Session Layer Bounded Latency in Wireless Mesh Networks

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Luca Conterio**

Student ID: 920261
Advisor: Federico Terraneo
Co-advisors: Prof. William Fornaciari
Academic Year: 2021-2022

Abstract

During the last years, **wireless networks** gained increasing importance in humans life, becoming leading actors in many scenarios. Consider for example the incredible spread of mobile devices and those related to the Internet of Things, together with commonly used technologies such as WiFi or ZigBee. Wireless technologies are usually targeted either to high-performance or **low-power** networks and, in this context, **TDMH** (*Time Deterministic Multi-Hop*) presents itself as a wireless communication stack capable of providing efficient low-power medium access control. Being also able to manage **multi-hop mesh** networks and adapt to network topology changes, TDMH is suitable for **real-time** applications, running on battery-powered devices and targeting **industrial control** use cases. In such applications, guarantees on the expected **latency** bounds for all the packets flowing through the network is crucial and, to the best of the author's knowledge, existing technologies are unable to combine then with the support for multi-hop network topologies. This thesis presents an extension to the existing TDMH **session layer**, through the design and implementation of two new interfaces between the application and the underlying network stack layers, enforcing guarantees on the latency bounds up to the upper ones. The optimized mechanisms are presented together with **reliability** experiments and the validation of the proposed solution through an exemplary use case of **distributed feedback control** over a wireless network. The conducted experiments confirm that TDMH is suitable for real-time control applications, being able to achieve high network **reliability** levels and to provide **deterministic latencies**.

Keywords: real-time, latency, wireless networks, session layer, industrial control

Abstract in Lingua Italiana

Negli ultimi anni le **reti wireless** hanno acquisito un'importanza sempre maggiore nella vita di tutti i giorni, diventando attori di primo piano in molti scenari. Si consideri ad esempio l'incredibile diffusione dei dispositivi mobili e relativi all'Internet of Things, assieme a tecnologie usate comunemente come WiFi o ZigBee. Le tecnologie wireless sono generalmente destinate a reti ad alte prestazioni o a **basso consumo energetico** e in questo contesto **TDMH** (*Time Deterministic Multi-Hop*) si presenta come uno stack di comunicazione wireless capace di fornire accesso al mezzo di comunicazione in modo efficiente e a bassa potenza. Essendo anche in grado di gestire reti **mesh multi-hop** e di adattarsi ai cambiamenti della topologia di rete, TDMH è adatto per applicazioni **real-time** eseguite su dispositivi alimentati a batteria e mirati a casi d'uso di **controllo industriale**. In tali applicazioni è cruciale fornire garanzie sui limiti di **latenza** per tutti i pacchetti che fluiscono attraverso la rete e, al meglio delle conoscenze dell'autore, le tecnologie esistenti non sono in grado di combinarle con il supporto a topologie di rete multi-hop. Questa tesi presenta un'estensione dell'esistente **livello di sessione** di TDMH attraverso la progettazione e l'implementazione di due nuove interfacce fra l'applicazione e lo stack di rete sottostante, fornendo garanzie sui limiti di latenza fino ai livelli più alti. I meccanismi e le ottimizzazioni realizzati vengono presentati insieme a esperimenti di **affidabilità** e alla validazione della soluzione proposta attraverso un caso d'uso esemplare di **controllo a feedback**, **distribuito** su una rete wireless. Gli esperimenti effettuati mostrano come TDMH sia adatto ad applicazioni di controllo real-time, essendo capace di raggiungere alti livelli di affidabilità di rete e di fornire **latenze deterministiche**.

Parole chiave: real-time, latenza, reti wireless, livello di sessione, controllo industriale

Contents

Abstract **i**

Abstract in Lingua Italiana **iii**

Contents **v**

1 Introduction **1**

1.1 Goal of This Work 1

1.2 Author's Contribution 2

2 Literature Review **3**

2.1 Low Power Wireless Networks 3

2.1.1 Low Power Listening 3

2.1.2 Channel Access Methods 4

2.2 Low Power Protocols 5

2.2.1 TSCH 5

2.2.2 DSME 6

2.2.3 LLDN 6

2.2.4 ABMP 7

2.2.5 TDMH Comparison 7

3 TDMH Overview **9**

3.1 Time Synchronization 9

3.2 Network Nodes 10

3.2.1 Master Node 10

3.2.2 Dynamic Nodes 11

3.3 Time Division 11

3.4 Protocol Phases 12

3.4.1 Downlink Phase 13

3.4.2 Uplink Phase 14

3.4.3 Data Phase 15
3.5 TDMH Network Stack 15
3.5.1 Network Layer 16
3.5.2 Data Link Layer 19
3.5.3 Session Layer 20
3.5.4 Physical Layer 24
3.6 Cryptography 25

4 Problem Statement **27**
4.1 Real-Time Systems and Latency 27
4.2 TDMH Streams API 28
4.3 Weak Points 29
4.4 Real Example 31
4.5 Desired Behavior 33

5 Proposed Solution **35**
5.1 Callbacks API 35
5.1.1 Overview 35
5.1.2 Implementation 35
5.2 Write/Wait API 41
5.2.1 Overview 42
5.2.2 Implementation 43
5.3 APIs Comparison 61
5.3.1 Callbacks API 61
5.3.2 Write/Wait API 62
5.4 APIs Interoperability 63
5.5 Data Phase 64
5.5.1 "As Soon as Possible" Policy 64
5.5.2 "As Late as Possible" Policy 67
5.5.3 Radio Startup Time and Cryptography 68
5.6 Schedule Expansion 69
5.6.1 Schedule Expansion Module 70
5.6.2 Schedule Distribution 71
5.6.3 Re-Transmitted Schedules 76
5.6.4 Streams Wake-Up Lists 76

6 Latency Computation **79**
6.1 Minimum Theoretical Latency 79

6.1.1 Radio Startup Time and Crypto 81
6.2 Callbacks API Latency 82
6.3 Write/Wait API Latency 84
6.4 Heterogeneous APIs Latency 87
6.5 Real Examples 89
6.5.1 Callbacks API 90
6.5.2 Write/Wait API 91
6.5.3 Heterogeneous APIs 93
6.6 Re-Scheduling Latency 94

7 Experiments **99**
7.1 Experiments Setup 99
7.1.1 WandStem Nodes 99
7.1.2 Latency Profiling 100
7.2 Simulations 100
7.3 Wireless Validation Experiments 102
7.3.1 Callbacks API 103
7.3.2 Write/Wait API 114
7.4 Distributed Control Loop 126
7.4.1 Setup 126
7.4.2 Results 129

8 Conclusions and Future Developments **135**
8.1 Future Developments 136

Bibliography **137**

List of Figures **141**

List of Tables **147**

Acknowledgements **149**

1 | Introduction

During the last decades, wireless networks gained increasing importance in a variety of different contexts. Wireless networks are nowadays widespread and used in more and more application fields. They are in fact commonly used in private homes but also in offices and productive plants. Wireless networks particularly increased their pervasiveness in the field of the Internet of Things and industrial control. The advantages that they have over wired networks include **mobility** and **flexibility**, for example when the number of devices needs to be updated or the network topology is evolving. Wireless networks allow limiting the infrastructure updates, which would be instead required in a wired setting. In addition, they have on their side lower cost and lower power consumption. Mobile devices, such as wearable ones, need to be energetically efficient, mainly due to their reduced size and the impossibility to carry a large battery with themselves.

In **industrial control** scenarios, **Wireless Sensor Networks** (WSNs) can be used to connect different nodes that are able to read and forward sensors information to other ones that are instead in charge of driving actuators. Indeed, the devices that compose the WSN cooperate in order to carry out a task. At the same time, industrial control applications have stricter requirements, especially related to time: **real-time constraints** are usually needed, with guarantees on data packets **latency** and **periodicity**.

Starting from the described background, **TDMH** (*Time Deterministic Multi Hop*) [25] was specifically designed for real-time industrial control applications over wireless networks. TDMH provides a deterministic *Medium Access Control* (MAC) mechanism, which allows fulfilling real-time and **low power** requirements, providing also the flexibility of **mesh** network topologies.

1.1. Goal of This Work

In real-time applications, for example in the field of industrial control, a **consistent and predictable latency** over time is something much more important than having the lowest possible latency.

Keeping in mind the application fields that TDMH targets, the goal of this thesis work is to provide guarantees on the minimum and maximum latency values of every data packet flowing through the wireless network.
Following TDMH phylosophy, it must be possible for the application to access the network stack through a limited set of primitives, while the underlying layers take care of managing the whole protocol. Then, the goal is to avoid delegating decisions to the application code and to the user. Any enforced guarantee has to be embedded into the protocol stack itself.

1.2. Author's Contribution

The main part of author's contribution consists in the analysis of the existing TDMH session layer implementation and its extension, aiming at providing an end-to-end **bounded latency** among network nodes. The mentioned extension includes the design and implementation of two APIs (*Application Programming Interface*) used to interface the application code and the session layer itself. Some required updates that were made to two of the main TDMH protocol mechanisms, the *data phase* and the *schedule expansion*, are also presented. The newly introduced features are evaluated and validated through the conducted experiments, proving that they do not affect the overall network reliability provided by TDMH. In the final experiment, TDMH is used in a **distributed control loop** over the wireless network, demonstrating its capabilities in this field.

2 | Literature Review

2.1. Low Power Wireless Networks

Among the successful wireless network protocols, some commercial and widely adopted solutions exist. They include technologies such as WiFi [2], Bluetooth [4], and ZigBee [5]. The WiFi protocol was designed to provide high data rates over TCP/IP networks. It aims at providing low latency but with no guarantees about it, especially when the traffic increases and the network becomes congested. Indeed, it enforces guarantees on the packets delivery, which is ensured, with manifest drawbacks on their latency. Moreover, WiFi was not designed to be a low-power protocol and its high power consumption makes it non-applicable to battery-powered devices, whose goal is to maximize batteries duration by minimizing the energy requirements.
Bluetooth was instead designed for low bandwidth and star network topology scenarios. This setting is indeed very common for smartphone accessories. Bluetooth radio range is limited to a few meters and the supported network topology is not enough for industrial control applications.
The last one, ZigBee, is a low-power targeted protocol that is capable of low data rates. Its advantage is to support also multi-hop network topology, by dividing network nodes into two categories: *router devices* and *end devices*. The drawback is that only end devices can enter a power-saving mode and, so, can be battery-powered, while router devices are better suited for a power supply.

2.1.1. Low Power Listening

Among the existing techniques to reduce the power consumption in wireless protocols, the one specified in the IEEE 802.15.4 standard [1] is called **low power listening** [17]. It is adopted for example by the B-MAC protocol, which is built on top of the mentioned standard. According to the specification, sender nodes transmit a long preamble, that can last for example 100 ms, before every packet, whose transmission instead lasts a few milliseconds. The receiver on the other side, periodically checks if the communication channel is free or a carrier-wave already exists. If it is the case, an incoming packet is being sent over the channel. The receiver sensing mechanism is called *carrier sensing*.

This check on the channel occupation gives some statistical certainty on packets reception, according to the period at which these some checks are performed.
Anyway, this approach reduces the energy needed for receiving, being more power-consuming on the transmission side, but its main drawback is that it does not eliminate collisions. As a consequence, it does not provide any guarantee on the delay in accessing the communication channel, making it neither deterministic nor predictable.
Moreover, due to the long packet preambles needed, the communication channel is not efficiently used.

2.1.2. Channel Access Methods

The choice of the best suitable *channel access method* is one of the main wireless protocol design challenges. In the setting of a wireless network in fact it may happen that multiple devices need to transmit or receive through using the same physical communication medium. If these devices happen to use the medium at the same time, **collisions** can take place. The consequent radio interference deteriorates the simultaneous transmissions, with possible data losses, since the recipient(s) would not be able to receive correctly formatted packets.

The mostly used approaches to solve the collisions problem are **CSMA/CA** (*Carrier Sensing Multiple Access with Collision Avoidance*) and **TDMA** (*Time Deterministic Multiple Access*) [26].

CSMA/CA

In CSMA/CA, when a network node has something to transmit, it has to check (listen) that the radio channel is not already occupied, i.e. no other node is transmitting. If the channel is free, the node can proceed with its own transmission. Otherwise, if the channel is occupied, the node executes the exponential **backoff** process, which is based on waiting a random time interval until the channel is sensed to be free.

CSMA/CA guarantees that, eventually, the node will be able to transmit. The price for avoiding collisions is then to have **no upper bound on the possible latency**, which is intrinsically unpredictable. This means that this mechanism is not compatible with applications in which real-time constraints have to be fulfilled and since there is no prior agreement on the time when packets will be sent, CSMA/CA protocols usually force devices to keep the radio transceiver in receive mode continuously, which is a concern in battery-operated devices.

TDMA

As opposed to CSMA/CA, TDMA addresses the problem of collisions. They are avoided by dividing time into discrete **time slots**. In each time slot then, at most one device at a time can transmit. The main drawback of TDMA is that it requires a distributed knowledge of the time **schedule**: every node must know which node is allowed to use the communication channel during each of the time slots. The need for a schedule computation implies some centralized or distributed orchestration and, thus, some form of **clock synchronization**, which complicates the protocol. On the other side, TDMA has the advantage of enforcing a **lower energy consumption**. Nodes can wake up and use the physical wireless medium only during their assigned time slot. The rest of the time nodes can simply save energy entering for example into a low power state. The time slots schedules also allow having deterministic **bounded latencies**, since **no collisions** can take place and there is no need of delaying transmissions.

2.2. Low Power Protocols

An upgraded version of the IEEE 802.15.4 exists and is called IEEE 802.15.4e. It specifies some new physical and MAC level standards. The most known and interesting ones are TSCH (Time Slotted Channel Hopping [14]), DSME (Deterministic and Synchronous Multi-channel Extension [13]), LLDN (Low Latency Deterministic Network [6]), and ABMP (Adaptive and Beacon-based Multi-Channel Protocol [9]).

2.2.1. TSCH

Among the cited standards, *Time Slotted Channel Hopping* (TSCH) is the most developed and the most present in research works. It is based on the peer-to-peer architecture presented in the IEEE 802.15.4 standard, but, instead of using the CSMA/CA channel access method, it implements a TDMA approach. In TSCH the active network nodes (called FFD, Fully Functioning Device, in IEEE 802.15.4) send beacons, which are used by the other nodes to synchronize. The network active nodes cooperate to build a schedule in a distributed manner. The computed schedule is used to enable the TDMA physical medium access and, thus, to assign communication links to channels and time slots.

Even though TSCH guarantees transmission reliability, but, at the same time, it does not provide any guarantee on the medium access delay bounds, which is a relevant drawback in the context of real-time systems.

2.2.2. DSME

A protocol that is able to handle multi-hop transmissions is the *Deterministic and Synchronous Multi-channel Extension* (DSME). In addition to being a time-synchronized protocol, it improves the IEEE 802.15.4 MAC standard. It has a multi-channel ability which means that it is able to use different radio channels at the same time (channel diversity).

The channel access mechanism that DSME employs is TDMA. Time is divided into transmission frames that are called multi-superframes and each of them is in turn divided into 16 superframe slots. Superframe slots are split into two parts:

- During the *Contention Access Period* (CAP), network nodes have to dispute the channel access. This is the most expensive part in terms of energy consumption, since nodes need to sense the physical channel to check if it is free.
- On the contrary, in the *Contention Free Period* (CFP) the single nodes are scheduled for transmission, so that each of them is assigned to specific time intervals.

In order to limit the power consumption, DSME uses a technique called *CAP reduction* that consists in limiting the usage of the CAP part and prioritizing the CFP usage.

2.2.3. LLDN

Another *Media Access Control* (MAC) protocol that is presented in the IEEE 802.15.4e standard is *Low Latency Deterministic Network* (LLDN). The main target of LLDN is the data collection from a set of sensors in the network, in fact it can handle up to 20 sensors transmitting data at $100Hz$. LLDN organizes time in superframes which in turn are divided into time slots: the nodes in the network access the superframe time slots through a CSMA/CA approach.

Even if LLDN enforces bounds on the maximum transmission latency, its main drawback consists in the fact that it only supports star network topologies, which are limited to a single hop.

2.2.4. ABMP

The last considered protocol based upon the IEEE 802.15.4e standard for the physical layer is the *Adaptive and Beacon-based Multi-Channel Protocol* (ABMP). Its main advantage is the ability of the network to dynamically allocate channels as a response to channels quality changes. Moreover, for what concerns the network topology, it supports the star and tree structures.

2.2.5. TDMH Comparison

Table 2.1 shows a comparison between the previously described protocols built on top of the IEEE 802.15.4 and TDMH.

It can be noticed from the table that there are two main categories of IEEE 802.15.4 protocols: those proving bounds on the achieved latencies, such as LLDN which is, however, limited to star topologies, and those that support multi-hop network structure. In the second category, providing guarantees on the latency bounds is not a common feature. TDMH tries to overcome the limitations of both categories by supporting generic mesh (and so multi-hop) topologies while, at the same time, providing a deterministic and predictable latency.

Feature	**TDMH**	**TSCH**	**DSME**	**LLDN**	**ABMP**
Multi-hop	✓	✓	✓		✓
Bounded latency	✓			✓	
Spatial redundancy	✓				
Temporal redundancy	✓	✓	Feasible	Feasible	Feasible
Topology Type	Mesh	Cluster-Tree	Cluster-Tree	Star	Star / Tree
Management	Centralized	Centr. / Distr.	Distributed	Centralized	Centr. / Distr.

Table 2.1: TDMH features compared to other IEEE 802.15.4 protocols.

Moreover, TDMH provides all the mentioned guarantees targeting **low-end single-core microcontrollers**, in which both the applicative code and the network stack run on the same processor. This is completely different from other approaches like Bolt [19] [12], which provides predictable messages transfer times among multiple processors and can also be applied to wireless networks.

3 | TDMH Overview

TDMH (*Time Deterministic Multi-Hop*) is a **real-time** protocol for **wireless sensor networks** (WSN) and **industrial control** applications, that is based on the IEEE 802.15.4 radio standard. It has a centralized network structure in which the main algorithms are executed by a master node. TDMH is also capable of **low power** consumption and it supports **mesh** networks, allowing **multi-hop** transmissions too, being also able to adapt to changes in the **network topology**. Low power is achieved through **time synchronization** among all the network nodes, so that they can access the physical transmission medium in a time deterministic fashion, which relies on *Time Deterministic Multiple Access* (TDMA). Time synchronization allows avoiding medium access contention, and, thus, carrier sensing or random exponential backoff: the nodes can wake up, and thus consume power, only during time slots in which they actually have something to send or receive. This mechanism also avoids non-deterministic delays when a node tries to get access to the physical transmission medium. It is then suitable for real-time applications and constraints. These constraints in the TDMH context are represented by the **period** of each communication channel between network nodes, which in turn are called **streams**.

TDMH is entirely written in C++ and implemented on top of *Miosix* [21], a real-time operating system developed by Terraneo Federico at Politecnico di Milano. Previous students Paolo Polidori, Federico Amedeo Izzo and Valeria Mazzola also contributed to the project through their master thesis. The TDMH project is open source and can be found on GitHub [22].

3.1. Time Synchronization

In order to provide a TDMA channel access mechanism, a fundamental role is played by the **time synchronization**, which is necessary to synchronize each node's internal clock. Employing a synchronization scheme, allows network nodes to avoid contention and collisions when accessing the radio channel. Carrier sensing indeed has a negative impact on power consumption and latency, since multiple attempts might be needed before a successful transmission occurs.

The algorithm employed by TDMH is FLOPSYNC-2 [23]. It was designed in order to compensate for possible **jitter**, temperature drift or errors caused by the PLL, avoiding clock errors that can lead to nodes desynchronization.
By using both a local clock model and a controller that is in charge of correcting non-idealities, FLOPSYNC-2 is able to reach a high precision. and a power consumption as low as $2.1\mu W$.
Even when the needed clock correction is significant, FLOPSYNC-2 guarantees the clock to be monotonic, meaning that adjustments are incrementally applied avoiding backward clock steps. This is not instead guaranteed when a node de-synchronizes from the network and has to re-join the network and synchronize again.

3.2. Network Nodes

Being TDMH a centralized protocol, a **master node** is always needed. All the other nodes are instead called **dynamic nodes**, since their status and their physical position may change over time.

3.2.1. Master Node

A single master node exists for each network and it is fixed.

The master node's clock is the main clock of the network and is used as a reference for synchronizing the clock of all the other nodes. For this reason, its clock is distributed over the network during the **timesync downlink** phase. This way all the nodes know the **network time**, represented by the master's clock.

The master node is also in charge of performing other main tasks, such as:

- Receiving from the dynamic nodes the global **network topology** during the **uplink** phase.
- During the uplink phase it is also in charge of handling requests of opening or closing *streams*. Streams are the abstraction exposed to the application by TDMH and will be discussed more in detail in Sec. 3.5.3.
- Thanks to the knowledge of the network graph, the master node is able to compute the routing for each stream over the graph and to compute schedules of the needed

transmissions and receptions. During the **schedule distribution downlink** phase, the master distributes the computed **schedule** is sent through to all the network nodes.

The master node can also open or close streams towards other nodes and send, receive or simply forward application data through the network.

3.2.2. Dynamic Nodes

One of the basic tasks performed by dynamic nodes is to synchronize their own clock with the master's one.
When new schedules are available, they also receive Dynamic nodes in the same network cooperate in order to collect the network graph topology information and to forward them to the master, during the **uplink** phase.

Dynamic nodes also receive from the master newly computed schedules and apply them to remain aligned with the new time slots allocation scheme.

As the master, dynamic nodes can also open or close streams. They also can send and receive application data, but, as the master, they are also in charge of forwarding information if they are on a stream path.

3.3. Time Division

Being TDMH a TDMA protocol, the network time known by all the nodes in the network is used to mark the time passing, and so also used to divide the protocol into different phases, occurring at different time slots. The repetition of all the phases is periodic and known to all the nodes, through TDMH configurations.

Different time units are defined by TDMH. In particular the protocol is organized in **tiles**, whose default duration is equal to 100 ms. Each tile then is divided into a configurable amount of slots.
Tiles can be either a **downlink tile** or an **uplink tile** and tiles sequence periodically repeats. This means that each tile starts with a **control slot**, being it a downlink or an uplink slot. Here is where the protocol phases execution takes place. The rest of the tile is occupied by data slots, in which application data packets are transmitted.
All tiles have the same duration, but downlink slots need more time in order to flood information from the master node to all the dynamic nodes. As a consequence, a lower

number of data slots is available in a downlink tile with respect to an uplink tile.
Each downlink slot can either be used for a timesync or for a schedule distribution.
More details on the protocol phases are described in Sec. 3.4.

Tiles are then grouped into **control superframes**. Superframes are the shortest repeating sequence of downlink and uplink tiles, so a control superframe corresponds to the period of phases execution.
The structure of control superframes can be statically specified inside the network configurations, but each superframe must contain at least one downlink and one uplink tile, in order for the network stack to perform both activities.

Tiles are also grouped into **data superframes**, which corresponds to the length of the schedules computed by the master node, which depend on the **period** of all the active streams. This means that data superframes length is dynamic and can change according to the evolution of the network structure.

Fig. 3.1 shows an example of phases executions, underlying the concepts of tiles, slots, and superframes.

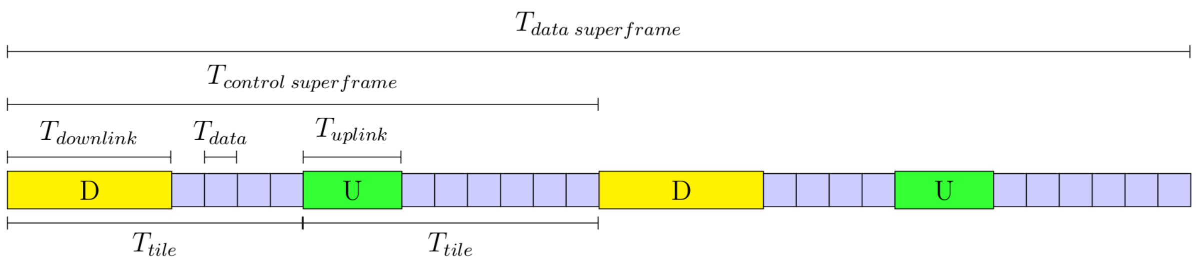

Figure 3.1: TDMH temporal organization. Transmission is divided into tiles. Tiles initial slots are occupied by control slots, either downlink (D) or uplink (U).

3.4. Protocol Phases

This section describes the main phases of the TDMH protocol, the **downlink phase** (divided into **timesync** and **schedule distribution**), the **uplink phase** and the **data phase**.

3.4.1. Downlink Phase

The downlink phase is used by the master node to distribute information to the other network nodes.
This includes both time synchronization and schedules information.

During the downlink phase, packets are distributed through the network using a **flooding** technique, which consists of a modified version of *Glossy* [8]. This technique also employs **constructive interference**, which can take place when radio packets having the same content are simultaneously transmitted by all the nodes at the same hop distance.

Timesync Downlink

The timesync is the downlink phase part that is dedicated to time synchronization.

During this phase, a node that needs to synchronize its clock remains in a continuously receiving state. After receiving the first packet, if a second one is received after the expected time interval, the node is considered synchronized and will start operating according to the TDMA mechanism.

Synchronized nodes periodically, during timesync pahses (i.e. every 100 tiles, which by default corresponds to 10 seconds), listen to synchronization packets to adjust their own clock. All the nodes forward the received packets, by also incrementing a counter that will signal to the other nodes their distance from the master node (in number of hops). A node's hop number is set when the node itself (re-)synchronizes and cannot change, until a desynchronization and a consequent re-synch take place. Forwarding of time synchronization packets is performed by all the nodes, even those that are currently non-synchronized.

Symmetrically, if a node loses three consecutive timesync packets, it also loses synchronization and a re-synchronization is needed.

Schedule Distribution Downlink

The second downlink subphase is represented by the schedule distribution.

During this phase, if a new schedule has been computed, the master distributes it. Otherwise, the latest schedule can be re-transmitted to all the nodes, for example after an explicit request of a dynamic node. It happens when schedule packets are lost and a node does not receive a complete schedule.

Notice that schedules can become big objects when the network is composed of tens of devices and the number of opened streams is relevant. A single downlink slot may not be enough for schedule distribution. Then, schedules are divided into smaller packets and distributed over multiple downlink slots. In order to increase the probability that all nodes correctly receive a complete schedule, schedule packets are transmitted multiple times, redundantly.

As for timesync packets, also packets sent during this phase are transmitted using the flooding technique.

Symmetrically to the uplink phase, the master also distributes **Info Elemenets**, which contain information related to the management of streams.

3.4.2. Uplink Phase

Symmetrically to the downlink phase, the uplink phase is used by the master node to collect information from the dynamic nodes.

All the nodes maintain a **neighbors table** that during this phase is converted to a bitmask and transmitted. It is guaranteed that reciprocity holds, meaning that if node a forwards a table in which node b is set as a neighbor, that node b will also forward a table in which node a is specified to be a neighbor too. This guarantee is given by cooperation among dynamic nodes and not by the master.

During the uplink slots, only one node transmits packets, while all the others try to receive them. So only one node can transmit during an uplink tile. Indeed, the uplink phase works in a *round-robin* fashion. The round-robin mechanism can take place without the master node intervention, relying on the distributed knowledge of the network time and of the maximum number of network nodes, that is a TDMH configuration parameter (it can be specified in the *NetworkConfiguration* object).

Thanks to this mechanism, when a node is receiving during an uplink slot, it can also know which node is currently transmitting. If a packet is received with a strong radio signal, then the transmitting node can be added as a neighbor by the receiver.
The master can use the received neighbors bitmasks to update the network topology.

During the uplink phase, another important piece of information is sent to the master node. **Stream Management Elements** (**SMEs**) are the stream control packets, that are used by nodes to open or close streams.

3.4.3. Data Phase

Finally, the data phase is used by both the master and dynamic nodes to exchange application data. The data phase traverses the TDMA schedule that was computed by the master node and distributed during the downlink phase reproducing it. Indeed, time slots reserved for application data are assigned to one or multiple streams in different nodes. The data phase is executed on each node and is in charge of managing packets that have to be transmitted or received by the node itself. Moreover, it also handles packets that only have to be forwarded to another node. If no action has to be performed in a specific slot, the node can sleep and save power.

In order to meet its real-time goals, TDMH does not guarantee the delivery of each packet. The possible loss is reliability is then faced by the use of **redundancy**. For each stream, a redundancy level can be specified, from one to three, meaning that each packet on that stream is re-transmitted a number of times equal to the specified value. Redundancy can be only temporal, with each packet being only sent multiple times, or also spatial. In the second scenario, packets are also sent over multiple redundant paths. Redundancy is also managed during the data phase.
Experiments showed that, even in an electromagnetically polluted environment, streams redundancy allows keeping the network reliability largely above 99%.
More details are described in Sec. 3.5.3.

3.5. TDMH Network Stack

TDMH can be considered to be a full **network stack** [11], since only a physical layer is needed in order for its functionalities to be properly fulfilled. All these functionalities are in fact implemented in a single protocol.
Fig. 3.2 represents the complete TDMH network stack and its layers.

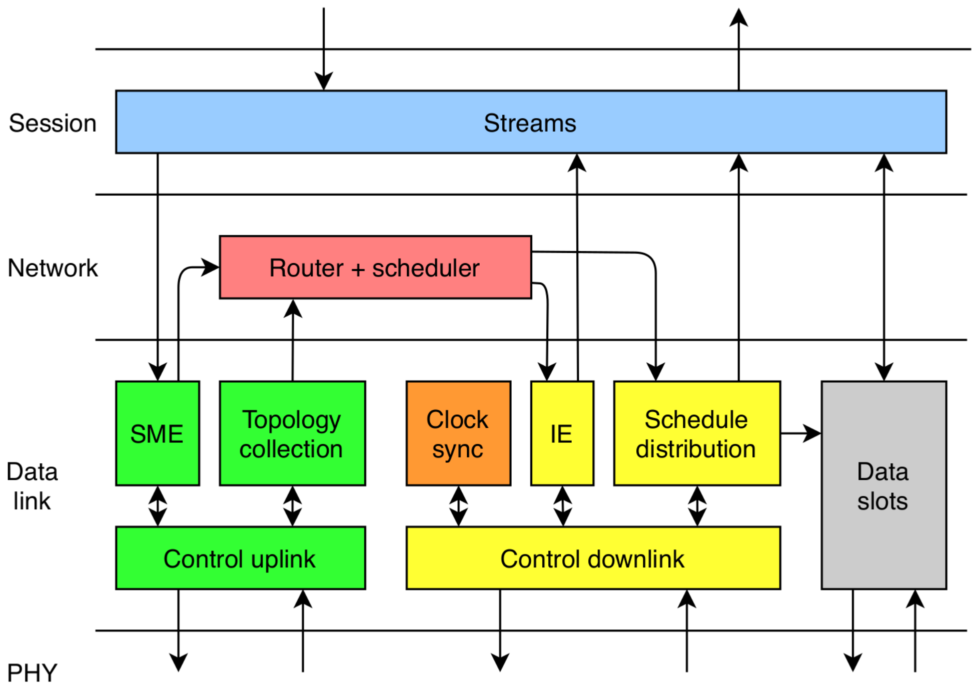

Figure 3.2: TDMH network stack overview.

3.5.1. Network Layer

In the **network layer** the router and the scheduler are executed by the master node, together with the schedule distribution and activation logic.
This section briefly explains how these mechanisms work.

Routing and Scheduling

Both the streams routing and scheduling processes are centralized in the master node.

The **router** is the component that, given the network topology, is in charge of finding possible paths between nodes.

The **scheduler** instead, given the network topology and the list of open streams, assigns transmissions to TDMH data slots. Notice that each transmission can happen in a single slot and belongs to a single stream. In fact, each one has a sender and receiver node (neighbor nodes), which do not necessarily coincide with the source and the destination of the stream, since packets can also be forwarded by intermediate nodes.
It is important to underline that new schedules are computed, and then distributed, only when the streams list changes (e.g. some new stream is opened or some is closed) or

topology changes affect existing streams, such that for example a re-routing is needed. After distribution, the schedule can be repeatedly executed until a new one is flooded through the network.

Channels Spatial Reuse

As said, each scheduled transmission takes place in a single slot, but in the same slot multiple transmission may be scheduled. This optimization is called **channel spatial reuse**.

Thanks to its centralized nature, when computing a schedule, the scheduler decides which node is allowed to transmit during a given data slot. It is able to perform checks on channels interference, based on the network topology graph and, consequently, it is possible to schedule two or more nodes to simultaneously transmit or receive radio packets in case the different transmissions do not interfere with each other.

Fig. 3.3 and Tab. 3.1 show an example of schedule for node 3, with an opened stream to the master node 0 and triple redundancy. In the example channel spatial reuse is exploited too.

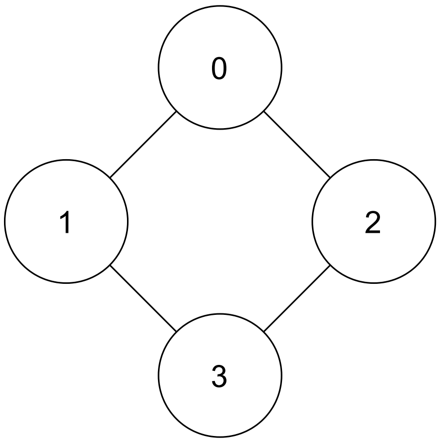

Figure 3.3: Example of network topology with four nodes. The master is node 0.

Redundant transmissions	...	Slot 4	Slot 5	Slot 6	Slot 7	Slot 8	Slot 9	...
1^{st}	..	$3 \rightarrow 1$			$1 \rightarrow 0$			...
2^{nd}	...		$3 \rightarrow 1$			$1 \rightarrow 0$		...
3^{rd}	...				$3 \rightarrow 2$		$2 \rightarrow 0$	...

Table 3.1: Example of scheduling for stream $3 \rightarrow 0$.

Schedule Distribution and Expansion

When a schedule has to be flooded through the network, during downlink slots, it is forwarded from node to node in its *implicit* (compact) form. Its elements are composed of a tuple containing:

- Stream ID
- Source node
- Destination node
- Offset in schedule (number of slots)
- Period

After the schedule is received by a node, it performs the **schedule expansion**. Each schedule element in which either the source or the destination coincides with the node identifier itself is expanded: transformed into an element that associates the stream with its offset in the schedule, its period and a specific action.
Forwarding buffers are also allocated during the schedule expansion process.
The resulting explicit schedule is an array whose size is equal to the number of slots in a complete schedule. *ExplicitScheduleElements* are tuples containing the stream ID, the offset inside the schedule and the associated action, in addition to the stream parameters (containing information such as the period and the redundancy level).

Possible actions associated to time slots are:

- *Send from stream* (`SENDSTREAM`): the node is the source of the stream and it is schedule for transmission.
- *Receive from stream* (`RECVSTREAM`): the node is the destination of the stream and it is schedule for reception.
- *Send from buffer* (`SENDBUFFER`) and *Receive from buffer* (`RECVBUFFER`): the node is neither the source nor the destination of the stream, but it is onto the stream path and it is scheduled for forwarding transmission or reception respectively.
- *Sleep* (`SLEEP`): the node is not scheduled in that specific slot. No action is needed.

As previously said, schedules are transmitted multiple times (triple redundancy) in order to minimize the probability that a node receives only a partial schedule. If this happens though, a re-transmission can be requested to the master node.

Schedule Activation

When the scheduler produces a new schedule, this is associated with a data structure containing information about the schedule itself, such as its ID and the number of tiles in it.
An important piece of information that is also distributed to all the nodes is the schedule **activation tile**. It represents the tile index at which the new schedule will become active. At the **activation tile**, the previous schedule is replaced by the new one and its playback starts in the data phase.

Schedule Length

A schedule is composed of various tiles. The schedule length can be considered as the number of tiles in it.

When multiple streams exist, the schedule length is the *least common multipl*e among all the streams period.

3.5.2. Data Link Layer

The **data link** layer is in charge of managing the previously explained protocol phases, from the time organization to the downlink and uplink phases. It is then in charge of managing everything that concerns network topology collection, clock synchronization, schedule distribution and stream management elements.

In this same layer data slots are managed, so transmission, reception or forward actions are performed according to the node's schedule knowledge.
The data link layer (data phase) main task in fact is to continuously reproduce the last valid schedule. In particular, the schedule is used in its explicit form, which is specific for every single node, since each node may have to take a different action in the same slot. This means that it executes the transmissions and receptions of application data through the scheduled streams, by reproducing the actions that are associated with each TDMH time slot.

During slots associated with transmission actions `SENDSTREAM` and `SENDBUFFER`, a sleep action until the start time of next slot is performed if no packet is available for transmission.

3.5.3. Session Layer

The **session layer** implements the streams management logic. In this thesis, the term session layer will be used when referring in general to the set of streams and their management logic.

Streams

The TDMH session layer provides applications the **streams** abstraction to establish communication among any two network nodes. They are similar to TCP sockets, in order to provide the user with a familiar interface. The difference is that, due to the real-time goals of TDMH, they also embed the notion of **transmission period**. In fact, TDMH streams only allow a single packet to be transmitted during a period.

Streams are identified by the tuple:

- Source node
- Destination node
- Source port
- Destination port

Period As previously introduced, TDMH enforces the notion of stream period, which is the time interleaving between two consecutive stream transmissions.
Fig. 3.4 shows a scenario with two streams, that have different periods. As it can be seen from the example, periods are measured in number of tiles.

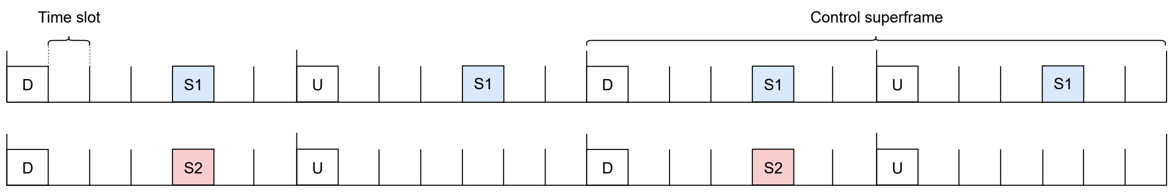

Figure 3.4: Multiple streams periods example. Stream *S1* has period 1 tile and *S2* period 2 tiles.

The period can be chosen according to the application requirements, since it has a direct impact on the frequency at which data are produced or received.
Assume to have a sensor node and a control algorithm running on another node: the

sensor node may specify a lower period for transmitting data (higher frequency), while the controller may output control actions at a lower frequency, and so transmitting them with a higher period, e.g. 1 second.

TDMH exposes a set of available stream periods, in the form of PK, where K indicates the number of period tiles.
Supported periods are the numbers 1, 2 and 5 multiplied by different powers of 10. For example, available periods can be $P1$, $P2$, $P5$, but also $P10$, $P100$, $P500$ and $P2000$. The minimum, in the current implementation, is $P1$ and the maximum is $P10000$.

The reason why only period values starting with 1, 2 or 5 are allowed, comes from the *Frobenius coin problem* [3]. These numbers allow finding a higher number by keeping their least common multiple as low as possible. As a consequence, schedules length is also kept low.

Redundancy As briefly introduced, TDMH does not guarantee the reception of each packet by the destination node. No packets queue is used and packets are sent without acknowledgments or re-transmissions, as it happens instead when using the TCP protocol, which guarantees that each packet that is transmitted is also eventually received by the recipient. Error detection mechanisms implemented in TCP are not applicable in the context of TDMH: a variable number of re-transmission would cause the end-to-end latency to be non-predictable, which is not compatible with a real-time network. As consequence, when using TDMH radio interference can lead to packets loss.
Since reliability in wireless networks is a fundamental aspect, some mitigation has to be implemented.

TDMH mitigates this issue by allowing the application to specify a redundancy level associated with each stream. Packets sent over that stream are transmitted a number of times equal to the specified level, which is a value between one and three. This allows the receiver to have multiple chances to get the incoming data, increasing stream's reliability. The redundancy level allows for a trade-off between bandwidth and reliability. Indeed, if all the streams use the minimum redundancy level (which actually is no redundancy) more streams can be scheduled, while if all the streams use the maximum level the reliability is increased but fewer data can flow through the network during the same time interval.

TDMH implements two types of redundancy:

- *Temporal Redundancy*: each packet transmission is simply duplicated and uses the same path between source and destination nodes.

- *Spatial Redundancy*: besides being duplicated, each packet is redundantly sent over multiple paths between source and destination nodes. Two paths are considered to be different if the intermediate nodes are different (excluding source and destination of course), in order to avoid single points of failure. If multiple paths do not exist in the network topology, the redundancy is downgraded: it may happen that the number of distinct paths is lower than the redundancy level or, in the worst case, the spatial redundancy is downgraded to a temporal one (if a single path exists).

The redundant transmissions and receptions are handled by the data phase. When some data is sent through a stream, the data phase manages the multiple transmission. When some data has to be received, only after the redundant slots have passed, the data phase returns the received data to the application, if something was actually received. This avoids delegating the same task to the application, which otherwise would have to handle multiple copies of the same data. Moreover, this strategy helps to keep the packets latency consistent, while returning the data to the application always after the first successful reception would make it fluctuate.

Fig. 3.5 represents two schedules in which two streams exist. Redundancy of *S1* is set to *triple* while *S2* has no redundancy.

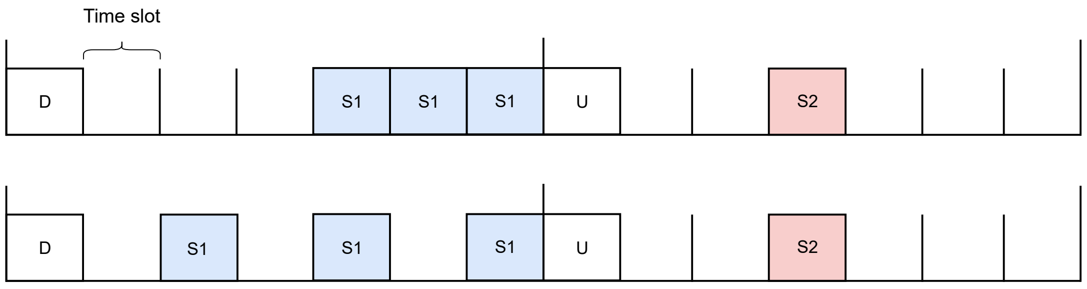

Figure 3.5: Different streams redundancy levels example. It shows both a scenario in which the redundant transmissions are consecutively scheduled and one in which they are not, for stream *S1*.

Clients and Servers Streams can be divided into two categories, called servers and clients.

A node that aims at opening a communication has to perform a *connect* operation to a specific node and a specific port. The node that receives the connection request on its side needs to have a server opened on that port.

Server streams are only able to listen to a specific port and accept incoming connections. Server streams cannot send or receive application data packets. When an incoming connection request is accepted, a client stream is returned. This is the stream that the node can use to carry on communication with other nodes.

Direction Another characteristic of streams is their direction. Streams direction can assume two possible values:

- `TX`: only the client is allowed to transmit data, while the server can only receive.
- `RX`: only the server is allowed to transmit data, while the client can only receive.

Primitives Streams expose primitives both to the application and to the data phase.

