -[255]-> 3
n5: received: [6/5] : 5
n5: [S] enqueued SME: SND 6 -[1]-> 0 RCV
n5: [S] enqueued SME: SND 6 -[255]-> 3 RCV
n5: neighbors: 3, 4, 6,
n5: [S] Forwarding SME: SND 6 -[1]-> 0
n5: [S] Forwarding SME: SND 6 -[255]-> 3
n7: received: [6/5] : 5
n4: received: [5/4] : 3, 4, 6
n4: received: [6/5] : 5
n4: [S] enqueued SME: SND 6 -[1]-> 0 RCV
n4: [S] enqueued SME: SND 6 -[255]-> 3 RCV
n4: neighbors: 2, 5,
n4: [S] Forwarding SME: SND 6 -[1]-> 0
n4: [S] Forwarding SME: SND 6 -[255]-> 3
n3: received: [5/4] : 3, 4, 6
n6: received: [5/4] : 3, 4, 6
n3: neighbors: 2, 5,
n2: received: [4/3] : 2, 5
n2: received: [5/4] : 3, 4, 6
n2: received: [6/5] : 5
n2: [S] enqueued SME: SND 6 -[1]-> 0 RCV
n2: [S] enqueued SME: SND 6 -[255]-> 3 RCV
n5: received: [4/3] : 2, 5
n2: received: [3/3] : 2, 5
n2: neighbors: 1, 3, 4,
n2: [S] Forwarding SME: SND 6 -[1]-> 0
n2: [S] Forwarding SME: SND 6 -[255]-> 3
n5: received: [3/3] : 2, 5
n3: received: [2/2] : 1, 3, 4
n4: received: [2/2] : 1, 3, 4
```

```
n1: received: [2/2] : 1, 3, 4
n1: received: [3/3] : 2, 5
n1: received: [4/3] : 2, 5
n1: [S] enqueued SME: SND 6 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 6 -[255]-> 3 RCV
n1: neighbors: 2,
n1: [S] Forwarding SME: SND 6 -[1]-> 0
n1: [S] Forwarding SME: SND 6 -[255]-> 3
n0: Received topology :[1/1->0]:0, 2
n0: Received topology :[4/3->2]:2, 5
n0: Topology added 4 -> 5
n0: Received topology :[5/4->4]:3, 4
n0: Topology added 5 -> 3
n0: [S] Opened stream 6 -[1]-> 0
n0: [S] Opened stream 6 -[255]-> 3
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 4]
n0: [3 - 5]
n0: [4 - 5]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n0: 2 -[1]-> 0
n0: 2 -[255]-> 7
n0: 3 -[1]-> 0
n0: 3 -[255]-> 8
n0: 4 -[1]-> 0
n0: 4 -[255]-> 1
n0: 5 -[1]-> 0
n0: 5 -[255]-> 2
n0: 6 -[1]-> 0
n0: 6 -[255]-> 3
n2: received: [1/1] : 0, 2
-----------------------------------------------
n7: neighbors: 6,
n7: [S] Opening stream to 0 with rate 1
n7: [S] Opening stream to 4 with rate 255
n7: [S] Sending SME: SND 7 -[1]-> 0
n7: [S] Sending SME: SND 7 -[255]-> 4
n6: received: [7/6] : 6
n6: [S] enqueued SME: SND 7 -[1]-> 0 RCV
n6: [S] enqueued SME: SND 7 -[255]-> 4 RCV
n6: neighbors: 5, 7,
n6: [S] Forwarding SME: SND 7 -[1]-> 0
n6: [S] Forwarding SME: SND 7 -[255]-> 4
n5: received: [6/5] : 5, 7
n5: received: [7/6] : 6
n5: [S] enqueued SME: SND 7 -[1]-> 0 RCV
n5: [S] enqueued SME: SND 7 -[255]-> 4 RCV
n5: neighbors: 3, 4, 6,
n5: [S] Forwarding SME: SND 7 -[1]-> 0
n5: [S] Forwarding SME: SND 7 -[255]-> 4
n7: received: [6/5] : 5, 7
n4: received: [5/4] : 3, 4, 6
n4: received: [6/5] : 5, 7
n4: received: [7/6] : 6
n4: [S] enqueued SME: SND 7 -[1]-> 0 RCV
n4: [S] enqueued SME: SND 7 -[255]-> 4 RCV
n4: neighbors: 2, 5,
```

```
n4: [S] Forwarding SME: SND 7 -[1]-> 0
n4: [S] Forwarding SME: SND 7 -[255]-> 4
n3: received: [5/4] : 3, 4, 6
n6: received: [5/4] : 3, 4, 6
n3: neighbors: 2, 5,
n2: received: [4/3] : 2, 5
n2: received: [5/4] : 3, 4, 6
n2: received: [6/5] : 5, 7
n2: [S] enqueued SME: SND 7 -[1]-> 0 RCV
n2: [S] enqueued SME: SND 7 -[255]-> 4 RCV
n5: received: [4/3] : 2, 5
n2: received: [3/3] : 2, 5
n2: neighbors: 1, 3, 4,
n2: [S] Forwarding SME: SND 7 -[1]-> 0
n2: [S] Forwarding SME: SND 7 -[255]-> 4
n5: received: [3/3] : 2, 5
