CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Integrating A^2 with TSCH:
Bringing Network-Wide Agreement into a Recent Standard for the Internet of Things

Master's thesis in Computer System and Networks

Johan Jinton

Mikael Larsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Integrating A^2 with TSCH

Bringing Network-Wide Agreement into a Recent Standard for the Internet of Things

Johan Jinton
Mikael Larsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Integrating A² with TSCH:
Bringing Network-Wide Agreement into a Recent Standard for the Internet of Things

Johan Jinton
Mikael Larsson

Supervisor: Olaf Landsiedel, Department of Computer Science and Engineering

Examiner: Magnus Almgren, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Abstract

The vision of Internet of Things (IoT) as we know it has emerged immensely in the last couple of years, as the ever-growing demand for information continuously keeps increasing. A substantial contributing factor to the evolved vision of IoT is due to the convergence of multiple technologies including ubiquitous wireless communication and commodity sensors, paving the way for Wireless Sensor Networks (WSNs). The majority of WSNs constitute formations of small embedded, battery-driven sensor nodes (motes), wirelessly exchanging sensor information related to environmental conditions such as temperature, humidity, pollution levels and so on. In today's society, WSNs are omnipresent in many different areas, stretching from the vehicle industry to areas concerning natural disaster prevention. In order to utilize the full potential of WSNs, motes need to reach network-wide agreement (consensus) in order for the whole network to agree on the outcome of a specific action. Consensus is a fundamental concept in distributed systems, but has received little attention in the low-powered wireless context due to the resource restricted constraints of such systems. Thus, the goal of this thesis is to enforce network-wide consensus on top of the IEEE 802.15.4e MAC-protocol standard, Time Slotted Channel Hopping (TSCH); used in research and industrial sensor networks. We present a new system, A^2-with-TSCH, integrating the already existing A^2 (Agreement in the Air) system to provide consensus primitives on top of the highly reliable MAC-protocol TSCH. We evaluate our A^2-with-TSCH system in the Contiki simulator Cooja and on the public testbed FlockLab. The FlockLab evaluation shows that our A^2-with-TSCH system reliably reinforces network-wide consensus, while achieving similar performance metrics as the original A^2-Synchrotron system.

Keywords: Internet of Things (IoT), Wireless Sensor Networks (WSN), motes, consensus, Time Slotted Channel Hopping (TSCH), Agreement in the Air (A^2).

Acknowledgements

We would like to express our deepest gratitude to our supervisor Olaf Landsiedel, for providing us with valuable feedback and guidance throughout this project. His commitment and overall knowledge has helped us immensely. In addition, we would also like to thank Beshr Al Nahas for his technical expertise and for taking his time to answer our questions. Finally, we would like to thank our examiner Magnus Almgren for providing feedback on the report.

Johan Jinton, Gothenburg, June 2018

Mikael Larsson, Gothenburg, June 2018

Contents

List of Figures

List of Tables

1
Introduction

Over the last few years, there has been an ever growing trend of having more and more devices exchanging data with each other, either directly or over the internet, without the help of human interaction; this is known as the Internet of Things (IoT). A key building block of the recent IoT is "Wireless Sensor Networks" (WSNs) [20], which are networks of small embedded computing devices with the capability to wirelessly send and receive measured sensor information. These embedded devices are often small and have limited CPU, memory and power resources. The aforementioned restrictions introduce new challenges of how to efficiently reduce power consumption, as well as how to implement reliable communication in the highly dynamic nature that comes with wireless networks; where failing nodes and signal interference is the rule rather than the exception.

However, in many applications it does not simply suffice to reliably exchange information with neighboring nodes, but the nodes also have to unanimously agree on the outcome of a specific action in the presence of failing nodes. This condition is called consensus, and is a well-known problem in the area of distributed computing.

1.1 Motivation

Wireless sensor networks has been and still is a fast growing central topic, extensively researched especially during the past two decades [27]. The process of monitoring and documenting the physical conditions of the environment through sensors and then gather the information for processing is widely adopted throughout the industry and different research fields; as sensors can measure a myriad of environmental conditions such as humidity, temperature, sound, pressure, wind etc.

The potential for WSNs are huge and we are at the brink of what is strongly believed to be a new revolution where WSNs are incorporated into so called cyber-physical systems (CPS), to further bridge the way for how we humans interact between the physical and the cyber world [22]. According to Rajkumar et al. [22]: “Cyber-physical systems (CPS) are physical and engineered systems whose operations are monitored, coordinated, controlled and integrated by a computing and communication cor”. Examples of such systems are smart grids, nano-robotics, autonomous vehicles and automatic pilot avionics.

Applications of wireless sensor networks and cyber-physical systems are heavily dependant on reliable communication for devices to be able to properly exchange information between one another. However, some applications also need to enforce network-wide agreement (consensus) in order for all devices to reliably agree on the outcome of a specific action. One future example, when consensus will be of utmost importance, is when autonomous self-driving vehicles have to agree on which car is first to enter an upcoming crossing. If the agreement is not unanimous and two cars think they are allowed to enter the crossing at the same time, the consequences can be disastrous.

A^2 (Agreement in the Air) introduced by Landsiedel et al. [1] is an upper layer protocol built on top of the media access control (MAC) layer protocol, Chaos [17]. A^2 provides group membership and multiple different consensus primitives such as the protocols two-phase and three-phase commit (2PC and 3PC).

TSCH [26] (Time Slotted Channel Hopping) is another MAC-layer protocol for ensuring low-latency, energy efficient and reliable communication in low-power and lossy networks (LLNs), such as wireless sensor networks. TSCH is an IEEE 802.15.4e standard and as stated in the RFC 7554 [26]: "TSCH was designed to allow IEEE 802.15.4 devices to support a wide range of applications including, but not limited to, industrial ones".

Consensus protocols in wired networks have been extensively researched by the research community for quite some time. However, to the best of our knowledge there is currently no support for achieving networks-wide consensus in any of the MAC-layer standards corresponding to wireless sensor networks; consequently, motivating the work of this thesis.

1.2 Aim & Key Results

The main goal of this thesis is to provide network-wide consensus to applications working on top of TSCH. More formally, this thesis describes how we develop a new system, A^2-with-TSCH, by integrating the A^2 protocols Max and 2PC on top of the MAC-protocol TSCH. The second goal of this thesis is to see whether A^2-with-TSCH is able to achieve similar performance and power consumption metrics as the original A^2-Synchrotron [1] system.

The results from our evaluation show that we consistently reach consensus for both the Max and 2PC protocol. We also achieve similar performance metrics as A^2-Synchrotron, when evaluating on a public testbed.

1.3 Limitations

In order to achieve the goal of this thesis, we specify the following limitations. First of all, we try to avoid making major changes to the current TSCH implementation

as we rely on TSCH for handling all low-level logic corresponding to the MAC layer. Additionally, we also try to make minimal modifications to the A^2 protocols, Max and 2PC, as our goal is to port these protocols on top of TSCH. Even though fault-tolerance is an important aspect of wireless sensor networks, A^2-with-TSCH will not be able to accommodate for failing nodes. Moreover, we limit our implementation to static topologies meaning that our system will not support runtime changes to the network topology. To clarify, all nodes need to be aware of each other at compile time, thereby restricting new nodes from joining the network.

1.4 Contributions

Below, we present a summary of our main contributions:

- We develop a new system, A^2-with-TSCH, where A^2 protocols are able to run on top of TSCH.
- We develop a middle-layer interface, Bridge.
- We provide application support for running the following A^2 protocols: Max and 2PC.
- We provide support for running multiple A^2 protocols concurrently.
- We develop a solution to remedy a synchronization issue in the current TSCH implementation.

1.5 Report Outline

This chapter gives a brief introduction to this thesis by motivating and presenting different problems within the research area, together with the limitations and contributions of this thesis. Chapter 2 contains background specific information corresponding to protocols and fundamental concepts, concerning our project. In Chapter 3 we present related work, describing synchronous MAC-protocols and consensus protocols for wireless sensor networks. Chapter 4 and 5 depict our system architecture design along with corresponding implementation specific details. In the two last chapters, we evaluate our A^2-with-TSCH system and conclude the work of our thesis by discussing future work and questions regarding ethics and sustainability.

2
Background

This chapter contains general information about important protocols and concepts, highly relevant to this project. The first section introduces the reader to what wireless sensor networks (WSNs) are, and the following section explains *consensus*, a central concept within the distributed systems field. In the consensus section, we also describe two-phase commit (2PC), a consensus protocol providing transactional guarantees. Furthermore, we describe the MAC protocols Time Slotted Channel Hopping (TSCH) and Chaos, together with the Agreement in the Air (A^2) system. Lastly we explain the low-power operating system Contiki, the simulation tool Cooja and the hardware mote Tmote Sky.

2.1 Wireless Sensor Networks

A wireless sensor network (WSN) is a group of dedicated sensor nodes, also known as motes, operating together by monitoring the physical conditions of the environment and sharing the information with neighboring motes [28]. There is also at least one mote in a WSN which differs from the other motes and this mote is called the sink or base station. The main goal for this mote is to process the shared data obtained from the other motes in the network. Since all the data sooner or later will arrive at the sink, the location of the sink in the network is of high importance. Depending on where the sink is placed it will affect the energy consumption and thereby also decreasing or increasing the lifetime of a WSN.

The nodes in a WSN are small, consisting only of a radio transceiver, micro controller and an electric circuit serving as an interface between the sensors and an energy source. Together with the small size, multiple other constraints also affect the WSN nodes and some examples are: the limited amount of energy, short communication range, low bandwidth and limited processing and storage. All these above mentioned constraints make WSNs more complicated than traditional networks and for instance restricts the operating system as well as the applications running on the motes to be both memory and energy efficient, thus also limiting the performance.

The popularity for WSNs skyrocketed with the convergence of multiple technologies including, omnipresent wireless communication, commodity sensors, embedded systems, etc., thus contributing to the evolved vision of what is known as the Internet of Things (IoT). Wireless sensor networks helps to further bridge the gap between the physical and the digital world, making it possible to take different actions based

on environmental conditions. Examples of different environmental measurements which can be gathered by the sensors are for instance wind, humidity, temperature, pollution levels and sound. Further examples of applications which use sensors networks are: military target tracking systems, different kinds of surveillance applications and biomedical health monitoring. The mentioned applications are only a handful and the list of possible applications can be enormous, thus the potential for WSNs are huge.

2.2 Consensus

Consensus is a central and a very well-known problem within the field of distributed computing. Reaching consensus within a network of nodes (processes), suggests that the overall system has to reliably agree on a single data value which is needed during computation in the presence of a number of faulty nodes. This property is fundamental for protocols where an action has to be taken unanimously by all the working nodes in a network, otherwise generating inconsistencies which is likely to provoke unknown or unwanted behaviors.

A consensus protocol can be divided into three main correctness properties which have to be fulfilled in order to be considered formally correct, namely: agreement, integrity and termination [5]. The agreement property simply means that all participating nodes have to agree on the same value and once the decided value has been chosen, it is final. Integrity is satisfied if the agreement value decided upon, is one of the proposed values originating from one of the participating nodes; note that the strictness of the integrity property may differ depending on the application. Termination implies that all nodes decides upon its agreement value within a finite number of execution steps.

Below we describe a fundamental consensus protocol, 2PC [14], which has a widespread use in applications where data needs to be shared under transactional guarantees.

2.2.1 Two-Phase Commit Protocol (2PC)

As the name suggests, the protocol consists of two phases: Voting phase and Commit/Abort phase. If the protocol completes without any failing nodes and also without any loss of messages, the protocol behaves as follows. The coordinator initiates the protocol and enters its voting phase by broadcasting a proposal message (query to commit) to each of the nodes (cohorts) in the network, and then waits until each vote has been received. Each cohort executes the transaction up until the point when they are asked to commit. Based on whether their actions succeeded, all cohorts reply the coordinator with an agreement message, which is either a "Yes to commit"-message or an "abort" message if something went wrong i.e., they are unable to commit.

If the coordinator receives a unanimous "Yes to commit"-message from every cohort, the coordinator initiates phase two by broadcasting a "commit"-message to everyone in the network. Each cohort receiving the commit message, completes their transaction and sends back an acknowledgement message to the coordinator. As soon as the coordinator has received an acknowledgement from each cohort, the coordinator will complete its own transaction consequently ending a successful run of the protocol. However, if any of the cohorts is unable to complete their transaction from phase one, the coordinator will receive at least one abort message. Instead of sending a commit message, the coordinator will instead broadcast "abort" to every node. All cohorts receiving abort, will undo their transaction and again reply with an acknowledgement to the coordinator. The coordinator also decides to undo its transaction, as soon as every acknowledgement has been received.

The above explanation of the protocol behavior only applies under the assumption that none of the nodes crash, behaves unexpectedly nor when there are no loss of messages. The greatest disadvantage of the two-phase commit protocol is that it is blocking, meaning that if the coordinator fails permanently while awaiting agreement votes from the cohorts during phase one, some nodes will never resolve their transactions. This is because cohorts will block resources until the reception of a commit/abort message from the coordinator which will never happen if the coordinator is permanently down.

2.3 Time Slotted Channel Hopping (TSCH)

Time Slotted Channel Hopping or Time Synchronized Channel Hopping, abbreviated as TSCH, is an RFC specification for how to develop a highly reliable Media Access Control (MAC) protocol for Low-power and Lossy Networks (LLNs). TSCH was proposed in 2012 as an amendment to the limitations of the already existing MAC layer portion of the IEEE 802.15.4 standard (2011) and was later enrolled into 802.15.4e, in 2015 [26]. We divide this section into two parts, where the first part describes the RFC 7554 and the second part explains details regarding an open-source implementation of TSCH, developed in the operating system Contiki-NG [3].

2.3.1 TSCH: RFC Specification

As the name suggests, TSCH sets out to provide both time synchronization and channel hopping to attain for low-power operations and to achieve high reliability in the dynamic nature of wireless communication. The protocol employs time slotting, where each node maintains a local schedule consisting of timeslots instructing each node when to receive, transmit or sleep. By doing so, each node knows exactly when to turn its radio on or off, consequently preserving as much of the battery lifetime as possible.

Each node is equipped with a clock to keep track of time, but in order to keep tight time synchronization, the nodes have to periodically resynchronize; due to the

fact that the clocks at the different nodes drift with respect to one another. The periodical resynchronization phase involves each node assigning itself a dedicated neighbor, time-source neighbor, to which each node passes its network time. The adapting nodes then adjust their clocks by calculating the difference in time between the expected arrival of the timing packet and packet's actual arrival. The timing information is added to all data packets that are exchanged, meaning that any node can resynchronize whenever it receives a packet from its time-source neighbor.

The channel hopping portion of the protocol benefits both in preserving battery life as well as avoiding contention, when multiple nodes want to transmit data at the same time. If multiple nearby nodes transmit data at the same time using the same frequency, interference is likely to disrupt the ongoing transmissions consequently draining the battery and also obstructing nodes from receiving data. To accommodate for the aforementioned problem, each local schedule also incorporates a slot offset together with a channel offset, for each scheduled timeslot. When node A is scheduled to transmit data to node B in a specific time slot, node B also has to be scheduled to receive data in the exact same slot using the same channel offset as node A. However, timeslots can also be shared, meaning that neighbouring nodes are allowed to transmit data at the same time using the same channel frequency; TSCH defines a backoff algorithm to avoid contention in these shared slots.