Primitives exposed to the application to perform transmissions and receptions are:

- *connect*: it is used by the application to open the communication with another node. On the other side, a server is needed to be running on the node and port specified as a parameter of the *connect* primitive.
- *write*: copies data to the stream's buffer. If the buffer is already occupied (e.g. multiple *write* calls during the same period), the execution blocks on a condition variable until the buffer is emptied by the data phase.
- *read*: symmetrically to the *write*, retrieves data from the stream's buffer. If the buffer is empty, the execution blocks on a condition variable until a new packet is received and added to the buffer.

The main primitives exposed by streams to the data phase instead are:

- *sendPacket*: used by the data phase to get the packet that has to be sent. The packet is retrieved from the stream's buffer (if it exists) and returned. If it is the first redundant call, it unlocks the *write* from waiting on the condition variable.
- *receivePacket*: used by the data phase to signal that a new packet is received. The new packet is passed as a parameter and copied to the stream's buffer. When the redundancy counter reaches a value equal to the stream redundancy level, it unlocks the *read* from waiting on the condition variable.
- *missPacket*: used by the data phase to signal that a packet was missed. The redundancy counter is incremented, since missed packets concur to the stream redundancy too. When the redundancy counter reaches the value of the stream redundancy, it unlocks the *read* from waiting on the condition variable.

As mentioned, servers expose other two methods:

- *listen*: a server is used to listen to a specific port and to wait for incoming communication requests.
- *accept*: when a communication request is received, the *accept* method is used to create a stream that can be used to communicate with the requesting node.

Stream Manager

The *StreamManager* is the component that is in charge of managing servers and streams instantiated by network nodes. Every node has a *StreamManager* since it is needed to hold all the streams information and to open or close streams according to information received by the network and the master node. Streams and servers status can change according to actions performed by the application and the stream API or events coming from the network, such as schedules, disconnections or Info Elements.

Streams primitive are exposed to the application and the data phase through the Sensor-Manager and it is implemented as a finite state machine.

3.5.4. Physical Layer

The TDMH **physical layer** implements a set of features that are worth mentioning:

- Packets transmissions can be scheduled for a future time instant. Not only the data to be sent has to be specified. According to the IEEE 802.15.4 standard, the transmission **jitter** has to be lower than 500 ns while TDMH physical layer achieves a value of 21 ns. This jitter in fact is needed to be as low as possible in TDMH, in order to be accurate enough for constructive interference to take place.
- Since packets are transmitted without acknowledgments or re-transmission logic, the physical layer only has to handle one packet at a time and no buffers are needed.
- Thanks to the nature of TDMH, the physical layer does not have to continuously listen to the radio channel for incoming packets, but it is already aware of the TDMA slots in which a reception is scheduled. This has of course benefits on the power consumption.

3.6. Cryptography

TDMH also supports **authentication** and **encryption** of all control and data messages [15]. It was designed to use symmetric keys safely stored in the nodes and to also be resistant to replay attacks.

All the dynamic nodes that join the network are required to authenticate the master node by means of a *challenge-response* mechanism.

The cryptographic scheme is centered around the problem of key management: a *rekeying* based on a hash chain mechanism allows to achieve periodic key rotation. The rekeying process takes place when a new schedule is distributed. During this process, a node must derive:

- A new master key.
- Three phases keys (for uplink, timesync and schedule distribution), used for authentication and encryption of packets relative to protocol phases.
- A new key for all the streams in which the node is involved as an endpoint (either as a source or destination node). Key of stream S is used for authentication and encryption of packets belonging to stream S.

Since the number of streams that need rekeying can be high, the rekeying process is implemented in such a way that it is possible to incrementally execute it across multiple downlink slots. All the keys are then replaced by the new ones when the new schedule is applied. Using TDMH control slots to perform such a process, does not affect the periodicity of data transmission.
Even if no new schedules are computed for a long time, the rekeying process is performed anyway after a timeout elapses.

4 | Problem Statement

4.1. Real-Time Systems and Latency

According to the *IEEE Technical Committee on Real-Time Systems*, a **real-time system** is "a computing system whose correct behavior depends not only on the value of the computation but also on the time at which outputs are produced" [10].
Real-time is sometimes confused with the notion of *high-performance computing*, which, as the name suggests, refers to the ability of a system to provide outstanding computational resources and performance. An example of real-time computation is instead an automotive anti-lock braking system (ABS). The ABS execution has to be carried out before a specific deadline elapses. The system has to guarantee that the task's deadline is not missed, as opposed to the high-performance computing requirement of completing its tasks as soon as possible. This means that no further performance optimizations are needed in a real-time scenario, as far as tasks deadline are respected.
Consider another example in which a web page is subject to high network traffic. The web page response time may get higher or lower together with the load amount, but it will complete eventually. Even in the worst-case in which the user request takes more than a timeout to be served, no catastrophic consequences take place. On the contrary, this variable delay is not admissible in a real-time system nor compatible with it.
Focusing on real-time systems, two different kinds exist: in *hard real-time* systems missing deadlines means producing catastrophic consequences, while in *soft real-time* systems a deterioration of performance is experienced when requirements are not met.
To summarize, the common thread among all the real-time systems is that processing fails if not completed within a specified deadline, which is relative to an event. Moreover, deadlines must always be met, regardless of system load or other external interferences.

One of the main elements related to real-time tasks and the requirement of fulfilling their deadline is **latency**, which in general is a measure of a delay. In a network, latency measures the time it takes for some data to get from its source to its destination across the network. Low latency is desirable in a wide range of use cases, but for real-time and possibly periodic tasks, but a consistent and bounded latency is usually more important. Imagine an industrial control loop, with a sensor node producing data and a control application, running on another node, that has to compute control output based on the received sensor data. If the transmission has a consistent amount of latency, the application can plan the processing accordingly. If sensor data is fed with an inconsistent amount of latency, whether large or small, there will sometimes be too much data, and sometimes not enough. If the application runs out of data, there will be no up-to-date output for some time or it will be delayed. On the contrary, some data may not be used if the application outputs are periodic or if it has not enough processing power to handle data coming in a shorter time interval.

4.2. TDMH Streams API

TDMH exposes a simple API for handling streams, providing a set of fundamental operations that can be performed with them.

Apart from the primitives needed to establish or conclude a communication among two devices, such as the methods *listen*, *accept*, *connect* and *close*, network nodes can exchange data through the *write* and *read* primitives. These two methods allow the application to send or receive data packets through a very common API, similar to the one used to read and write from or to a file, that completely masks the lower layers of the network stack.

Alg. 4.1 and Alg. 4.2 shows the basic behavior respectively of a node that opens a stream and sends application data through it and of a node that accepts an incoming stream opening request, receiving data.

Algorithm 4.1 Basic application behavior, sender side

```
1:  stream ← connect(dest, port, ...)
2:  if stream < 0 then
3:      print("Error")
4:  else
5:      while stream.status == ESTABLISHED do
6:          data ← ...
7:          write(stream, data)
8:      end while
9:      close(stream)
10: end if
```

Algorithm 4.2 Basic application behavior, receiver side

```
1: server ← listen(port, ...)
2: if server < 0 then
3:     print("Error")
4: else
5:     stream ← accept(server)
6:     while stream.status == ESTABLISHED do
7:         read(stream, data)
8:         ...
9:     end while
10: end if
```

4.3. Weak Points

Despite the usage simplicity of the current API, it doesn't provide the required guarantees on the end-to-end latency of a any packet sent through the network. In particular, it leaves to the application the task of synchronizing the data packets generation and transmission request with the actual transmission time.

Being TDMH a TDMA network protocol, each time slot is dedicated to the transmission over a specific network hop (or a set of hops, if channel reuse is possible). Indeed, the data phase running on a node always sends or receives packets during the slots assigned to that specific node. So, the data phase is always synchronized with the network schedule. What the *write* primitive does is to copy the data generated by the application code (upper layer) to the session layer, which holds it until the data phase is ready to get it and to send it over the network, through the underlying layers. The session layer buffer can hold two packets for each one of the existing streams. Each *Stream* object that is part of this network stack layer, can hold one packet to be transmitted and one packet to be received at the same time.

From the application point of view, the call to *write* corresponds to the transmission request, but, since the application code is free to call this primitive at any time instant, the delay between the time at which the application requests the transmission and the actual time at which it is sent through the physical layer varies according to the application logic. If *write* is called exactly after the data phase executes and checks if some new data exists in the session layer buffer, this delay is maximum. In fact, the data phase will find the stream buffer to be empty and the packet will be read and sent by the data phase thread

during the next stream's period (when it will find the packet in the buffer). Assuming that latency is computed from the moment in which the packet is created by the sending node, to the moment in which it becomes available by the destination node application, the described delay directly affects the end-to-end latency value.
Moreover, if the application unconditionally generates new packets one after the other, it may happen that, after the first *write* is performed, the session layer buffer remains occupied while a successive call to the *write* primitive takes place. Since streams periodicity is always guaranteed and respected, the subsequent transmissions will be delayed in the future (i.e. to the following stream's period), increasing the latency of successive packets.

The explained behavior is shown in Fig. 4.1. Assume to have a single stream used for transmitting some data over the network, with period P. When the first write w_1 is performed, the session layer does not hold any data yet, so the first packet p_1 is immediately copied to the session layer. At this point, the application might immediately produce another data packet, before p_1 is actually sent (which takes place at time t_1, in the first available slot). Since streams periods have to be guaranteed, the second write w_2 blocks until the buffer is emptied and p_1 sent, which means until the following slot is assigned to the sending stream. It is immediate to understand that in this scenario, packet p_2 will have a higher end-to-end latency than p_1, since it is forced to wait some time until the buffer is first cleared.
From the third write operation w_3 on, the latency stabilizes: since w_2 blocks until the completion of w_1, from this point write operations will be aligned with the transmission slots of the mentioned stream, but the latency is as high as two times the stream period.

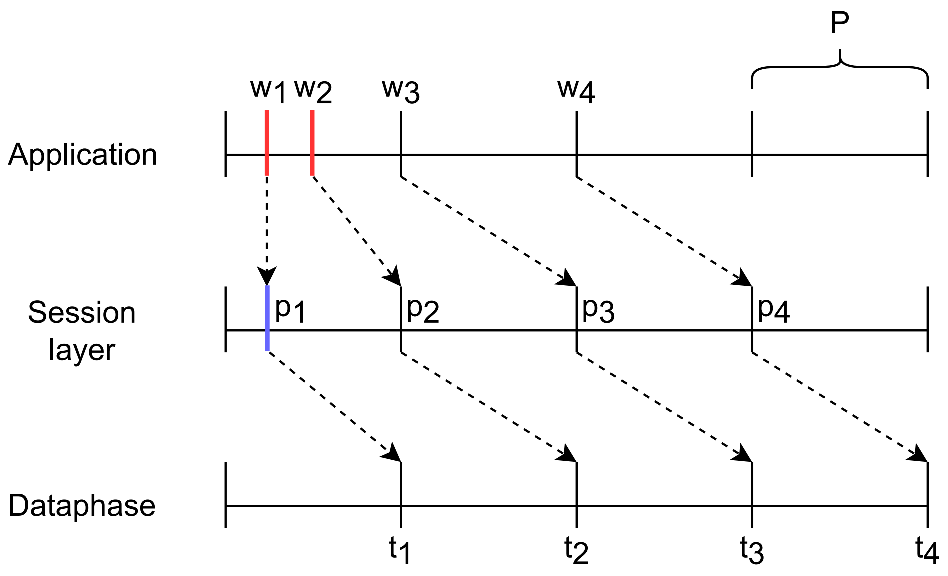

Figure 4.1: Undesired behavior in which write operations w_i are delayed by two periods. p_i represent ready for transmission packets, hold by the session layer buffer, and t_i the transmission slots, with period P.

In the shown scenario, the latency value directly depends on the stream period and, in particular, its value is a multiple of the period itself. Two streams sending the same amount of data through the same network path may happen to have latencies that can differ by few orders of magnitude, for example if they transmit with very different periods (e.g. 1 tile and 100 tiles period).

The above example shows that the problem affecting the latency value is concentrated on the sender side, since it is the transmission that is delayed if the application produces data too fast. The reception instead is managed by the data phase, which only executes according to the scheduled time slots and returns the data to the application after it is (redundantly) received.

4.4. Real Example

This section shows a simple example underlining the entity of the latency on a star network, composed of three nodes: two dynamic nodes (with IDs 1 and 2) sending packets over a stream opened towards the master node.
Streams were opened using a period equal to one tile and the tile duration was 100 ms. So, ten packets were sent every second. When a packet is sent, a timestamp is added to it, so that the master can log the delay from the moment in which the packet was created

and the one in which it is received. The experiment was left running until ten thousand data packets were sent by both nodes.

Fig. 4.2 shows the measured latency of packets that are sent through stream $1 \rightarrow 0$. It can be seen that the average latency has a value of around 220 ms, which corresponds to two times the stream period.

The first sent packet, instead, has a latency of only 60ms, due to the fact that the session layer buffer is empty when the application calls *write* for the first time. This leads the measured jitter, with respect to the latency mean value, to be as high as 162 ms. Even considering only packets after the latency stabilizes, the jitter is equal to 37 ms, which represents a variation of almost the 17% with respect to the average latency value. A detailed view on the initial packets is shown in Fig. 4.3. It can be seen that even in correspondence of the plateau, where the measured latency is more stable, the amplitude of the oscillations has a value of 4 ms. Such a variation is not tolerable since it has the same order of magnitude as the TDMH slots dimension.

The mentioned problems affect the overall latency standard deviation, which is in the order of a few milliseconds.

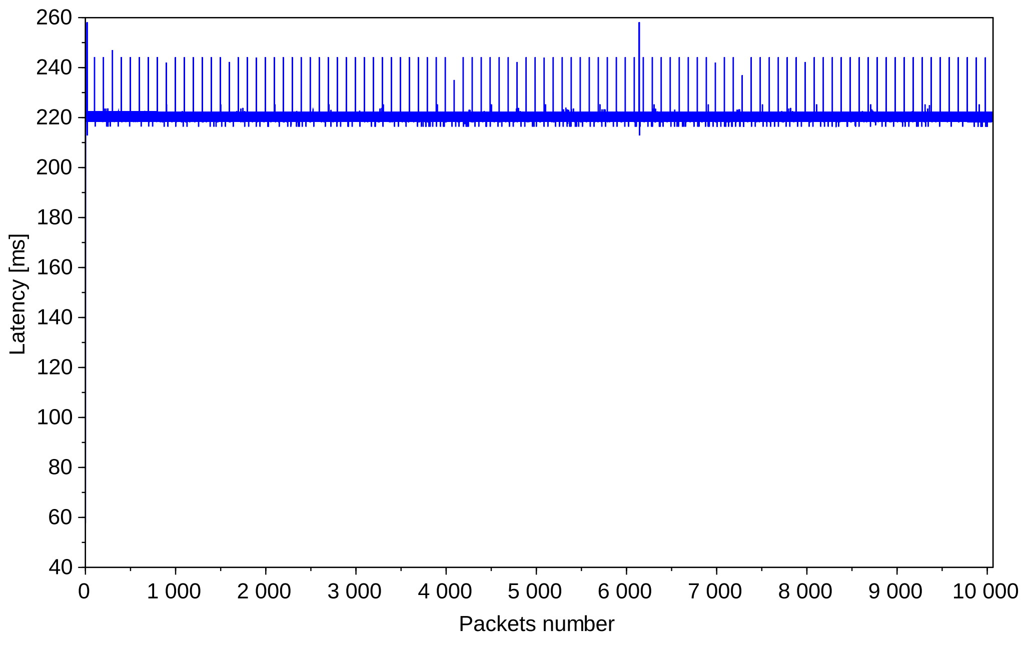

Figure 4.2: Measured latency of packets sent over stream $1 \rightarrow 0$.

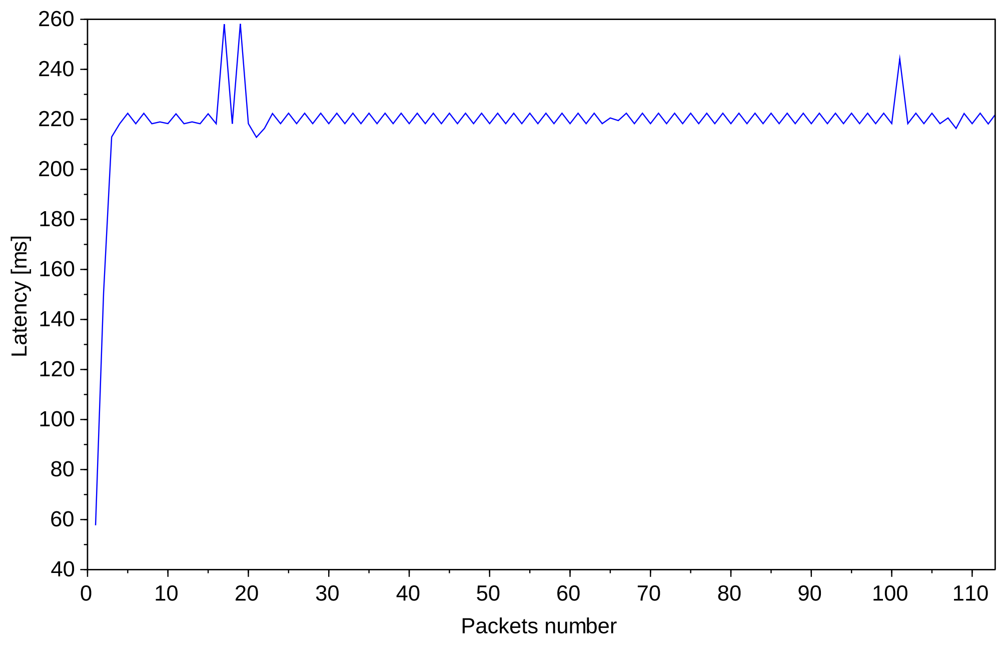

Figure 4.3: Detail of the measured latency of the first 110 packets sent over stream 1 → 0.

Stream	Stream period	Packets num.	Avgerage latency	Jitter	Standard deviation
1 → 0	100 ms	10050	220.476 ms	162.761 ms	3.700 ms

Table 4.1: Latency statistics for stream 1 → 0 when using the current session layer implementation.

4.5. Desired Behavior

As mentioned, in the current implementation the end-to-end latency of data packets is influenced by the application logic. This behavior is of course undesired and latency must be kept constant even if the user changes the application code and its logic or actions.

Furthermore, the desired latency shall not depend on the stream period, which should only represent the maximum admissible latency. The latency measure, instead, should only depend on the current schedule and on the specified redundancy value for that stream, which directly affects the number of slots needed to deliver a packet. Even if two streams have very different periods, for example 1 and 100 tiles, the end-to-end latency of the data packets sent over them should be the same if they span the same amount of schedule slots, since their delivery will take the same amount of time slots as well. This means that each stream's **latency** has to be, not only **measurable** but also **predictable** by knowing the **current schedule**, which implicitly means also knowing streams redundancy level. No information about streams periods is instead needed.

It is required that the data packets generation and the *write* operations are performed slightly before the start of each transmission slot, so that the new packet will always find the session layer buffer empty and can be transmitted as soon as possible (i.e. in the next slot assigned to the stream on which the packet has to be sent). The time passing between the moment in which the application generates a packet and its transmission should also be known.
This behavior is pictured in Fig. 4.4.

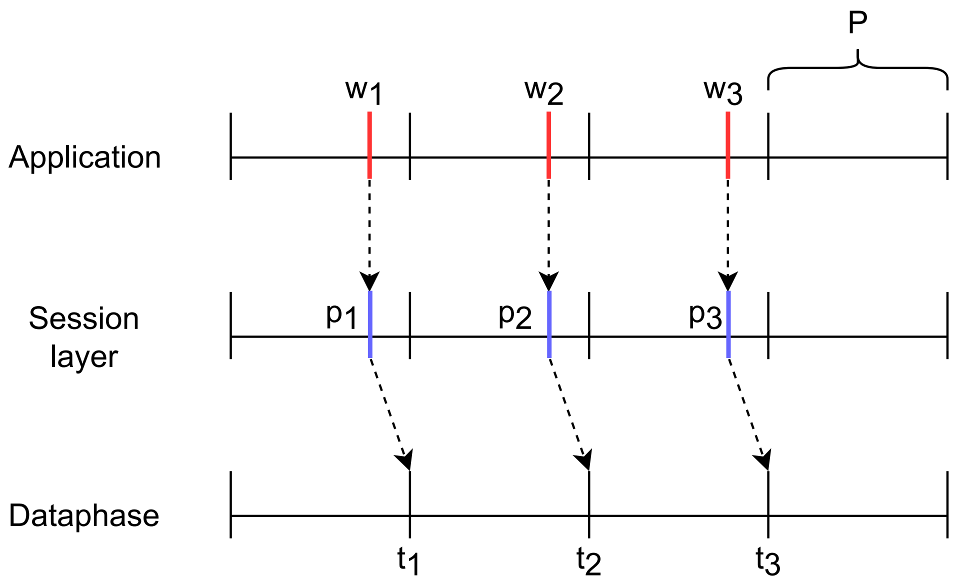

Figure 4.4: Desired application and session layer behavior.

It is needed to enforce guarantees on the end-to-end latency inside the network stack lower layers, by also providing some primitives through which the application code can be able to synchronize itself with the data phase and, so, with the streams schedule without any explicit knowledge about it. Indirectly, then, the aim is to align packets generation and transmission requests with the actual transmission slots.

In the following, the proposed solutions are presented, together with an overview of how latency bounds can be computed in different scenarios.

5 | Proposed Solution

As briefly introduced in the previous chapter, the proposed solution has to guarantee a bounded latency for all the packets flowing through the network. No matter what the stream period is, the latency boundaries should only depend on the required stream redundancy (none, double or triple) and on the current schedule.
It is also preferable that the proposed solution maintains the ease of use of the existing API, while leaving some flexibility to the application and to the user.

In this chapter, the two implemented solutions are discussed, together with the required updates to two of the main TDMH modules, the *data phase* and the *schedule expansion*.

5.1. Callbacks API

5.1.1. Overview

As the name suggests, the first proposed API is based upon a **callback functions** mechanism. After opening a stream, the application can specify two functions to be executed respectively before a packet is sent and after a packet is received. Indeed these two functions are the interface among the application and the underlying layers, preparing the data to be transmitted, by copying it to the session layer, and handling the reception of new packets.

The idea behind the callbacks API implementation is to keep it simple, avoiding introducing new threads to the system.

5.1.2. Implementation

When creating a new stream, the application can specify the send and receive callback functions. Each callback then needs to be called by the underlying network stack during the required time slot assigned to the stream in question. This way, since the lower layers know the current schedule, the management of receptions and transmissions will always be synchronized with the execution of the callback and, as a consequence, with the application, being these callbacks the only interface among the upper and lower layers in the stack.

The prototype of the mentioned functions is:

```
void callback(void* data, unsigned int* size, StreamStatus status)
```

The first parameter is a pointer to the packet buffer, while the second one is a pointer to the number of bytes to be written or read to or from the buffer. The basic idea is that these callbacks are used by the application to copy data to or from the session layer. In fact, the aim of a send callback is to copy the data to be transmitted to the session layer buffer, while the receive callback has to read the data from it A send callback will set the value of **size* according to the number of bytes to be sent, while a receive callback will access **size* to retrieve the number of bytes to be read. The third parameter instead allows the application to have a feedback on the current status of the stream. Since streams can be closed during the execution, even without an explicit request from the application, for example if a radio link in the network becomes too weak, the stream status parameter is useful to avoid leaving the application in a waiting state without any feedback. In this sense, this parameter works as an error code.

Through the *setSendCallback* and *setRecvCallback* methods, callback functions can be set into the *Stream* class, in order to signal to the session layer that this API has to be used. This way, they can also be called inside the same class when *Stream*'s public methods *sendPacket(Packet& data)*, *receivePacket(const Packet& data)* and *missPacket()* are executed. These methods indeed are called by the data phase thread. Alg. 5.1 shows how to set a stream send callback, but the same holds for the receive one.

Algorithm 5.1 Basic application behavior to set a stream callback.

```
procedure CALLBACK(*data, *size, status)
if stream.status != ESTABLISHED then
    close(stream)
else
    *data ← ...
end if
end procedure

stream ← connect(dest, port, ...)
if stream < 0 then
    print("Error")
else
    setSendCallback(stream, &CALLBACK)
end if
```

In order to retrieve the data to be sent from the application, the send callback is executed when the data phase calls *sendPacket* for the first time on the current packet, that is before the first redundant transmission of the packet.

On the contrary, the receive callback is called after *receivePacket* (or *missPacket*) is called R times, where R is equal to the redundancy value of the mentioned stream: the received data indeed is really available to the application for processing only after the R redundant receptions (or misses) have completed.

See Fig. 5.1 and 5.2 for reference.

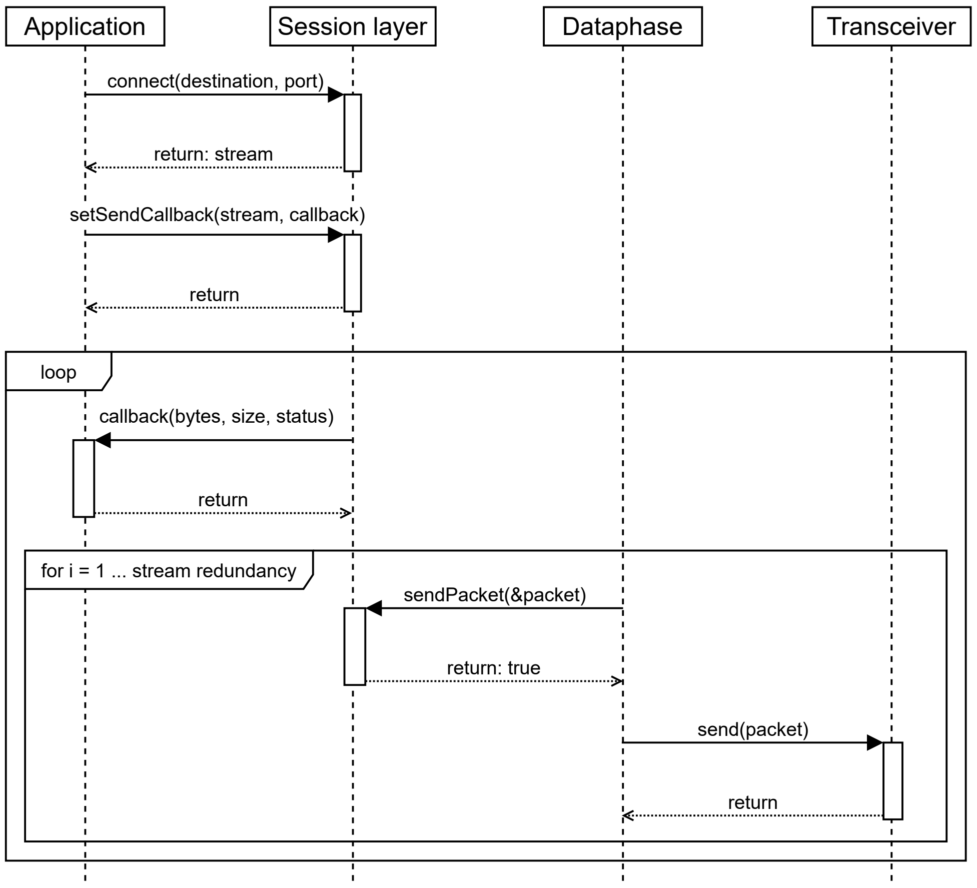

Figure 5.1: Sequence diagram showing the send callback mechanism.

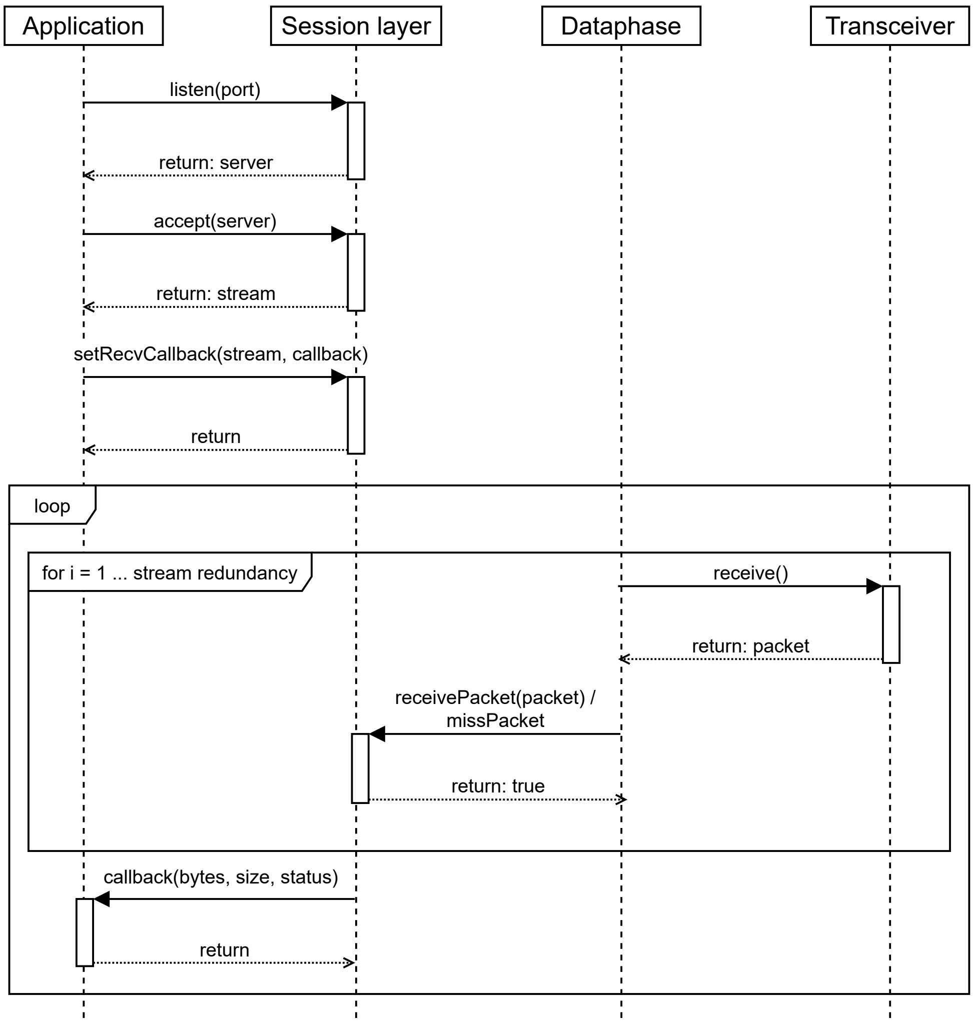

Figure 5.2: Sequence diagram showing the receive callback mechanism.

When computing the duration of each single time slot, the execution time of the callback functions has to be considered too, to ensure that they do not occupy some time needed to perform other operations, for which the data slot duration was sized.
For these reasons, the application can specify the callbacks execution time inside the *NetworkConfiguration* object, which has to be the same in each of the nodes connected to the TDMH network. At startup, the specified value is directly summed to the data phase duration, thus directly influencing the data slot dimension. It must also be the same value among all the nodes and each node using this API must ensure that the given callbacks execution time, for each stream, is an upper bound of the real time needed to execute both send and receive callback functions. In order to also handle the case in which a node may need to receive and then to send during two consecutive slots, and so the execution of the receive callback and the send callbacks may be overlapped, the double of

the callbacks execution time is actually added to the data slot duration.

In the current implementation, the data phase calls the stream methods *sendPacket* or *receivePacket* at each execution, which may mean at each slot assigned to the stream in question.
Fig. 5.3 shows the wrong behavior in which the callback is executed at the start time of the slot assigned to streams *S1* and *S2* (when *sendPacket* is called). In this case, the real transceiver transmission would be delayed after the callback execution, causing the packet to be sent too late.

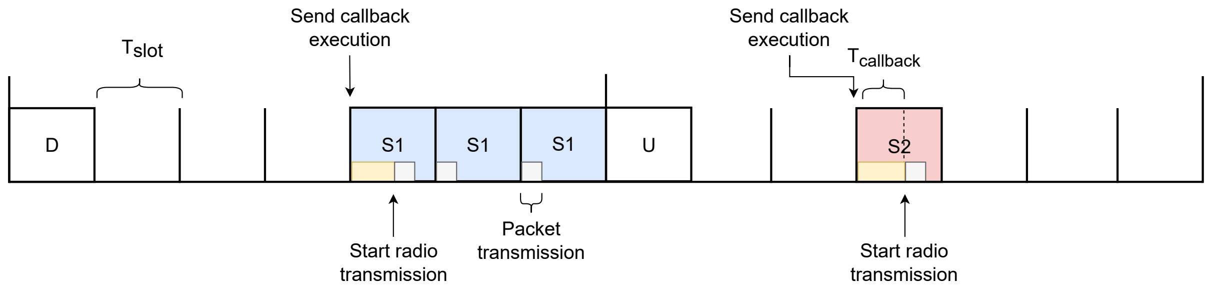

Figure 5.3: Undesired send callback execution behavior, which would delay the packet transmission.

When sending a packet, to avoid executing the send callback when the stream's transmission slot is already started, and so delaying the real radio transmission, the data phase execution (call to *sendPacket*) has to be anticipated to the slot start time minus the callback execution time.

The correct behavior, in which the callback is executed at the slot start time minus the needed time to actually execute the function, ensuring that the real radio transmission is aligned with the slot start time, is instead represented in Fig. 5.4 and 5.5 that respectively show the execution of a send and a receive callback.
It might seem that in the setting shown in Fig. 5.4 the callback execution still consumes some time allocated to the previous slot with respect to the slot assigned to streams *S1* and *S2*, but even if the previous slot is assigned to another stream, the callback execution time is already accounted and added the slot duration, so some dedicated time is allocated for the execution of the send callback. Reception in the previous slot will end for sure before the send callback execution is started. One could imagine that the time needed to execute the send callback is allocated in the previous slot with respect to the transmission one, while the time needed by the receive callback is allocated in the same reception slot. Anyway, during reception and transmission, the CPU is not fully used, since these two

actions are delegated to the radio hardware, so the callback can be executed even if it happens to be overlapped with a packet transmission.

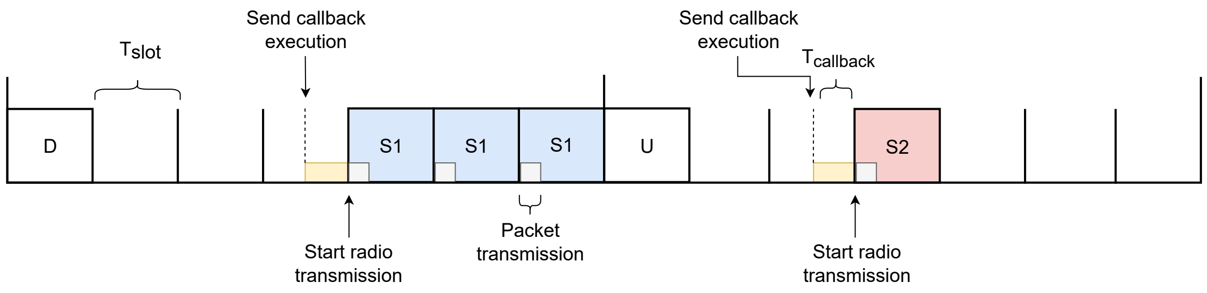

Figure 5.4: Desired and implemented send callback behavior (transmission side).

Symmetrically, Fig. 5.5 shows that the receive callback is executed after all the redundant receptions have passed (three for stream *S1* and only one for *S2*), in the time interval allocated to the execution of the callback.

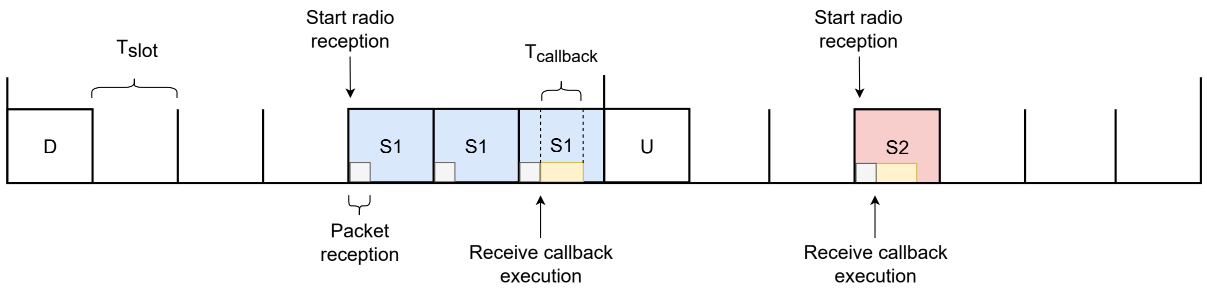

Figure 5.5: Desired and implemented receive callback behavior (reception side).

5.2. Write/Wait API

The callbacks API, although having a very low overhead, has the side effect of making the network slot size dependent on the execution time of the callback functions. Thus, even if this solution is suitable for deeply embedded applications where the callback operation is often as simple as reading from a sensor, is not flexible enough for more general computation. For this reason, a second API has been introduced to better decouple the network and application layer.

5.2.1. Overview

The second proposed and implemented API aims at providing a more *"file oriented"* interface, such as the one that TDMH already provides. It is in fact an extension of the existing interface. The *write/wait* API is intended for streams that are used to transmit some data, since, as described in Ch. 4, the problems of the current implementation reside in the transmission side. The aim is to provide an API that is able to somehow synchronize the *write* operation performed by the application with the data phase execution.
This API extends the set of available TDMH primitives. The *write* function is still present, but the *wait* primitive is added to the set of available ones: it can be used by the application in order to enter a waiting state until the moment at which the *write* execution has to take place, usually a small amount of time before the actual transmission time of the required stream. The *read* remains available for those streams that have to receive packets.

In the desired behavior, the application code can specify a **time advance** value when opening a new stream through the *connect* primitive. Then, the application will put itself into a **waiting** state (*wait* primitive) on the transmitting stream. The idea is that the stream (and the waiting application too) will be automatically **woken up** an amount of time equal to the specified advance before its assigned transmission slot. Each stream can specify a different wake-up advance value, according to the duration of the operations needed by the application to prepare the data to be sent. In fact, the time between the wake-up instant and the stream's slot start is thought to be used by the application to produce the new data and to call the *write* primitive, through which the new packet is built inside the session layer.

Fig. 5.6 exemplifies the moment in which the stream should be woken up and Alg. 5.2 shows the steps required for a basic usage of this API.

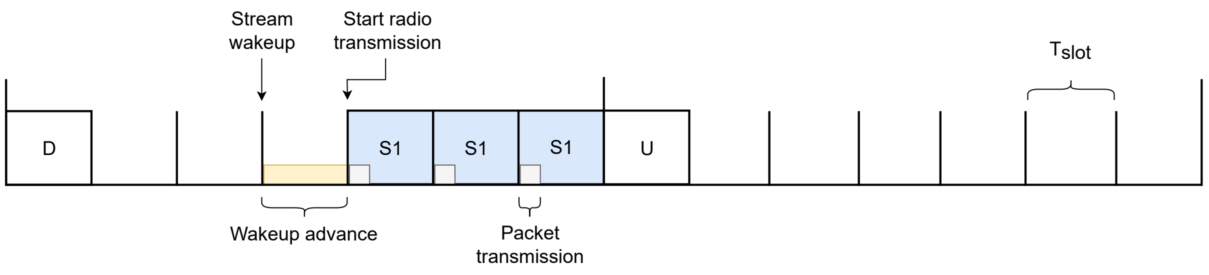

Figure 5.6: Desired behavior for stream transmission using the write/wait API.

Algorithm 5.2 Basic application behavior, with the Write/Wait API

```
1: stream ← connect(dest, port, timeAdvance, ...)
2: if stream $<$ 0 then
3:     print("Error")
4: else
5:     while stream.status == ESTABLISHED do
6:         wait(stream)
7:         data ← ...
8:         write(stream, data)
9:     end while
10:    close(stream)
11: end if
```

5.2.2. Implementation

Stream and StreamManager

The *Stream* class was extended by adding the two methods *wait* and *wakeup*:

- *wait*: this method blocks the caller on a condition variable. It has to be used by the application to wait until stream's transmission slot minus the required time advance.
- *wakeup*: it unlocks (signals) the condition variable used by *wait* to block the caller. It is called internally in the session layer and wakes up the stream and, consequently, the application.

Both methods can be accessed from outside through the *StreamManager*, where two homonymous proxy methods exist and take as a parameter the stream file descriptor or the *StreamId* of the required stream, respectively:

```
int wait(int fd)
bool wakeup(StreamId id)
```

The *write* method is also updated. In the current implementation, it is a blocking method: if the session layer buffer is already occupied, it waits on a condition variable until the buffer is cleared. This blocking mechanism was needed to guarantee the streams periods, but in the new API this is no more needed. The application blocks on the *wait* method until its wake-up time is reached, so the periodicity is enforced by the session layer internal algorithm that manages the wake-up actions. When the time advance is correctly sized, this mechanism ensures that the *write* operation will always find the buffer empty. Thus the *write* can proceed to copy the data to the packet buffer and to return without blocking. Anyway, the *write* will block on the same condition variable if the application continuously calls it, without using the *wait* primitive.

Assumptions

In order for this API to be correctly implemented, a few assumptions have been made:

1. If the stream wake-up advance is high enough, it might happen that a stream wake-up time does not fall inside the tile in which the stream has to transmit, but in a previous one. For the explained reason, the maximum allowed wake-up advance for any stream is equal to the duration of a TDMH tile, which is specified by the user inside the *NetworkConfiguration* object. This assumption is necessary to bound to one tile the maximum overlap between two consecutive schedules.
2. Whenever a new schedule is activated, the data structures relative to the last valid schedule have to be replaced by the new ones. Moreover, as said, the maximum overlap among the streams wake-up times belonging to two different schedules is one tile. As a consequence of the first assumption then, new schedules have to be received at least one tile before their own activation. This last assumption will be discussed in the following, together with the proposed solution.
3. Whatever value the application provides for the wake-up advance time, it is rounded to the nearest multiple of a data slot duration. This assumption is needed in order to keep the time instants in which all the events in system occur aligned with the time division slots. Starting from the schedule computed by the master node and then forwarded to all the dynamic nodes, the available information about the moment at which a stream has to transmit is represented by the offset, inside the same schedule, of the transmission slot assigned to that stream. Indeed, it is comfortable to keep reasoning in terms of number of slots and slot offset values.

Streams Wake-Up List

During the schedule expansion phase, the data structures required by the correct functioning of the *write/wait* API are computed and constructed. In particular, what is needed is a list whose elements contain:

- The *StreamId* of the corresponding stream.

- The *wake-up time* for the same stream, computed according to the stream transmission offset inside the schedule.

The wake-up list elements are represented through the *StreamWakeupInfo* data structure.

This list defines all the time instants at which the algorithm should execute and consequently act, waking up the required streams. Its elements are relative to the explicit schedule elements associated to a `SENDSTREAM` action (i.e. the stream has to transmit something and the current network node is the stream's source endpoint).