n3: received: [2/2] : 1, 3, 4
n4: received: [2/2] : 1, 3, 4
n1: received: [2/2] : 1, 3, 4
n1: received: [5/4] : 3, 4, 6
n1: received: [6/5] : 5, 7
n1: [S] enqueued SME: SND 7 -[1]-> 0 RCV
n1: [S] enqueued SME: SND 7 -[255]-> 4 RCV
n1: neighbors: 2,
n1: [S] Forwarding SME: SND 7 -[1]-> 0
n1: [S] Forwarding SME: SND 7 -[255]-> 4
n0: Received topology :[1/1->0]:0, 2
n0: Received topology :[2/2->1]:1, 3, 4
n0: Received topology :[3/3->2]:2, 5
n0: [S] Opened stream 7 -[1]-> 0
n0: [S] Opened stream 7 -[255]-> 4
n0: [T] Current topology:
n0: [0 - 1]
n0: [1 - 2]
n0: [2 - 3]
n0: [2 - 4]
n0: [3 - 5]
n0: [4 - 5]
n0: [S] Streams opened:
n0: 1 -[1]-> 0
n0: 1 -[255]-> 6
n0: 2 -[1]-> 0
n0: 2 -[255]-> 7
n0: 3 -[1]-> 0
n0: 3 -[255]-> 8
n0: 4 -[1]-> 0
n0: 4 -[255]-> 1
n0: 5 -[1]-> 0
n0: 5 -[255]-> 2
n0: 6 -[1]-> 0
n0: 6 -[255]-> 3
n0: 7 -[1]-> 0
n0: 7 -[255]-> 4
n2: received: [1/1] : 0, 2
```

8.3 Schedule downlink and data phase

The schedule, as previously stated, is statically coded and distributed. The nodes set as senders in them will send an *Hello* message during the assigned timeslot. A first example is realized in the diamond topology in which the master node sends a message in the first timeslot to node 1, which forwards it to node 3. Once received, in the next turn it replies forwarding the reply to node 1 which then reaches the master node. The output showing the schedule distribution to achieve this result and how the messages travel the network is shown in listing 8.9. In lines 2-3 the distributed schedule is shown. It is about 2 additions (*"+"* symbol at line start), one with id 0 and one with id 1. The first one spans two hops. The first transmission is from node 0 to node 1 (node id is enclosed in square parenthesis) during data slot 0 (the slots are in the middle of arrows, representing transmissions, preceded by *"at"* symbols), while the second one is to node 3 at data slot 1. The syntax in the logs is repeated for all the examples.

Listing 8.9: Ping-pong schedule distribution and data transmission on the diamond topology

```
1  n0: [SC] Sending schedule delta of 2 packets:
2  n0:     + 0 [0] -@0-> [1] -@1-> [3]
3  n0:     + 1 [3] -@2-> [1] -@3-> [0]
4  n1: [SC] Receive #0 @0 <- 0 forwardee
5  n1: [SC] Send #0 @1 -> 3 forwarder
6  n1: [SC] Receive #1 @2 <- 3 forwardee
7  n1: [SC] Send #1 @3 -> 0 forwarder
8  n3: [SC] Receive #0 @1 <- 1 receiver
9  n3: [SC] Send #1 @2 -> 1 sender
10 n0: [D] Sent packet with size 32 at 152362156000
11 n1: [D] Received forwarded with size 32 packet at 152362156000 with
       error 0
12 n1: [D] Forwarded packet with size 32 at 152368606000
13 n3: [D] Received packet with size 32 at 152368606000:
14 n3: Hello, I am 0
15 n3: [D] Sent packet with size 32 at 152374606000
16 n1: [D] Received forwarded with size 32 packet at 152374606000 with
       error 0
17 n1: [D] Forwarded packet with size 32 at 152380606000
18 n0: [D] Received packet with size 32 at 152380606000:
19 n0: Hello, I am 3
```

Another example, involves the master node as router. This case, also run on the diamond topology, shows packet going from node 2 to node 3 passing by the

master node and then from node 1, starting from the first timeslot. The log is shown in listing 8.10.