Nodes which are already a part of the network, periodically send out Enhanced Beaconing (EB) packets on all frequencies, to announce to other nodes about the presence of the network. These packets contain essential information about the network, used by newcomer nodes in order to properly associate to the existing network; an example of such information is the length of a timeslot. Nodes wishing to associate simply listens to all frequencies, one at a time, until it encounters one or more EB packets. The information contained in the EB packet is then used to synchronize to the network.

2.3.2 TSCH: A MAC Protocol of Contiki-NG

In Contiki-NG, nodes wanting to associate to a network listens for EB-packets from nodes already connected to the existing network. Once an EB-packet is received, the node joining the network will select its time-source neighbor to be the node it received the EB-packet from. This process continues until all nodes are associated to the network, consequently forming a directed acyclic graph (DAG). As illustrated in Figure 2.1, all nodes except the root node (initiator) will have a single time-source parent from which they frequently receive synchronization information. As every node has a directed path to the initiator, the nodes will stay relatively synchronized to the initiator and as a consequence also stay relatively synchronized to all the other nodes in the network.

Nodes are allowed to change time-source, whenever they experience a bad connection to their existing time-source. When a node receives a packet from its time-source, it resets a timer keeping track of the time since the node last received synchroniza-

tion information. If the timer expires, nodes start sending keep-alive packets (KA packets) indicating that they need immediate response from their time-source. If a time-source fails to acknowledge any of the KA packets, nodes will appoint a backup node as their new time-source i.e., the last node they received an EB-packet from.

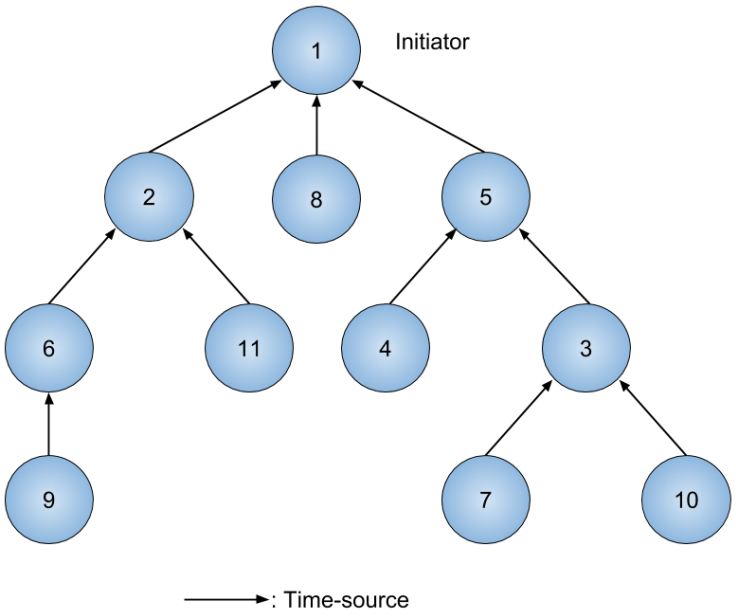


Figure 2.1: *An example of a directed acyclic graph, illustrating how TSCH maintains relative synchronization between all nodes.*

2.4 Chaos

Chaos, introduced by Landsiedel et al. [17], is another MAC layer protocol providing all-to-all data sharing in low-power wireless networks. Communication in Chaos is based on synchronous transmissions where data is incrementally merged and shared among all nodes. Examples of applications provided by Chaos are: aggregate functions, network-wide agreement (consensus), reliable data dissemination and multiple communication patterns.

In Chaos, synchronous transmission is key in order to maintain efficiency. Nodes that want to transmit data, simply flood the network in a synchronous manner and other nodes within communication range have a high probability of receiving that data, due to the capture effect [18]. Upon reception, nodes combine the received data with its own local data and continues to, again, transmit the merged data synchronously until every node share the same data. Every node in the Chaos network maintains a local clock as a reference for when to turn on the radio and initiate synchronous transmission; this happens during a Chaos slot which always has the same duration. Every Chaos slot has to be tightly synchronized at all nodes, so that every node is able to participate in the communication. As a consequence of

having tightly synchronized slots, all nodes have to participate in a network-wide time synchronization, due to the fact that clocks at different nodes drift with respect to each other. Each packet header contains a 1-byte value, initially set to 1 by the initiator which is incremented each time a packet is relayed by some other node. A node can then precisely compute the beginning of a slot start, based on the received slot counter and an estimation time of the slot duration.

Data packets that are to be transmitted to other nodes in Chaos, consist of two parts: flags and payload. A flag is just a 1-bit value correlating to a node in the network, thus the number of flags that are maintained at each node and sent to other nodes corresponds to the total number of nodes currently in the network. The flags serve as a way for each node to keep track of which of the other nodes have participated to the resulting payload of an arriving packet. When a packet is received, each node starts by inspecting the flags of that message to see if there is any new information. If the newly received packet has any of the flags set to 1, which has not already been documented in the local flags kept by the receiving node, then there is new information to consider. If this is the case, the node starts by updating its own flags, inspects the payload of the message and finally combines its own data with the newly received payload, using the merge operator; the merge operator is defined by the Chaos users and can for example be max(a,b), returning the largest value of a and b.

2.5 Agreement in the Air (A^2)

Agreement in the Air (A^2) is an upper layer protocol developed by Al Nahas et al. [1] and is built on top of a new transmission kernel, Synchrotron. Synchrotron extends on Chaos and provides extra functionalities such as frequency hopping, multiple parallel channels and improved MAC layer security. It also provides a more advanced high-precision synchronization timer by utilizing a technique called Virtual High-resolution Time (VHT). This extended MAC layer provides a reliable base layer which gives the A^2 protocol a solid foundation to extend on.

The main task of A^2 is to provide advanced network-wide communication primitives, with the main goal to achieve consensus and group membership. It inherits some basic communication primitives from Chaos known as disseminate, collect and aggregate. These primitives are then used to develop more advanced protocols such as Vote, 2PC and 3PC. The Vote primitive is a key functionality of A^2 where nodes vote for or against a proposal given by a coordinator. The primitive is also used by the nodes to agree on certain values such as an ID of an encryption key or the specific channel hopping sequence that should be used when communicating.

As previously mentioned, A^2 also supports 2PC and 3PC which are built on top of the network-wide voting primitive. The 2PC protocol consists of two phases where in the first phase a voting is performed and in the second phase a coordinator disseminates the result of the vote to all nodes that participated. The result broadcasted

by the coordinator is either commit if all nodes agreed, otherwise abort. The 3PC protocol is an extension of the 2PC protocol and further provides a third phase, preventing failing nodes from blocking the protocol; the main weakness of the 2PC protocol.

In addition to the protocols Vote, 2PC and 3PC, A^2 also provides group membership services such as a join operation and a leave operation. These primitives are essential when the network is dynamic and nodes can leave and join the network at any time. The join primitive consists of informing all nodes in the network that a new node is joining. This is done in two phases, where in the first phase the coordinator sends a message to collect information from the nodes that wants to join the network. If a node wants to join the network it responds with its node ID. In the second phase the coordinator sends a new message, informing all nodes about the new topology change. The leave primitive is invoked if a node is inactive and not participating in the network communication. The excluding of the inactive node is done by the coordinator which sends a new message, informing all nodes in the network that the topology has changed.

2.6 Contiki

Contiki [9] is an operating system developed for low-power micro-controllers. These micro-controller devices are in a very constrained environment, for instance, the size of the memory is often small and the CPU capacity is very limited. Contiki is perfect for these types of constrained environments since it is a lightweight operating system.

Contiki is built upon an event driven system to reduce the memory usage while still supporting a multithreaded environment. Usually each thread needs to have its own memory space preallocated on the stack which often leads to overprovisioning. Another common problem is that the memory allocated to a specific thread can not be shared with another thread. The allocated memory is hence only available to the thread which initially allocated the memory. Contiki solves these above mentioned problems by implementing processes as event handlers, by doing so all processes use the same stack and share the same memory resources consequently lowering the total memory usage. To further decrease the memory usage, Contiki also makes use of something called protothreads which is a special thread requiring less memory overhead than ordinary threads. The protothread is also commonly used among event driven operating system to simplify programming by reducing the abstraction of the complex state machine and providing a blocking wait condition to reduce the number of state machines [11].

In Contiki, the program code is executed in a so called cooperative mode, meaning that the code runs sequentially with respect to other cooperative code. The cooperative code is running without interruption, which makes it impossible to invoke any sort of interrupt handler. Since interrupts are essential for preempting execution of program code and to properly react on external events, Contiki also supports another mode called preemptive mode. Code that runs in the preemptive context

can at any time stop the cooperative code from executing, giving Contiki support for preemptive multi-threading on top of an event driven kernel.

A common misconception regarding Contiki is that it provides some sort of power saving capability. According to Dunkels et al. [9] the Contiki kernel does not contain any type of explicit power saving abstractions, but instead lets the application and the programmer provide this functionality if needed. For instance the kernel provides access to a so called event queue which can be used to check if any process is to be scheduled, if this is not the case, the system can be put into hibernation.

2.7 Cooja

Cooja [21] is a Java-based network simulator specifically designed for wireless sensor networks, and is shipped together with the Contiki operating system. Cooja makes the process of development, debugging and evaluation less of a tedious task, as the compiled code does not have to be transferred into multiple physical hardware motes every time the code is to be tested. Cooja also supports sensor nodes where each of them can be of a different type, differing not only on the software-level but also the hardware-level, therefore allowing simulations of heterogeneous networks.

Simulated motes in Cooja have three basic properties, namely the node type, its data memory and the hardware peripherals. Two or more nodes sharing the same node type, implies that they are initialized with the same data memory and also run the same program code on the same hardware peripherals; during execution, however, the nodes' data memory will differ due to e.g. external inputs.

Cooja is a cross-level simulator meaning that it allows for simultaneous simulations at the network level, the operating systems level, as well as the machine code instruction set level. When designing and implementing, for example, routing protocols in the network level, the most significant factors might be the radio device as well as the medium, and not the actual hardware running the protocol; hence the advantage of having a cross-level division.

Another great feature of Cooja is that simulation setups and entire simulation states can be saved, to later be restored at a different time with all simulation specific parameters unaltered.

2.8 Tmote Sky Board

Tmote Sky [4] is a low-power wireless module, adopted in settings such as sensor networks and monitoring applications. The board employs industry standards like USB and IEEE 802.15.4, allowing it to interoperate smoothly with other devices. By integrating the industry standards together with a humidity, temperature, and light sensor as well as flexible hardware peripherals, the Tmote Sky has a widespread use in several mesh networking applications. Tmote sky is a successor and replacement

to Moteiv's popular TelosB design [15].

The ultra low-power operations of the Tmote sky are due to the Texas Instrument MSP430 micro-controller (8MHz) which features 10kB RAM, 48kB flash and 128B of information storage. Other examples of some key characteristics of the Tmote Sky are: ultra low current consumption, fast wakeup from sleep (< 6 µs), programming and data collection via USB and an integrated onboard antenna reaching up to 50m indoors / 125m outdoors.

3
Related Work

We divide this chapter into two different parts. The first part introduces the synchronous MAC protocol Glossy [13], pioneering the field of synchronous transmissions in WSNs. This is followed by a communication protocol built on top of Glossy known as LWB [12], along with a comparison between Glossy and TSCH. The second part analyzes several consensus protocols in the context of WSNs and concludes with a discussion comparing their main differences.

3.1 Synchronous Communication Protocols

Synchronous communication protocols transmit and receive data packets at specific time intervals. This type of synchronous communication is mostly suited for so called duty-cycled networks, where nodes are scheduled to synchronously turn on the radio at the same point in time to participate in communication, and then turn off the radio to perform post processing. Below we describe Glossy [13], one of the first synchronous MAC protocols which paved the way for further research in the field of synchronous communication in WSNs. We continue by explaining LWB [12], a communication protocol working on top of Glossy. Multiple other protocols have been developed on top of Glossy with the main focus on data collection, examples of these are Splash [6] and Choco [23]; however, these will not be mentioned further.

3.1.1 Glossy

Glossy [13] is a synchronous communication protocol released in 2011, which pioneered the field of synchronous transmissions for WSNs. The foundation of Glossy builds upon its flooding architecture which exploits constructive interference for fast network flooding, and as a consequence also achieves implicit time synchronization. Glossy derives a timing requirement to establish constructive interference (super position) with a high probability, for multiple concurrent transmissions of the same packet; thus allowing a receiver to properly decode the packet even in the absence of capture effects.

Signal interference occurs when spatially dense nodes transmit data concurrently with the same frequency, as signals are likely to overlap in time and space. Interference can be destructive, meaning that when multiple signals meet they interfere in a way which prevents the receiver from properly detecting the overlapping signals. Glossy, on the other hand, instead uses interference to its advantage by exploiting

constructive interference; this means that the receiver is able to properly demodulate the superpositioned baseband signals generated by multiple transmitters.

To accommodate for synchronous transmissions, Glossy exploits its flooding communication primitive to implicitly establish time synchronization. A 1-byte relay-counter field is incorporated into each packet, starting with the value 0 in the initial packet sent by the initiator. Once the initial packet has been received by the neighboring nodes, the relay-counter is incremented by 1 before relaying the packet to other nodes. Consequently, the relay-counter works as a way for the nodes to infer how many times a packet has been relayed. Furthermore, this field is used to define the length of a slot (T_{slot}) as the time between the start of a packet transmission with relay-counter set to 0 and the consecutive packet transmission with relay-counter set to 1. Each node locally estimates T_{slot} by using the timestamps observed at different radio interrupts. The time when the initiator started the protocol, known as the synchronization reference time, is then calculated at every node based on the relay-counter and the local estimate of T_{slot}.

3.1.2 LWB: Low-Power Wireless Bus

LWB [12] is a communication protocol built on top of Glossy, with its main purpose to simplify the architecture of low-power wireless systems by turning multi-hop networks into a shared communication bus. This is done by utilizing the functionalities provided by Glossy such as network flooding, time synchronization and high reliability. LWB supports multiple traffic patterns, is resilient to topology changes and is able to handle fixed infrastructure where some nodes in the network are acting as collecting points, such as sinks and base stations. According to [12] LWB provides superior performance in many-to-one and many-to-many scenarios compared to other state-of-the-art routing and link-layer protocols.

LWB can be divided into three components namely, the bus interface, the application and the scheduler. The application contains user specific code and only interacts with the bus interface. Furthermore the bus interface is the core of the LWB system and handles all communication between the application, Glossy and the scheduler. In LWB, the application communicates directly with the bus interface and replaces the standard network stack, which results in that no extra layers are required.

Since LWB is built on top of Glossy which is a synchronous communication protocol, the LWB protocol operation can be divided into rounds and slots. In LWB nodes perform communication in so called communication rounds. Each communication round then consists of multiple slots which can be grouped into three different types namely, scheduling, data and contention. The scheduling slots always occur first in a round and is used for the host (coordinator) to distribute information regarding in which slots each node is allowed to send its data. In the beginning of a data slot at most one node initiates a flood by broadcasting its data and all other nodes reads this data and perform flooding to forward the message. This procedure can be comparable with a bus, where the flooding initialization is the same as adding

a message to the bus and the forwarding and receiving part can be seen as reading the message from the bus. Lastly, in the end of a round, the contention slots are used in order for all nodes to inform the coordinator about any specific requests. For example, it is possible for nodes to inform the coordinator about any changes in their traffic demand for the upcoming round and the coordinator then takes this into account when calculating the new schedule.