In a first implementation, the wake-up list was only composed by the time of the first appearance of each schedule element associated to a `SENDSTREAM` action, minus the stream's advance time. In fact, streams with lower periods may appear multiple times in the schedule, according to the schedule length itself. Moreover, only the first redundant packet of each of the mentioned occurrences was added to the list. Every time an element from the list was used, its wake-up time was incremented by a time amount equal to its own period, in order to handle streams periodicity. It can be simply demonstrated that it may happen that the elements in the list become no more order by their own wake-up time, even if they were in principle. This undesired situation would make the algorithm wake up streams at the wrong time or skip some of them. For this reason, an implementation using a C++ priority queue was also evaluated. The priority queue guarantees that the element with the lowest wake-up time (i.e. highest priority) is always the front element. Getting a copy of the priority queue front element is $O(1)$, but if an element has to be removed or added, the complexity is $O(log(n))$. After getting the front element of the queue, it should be removed, updated and reinserted into the data structure. A non-constant access time to this data structure is not compatible with an algorithm that aims at providing constant latencies and, consequently, also its execution times have to be consistent.

In the final solution, the first redundant transmission of each stream appearance whose action in the *explicit schedule* is `SEDNSTREAM` is added to the list. Indeed, if at every period a stream has to send a redundant packet, the stream itself needs to be woken up only before the first redundant transmission in order to perform a *write* operation. The following redundant packets are already managed and transmitted by the data phase. The wake-up slot (i.e. wake-up offset) and the wake-up time of each stream is computed by relying on its own offset, specified in the implicit schedule, and the stream wake-up advance:

$$wakeupSlot = offset - \frac{wakeupAdvance}{T_{slot}}$$

$$wakeupTime = (wakeupSlot \cdot T_{slot}) + T_{act}$$

where T_{slot} is the duration of a single data slot and T_{act} is the time instant at which the schedule will be activated. Wake-up times are always absolute times and they are computed using the schedule activation time as a base time.

Tab. 5.1 represents the wake-up list relative to the schedule presented in Fig. 5.7. Two streams are scheduled, (*S1* with period 1 tile and *S2* with period 2 tiles) with double redundancy. The schedule is composed by a single control superframes, the duration of each slot is $T_{slot} = 6\ ms$ and the stream advance is set to $T_{advance} = T_{slot}$. The schedule is activated at time T_{act}.

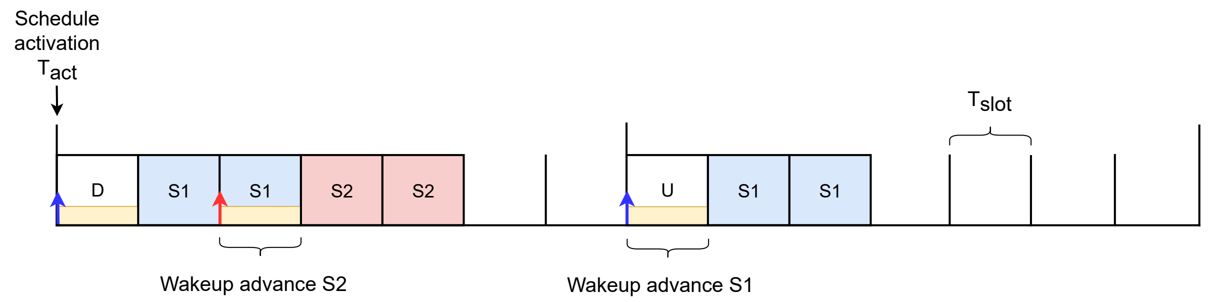

Figure 5.7: Example of schedule containing streams with different periods and their relative advance. Required wake-up time instants are represented by an upwards blue arrow for stream *S1* and a red one for stream *S2*. Schedule length is two tiles.

Stream ID	**Wakeup time**
$S1$	$T_{act} + 0\ ms$
$S2$	$T_{act} + 12\ ms$
$S1$	$T_{act} + 42\ ms$

Table 5.1: Resulting streams wake-up list from the final implementation. It contains the wake-up times of all the streams occurrences in the schedule, excluding redundant slots.

Stream ID	**Wakeup time**
$S1$	$T_{act} + 0\ ms$
$S2$	$T_{act} + 12\ ms$

Table 5.2: Resulting streams wake-up list from the first implementation. It contains only the wake-up time of first streams appearance in the schedule, excluding redundant slots.

Stream Wake-Up Scheduler

The main element on which the *write/wait* API relies for its correct functioning consists in an active object called *StreamWakeupScheduler*, which is in charge of waking up streams at the needed time, according to the current schedule and their required advance. It relies on the previously described streams wake-up list, which it directly receives from the TDMH module in charge of carrying out the schedule expansion process.

As shown in Fig. 5.8, the wake-up mechanism is internally managed by the TDMH stack session layer, since also the *StreamWakeupScheduler* is part of it.

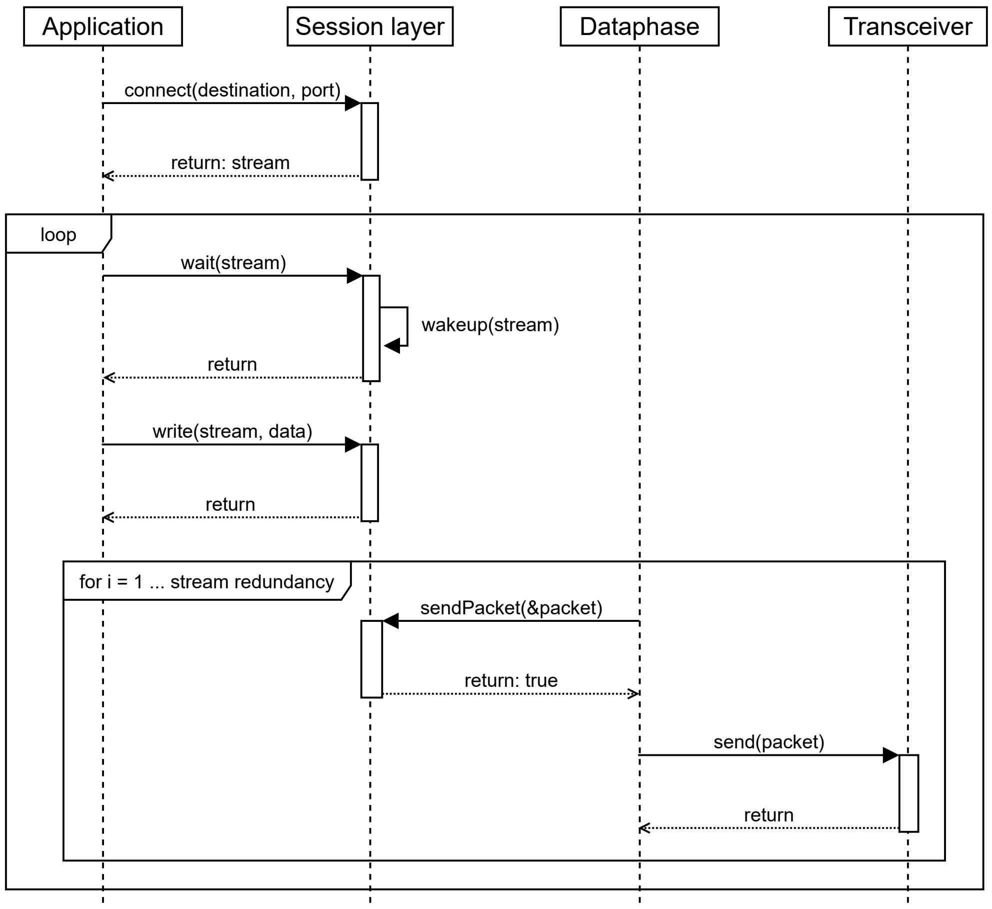

Figure 5.8: Sequence diagram showing the interaction between application, session layer and dataphase when using the write/wait API.

Basic mechanism The algorithm is based on the playback of the streams wake-up list, a list in which the required stream transmissions are ordered by their own wake-up time, which in this case represents the elements priority. The list is used as a circular buffer and represents a queue, in which the position of the element with the lowest wake-up time

is always the front one and the cost for getting it is $O(1)$. This is the desired temporal complexity, since it allows the algorithm to always have consistent computation timings with as less variability as possible.
At each iteration:

1. The *StreamWakeupScheduler* thread, takes the front element from the queue.
2. The removed element is replaced by an updated one. The update consists in incrementing the wake-up time by the duration of a schedule, that in turn is computed as:
   $$scheduleDuration = \#tilesInSchedule \cdot tileDuration$$
   In this way, since all stream occurrences in the schedule also appear in the wake-up list, elements are consumed once per schedule repetition and their ordering is maintained.
3. The thread sleeps until the wake-up time of the element removed from the queue is reached. When returning from the sleep, the *StreamId* contained in the element read from the queue is used to wake up the relative stream, through the *StreamManager*.
4. The algorithm continues by taking a new element from the queue and by repeating the above steps.

The described behavior is reported in Fig. 5.9.

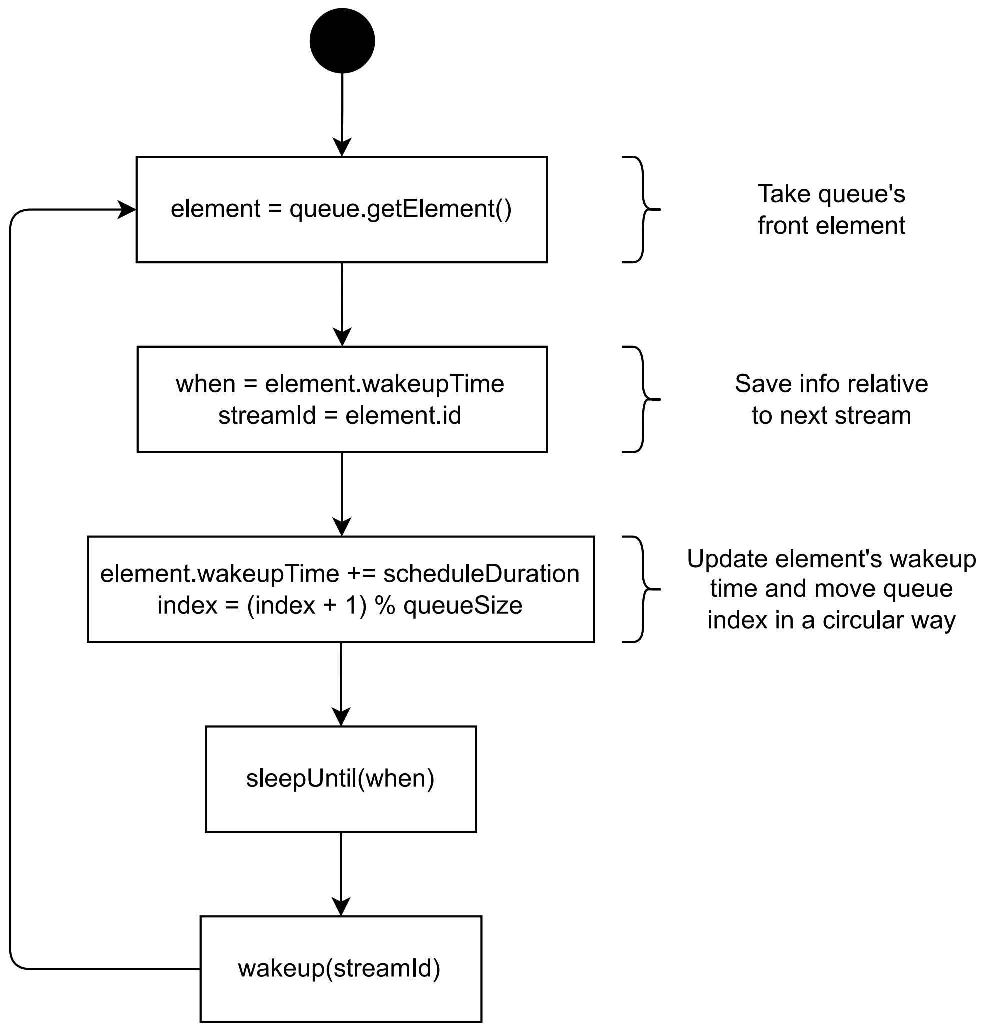
element = queue.getElement()
Take queue's front element
when = element.wakeupTime
streamId = element.id
Save info relative to next stream
element.wakeupTime += scheduleDuration
index = (index + 1) % queueSize
Update element's wakeup time and move queue index in a circular way
sleepUntil(when)
wakeup(streamId)

Figure 5.9: Flow diagram showing the basic mechanism of the *StreamWakeupScheduler*.

Finite State Machine The *StreamWakeupScheduler* algorithm is implemented through a simple finite state machine (FSM) with three states:

- `IDLE`: this is the initial state. The FSM remains in this state until the first schedule is received. After the schedule is received, the FSM moves to the `AWAITING ACTIVATION` state.
- `AWAITING ACTIVATION`: this state is used to manage the period that goes from the reception of a new schedule, until its activation tile is reached. When the new schedule is activated, the FSM moves to the `ACTIVE` state. Data structures related to the previous schedule are replaced by the newer ones.
- `ACTIVE`: the FSM remains in this state until no new schedule is received and the basic wake-up list playback is performed.

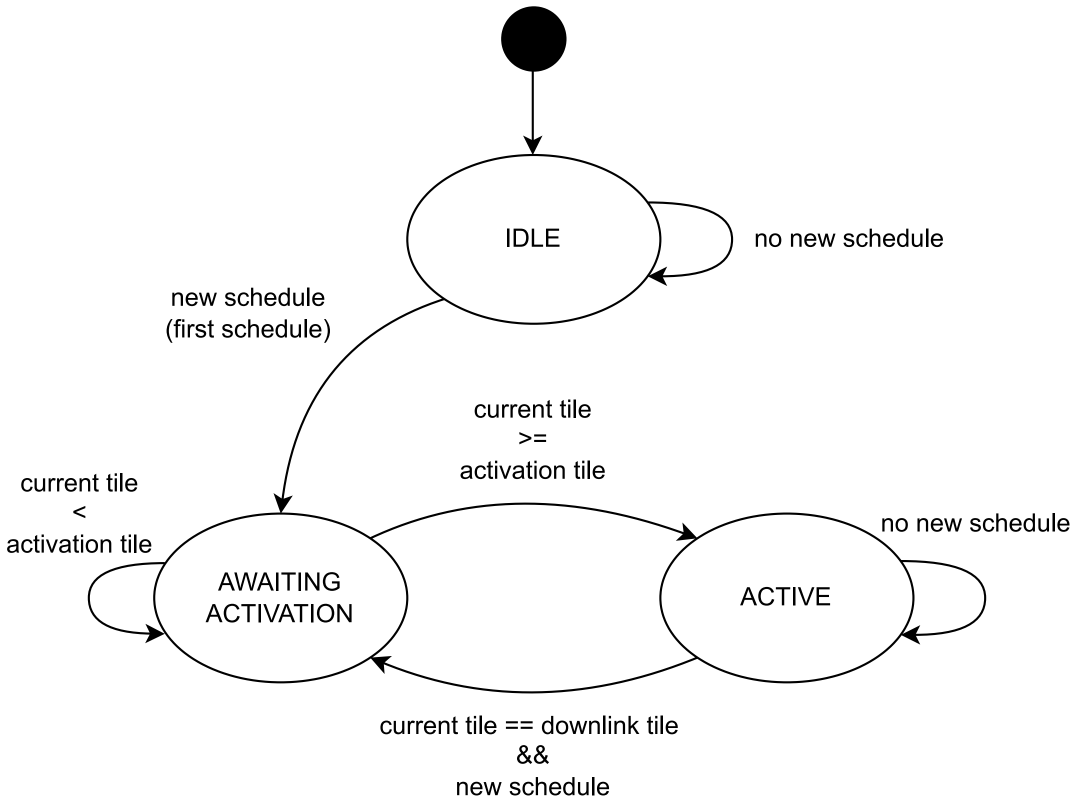

Figure 5.10: *StreamWakeupScheduler* finite state machine.

More details on the actions performed in the different FSM states are discussed in the following.

Handling New Schedules

New schedules are distributed during the downlink phase. This means that during downlink tiles, in particular at the end of downlink slots, the *StreamWakeupScheduler* should check if a new schedule has been received, in order to trigger a transition in the finite state machine and to update the current data structures.

According to the FSM state, different actions are performed:

- While in the `IDLE` state, the algorithm continuously performs a sleep action until the next tile time plus the duration of a downlink slot. This way it will always execute at the end of downlink slots, checking if a new schedule has been received and, if needed, move to the `ACTIVE` state.
- The solution to enforce checking for new schedules while in the `ACTIVE` state is to also add the *downlink slots end time* to the wake-up list, with no advance time.

Then, a new member is also added to the list elements to indicate the *element type*, being it a `STREAM` or a `DOWNLINK`. If the algorithm wakes up at the end of a downlink slot it checks for the existence of a new schedule and, if it is the case, it moves to the `AWAITING ACTIVATION` state.
Tab. 5.3 is the updated version of the wake-up list shown in Fig. 5.7 and Tab. 5.1, by including downlink slots too. In the example, the schedule is composed of one control superframe, which contains one downlink slot.

Type	**Stream ID**	**Wakeup time**
STREAM	$S1$	$T_{act} + 0\ ms$
DOWNLINK	–	$T_{act} + 6\ ms$
STREAM	$S2$	$T_{act} + 12\ ms$
STREAM	$S1$	$T_{act} + 42\ ms$

Table 5.3: Streams wake-up list containing also the downlink slots end time, referred to example in Fig. 5.7.

- Until the FSM remains in the `AWAITING ACTIVATION`, a new schedule has already been received but its activation time has not been reached yet. Then, no check is needed, since no re-scheduling can take place until the last computed schedule has been activated, but at that time the FSM would already be gone back to `ACTIVE`.

Handling Streams with Different Periods

In this solution, streams with different periods are easily managed. All the streams will always appear at least once in the schedule and whenever the algorithm takes an element from the queue also updates it with its next wake-up time. Since the schedule length, in number of tiles, is equal to the least common multiple among all the existing streams periods, actually streams appear a number of times equal to:

$$\#streamOccurrencies = \frac{\#tilesInSchedule}{streamPeriod}$$

So, a stream will be woken up a number of times equal to $\#streamOccurrencies$ per schedule repetition and its relative elements in the queue are also updated the same amount of times. Every time an element is updated, it had the lowest wake-up time and becomes the element with the highest one. This mechanism ensures that elements in the wake-up queue are always ordered and no problems arise while dealing with different stream periods. A schedule containing streams with different periods was represented in Fig. 5.7 and Tab. 5.1.

Handling Streams with Same Wake-Up Time

Consider two streams, *S1* and *S2* with the same wake-up time t_w and *S1* ordered before *S2* in the queue. After waiting until t_w in order to wake-up *S1*, the *StreamWakeupScheduler* thread would again sleep until t_w to wake-up *S2*. Since t_w is a time that has already passed, the *sleepUntil* primitive provided by TDMH immediately returns and the second stream is also woken up.

The same can happen among a stream and a downlink slot end time: the stream wake-up action, in this case, has higher priority, so the `STREAM` element is ordered before the `DOWNLINK` one. This means that the check on newly arrived schedules will be executed after the stream wake-up has been performed. In such a situation, the stream has to be woken up, since if a new schedule was not yet received, the older one is still valid. Even if a new schedule is received during the mentioned downlink slot, it will be activated some tiles in the future. By design, the schedule distribution always ends at least one tile before the same schedule activation. So, when the new schedule is received, the stream is for sure still open. It may then be removed as soon as the new schedule becomes active. This is the first reason why new schedules have to be received at least one tile before their own activation.

Streams Wake-Up Lists Subdivision

Problems can arise when a stream specifies an advance that makes its wake-up time to take place in the previous tile with respect to the one in which the stream has to transmit.
In Fig. 5.11 it can be seen how stream *S1* needs to be woken up in correspondence of the last slot of tile *N-1* for transmitting during tile *N*, since its advance is set to $T_{advance} = 2 \cdot T_{slot}$.

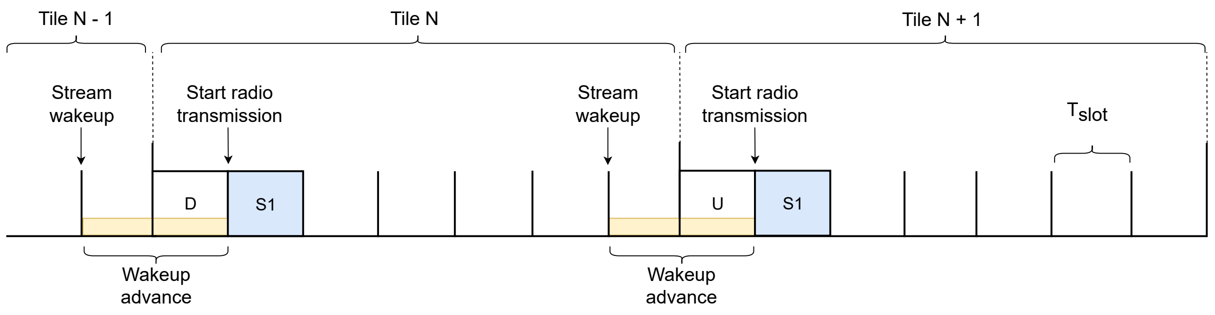

Figure 5.11: Example of schedule in which the stream *S1* has to be woken up during the previous tile w.r.t. the transmission slot tile.

The main problem, in this case, is that a stream may need to be woken up before the schedule itself is activated. For example, assume that a new schedule has to be activated at tile N: the wake-up of stream *S1* has to be performed during the previous tile with respect to schedule activation. This is the main reason why the maximum allowed wake-up advance is one tile and, thus, it is needed to receive new schedules at least one tile before their own activation, in order to have time to wake-up streams during the previous tile, if needed.

As said, the main problem is when a new schedule is going to be activated. Indeed, not all the streams that have a high wake-up advance cause this problem, but only those that need to transmit during the first schedule tile (since the maximum allowed wake-up advance is exactly one tile). So, only those that have a negative wake-up offset (in number of slots) with respect to the schedule activation moment. In the previous example, stream *S1* has to transmit during slot 2 in tile N, but its wake-up offset is -1 slot with respect to tile N (i.e. the schedule activation tile). In other words, only those streams whose first wake-up time relative to the new schedule is lower than its activation time.

In order to handle these corner cases, when computing the wake-up times, streams are divided into two lists:

- *currQueue*: it contains the wake-up time of those streams that have to be woken up and transmit while the schedule is already active.
- *nextQueue*: it contains the wake-up time of those streams that have to be woken up during the previous tile with respect to the one in which the schedule has to be activated (i.e. they transmit during the "next" schedule repetition).

Referring to the previous example, the wake-up time of the first occurrence of stream *S1* is added to the *nextQueue*, while the second one to the *currQueue*.

Both lists are relative to specific schedules. From now on they will be indicated as $currQueue_K$ and $nextQueue_K$, where K is the schedule identifier, which is an incremental value.

This distinction allows to perform different operations according to the current FSM state:

- `ACTIVE`: in this state the algorithm is running in a "stable scenario", meaning that a schedule is currently active and no new ones have been received. At each iteration, the algorithm takes the element with lower wake-up time among $currQueue_K$ and $nextQueue_K$, associated with the current schedule. In fact, at each schedule repetition also elements inside $nextQueue_K$ have to be checked out.
- `AWAITING ACTIVATION`: in this other state instead a new schedule has been received, so the $currQueue_{K+1}$ and the $nextQueue_{K+1}$ associated with the new schedule already exist. In particular:
  - It may happen that a schedule is received several tiles before its activation. If the current tile is not the last one before the activation, the algorithm acts exactly as in the `ACTIVE` state, considering always that the maximum allowed wake-up advance for a stream is one tile.
  - If instead, the current tile is the last one before the new schedule activation, the algorithm, in order to take an element from the streams wake-up lists, compares the $currQueue_K$ associated to the current schedule and the $nextQueue_{K+1}$ of the new schedule. This is because by checking also $nextQueue_K$ the algorithm may wake up a stream whose offset changed in the new schedule (and so also the needed wake-up time) or directly removed. Waking up a stream in $nextQueue_K$ that has been moved in the new schedule means waking it up at the wrong time, causing a variation in the experienced latency. If a stream was removed instead, it would cause a useless wake-up, that will not correspond to any transmission. New streams may also appear in $nextQueue_{K+1}$ and they have to be woken up before even activating the new schedule. Indeed, at this point, $nextQueue_K$ represents wake-up times of transmissions that would take place after schedule $K+1$ activation (since its elements have been updated over time) and schedule K is instead no more valid. Therefore, it has to be ignored.

To be more specific, there are no guarantees that the algorithm wakes up during the last tile before the new schedule activation. For example, if both $nextQueue_K$ and $nextQueue_{K+1}$ are empty, the last tile before activation is an uplink tile (so no check on downlink slots is performed) and streams have a period higher than 1. It is instead guaranteed that the *StreamWakeupScheduler* executes at least once per control superframe, since in each of them at least one downlink slot has to be present. So, actually while in the `AWAITING ACTIVATION` state, the algorithm checks if the last superframe before the new schedule activation has been reached, which also includes the last tile. By doing so, the $nextQueue_{K+1}$ is also checked in previous tiles of the same control superframe, not only the last one, to which by design the wake-up times contained in $nextQueue_{K+1}$ can not belong. This is not a concern: $nextQueue_K$ is already has to be ignored (its streams were already woken up during the first tile of schedule K) and comparing the first element of two lists is not a heavy operation at all.

Notice that, in the case in which the next considered wake-up time is successive to the new schedule activation, the algorithm waits until the schedule activation time, in order to swap the data structures before proceeding, without removing or updating any element in any queue.

To summarize, Fig. 5.12 to Fig. 5.15 represent the described mechanism.

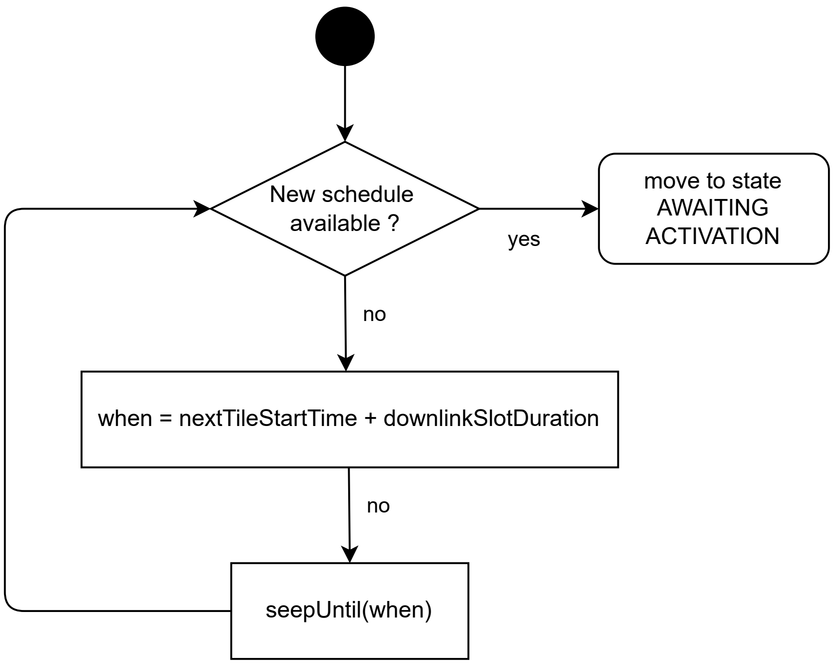
New schedule available ?
yes
move to state AWAITING ACTIVATION
no
when = nextTileStartTime + downlinkSlotDuration
no
seepUntil(when)

Figure 5.12: *StreamWakeupScheduler* `IDLE` state.

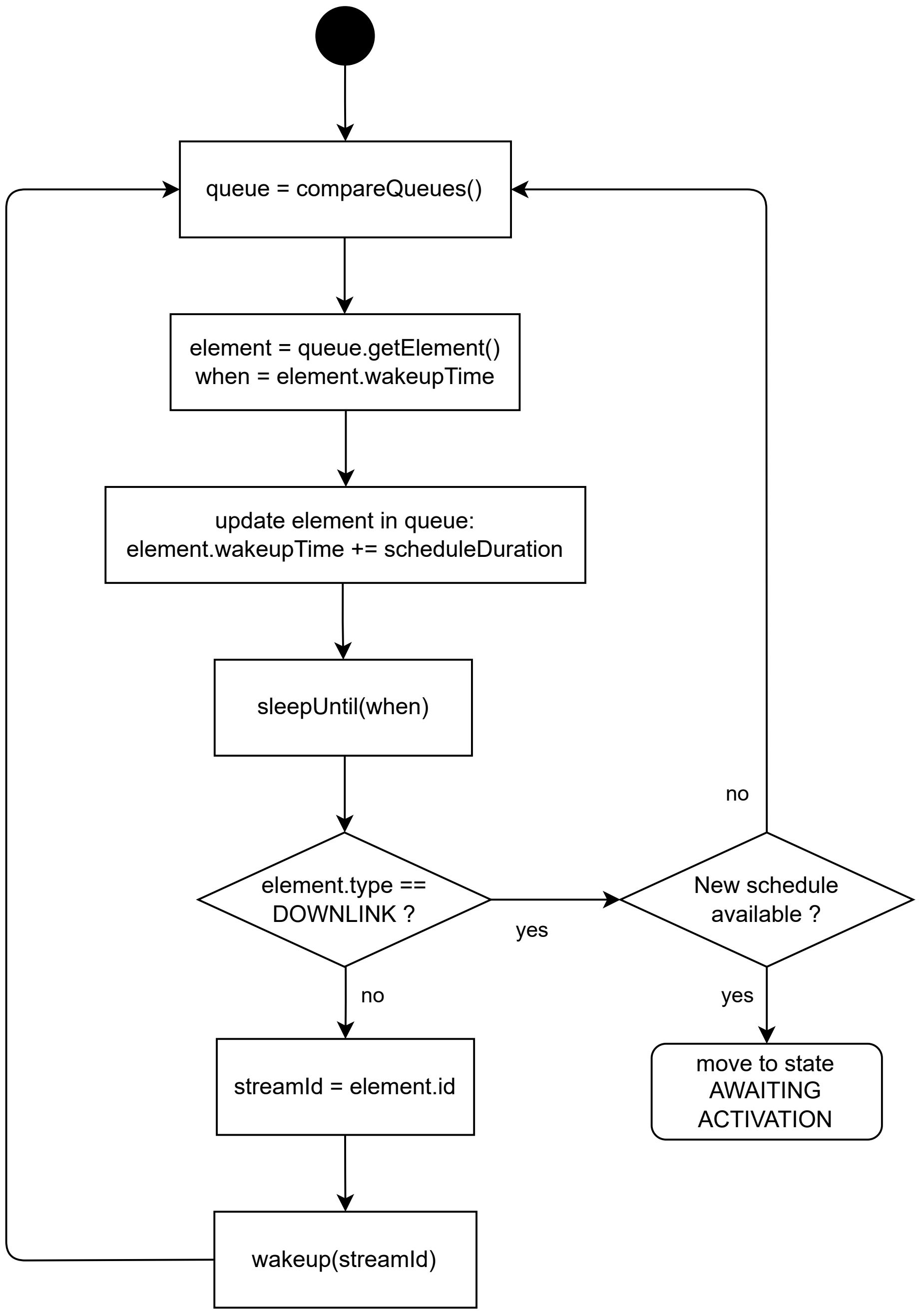
queue = compareQueues()
element = queue.getElement()
when = element.wakeupTime
update element in queue:
element.wakeupTime += scheduleDuration
sleepUntil(when)
element.type == DOWNLINK ?
yes
no
New schedule available ?
no
yes
move to state AWAITING ACTIVATION
streamId = element.id
wakeup(streamId)

Figure 5.13: *StreamWakeupScheduler* `ACTIVE` state.

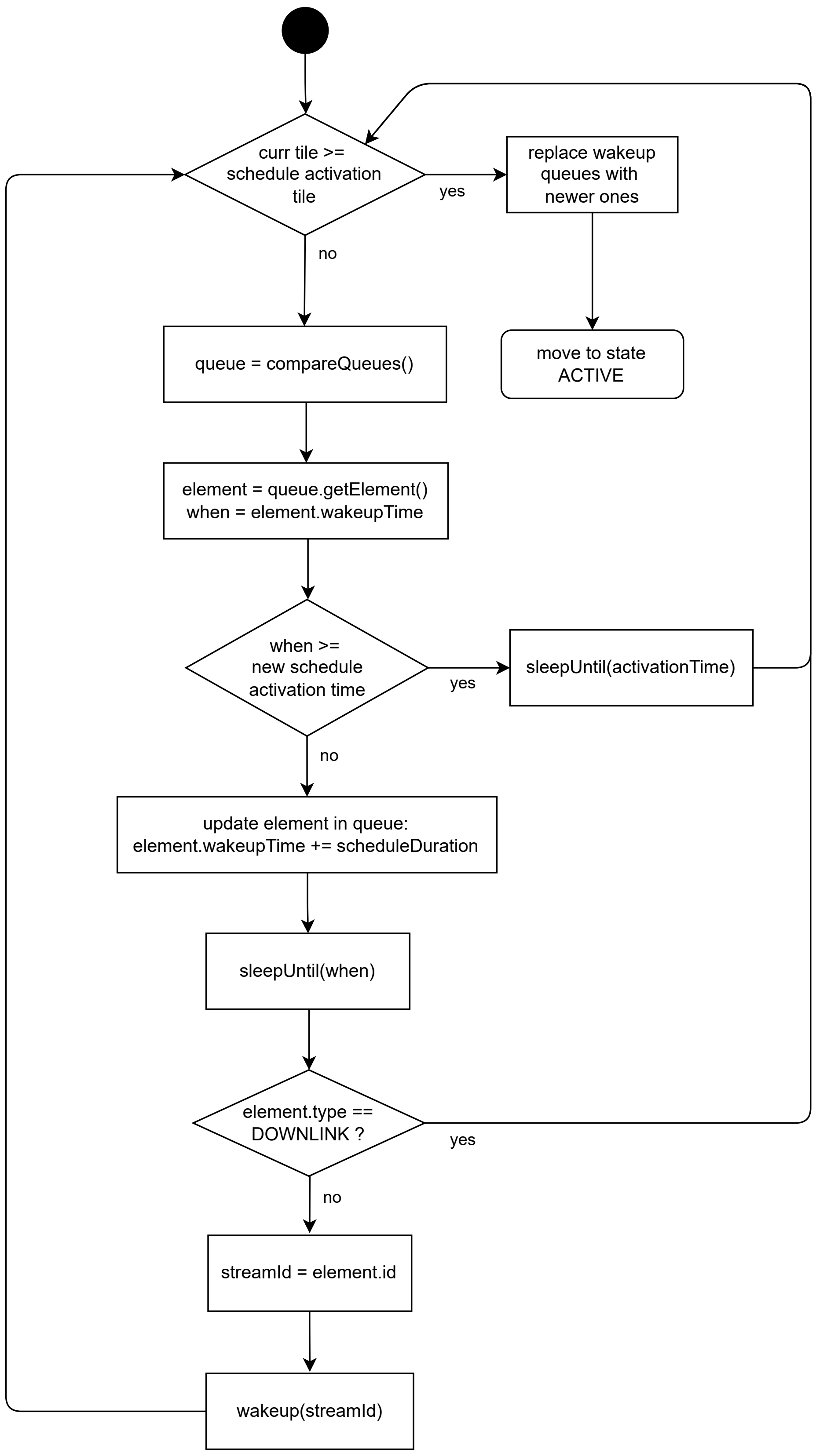
curr tile >= schedule activation tile
yes
replace wakeup queues with newer ones
move to state ACTIVE
no
queue = compareQueues()
element = queue.getElement()
when = element.wakeupTime
when >= new schedule activation time
yes
sleepUntil(activationTime)
no
update element in queue:
element.wakeupTime += scheduleDuration
sleepUntil(when)
element.type == DOWNLINK ?
yes
no
streamId = element.id
wakeup(streamId)

Figure 5.14: *StreamWakeupScheduler* `AWAITING ACTIVATION` state.

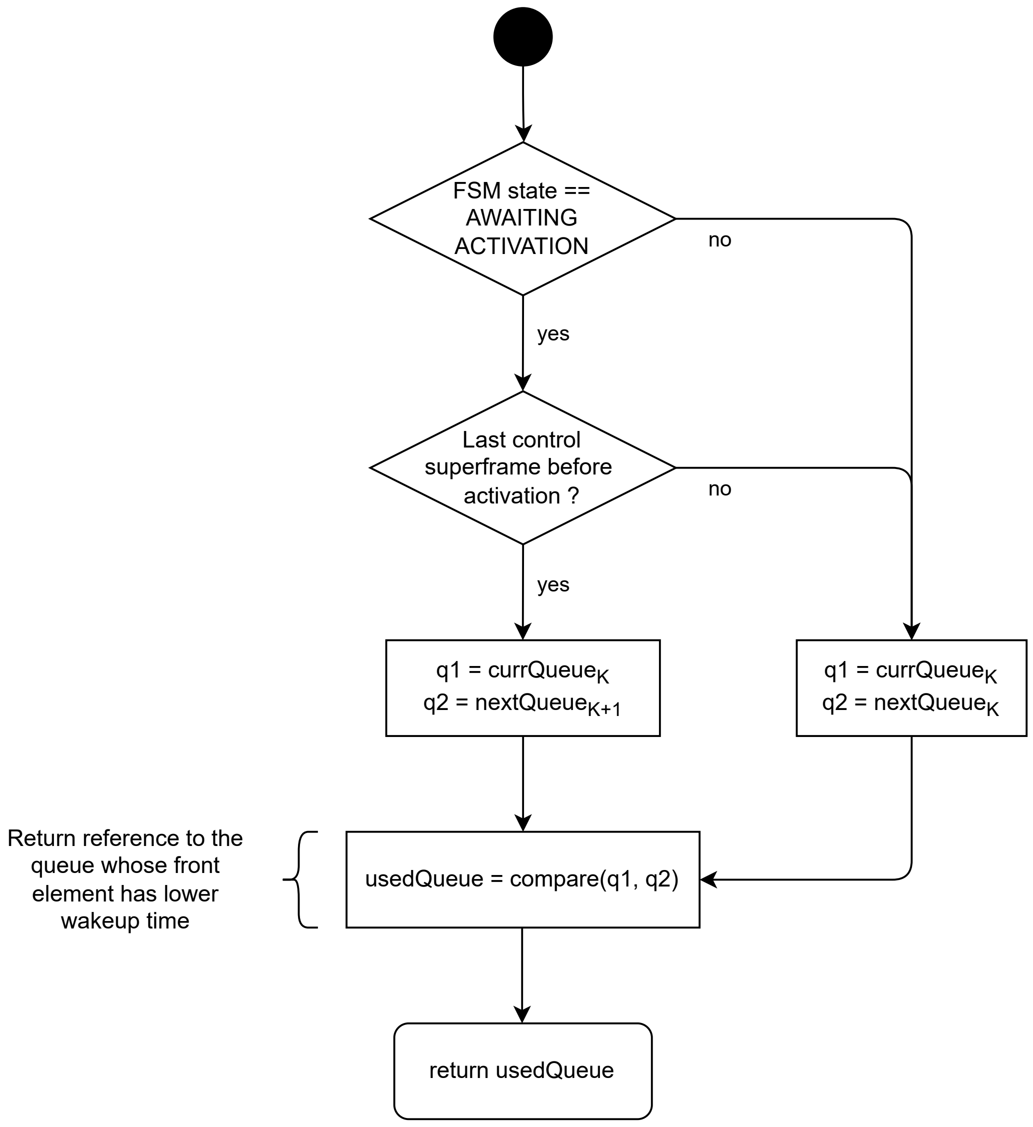
FSM state == AWAITING ACTIVATION
no
yes
Last control superframe before activation ?
no
yes
q1 = currQueue$_K$
q2 = nextQueue$_{K+1}$
q1 = currQueue$_K$
q2 = nextQueue$_K$
Return reference to the queue whose front element has lower wakeup time
usedQueue = compare(q1, q2)
return usedQueue

Figure 5.15: *compareQueues()* function used to get a reference to the queue from which the next element has to be taken, according to the current FSM state.

Handling Removed and Added Streams

As said in the previous paragraph, closing a stream or receiving a schedule in which a stream (or more than one) has been removed (e.g. after a topology change or de-synchronization) may generate some corner cases around the moment at which the new schedule has to be activated. In general, it has to be avoided to wake up streams that are no more existing and to skip waking up new streams, which should instead take place.

Consider the basic situation shown in Fig. 5.16, in which in schedule K only stream *S1* exists. Analogously, in schedule $K+1$ only stream *S2* is present. Both streams have a period of 1 tile and their wake-up advance is $T_{advance} = T_{slot}$. In this case, both $nextQueue_K$ and $nextQueue_{K+1}$ are empty since the wake-up time of both streams appearances is contained in the same tile as their respective transmission slot. The algorithm will normally proceed taking elements from $currQueue_K$ and then from $currQueue_{K+1}$ after schedule $K+1$ is activated. After the first two *S1* wake-ups, the following wake-up time for stream *S1* would be equal to schedule $K+1$ activation time. The algorithm will then sleep until T_{K+1} (schedule $k+1$ activation time), without taking the element from $currQueue_K$. At this point, when the activation is reached, the data structures are replaced by the newer ones and a new wake-up of stream *S1* cannot take place. In fact, from this point on, only $currQueue_{K+1}$ will be used. Referring to the example, a new element will be extracted from the queue and immediately woken up, in correspondence to schedule $K+1$ activation.

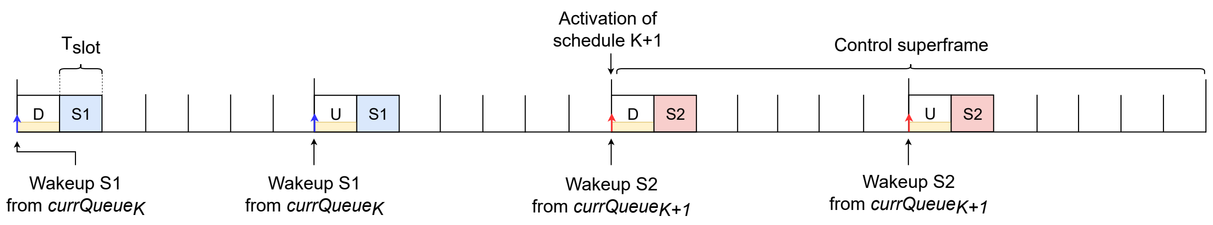
Tslot
Activation of schedule K+1
Control superframe
D
S1
U
S1
D
S2
U
S2
Wakeup S1 from currQueueK
Wakeup S1 from currQueueK
Wakeup S2 from currQueueK+1
Wakeup S2 from currQueueK+1

Figure 5.16: Example in which $nextQueue_K$ and $nextQueue_{K+1}$ are empty. Upward blue and red arrows indicate wake-up actions that take place, respectively from streams *S1* and *S2*

Fig. 5.17 shows, instead, an example of a possible corner case. Stream *S1* has a wake-up advance equal to $T_{1,advance} = 2 \cdot T_{slot}$ and it is removed in the new schedule. A new stream, *S2*, is also opened and its wake-up advance is set to $T_{2,advance} = 4 \cdot T_{slot}$.