Listing 8.10: Routing though the master node in the diamond topology

```
1  n0: [SC] Sending schedule delta of 1 packets:
2  n0:      + 0 [2] -@0-> [0] -@1-> [1] -@2-> [3]
3  n1: [SC] Receive #0 @1 <- 0 forwardee
4  n1: [SC] Send #0 @2 -> 3 forwarder
5  n2: [SC] Send #0 @0 -> 0 sender
6  n3: [SC] Receive #0 @2 <- 1 receiver
7  n2: [D] Sent packet with size 32 at 152362606000
8  n0: [D] Received forwarded with size 32 packet at 152362606000 with
       error 0
9  n0: [D] Forwarded packet with size 32 at 152368156000
10 n1: [D] Received forwarded with size 32 packet at 152368156000 with
       error 0
11 n1: [D] Forwarded packet with size 32 at 152374606000
12 n3: [D] Received packet with size 32 at 152374606000:
13 n3: Hello, I am 2
```

The last example, that shows how two nodes at the opposite ends of a network can send themselves a message contemporarily, is displayed in listing 8.11. In this case the transmission is accomplished in a double tailed kite topology in which messages travel during timeslots 0 and 1 from the ends to nodes 2 and 5 (lines 4-5, 8 and 18, 22-23, 25 for the scheduling, 26-33 for the data), respectively. Then in dataslot 2, data is sent from 2 to 3 (lines 9, 12 and 34-35), in slot 3 from 5 to 4 (lines 14, 19 and 36-37). Later, in dataslot 4, 4 sends its data to 2 (lines 10, 15, 38-39) and in 5, node 3 sends to 5 (lines 13, 16, 40, 42). Finally, the last two hops are traveled and the data reaches the final destinations (lines 6, 7, 11, 17, 20-21, 24 and 41, 43-51). Node 0 is reached in dataslot 6 and node 7 in dataslot 7.

Listing 8.11: Two nodes sending each other a packet contemporarily in the double tail kite topology

```
1  n0: [SC] Sending schedule delta of 2 packets:
2  n0:      + 0 [0] -@0-> [1] -@1-> [2] -@2-> [3] -@4-> [5] -@5-> [6] -
       @6-> [7]
3  n0:      + 1 [7] -@0-> [6] -@1-> [5] -@3-> [4] -@5-> [2] -@6-> [1] -
       @7-> [0]
4  n1: [SC] Receive #0 @0 <- 0 forwardee
```

```
n1: [SC] Send #0 @1 -> 2 forwarder
n1: [SC] Receive #1 @6 <- 2 forwardee
n1: [SC] Send #1 @7 -> 0 forwarder
n2: [SC] Receive #0 @1 <- 1 forwardee
n2: [SC] Send #0 @2 -> 3 forwarder
n2: [SC] Receive #1 @5 <- 4 forwardee
n2: [SC] Send #1 @6 -> 1 forwarder
n3: [SC] Receive #0 @2 <- 2 forwardee
n3: [SC] Send #0 @4 -> 5 forwarder
n4: [SC] Receive #1 @3 <- 5 forwardee
n4: [SC] Send #1 @5 -> 2 forwarder
n5: [SC] Receive #0 @4 <- 3 forwardee
n5: [SC] Send #0 @5 -> 6 forwarder
n5: [SC] Receive #1 @1 <- 6 forwardee
n5: [SC] Send #1 @3 -> 4 forwarder
n6: [SC] Receive #0 @5 <- 5 forwardee
n6: [SC] Send #0 @6 -> 7 forwarder
n6: [SC] Receive #1 @0 <- 7 forwardee
n6: [SC] Send #1 @1 -> 5 forwarder
n7: [SC] Receive #0 @6 <- 6 receiver
n7: [SC] Send #1 @0 -> 6 sender
n0: [D] Sent packet with size 32 at 152362156000
n1: [D] Received forwarded with size 32 packet at 152362156000 with
     error 0
n7: [D] Sent packet with size 32 at 152362606000
n6: [D] Received forwarded with size 32 packet at 152362606000 with
     error 0
n1: [D] Forwarded packet with size 32 at 152368606000
n6: [D] Forwarded packet with size 32 at 152368606000
n2: [D] Received forwarded with size 32 packet at 152368606000 with
     error 0
n5: [D] Received forwarded with size 32 packet at 152368606000 with
     error 0
n2: [D] Forwarded packet with size 32 at 152374606000
n3: [D] Received forwarded with size 32 packet at 152374606000 with
     error 0
n5: [D] Forwarded packet with size 32 at 152380606000
n4: [D] Received forwarded with size 32 packet at 152380606000 with
     error 0
n3: [D] Forwarded packet with size 32 at 152386606000
n5: [D] Received forwarded with size 32 packet at 152386606000 with
     error 0
n4: [D] Forwarded packet with size 32 at 152392606000
n5: [D] Forwarded packet with size 32 at 152392606000
n2: [D] Received forwarded with size 32 packet at 152392606000 with
     error 0
n6: [D] Received forwarded with size 32 packet at 152392606000 with
     error 0
n2: [D] Forwarded packet with size 32 at 152398606000
n6: [D] Forwarded packet with size 32 at 152398606000