3.1.3 Discussion: Comparing TSCH and Glossy

TSCH and Glossy are both synchronous communication protocols, operating at the MAC layer portion of the network stack. As LWB builds upon Glossy and our discussion concerns synchronous communication on the MAC level, LWB will not be part of the discussion comparison.

Both TSCH and Glossy maintain time synchronization in order to establish synchronous transmissions, consequently preserving energy by only having the radio turned on at times when communication is expected to take place. However, the way that the protocols performs communication and achieves time synchronization differs greatly between TSCH and Glossy; we will therefore limit our discussion to focus on these two topics.

The TSCH protocol supports both broadcast (flooding) and reliable unicast as communication primitives, whereas Glossy builds upon its flooding architecture and does not support unicast transmissions. Glossy is also limited by having to relay the same packet during a protocol round, in order for constructive interference to take place. TSCH on the other hand, is allowed to send different packets using any of its two communication primitives during a protocol round; yet, destructive interference is a problem in TSCH when nodes communicate using the same frequency and when the location of nodes are spatially dense. TSCH does, however, support the use of a back-off algorithm for evading some occurrences of destructive interference when nodes communicate using the same frequency, yet this strategy does not completely resolve the issue.

As previously mentioned, TSCH will experience a performance degradation due to destructive interference when nodes communicate using the same frequency and are located spatially dense, while Glossy does not show any performance dependency on node density. TSCH has the possibility to cope with the aforementioned problem by utilizing frequency diversity, also known as channel hopping. The communication frequency is decided based upon an absolute slot number (slot offset) and the channel offset specified in the currently used TSCH schedule. This allows for concurrent transmissions with multiple different frequencies which will not interfere even in the case when nodes are located close to each other. This however, has the disadvantage of nodes only receiving packets sent with the same frequency as the nodes are currently listening to; thus, flooding in TSCH will not be possible in an area crowded with nodes, without experiencing a noticeable performance degradation.

As explained in Section 3.1.1, nodes running Glossy solve time synchronization implicitly by incorporating a 1-byte field into all packets exchanged during communication. The nodes then make use of this field together with a slot length estimation T_{slot}, to correctly adjust their clocks' sampling rate to keep the nodes relatively synchronized to each other. TSCH similarly supports integration of timing information into exchanged packets, but reinforces the use of two different synchronization policies which can be used independently or in combination depending on the network requirements; for further details on the two policies, see Section 2.3.

3.2 Agreement Protocols in WSNs

Consensus in WSNs is not a widely researched topic, especially not in the context of network-wide agreement. To the best of our knowledge most agreement protocols, available for WSNs, are limited to pairwise agreement between nodes [2, 25, 7, 8]. Below we describe the two pairwise agreement protocols JAG [2] and 6P [25], and finally compare these against the A^2 protocol.

3.2.1 JAG: Jamming-based AGreement

JAG [2] is a protocol designed to achieve pairwise agreement between nodes, in a network under external radio interference. In the presence of heavy interference, packets may be corrupted and as a result become undecodable when processed by a receiving node. JAG has solved this problem by using a phenomena called jamming, where a special jamming sequence is transmitted to indicate agreement between nodes.

The design of the agreement protocol is based upon a 3-way handshake between two nodes. A handshake consists of multiple packets transmitted to agree upon a certain condition. In a standard 3-way handshake the communication ends with an acknowledgement packet indicating the completion of the handshake. In JAG a standard 3-way handshake is performed but with one big difference, instead of sending an acknowledgment the handshake ends with a transmitted jamming sequence. The nodes performing the handshake have to be tightly synchronized in order for the receiver to successfully sample the transmitted jamming sequence, since the jamming signal needs to be sampled when it is on the air. The jamming signals in JAG can be generated by standard motes without any extra hardware.

JAG is evaluated in [2] against another protocol known as the 2-way handshake Message-based AGreement (2-MAG). According to the results, JAG outperforms the message-based approach 2-MAG when it comes to reaching agreement under interference. In [2] it is also described how JAG can be embedded with other communication protocols on the MAC level such as Chrysso [16] and CoReDac [24]. In both Chrysso and CoReDac the 2-way handshake can be replaced by the 3-way handshake introduced in JAG.

3.2.2 The 6P Protocol

The 6top Protocol (6P) [25] is a scheduler developed on top of 6TiSCH (the most recent development of TSCH which includes support for IPv6). The main task of the 6P protocol is to allow nodes to send scheduling requests and collectively reach an agreement. It is of high importance that a modification of a TSCH schedule is agreed upon by the neighbouring nodes, since otherwise future communication might fail if nodes operate using different schedules. To handle this problem the 6P protocol makes use of something called transactions. A transaction is a handshake consisting of multiple steps where all steps have to be completed, by both nodes, for the transaction to be successful; if the handshake fails, the transaction is aborted. Below are two different transactions presented that are supported by the 6P protocol, namely the 2-step and 3-step 6P transaction.

Messages sent during a transaction in the 6P protocol consists of five important parameters: type, code, numcells, cell-list and seqnum (sequence number). The type of a message determines whether a message is a request, response or confirmation. The numcells parameter indicates how many scheduling cells should be changed, and the cell-list parameter contains a suggestion list of cells that the initiator node is able to modify. The code parameter determines what operation should be performed and can either be delete, add or relocate. Lastly, the sequence number is used to match the messages to a specific transaction.

An initiating node of the 2-step transaction starts the first step by sending a request message with the corresponding cells it is able to modify (candidate cells) and then waits for an acknowledgement. In case the initiator successfully receives an acknowledgement, a timeout timer is set; the timeout timer specifies the maximum waiting time before the current transaction is to be aborted. In the second step of the protocol, the receiver sends a response message containing a subset of the candidate cells it wants to modify, also called *selected cells*. If the initiator receives a response message with the same sequence number as was initially sent in the first step and agrees upon the selected cells, an acknowledgement message is transmitted and the timer is turned off. After a successful completion of the transaction, the two participating nodes will modify their schedule according to the agreement.

The first step of the 3-step transaction starts off in the same way as in the 2-step transaction but with one main difference, the initiator does not include any candidate cells in the request message but instead allows the receiving node to make this decision. In the second step, the node which received the request message then transmits a response message containing its suggestion of candidate cells, waits for an acknowledgement and then starts a timeout timer subsequent to the acknowledge reception. In the third and final step the initiator responds with a confirmation message including the selected cells it decided to modify. Figure 3.1 and 3.2 showcase an example of the 2-step and 3-step transaction of the 6P protocol.

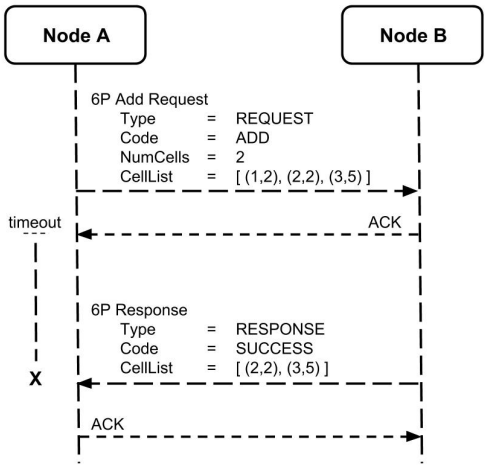


Figure 3.1: *An example of a 2-step transaction of the 6P protocol [25].*

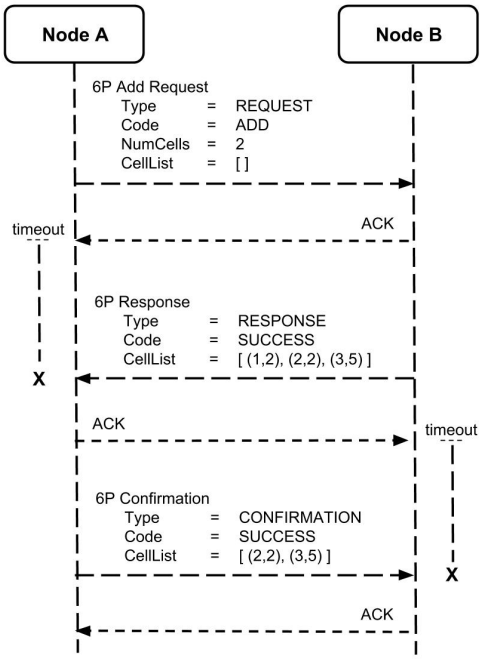


Figure 3.2: *An example of a 3-step transaction of the 6P protocol [25].*

3.2.3 Consensus Comparison: A^2, JAG and 6P

Both JAG and the 6P protocol achieve consensus between pairs of nodes by using a handshake procedure. This restricts the two protocols of not being able to establish consensus among nodes which are outside communication range, in a multi-hop network topology. A^2 on the other hand, achieves consensus on a network-wide basis, with the only requirement being that the topology is connected. While JAG and 6P only achieve pairwise consensus, A^2 supports multiple distributed consensus protocols: Vote, Max, 2PC and 3PC. Furthermore, these protocols integrated into A^2, have the same data format for communication and use the same number of predefined states to construct the protocols' corresponding state machines. This makes A^2 easy to extend for further development of other consensus protocols and services, in contrast to 6P and JAG which are limited by their specific purposes and predefined set of rules for achieving pairwise consensus.

4
Design

In this chapter we present an overview of our A^2-with-TSCH system. Section 4.1 introduces our system architecture which consists of an illustration of our design, together with a design description of each component. Furthermore, we conclude this chapter by discussing our design preferences together with a small comparison between our A^2-with-TSCH system and the original system, A^2-Synchrotron.

4.1 System Architecture

In this section, we provide a high-level overview of each component in our system architecture, together with a more in-depth description of one of our main contributions: Bridge. Figure 4.1 illustrates the original A^2-Synchrotron system and Figure 4.2 portrays a high-level representation of our system design which we call A^2-with-TSCH. The reason why we showcase both systems next to each other is to clearly emphasize on the design differences of the original A^2-Synchrotron system from which we developed our A^2-with-TSCH system. As Figure 4.2 illustrates, our system architecture consists of A^2 integrated on top of the MAC-layer protocol TSCH. We also introduce Bridge, a middle-layer interface working as a mediator for interconnecting communication between A^2 and TSCH.

4.1.1 A^2

The A^2 system is a library collection of different isolated protocols, sharing the same design structure with similar protocol properties, but on the other hand accomplishing different results. When referring to the A^2 system, it is the design structure shared among all protocols which we refer to. In the original system, illustrated in Figure 4.1, A^2 encompasses several consensus protocols, namely: Vote, Max, 2PC and 3PC. This differs from our system architecture in A^2-with-TSCH, where only Max and 2PC has been implemented. We only implement Max and 2PC as a proof of concept, for showing that A^2 protocols in fact are able to run on top TSCH and is not limited to only run on top of Chaos. We did not implement Vote and 3PC due to time restrictions, however, our systems design makes it easy for future developers to port the remaining protocols with little to no effort.

All A^2 protocols are state machines assembled differently depending on the protocol, yet the construction of an A^2 state machine is limited by a predefined number of states. Considering that an A^2 protocol is a state machine, its primary purpose is

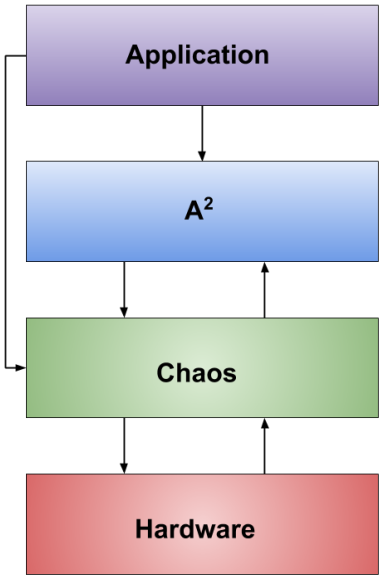


Figure 4.1: *The original system architecture: A^2-Synchrotron.*

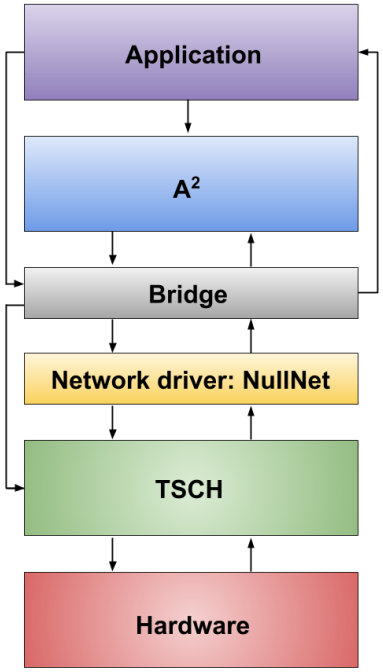


Figure 4.2: *The final system architecture: A^2-with-TSCH.*

to generate a new state, based on a previous state and information contained in the most recent data packet received thus far. The newly computed state will later be passed down to the lower layer for further processing, which in this case will be handled by Bridge; described further in Section 4.1.5. Moreover, the packets exchanged between nodes when running the different A^2 protocols, all share the same data format: payload and flags. As all A^2 protocols share the same design structure, this makes it straightforward to run multiple A^2 protocols concurrently without keeping track of different states and packet formats for different protocols.

4.1.2 Application

The application layer is the topmost layer of the system architecture, designed and controlled by an application developer. An application requesting to run the A^2 system on top of TSCH, needs to import the corresponding A^2 protocol(s) it wants to maintain, and also initialize Bridge for the whole system to work. The application initializing Bridge also provides a reference back to itself for Bridge to be able to return control to the application. Consequently, applications running A^2 on top of TSCH need to follow specific implementation guidelines; we explain these guidelines in Section 5.1.

4.1.3 TSCH

TSCH is a reliable synchronous communication protocol that handles all radio communication, by using the underlying hardware as explained in Section 2.8. For

achieving synchronous communication, TSCH has to maintain a tight time synchronization with the other motes in the network. TSCH provides a useful API for initializing TSCH and also for regulating a schedule, controlling how often nodes should turn on the radio in order to perform synchronous communication. TSCH is able to communicate with the network driver NullNet, by utilizing a callback function which TSCH invokes upon a successful packet reception.

4.1.4 Network Driver: NullNet

In A^2-with-TSCH, NullNet is our network driver. NullNet is a simple network driver, which forwards packets between Bridge and TSCH. In case of a transmission, NullNet receives outgoing data from Bridge and creates a packet by appending the destination address to the packet header. Nullnet then puts the packet in a packet buffer, accessible only by TSCH and NullNet, and invokes a callback to TSCH. Upon a packet reception, TSCH invokes a callback function to NullNet which indicates that there exists an incoming packet in the packet buffer. NullNet extracts the corresponding data and forwards it to Bridge.

4.1.5 Bridge: The Middle-layer Interface

We present Bridge, a middle-layer interface for interpreting communication between A^2 and TSCH. We appoint Bridge to interpret communication between A^2 and TSCH, considering that the two layers are not compatible by default.