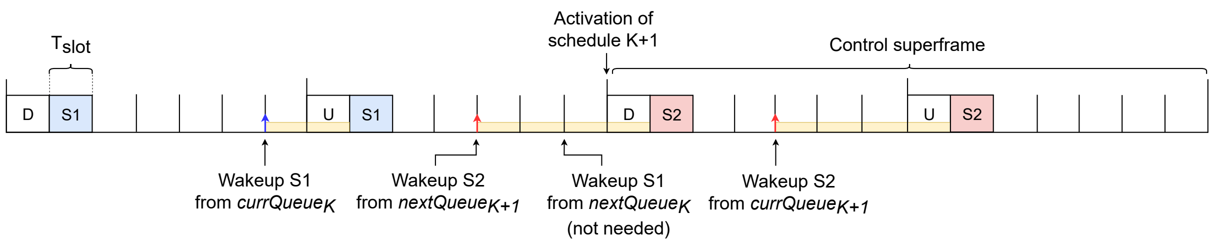
Tslot
Activation of schedule K+1
Control superframe
D
S1
U
S1
D
S2
U
S2
Wakeup S1 from currQueueK
Wakeup S2 from nextQueueK+1
Wakeup S1 from nextQueueK (not needed)
Wakeup S2 from currQueueK+1

Figure 5.17: Example in which a stream that appears also in the $nextQueue_K$ is removed from the schedule and a new one is added. Upward blue and red arrows indicate wakeup actions that take place, respectively from streams *S1* and *S2*.

According to the algorithm behavior explained, after receiving the new schedule, the FSM moves to the `AWAITING ACTIVATION` state. Here, after the first downlink slot, it notices that the last control superframe before the new schedule activation has been reached. From this point then, $nextQueue_K$ is ignored and only $currQueue_K$ and $nextQueue_{K+1}$ are considered. In the example shown in Fig. 5.17, after the first *S1* transmission takes place (it was woken up during the previous tile), the algorithm also wakes up the second appearance of *S1*. Since $nextQueue_{K+1}$ also contains a new appearance of *S2*, the wake-up of this second stream will take place. The following appearance of *S1* is instead skipped since it is contained in $nextQueue_K$, which is ignored, and its relative transmission would take place after the activation of the new schedule $K+1$, in which *S1* is no more present. After the activation of schedule $K+1$ the algorithm moves back to the `ACTIVE` state, proceeding with its normal execution until a new schedule is computed.

Apart from not waking up streams that are no more existing in the schedule, it is important to wake up the newly added streams without skipping them. As shown in the previous example, also newly added streams (as for *S2*) are managed.

This mechanism also handles the case in which a stream is moved inside the new schedule (i.e. is re-scheduled with a different offset). In fact, the situation in which a stream was moved from slot i to slot j can be assimilated to the case in which the stream at offset i was closed and a new stream with offset j was instead added to the schedule, which resembles the previously shown examples.

Notice that the *wakeup* method of the *Stream* class is also called every time a stream gets removed from a schedule, either if explicitly closed or due to a de-synchronization or topology change. This is necessary to prevent the application to remain stuck forever on the *wait* call, without being able to react, for example requesting the stream to be opened again.

Handling Re-Transmitted Schedules

During the distribution of a new schedule, the schedule itself is split into multiple packets and sent to all the nodes in the network. Each of these packets is redundantly transmitted. Whenever all the redundant transmissions of one or more schedule packets fail, the dynamic node receives an incomplete schedule. In this case, the node issues a retransmission request. It can happen that when finally the schedule is correctly received by all the nodes, the activation tile for that schedule has already passed. If it is the case the wake-up lists may contain wake-up time values that are in the past. If this happens, when the *StreamWakeupScheduler* takes an element from the queue and performs a sleep, this last action would immediately return, causing some consecutive and very fast streams wake-up actions, until the wake-up times align again with the current time (since the wake-up time is incremented every time an element is taken from the queue).

On the contrary, in this situation, wake-up time values have to be considered to be invalid. To avoid undesired wake-ups, all the elements of the wake-up lists are incremented and aligned to the next schedule repetition (i.e. to the next data superframe start time).

This mechanism is shown in Fig. 5.18. In this example, all the wake-up times would be aligned to T_{actNew}, so they all would be incremented by an amount of time equal to $T_{actNew} - T_{act}$.

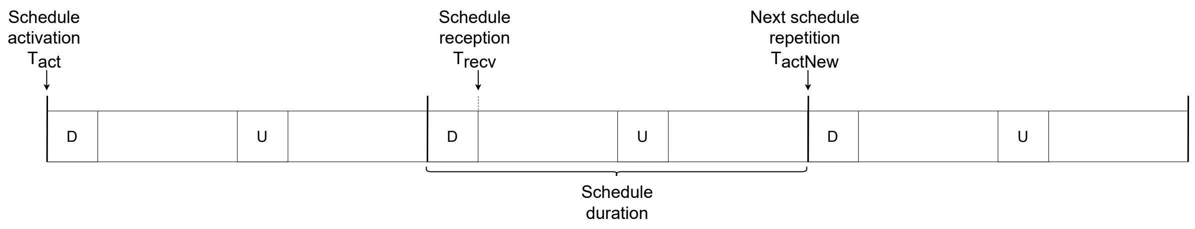

Figure 5.18: Example in which a schedule is received after its own activation tile has already passed.

The drawback is that some possible intermediate wake-up actions are not performed, leading to some missed packets until T_{actNew} is reached, but, at that point, the algorithm will start again from a consistent state.

5.3. APIs Comparison

In this section, a comparison among the two presented API is provided, underlining the pros and cons of the two implementations.

5.3.1. Callbacks API

The proposed callbacks implementation has on its side the ease of use: only the callback functions and one number representing the upper bound of the execution time of the callbacks themselves have to be specified by the user inside the application code.

The implemented API also exploits the existence in TDMH of the data phase, which is already in charge of managing transmissions and receptions from the opened streams at the correct time and slot. Avoiding new threads should not be underestimated, since threads also bring with them problems related to synchronization and harder debugging.

On the other side, the API relies on the fact that the user specifies a correct value for the callbacks execution time, which also has to be minimized as much as possible, by limiting the number of operations performed inside the callback itself. These operations should be limited to copy data from or to the session layer and some quick processing. Anyway, the user is responsible for the implementation of the callback functions and if something goes wrong inside them, the data phase (and so the entire MAC) will be affected, e.g. by unexpected delays, which can even make the node non-synchronized with the TDMH time slots (and so, for example, causing packets to be missed and the node's reliability to get lower).
Anyway, it is impossible to avoid leaving to the user the responsibility of specifying network-related parameters, as it already happens for example for the TDMH tile duration and the maximum number of allowed hops. Since the callbacks execution time directly influences the time slots duration, it is considered as a protocol-related parameter.

Since callbacks execution time is considered to make time slots longer and it is usually an upper bound of the real needed computation time, some more time elapses among two consecutive streams transmission and reception inside the schedule. This amount of time is allocated specifically for callbacks execution. Moreover, callback functions are executed in the same thread of the data phase and the MAC, which is the highest priority one. For the two mentioned reasons, when using this API, latency is less affected by the amount of computational load the node is subject to.

5.3.2. Write/Wait API

This second presented API does not add any usage difficulty with respect to the existing implementation. In fact, for the user it is needed only to add a parameter to the *connect* primitive (the stream's wake-up advance time) and to also call the *wait* function among two consecutive *write* operations. It provides some flexibility in the choice of the wake-up advance too, that can be differently sized for each stream, according to the needed computations.

The implementation of the *StreamWakeupScheduler* as an active object, allows to better integrate it with the codebase, without the need of re-designing a substantial part of existing modules. It has to be considered that adding a new thread may require some attention to the synchronization mechanisms, but since the algorithm only needs one input (the streams wake-up lists) this is not a big deal. Furthermore, this thread spends most of the time sleeping and waiting the correct time to wake-up the next stream or the next schedule reception.

As for the callbacks API, the sizing of the wake-up advance is left to the user, who also has to guarantee that the application processing time does not exceed the specified value, otherwise packets misses may take place, reducing the network reliability. But if the time advance is correctly sized, the API guarantees that the application will always find a free session layer buffer when calling the *write* primitive, which ensures that the new packet will be sent during the first available transmission slot.
Moreover, it is left to the user to remember to call the *wait* primitive, in order to synchronize with the underlying network stack layers execution and, so, with streams transmission slots.

With respect to the callbacks implementation, the *StreamWakeupScheduler* is executed in a thread that has lower priority, since only the MAC has the maximum available one. In addition, when a stream is woken up, the application will perform its computation during a previous slot with respect to the stream transmission one, possibly being overlapped with other operations of the system, without having a dedicated time interval, as it is for the callbacks API instead. The execution of the *read* primitive on the reception side also takes place on a thread that does not have the highest possible priority. This means that when the computational load on the node becomes very high, the latency measures are affected by a higher noise with respect to the callback functions implementation, e.g. during the experiments in which all the nodes open streams towards the master which is in charge of logging all the network events and received messages, apart from executing the main TDMH algorithms, like the topology collection and the scheduler. This last one in particular, when the network contains tens of streams, may need a time that is in the order of some seconds.

5.4. APIs Interoperability

The two presented APIs respect all the time constraints defined by the TDMH protocol. This means that all the operations that involve transmission and reception of data packets are, of course, always synchronized with TDMH data slots, according to the active schedule. As such, the two APIs are completely alternative and **interoperable** the one with the other. Applications running on different nodes, in the same network, can use heterogeneous APIs. A single node can also be using different APIs for distinct streams. The extreme case is when for the same stream, the transmitting node uses one API and the receiver instead uses the other one.

5.5. Data Phase

Some modifications to the current implementation of the data phase are needed, in order to reach the thesis goal, due to its incompatibility with the proposed APIs. In this section, a description of the current data phase policy is provided. Then, the updates made to the current implementation are presented, both on the transmission and the reception side, switching to a new policy, referred to as "as late as possible" policy.

5.5.1. "As Soon as Possible" Policy

The data phase implementation is based on the *"as soon as possible"* policy, meaning that as soon as an operation ends, the following one can start. This is made in order to occupy as much as possible idle time intervals. Moreover, the current implementation tries to anticipate the moment in which data is retrieved from the session layer, in order to consequently anticipate the moment in which the buffer becomes empty and the next transmission operation can be completed. After retrieving the packet, the data phase schedules the radio transmission for the next involved stream time slot. The implemented policy does not guarantee that the data phase is executed always with the same advance with respect to the moment in which a transmission slot starts.

When the callbacks API is used, the described data phase behavior would change the moment at which the transmission callback is executed, having reflections on the end-to-end latency of data packets.

If instead the new *write* primitive is used, it can happen that the application provides to the session layer the data to be sent after the data phase has already been executed during the current period. In this situation, when the data phase accesses the session layer buffer, it does not find any packet ready and so the transmission does not take place, leading to a missed packet. In the previous implementation of the *write* method this could not happen, since the it was a blocking procedure, which was unlocked by the data phase itself. In order to retrieve the data, the data phase uses the *Stream* class method *sendPacket*, which returns the actual packet to be sent. Then it waits until the start time of the transmission slot, where the packet is sent over the radio module. But, as described, when the data phase checks for the existence of a ready-to-be-sent packet, the *write* may not have been executed yet. In that case, no packet is ready and the transmission will be skipped. Assume for example that a stream transmission is scheduled during the first slot after the downlink one. Even if the application specifies a correctly sized time advance for the needed computations (e.g. 1 time slot), the *write* operation could take place sometime after the *wakeup* one. The application, then, is affected by the fact that the data phase execution may be anticipated. For example, during downlink timesync slots usually the time needed for executing the synchronization algorithm is lower than the computation time of other downlink slots. Therefore, the data phase may check if the packet is ready before it was actually created, as depicted in Fig. 5.19.

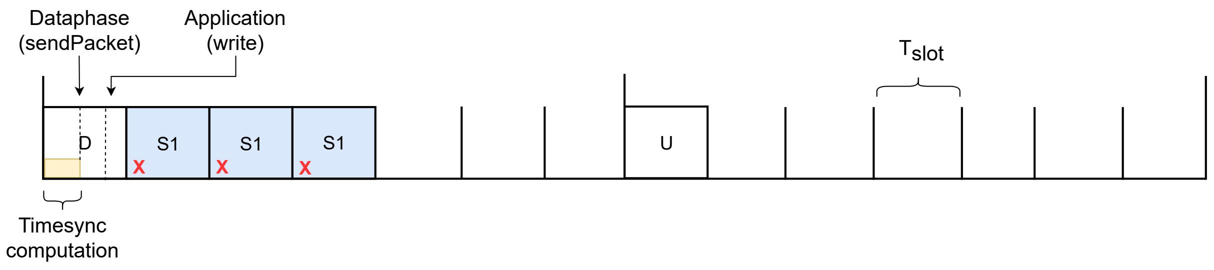

Figure 5.19: Undesired situation in which the data phase execution is anticipated before the *write* primitive.

In the desired behavior, it must be guaranteed that the data phase checks for a new packet existence after the *write* has already been executed.

In addition to what has been said, the current data phase implementation for the reception side has a direct impact on the value of the end-to-end latency. In general, the data phase:

- Calls the *Stream* method *receivePacket* when a packet is received.
- Calls the *Stream* method *missPacket* when a packet is missed.

Consider now the last of a group of redundant transmissions. The latency changes according to whether the last redundant packet is correctly received or misses. In fact, if something is being received, the data is returned to the application after a time equal to T_{tx}, with respect to the last redundant slot start time. T_{tx} is indeed the time needed to transmit (and receive) the packet.
But, as soon as the data phase on the receiving side notices that the packet was missed, it calls *missPacket*, which immediately returns the received data to the application (if any was received during previous redundant transmissions).

Fig. 5.20 and 5.21 show the described situation. Remember that if some data is received, it will be always delivered to the application after all the redundant receptions (or misses). Then, in the case in which the last redundant packet is received, even if some of the previous ones were missed, the time needed for a complete packet reception from stream

S1 can be computed as:

$$T_{rx} = (R - 1) \cdot T_{slot} + T_{tx}$$

where R is the stream redundancy value, T_{slot} is the duration of a single TDMH data slot and T_{tx} is the time needed to transmit the packet.
But, in the case the last redundant packet is missed (Fig. 5.21), T_{rx} is lower, since it is not necessary for the data phase to wait until the last redundant transmission is completed too. So, for stream *S1* the reception time becomes:

$$T_{rx} = (R - 1) \cdot T_{slot}$$

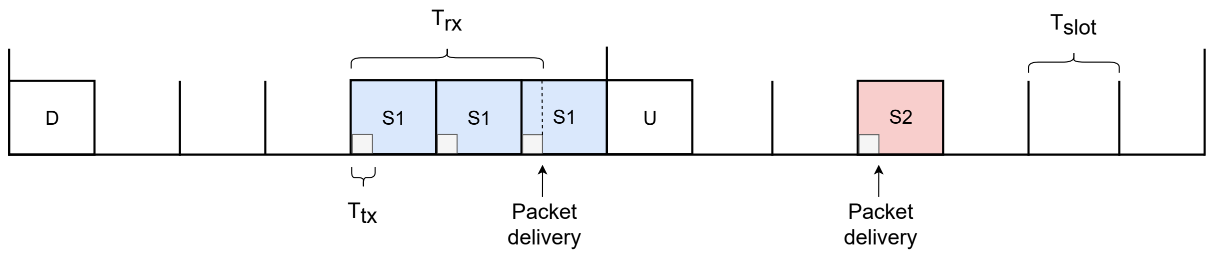

Figure 5.20: Example in which the last redundant packet is received by the destination node.

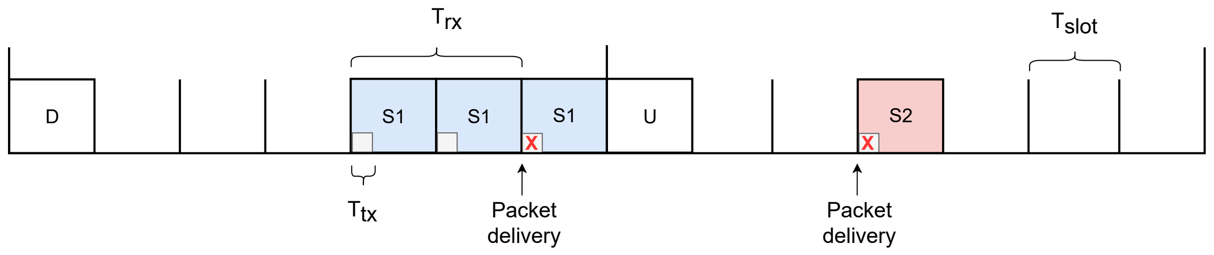

Figure 5.21: Example in which the last redundant packet is missed by the destination node.

If misses occur during other redundant transmissions that are not the last one, no problems take place, since the successive redundant packets still have to be sent (and received) and the data phase cannot call *missPacket* beforehand. So, at least $(R-1) \cdot T_{slot}$ elapse before the received data is returned to the application.

As introduced, the current implementation may lead to undesired packets misses and the variability of the time needed to perform a complete packet reception has a direct impact on the end-to-end latency. This policy is no more compatible with the implemented APIs and with the goal of having constant, or at least bounded, latency for all the transmitted packets.

5.5.2. "As Late as Possible" Policy

On the contrary, the required policy is *"as late as possible"*. In this context, this means that the data phase has to retrieve the data to be sent as close as possible to the real transmission slot.

According to the used API this has two different meanings:

- *Callbacks API*: the data phase has to wait until the slot start time minus the callbacks execution time, that is the time instant in which the send callback has to be executed. After executing the callback, the packet to be sent will be ready in the session layer, thus the data phase can get it and the transmission can be scheduled for the slot start time.
- *Write/Wait API*: the data phase can wait until the transmission slot starts, since it will only have to get the data from the session layer and send it. Since streams can specify the advance time in which operations are performed to produce the new data to be sent (e.g. sampling of a sensor), if the advance is correctly sized, it is guaranteed that the data phase will find a ready packet at the slot start time.

On the receiving side instead, the *"as late as possible"* policy implies that the data phase has to delay the execution of *receivePacket* (or *missPacket*) as much as possible after the reception (or miss) of all the redundant packets. It has to be kept in mind that any delay introduced in this phase has a direct impact on the time instant at which the received data will be delivered to the application.
When the TDMH MAC context is started, it computes the time needed to transmit a packet with a size equal to the maximum allowed one. This is necessary to allocate time for packets transmission inside the data slots, up to the maximum size. The same computation can be further exploited in the data phase context: whenever a packet is received or missed the data phase can wait until the slot start time plus the maximum time needed for a single packet transmission, $T_{tx,max}$.
Fig. 5.22 shows the mechanism. Considering stream *S1*, the time needed for a complete reception of the triple redundant packet and deliver the it to the application is now fixed and it is computed as:

$$T_{rx} = 2 \cdot T_{slot} + T_{tx,max}$$

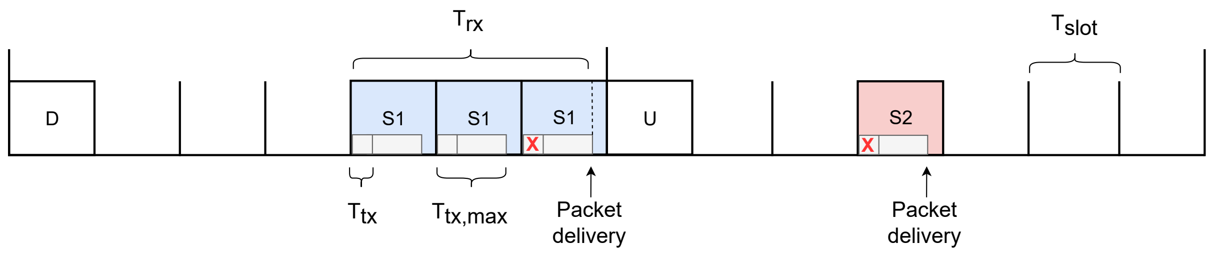

Figure 5.22: Dataphase correct behavior on the receiving side, including the $T_{tx,max}$ time slack.

Forcing the data phase to wait after each receive (or miss) will increase the end-to-end latency by a few milliseconds, such as only packets with maximum size are exchanged through the network. This is the price to pay for keeping the latency constant, instead of having it fluctuate depending on whether the last redundant packet was received or missed. Indeed, $T_{tx,max}$ is an upper bound of the transmission time of data packets, since it only considers maximum size packets (size already bounded by TDMH). The real packets size is for sure lower than or equal to the maximum one and so is their transmission time too.

5.5.3. Radio Startup Time and Cryptography

In the real world, every time the physical layer has to be accessed for transmission or reception, it needs some time to set up. This setup time is considered in the data phase: if the action associated with the next slot is to send or to receive, the radio startup time is accounted for executing the data phase with the correct time advance before the slot start time, in order to leave the needed time to the radio setup. This setup time in TDMH amounts to $T_{startup} = 0.5\ ms$.

Moreover, if the crypto is enabled, each packet has to be encrypted before its own transmission and decrypted after the reception. The time needed for this operation was measured to be $T_{encrypt} = 0.11\ ms$ and $T_{decrypt} = 0.12\ ms$.

This means that if the callbacks API is used, the send callback execution will be always anticipated by a total of $T_{startup} + T_{encrypt}$. This constant advance has an impact on the end-to-end latency too. It is not a big concern, since it is always a fixed value and does not affect latency's variability, but it has to be considered when looking at absolute values, for example when comparing the theoretical and the real latency entity.
The radio startup time does not have any reflection on the receive callback instead, since this function is called after the packet has been received (or missed), and so after the

radio already operated and the packet (if any) has been decrypted. The receive callback is only delayed by an amount of time equal to the decryption time of the incoming packet.

If the *write/wait* API is used, the data phase transmission advance time does not affect the end-to-end latency, since it is overlapped the stream wake-up advance, but it implies that the real time left to the application to produce a new packet and to copy it to the session layer is the specified wake-up advance time minus the data phase one, that is: $T_{advance} - (T_{startup} + T_{encrypt})$. This is the time the application has for computation between its own wake-up and the moment in which the data phase will check if a new packet is ready for transmission. On the reception side, though, the time needed for packet decryption has to be considered, as it delays the moment at which the data is delivered to the application.

Fig. 5.23 shows a detailed view of the explained radio startup time and the time allocated to cryptography execution.

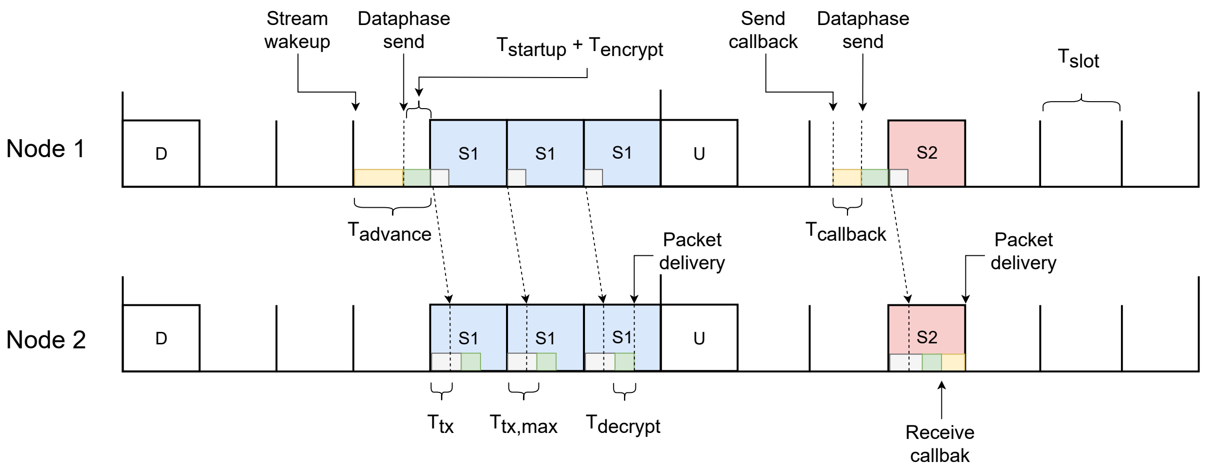

Figure 5.23: Example that includes radio startup and crypto timings. Stream *S1* uses the *write/wait* API while stream *S2* uses the callbacks one.

5.6. Schedule Expansion

As described in the TDMH overview chapter (Sec. 3.5.1), schedules are distributed over the network in an implicit form. Whenever the implicit schedule is received by a node, it has to perform some operations in order to make the schedule usable by the data phase and the session layer:

- *Streams rekeying*: every time a new schedule is received, and then activated, all the cryptography keys associated with any existing stream have to be updated.

If the number of opened streams is high, this processing may require more than a downlink slot to be completed. For this reason, the master node computes the number of downlink slots needed for the rekeying process and selects the tile at which the new schedule will be activated accordingly, allocating the required slots. This way, the rekeying process can be split over multiple downlink slots.

- *Schedule expansion*: it is the process of transforming the received implicit schedule into an explicit one, that will be used by the data phase. Again, this operation may require multiple downlink slots to be completed if the scheduled streams number is very high, especially now that during this process the streams wake-up lists used by the *StreamWaitScheduler* are built too. This section discusses the updates made to split the schedule expansion over multiple downlink slots and to build the stream wake-up lists.

Fig. 5.24 shows an example in which one downlink slot after the schedule reception is needed for rekeying and other two slots are dedicated to the schedule expansion. It exemplifies the desired behavior.

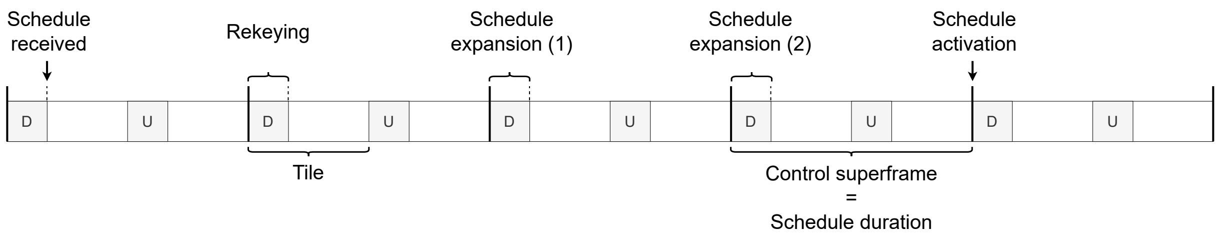

Figure 5.24: Example in which the schedule expansion is executed across two downlink slots.

5.6.1. Schedule Expansion Module

A new module, inside the downlink phase one, was implemented in order to manage the schedule expansion over multiple downlink slots.

In particular the *ScheduleExpander* class was created. It exposes a set of methods through which the expansion process can be started or an advance in the process can be triggered.

startExpansion The *startExpansion* method, as its name suggests, is called by the schedule distribution when the expansion process needs to start. During this phase, a few simple operations are performed:

- The information contained in the new *schedule header*, such as the number of schedule slots, its duration and activation tile (and relative time), are stored.
- The number of streams and downlink slots contained in the schedule is computed, in order to preallocate vectors for the explicit schedule and for the streams wake-up lists.
- All the required data structures are allocated (or cleared, if the process was already executed in the past).

needToContinueExpansion This method returns a boolean value indicating whether the expansion process has been completed or not. After starting the process then, at each downlink slot, this method is used to check if some streams still need to be expanded or not. This method returns true until the expansion process has not reached the end of the implicit schedule.

continueExpansion In case some streams still need to be expanded, the *continueExpansion* method is used to proceed into the process. Here is where the real expansion takes place and the stream wake-up lists are built. At each call, a number of iterations equal to the maximum number of expansions per downlink slot is performed, until the implicit schedule end is reached. At this point, the process terminates.

getExplicitSchedule This last method is simply used to retrieve the explicit schedule after the expansion process has been completed.

5.6.2. Schedule Distribution

The schedule distribution module is where the control of this entire phase is implemented. It relies on two different state machines, one running on the master node and the other one on all the dynamic nodes. Both the master and the dynamic state machines already provide a state that is dedicated to the rekeying of all the existing streams. The same state will also handle the schedule expansion process. In the current implementation this process is executed when the new schedule activation tile is reached. Since it has to be possible to execute the expansion over multiple downlink slots, the master node also considers the required number of slots by the schedule expansion process. In particular, when computing the activation tile of the new schedule, the master sums the required slots both for the rekeying and for the expansion. The total amount is used as the minimum number of tiles after which the new schedule can be activated. Due to the requirement of the *write/wait* API of receiving the new schedule at least one tile before the activation, this number is also incremented by one. The rekeying state of the two finite state machines is then renamed as *processing state*, in which both the rekeying and the expansion processes take place.

Master Node

When entering the *processing state*, the master node already knows both the number of downlink slots needed by the rekeying and those needed by the schedule expansion (according to the maximum number of streams that can be expanded per slot). Then, it executes the rekeying for a number of slots equal to *#rekeyingSlots*. When the *#rekeyingSlots* are reached, the expansion is started, through the *startExpansion* method. The expansion proceeds during the following downlink slots (*continueExpansion*) until the overall required number of slots is reached, that is:

$$\#rekeyingSlots + \#expansionSlots$$

At this point the expansion is complete and the master node only has to wait until the new schedule activation.
The master node behavior is shown in Fig. 5.25, while Fig. 5.26 reports the slightly modified state machine.

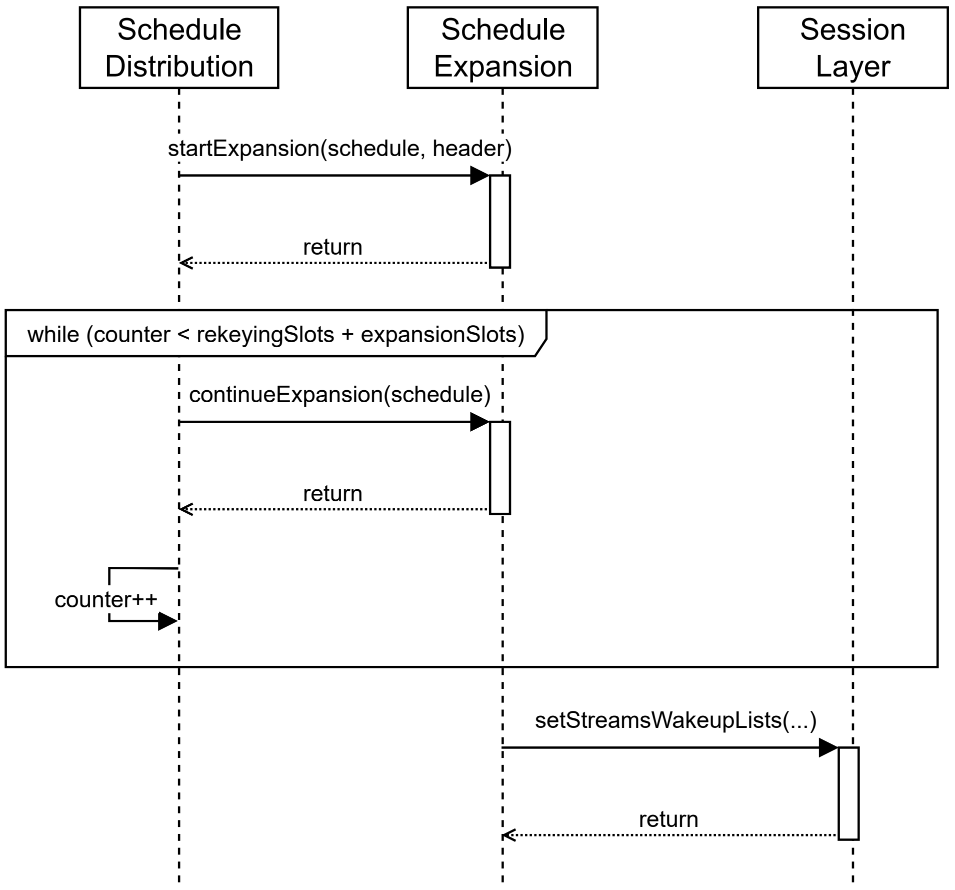
Schedule Distribution
Schedule Expansion
Session Layer
startExpansion(schedule, header)
return
while (counter < rekeyingSlots + expansionSlots)
continueExpansion(schedule)
return
counter++
setStreamsWakeupLists(...)
return

Figure 5.25: Sequence diagram showing the schedule expansion mechanism on the master node, after the rekeying already took place (#*rekeyingSlots* already elapsed).

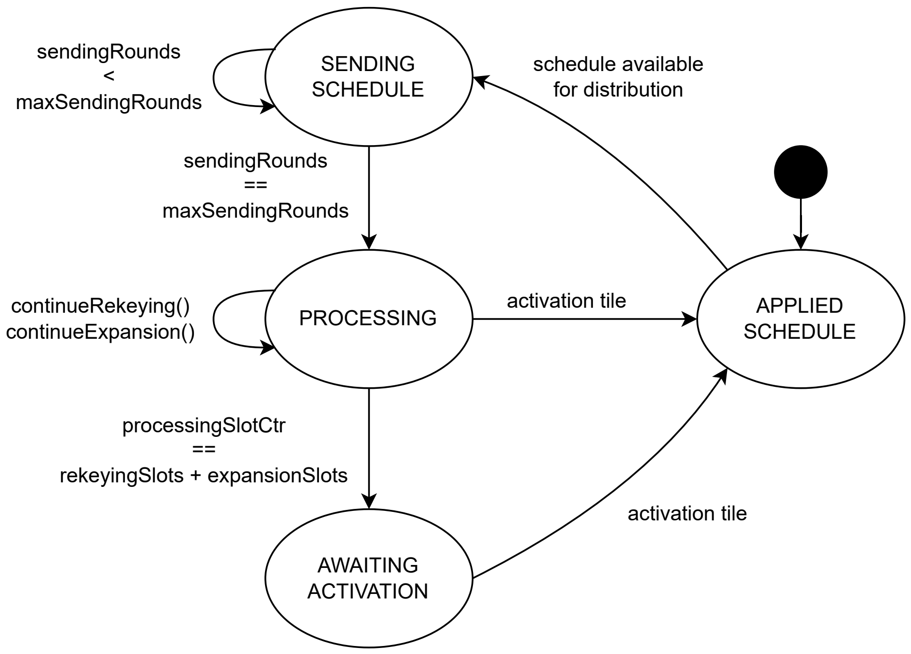

Figure 5.26: Master schedule distribution finite state machine.

Dynamic Nodes

Dynamic nodes do not directly know the number of needed slots for rekeying and schedule expansion. What they can do is to continue the rekeying process until the *needToContinueRekeying* method of the *StreamManager* returns a true value. When rekeying is done, the expansion process can be started and continued during the following downlink slots until, symmetrically, the *needToContinueExpansion* method of the *ScheduleExpander* class returns true. Indeed, the *continueExpansion* method will proceed, at each call, by a number of iterations equal to the maximum expansions per slot, so it will always be synchronized with the number of slots computed by the master.
Fig. 5.25 and Fig. 5.26 show the dynamic nodes behavior and their updated state machine.

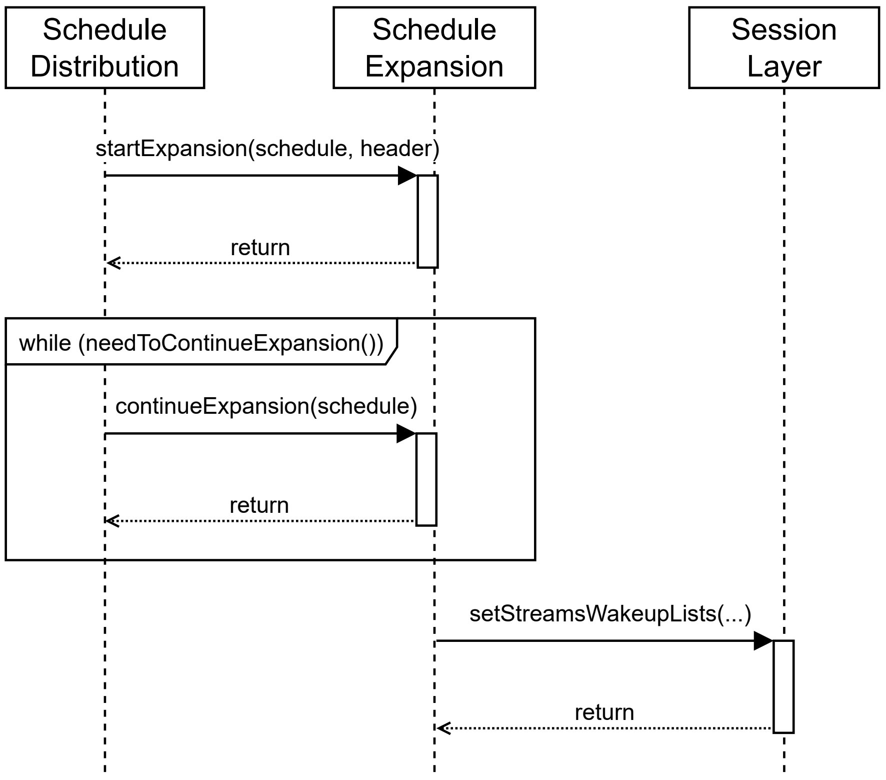
Schedule Distribution
Schedule Expansion
Session Layer
startExpansion(schedule, header)
return
while (needToContinueExpansion())
continueExpansion(schedule)
return
setStreamsWakeupLists(...)
return

Figure 5.27: Sequence diagram showing the schedule expansion mechanism on the dynamic nodes, after the rekeying already took place (*#rekeyingSlots* already elapsed).

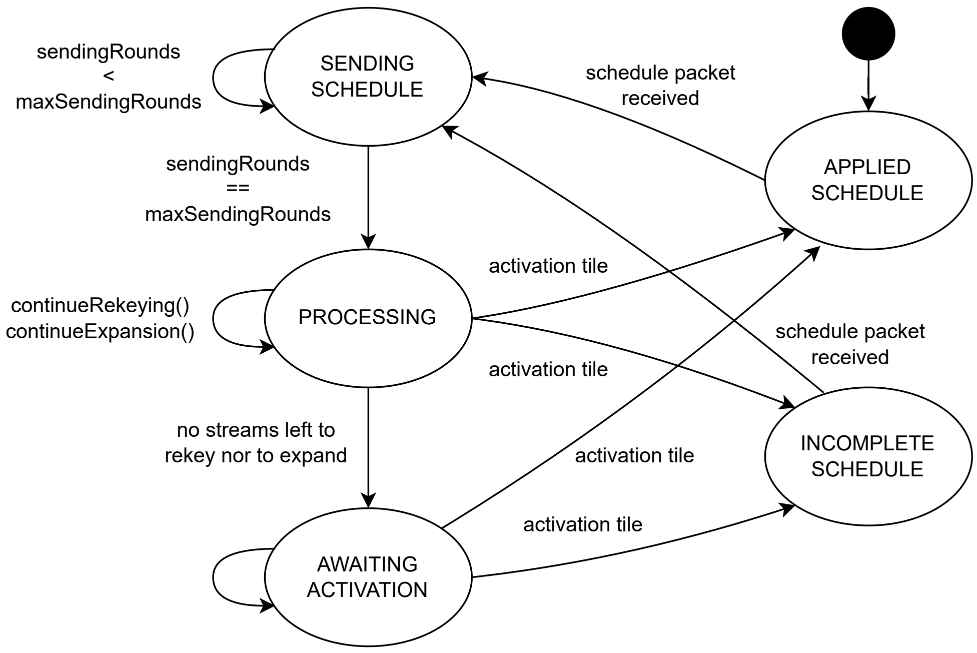

Figure 5.28: Dynamic schedule distribution finite state machine.

5.6.3. Re-Transmitted Schedules

It may happen that an incomplete schedule is received by a dynamic node if one or more of the schedule packets are repeatedly missed. In fact, these packets are redundantly distributed as well. The dynamic node then will issue to the master a schedule re-transmission request, by sending a *StreamManagementElement*. The dynamic node then activates an empty schedule and all the local streams are removed. When the new schedule is finally correctly received, the expansion has to take place.
It may occur that when the re-transmitted schedule is received by the mentioned node, the schedule activation tile has already passed. Even in this situation, the schedule expansion is executed and, as soon as it ends, the received schedule is activated.

5.6.4. Streams Wake-Up Lists

During the schedule expansion process, all the information needed to build the streams wake-up lists is already available: schedule activation tile, *StreamId*, period, offset inside the schedule and the action associated with each stream and slot. It is the perfect phase in which also building the lists needed to manage the streams that use the *write/wait* API. As said, it is necessary that the process can be split over different downlink slots.

An advantage of building these wake-up lists during the expansion process is that they can be **pre-allocated**. In fact, the number of elements to be put in the two lists can be computed a-priori on the implicit schedule, before starting the expansion, including the number of downlinks too. The number of downlink slots added to the list is the number of downlinks per control superframe multiplied by the number of control superframes in a schedule (i.e. in a data superframe):

$$\#downlinksInControlSuperframe \cdot \#controlSuperframesInSchedule$$

A first implementation consisted in a simple iteration over the implicit schedule and, while building the explicit schedule whenever a transmission stream was found it was added to the wake-up list, ignoring redundant transmissions. Since different streams can have different wake-up advances, the insertion into the list was not guaranteed to be ordered according to the streams wake-up time. At the end of the process then a list sort was required. Even though the loop could be split by performing a maximum number of iterations per downlink slot, the final sort could not.

In order to completely split the process, both the expansion and the construction of the wake-up lists have to be performed inside the same loop. This means that stream wake-up information has to be added to the list in an ordered manner and, thus, some elements shifting may be needed. As a consequence, the time needed in the worst-case scenario has to be measured in order to compute the maximum number of stream expansions (i.e. loop iterations) that can be executed inside a single downlink slot. The worst-case scenario is represented by a stream whose action is `SENDSTREAM` (because it has to be added to the wake-up list), when it is the last being added to the list and that has the lowest wake-up time among all the streams (since it would require shifting all the already inserted list elements). Moreover, if at that iteration a downlink slot end time has to be added to the list too, the required time for a single expansion increases again.
So, even though inserting elements in an ordered manner inside a fixed-size array is not $O(1)$, building the lists in such a way allows minimizing the usage of dynamic allocation.

6 | Latency Computation

This chapter shows how to compute the expected lower and upper bounds of streams latency under different conditions for both the proposed APIs, also when using them heterogeneously in the same network.
Packets propagation delay is ignored since nodes are always enough close to each other.

6.1. Minimum Theoretical Latency

The minimum possible end-to-end latency is achieved in an ideal and optimal scenario in which:

- The application transmission request (*write* primitive or send callback) takes place in correspondence of the transmission slot.
- The application computation time to generate a new packet is null.
- Copying data to or from the session layer takes zero time.
- The redundant transmissions (or receptions) of a stream are scheduled inside consecutive slots.
- Radio startup time is null and crypto, if used, also takes zero time.

In the described scenario, the end-to-end latency, from data generation to data availability after its reception, would be:

$$L = (R - 1) \cdot T_{slot} + T_{tx,max}$$

where T_{slot} is the duration of a single data slot, R is the stream's redundancy and $T_{tx,max}$ is the time needed by the radio transceiver to transmit a packet of maximum size. In an ideal scenario, this is the lowest achievable latency, so it represents a lower bound when either one API or the other is used.