n1: [D] Received forwarded with size 32 packet at 152398606000 with
     error 0
n7: [D] Received packet with size 32 at 152398606000:
n7: Hello, I am 0
n1: [D] Forwarded packet with size 32 at 152404606000
n0: [D] Received packet with size 32 at 152404606000:
n0: Hello, I am 7
```

Chapter 9

Conclusions

In this thesis a new Medium Access Control protocol based on 802.15.4 2.4 GHz O-QPSK physical layer was presented. The target was to make a protocol that solves the required delay and the strict data rate constraints in multi-hop WSNs. The starting point was time synchronization implemented at the physical layer, using Glossy and FLOPSYNC-2 for error compensation. This uses a flooding mechanism which is initialized by the root node and reaches all the nodes traveling all the network hop by hop. This workflow made centralized network management more interesting and convenient, and therefore it was chosen. However, for achieving this, collection of information about the network in order to manage it is necessary. More in detail, the master node needs information about the topology and the transmission requirements of the nodes, namely streams. Then, the master node computes a schedule that will be distributed to the nodes, making the nodes able to send data to other nodes and forward data through multiple hops. The process repeats after a predefined timestamp, identifying the slotframe. The actual distribution of the phases within the slotframe is managed by an interleaving function whose job is distributing evenly the different phases' timeslots over the whole slotframe. Phases are firstly divided into two macrogroups: downlink and uplink. The first is used for time synchronization and schedule distribution, the second for transmitting stream management and topology information until reaching the master node, achieved by traveling one hop per slotframe. The managed topologies are either mesh

or tree, defining different contents for the topology collection part of the uplink message. A software engineering project of the protocol has been realized using UML class diagrams for representing its operational structure and a possible implementation. Then it was realized and simulated in OMNeT++, analyzing the resulting logs phase by phase.

A physical implementation of the protocol would be helpful for having an hardware test bed, which would be a better environment for running tests and knowing what the actual performances of the protocol are. It would be interesting to test it by varying the different dimensions (number of nodes, hops, slotframe duration, etc.), the topology collected, the deployment configurations (e.g. how performances change from a balanced tree to a mesh), the possible streams configurations in terms of data rate, number of involved nodes and amount of opened streams. Vice-versa, the constraints imposed by the protocol, like the proportions among the phases within the slotframe and how the slotframe is defined, should be better investigated, in order to understand if relaxing them would improve it. But, before of that, a real scheduler needs to be implemented, requiring the problem to be deeply investigated and formalized. This would also require giving an actual meaning to the data rate field of SMEs. Moreover, it would also give a better knowledge about the amount of information needed to be transmitted to the nodes, making it simpler to analyze the proposed schedule distribution formats and choosing among them or proposing a new one. Another important aspect is the security. The protocol should be able to transmit encrypted data, in order to protect intruders for joining the network or being able to receive message content. Then, under certain conditions, enabling nodes to obtain a network address without statically configure it would be desirable, therefore a protocol (similar to DHCP but at level 2, so not based on mapping MAC to IP address and vice versa) for realizing so could be helpful.