Communication between Bridge and A^2 consists of exchanging states, payloads and flags. Bridge requests a new state from the currently running A^2 protocol by forwarding the current state and the most recent data. Once Bridge obtains a new state, Bridge interprets this information and performs certain actions, consequently informing TSCH whether to transmit, receive or sleep in the upcoming timeslot.

Evident from Figure 4.3, Bridge interacts with the network driver NullNet in order to exchange incoming and outgoing data packets with TSCH. Bridge simply passes outgoing data to NullNet and receives incoming data from NullNet.

Bridge also communicates with the application layer to notify an application when its corresponding protocol is supposed to run and also when the protocol is complete. Bridge determines the next application by using a scheduler; for further reading concerning the scheduler, see Section 5.3.

4.2 Discussion

When considering the two architectural designs, in Figure 4.1 and Figure 4.2, one fundamental difference separates these designs, namely the MAC-protocol. In A^2-Synchrotron, Chaos is customly equipped to interoperate smoothly with A^2-protocols thereby making them tightly coupled. For example, Chaos has support for A^2 states and is built to perform actions based on these different states. This is not the case

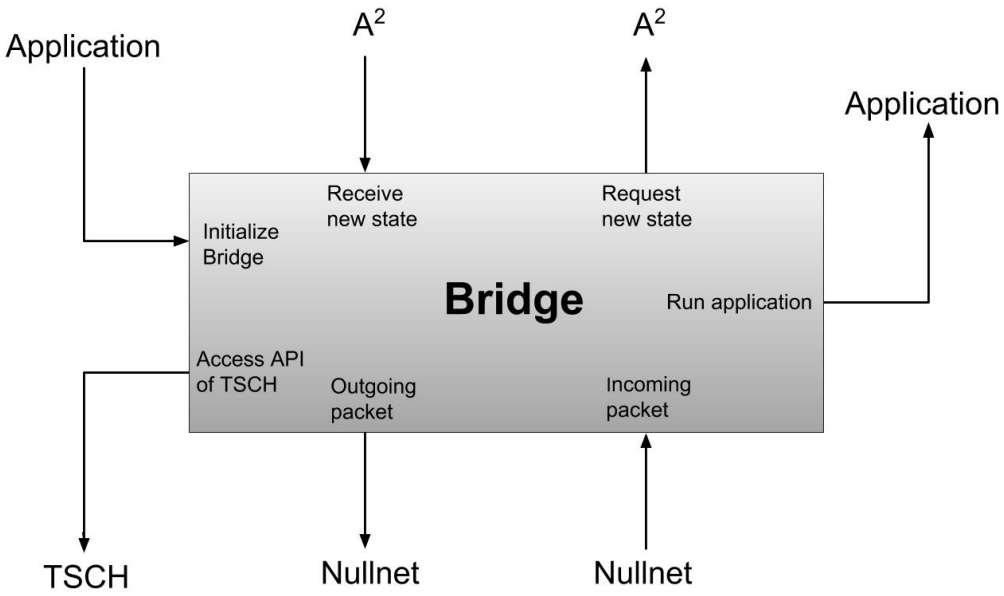


Figure 4.3: *A detailed illustration of the Bridge interface.*

for TSCH, as TSCH sets out to be as generic as possible and is not customly built to work with a particular upper-layer protocol. We therefore choose to separate the necessary A^2 logic from Chaos and create a new separate module, Bridge, in order to make A^2 operate on top of TSCH. Below we discuss different placement strategies regarding our Bridge module.

We decided to place the majority of our middle-layer Bridge logic in an isolated module, separated from the A^2 protocols and TSCH. It would also be feasible to place all of Bridge logic inside of TSCH, as TSCH is open-source. However, this approach is counterintuitive considering that Bridge has nothing to do with MAC-layer logic. Also, if we decide to place Bridge inside of TSCH, our contribution to the open-source community will most likely go unused and clutter the already existing TSCH protocol. Another alternative approach would be to place the Bridge logic inside each of the A^2 protocols; in this case it becomes clear that we would get repetitive code i.e., unnecessary redundancy. Clearly, none of the two aforementioned placement strategies will produce a beneficial design choice, thereby motivating our approach of placing Bridge in isolation.

An additional design approach could be to integrate the network driver NullNet together with the middle-layer interface Bridge, as a single unit. However, our design decision of placing the network driver as a standalone component below Bridge, gives more of a coherent overview to the layered protocol structure and also separates the network driver logic from Bridge logic.

In our A^2-with-TSCH system, Bridge is able to directly communicate with TSCH. The reason for this is because Bridge needs access to the API of TSCH, which cannot be obtained directly from NullNet; as communication between NullNet and TSCH simply consists of exchanging incoming and outgoing packets. Furthermore,

the explicit communication between Bridge and the application layer may also seem illogical, however, it is actually of high importance for several reasons, for example to be able to initialize the Bridge; for further explanations related to this topic, see Section 5.2.1.

5
Implementation

The following chapter presents the different components of the final A^2-with-TSCH system. This chapter starts off by describing our implementation of an application structure, followed by a more detailed description regarding the implementation of Bridge. Furthermore, the application scheduler is presented together with a more in depth discussion concerning certain components of the A^2-with-TSCH system. We implement A^2-with-TSCH in the low-power operating system Contiki and we extend upon the TSCH implementation of Contiki-NG [3]. Our contributions mainly consist of the implementation regarding Bridge, other components such as the packet buffer and NullNet are not part of our implementation and belongs to the Contiki-NG platform.

5.1 Application Structure

In our A^2-with-TSCH system, an application is capable of running one of the following A^2 protocols: Max or 2PC. In order for an application to run an A^2 protocol, it needs to import the corresponding header file. By doing this, the application can access protocol specific functions necessary to initialize the protocol and obtain protocol results.

Bridge is responsible for determining when an application shall initialize its A^2 protocol as well as when the application is able to obtain the final protocol results. For that reason, all applications need to create an instance of themselves to make them accessible by Bridge. We choose to represent every application instance as a compound data structure, illustrated in Figure 5.1. Our compound data structure, *a2_app*, consists of the following fields: *name*, *max_slots*, *round_frequency*, *wait_time*, *app_begin* and *app_end*. The *name* field works as an application identifier providing a reference name to the application. *max_slot* specifies an upper-bound for how many timeslots an A^2 protocol is allowed to run before protocol progression halts. The *round_frequency* specifies how often the application aspires to run, for example, a round frequency set to 1 implies that the application wishes to run every round, while a round frequency set to 2 suggest that the application desires to run every other round. The *wait_time* field is not provided by the application but is instead exclusively handled by the application scheduler; we present a more detailed description of the application scheduler in Section 5.3. Lastly, the application is obligated to provide two separate callback functions, *app_start* and *app_end*. Bridge will invoke the *app_start* function in the beginning of a round to

let the application initialize its A^2 protocol. The *app_end* function is also invoked by Bridge, at the end of the same round in order for the application to retrieve the final protocol results.

Finally, once the application is registered as an *a2_app*, it needs to hand over execution to Bridge which subsequently commences the A^2-with-TSCH system.

```
typedef struct a2_app {
  char* name;
  uint16_t max_slots;
  uint16_t round_frequency;
  uint16_t wait_time;
  app_callback app_begin;
  app_callback app_end;
} a2_app_t;
```

Figure 5.1: *The application structure written in C.*

5.2 Bridge Implementation

In this section we start by describing the initialization phase of Bridge and then we move on to more details regarding the execution in each timeslot. Furthermore we also describe how the Bridge is connected to the A^2 protocols and specific implementations concerning time synchronization.

5.2.1 Initialization

Bridge is mainly responsible for connecting all the layers together but also handles the initialization of TSCH. During the initialization phase, Bridge performs the following actions:

- Bridge requests the next app to run, from the application scheduler; this action is only done by the initiator node.
- Bridge initializes the standard random number generator of Contiki.
- Bridge provides a callback function to NullNet.
- Finally, Bridge initializes TSCH.

The initiator is the only node which invokes the application scheduler, since it needs to inform when the next application should run. Moreover, the standard random number generator is initialized, to later be used by the A^2 protocols. Bridge additionally provides a callback function to NullNet. Nullnet saves this function and invokes it once NullNet's own callback function is triggered by TSCH; Section 5.2.2 describes this process in more detail. Lastly, Bridge needs to initialize TSCH consequently handing over execution to the MAC-layer.

5.2.2 Timeslot Execution

TSCH maintains a real-time timer (*rtimer*) which is set to trigger in the beginning of each timeslot. The *rtimer* preempts any currently running process in order to always provide code execution to TSCH. As a consequence, TSCH is responsible for providing code execution to the upper-layers. The standardized behavior of TSCH is to invoke a callback function in the post-processing stage of a timeslot, only if a successful data packet reception is made. This default behavior of TSCH is not sufficient, as A^2 protocols expect to run in each timeslot independently of whether a data packet is received or not. We therefore modify the default behavior of TSCH to instead always invoke a callback to NullNet, in the post-processing stage of every timeslot. Figure 5.2 illustrates TSCH's default behavior and Figure 5.3 depicts the modified behavior of TSCH.

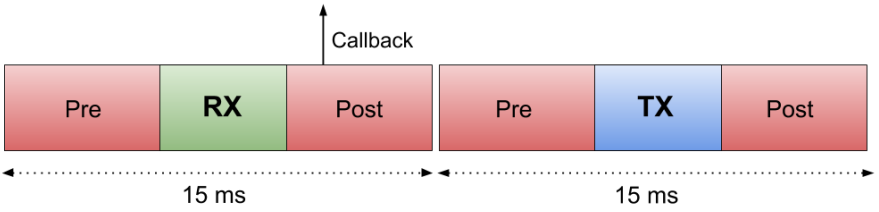


Figure 5.2: *An illustration of the default behavior of TSCH. A callback to the upper-layer is only invoked upon a successful packet reception.*

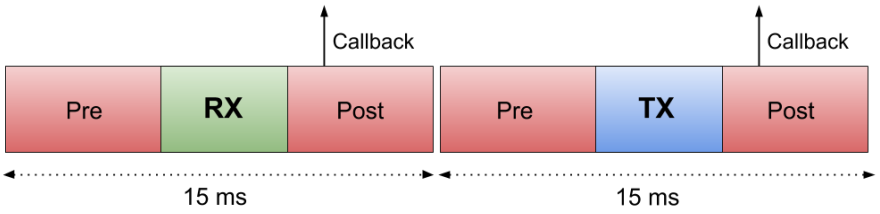


Figure 5.3: *In illustration of the modified behavior of TSCH. A callback to NullNet is always invoked in the post-processing stage, regardless of whether a packet is received or not.*

5.2.3 Communication

Bridge handles all incoming and outgoing communication concerning A^2 protocol data. We appoint NullNet as the network driver, responsible for forwarding outgoing data to TSCH and receiving incoming data from TSCH. Considering that TSCH is the MAC-layer protocol, it is solely in charge of transmissions and receptions of packets. As we build our system on top of TSCH, our implementation does not handle packet communication on the MAC-level; we therefore rely on TSCH to handle communication for us. It is also important to note that our implementation concerning packet transmission and reception is limited to the communication between Bridge and NullNet. This section starts off by describing the packet format used in

the A^2-with-TSCH system, and continues to outline the execution flow of a packet transmission and a successful packet reception.

5.2.3.1 Packet Structure

We choose to implement our own packet structure, comprising of information that needs to be exchanged by all nodes in the network. Figure 5.4 illustrates that the packet structure, *a2_tsch_packet*, consists of a header field and a payload field. We place information maintained by Bridge into the header field, while information related to A^2 protocols is placed in the payload (*a2_payload*).

The header field is comprised of 5 bytes in total, where the first 3 bytes consist of scheduling information: *next_app_index* and *rounds_to_next_app*. The *rank* field constitutes the remaining two bytes and is used by the A^2-with-TSCH ranking system in order to improve on an existing synchronization problem, present in the current TSCH implementation; see Section 5.2.6 for an explanation of the problem, together with our corresponding solution.

The *a2_payload* field is handled by the currently running A^2 protocol and has an unspecified length. Different A^2 protocols expect different number of bytes in the payload field. As a result, we employ a dynamic array which is allocated at runtime with the expected number of bytes, in the beginning of a protocol round. Figure 5.5 shows that the *a2_payload* field can be further divided into protocol data and flags. However, these two fields are exclusively handled by A^2 protocols to ensure protocol progression, and will not be mentioned further as they are not part of our work.

```
typedef struct __attribute__((packed))
a2_tsch_header {
  uint8_t next_app_index;
  uint16_t rounds_to_next_app;
  uint16_t rank;
} a2_tsch_header_t;

typedef struct __attribute__((packed))
a2_tsch_packet {
  a2_tsch_header_t header;
  uint8_t a2_payload[];
} a2_tsch_packet_t;
```

Figure 5.4: *The data structure for storing packet information in A^2-with-TSCH, written in C.*

5.2.3.2 Transmission

In order to transmit a packet from Bridge, we first need to contact NullNet. NullNet takes two parameters: a reference pointing to the outgoing data and its total

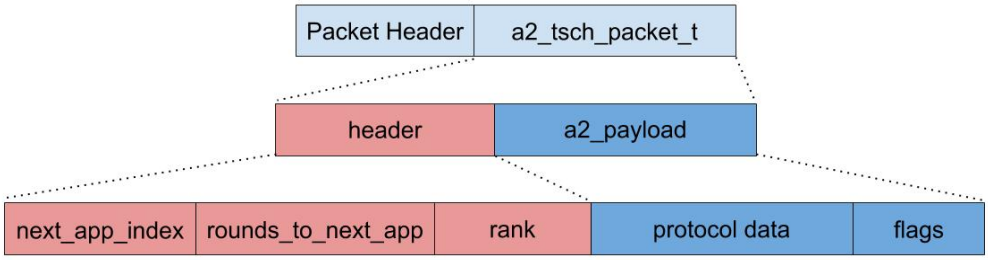


Figure 5.5: *A visual representation of an a2_tsch_packet.*

length. Thus, we provide a reference to a memory location pointing to the outgoing *a2_tsch_packet*, along with its total length in bytes. Below, we explain the whole step-by-step process of a packet transmission in A^2-with-TSCH; Figure 5.6 illustrates a graphical representation of the full process.

1. TSCH invokes its callback function to NullNet in the post-processing stage of a timeslot.
2. NullNet hands over execution to Bridge, by invoking its callback.
3. Bridge forwards its outgoing data to NullNet.
4. NullNet creates a packet by adding the data as payload, attaches a header field containing a destination address and subsequently adds the packet to the packet buffer. We restrict NullNet to only append the broadcast address as the destination address of all outgoing packets, as all A^2 protocols are flooding based.
5. NullNet contacts TSCH, signaling that there exists an outgoing packet in the packet buffer.
6. TSCH extracts the packet from the packet buffer and further creates the final packet to be sent, by attaching TSCH specific header fields such as timing information. TSCH then adds the packet to its outgoing transmission queue.
7. The packet is finally sent by TSCH, in the upcoming timeslot.