Fig. 6.1 shows the ideal scenario described above. In the example two nodes exists, *Node*

1, which transmits data, and *Node 2* that only receives packets. Also, two streams exist, *S1* that has redundancy equal to three and *S2* that does not have any redundancy (i.e. packets on this stream are sent only once). Consider stream *S1* first: a packet sent over *S1* needs two entire data slots, in which the first two repetitions of the packet are sent, plus $T_{tx,max}$. During the third slot after starting the transmission, only a time equal to $T_{tx,max}$ is needed before the data becomes available to the application for processing. Then, the end-to-end latency of stream *S1* is equal to:

$$L_1 = 2 \cdot T_{slot} + T_{tx,max}$$

Transmitting a packet through *S2* instead only requires:

$$L_2 = T_{tx,max}$$

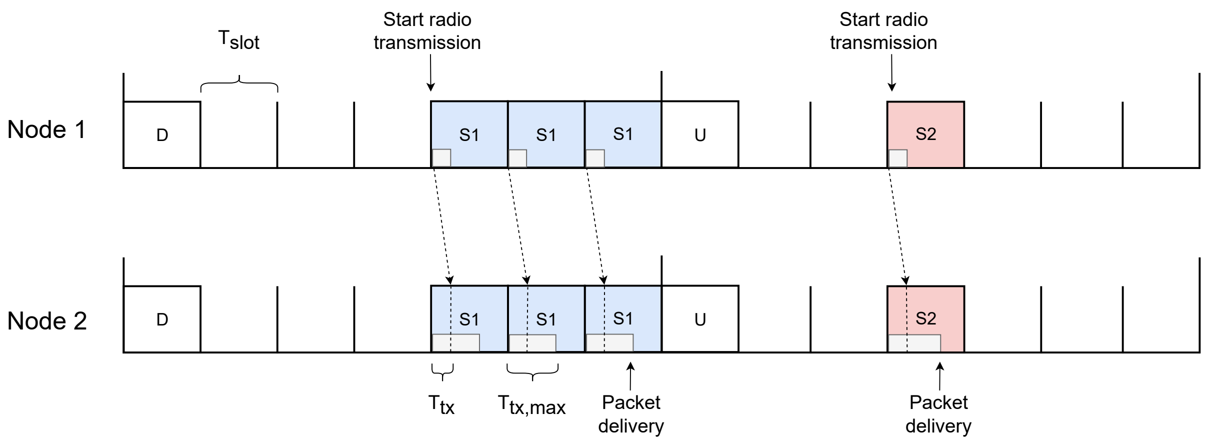

Figure 6.1: End-to-end latency lower bound example.

Of course, scenarios in which the stream's assigned slots are not consecutive inside the schedule exist too. This may happen if intermediate slots are assigned to other streams or if packets need to traverse multiple hops before reaching the destination node (so they have to be forwarded from node to node). The latency, in this case, is higher than the previously computed one, since more slots elapse between the start of the transmission and the end of the reception.
Then, more in general, the end-to-end latency can be expressed as:

$$L = (n_{slots} - 1) \cdot T_{slot} + T_{tx,max}$$

where n_{slots} is the number of slots spanned by the stream for which the latency is being computed, including the intermediate ones that may be allocated to other streams. In other words, it is the number of slots that elapse from the first to the last transmission slot of the considered stream, extremes included. As a consequence, n_{slots} is always greater than or equal to the stream redundancy value.

In the example shown in Fig. 6.2 stream *S1* spans a number of slots equal to $n_{slots} = 4$ and its latency is increased by one data slot with respect to the previous example:

$$L_1 = 3 \cdot T_{slot} + T_{tx,max}$$

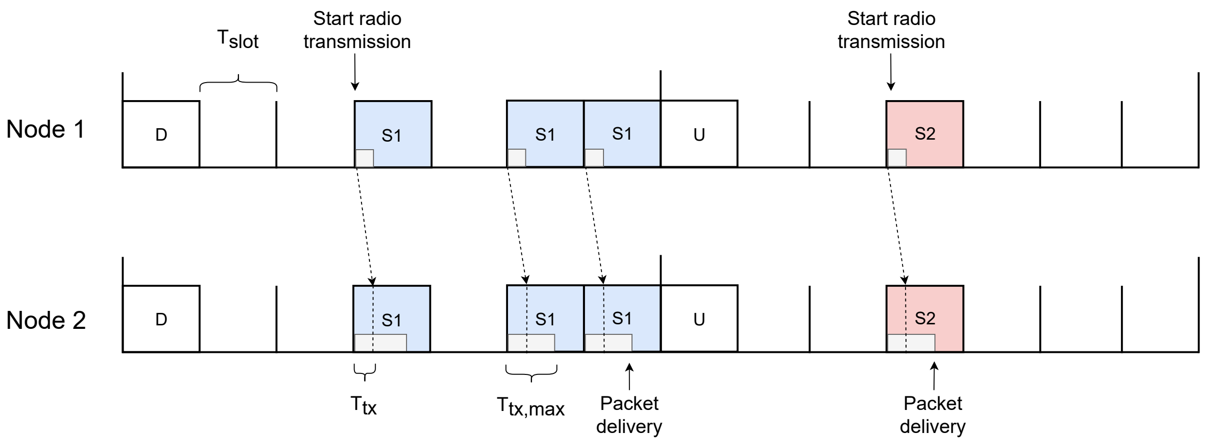

Figure 6.2: End-to-end latency lower bound example, in which stream *S1* transmission slots are not consecutive in the schedule.

6.1.1. Radio Startup Time and Crypto

Now consider the same assumptions as in Sec. 6.1, except for the fact that the physical layer takes some time before being able to transmit and the time needed to encrypt and decrypt packets is not null.

In this scenario, the radio startup time and the time needed by the cryptography execution concur to the end-to-end latency, regardless of the used API.
In case the redundant stream transmissions are schedule to consecutive slots, the latency lower bound can be computed as:

$$L = T_{startup} + (n_{slots} - 1) \cdot T_{slot} + T_{tx,max} + T_{crypto}$$

where $T_{crypto} = T_{encrypt} + T_{decrypt}$.

6.2. Callbacks API Latency

As a starting point, assume that both the radio startup and the cryptography execution take zero time. Assume also that on the transmission side the application computation takes zero time, while on the reception side it takes the entire specified callbacks execution time. The latency relative to this scenario represents an upper bound for the callbacks API latency, since the data generation takes place as soon as the send callback is called, while the data is made available as late as possible. It can be computed as:

$$L_{callback} = T_{sendCallback} + L + T_{recvCallback}$$

where L is the theoretical lower bound computed in Sec. 6.1 and $T_{sendCallback}$ and $T_{recvCallback}$ are the execution times of the two callbacks.
The above formula can be extended to:

$$L_{callback} = T_{sendCallback} + (n_{slots} - 1) \cdot T_{slot} + T_{tx,max} + T_{recvCallback}$$

where T_{slot} is the duration of a single data slot, where n_{slots} is the number of slots spanned by the stream and $T_{tx,max}$ is the time needed by the radio transceiver to transmit the required packet.

As previously explained, only one value can be specified for the callbacks execution time ($T_{callback}$), which is an upper bound of the real execution time among all the existing callbacks in any node and for any stream. Thus, the above formula simplifies to:

$$L_{callback} = (n_{slots} - 1) \cdot T_{slot} + T_{tx,max} + 2 \cdot T_{callback}$$

In the example shown in Fig. 6.3 it can be seen how the latency upper bound for stream *S1* is exactly:

$$L_{callback,1} = 2 \cdot T_{slot} + T_{tx,max} + 2 \cdot T_{callback}$$

Stream *S2* latency is instead:

$$L_{callback,2} = T_{tx,max} + 2 \cdot T_{callback}$$

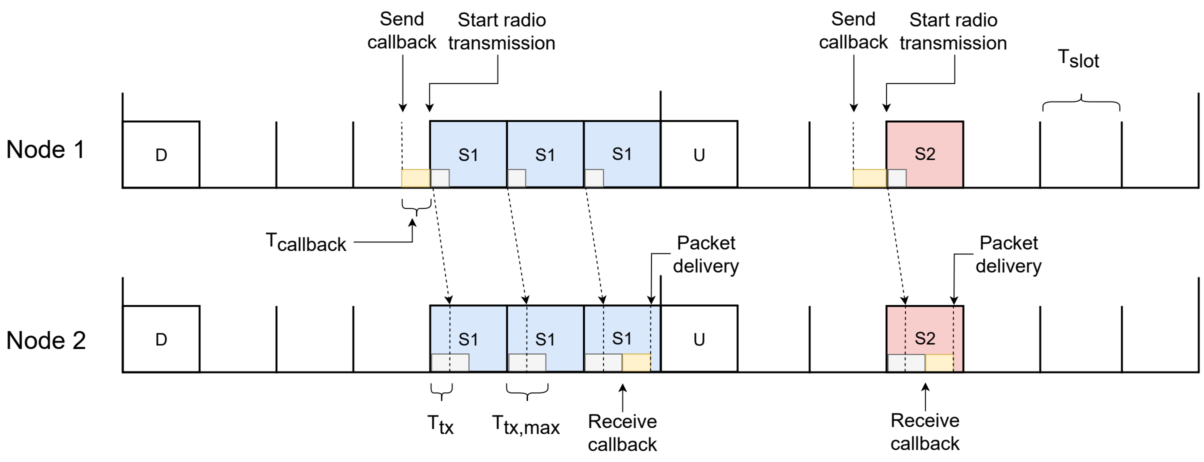

Figure 6.3: End-to-end latency upper bound example when using the callbacks API.

In a scenario in which stream's assigned slots are not consecutive, the maximum latency increases by a number of slots that is equal to the increase in the theoretical one. In the example shown in Fig. 6.4 stream *S1* latency increases by one slot:

$$L_1 = 3 \cdot T_{slot} + T_{tx,max} + 2 \cdot T_{callback}$$

On the contrary stream *S2* is not affected by any variation.

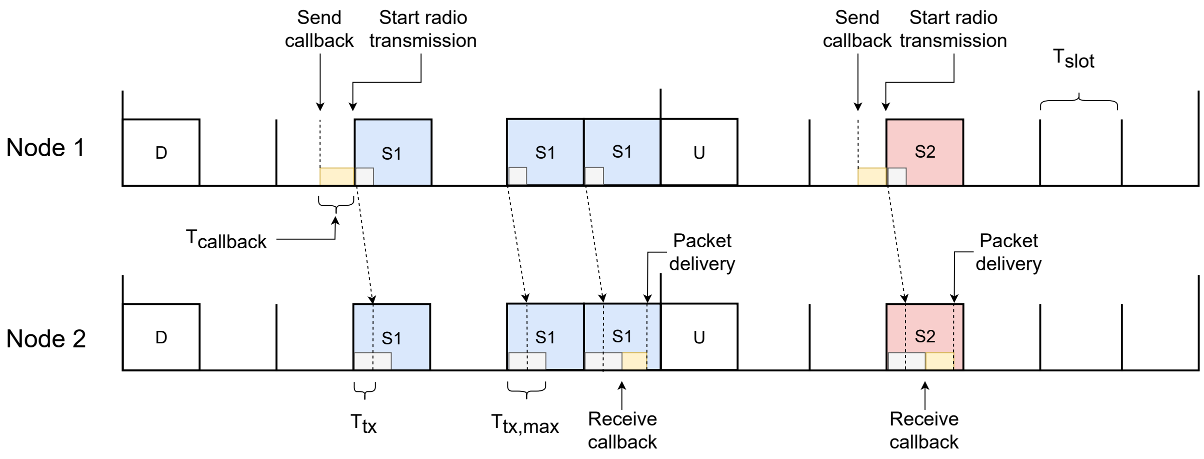

Figure 6.4: End-to-end latency upper bound example when using the callbacks API, in which stream *S1* transmission slots are not consecutive in the schedule.

In the real case, on the transmission side, the application may need some time to generate the new data, delaying it towards the transmission slot start. The receive callback function may also need less time to be completed than the user-provided execution time. As such, when receiving a packet, it would be delivered to the application before the time computed

for $L_{callback}$. The real latency increases or decreases according to the real execution time of the receive callback. It instead increases if the real execution time of the send callback decreases, while it decreases if the send callback execution time increases.
These factors give as a result that the real end-to-end latency L_{real} will be bounded between the lowest achievable latency L and the upper bound $L_{callback}$:

$$L < L_{real} \leq L_{callback}$$

Notice that in both the formulas for L and $L_{callback}$ the only variable factor is n_{slots}, which is also the only element that depends on the current streams schedule. Therefore, any change in the schedule would affect in the same way both the lower and upper latency bound (e.g. if the redundant stream transmissions are not consecutive). As a consequence, the bound difference is always constant:

$$L_{callback} - L = k$$

where k is a constant time amount and equal to:

$$k = 2 \cdot T_{callback}$$

This means that, excluding the latency absolute value changes across different schedules, the maximum theoretical jitter is fixed, since latency can fluctuate at most by a constant amount between its own bounds. The jitter depends neither on the streams period nor on the current streams schedule, but only on the callbacks execution time $T_{callback}$, which is a constant value. To summarize, the achieved latency is deterministic and predictable by only knowing the current schedule.

Considering an even more realistic situation, the data phase is executed accounting for the time needed by the radio to startup and to encrypt and decrypt a packet, as explained in Sec. 5.5.3. This amount of time ($T_{startup} + T_{crypto}$), has to be accounted for both the lower and the upper bounds. The difference between the two bounds is thus kept constant and is still equal to $2 \cdot T_{callback}$.

6.3. Write/Wait API Latency

As explained in Sec. 5.2, each stream can specify a wake-up advance, $T_{advance}$, that is a multiple of the duration of a TDMH data slot. So, the value of $T_{advance}$ directly affects the latency upper bound value.

Again, assume that, as a starting point, the radio is immediately ready for transmission and reception when requested and that the cryptography execution does not take any time. Assuming that the *write* operation is called as soon as the application is woken up from its waiting state (null processing time), starting from the theoretical formula, the latency in the current scenario can be derived as:

$$L_{write/wait} = T_{advance} + L$$

$$L_{write/wait} = T_{advance} + (n_{slots} - 1) \cdot T_{slot} + T_{tx,max}$$

Referring to Fig. 6.5 it can be seen that there exist two streams *S1* and *S2*, having respectively a wake-up advance time equal to:

$$T_{advance,1} = T_{slot}$$

$$T_{advance,2} = 2 \cdot T_{slot}$$

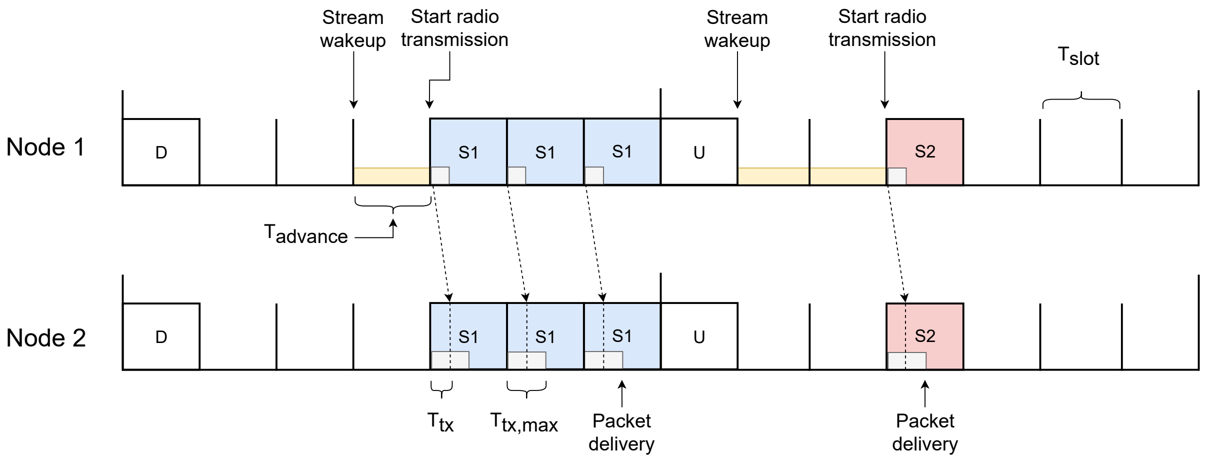

Figure 6.5: End-to-end latency upper bound example when using the *write/wait* API.

Then the latency upper bound for stream *S1* can be computed as:

$$L_{write/wait,1} = T_{advance,1} + 2 \cdot T_{slot} + T_{tx,max}$$

$$L_{write/wait,1} = 3 \cdot T_{slot} + T_{tx,max}$$

And for stream *S2*:

$$L_{write/wait,2} = T_{advance,2} + T_{tx,max}$$

$$L_{write/wait,2} = 2 \cdot T_{slot} + T_{tx,max}$$

In a scenario in which stream's assigned slots are not consecutive, like the one represented in Fig. 6.6, the overall latency may increase. Indeed, stream *S1* latency is increased by one slot, while stream *S2* is not affected by any change in this setting, since it has no redundancy (and its transmissions cannot be split over different slots).
The upper bound expression is then:

$$L_{write/wait,1} = T_{advance,1} + 3 \cdot T_{slot} + T_{tx,max}$$

$$L_{write/wait,1} = 4 \cdot T_{slot} + T_{tx,max}$$

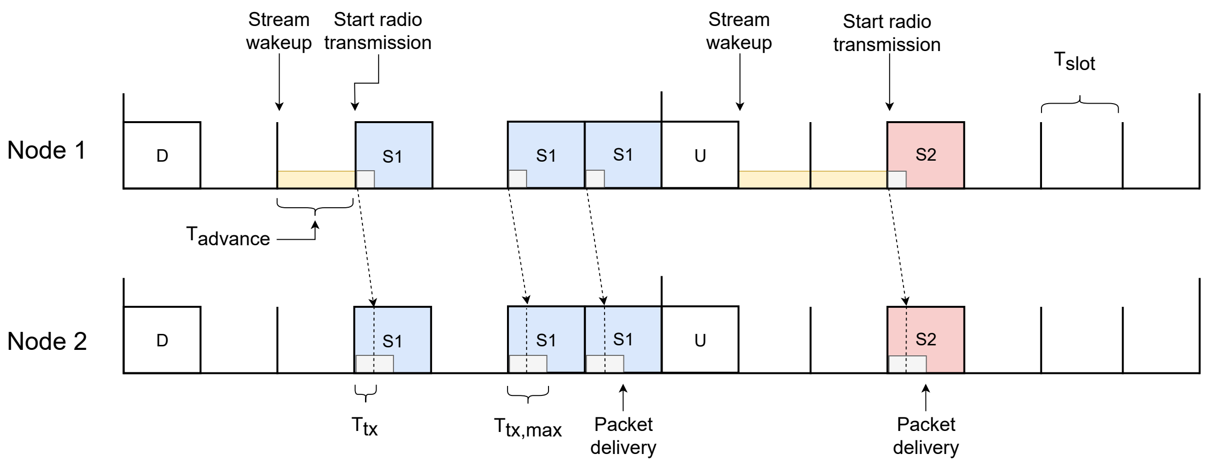

Figure 6.6: End-to-end latency upper bound example when using the *write/wait* API, in which stream *S1* transmission slots are not consecutive in the schedule.

The real latency L_{real} then is maximum if the *write* primitive is called exactly in correspondence of the stream's wake-up time (e.g. in an ideal scenario in which the application processing takes null time). In any real scenario in which the application is woken up and has to perform some computation, the *write* will be called sometime after the stream's wake-up time. In this situation, the end-to-end latency is lower than in a null processing time scenario. The latency $L_{write/wait}$ is then an upper bound of the real one:

$$L < L_{real} \leq L_{write/wait}$$

As for the callbacks scenario, since the only variable factor in both L and $L_{write/wait}$ formulas is n_{slots}, which affects both bounds by the same amount at any schedule change, it holds that:

$$L_{write/wait} - L = k$$

where k is a constant time amount and equal to:

$$k = T_{advance}$$

Due to the required radio startup and packet encryption time, the data phase execution takes place sometime before the real transmission slot start. This amount of time does not affect both the latency bounds, but only the lower one. In fact, the *write* primitive execution is not further anticipated as it happens with the send callback, so the upper bound remains unchanged. Anyway, the application has to produce the data and copy it to the session layer before the data phase checks if a new packet is ready, which takes place exactly $T_{startup} + T_{encrypt}$ before the transmission slot start time. The lower bound thus is increased by $T_{startup} + T_{encrypt}$. Furthermore, both bounds are incremented by the time needed for decrypting the received packets, $T_{decrypt}$.
As a consequence it holds that the two bounds difference is:

$$L_{write/wait} - L = T_{advance} - (T_{startup} + T_{encrypt})$$

Thus, in this case the bounds are stricter, since their difference is lower.

Again, a deterministic end-to-end latency is achieved, since its absolute value only depends on the current schedule and the specified wake-up time advance for the stream in question, which is constant. As for the callbacks case, the difference between the two latency bounds (i.e. maximum jitter) for a specific stream does not depend on the current schedule either, but only on the specified API parameter $T_{advance}$ (an possibly on the data phase time advance) and so it is constant.

6.4. Heterogeneous APIs Latency

As introduced, the two APIs are interoperable. If they are used together on the two sides of the same stream, the only difference with respect to the already presented end-to-end latency computation is that the upper bound will be a combination of the formulas shown in the two previous sections.

For simplicity assume that cryptography is not used and the radio startup time is null. As an example, assume to have two nodes: the first one uses the *write/wait* API to send packets, while the second one uses a receive callback. Looking at the example in Fig. 6.7, it can be seen that stream *S1* has a wake-up advance equal to one data slot: $T_{advance} = T_{slot}$. Moreover, after receiving a packet, the receive callback has to be executed ($T_{callback}$). Then, upper bound for *S1* is:

$$L_1 = T_{advance} + 2 \cdot T_{slot} + T_{tx,max} + T_{callback}$$

$$L_1 = 3 \cdot T_{slot} + T_{tx,max} + T_{callback}$$

Stream *S2* instead has a wake-up advance time equal to $T_{advance} = 2 \cdot T_{slot}$. Its end-to-end latency upper bound is then equal to:

$$L_2 = T_{advance} + T_{tx,max} + T_{callback}$$

$$L_2 = 2 \cdot T_{slot} + T_{tx,max} + T_{callback}$$

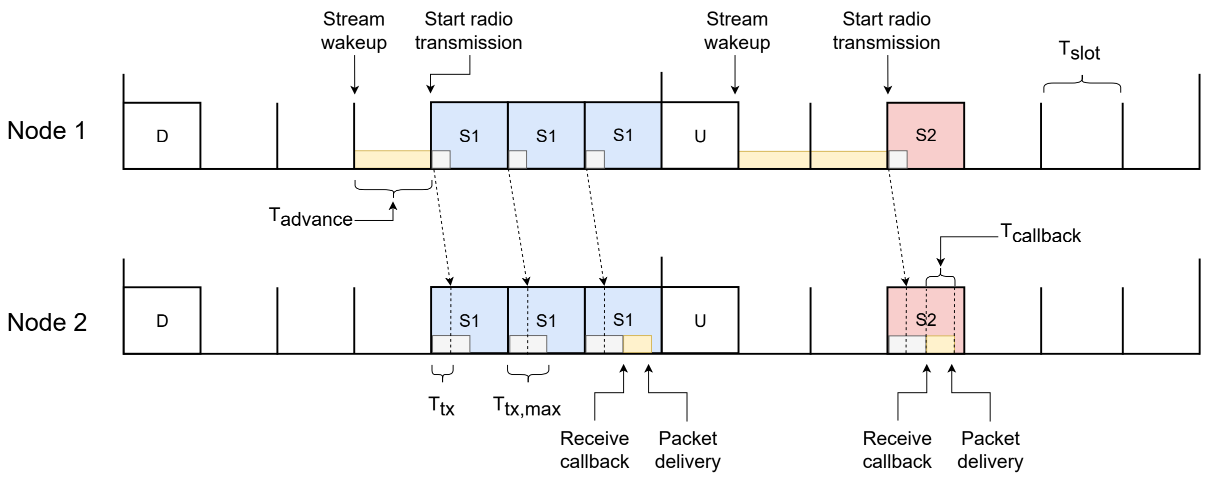

Figure 6.7: End-to-end latency upper bound example when using heterogeneous APIs (*write/wait* for transmission and callback for reception).

Symmetrically, the sender can use the callbacks API while the receiver can use the *read* primitive, which is a blocking call that returns as soon as all the redundant packets have been received. Considering stream *S1* in Fig. 6.8, its end-to-end latency upper bound is simply:

$$L_1 = T_{callback} + 2 \cdot T_{slot} + T_{tx,max}$$

While for stream *S2* it is is computed as:

$$L_2 = T_{callback} + T_{tx,max}$$

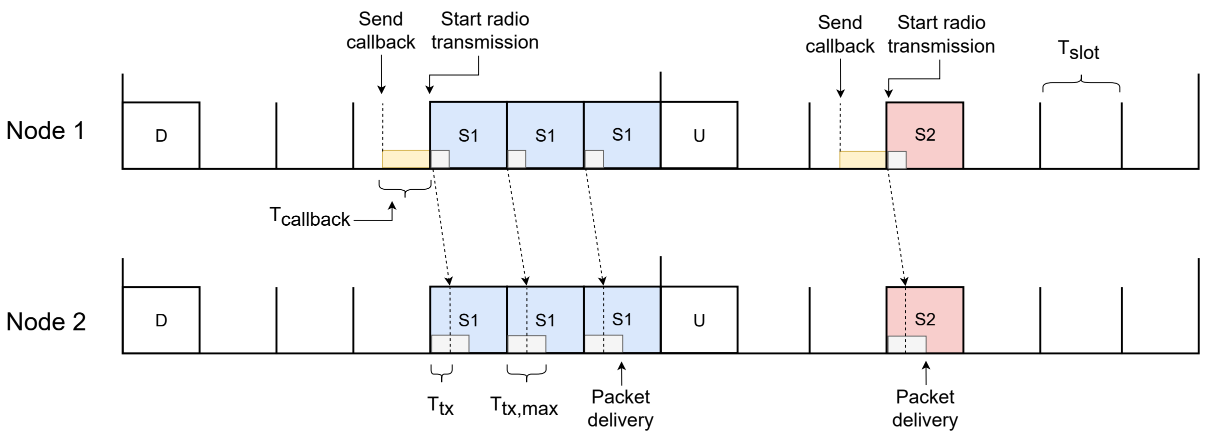

Figure 6.8: End-to-end latency upper bound example when using heterogeneous APIs (callback for transmission and *read* for reception).

6.5. Real Examples

In this section, some examples in which the latency bounds are computed through the previously-described formulas are presented.

Small experiments were conducted to check that the latency bounds are respected, using the two APIs separately and also using them to handle transmissions and receptions on the same stream. The setup includes two nodes, a master and a dynamic one. The dynamic node opens a stream to the master and sends packets with a period of 10 tiles ($P = 1000\ ms$). The stream redundancy is set to triple.

When using the *write/wait* API the stream wake-up advance is a single slot: $T_{advance} = T_{slot} = 6\ ms$. When using the callbacks API, the callbacks execution time is instead set to 0.5 ms, which leads to TDMH slots to be as long as $T_{slot} = 7\ ms$. The physical radio startup time is $T_{startup} = 0.5\ ms$ and the time allocated to cryptography execution is $T_{crypto} = T_{encrypt} + T_{decrypt} = 0.11\ ms + 0.12\ ms = 0.23\ ms$. Finally, the time needed to transmit a packet of maximum size is $T_{tx,max} = 4.448\ ms$.

Consider that in these examples almost no computation is performed for generating data packets (they only contain an incremental counter). For this reason, the measured latency is almost exactly in the middle of the lower and the upper bounds for the callbacks API, since the bounds difference is $2 \cdot T_{callback}$ and the entire callback execution time on the sender side affects the latency, while the receiving one is almost immediate. For the *write/wait* API instead, it is unbalanced towards the upper one, since packets generation takes place almost immediately after the stream is woken up.

In all these small experiments the average latency falls in between the lower and upper bounds and the maximum measured jitter never exceeds the bounds difference, as explained in the previous sections.

6.5.1. Callbacks API

The first example uses the callbacks API both for the transmission and for the reception of application data packets.

Fig. 6.9 reports the measured latency among the two nodes and Fig. 6.10 shows the distribution of packets latency. Latency bounds and measured statistics are then reported in Tab. 6.1 and 6.2.
It can be seen how the upper and lower bounds difference is exactly 1 ms, which corresponds to $2 \cdot T_{callback}$.

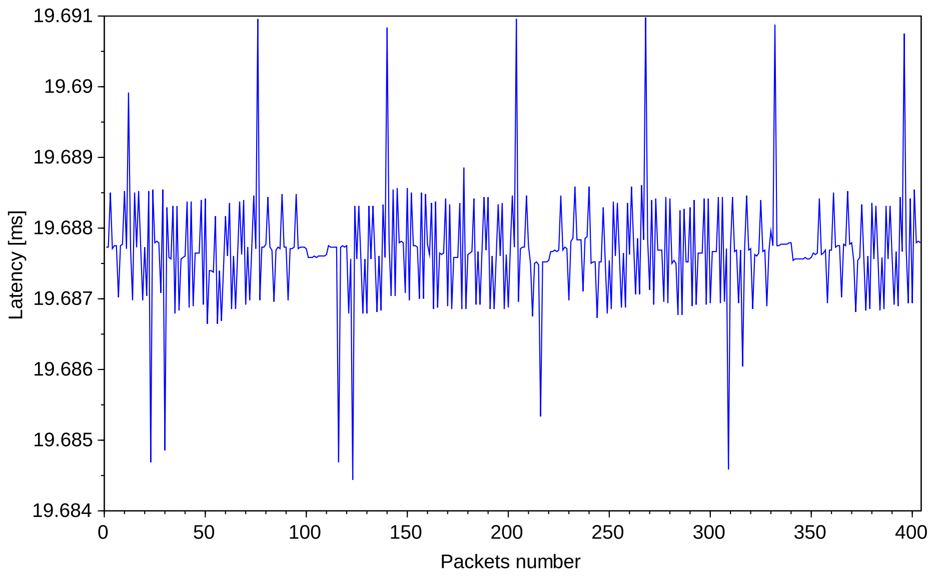

Figure 6.9: Single hop packets measured latency using the callbacks API.

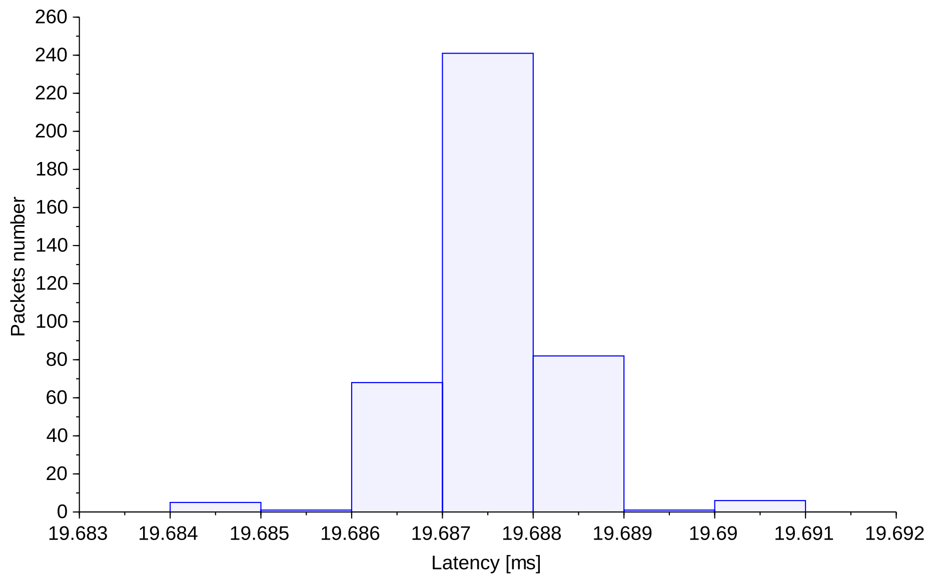

Figure 6.10: Single hop packets latency distribution using the callbacks API.

Packets num.	Average latency	Jitter	Standard deviation
400	19.6877 ms	3.31 μs	0.74 μs

Table 6.1: Single hop packets measured latency statistics using the callbacks API.

Lower bound	Upper bound
19.178 ms	20.178 ms

Table 6.2: Single hop packets latency bounds using the callbacks API.

6.5.2. Write/Wait API

This example, instead, uses the *write/wait* API for the transmission and the *read* primitive for packets reception.

Fig. 6.11 reports the measured latency among the two nodes and Fig. 6.12 shows the distribution of packets latency. Latency bounds and measured statistics are then reported in Tab. 6.3 and 6.4.

It can be seen how the upper and lower bounds difference is exactly 5.39 ms, which corresponds to $T_{advance} - (T_{startup} + T_{encrypt})$.

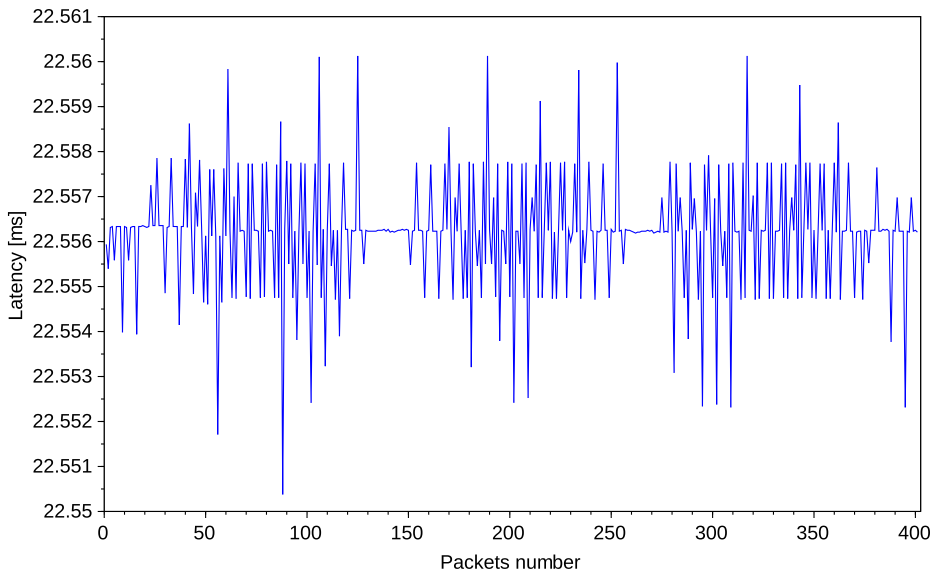

Figure 6.11: Single hop packets measured latency using the *write/wait* API.

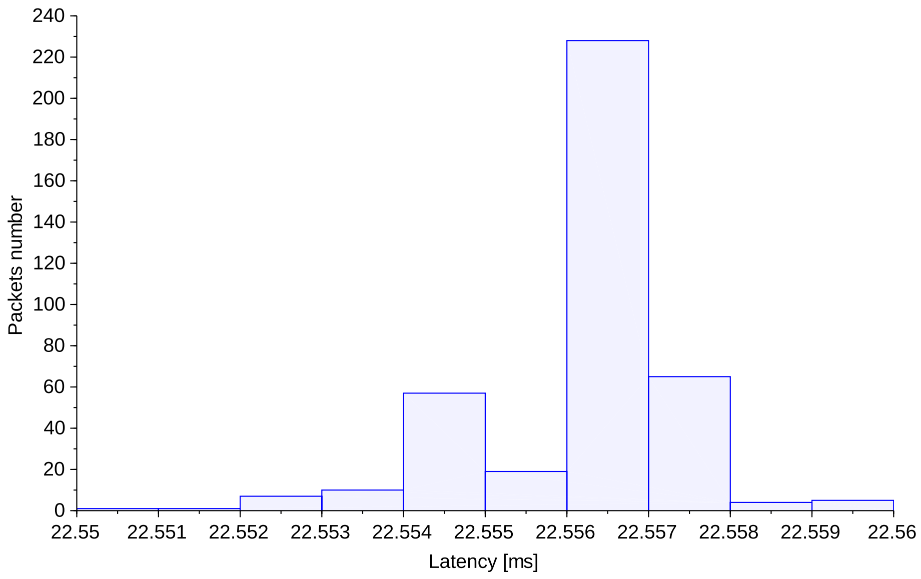

Figure 6.12: Single hop packets latency distribution using the *write/wait* API.

Packets num.	Average latency	Jitter	Standard deviation
400	22.5562 ms	5.83 μs	1.28 μs

Table 6.3: Single hop packets measured latency statistics using the *write/wait* API.

Lower bound	Upper bound
17.178 ms	22.568 ms

Table 6.4: Single hop packets latency bounds using the *write/wait* API.

6.5.3. Heterogeneous APIs

The third example uses the *write/wait* API for transmission but the packet reception is handled through a callback, so also in this case slots duration is 7 ms, rather than 6 ms.

Fig. 6.13 reports the measured latency among the two nodes and Fig. 6.14 shows the distribution of packets latency. Latency bounds and measured statistics are then reported in Tab. 6.5 and 6.6.
In this case the upper and lower bounds difference is 6.89 ms.

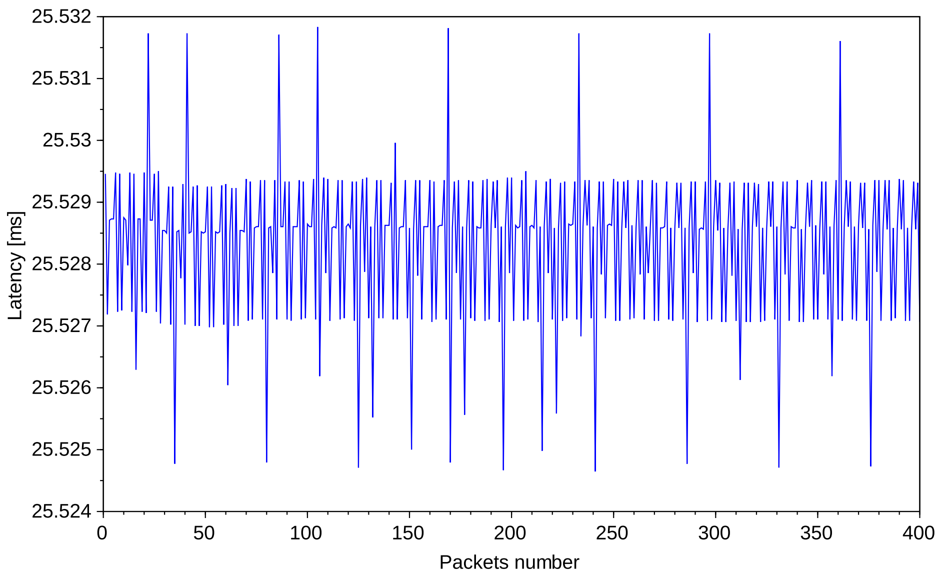

Figure 6.13: Single hop packets measured latency using heterogeneous APIs.

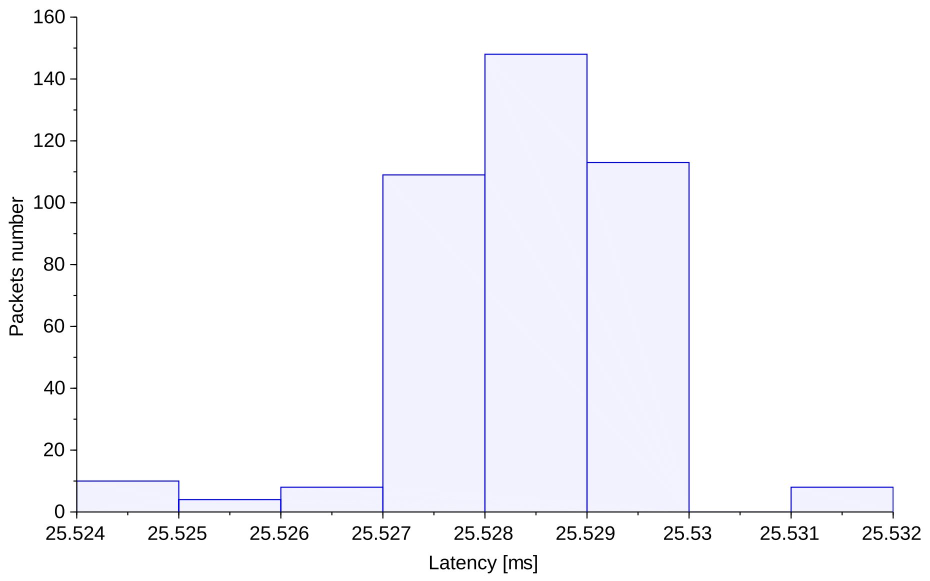

Figure 6.14: Single hop packets latency distribution using heterogeneous APIs.

Packets num.	Average latency	Jitter	Standard deviation
400	25.5283 ms	3.68 μs	1.18μs

Table 6.5: Single hop packets measured latency statistics using heterogeneous APIs.

The latency lower and upper bound are reported in Tab. 6.6.

Lower bound	Upper bound
19.178 ms	26.068 ms

Table 6.6: Single hop packets latency bounds using heterogeneous APIs.

6.6. Re-Scheduling Latency

The latency bounds computed in the examples shown in Sec. 6.5 are relative to a single schedule, they are valid as long as a schedule is active. They are not useful around the exact point in which a new schedule is activated and the previous one becomes invalid. Latency has to be separately evaluated for the two distinct schedules.

Fig. 6.15 shows the latency trend of a stream with triple redundancy and that uses the *write/wait* API. The stream S is always scheduled on the first hop, but it is assigned to different transmission slots, according to the active schedule. During some time its assigned slots are interleaved by an uplink slot, whose duration is $T_{uplink} = 2 \cdot T_{slot}$. Stream's wake-up advance is instead set to $T_{advance} = T_{slot} = 6\ ms$. Again, the time needed to transmit a packet of maximum size is $T_{tx,max} = 4.448\ ms$.

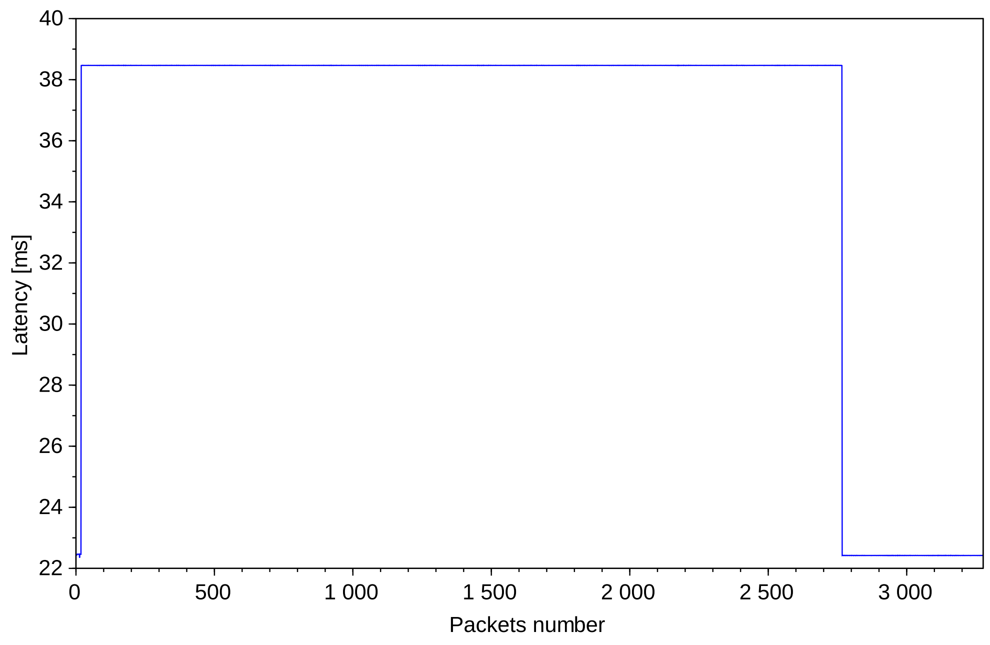

Figure 6.15: Single hop packets latency for stream S when re-schedulings take place.