The proposed protocol tries to fill a gap, making it possible for real-time and industry automation systems being able to rely on wireless communication. This technology has been a taboo for them and what this protocol tries to do is to break it down, even with the creation of new research questions.

Bibliography

[1] Wikipedia contributors. (2018-01). Tcp/ip protocol suite, [Online]. Available: https://en.wikipedia.org/wiki/Internet_protocol_suite.

[2] ——, (2018-01). Iso/osi model, [Online]. Available: https://en.wikipedia.org/wiki/OSI_model.

[3] "Ieee standard for ethernet", *IEEE Std 802.3-2015 (Revision of IEEE Std 802.3-2012)*, pp. 1–4017, Mar. 2016. DOI: 10.1109/IEEESTD.2016.7428776.

[4] "Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications", *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, Dec. 2016. DOI: 10.1109/IEEESTD.2016.7786995.

[5] "Ieee standard for low-rate wireless networks", *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pp. 1–709, Apr. 2016. DOI: 10.1109/IEEESTD.2016.7460875.

[6] ——, (2018-01). Csma/cd, [Online]. Available: https://en.wikipedia.org/wiki/CSMA/CD.

[7] ——, (2018-01). Csma/ca, [Online]. Available: https://en.wikipedia.org/wiki/CSMA/CA.

[8] ZigBee alliance. (2018-01). Zigbee, [Online]. Available: http://www.zigbee.org/.

[9] P. Huang, L. Xiao, S. Soltani, M. W. Mutka, and N. Xi, "The evolution of mac protocols in wireless sensor networks: A survey", *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 101–120, First 2013, ISSN: 1553-877X. DOI: 10.1109/SURV.2012.040412.00105.

[10] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks", in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, 2002, 1567–1576 vol.3. DOI: 10.1109/INFCOM.2002.1019408.

[11] ——, "Medium access control with coordinated adaptive sleeping for wireless sensor networks", *IEEE/ACM Transactions on Networking*, vol. 12, no. 3, pp. 493–506, Jun. 2004, ISSN: 1063-6692. DOI: 10.1109/TNET.2004.828953.

[12] G. Lu, B. Krishnamachari, and C. S. Raghavendra, "An adaptive energy-efficient and low-latency mac for data gathering in wireless sensor networks", in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, Apr. 2004, pp. 224–. DOI: 10.1109/IPDPS.2004.1303264.

[13] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient collision-free medium access control for wireless sensor networks", in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, ser. SenSys '03, Los Angeles, California, USA: ACM, 2003, pp. 181–192, ISBN: 1-58113-707-9. DOI: 10.1145/958491.958513. [Online]. Available: http://doi.acm.org/10.1145/958491.958513.

[14] W. Z. Song, R. Huang, B. Shirazi, and R. LaHusen, "Treemac: Localized tdma mac protocol for real-time high-data-rate sensor networks", in *2009 IEEE International Conference on Pervasive Computing and Communications*, Mar. 2009, pp. 1–10. DOI: 10.1109/PERCOM.2009.4912757.

[15] K. Moriyama and Y. Zhang, "An efficient distributed tdma mac protocol for large-scale and high-data-rate wireless sensor networks", in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, Mar. 2015, pp. 84–91. DOI: 10.1109/AINA.2015.170.

[16] Q. Yu, J. Chen, Y. Fan, X. Shen, and Y. Sun, "Multi-channel assignment in wireless sensor networks: A game theoretic approach", in *2010 Proceedings IEEE INFOCOM*, Mar. 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5461935.

[17] S. Petersen and S. Carlsen, "Wirelesshart versus isa100.11a: The format war hits the factory floor", *IEEE Industrial Electronics Magazine*, vol. 5, no. 4, pp. 23–34, Dec. 2011, ISSN: 1932-4529. DOI: 10.1109/MIE.2011.943023.

[18] Wikipedia contributors. (2018-01). Cipher Block Chaining Message Authentication Code, [Online]. Available: https://en.wikipedia.org/wiki/CBC-MAC.