5.2.3.3 Reception

Bridge is the topmost-layer which processes data contained in packets, initially received by TSCH. The raw data ending up at Bridge, is merely a sequence of bytes; in order to make sense of it, we parse the data into an *a2_tsch_packet*. The header information of an *a2_tsch_packet* is then further processed by Bridge, while the *a2_payload* is simply passed to the currently running A^2 protocol. Bridge performs a simple validation check before parsing the raw data, by validating the total length of the received data. If the length of the raw data exceeds the allocated memory of an *a2_tsch_packet*, a memory overflow occurs. On the other hand, if we obtain less data than expected, faulty values will be inserted into the *a2_tsch_packet* resulting in unwanted behaviors. The total length of the raw data therefore needs to match

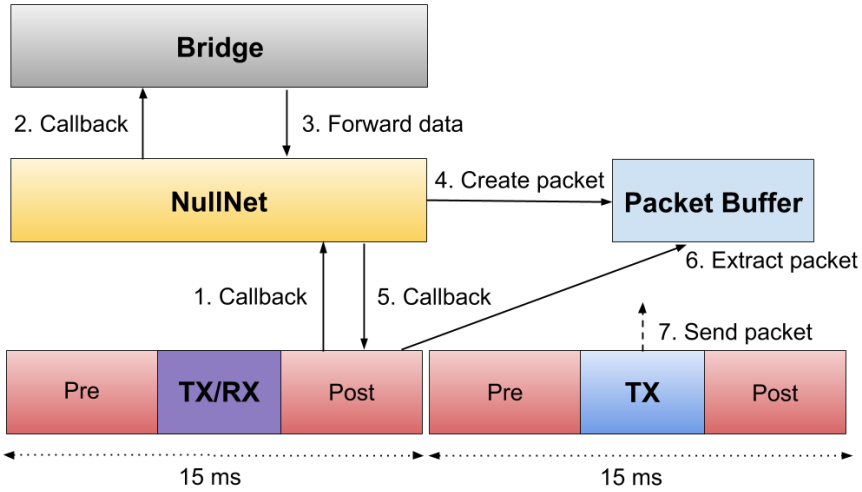


Figure 5.6: *A step-by-step illustration of a packet transmission in A²-with-TSCH.*

the expected length of an *a2_tsch_packet*, otherwise the raw data is discarded. Below we explain the entire step-by-step process of how a packet reception is performed in A²-with-TSCH; Figure 5.7 provides a graphical illustration of how this process is done.

1. TSCH strips the newly received packet from all TSCH-specific header fields and adds it to the packet buffer.
2. TSCH notifies NullNet that a new packet is available in the packet buffer.
3. NullNet extracts the packet from the buffer and saves a reference to the raw data and an integer value specifying the raw data length.
4. NullNet forwards the raw data to Bridge, together with a variable specifying the length of the raw data.
5. Bridge performs a length validation of the raw data, before it is further processed.

5.2.4 Interconnecting A² and TSCH

The primary purpose of the Bridge is to act as a middle layer, handing communication between TSCH and the A² protocols. In order for A² protocols to run on top of TSCH, Bridge needs to be able to mediate actions produced by the protocols. In each timeslot, we requests a new state from the currently running protocol. Bridge instructs TSCH what shall happen in the upcoming timeslot, based on the new state we received. As a result, Bridge needs to be aware of all the states that can be produced by the protocols: TX, RX and OFF. In case of the new state being TX, a packet containing the latest protocol data is scheduled for transmission. If the new state is RX, the Bridge does not do anything; since TSCH does not receive any instructions of what to do in the upcoming timeslot, TSCH simply turns on its

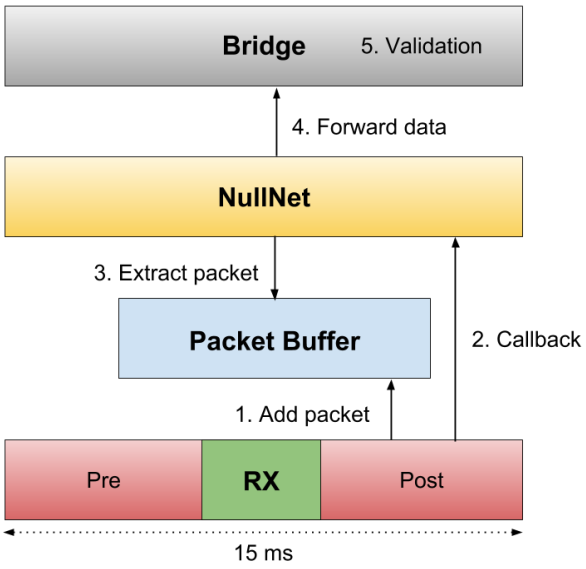


Figure 5.7: *A step-by-step illustration of a packet reception in A²-with-TSCH.*

radio by default to listen for incoming packets. Lastly, the state OFF indicates that a protocol has completed. As a consequence, Bridge manipulates the current TSCH schedule instructing TSCH to turn off the radio for the remaining round.

5.2.5 Dedicated EB-timeslots

By default TSCH is allowed to send EB-packets whenever it is necessary. This behavior is not favourable during a protocol round as transmissions of EB-packets are likely to interfere with data packets, and as a consequence interrupts progression of the current A^2 protocol.

We introduce dedicated EB-timeslots, a mechanism for only allowing transmission of EB-packets in specific timeslots. During these timeslots, nodes are not allowed to send data-packets; instead, nodes either transmit or listen for EB-packets. As a result, EB-packets do not interrupt progression of A^2 protocols. We also effectively decrease the time it takes for nodes to associate to the existing network, consequently reducing energy consumption. We allow a user to specify the frequency of dedicated EB-timeslots, where we set a default value of 21; indicating that EB-timelots will occur every 21th timeslot.

5.2.6 Time Synchronization in TSCH

In this section, we start by presenting an overview of a synchronization problem in the current TSCH implementation of Contiki-NG. In Section 5.2.6.2 we also propose our solution to the problem by introducing a ranking system.

5.2.6.1 Problem Statement

In TSCH, nodes constantly need to receive synchronization information from their time-source neighbor in order to stay relatively synchronized to the other nodes in the network. This becomes a problem if a node's time-source is far away or the majority of packets, sent from the time-sourcee, are lost due to interference. If a node has not received synchronization information within a certain period of time, TSCH enforces the node to start sending keep-alive packets to its time-source. However, these packets are not likely to end up at the time-source either, since both nodes presumably have a bad connection between one another in the first place. If none of the transmitted EB-packets arrive at the time-soure, TSCH will select its backup node to be the new time-source; i.e., the node it last received an EB-packet from. By changing a node's time-source to any random node, a cycle has a possibility of being created consequently breaking the directed acyclic synchronization graph. As a consequence, the nodes which are constituting a cycle will form a synchronization cluster. Every node in the cluster will stay relatively synchronized to each other, while slowly drifting out of sync with the remaining nodes of the network. We propose a solution to the aforementioned problem, by introducing a ranking system into our A^2-with-TSCH system. Our solution enforces regulated adjustments of nodes' time-source neighbors, to avoid cycle formations in the network's synchronization DAG.

5.2.6.2 Ranking System

We assign a rank to each node upon associating to a network. The initiator gets a rank of 0, while the remaining nodes are assigned an initial rank value of 65535; the maximum value of a 16-bit unsigned integer. Once a non-initiator node receives a data packet from its time-source, it updates its rank to one higher than the received rank value. This will create a tree hierarchy where all rank values represent the distance from the root (initiator). For example, nodes connected to the initiator will have a rank of 1 and nodes connected to these nodes will have a rank of 2, and so on. Additionally, all nodes need to have a directed path to the initiator, in order to stay relatively synchronized. Figure 5.8 shows an example where *Node 4* wants to change its time-source. The following conditions need to be met in order for the switch to take effect:

- *Node 4* is not the initiator node.
- The KA timer of *Node 4* has expired.
- A data packet needs to be received, containing a rank value less than or equal to the local rank of *Node 4*. Hence, *Node 4* is not allowed to change its time-source neighbor to be a node with a higher rank value than itself; consequently preventing formations of cycles.
- The sender of the data packet is not already the current time-source of *Node 4*.
- *Node 4* has not changed its time-source during the last two protocol rounds.

Following a time-source change, all subtrees belonging to the node that recently changed time-source need to adjust their rank according to the new topology. The new rank value is propagated down the subtrees in a passive manner, since rank values are only updated when data packets are received during protocol rounds. Additionally, once a node has successfully changed its time-source, we restrict this node from performing a new time-source switch until two protocol rounds have passed; this ensures that the new rank value has propagated down the tree, with a high probability.

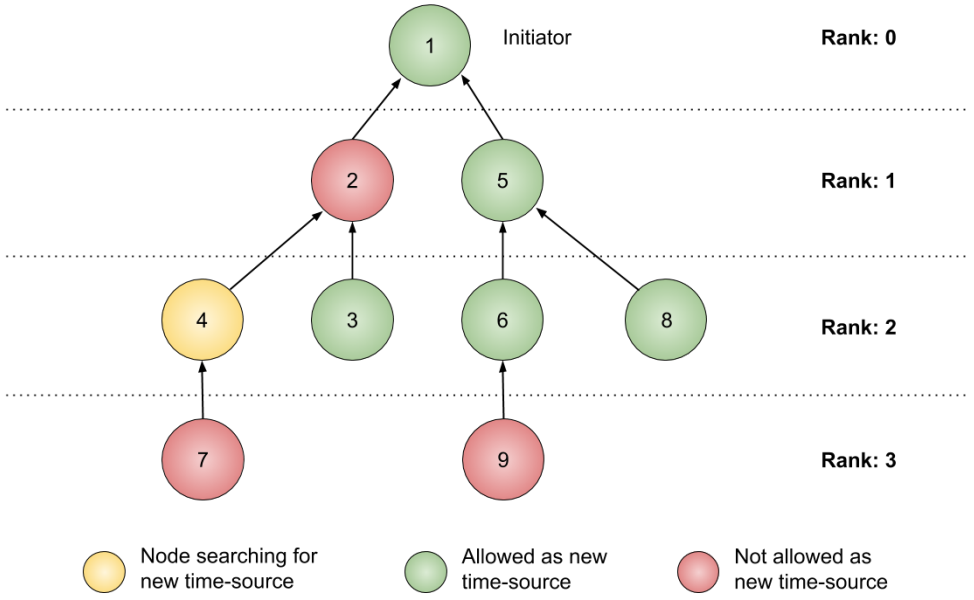


Figure 5.8: *An example of the ranking system, where Node 4 wants to change time-source. The tree illustrates all possible time-source changes available to Node 4.*

5.3 Application Scheduler

To be able to switch between multiple applications during runtime, we present a simple application scheduler. The scheduler provides an interface, giving users the possibility to specify how frequently they want to schedule every application. As of the final implementation, applications are only able to run Max or 2PC, however, the scheduler supports any application and is not dependent on the A^2 protocols. Below we describe the functionality of the scheduler which performs two different functions depending on whether a node is the initiator node or not.

The function used by the initiator, provides the next application to be scheduled. The scheduler does this by going through a list of all registered applications and finds one or multiple applications with a round frequency that is an even multiple of the round number closest into the future. Consider an example where only one application is registered, and its corresponding *round_frequency* is set to 5. This implies that the application wants to be scheduled when the round number is an

even multiple of 5; assuming the current round number is 6, the application will be scheduled to run in round 10.

In the case when two or more applications desire to run in the same round, we are faced with a scheduling conflict. Figure 5.9 illustrates an example of a scheduling conflict, where Max and 2PC aspires to run in the same round. The scheduler solves this problem by examining the applications' corresponding *wait_time* variable and selects the application with the highest waiting time. This mechanism deals with scheduling conflicts by prioritizing applications with a higher waiting time, consequently preventing certain applications from starving other applications. Once the scheduler has found the next application to be scheduled, it will proceed to set the scheduled application's waiting time to 0 and increment the remaining applications' waiting time by 1. Lastly, the scheduler returns the index position of where the scheduled application can be found (*next_app_index*), together with the number of rounds until the application shall run (*rounds_to_next_app*).

The second scheduling function, employed by all non-initiator nodes, is used to obtain an application by providing an index as input to the scheduler. If the index links to a registered application, the scheduler returns the corresponding application. Otherwise a null-pointer will be returned, indicating that no application was found.

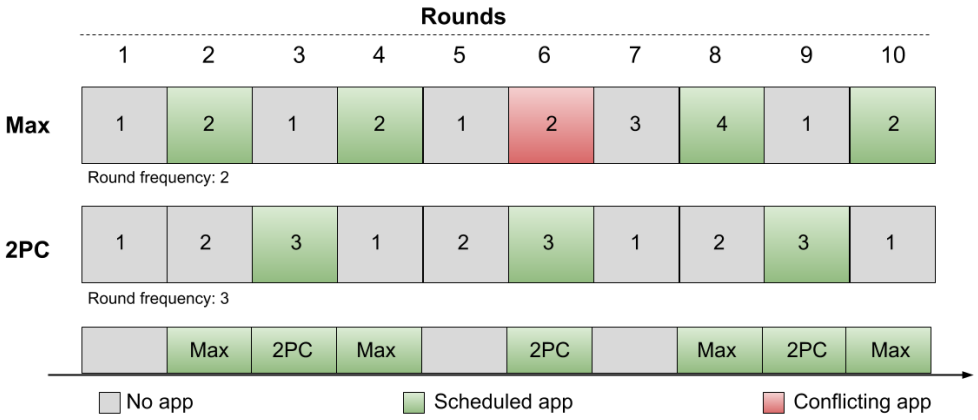


Figure 5.9: *An illustration of how the application scheduler, employed by the initiator, determines which app should run in a specific round. The numbers inside the boxes represent the number of rounds an application has been waiting since its last run (wait_time) and is used to handle scheduling conflicts; higher waiting time means higher priority. The Max application has a round frequency of 2 and therefore wants to be scheduled every other round, while 2PC has a round frequency of 3 and wants to be scheduled every third round.*

5.3.1 Disseminate and Collect Scheduling Information

The initiator node calls the application scheduler in the beginning of each round and obtains the variables: *next_app_index* and *rounds_to_next_app*. These two

variables are stored in the header of an *a2_tsch_packet* (illustrated in Figure 5.5) and then distributed to all nodes in the network. It is crucial for the initiator to disseminate this information to all the remaining nodes in order for them to be able to access the next app so that all nodes are relatively synchronized, meaning they all run the same app in the same round. Nodes receiving the scheduling information update their own header field of their outgoing *a2_tsch_packet* and continue to disseminate this information until every node is informed about the new scheduling information. Since the variable *rounds_to_next_app* instructs all nodes in how many rounds the next app is to be scheduled, no additional data needs to be exchanged until the next application runs. If there is no application to be scheduled during a round, all nodes instead perform a so called waiting round. During a waiting round, nodes only turn on their radio at dedicated EB-timeslots to exchange synchronization information.

5.3.2 Initial Default Protocol

The initial default protocol is built upon the Vote primitive of the original A^2-Synchrotron system. The protocol is known to all nodes and is not linked to any application. It is the first protocol to run and is used to collect and disseminate the most recent scheduling information provided by the initiator, until every node is informed about what application to run next and in which round it is to be scheduled. All nodes run the protocol in their first round and continues to run it in every consecutive round, until consensus is reached for the first time; when consensus is reached every node is connected to the network and aware of the initiator's scheduling information. Subsequently, all nodes continue to exchange scheduling information in every upcoming application round, without having to rely on the initial default protocol.

5.4 Discussion

In this section we discuss a subset of the implemented components, which are restricted to certain limitations consequently contributing with implications to the overall A^2-with-TSCH system.