When the stream is scheduled with three consecutive redundant transmissions (Tab. 6.7), its latency upper bound amounts to:

$$L_{upper,1} = T_{advance} + (n_{slots} - 1) \cdot T_{slot} + T_{tx,max} + T_{decrypt}$$

Since $T_{advance} = T_{slot}$ it follows that:

$$L_{upper,1} = 3 \cdot T_{slot} + T_{tx,max} + T_{decrypt} = 22.568\ ms$$

The average measure value in this scenario was instead $22.547\ ms$.

Notice that since the *write/wait* API is used, only the packets decryption time is considered in the computation of the latency upper bound, while the encryption and radio startup time do not have any effect, as explained in Sec. 6.3.

Schedule 1	
t_0	*Downlink*
t_1	*Downlink*
t_2	Stream S
t_3	Stream S
t_4	Stream S
t_5	...

Table 6.7: Example of schedule in which stream S transmits during three consecutive slots.

Schedule 2	
t_0	*Downlink*
t_1	*Downlink*
t_2	...
t_{14}	Stream S
t_{15}	Stream S
t_{16}	*Uplink*
t_{17}	*Uplink*
t_{18}	Stream S
t_{19}	...

Table 6.8: Example of schedule in which stream S transmissions are interleaved by the uplink phase slots.

In the second scenario (Tab. 6.8) the three redundant transmissions are interleaved by the uplink phase, which occupies two data slots. The number of slots spanned by the stream then includes also the two slots allocated for the uplink phase and amounts to $n_{slots} = 5$. At the end of any tile, a slack time is also added, which amounts to $T_{slack} = 4\ ms$.

The latency upper bound is:

$$L_{upper,2} = T_{advance} + (n_{slots} - 1) \cdot T_{slot} + T_{slack} + T_{tx,max} + T_{decrypt}$$

$$L_{upper,2} = 5 \cdot T_{slot} + T_{slack} + T_{tx,max} + T_{decrypt} = 38.568\ ms$$

The average measure value in this scenario was instead 38.542 ms.

The same can happen if a stream is scheduled on a hop that is not the first one, so its transmission slots are interleaved by slots in which the same packets have to be forwarded through the following hops. The absolute end-to-end latency value indeed increases. In the example of Tab 6.9, packets relative to stream $1 \rightarrow 0$ follow the path $1 \rightarrow 2 \rightarrow 0$. Then, n_{slots} for stream $1 \rightarrow 0$ assume the value of 6 slots, from t_2 to t_7 included.

	Schedule 3
t_0	*Downlink*
t_1	*Downlink*
t_2	$1 \rightarrow 2$
t_3	$2 \rightarrow 0$
t_4	$1 \rightarrow 2$
t_5	$2 \rightarrow 0$
t_6	$1 \rightarrow 2$
t_7	$2 \rightarrow 0$
t_8	...

Table 6.9: Example of schedule in which stream $1 \rightarrow 0$ transmissions path to the receiver node is two-hops long and passes through node 2.

The presented example shows that only around a new schedule activation a high variation of the latency may be experienced, since the number of slots spanned by streams may change, but this phenomenon is limited to a single packet. During the plateau intervals between two schedules, latency is always bounded and all the consideration done in this chapter are valid.
It is important to underline that (re-)scheduling is intrinsic to the TDMA nature of the protocol. It is a necessary operation that cannot be avoided. Moreover, re-schedulings triggered by network topology changes are infrequent and, when new streams are opened, the scheduler tries to avoid overturning the already existing streams. In conclusion, re-schedulings only affect the latency absolute value, not its variability.

7 | Experiments

In this chapter different experiments are proposed, consisting of both simulations and real-world ones. The conducted experiments are divided into two categories. The first ones aim at evaluating streams reliability and latency, by validating the implemented functionalities and proving that they do not interfere with the overall network reliability provided by TDMH. Successively, the implementation is evaluated in a control loop scenario over a wireless network.

7.1. Experiments Setup

During all the experiments, the dynamic nodes open a stream to the master, using triple redundancy. The packet payload contains a counter that increases with each packet, allowing to evaluate packet loss by analyzing missing counter values from the master's logs.

7.1.1. WandStem Nodes

TDMH is designed to run on top of the *Miosix Operating System* [21] and, more specifically, on the *WandStem* nodes [20], which use the low-power EFM32GG microcontroller by *SiliconLabs*, based on an ARM Cortex-M3 core running at 48 MHz. This microcontroller has a 1 MB flash memory and a 128 KB RAM on-chip. These devices are equipped with a 2.4 GHz radio module and are capable of keeping track of time with a 21 ns resolution, which is a necessary condition to perform hardware timestamping, needed to make the FLOPSYNC-2 time synchronization algorithm work. WandStem nodes consumption can get as low as 2.4 μA during deep-sleep, which makes them suitable to be battery-powered devices.

7.1.2. Latency Profiling

When creating a new packet, sender nodes also insert in its payload the current network time timestamp. When the packet is received by the master, the first operation it performs is to take the reception timestamp and to log the elapsed time between creation and reception of the packet. Through master's logs, the history of each packet's latency can be reconstructed.
A script is then used to analyze the log and to produce a plot, using *Scilab* [7], for each sender stream latency, along with statistics for each of those streams (e.g. latency average, standard deviation, etc.).

Consider that the master node in this setting is heavily loaded. It has one open stream for each other node in the network and has to compute schedules and lead all the other mechanisms of the protocol.
Moreover, the high amount of debugging prints increases the load, so latency values can have higher oscillations. But those debugging prints are necessary to check that everything is correctly working in the network and to produce a complete log of the experiment. This is the setting that allows having the best possible observability of the system at the moment.

7.2. Simulations

A previous student doing his master thesis [18] on TDMH, Paolo Polidori, developed an interface to run TDMH in *OMNeT++* [16], a well known network simulator based on discrete event simulation. An interface compatible with Miosix was developed, to expose the radio module, timers and logging, that use the OMNeT++ primitives instead of those provided by the Miosix kernel. This design allows having a single code base that can be either executed on the real hardware or on the network simulator, simplifying testing and troubleshooting, which is usually a hard and time-consuming task on the real hardware, especially in a distributed setting.

Being able to execute TDMH in a simulated environment allows developing and testing the new components and features with the simplicity of deploying different topologies. This allows to also test all the possible corner cases, which are harder to be tested in a real wireless setting, due to the distributed nature of the system which makes it non fully observable and to the fact that sometimes they are hardly reproducible.

Setup

As for real-world experiments, also during simulations, all the dynamic nodes open a stream with the master as a destination. The exchanged data packets are also the same as described previously in Sec. 7.1.

Thanks to the simulated environment, different network topologies can be tested out with ease, trying to test all the possible corner cases, such as different stream schedules and multi-hop networks, which affect the required streams wake-up time.
Fig. 7.1 to Fig. 7.4 show some examples of the topologies used for testing.

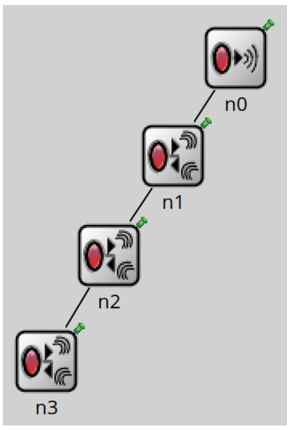

Figure 7.1: *Line4* network topology.

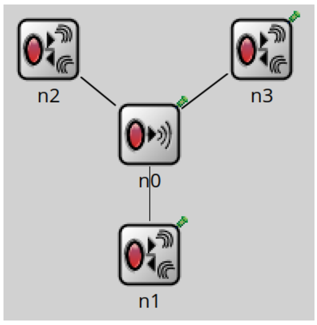

Figure 7.2: *Star4* network topology.

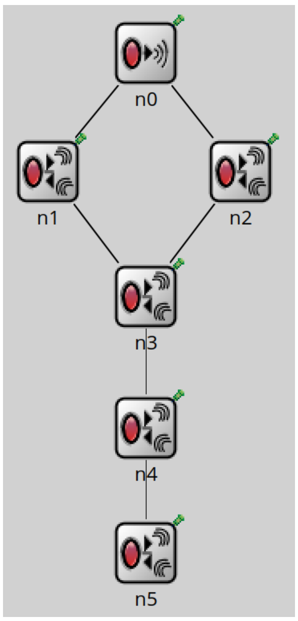

Figure 7.3: *Kite* network topology.

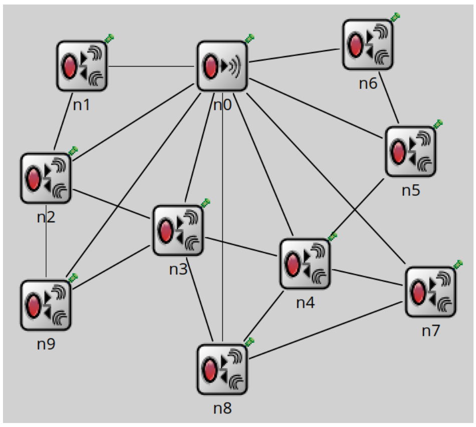

Figure 7.4: *PartialMesh* network topology.

Results

Simulations on the different network topologies were repeated until positive results were achieved.

Simulations results are considered positive when the network reliability is constant and equal to 100%, meaning that no misses occur. Indeed, during simulations, packets can be missed only if the implemented functionalities are not properly working (e.g. in some corner cases), since transmission is not affected by radio interference.

For what concerns latency instead, simulation experiments results are considered positive and accepted only if the latency standard deviation is equal to zero for each stream (when no re-scheduling takes place). Since during simulations the computation time is assumed to be zero (i.e. it is not considered for simulating the time flowing), packets latency must be constant.

7.3. Wireless Validation Experiments

Different wireless experiments were conducted in an incremental way: starting from smaller networks composed of three to seven nodes, in order to check that the thesis goal was actually achieved (that is showing that nodes latencies are stable over time). Other experiments, in which all the available WandStem nodes were deployed, were conducted on the first floor of the Building 21 of Politecnico di Milano, reproducing the node placement used in the paper presented by Terraneo et al. at the RTSS 2018 conference [24], to have a set of reference results. This method allows checking that the newly implemented features do not undermine the already existing protocol stack and network stability.

The complete setup is comprised of a master node (node 0) and 13 dynamic nodes, numbered from 1 to 13. All the streams connecting dynamic nodes to the master have a period of 10 tiles (i.e. 1 s).

Each experiment was repeated for both the described APIs (the callbacks and the write/wait one), each time leaving the test running for multiple hours. In particular for both the APIs two experiments are presented here, each conducted under different electromagnetic interference conditions.

Fig. 7.5 shows the positioning of all the nodes.

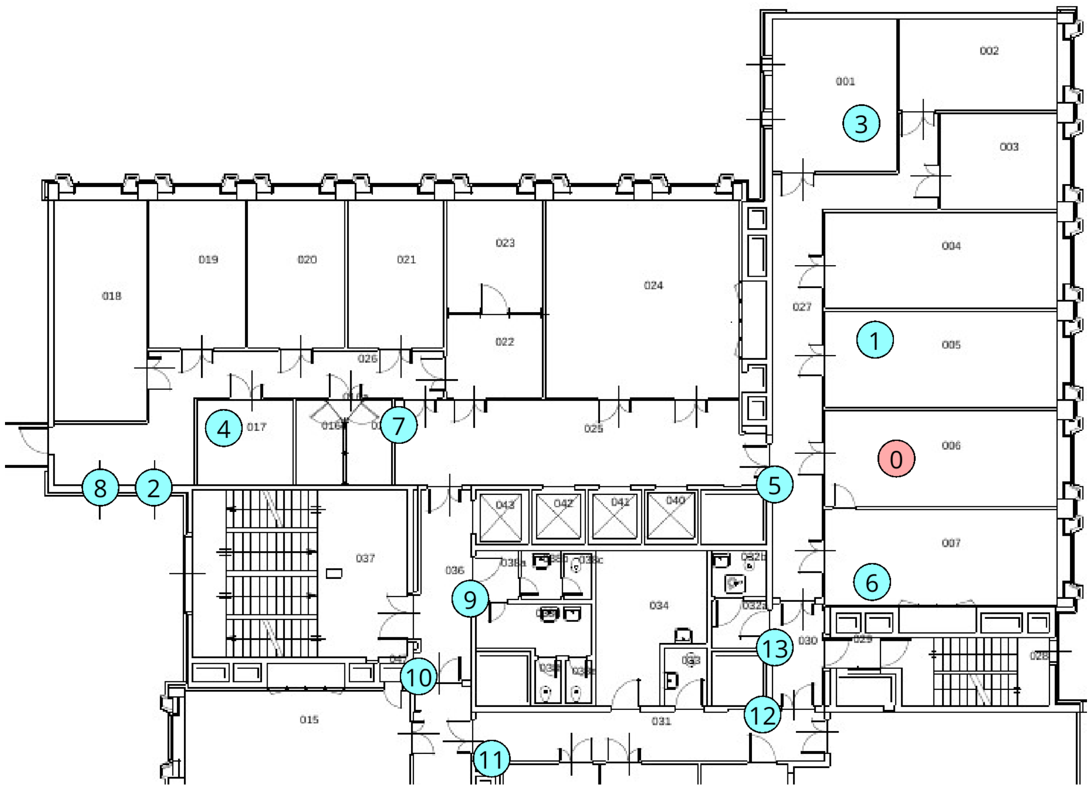

Figure 7.5: WandStem nodes placement at Building 21 of Politecnico di Milano. The master node is highlighted by an orange circle.

7.3.1. Callbacks API

In the experiments in which the callbacks API is used, the send and receive callback functions only perform a *memcpy* operation to retrieve the received packet from the session layer or to pass it a new packet to be sent, containing an incremental counter and the current network timestamp. Considering that the master node is heavily loaded in the experiments setup, much more than all the other nodes, the callbacks execution time was set to 0.5 ms in order to leave a good margin. The specified callbacks execution time leads the slot duration to be equal to 7 ms.

High Interference Experiment

Setup The first complete experiment using all the 14 available WandStem nodes and the callbacks API was conducted for 4 hours during a working afternoon, in which the WiFi usage is intensive and has drawback on the achieved network reliability. Indeed, WandStem nodes radio module uses the 2.4 GHz band as the WiFi.

The complete resulting topology is represented in Fig. 7.6.

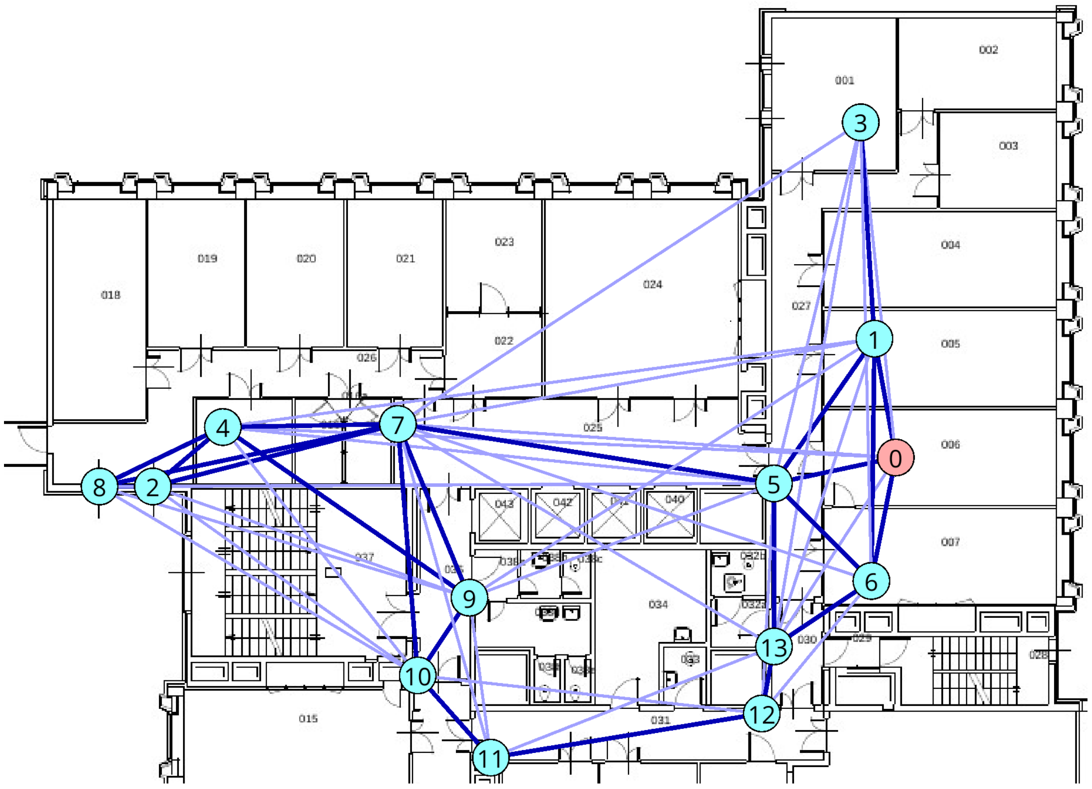

Figure 7.6: First callbacks API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones.

Results Every node sent around 14 thousand packets through the opened stream and, overall, the network showed a reliability of 99.88%, with all the streams being above 99.50%.

Detailed reliability results for each stream are reported in Tab. 7.1, including reliability values for single, double and triple streams redundancies.

Stream	**Sent packets**	**Reliability Single**	**Reliability Double**	**Reliability Triple**
1-0	13944	99.99%	100.00%	100.00%
2-0	13935	99.08%	99.80%	99.88%
3-0	13933	99.15%	99.99%	100.00%
4-0	13949	99.36%	99.86%	99.97%
5-0	13949	99.73%	100.00%	100.00%
6-0	13935	98.41%	99.33%	99.51%
7-0	13949	99.63%	99.95%	99.99%
8-0	13949	98.85%	99.94%	99.99%
9-0	13933	99.43%	99.82%	99.87%
10-0	13933	99.16%	99.94%	99.99%
11-0	13933	97.81%	99.49%	99.58%
12-0	13933	97.44%	99.38%	99.61%
13-0	13935	97.45%	99.39%	100.00%
Total	181210	98.88%	99.76%	99.88%

Table 7.1: Stream seliability during the first callbacks API experiment.

Plots in Fig. 7.7 to Fig. 7.19 show, for each node from 1 to 13, the latency of every packet received by the master. It can be seen that the plot trend is the same for each stream: higher latencies are experienced in correspondence with the execution of the scheduler in the master node. Still, latency never deviates more than 40 μs from the average of the relative stream, even while the scheduler is running and the master has a higher computational load. The maximum experienced jitter indeed is 40.03 μs for stream $4 \rightarrow 0$. Furthermore, the average latency standard deviation among all the streams is 1.47 μs, which is orders of magnitude lower than the measured average latency for each of them.

The mentioned resulting values are reported in Tab. 7.2.

Stream	Average latency	Max. Jitter	Standard deviation
1-0	19.5714 ms	34.91 μs	1.44 μs
2-0	198.5713 ms	29.87 μs	1.42 μs
3-0	77.5717 ms	33.72 μs	1.43 μs
4-0	61.5716 ms	40.03 μs	1.79 μs
5-0	19.5713 ms	31.30 μs	1.35 μs
6-0	47.5716 ms	31.45 μs	1.38 μs
7-0	47.5717 ms	37.24 μs	1.29 μs
8-0	98.5716 ms	33.28 μs	1.38 μs
9-0	135.5717 ms	27.61 μs	1.30 μs
10-0	68.5716 ms	32.98 μs	1.39 μs
11-0	619.5716 ms	32.18 μs	1.54 μs
12-0	70.5720 ms	36.74 μs	1.51 μs
13-0	340.5720 ms	36.22 μs	1.85 μs
Total	–	33.66 μs	1.47 μs

Table 7.2: Streams latency statistics during the first callbacks API experiment.

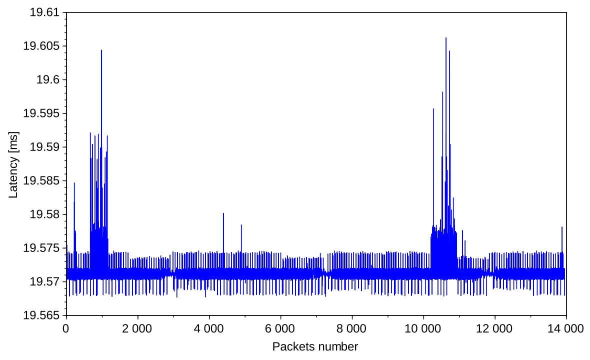

Figure 7.7: Latency for stream 1 → 0.

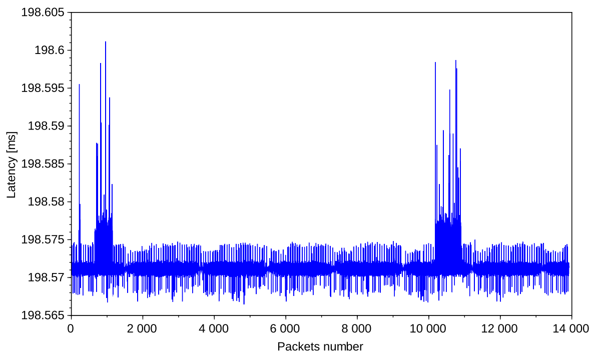

Figure 7.8: Latency for stream 2 → 0.

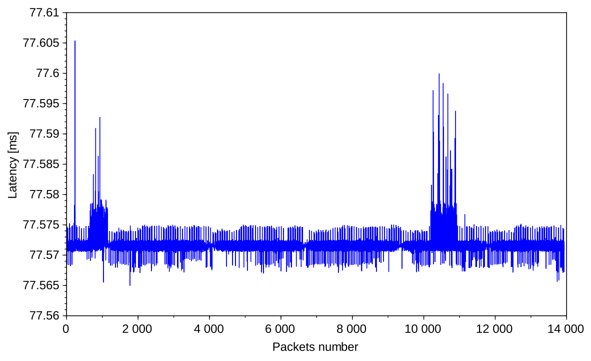

Figure 7.9: Latency for stream 3 → 0.

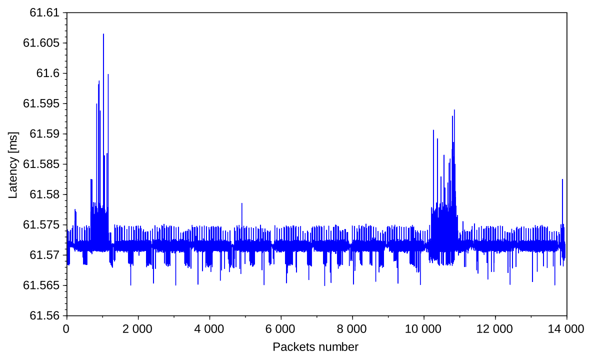

Figure 7.10: Latency for stream 4 → 0.

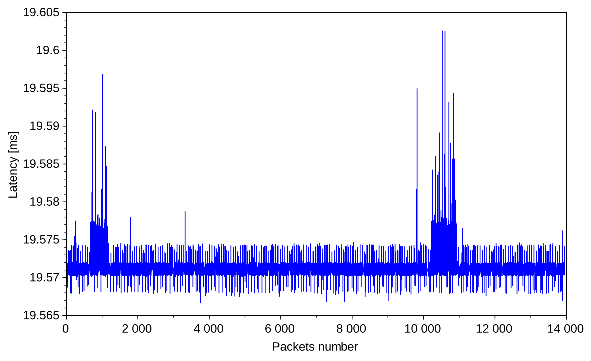

Figure 7.11: Latency for stream 5 → 0.

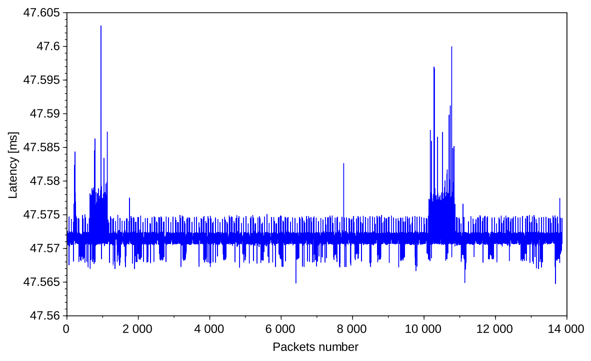

Figure 7.12: Latency for stream 6 → 0.

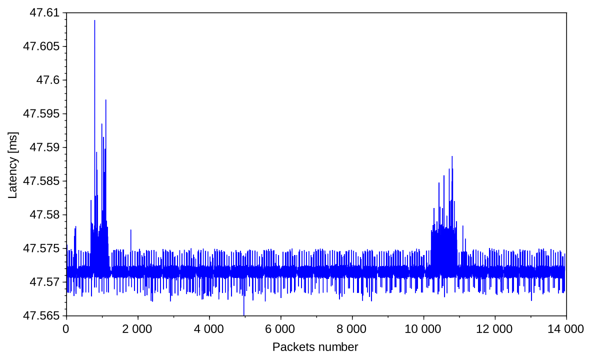

Figure 7.13: Latency for stream 7 → 0.

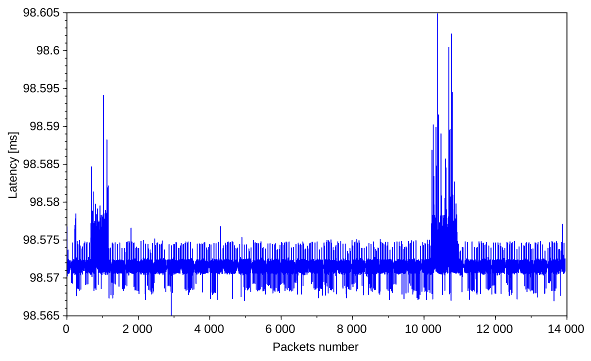

Figure 7.14: Latency for stream 8 → 0.

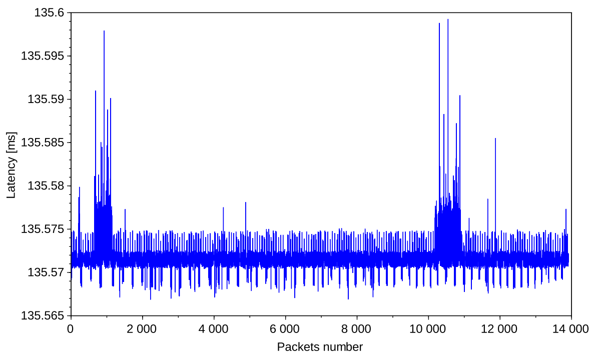

Figure 7.15: Latency for stream 9 → 0.

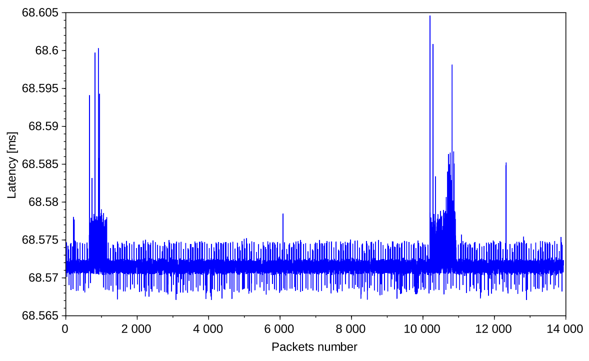

Figure 7.16: Latency for stream 10 → 0.

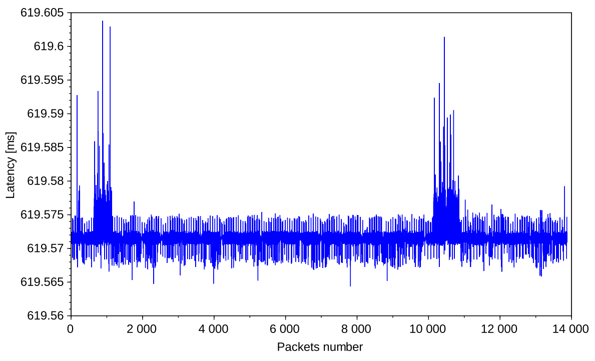

Figure 7.17: Latency for stream 11 → 0.

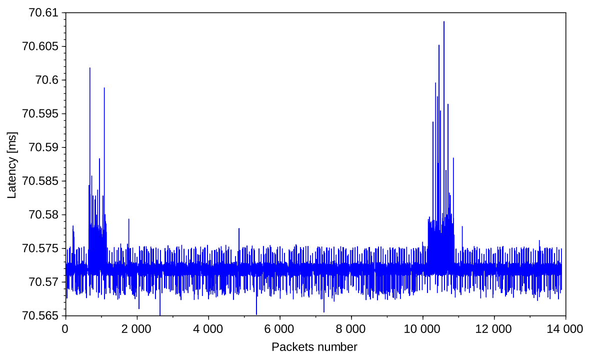

Figure 7.18: Latency for stream 12 → 0.

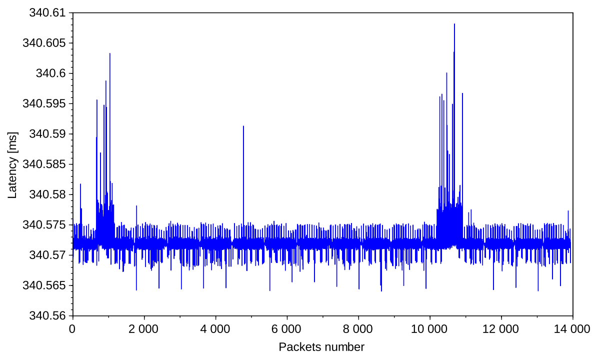

Figure 7.19: Latency for stream 13 → 0.

Low Interference Experiment

Setup The second complete experiment using all the 14 available WandStem nodes and the callbacks API was conducted for 6 hours, during less hostile hours.

The complete resulting topology is represented in Fig. 7.20.

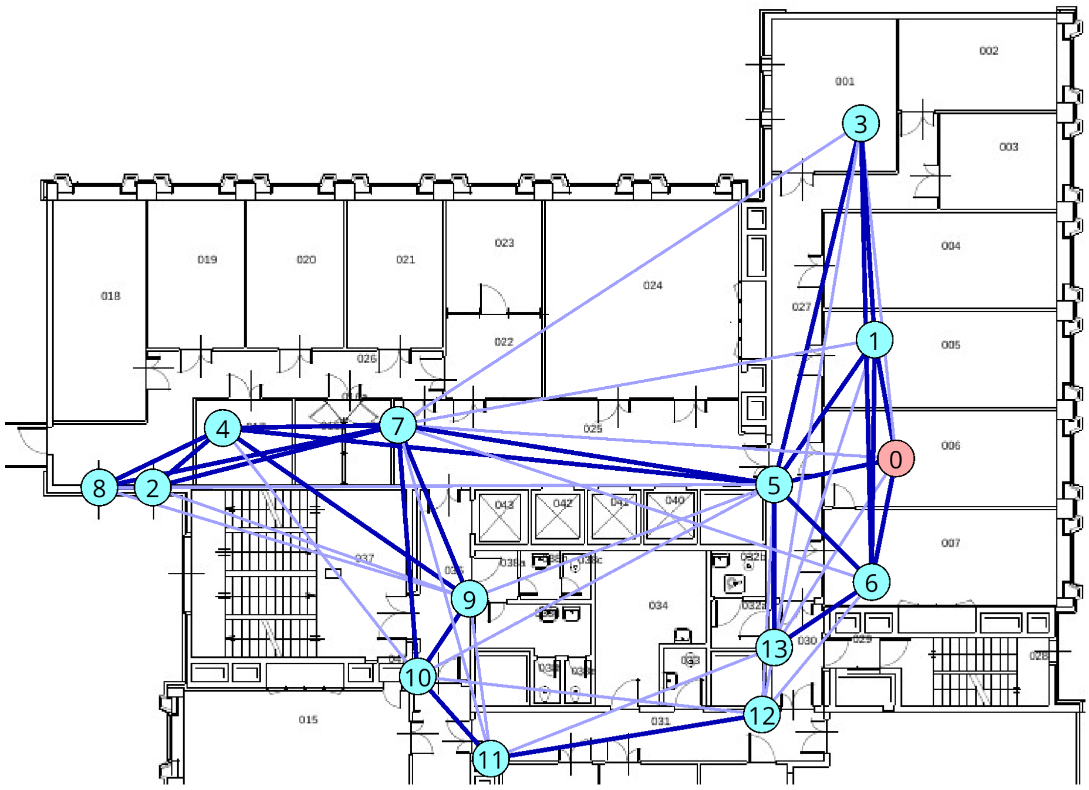

Figure 7.20: Second callbacks API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones.

Results Every node sent around 23 thousand packets through the opened stream and, overall, the network showed a reliability of 99.94%, with more than half of the streams have 100.00% of packets correctly delivered and all of them being above 99.80%, except for stream 12 $\rightarrow$ 0. It can be seen from the network topology in Fig. 7.20 that 12 does not have a strong link to node 13 (that was instead established during the first experiment) and so its packets are scheduled to follow a 5-hops long path to reach the master, through nodes 11, 10, 7 and 5.
Detailed reliability results for each stream are reported in Tab. 7.3, including reliability values for single, double and triple streams redundancies.

Stream	Sent packets	Reliability Single	Reliability Double	Reliability Triple
1-0	22850	100.00%	100.00%	100.00%
2-0	22595	99.77%	100.00%	100.00%
3-0	22839	99.78%	99.98%	100.00%
4-0	22777	99.78%	99.97%	100.00%
5-0	22787	99.98%	100.00%	100.00%
6-0	22843	99.99%	100.00%	100.00%
7-0	22787	99.96%	100.00%	100.00%
8-0	22594	99.83%	100.00%	100.00%
9-0	22608	99.26%	99.81%	100.00%
10-0	22839	99.87%	99.90%	99.90%
11-0	22832	99.78%	99.81%	99.81%
12-0	22832	99.38%	99.54%	99.54%
13-0	22777	99.45%	99.88%	100.00%
Total	295960	99.76%	99.79%	99.94%

Table 7.3: Streams reliability during the second callbacks API experiment.

Plots in Fig. 7.21 to Fig. 7.33 show, for each node from 1 to 13, the latency of every packet received by the master. In this second experiment no re-scheduling processes took place and so also latency plot have a flatter trend.

Very few latency spikes (experienced for single packets and never on multiple consecutive packets) can be noticed and the highest measured jitter, with respect to the mean latency value, is 32.29 μs for stream 10 $\rightarrow$ 0. The average latency standard deviation among all the streams is 0.82 μs, which is again a really appreciable result. This also highlights the fact that the highest latency spikes are restricted to very few samples.
The mentioned resulting values are reported in Tab. 7.4.

Stream	Average latency	Jitter	Standard deviation
1-0	19.5712 ms	23.20 μs	0.90 μs
2-0	633.5715 ms	12.94 μs	0.83 μs
3-0	49.5715 ms	6.83 μs	0.80 μs
4-0	47.5716 ms	23.87 μs	0.80 μs
5-0	19.5715 ms	27.88 μs	0.78 μs
6-0	19.5715 ms	25.30 μs	0.80 μs
7-0	84.5711 ms	20.44 μs	0.80 μs
8-0	642.5715 ms	6.73 μs	0.78 μs
9-0	91.5715 ms	7.11 μs	0.81 μs
10-0	119.5715 ms	32.29 μs	0.82 μs
11-0	226.5715 ms	20.21 μs	0.93 μs
12-0	256.5715 ms	18.92 μs	0.83 μs
13-0	277.5715 ms	24.31 μs	0.80 μs
Total	–	19.29 μs	0.82 μs

Table 7.4: Streams latency statistics during the first callbacks API experiment.

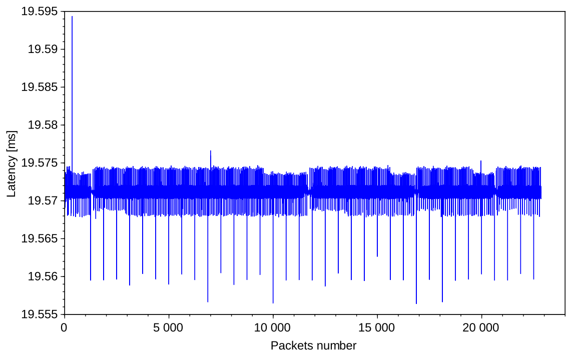

Figure 7.21: Latency for stream 1 $\rightarrow$ 0.

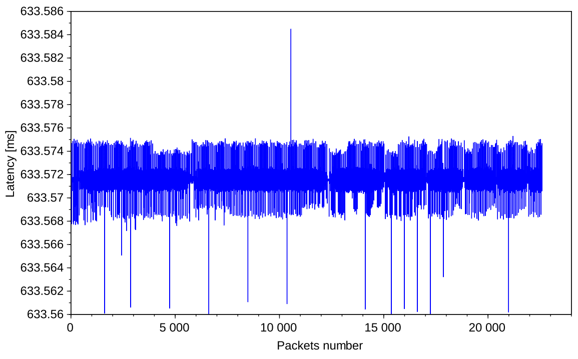

Figure 7.22: Latency for stream 2 $\rightarrow$ 0.

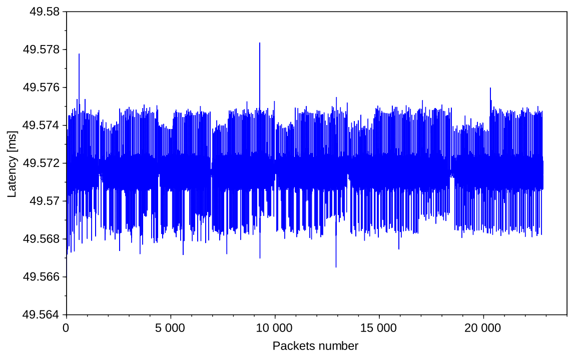

Figure 7.23: Latency for stream 3 → 0.

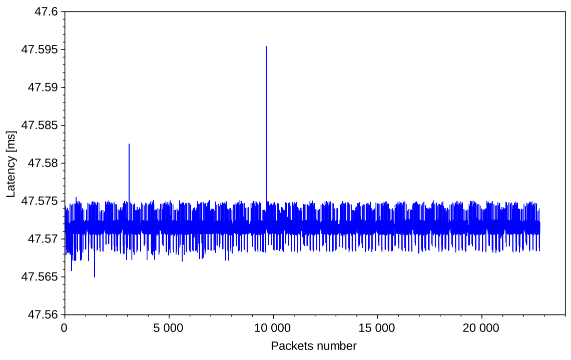

Figure 7.24: Latency for stream 4 → 0.

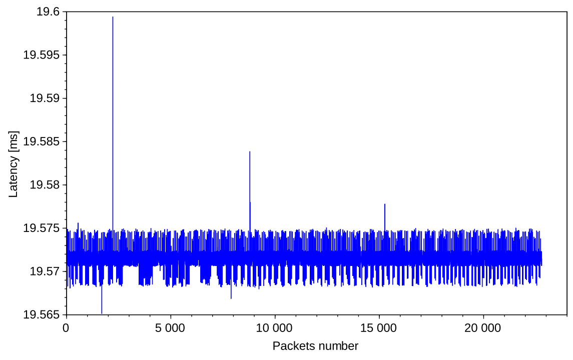

Figure 7.25: Latency for stream 5 → 0.

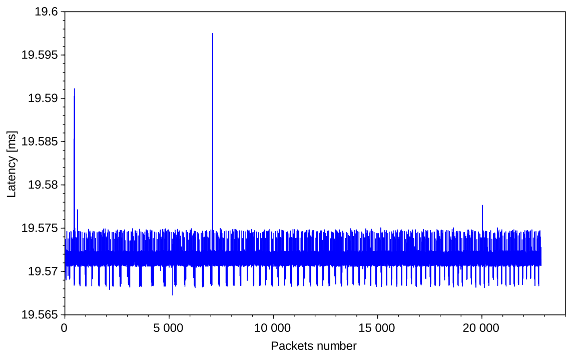

Figure 7.26: Latency for stream 6 → 0.

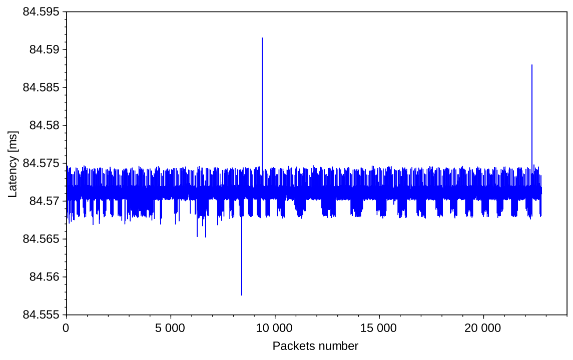

Figure 7.27: Latency for stream 7 → 0.

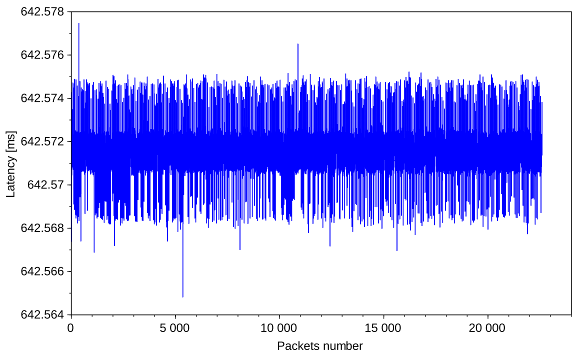

Figure 7.28: Latency for stream 8 → 0.

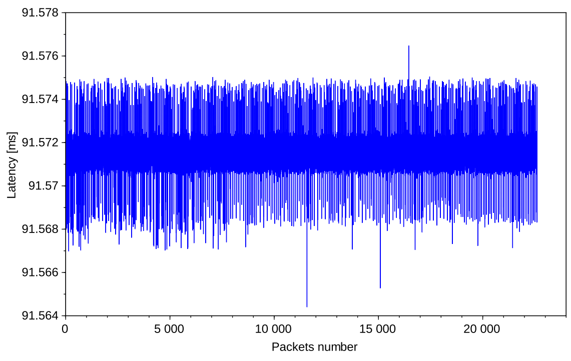

Figure 7.29: Latency for stream 9 → 0.

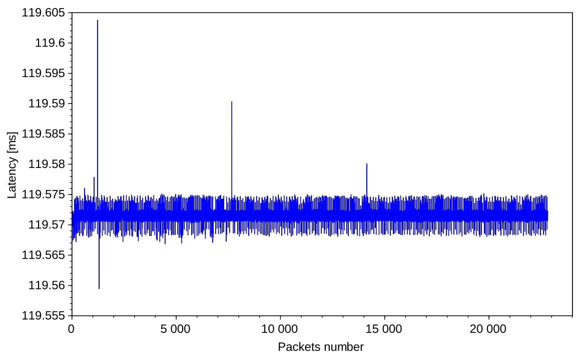

Figure 7.30: Latency for stream 10 → 0.

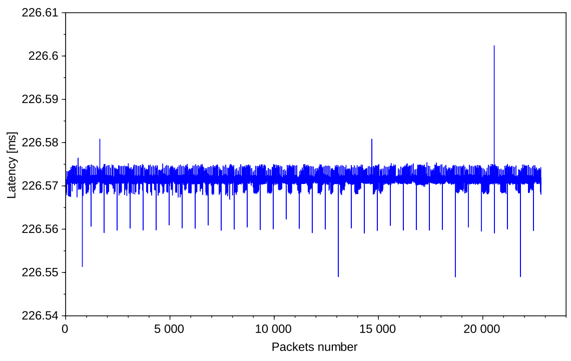

Figure 7.31: Latency for stream 11 → 0.