[19] ——, (2018-01). Advanced encryption standard, [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[20] D. De Guglielmo, S. Brienza, and G. Anastasi, "Ieee 802.15. 4e: A survey", *Computer Communications*, vol. 88, pp. 1–24, 2016.

[21] 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)*, Sep. 2012, pp. 327–332. DOI: 10.1109/PIMRC.2012.6362805.

[22] M. Ojo and S. Giordano, "An efficient centralized scheduling algorithm in ieee 802.15.4e tsch networks", in *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, Oct. 2016, pp. 1–6. DOI: 10.1109/CSCN.2016.7785164.

[23] N. Accettura, M. R. Palattella, G. Boggia, L. A. Grieco, and M. Dohler, "Decentralized traffic aware scheduling for multi-hop low power lossy networks in the internet of things", in *2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, Jun. 2013, pp. 1–6. DOI: 10.1109/WoWMoM.2013.6583485.

[24] K. Muraoka, T. Watteyne, N. Accettura, X. Vilajosana, and K. S. J. Pister, "Simple distributed scheduling with collision detection in tsch net-

works", *IEEE Sensors Journal*, vol. 16, no. 15, pp. 5848–5849, Aug. 2016, ISSN: 1530-437X. DOI: 10.1109/JSEN.2016.2572961.

[25] A. K. Demir and S. Bilgili, "Diva: A distributed divergecast scheduling algorithm for ieee 802.15.4e tsch networks", *Wireless Networks*, Sep. 2017, ISSN: 1572-8196. DOI: 10.1007/s11276-017-1580-4. [Online]. Available: https://doi.org/10.1007/s11276-017-1580-4.

[26] T. Chang, T. Watteyne, K. Pister, and Q. Wang, "Adaptive synchronization in multi-hop tsch networks", *Computer Networks*, vol. 76, pp. 165–176, 2015, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2014.11.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128614003922.

[27] A. Elsts, S. Duquennoy, X. Fafoutis, G. Oikonomou, R. Piechocki, and I. Craddock, "Microsecond-accuracy time synchronization using the ieee 802.15.4 tsch protocol", in *2016 IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops)*, Nov. 2016, pp. 156–164. DOI: 10.1109/LCN.2016.042.

[28] Linear Technologies. (2018-02). Smartmesh ip™, [Online]. Available: http://www.linear.com/products/smartmesh_ip.

[29] M. Köstler, F. Kauer, T. Lübkert, V. Turau, J. Scholz, and A. von Bodisco, "Towards an open source implementation of the ieee 802.15. 4 dsme link layer", *Proceedings of the 15. GI/ITG KuVS Fachgespräch Sensornetze, J. Scholz and A. von Bodisco, Eds. University of Applied Sciences Augsburg, Dept. of Computer Science*, 2016.

[30] Wikipedia contributors. (2018-01). Hidden node problem, [Online]. Available: https://en.wikipedia.org/wiki/Hidden_node_problem.

[31] 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*, Apr. 2011, pp. 73–84.

[32] T. Schmid, P. Dutta, and M. B. Srivastava, "High-resolution, low-power time synchronization an oxymoron no more", in *Proceedings of the 9th*

ACM/IEEE International Conference on Information Processing in Sensor Networks, ser. IPSN '10, Stockholm, Sweden: ACM, 2010, pp. 151–161, ISBN: 978-1-60558-988-6. DOI: 10.1145/1791212.1791231. [Online]. Available: http://doi.acm.org/10.1145/1791212.1791231.

[33] F. Terraneo, F. Riccardi, and A. Leva, "Jitter-compensated vht and its application to wsn clock synchronization", in *2017 IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2017, pp. 277–286. DOI: 10.1109/RTSS.2017.00033.

[34] 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*, Dec. 2014, pp. 11–20. DOI: 10.1109/RTSS.2014.14.

[35] F. Terraneo, A. Leva, S. Seva, M. Maggio, and A. V. Papadopoulos, "Reverse flooding: Exploiting radio interference for efficient propagation delay compensation in wsn clock synchronization", in *2015 IEEE Real-Time Systems Symposium*, Dec. 2015, pp. 175–184. DOI: 10.1109/RTSS.2015.24.

[36] Federico Terraneo. (2018-01). Miosix kernel, [Online]. Available: https://miosix.org/.

[37] OpenSim Ltd. (2018-02). Omnet++, [Online]. Available: https://www.omnetpp.org/.