5.4.1 Application Scheduler

A negative aspect of having the application scheduler is the additional energy consumption, spent during the initial default protocol. Yet, once consensus has been reached for the first time and every node is aware of the scheduling information, the initial default protocol is terminated. From this point forward, nodes continue to exchange scheduling information in application rounds, consequently avoiding additional overhead linked to the initial default protocol.

The total message overhead of the scheduling information consists of 3 bytes, the next application index (1 byte) and rounds to next application (2 bytes). The sched-

uler could be implemented without the variable rounds_to_next_app consequently saving 2 bytes of overhead, however, omitting this variable would require applications to be scheduled consecutively, in every round. By including rounds_to_next_app, we give the possibility of not having to schedule applications in each consecutive round, but instead allowing waiting rounds in-between application rounds. In contrast to always running an application in every round, waiting rounds saves energy by only having the radio turned on during dedicated EB-timeslots, consequently increasing the overall up-time of the system.

Even though the application scheduler allows for scheduling of multiple applications, it does come with some limitations. A summary of the limitations related to the scheduler is listed below:

- Maximum number of applications is limited to 256 (2^8).
- Maximum number of consecutive waiting rounds is limited to 65 536 (2^{16}) which approximates to 3 days, if a round consists of 256 timeslots where each timeslot is 15 ms.
- The scheduling algorithm does not allow users to specify different scheduling disciplines.

5.4.2 Dedicated EB-timeslots

Dedicated EB-timeslots are used in order to control the transmission of EB-packets and is configurable by the user. During protocol rounds we choose to disable dedicated EB-timeslots, since this would just slow down the performance of the protocol and give inaccurate evaluation results, when comparing against the A^2-Synchrotron system. We are aware of the downside of disabling EB-packets during protocol rounds, namely that no nodes are capable of joining the network and so this would be a bad setting in a dynamic environment. However, since our implementation of A^2-with-TSCH primarily focuses on the static context, this is out of scope of this thesis.

As mentioned in Section 5.3.1, dedicated EB-timeslots are used in waiting rounds between protocols but are also activated during the initial default protocol, when nodes try to associate to a network. The EB-packets are essential when nodes try to associate to a network, since these are the only packets that are required by TSCH for establishing a new connecting to a network. In the waiting rounds we choose to make use of dedicated EB-timeslots, but we are aware that any type of packets could be transmitted and EB-packets are not required. This is because the main goal of the waiting rounds is to only maintain synchronization between nodes and not to invite new nodes to the network.

When it comes to the configuration of dedicated EB-timeslots, we let the user specify the frequency of how often these should occur. One should note that dedicated EB-timeslots only reserve timeslots for TSCH to send EB-packets and do not control the creation of EB-packets. The frequency of how often TSCH should generate

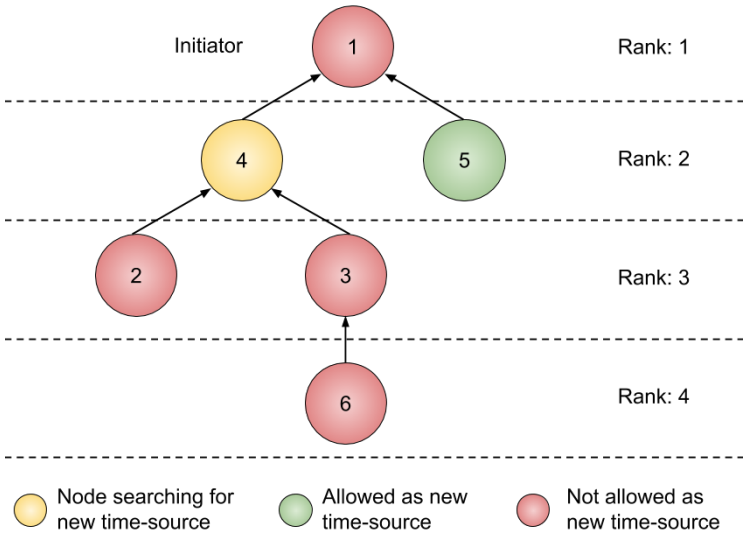
EB-packets can be specified in the TSCH configuration and in order to achieve a good behavior both these settings need to be configured. For example if TSCH creates EB-packets more often than there are slots to send these, all nodes will transmit EB-packets in each EB-timeslot and no node will be receiving. On the other hand if the EB-timeslots occur too frequently compared to the creation of EB-packets, energy will be wasted and the power consumption will increase. This is because all nodes will have their radio on and no one will transmit in the majority of the EB-timeslots. Both the aforementioned problems prevent nodes from receiving EB-packets and thus extends the process of associating to a network. The optimal solution is obviously to have decent configuration of both these parameters to achieve a good behavior. We have not performed a deep analysis of the best ratio between these two parameters, since this is not in the scope of this thesis.

5.4.3 Ranking System

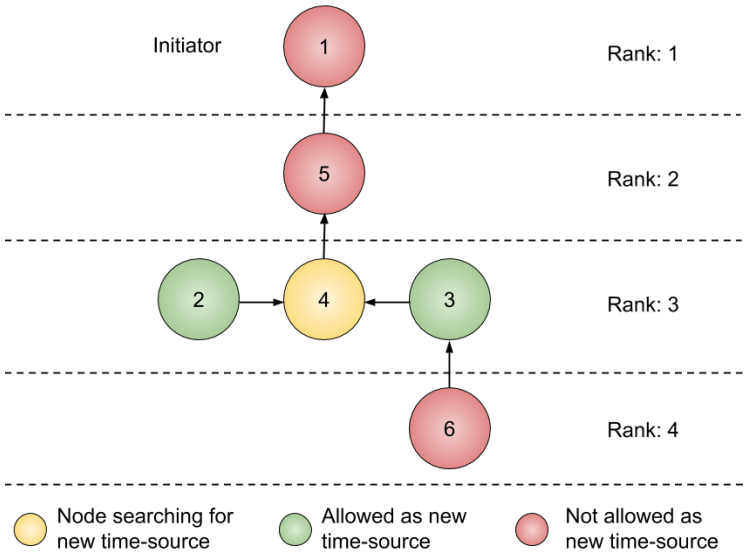
As mentioned in Section 5.2.6.1, the default implementation of TSCH reinforces uncontrolled time-source switches which consequently may lead to cycle formations in the synchronization DAG of a network, causing the nodes to desynchronize. The issue of nodes desynchronizing, arises more often than not when performing tests and evaluations of our A^2-with-TSCH system, driving us to implement the ranking system as a remedy to the problem. The remedy has been a success as the aforementioned problem has not resurfaced. Nonetheless, we are aware that our solution is not bulletproof, and the code placement concerning the ranking system is far from optimal.

In our implementation of the ranking system there is one plausible corner case which might lead to creation of cycles. Consider for example the case in Figure 5.10, where *Node 4* changes its time-source to instead be *Node 5*; as we illustrate in 5.10a, both nodes have the same local rank value prior to the time-source switch. After the time-source change, *Node 4* proceeds to update its local rank value to be one rank greater than of *Node 5*. However, as illustrated in Figure 5.10b, the children of *Node 4* have no idea of the new rank value being assigned to their parent, until every child has received a message containing the new rank from *Node 4*. This means that for some time, *Node 4* and its corresponding children will all be on the same rank level. During this time, *Node 4* could potentially switch time-source again, to one of its children; this would lead to a creation of a cycle. We try to counter this problem by forcing a node which recently made a time-source switch, to wait for at least two protocol rounds before making another switch. We do this to increase the probability that the new rank value has propagated down to the children. However, we are aware that this does not completely resolve the issue, as we have no way of knowing if two protocol rounds are enough in order for *Node 4* to inform all of its children about its new rank. However, our solution to the problem greatly reduces the possibility of nodes forming cycles in the synchronization DAG.

As of the final implementation of A^2-with-TSCH, the code related to the ranking system resides as a part of the Bridge implementation. We choose to implement



(a) *Node 4 is about to make a time-source change. Node 4 can change its time-source to a node with a lower rank or with the same rank as itself, which is only Node 5 in this case. Note that Node 4 is not allowed to change to Node 1, as it is already the time-source of Node 4.*



(b) *Node 4 has successfully changed its time-source to instead be Node 5. Node 4 is about to make a new time-source switch, however, Node 4 has not successfully propagated its new rank value to its children. As a consequence, Node 4 has the possibility of changing its time-source to either Node 2 or Node 3. If Node 4 changes its time-source to one of its children, a cycle will be created and consequently break the synchronization DAG.*

Figure 5.10: *An example demonstrating a flaw that has a possibility of occurring in our ranking system.*

the ranking system as part of the Bridge implementation, only because of time

restrictions. The preferred alternative would be to place the ranking system as part of the current implementation of TSCH, as we encountered the problem in TSCH and TSCH should solely be responsible for time synchronization. Additionally, the existing desychronization problem still resides in the TSCH code, and is likely to cause issues for other developers creating applications on top of TSCH. We will therefore report the existing synchronization problem to the developer team behind TSCH, and hopefully our solution might facilitate in resolving the issue related to uncontrolled time-source switches.

6
Evaluation

In this chapter, we present our evaluation by comparing the results between A^2-with-TSCH and A^2-Synchrotron. We perform the evaluation in the Cooja simulator and on the public testbed FlockLab. The first section starts off by describing our evaluation setup and the corresponding metrics of interest. Furthermore in Section 6.2, we present the results by running the same protocols with identical setups for both systems. We also provide a discussion in Section 6.2.1.3 and 6.2.2.3, where we discuss our findings.

6.1 Evaluation Setup

We evaluate our A^2-with-TSCH system in the Cooja simulator and on the real-world testbed FlockLab. We perform all of our tests on emulated and real Tmote Sky motes. We compare A^2-with-TSCH against A^2-Synchrotron by running two separate applications, with the following protocols on both systems:

- *Max*: network aggregation protocol, where nodes exchange a 4-byte data value and collectively calculate the maximum of all values exchanged.
- *2PC*: consensus protocol supporting transactional guarantees, for further reading see Section 2.2.1.

Applications running in the A^2-with-TSCH system all run with 15 ms timeslots. However, applications running in A^2-Synchrotron run with a timeslot length of 4 ms (Max) and 4.75 ms (2PC). All applications of both systems are configured to transmit 6 times during their corresponding flood phase, before nodes turn off their radios. The flood phase is initiated when a node has received the final agreement value. Both systems are also configured to have a minimum timeout period of 6 timeslots and a maximum of 10 timeslots, and is randomized for each timeout period; a timeout is equivalent to a packet transmission which terminates periods of protocol inactivity, i.e., when nodes do not receive any packets or do not receive any new A^2 data for some timeslots.
During our evaluation we also consider the following metrics of interest:

- *Consistency*: the ratio of rounds all nodes reliably complete a protocol with the same agreement value, over the total number of rounds.
- *Radio-on time*: the total time the radio is on, during a round.

- *Radio Duty Cycle*: the ratio of time the radio is on during a 1 minute interval.
- *Complete Latency*: the number of slots until all nodes are aware of the final agreement value.
- *OFF Latency*: the number of slots until all nodes have turned off their radio.

In our evaluation, we run our test over a period of 100 rounds and consider a representative round for presenting our metrics. We consider a representative round as the round closest to a calculated average. We start by measuring the timeslot, when each node turn off their radio and then we calculate a round average, for all 100 rounds. Lastly, we calculate a final average, by considering all 100 round averages; the representative round is the round closest to the final average.

We measure the radio-on time and the duty cycle by using the built-in Eneregest module [10] of Contiki. The Energest module is a software-based tool, providing energy estimations for resource constrained devices. Energest computes energy estimation based on prior knowledge of the power consumption related to the hardware component of interest; in our case we consider the energy consumption of the Texas Instruments CC2420 radio transceiver.

6.1.1 Cooja

The evaluation in Cooja is performed on a 5-hop network topology of 27 nodes, where each node is emulated as a Tmote Sky mote. We show an illustration of the network topology in Figure 6.1.

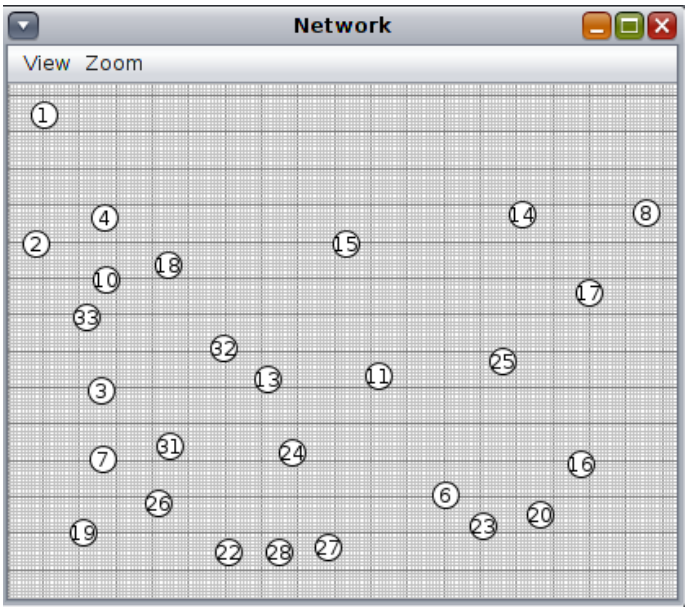


Figure 6.1: *Illustration of the network topology used in the Cooja evaluation.*

Figure 6.2: *Illustration of the network topology used in the FlockLab evaluation [19].*

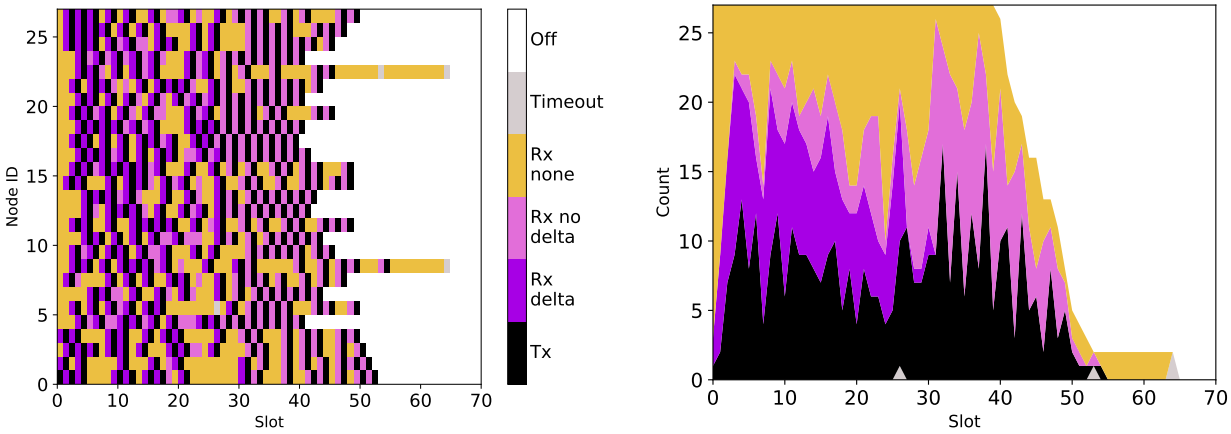
6.1.2 FlockLab

FlockLab is a testbed located in Zurich and has currently support for a total of 27 nodes. In Figure 6.2, we illustrate the FlockLab topology that we use in this evaluation.