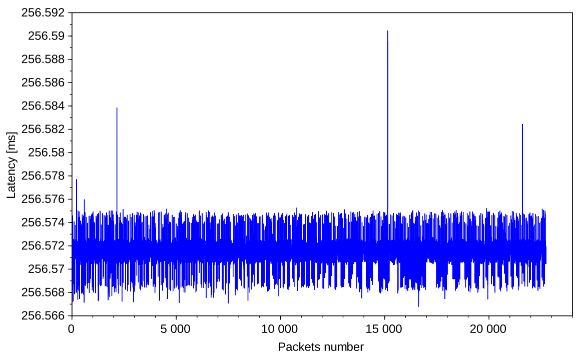

Figure 7.32: Latency for stream 12 → 0.

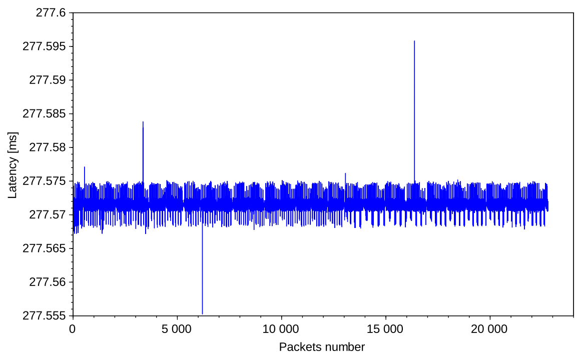

Figure 7.33: Latency for stream 13 → 0.

7.3.2. Write/Wait API

When using the *write/wait* API, dynamic nodes open a stream to the master node and, after a successful connection, simply wait until their assigned slot. When they are woken up by the session layer, they instantiate the new packet (with an incremented counter and the current timestamp) and call the *write* primitive. After the *write* returns, they put themselves again into the waiting state.
TDMH data slots duration, in this case, is 6 ms. The wake-up advance time required by all the streams is instead equal to two slots (12 ms).

High Interference Experiment

Setup The first complete experiment using all the 14 available WandStem nodes and the *write/wait* API was conducted for almost 3.5 hours during a working afternoon.

The complete resulting topology is represented in Fig. 7.34.

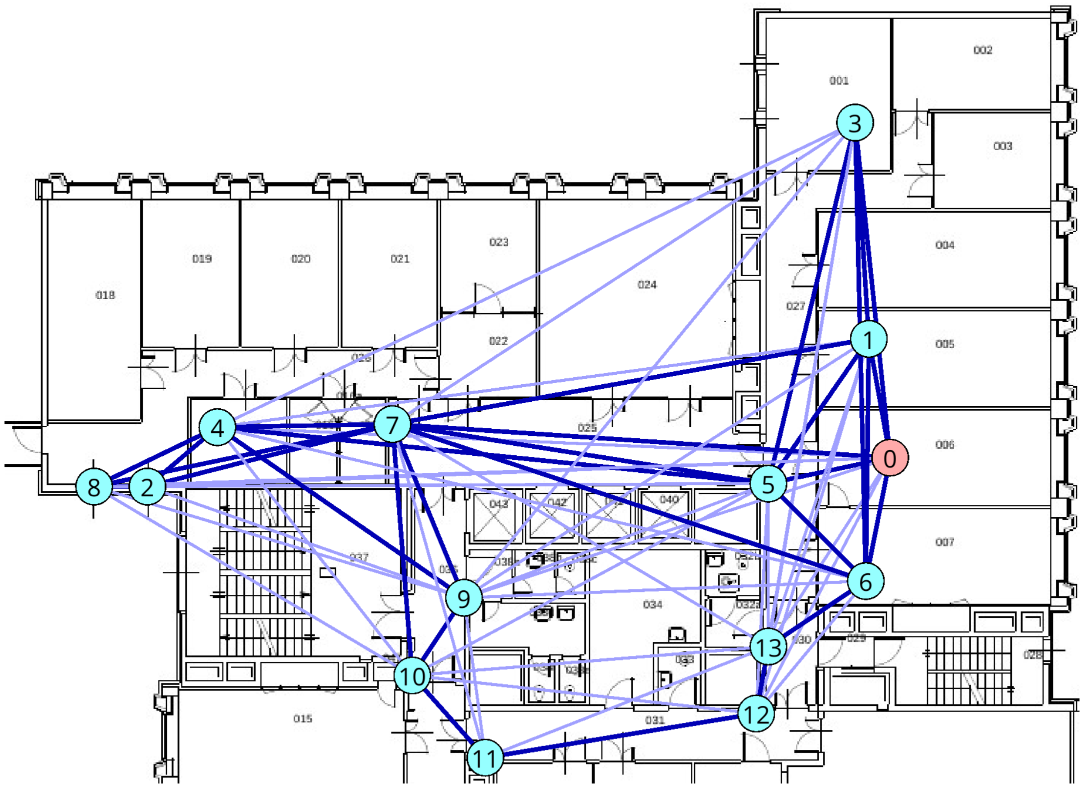

Figure 7.34: First write/wait API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones.

Results Every node sent around 11.5 thousand packets through the opened stream and, overall, the network showed a reliability of 99.88%, with all the streams being above 99.60%.

Detailed reliability results for each stream are reported in Tab. 7.5, including reliability values for single, double and triple streams redundancies.
It is interesting to see how redundancy almost completely cancels packets loss on stream $2 \rightarrow 0$, passing from only 87.84% of reliability with single redundancy (i.e. no redundancy), up to 99.80% with triple redundancy.

Stream	**Sent packets**	**Reliability Single**	**Reliability Double**	**Reliability Triple**
1-0	11515	98.58%	99.70%	99.79%
2-0	11524	87.84%	93.92%	99.80%
3-0	11504	94.63%	99.19%	99.63%
4-0	11525	96.60%	99.72%	100.00%
5-0	11520	98.75%	99.92%	99.98%
6-0	11514	99.90%	100.00%	100.00%
7-0	11525	96.84%	99.44%	99.70%
8-0	11525	96.13%	99.56%	99.99%
9-0	11504	95.74%	99.39%	99.94%
10-0	11504	95.96%	99.53%	99.98%
11-0	11504	95.67%	99.24%	99.70%
12-0	11219	98.23%	99.96%	100.00%
13-0	11514	97.92%	98.00%	99.87%
Total	149397	96.36%	99.04%	99.88%

Table 7.5: Streams reliability during the first write/wait API experiment.

In this experiment, one re-scheduling took place after few packets from the beginning. The activation time of the second schedule is called T_{act}. In order to compute statistics about each stream, the latencies measured before T_{act} have been aligned to the ones measured after this time instant, in order to compute some metrics such as the jitter and the standard deviation. A value equal to the difference of the average latency measured in the two periods was subtracted (or summed) to the latency of packets that were sent before T_{act}.

As expected, the measured latency appears to be noisier than using the callbacks API.

Anyway, the highest jitter with respect to the latency mean value is 107.53 μs for stream $8 \rightarrow 0$. The average latency standard deviation among all the streams is 4.93 μs and, in fact, the highest experienced latency spikes are very few isolated outliers.

Stream	Average latency	Jitter	Standard deviation
1-0	28.6518 ms	90.60 μs	4.98 μs
2-0	52.6999 ms	104.03 μs	6.29 μs
3-0	292.6250 ms	99.79 μs	4.20 μs
4-0	46.6542 ms	89.75 μs	5.02 μs
5-0	28.6252 ms	65.57 μs	4.49 μs
6-0	28.6728 ms	97.18 μs	4.53 μs
7-0	28.5981 ms	76.00 μs	4.60 μs
8-0	92.6268 ms	107.53 μs	5.60 μs
9-0	52.5990 ms	80.32 μs	4.71 μs
10-0	92.6185 ms	73.28 μs	4.80 μs
11-0	470.7479 ms	102.21 μs	5.22 μs
12-0	428.6189 ms	83.59 μs	4.48 μs
13-0	228.7199 ms	99.11 μs	5.13 μs
Total	–	89.92 μs	4.93 μs

Table 7.6: Streams latency statistics during the first *write/wait* API experiment.

Plots in Fig. 7.35 to Fig. 7.52 show, for each node from 1 to 13, the latency of every packet received by the master during the plateau, with a zoomed plot around the re-scheduling moment for the affected nodes.

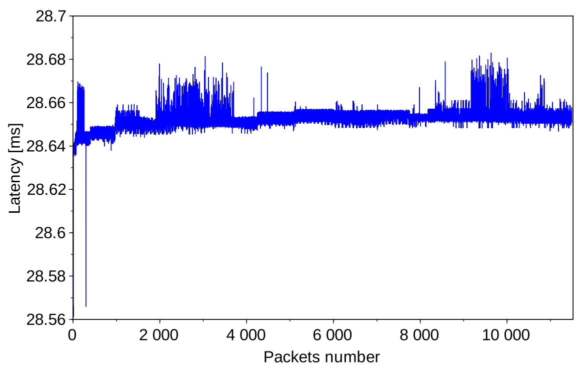

Figure 7.35: Latency for stream $1 \rightarrow 0$.

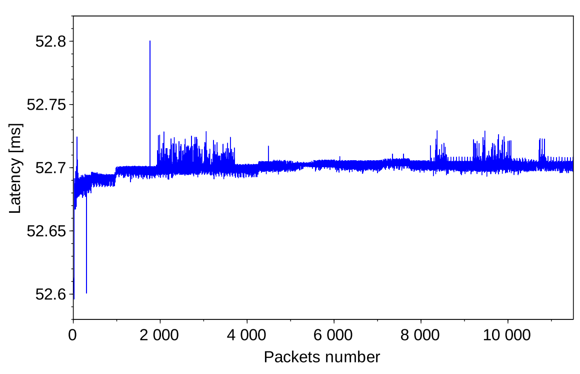

Figure 7.36: Latency for stream $2 \rightarrow 0$.

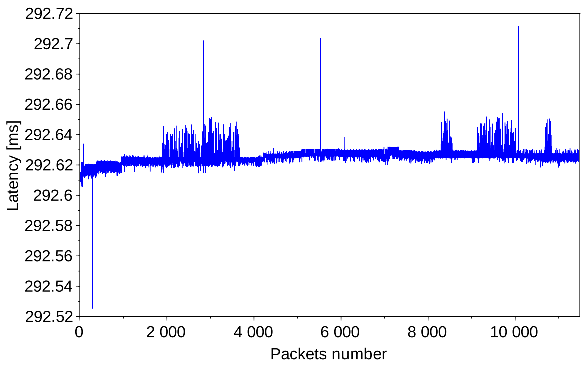

Figure 7.37: Latency for stream 3 → 0.

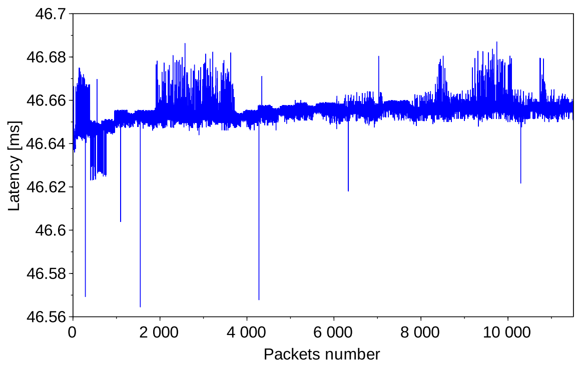

Figure 7.38: Latency for stream 4 → 0.

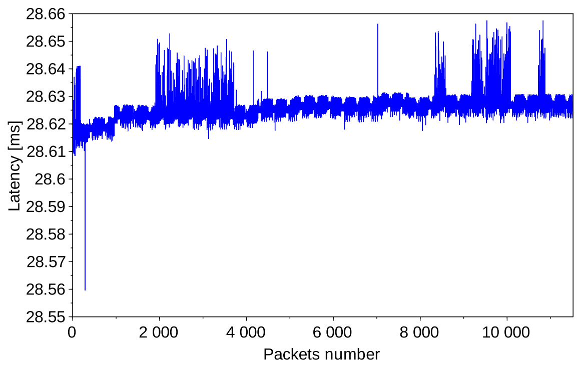

Figure 7.39: Latency for stream 5 → 0.

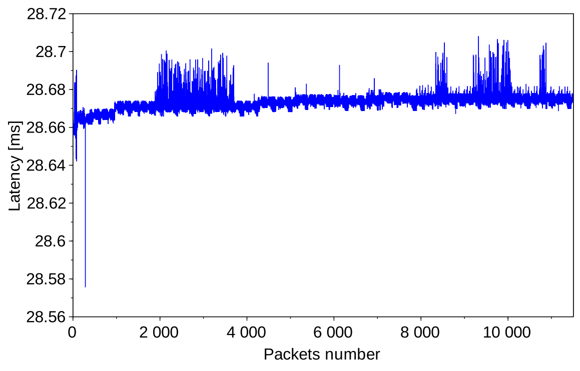

Figure 7.40: Latency for stream 6 → 0.

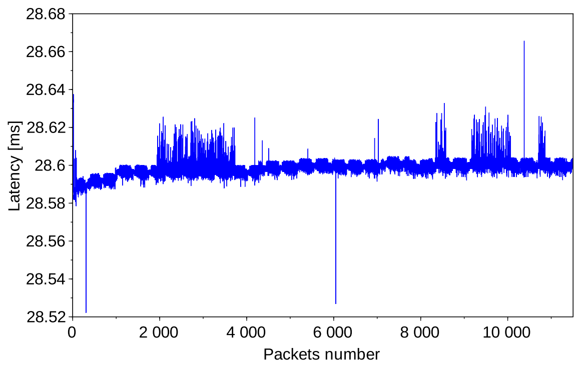

Figure 7.41: Latency for stream 7 → 0.

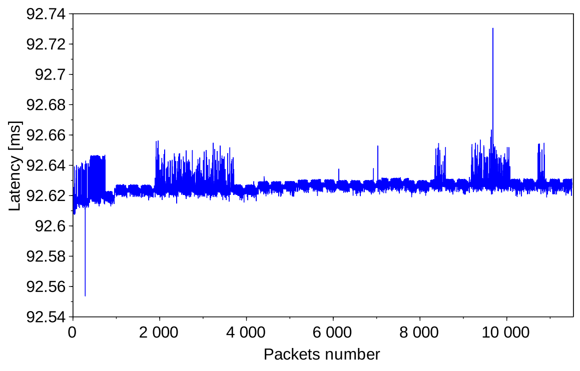

Figure 7.42: Latency for stream 8 → 0.

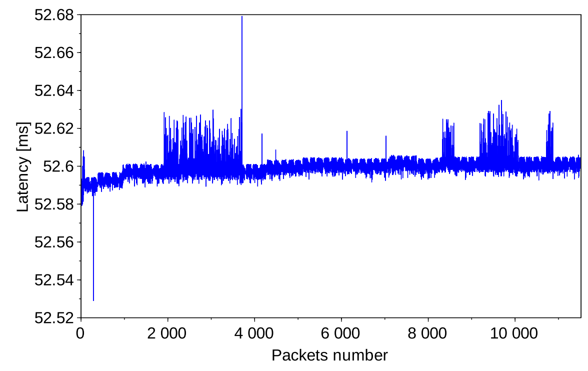

Figure 7.43: Latency for stream 9 → 0.

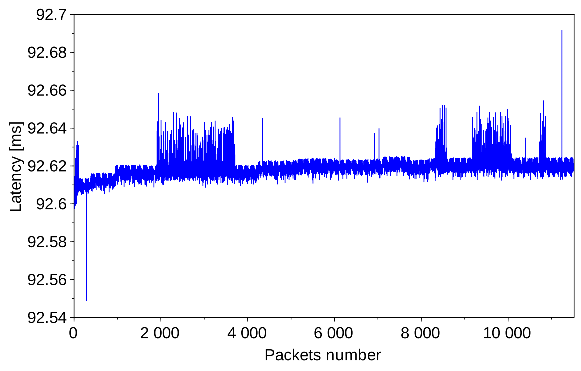

Figure 7.44: Latency for stream 10 → 0.

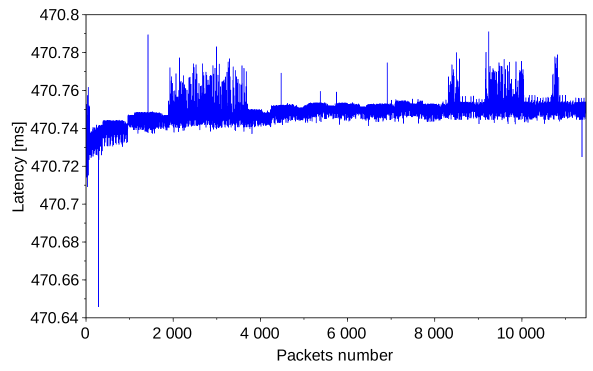

Figure 7.45: Latency for stream 11 → 0.

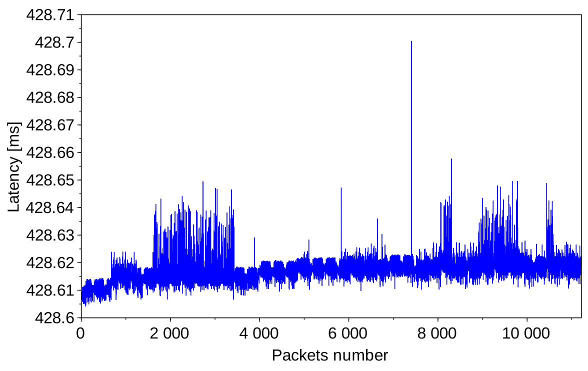

Figure 7.46: Latency for stream 12 → 0.

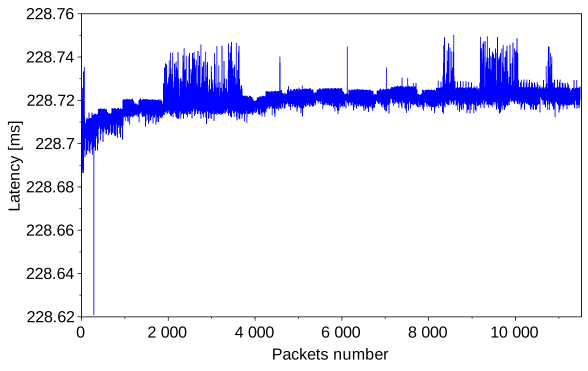

Figure 7.47: Latency for stream 13 → 0.

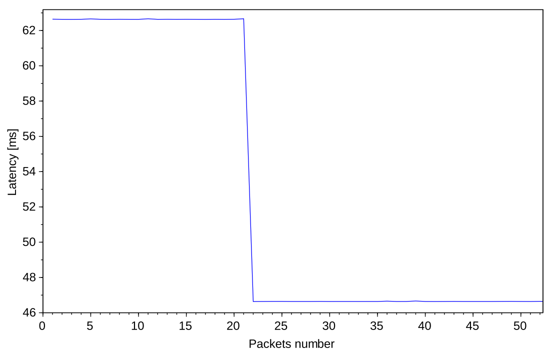

Figure 7.48: Latency for stream 4 → 0 around re-scheduling.

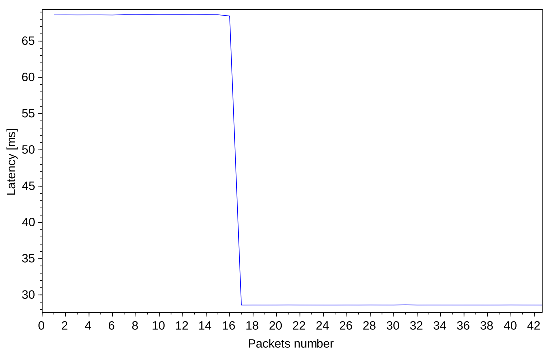

Figure 7.49: Latency for stream 5 → 0 around re-scheduling.

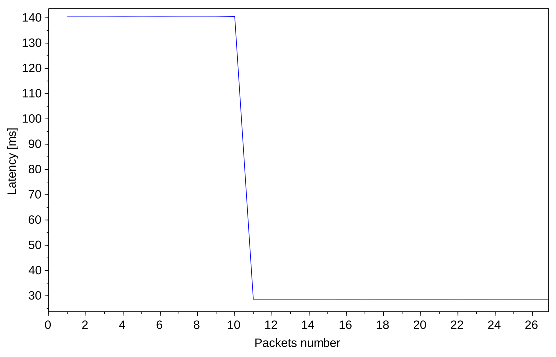

Figure 7.50: Latency for stream 6 → 0 around re-scheduling.

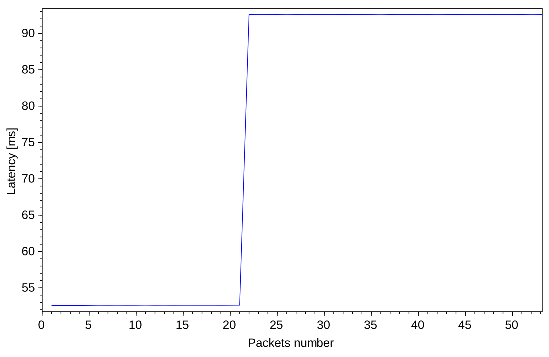

Figure 7.51: Latency for stream 8 → 0 around re-scheduling.

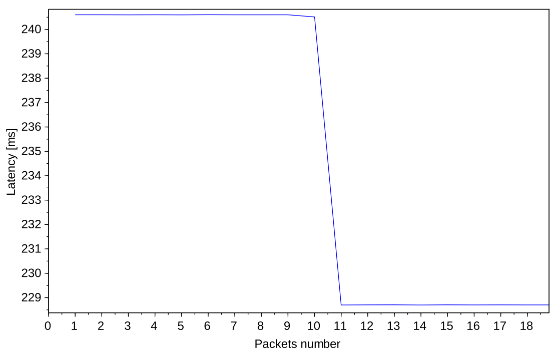

Figure 7.52: Latency for stream 13 → 0 around re-scheduling.

Low Interference Experiment

Setup The second complete experiment using all the 14 available WandStem nodes and the *write/wait* API was conducted for almost 14.5 hours, during less hostile hours. This is the longest experiment among the ones presented.

The complete resulting topology is represented in Fig. 7.20.

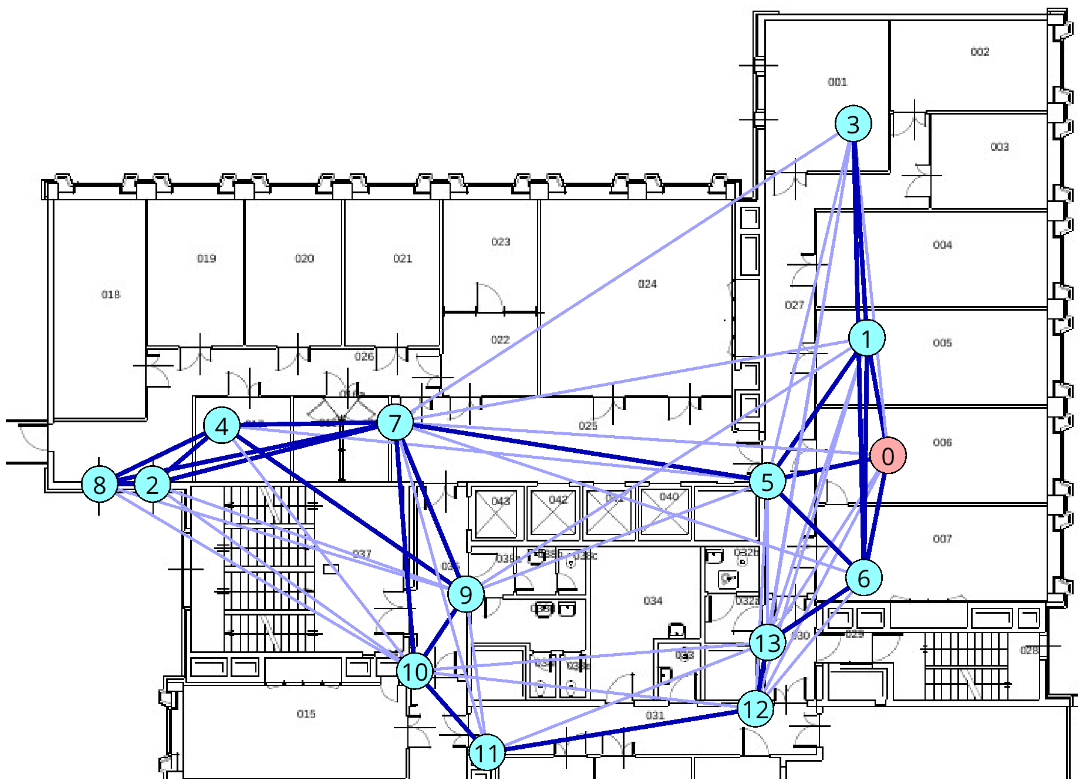

Figure 7.53: Second write/wait API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones.

Results Every node sent around 50 thousand packets through the opened stream and, overall, the network showed a reliability of 99.97%, with the lowest one being 99.87% for stream $2 \rightarrow 0$ and more than a half of the streams having 100.00% of packets correctly delivered.

Detailed reliability results for each stream are reported in Tab. 7.5, including reliability values for single, double and triple streams redundancies.

Stream	Sent packets	Reliability Single	Reliability Double	Reliability Triple
1-0	50245	99.90%	99.98%	99.99%
2-0	50251	99.63%	99.85%	99.87%
3-0	50251	99.27%	99.94%	100.00%
4-0	50251	99.70%	99.86%	99.90%
5-0	50251	99.82%	100.00%	100.00%
6-0	50013	100.00%	100.00%	100.00%
7-0	50251	99.67%	99.96%	100.00%
8-0	50251	99.73%	99.87%	100.00%
9-0	50221	99.69%	99.86%	99.91%
10-0	50221	99.68%	99.89%	99.91%
11-0	50234	99.42%	100.00%	100.00%
12-0	50234	99.82%	99.99%	100.00%
13-0	50234	99.87%	100.00%	100.00%
Total	652908	99.71%	99.94%	99.97%

Table 7.7: Streams reliability during the second write/wait API experiment.

In this experiment, one re-scheduling took place after few packets from the beginning. Latency values of packets sent before the second schedule was activated have been aligned to the following ones (as in the previous experiment) in order to compute some statistics.

Even if in this experiment the measured latency appears to be noisier for all the streams, it can be noticed that it never deviates more than 125.21 μs from the latency average of the relative stream. The mentioned highest jitter was measured for stream $5 \rightarrow 0$. In this scenario, the standard deviation is higher than in the previous experiments, but the its average among all the streams is 6.40 μs, which is still a good result. Again, this underlines how the packets whose latency deviates the most from the average value are an extremely low number.

The mentioned resulting values are reported in Tab. 7.8.

Stream	Average latency	Jitter	Standard deviation
1-0	28.6324 ms	87.65 μs	6.41 μs
2-0	64.6336 ms	107.21 μs	6.30 μs
3-0	62.6073 ms	86.63 μs	6.31 μs
4-0	98.6351 ms	92.72 μs	6.30 μs
5-0	28.7128 ms	125.21 μs	7.07 μs
6-0	68.6268 ms	48.56 μs	6.83 μs
7-0	46.6316 ms	92.39 μs	6.28 μs
8-0	216.6070 ms	105.74 μs	5.99 μs
9-0	198.6059 ms	93.27 μs	6.07 μs
10-0	216.6551 ms	113.17 μs	6.79 μs
11-0	322.6260 ms	79.42 μs	6.16 μs
12-0	104.6260 ms	84.30 μs	6.43 μs
13-0	68.6531 ms	87.40 μs	6.29 μs
Total	–	92.59 μs	6.40 μs

Table 7.8: Streams latency statistics during the second *write/wait* API experiment.

Plots in Fig. 7.54 to Fig. 7.74 show, for each node from 1 to 13, the latency of every packet received by the master during the plateau, with a zoomed plot around the rescheduling moment for the affected nodes. The little stairs that can be seen in the plots trend (easily noticeable in Fig. 7.59) are due to the fact that the master outputs to the log an increasing amount of bytes, mainly cause of the increasing timestamps values, through a thread whose priority is not the highest. Nevertheless, these stairs amount to around 5 μs and would be eliminated in a non-debugging setting, in which the latency results would be even better.

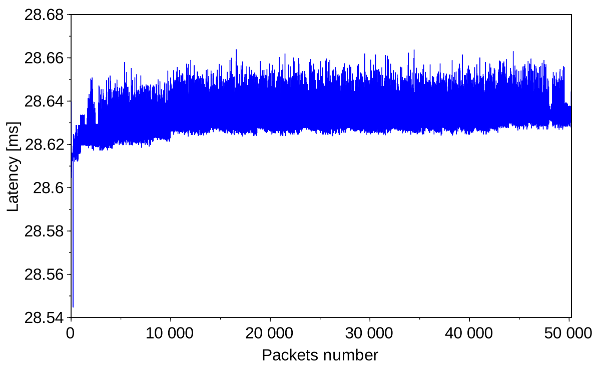

Figure 7.54: Latency for stream 1 → 0.

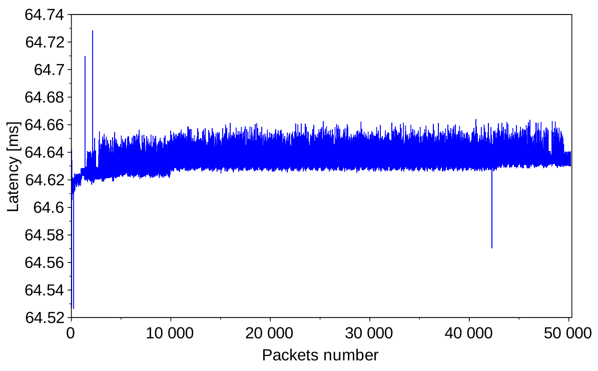

Figure 7.55: Latency for stream 2 → 0.

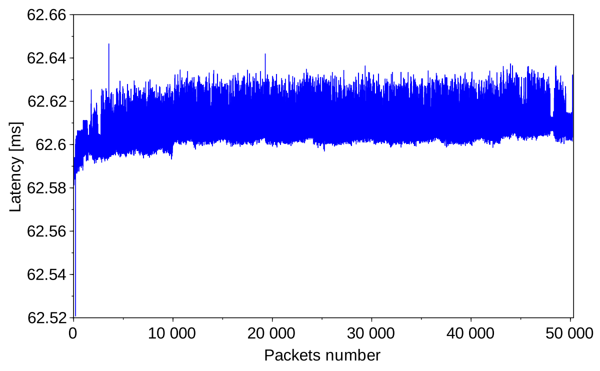

Figure 7.56: Latency for stream 3 → 0.

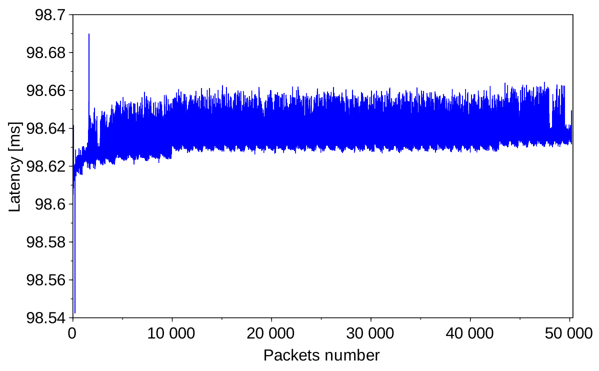

Figure 7.57: Latency for stream 4 → 0.

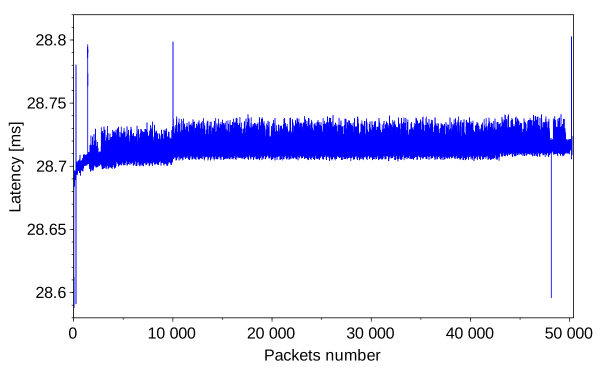

Figure 7.58: Latency for stream 5 → 0.

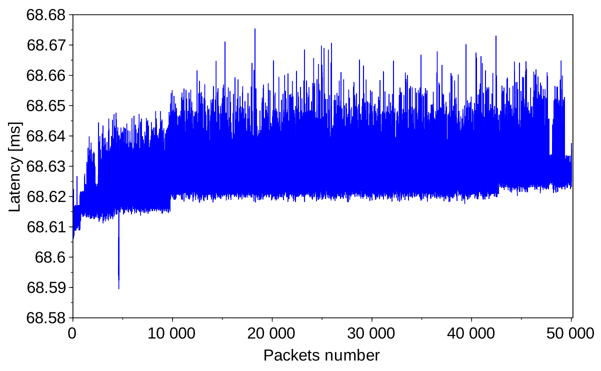

Figure 7.59: Latency for stream 6 → 0.

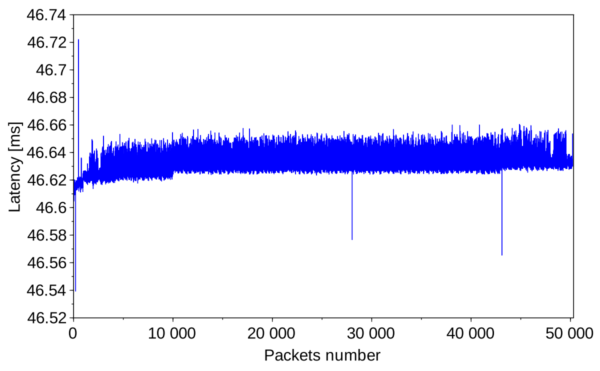

Figure 7.60: Latency for stream 7 → 0.

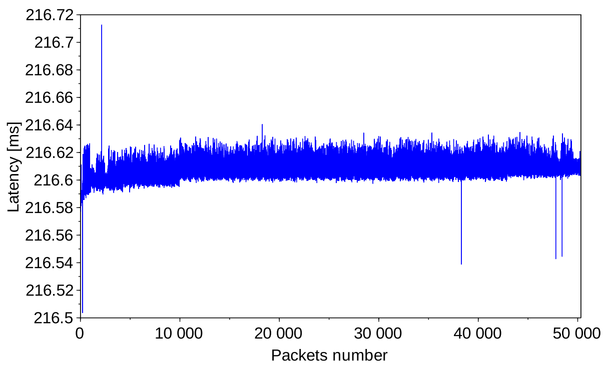

Figure 7.61: Latency for stream 8 → 0.

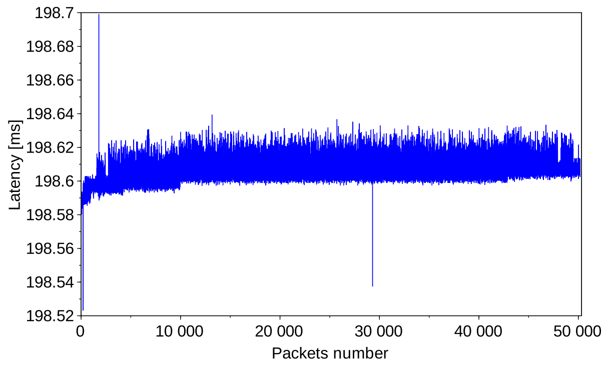

Figure 7.62: Latency for stream 9 → 0.

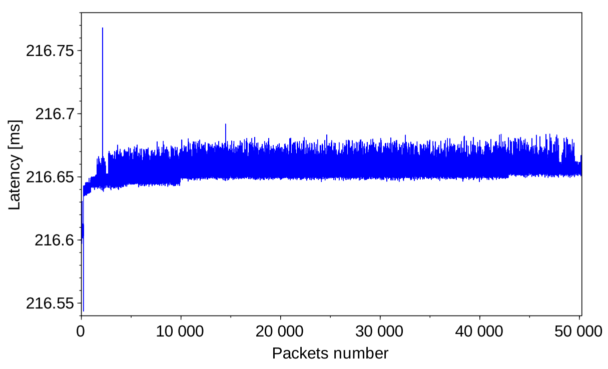

Figure 7.63: Latency for stream 10 → 0.

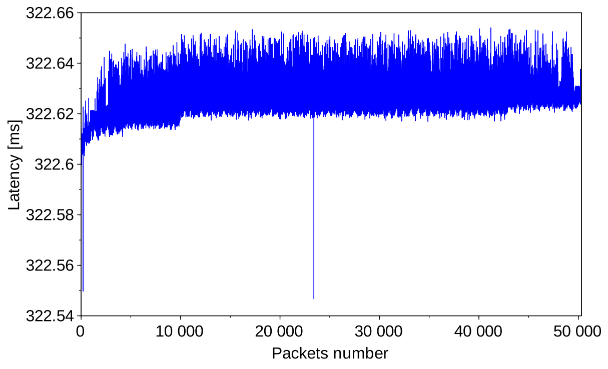

Figure 7.64: Latency for stream 11 → 0.

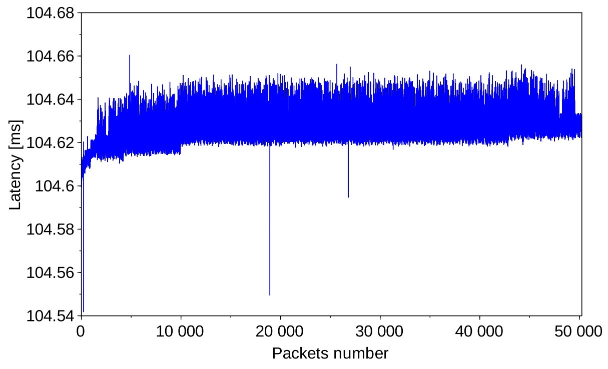

Figure 7.65: Latency for stream 12 → 0.

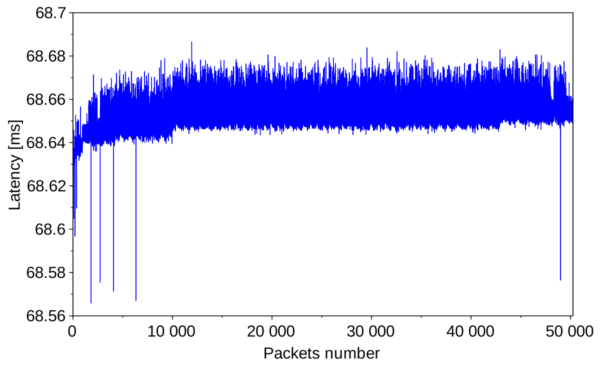

Figure 7.66: Latency for stream 13 → 0.

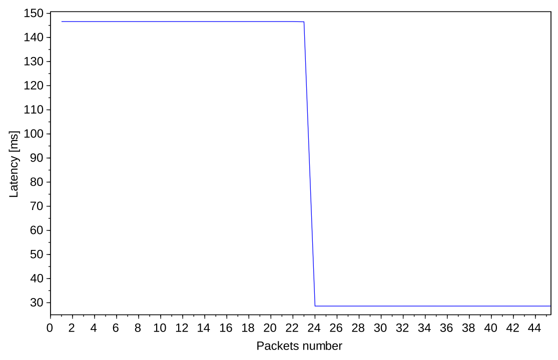

Figure 7.67: Latency for stream 1 → 0 around re-scheduling.

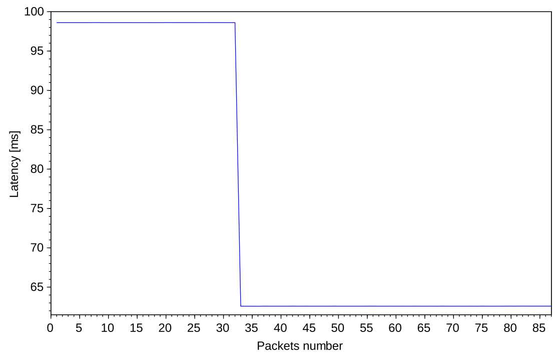

Figure 7.68: Latency for stream 3 → 0 around re-scheduling.

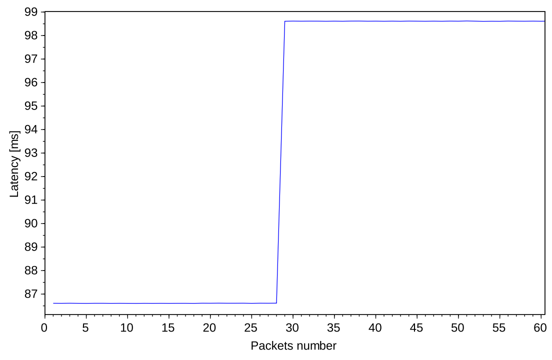

Figure 7.69: Latency for stream 4 → 0 around re-scheduling.

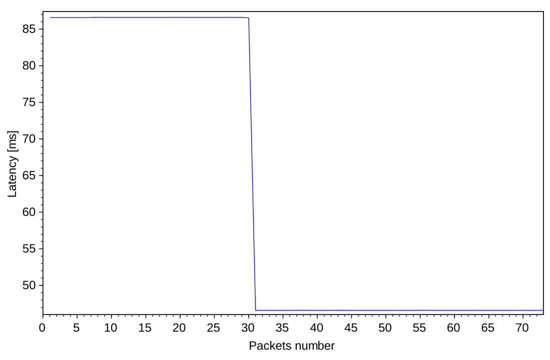

Figure 7.70: Latency for stream 7 → 0 around re-scheduling.

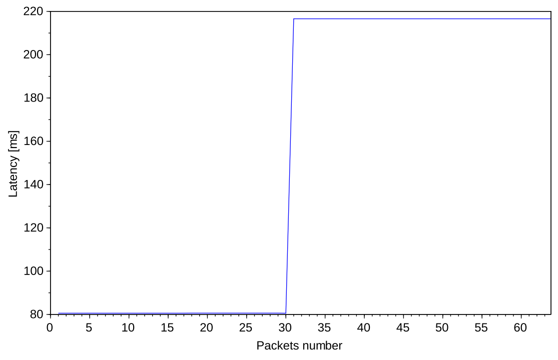

Figure 7.71: Latency for stream 8 → 0 around re-scheduling.

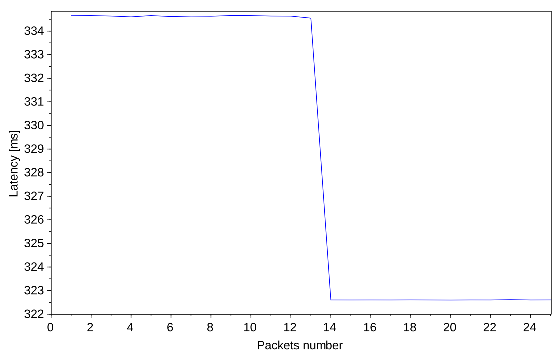

Figure 7.72: Latency for stream 11 → 0 around re-scheduling.

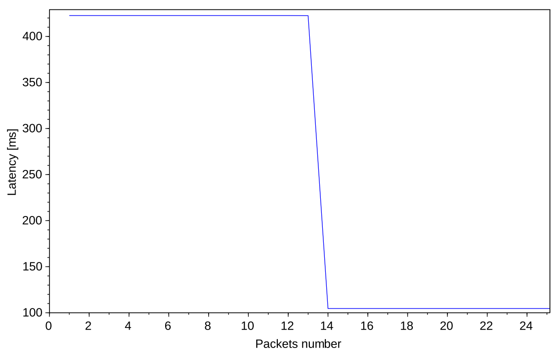

Figure 7.73: Latency for stream 12 → 0 around re-scheduling.