6.2 Results

In this section we present the results of our evaluations when comparing A^2-with-TSCH against A^2-Synchrotron. We divide this section into two parts, namely the evaluation of Max and the evaluation of 2PC. We perform our evaluation by running a total of 8 tests, each consisting of 100 rounds. We demonstrate our results by plotting each node's corresponding activity and activity distribution for each timeslot. Below we give a short explanation of what each activity implies:

- *TX*: a node is performing a packet transmission.
- *RX_NONE*: a node has its radio receiver on, but did not complete a successful packet reception.
- *RX_NO_DELTA*: a node successfully received a packet, but did not receive any new A^2 protocol information.
- *RX_DELTA*: a node successfully received a packet which contributed with new A^2 protocol information.
- *TIMEOUT*: a node has terminated a period of inactivity, by performing a packet transmission.
- *OFF*: a node has turned off its radio, indicating the completion of an A^2 protocol; i.e., a node has received the final agreement value and has finished



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.3: *Cooja evaluation: Representative round when running Max in A²-with-TSCH.*

its flooding phase.

6.2.1 Max

In this section we present the results when running Max in Cooja and on FlockLab, on both our A²-with-TSCH system as well as on A²-Synchrotron. Finally, we compare the different results and provide a discussion related to the outcome and characteristics of the different tests.

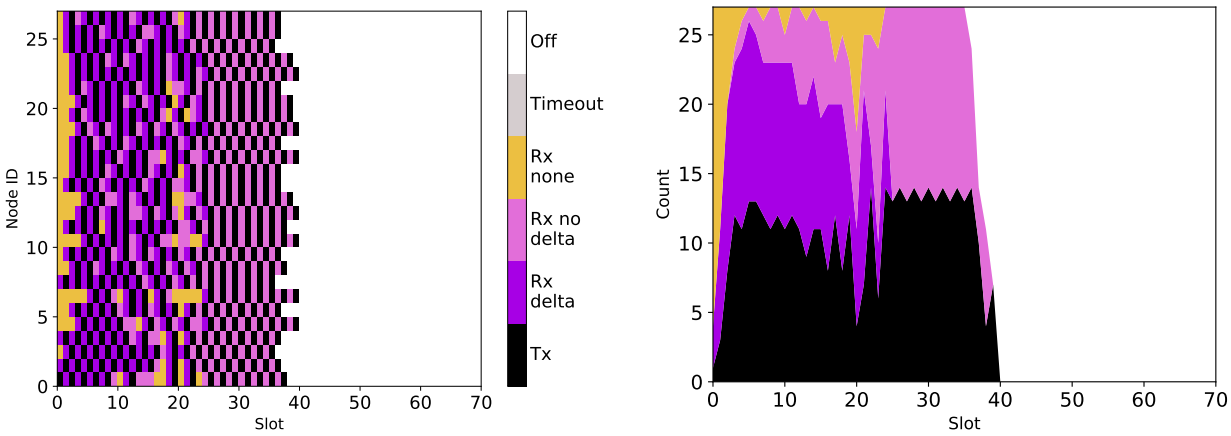
6.2.1.1 Cooja

We present our results when evaluating the Max protocol in Cooja. Figure 6.3 and 6.4 depict graphical illustrations of the node activity during a representative round, for the two individual systems. We also summarize the result metrics in Table 6.1 and compare our metrics of the A²-with-TSCH system against A²-Synchrotron.

As presented in Table 6.1, both systems achieve network-wide agreement for all 100 rounds when running the Max application in Cooja. A²-Synchrotron outperforms A²-with-TSCH for every metric and has approximately 33.37% less radio-on time, compared to our system. However, both systems have a similar *Complete latency* average and standard deviation. The *OFF latency* mean differs with around 10 timeslots and as can be seen in Figure 6.4a our system has a rather larger *OFF latency* standard deviation. This is because two nodes turn off their radio a great deal later compared to the other nodes.

6.2.1.2 FlockLab

In this section we present the FlockLab results when running the Max protocol. Figure 6.5 and 6.6 illustrate the different system results, by showing the activity in



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.4: *Cooja evaluation: Representative round when running Max in A²-Synchrotron.*

Cooja: Max	A2-with-TSCH	A2-Synchrotron
Consistency	100 %	100 %
Radio-on Time (Mean)	82.24 ms	54.80 ms
Radio Duty Cycle (Mean)	0.137 %	0.091 %
Complete Latency (Mean)	26.85 slots	22.89 slots
Complete Latency (Std. Dev)	1.6 slots	1.57 slots
OFF Latency (Mean)	48.41 slots	38.07 slots
OFF Latency (Std. Dev)	6.24 slots	1.41 slots

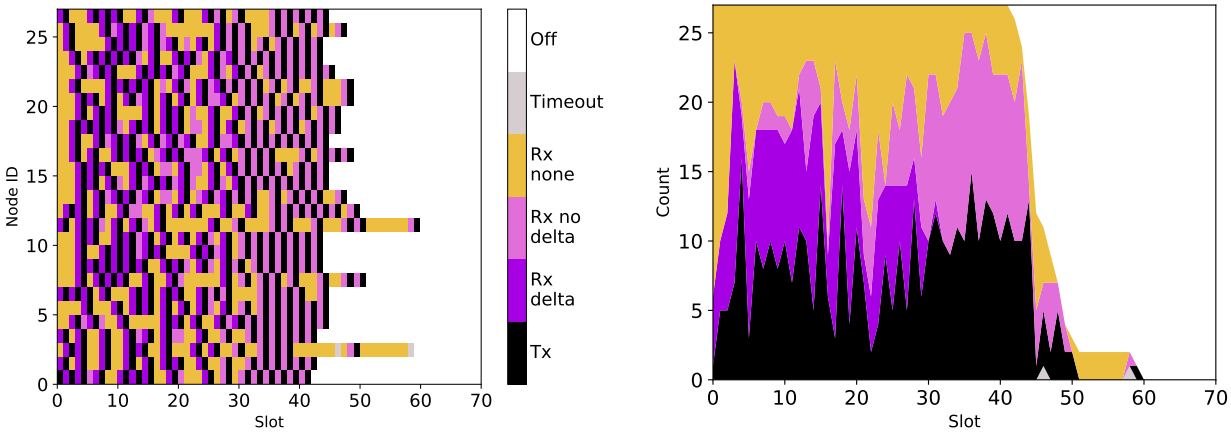
Table 6.1: *Metrics for the evaluation of Max in Cooja.*

each timeslot of a representative round. Table 6.2 presents the metrics corresponding to each figure.

As can be seen in Table 6.2, the *radio-on time* and *radio duty cycle* are slightly lower for A²-Synchrotron compared to A²-with-TSCH. The mean and the standard deviation for *Complete latency* and *OFF latency* are very similar for both systems.

6.2.1.3 Discussion

When performing evaluations of the Max protocol in Cooja, A²-Synchrotron surpasses A²-with-TSCH when looking at both the performance and energy consumption metrics. The figures 6.3b and 6.4b clearly show that A²-with-TSCH has a significantly higher rate of RX_NONES, compared to A²-Synchrotron. This delays the protocol progression and as a consequence, negatively affects the overall result metrics of our system. Additionally, the high rate of RX_NONES in the end phase of the protocol, causes some nodes to fall behind while a majority of nodes complete the protocol and turns off their radio. Nodes that fall behind will not receive any data during the last part of their flood phase, causing them to trigger consecutive



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.5: *FlockLab evaluation: Representative round when running Max in A²-with-TSCH.*

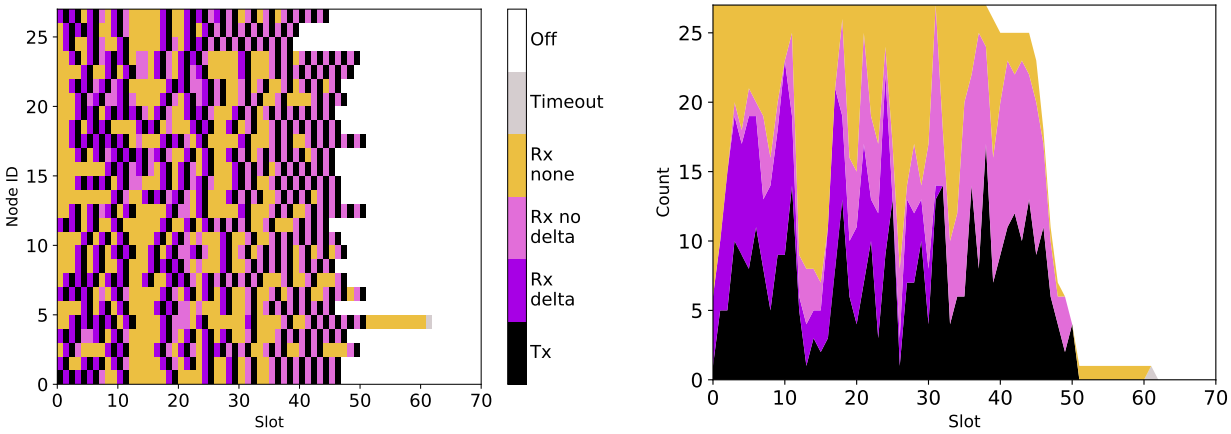
FlockLab: Max	A2-with-TSCH	A2-Synchrotron
Consistency	100 %	100 %
Radio-on Time (Mean)	82.46 ms	67.50 ms
Radio Duty Cycle (Mean)	0.137 %	0.113 %
Complete Latency (Mean)	27.70 slots	28.22 slots
Complete Latency (Std. Dev)	2.05 slots	2.20 slots
OFF Latency (Mean)	47.89 slots	47.56 slots
OFF Latency (Std. Dev)	4.23 slots	3.93 slots

Table 6.2: *Metrics for the evaluation of Max on FlockLab.*

timeout periods. This gives rise to the high *OFF latency* standard deviation observed in our system, which is not the case in A²-Synchrotron due to the low rate of RX_NONES.

When comparing the Max protocol results of the real-world evaluation on the FlockLab testbed, we find that A²-with-TSCH has similar performance metrics compared to A²-Synchrotron. Both systems display a matching high ratio of RX_NONES throughout the whole protocol round, which is likely the common denominator for their equivalent performance outcome. When considering the metrics related to energy consumption, A²-Synchrotron exhibits a lower radio-on time and radio duty cycle. This is due to the fact that Chaos keeps the radio on for a shorter period of time, during each timeslot, in contrast to TSCH.

We observe that A²-with-TSCH demonstrate resembling metrics in Cooja as well as on the FlockLab testbed, which is surprisingly not the case for A²-Synchrotron. It is difficult to draw any definitive conclusions on why A²-Synchrotron differs so greatly in terms of performance, between the simulation and the testbed evaluation. One aspect that might play a role in this, could be the network topology difference



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.6: *FlockLab evaluation: Representative round when running Max in A²-Synchrotron.*

between Cooja and FlockLab, as performance will differ from topology to topology. Additionally, Cooja is only a simulator which emulates the behavior of real-world nodes. Cooja is most likely not able to emulate all behaviors correctly, as there are multiple external factors which have to be taken into account. For example, even though Cooja simulates signal interference between transmitting nodes, there might be other devices present during the evaluation in FlockLab that operates in the 2.4 GHz spectrum; these devices might contribute with external interference not accounted for in Cooja.

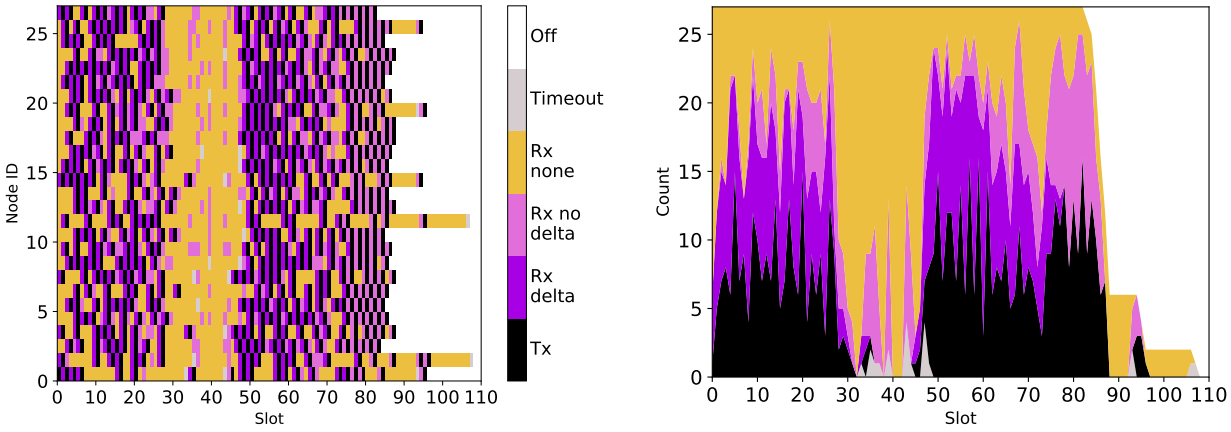
6.2.2 2PC

In this section we present the evaluation results when running 2PC on both Cooja and FlockLab. The evaluation setup is exactly the same as for the Max protocol.

6.2.2.1 Cooja

In the figures 6.7 and 6.8, we present the results of the Cooja evaluation in A²-with-TSCH and A²-Synchrotron. We demonstrate our results by showing the activity for each node in every timeslot and the overall activity distribution.

From Table 6.3, we again see that A²-Synchrotron outperforms our A²-with-TSCH systems in terms of performance when considering the average *Complete latency* and *OFF latency*. A²-Synchrotron experiences a timeout period around the time when nodes receive their final RX_DELTA, consequently contributing to the large *Complete latency* standard deviation. Additionally, A²-Synchrotron has approximately 42% less radio-on time in contrast to our system.



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.7: *Cooja evaluation: Representative round when running 2PC in A²-with-TSCH.*

Cooja: 2PC	A2-with-TSCH	A2-Synchrotron
Consistency	100 %	100 %
Radio-on Time (Mean)	179.28 ms	103.68 ms
Radio Duty Cycle (Mean)	0.299 %	0.173 %
Complete Latency (Mean)	74.81 slots	51.44 slots
Complete Latency (Std. Dev)	1.49 slots	5.31 slots
OFF Latency (Mean)	90.26 slots	63.33 slots
OFF Latency (Std. Dev)	6.22 slots	2.75 slots

Table 6.3: *Metrics for the evaluation of 2PC in Cooja.*

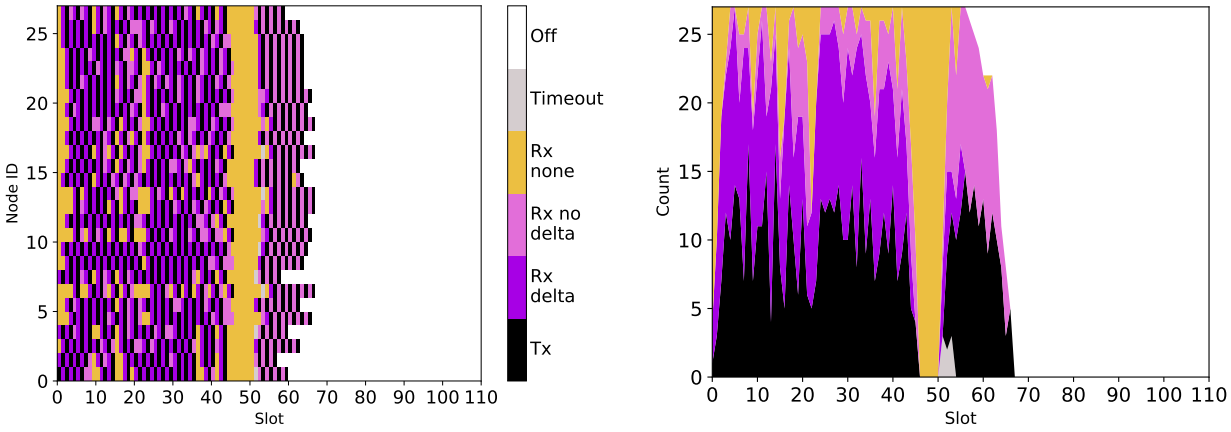
6.2.2.2 FlockLab

We evaluate the 2PC protocol for both systems, running on the FlockLab testbed. Figure 6.9 shows a graphical illustration corresponding to the A²-with-TSCH results, while Figure 6.10 showcases the results for the A²-Synchrotron system.