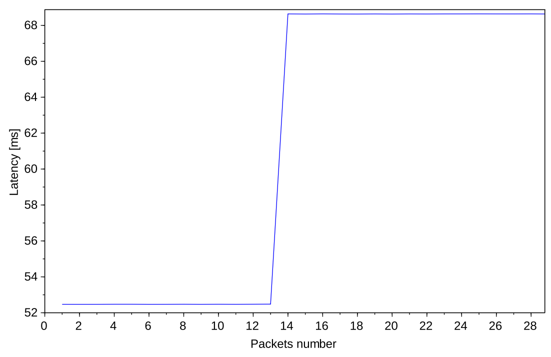

Figure 7.74: Latency for stream 13 → 0 around re-scheduling.

7.4. Distributed Control Loop

An experiment in a real-world control loop scenario was also conducted. It aims at proving that a feedback control loop can be achieved through a wireless network, by using TDMH. The goal of this experiment is to control the temperature of a tube furnace for semiconductors synthesis.

7.4.1. Setup

Nodes and Network Setup The network setup consists of a feedback controller running on the master node. The controller receives temperature measures from a node connected to the output serial port of the furnace electronics, on which temperature values (*process variable*) are written. The actuator is instead another dynamic node, connected to the furnace input serial port: after receiving the *control variable* from the controller it writes the value to the serial port. The control action is represented by the power percentage to be fed to the furnace heating element (i.e. it corresponds to the duty cycle of the *pulse-width modulation* (PWM) signal fed to the heating element).
Other nodes are connected to the network in order to have multiple hops between these three main nodes. All these other devices do not open any stream but participate in the network topology.

Streams period is set to 10 tiles and each tile lasts 100 ms, which means that nodes send a data packet over their streams every second. Streams are also opened requiring triple spatial redundancy.

Moreover, the sensor node uses the *write/wait* API with a wake-up advance equal to $T_{advance} = 2 \cdot T_{slot} = 12\ ms$, while the other two nodes always send data as a reaction to a received packet. For example, the controller only sends data to the actuator after a temperature sample is received (and after computing the new control action).

In order to log all the data, the actuator also measures the latency from the sensor to itself and sends back the measure to the master node. This is possible because the sensor also forwards samples timestamps, together with the measured temperature value.

Since the temperature sampling period and the control period are set to 1 s and between sensor and actuator two steps elapse, the actuator forces a sleep of 2 s with respect to the received sensor sample timestamp before actuating, minus a slack time needed for the actuation itself. The combination of the usage of the *write/wait* API on the sensor node and the two seconds sleep in the actuator guarantees to have a constant control period.

Fig. 7.75 shows the interactions between the three main network nodes, the sensor, the controller and the actuator.

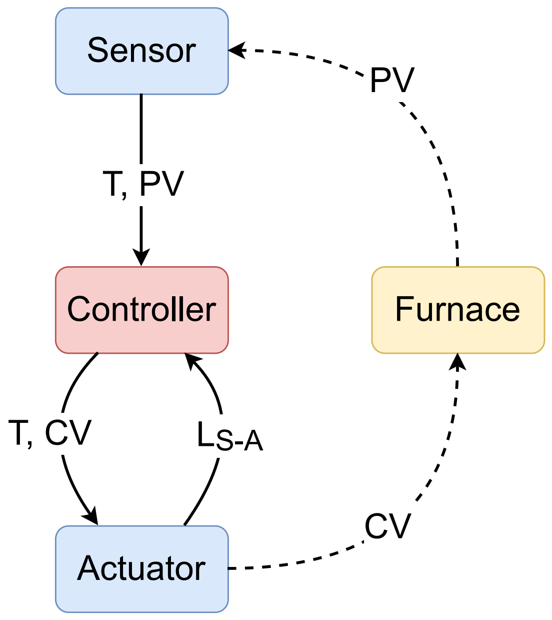

Figure 7.75: Interactions among sensor, controller and actuator nodes. Solid line arrows represent streams, through which the sampling timestamp (T), process variable (PV) and the control variable (CV) are exchanged. The actuator also transmits the sensor-actuator latency (L_{S-A}), back to the master. Interactions with the furnace are also shown: dashed arrows represent serial port communication.

Controller Design The furnace process is modeled as:

$$P(s) = \frac{\mu}{1 + \tau s}$$

where $\mu = 1250$ and $\tau = 200$.

Through the *implicit Euler* formula, substituting $s = \frac{z-1}{zTs}$ in the expression of $P(s)$, the process model in discrete time can be found:

$$P(z) = \frac{\mu T z}{(T_s + \tau)z - \tau}$$

T_s represents the control period and is set to 1 s. This is also the network nodes streams period.

Knowing that $P(z) = \frac{y(k)}{u(k)}$, the expression of the process variable $y(k)$ can be derived:

$$y(k) = \frac{\tau}{\tau + T_s} y(k-1) + \mu \frac{T_s}{\tau + T_s} u(k)$$

where $u(k)$ is the control variable.

The error $e(k)$ is expressed as the different between the required set point value $\bar{y}$ and the measured process variable:

$$e(k) = \bar{y} - y(k)$$

The controller was designed through the *direct synthesis for set point tracking* method, in which the controller design is based on the process model and the desired closed-loop transfer function.
The desired transfer function in this case is:

$$F(z) = \frac{1 - \alpha}{z(z - \alpha)}$$

where α is a design parameter and is equal to $\alpha = 0.9985$. It was selected in order to force the controller to reach the desired setpoint in around one hour.

The controller $R(z)$ is found knowing that:

$$F(z) = \frac{R(z)P(z)}{1 + R(z)P(z)}$$

The resulting controller expression is:

$$R(z) = \frac{(1-\alpha)(T_s+\tau)z^{-2} - \tau(1-\alpha)z^{-3}}{\mu T_s - \alpha\mu T_s z^{-1} - \mu T_s(1-\alpha)z^{-2}}$$

Finally, knowing that $R(z) = \frac{u(k)}{e(k)}$, the expression for the control variable $u(k)$ can be derived:

$$u(k) = \alpha u(k-1) + (1-\alpha)u(k-2) + (1-\alpha)\frac{T_s+\tau}{\mu T_s}e(k-2) - (1-\alpha)\frac{\tau}{\mu T_s}e(k-3)$$

As required, the found expression of $u(k)$ depends on the two previous steps.

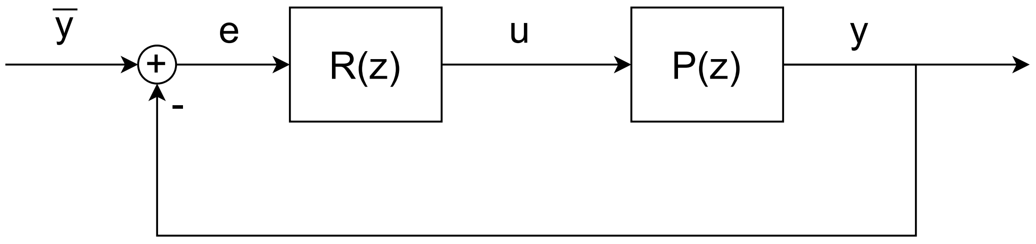

Figure 7.76: Feedback control loop representation, showing the set point $\bar{y}$, the error e, the control variable u and the process variable y.

Handling Packets Misses Particular attention is needed to the fact that in a distributed wireless setting data packets can be lost. The actuator may not receive the next control action, but this is not a big concern since it would maintain the previous one. When instead sensor samples are missed, some strategy on the controller logic is needed. The implemented logic does not compute a new control action when a packet is missed, since no new sensor sample exists. When finally a new packet is received, the controller is re-initialized, based on the last available control action and error. This avoids bumps on computed the control action, while instead, it continues following a smooth trend.

7.4.2. Results

Simulation The designed controller was first of all simulated, together with the process model, using *Scilab*. Then, the furnace microcontroller was used in a hardware-in-the-loop fashion, simulating the process behavior through the expression of $y(k)$, without actually feeding power to the nor sampling the temperature sensor. This was useful to test both the network and the serial communication. The real setup was used only when everything was tested out and correctly working.

Real Experiment The real experiment was deployed on the first floor of Building 21 at Politecnico di Milano and a total of 13 WandStem nodes were used. Unfortunately, node 11 experienced some power supply issue that made it difficult to form strong links with other nodes (too low RSSI), leading it to continuously lose packets and de-synchronize from the network. For this reason, it was not deployed, since it was not strictly necessary for the correct experiment outcome.

Nodes 12 and 9 respectively represent the sensor and the actuator nodes and were connected via serial communication to the furnace electronics, while node 0 is the master node and executes the designed feedback controller.
For this experiment, the desired temperature set point is 1000 $°C$.

The resulting network setup is shown in Fig. 7.77. It can be seen how packets need to traverse multiple hops on the path that connects the controller to the sensor and actuator nodes. A minimum of 3 hops are traversed between controller and actuator, while 4 are needed to reach the sensor node. Indeed, the controller-actuator stream was scheduled to use the path $0 \rightarrow 5 \rightarrow 1 \rightarrow 9$, while the stream sensor-controller used instead the two redundant paths $12 \rightarrow 9 \rightarrow 1 \rightarrow 5 \rightarrow 0$ and $12 \rightarrow 9 \rightarrow 1 \rightarrow 4 \rightarrow 5 \rightarrow 0$.

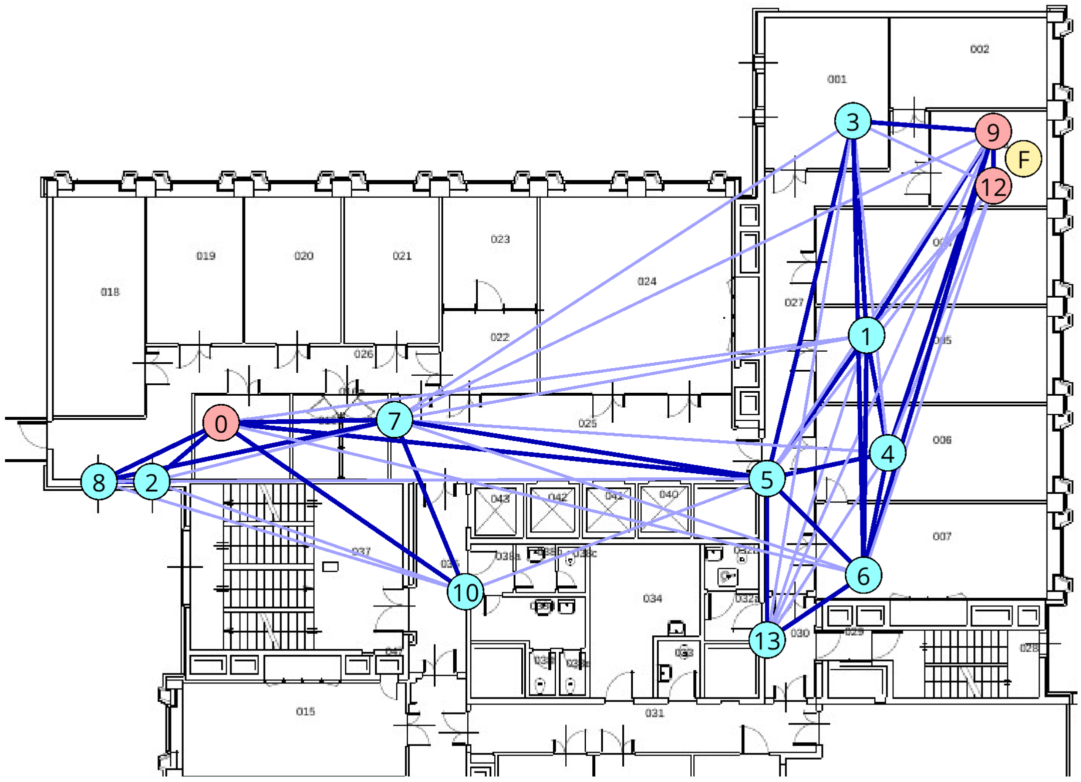

Figure 7.77: Control loop experiment network setup. The controller (0), sensor (12) and actuator (9) nodes are underlined in orange. The furnace (F) is identified by the yellow circle. Strong wireless links are represented by dark blue lines, while light blue lines indicate the weak ones.

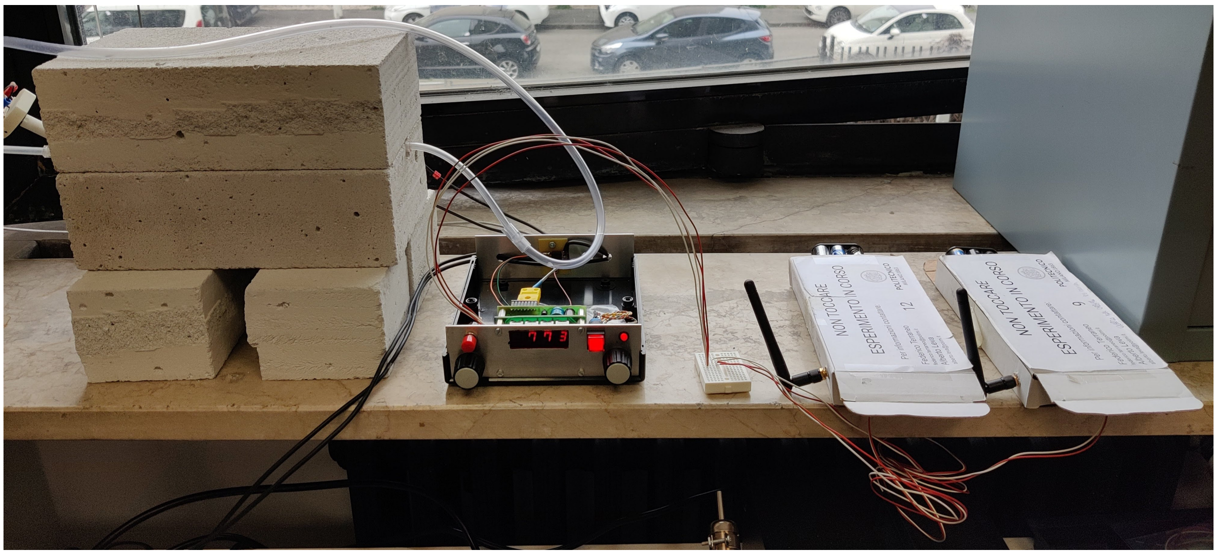

Figure 7.78: Experiment setup showing the furnace electronics and the sensor and actuator nodes, during the temperature ascent. Current temperature is shown on the 7-segment display.

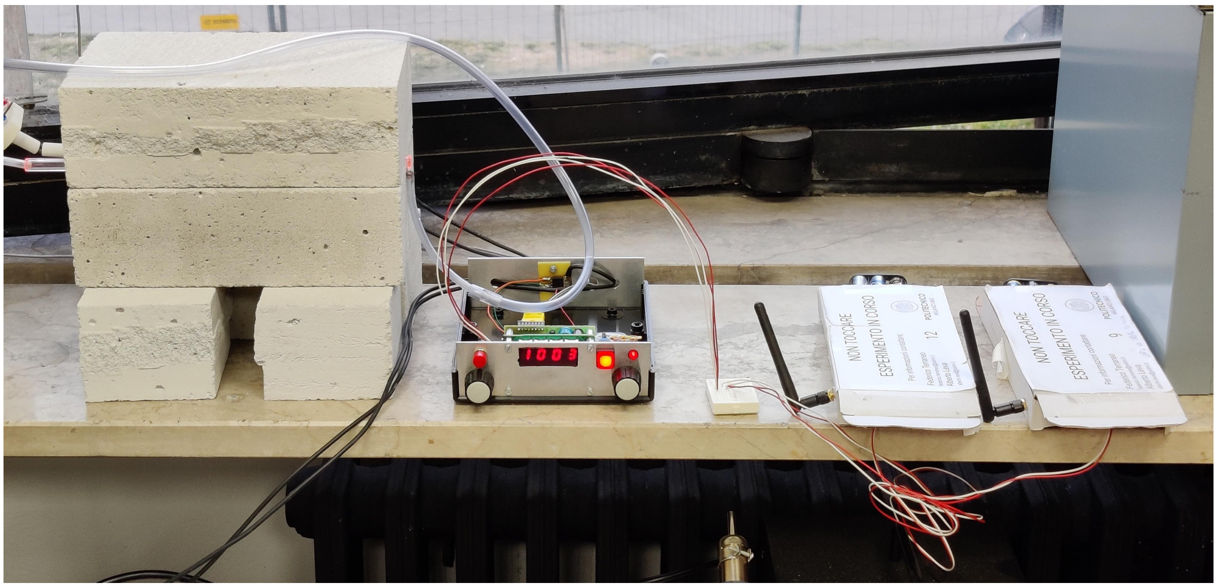

Figure 7.79: Experiment setup showing the furnace electronics and the sensor and actuator nodes, when stability was reached. Current temperature is shown on the 7-segment display.

Fig. 7.80 shows the complete temperature curve and the relative control action.
As it can be seen, the temperature setpoint of 1000 $°C$ was slightly overshot, with an error of 3.8% with respect to the set point. Indeed, the maximum reached temperature was 1038 $°C$. Overall, the controller reached the steady-state and the setpoint temperature in around 4 hours.
The overshoot is due to an underestimation of the process gain. The real gain is higher than the identified one and, so, during the real experiment a lower power (i.e. lower control action) is needed to reach the desired temperature, with respect to the one needed during simulations. Nevertheless, the controller was able to reach the setpoint without further oscillations and was tolerant to packets that were missed due to the distributed wireless setting.

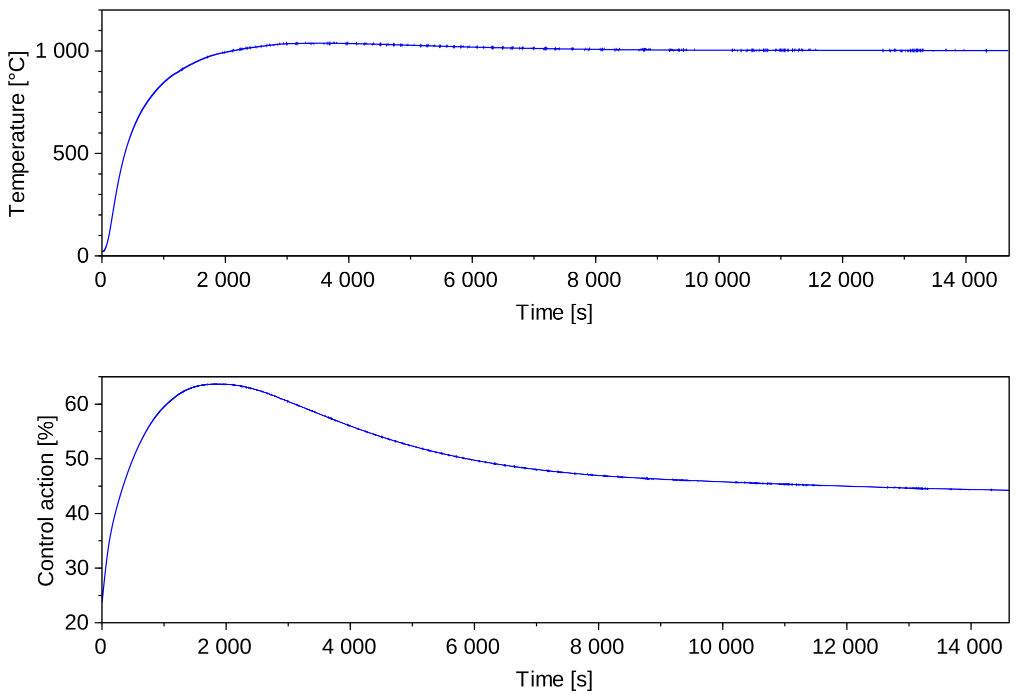

Figure 7.80: Temperature (top) and control action (bottom) plots during the distributed control loop experiment.

In terms of reliability, Tab. 7.9 shows the results. Due to the partial network observability, only the information that can be retrieved from the master node log is shown, so only information that is relative to the master node incoming streams.
The shown results are not the best possible reliability ones if compared to previously reported validation experiments, but they show that the control scheme is robust even if packet misses occur.

Stream	**Sent packets**	**Reliability Single**	**Reliability Double**	**Reliability Triple**
S-C	14786	94.72%	99.46%	99.73%
A-C	14704	94.65%	99.20%	99.48%

Table 7.9: Reliability of sensor-controller (S-C) and actuator-controller (A-C) streams during the control loop experiment.

Since the actuator replies to the controller (master) only after receiving a packet, if a packet on the stream controller-actuator is missed, the following packet on the stream actuator-controller is missed too. A packet sent by the actuator itself can be independently lost too. This means that $R_{C-A} \geq R_{A-C}$, where R_{C-A} and R_{A-C} represent the reliability on the controller-actuator and actuator-controller streams.

Finally, the end-to-end latency between sensor and actuator proved to achieve very good results. The achieved latency standard deviation is as low as 0.74 μs, while the measured jitter is 3.34 μs, representing the maximum latency deviation from the latency mean value. The complete measured latency is reported in Fig. 7.81. It may seem really noisy, but actually the maximum measured peak-to-peak difference is 4.62 μs over a set of almost 15 thousand packets. This is a better result than the ones presented in the validation experiments, in which all the nodes open a stream to the master node.

Average Latency	**Maximum Jitter**	**Standard Deviation**
1997.7783 ms	3.34 μs	0.74 μs

Table 7.10: Sensor to actuator latency statistics measured during the control loop experiment.

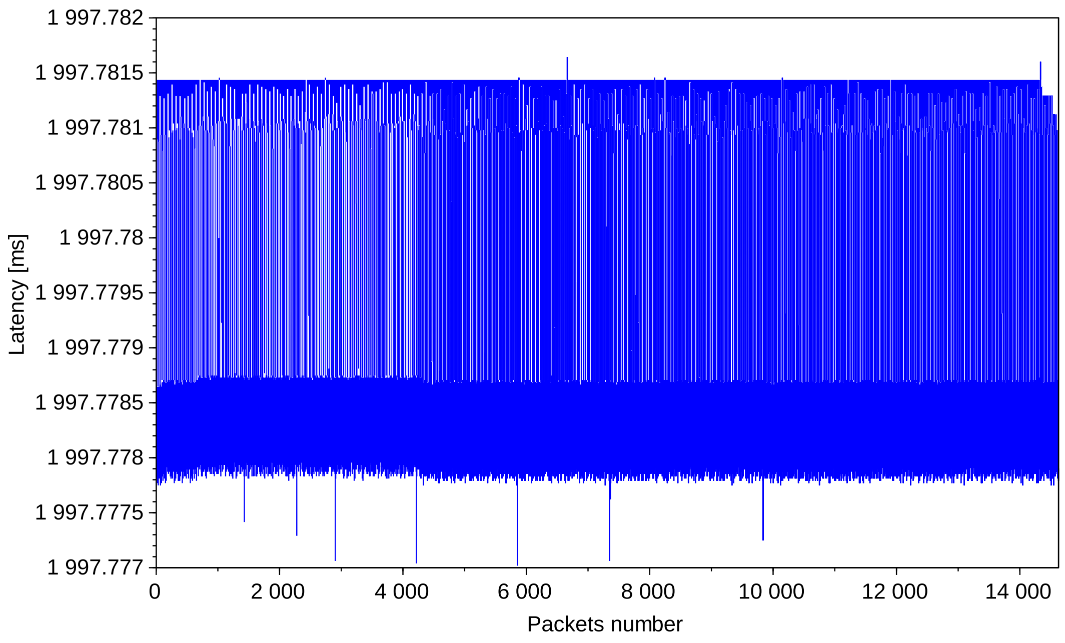

Figure 7.81: Measured latency from sensor to actuator during the distributed control loop experiment.

8 | Conclusions and Future Developments

This thesis presented two APIs for interfacing the application with the session layer of TDMH, a low-power wireless network stack designed for real-time applications. The two implementations offer the user different methods for synchronizing the application code with the underlying network stack according to the application needs, enforcing guarantees on the end-to-end latency bounds for every packet that flows through the network, not only at the physical layer but also up to the application one.
The design of the new APIs was driven by the problem of having, not only bounded, but also predictable latencies, which is a key aspect for real-time applications. The presented approach proved to be able to provide the required guarantees also having the application and the TDMH network stack executed on the same low-power device, without the need of higher-end hardware. The latency bounds provided by TDMH, thanks to the new implementation, do not depend on the streams period, but only on the current streams schedule and their difference is always constant. This makes the **latency always deterministic and predictable**.
Experiments were conducted using the existing WandStem nodes, which are equipped with a single-core microcontroller, and show that, in the worst-case scenario, a latency jitter in the order of one hundred microseconds is achieved, with a very low standard deviation and without any drawback on the network reliability provided by TDMH. Moreover, the highest jitter is experienced for very few packets, as the low standard deviation underlines. The distributed temperature control experiment that was presented also confirms that the provided latency guarantees and the high level of network reliability make TDMH suitable for real-time applications, such as industrial control, over a wireless network.

8.1. Future Developments

Imagining the usage of TDMH in a real industrial control setup, some other features may be needed. First of all the possibility of specifying a hierarchy among the network nodes, which has to be considered during the computation of the streams schedule. For example, assuming to have three nodes, a sensor, a controller and an actuator node, it may be necessary to enforce a chain among the streams that have the sensor as a source and the controller as a destination and the ones that go from the controller to the actuator. Then, it should be possible to indirectly specify the task assigned to each node (e.g. sampling a sensor) in the form of a **hierarchy among** existing **streams** and this knowledge can be used by the scheduler to fine-tune the schedules themselves.

Moreover, the current session layer implementation does not allow to have **bidirectional streams**. When opening a new stream, the application has to specify the required direction, which indicates whether data packets flow from the client to the server or vice-versa. Instead of using two distinct streams, bidirectional streams would simplify the communication setup among nodes in a more complex scenario in which network devices not only transmit or receive data but need to perform both tasks.

Bibliography

[1] IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC sublayer. *IEEE Std 802.15.4e-2012 (Amendment to IEEE Std 802.15.4-2011)*, pages 1–225, 2012. doi: 10.1109/IEEESTD.2012.6185525.

[2] 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 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pages 1–4379, 2021. doi: 10.1109/IEEESTD.2021.9363693.

[3] D. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon. Faster algorithms for frobenius numbers. *The Electronic Journal of Combinatorics*, 12(1), June 2005. URL `https://doi.org/10.37236/1924`.

[4] Bluetooth Special Interest Group. URL `https://www.bluetooth.com`. (Visited 2022-03-26).

[5] Connectivity Standards Alliance. URL `https://csa-iot.org/all-solutions/zigbee/`. (Visited 2022-03-26).

[6] D. De Guglielmo, S. Brienza, and G. Anastasi. IEEE 802.15.4e: A survey. *Computer Communications*, 88:1–24, 2016. ISSN 0140-3664. doi: https://doi.org/10.1016/j.comcom.2016.05.004. URL `https://www.sciencedirect.com/science/article/pii/S0140366416301980`.

[7] ESI Group. URL `https://www.scilab.org`. (Visited 2022-03-26).

[8] 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*, pages 73–84, 2011.

[9] R. D. Gomes, C. Benavente-Peces, I. E. Fonseca, and M. S. Alencar. Adaptive and Beacon-based multi-channel protocol for Industrial Wireless Sensor Net-

works. *Journal of Network and Computer Applications*, 132:22–39, 2019. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2019.01.025. URL `https://www.sciencedirect.com/science/article/pii/S1084804519300396`.

[10] IEEE Technical Committee on Real-Time Systems. Terminology and Notation. URL `https://cmte.ieee.org/tcrts/education/terminology-and-notation`. (Visited 2022-03-26).

[11] F. A. Izzo. A time deterministic communication stack for mesh networks, 2019. URL `http://hdl.handle.net/10589/149392`.

[12] R. Jacob, M. Zimmerling, P. Huang, J. Beutel, and L. Thiele. End-to-End Real-Time Guarantees in Wireless Cyber-Physical Systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 167–178, 2016. doi: 10.1109/RTSS.2016.025.

[13] W.-C. Jeong and J. Lee. Performance evaluation of IEEE 802.15.4e DSME MAC protocol for wireless sensor networks. In *2012 The First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT)*, pages 7–12, 2012. doi: 10.1109/ETSIoT.2012.6311258. URL `https://doi.org/10.1109/etsiot.2012.6311258`.

[14] S.-S. Joo, B.-S. Kim, J.-A. Jun, and C.-S. Pyo. Enhanced MAC for the bounded access delay. In *2010 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 423–424, 2010. doi: 10.1109/ICTC.2010.5674810.

[15] V. Mazzola. A secure transport layer for the TDMH network stack with key management, authentication, and encryption services, 2020. URL `http://hdl.handle.net/10589/166303`.

[16] OpenSim Ltd. OMNeT++. URL `https://www.omnetpp.org`. (Visited 2022-03-26).

[17] J. Polastre, J. Hill, and D. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, page 95–107, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138792. doi: 10.1145/1031495.1031508. URL `https://doi.org/10.1145/1031495.1031508`.

[18] P. Polidori. A time deterministic MAC protocol for low latency multi-hop wireless networks, 2017. URL `http://hdl.handle.net/10589/140120`.

[19] F. Sutton, M. Zimmerling, R. Da Forno, R. Lim, T. Gsell, G. Giannopoulou, F. Ferrari, J. Beutel, and L. Thiele. Bolt: A stateful processor interconnect.

In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, page 267–280, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336314. doi: 10.1145/2809695.2809706. URL `https://doi.org/10.1145/2809695.2809706`.

[20] F. Terraneo. WandStem: The next generation low-power sensor node. URL `https://miosix.org/wandstem.html`. (Visited 2022-03-26).

[21] F. Terraneo. Miosix kernel, 2008. URL `https://miosix.org`. (Visited 2022-03-26).

[22] F. Terraneo. TDMH: A time deterministic wireless network stack, 2020. URL `https://github.com/fedetft/tdmh`. (Visited 2022-03-26).

[23] 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*, pages 11–20, 2014. doi: 10.1109/RTSS.2014.14.

[24] F. Terraneo, P. Polidori, A. Leva, and W. Fornaciari. TDMH-MAC: Real-Time and Multi-hop in the Same Wireless MAC. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 277–287, 2018. doi: 10.1109/RTSS.2018.00044.

[25] F. Terraneo, F. A. Izzo, A. Leva, and W. Fornaciari. TDMH: a communication stack for real-time wireless mesh networks, 2020. URL `https://arxiv.org/abs/2006.03554`.

[26] Q. Wang, K. Jaffrès-Runser, Y. Xu, J.-L. Scharbarg, Z. An, and C. Fraboul. TDMA versus CSMA/CA for wireless multi-hop communications: A comparison for soft real-time networking. In *2016 IEEE World Conference on Factory Communication Systems (WFCS)*, pages 1–4, 2016. doi: 10.1109/WFCS.2016.7496512.

List of Figures

3.1 TDMH temporal organization. Transmission is divided into tiles. Tiles initial slots are occupied by control slots, either downlink (D) or uplink (U). 12
3.2 TDMH network stack overview. 16
3.3 Example of network topology with four nodes. The master is node 0. . . . 17
3.4 Multiple streams periods example. Stream *S1* has period 1 tile and *S2* period 2 tiles. 20
3.5 Different streams redundancy levels example. It shows both a scenario in which the redundant transmissions are consecutively scheduled and one in which they are not, for stream *S1*. 22

4.1 Undesired behavior in which write operations w_i are delayed by two periods. p_i represent ready for transmission packets, hold by the session layer buffer, and t_i the transmission slots, with period P. 31
4.2 Measured latency of packets sent over stream $1 \rightarrow 0$. 32
4.3 Detail of the measured latency of the first 110 packets sent over stream $1 \rightarrow 0$. 33
4.4 Desired application and session layer behavior. 34

5.1 Sequence diagram showing the send callback mechanism. 38
5.2 Sequence diagram showing the receive callback mechanism. 39
5.3 Undesired send callback execution behavior, which would delay the packet transmission. 40
5.4 Desired and implemented send callback behavior (transmission side). . . . 41
5.5 Desired and implemented receive callback behavior (reception side). 41
5.6 Desired behavior for stream transmission using the write/wait API. 42
5.7 Example of schedule containing streams with different periods and their relative advance. Required wake-up time instants are represented by an upwards blue arrow for stream *S1* and a red one for stream *S2*. Schedule length is two tiles. 46

5.8 Sequence diagram showing the interaction between application, session layer and dataphase when using the write/wait API. 47
5.9 Flow diagram showing the basic mechanism of the *StreamWakeupScheduler*. 49
5.10 *StreamWakeupScheduler* finite state machine. 50
5.11 Example of schedule in which the stream *S1* has to be woken up during the previous tile w.r.t. the transmission slot tile. 53
5.12 *StreamWakeupScheduler* `IDLE` state. 55
5.13 *StreamWakeupScheduler* `ACTIVE` state. 56
5.14 *StreamWakeupScheduler* `AWAITING ACTIVATION` state. 57
5.15 *compareQueues()* function used to get a reference to the queue from which the next element has to be taken, according to the current FSM state. . . . 58
5.16 Example in which $nextQueue_K$ and $nextQueue_{K+1}$ are empty. Upward blue and red arrows indicate wake-up actions that take place, respectively from streams *S1* and *S2* 59
5.17 Example in which a stream that appears also in the $nextQueue_K$ is removed from the schedule and a new one is added. Upward blue and red arrows indicate wakeup actions that take place, respectively from streams *S1* and *S2*. 59
5.18 Example in which a schedule is received after its own activation tile has already passed. 61
5.19 Undesired situation in which the data phase execution is anticipated before the *write* primitive. 65
5.20 Example in which the last redundant packet is received by the destination node. 66
5.21 Example in which the last redundant packet is missed by the destination node. 66
5.22 Dataphase correct behavior on the receiving side, including the $T_{tx,max}$ time slack. 68
5.23 Example that includes radio startup and crypto timings. Stream *S1* uses the *write/wait* API while stream *S2* uses the callbacks one. 69
5.24 Example in which the schedule expansion is executed across two downlink slots. 70
5.25 Sequence diagram showing the schedule expansion mechanism on the master node, after the rekeying already took place (*#rekeyingSlots* already elapsed). 73
5.26 Master schedule distribution finite state machine. 74

5.27 Sequence diagram showing the schedule expansion mechanism on the dynamic nodes, after the rekeying already took place (*#rekeyingSlots* already elapsed). 75
5.28 Dynamic schedule distribution finite state machine. 75

6.1 End-to-end latency lower bound example. 80
6.2 End-to-end latency lower bound example, in which stream *S1* transmission slots are not consecutive in the schedule. 81
6.3 End-to-end latency upper bound example when using the callbacks API. . 83
6.4 End-to-end latency upper bound example when using the callbacks API, in which stream *S1* transmission slots are not consecutive in the schedule. 83
6.5 End-to-end latency upper bound example when using the *write/wait* API. 85
6.6 End-to-end latency upper bound example when using the *write/wait* API, in which stream *S1* transmission slots are not consecutive in the schedule. 86
6.7 End-to-end latency upper bound example when using heterogeneous APIs (*write/wait* for transmission and callback for reception). 88
6.8 End-to-end latency upper bound example when using heterogeneous APIs (callback for transmission and *read* for reception). 89
6.9 Single hop packets measured latency using the callbacks API. 90
6.10 Single hop packets latency distribution using the callbacks API. 91
6.11 Single hop packets measured latency using the *write/wait* API. 92
6.12 Single hop packets latency distribution using the *write/wait* API. 92
6.13 Single hop packets measured latency using heterogeneous APIs. 93
6.14 Single hop packets latency distribution using heterogeneous APIs. 94
6.15 Single hop packets latency for stream S when re-schedulings take place. . . 95

7.1 *Line4* network topology. 101
7.2 *Star4* network topology. 101
7.3 *Kite* network topology. 101
7.4 *PartialMesh* network topology. 101
7.5 WandStem nodes placement at Building 21 of Politecnico di Milano. The master node is highlighted by an orange circle. 103
7.6 First callbacks API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones. 104
7.7 Latency for stream $1 \rightarrow 0$. 106
7.8 Latency for stream $2 \rightarrow 0$. 106
7.9 Latency for stream $3 \rightarrow 0$. 107

7.10 Latency for stream 4 → 0. 107
7.11 Latency for stream 5 → 0. 107
7.12 Latency for stream 6 → 0. 107
7.13 Latency for stream 7 → 0. 107
7.14 Latency for stream 8 → 0. 107
7.15 Latency for stream 9 → 0. 108
7.16 Latency for stream 10 → 0. 108
7.17 Latency for stream 11 → 0. 108
7.18 Latency for stream 12 → 0. 108
7.19 Latency for stream 13 → 0. 108
7.20 Second callbacks API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones. 109
7.21 Latency for stream 1 → 0. 111
7.22 Latency for stream 2 → 0. 111
7.23 Latency for stream 3 → 0. 112
7.24 Latency for stream 4 → 0. 112
7.25 Latency for stream 5 → 0. 112
7.26 Latency for stream 6 → 0. 112
7.27 Latency for stream 7 → 0. 112
7.28 Latency for stream 8 → 0. 112
7.29 Latency for stream 9 → 0. 113
7.30 Latency for stream 10 → 0. 113
7.31 Latency for stream 11 → 0. 113
7.32 Latency for stream 12 → 0. 113
7.33 Latency for stream 13 → 0. 113
7.34 First write/wait API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones. 114
7.35 Latency for stream 1 → 0. 116
7.36 Latency for stream 2 → 0. 116
7.37 Latency for stream 3 → 0. 117
7.38 Latency for stream 4 → 0. 117
7.39 Latency for stream 5 → 0. 117
7.40 Latency for stream 6 → 0. 117
7.41 Latency for stream 7 → 0. 117
7.42 Latency for stream 8 → 0. 117

7.43 Latency for stream 9 → 0. 118
7.44 Latency for stream 10 → 0. 118
7.45 Latency for stream 11 → 0. 118
7.46 Latency for stream 12 → 0. 118
7.47 Latency for stream 13 → 0. 118
7.48 Latency for stream 4 → 0 around re-scheduling. 119
7.49 Latency for stream 5 → 0 around re-scheduling. 119
7.50 Latency for stream 6 → 0 around re-scheduling. 119
7.51 Latency for stream 8 → 0 around re-scheduling. 119
7.52 Latency for stream 13 → 0 around re-scheduling. 119
7.53 Second write/wait API experiment network topology. The master node is highlighted by an orange circle. Dark blue indicate strong links, light blue represent weak ones. 120
7.54 Latency for stream 1 → 0. 122
7.55 Latency for stream 2 → 0. 122
7.56 Latency for stream 3 → 0. 123
7.57 Latency for stream 4 → 0. 123
7.58 Latency for stream 5 → 0. 123
7.59 Latency for stream 6 → 0. 123
7.60 Latency for stream 7 → 0. 123
7.61 Latency for stream 8 → 0. 123
7.62 Latency for stream 9 → 0. 124
7.63 Latency for stream 10 → 0. 124
7.64 Latency for stream 11 → 0. 124
7.65 Latency for stream 12 → 0. 124
7.66 Latency for stream 13 → 0. 124
7.67 Latency for stream 1 → 0 around re-scheduling. 125
7.68 Latency for stream 3 → 0 around re-scheduling. 125
7.69 Latency for stream 4 → 0 around re-scheduling. 125
7.70 Latency for stream 7 → 0 around re-scheduling. 125
7.71 Latency for stream 8 → 0 around re-scheduling. 125
7.72 Latency for stream 11 → 0 around re-scheduling. 125
7.73 Latency for stream 12 → 0 around re-scheduling. 126
7.74 Latency for stream 13 → 0 around re-scheduling. 126

7.75 Interactions among sensor, controller and actuator nodes. Solid line arrows represent streams, through which the sampling timestamp (T), process variable (PV) and the control variable (CV) are exchanged. The actuator also transmits the sensor-actuator latency (L_{S-A}), back to the master. Interactions with the furnace are also shown: dashed arrows represent serial port communication. 127
7.76 Feedback control loop representation, showing the set point $\bar{y}$, the error e, the control variable u and the process variable y. 129
7.77 Control loop experiment network setup. The controller (0), sensor (12) and actuator (9) nodes are underlined in orange. The furnace (F) is identified by the yellow circle. Strong wireless links are represented by dark blue lines, while light blue lines indicate the weak ones. 131
7.78 Experiment setup showing the furnace electronics and the sensor and actuator nodes, during the temperature ascent. Current temperature is shown on the 7-segment display. 131
7.79 Experiment setup showing the furnace electronics and the sensor and actuator nodes, when stability was reached. Current temperature is shown on the 7-segment display. 132
7.80 Temperature (top) and control action (bottom) plots during the distributed control loop experiment. 133
7.81 Measured latency from sensor to actuator during the distributed control loop experiment. 134

List of Tables

2.1 TDMH features compared to other IEEE 802.15.4 protocols. 7

3.1 Example of scheduling for stream $3 \to 0$. 17

4.1 Latency statistics for stream $1 \to 0$ when using the current session layer implementation. 33

5.1 Resulting streams wake-up list from the final implementation. It contains the wake-up times of all the streams occurrences in the schedule, excluding redundant slots. 46
5.2 Resulting streams wake-up list from the first implementation. It contains only the wake-up time of first streams appearance in the schedule, excluding redundant slots. 46
5.3 Streams wake-up list containing also the downlink slots end time, referred to example in Fig. 5.7. 51

6.1 Single hop packets measured latency statistics using the callbacks API. . . 91
6.2 Single hop packets latency bounds using the callbacks API. 91
6.3 Single hop packets measured latency statistics using the *write/wait* API. . 93
6.4 Single hop packets latency bounds using the *write/wait* API. 93
6.5 Single hop packets measured latency statistics using heterogeneous APIs. . 94
6.6 Single hop packets latency bounds using heterogeneous APIs. 94
6.7 Example of schedule in which stream S transmits during three consecutive slots. 96
6.8 Example of schedule in which stream S transmissions are interleaved by the uplink phase slots. 96
6.9 Example of schedule in which stream $1 \to 0$ transmissions path to the receiver node is two-hops long and passes through node 2. 97

7.1 Stream seliability during the first callbacks API experiment. 105
7.2 Streams latency statistics during the first callbacks API experiment. 106

7.3 Streams reliability during the second callbacks API experiment. 110
7.4 Streams latency statistics during the first callbacks API experiment. 111
7.5 Streams reliability during the first write/wait API experiment. 115
7.6 Streams latency statistics during the first *write/wait* API experiment. . . . 116
7.7 Streams reliability during the second write/wait API experiment. 121
7.8 Streams latency statistics during the second *write/wait* API experiment. . 122
7.9 Reliability of sensor-controller (S-C) and actuator-controller (A-C) streams during the control loop experiment. 133
7.10 Sensor to actuator latency statistics measured during the control loop experiment. 134

Acknowledgements

I'd like to thank both my advisor and co-advisor, Federico Terraneo and professor William Fornaciari, for allowing me to work on this great project. I'm especially grateful to Federico Terraneo for assisting me during the entire thesis process, teaching me countless things, a great part of which are not even related to the thesis itself. A different type of gratitude goes to my family, my friends and my girlfriend Patricia, who indirectly shared with me the journey at Politecnico di Milano, without ever ceasing to support me and contributing to alleviate the encountered difficulties. I wish to thank also all my classmates with whom I spent most of the time throughout these long years, during lessons, projects and challenges, and especially Davide and Nicolò for pushing me to always do better. Finally, a special thanks goes to all the members of the Skyward Experimental Rocketry students association, who shared with me frustrations and satisfactions, and without whom the university experience would not have been the same.