Our evaluation results of 2PC on FlockLab, illustrated in Table 6.4, show that A²-Synchrotron achieves a lower duty cycle and a lower radio-on time compared to A²-with-TSCH. However, A²-with-TSCH accomplish a lower *Complete latency* and *OFF latency*, while A²-Synchrotron shows a lower standard deviation in both *Complete latency* and *OFF latency*.

6.2.2.3 Discussion

When looking at the evaluation results of 2PC in Cooja, A²-Synchrotron outperforms A²-with-TSCH in a majority of the metrics, especially when considering energy consumption. The low energy consumption strongly relates to their low *OFF latency*, as nodes turn off their radio 27 timeslots earlier compared to A²-with-TSCH. Also, nodes in A²-Synchrotron keep their radio on for a smaller period of time in each



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.8: *Cooja evaluation: Representative round when running 2PC in A²-Synchrotron.*

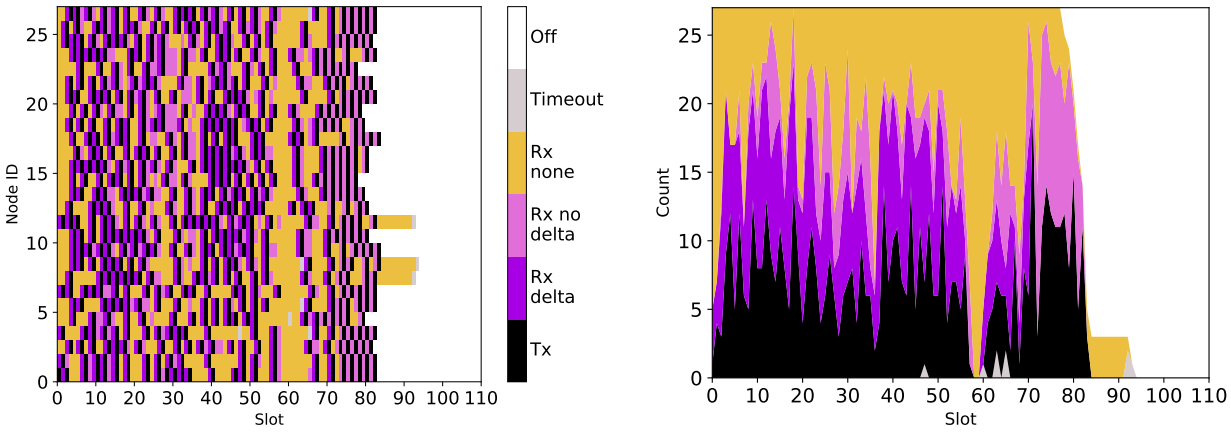
FlockLab: 2PC	A2-with-TSCH	A2-Synchrotron
Consistency	100 %	100 %
Radio-on Time (Mean)	169.61 ms	126.02 ms
Radio Duty Cycle (Mean)	0.283 %	0.210 %
Complete Latency (Mean)	69.33 slots	72.73 slots
Complete Latency (Std. Dev)	1.89 slots	1.32 slots
OFF Latency (Mean)	83.96 slots	86.04 slots
OFF Latency (Std. Dev)	4.01 slots	2.67 slots

Table 6.4: *Metrics for the evaluation of 2PC on FlockLab.*

timeslot, contributing to an overall lower energy consumption.

In the FlockLab evaluation, A²-with-TSCH and A²-Synchrotron are both similar performance-wise. Both systems have a matching *Complete latency* and *OFF latency* mean, indicating that our system is comparable with A²-Synchrotron on the FlockLab testbed. However, it would be bold of us to conclude that our system is at least as good as A²-Synchrotron, considering that we have only performed real-world tests on the FlockLab testbed. Additionally, as our tests have been limited to 27 nodes with an evaluation duration of 100 rounds, a more extensive evaluation needs to be performed before drawing any definitive conclusions.

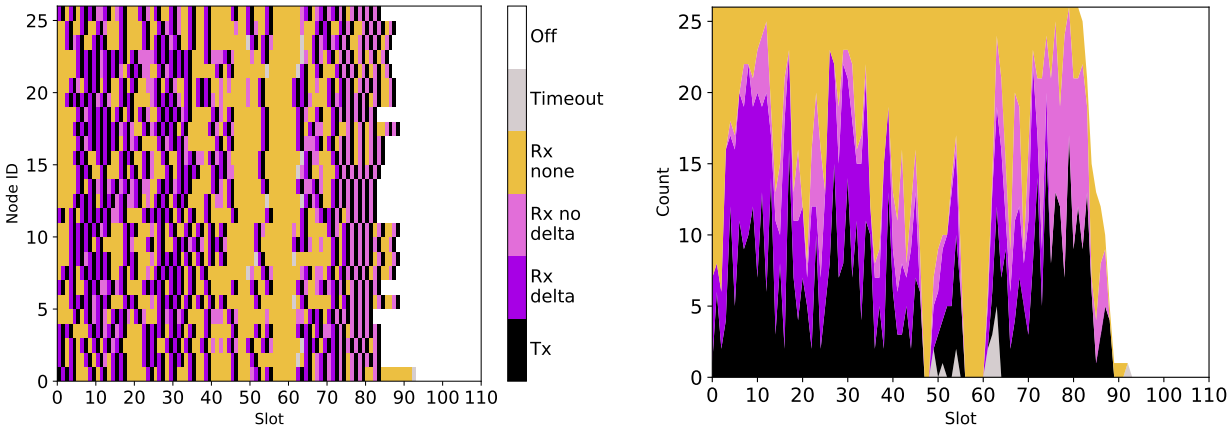
For A²-Synchrotron, both the 2PC and the Max evaluation differentiate significantly when comparing the results of the Cooja simulation against the real-world tests on FlockLab. In contrast, A²-with-TSCH demonstrates similar results when evaluating in Cooja and on FlockLab, for both Max and 2PC. We cannot say why this is the case, but as Cooja is a simulator, behaviors and external input may affect the evaluation when running on real hardware motes.



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.9: *FlockLab evaluation: Representative round when running 2PC in A²-with-TSCH.*



(a) *Activity in each timeslot for every individual node.*

(b) *Activity distribution in each timeslot for all nodes.*

Figure 6.10: *FlockLab evaluation: Representative round when running 2PC in A²-Synchrotron.*

7

Conclusion & Future Work

To conclude the work of this thesis, we start by presenting our contributions and subsequently discuss future work related to improvements that could be made to our system. Finally we consider ethical and sustainability questions linked to our project.

7.1 Conclusion

In this thesis we present a new system A^2-with-TSCH, providing network-wide consensus to applications running on top of TSCH in the Contiki [9] operating system. As the name suggests, A^2-with-TSCH is comprised of the already existing system Agreement in the Air (A^2) [1] on top of the highly reliable and synchronous MAC-layer protocol Time Slotted Channel Hopping (TSCH) [26]; primarily used in the context of low-powered wireless sensor networks. The final system currently supports two consensus protocols, Max and 2PC, which can run separately or as concurrent applications. The design structure of A^2-with-TSCH makes it easy for other developers to further extend the system by, for example, porting the remaining A^2 protocols: Vote and 3PC, or implement other A^2 protocols of their own.

We evaluate A^2-with-TSCH against A^2-Synchrotron in the Contiki simulator Cooja [21] and on the testbed, FlockLab [19]. The results show that A^2-Synchrotron outperforms A^2-with-TSCH with respect to power consumption, in both Cooja and Flocklab. Nevertheless, A^2-with-TSCH shows similar performance results compared to A^2-Synchrotron, on the Flocklab testbed.

7.2 Future Work

As of now, our final implementation of the A^2-with-TSCH system only supports the A^2 protocols Max and 2PC. Future work could be to include the remaining consensus protocols, Vote and 3PC, of the A^2-Synchrotron system. Developers can also implement their own A^2 protocols and integrate them into our system, to further extend the system.

Our system is limited to only run in a static context, meaning that all nodes need to be aware of each other during compile time for consensus to be reached; this restricts new nodes from joining the network and also restricts nodes from leaving

the network, by for example crashing. The option of only running in a static context and the lack of fault-tolerance, is probably the greatest drawback of our system and is therefore something which can be improved upon. A^2-Synchrotron incorporates a join and leave service to tackle the aforementioned issues of A^2-with-TSCH, and can therefore be used as a template to further improve our system.

Another topic of interest would be to test the scalability of our A^2-with-TSCH system. We were only able to evaluate our system for 27 nodes on the FlockLab testbed due to time restrictions, which limits our knowledge regarding scalability. It would be interesting to see how our system performs in terms of performance and power consumption with an increasing number of nodes, and to also see whether the system is of any practical use in large-scale WSNs.

7.3 Ethics and Sustainability

Nowadays sensor networks are used everywhere to monitor and gather data related to environmental conditions of their surroundings. This data can then be used to aid or to help optimize the way we do things. Consider an example where a sensor network monitors the power consumption inside a building. The gathered data can, for example, map different areas with their corresponding power consumption and pinpoint problem areas where optimizations can be done. Wireless sensor networks can also be used to measure water-quality and air-pollution levels. This data can reveal whether our way of living is sustainable or if it has detrimental impacts on the environment, and as a consequence might educate people how to make better life choices. Additionally, wireless sensor networks often set out to be as energy efficient as possible, as a majority of sensor networks rely on battery for power. Following these facts, it is easy to see that wireless sensor networks can be of help when trying to solve sustainability issues in many different areas, and they also consume low amounts of energy while doing so.

When considering wireless sensor networks, we also need to acknowledge the ethical aspects. As wireless sensor networks are able to monitor their surroundings, they also have the potential of surveilling people and gather sensitive data without their knowledge. Such sensitive data could, for example, be used to track people's whereabouts or it could conceivably be used for extortion. Wireless sensor networks also have a wide-spread use in critical infrastructures. If the security is poor for these sensor networks, an adversary could collect classified information or take down a critical system, consequently inflicting harm or subjecting people's lives to danger.

Bibliography

[1] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. "Network-wide consensus utilizing the capture effect in low-power wireless networks". In: *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys 2017)*. 2017, p. 14.

[2] Carlo Alberto Boano, Marco Antonio Zuniga, Kay Römer, and Thiemo Voigt. "Jag: Reliable and predictable wireless agreement under external radio interference". In: *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*. IEEE. 2012, pp. 315–326.

[3] *Contiki-NG: The OS for Next Generation IoT Devices*. `http://contiki-ng.org/`. 2017. (Visited on 05/31/2018).

[4] Moteiv Corporation. *Tmote Sky: Datasheet*. 2006.

[5] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson education, 2005.

[6] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong. "Splash: Fast data dissemination with constructive interference in wireless sensor networks." In: *NSDI*. 2013, pp. 269–282.

[7] Wenliang Du, Jing Deng, Yunghsiang S Han, Shigang Chen, and Pramod K Varshney. "A key management scheme for wireless sensor networks using deployment knowledge". In: *INFOCOM 2004. Twenty-third Annual Joint conference of the IEEE computer and communications societies*. Vol. 1. IEEE. 2004.

[8] Wenliang Du, Jing Deng, Yunghsiang S Han, Pramod K Varshney, Jonathan Katz, and Aram Khalili. "A pairwise key predistribution scheme for wireless sensor networks". In: *ACM Transactions on Information and System Security (TISSEC)* 8.2 (2005), pp. 228–258.

[9] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pp. 455–462.

[10] Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. "Software-based on-line energy estimation for sensor nodes". In: *Proceed-*

ings of the 4th workshop on Embedded networked sensors. ACM. 2007, pp. 28–32.

[11] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. ACM. 2006, pp. 29–42.

[12] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. "Low-power wireless bus". In: *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. ACM. 2012, pp. 1–14.

[13] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. "Efficient network flooding and time synchronization with glossy". In: *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE. 2011, pp. 73–84.

[14] James Gray. "Notes on data base operating systems". In: *Operating Systems*. Springer, 1978, pp. 393–481.

[15] MEMSIC Incorporation. *TelosB: Datasheet*. 2003.

[16] Venkatraman Iyer, Matthias Woehrle, and Koen Langendoen. "Chrysso — A multi-channel approach to mitigate external interference". In: *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*. IEEE. 2011, pp. 449–457.

[17] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. "Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale". In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2013, p. 1.

[18] Krijn Leentvaar and Jan Flint. "The capture effect in FM receivers". In: *IEEE Transactions on Communications* 24.5 (1976), pp. 531–539.

[19] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. "Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems". In: *Proceedings of the 12th international conference on Information processing in sensor networks*. ACM. 2013, pp. 153–166.

[20] Luca Mainetti, Luigi Patrono, and Antonio Vilei. "Evolution of wireless sensor networks towards the internet of things: A survey". In: *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*. IEEE. 2011, pp. 1–6.

[21] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. "Cross-level sensor network simulation with Cooja". In: *Local computer networks, proceedings 2006 31st IEEE conference on*. IEEE. 2006, pp. 641–648.

[22] Ragunathan Raj Rajkumar, Insup Lee, Lui Sha, and John Stankovic. "Cyber-physical systems: the next computing revolution". In: *Proceedings of the 47th design automation conference*. ACM. 2010, pp. 731–736.

[23] Makoto Suzuki, Yasuta Yamashita, and Hiroyuki Morikawa. "Low-power, end-to-end reliable collection using glossy for wireless sensor networks". In: *Vehicular Technology Conference (VTC Spring), 2013 IEEE 77th*. IEEE. 2013, pp. 1–5.

[24] Thiemo Voigt and Fredrik Osterlind. "CoReDac: Collision-free command-response data collection". In: *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*. IEEE. 2008, pp. 967–973.

[25] Qin Wang, Xavier Vilajosana, and Thomas Watteyne. *6top Protocol (6P)*. Tech. rep. Internet Engineering Task Force, Mar. 2018. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-6tisch-6top-protocol-10`.

[26] Thomas Watteyne, Maria Rita Palattella, and Luigi Alfredo Grieco. *Using IEEE 802.15.4e Time-Slotted Channel Hopping (TSCH) in the Internet of Things (IoT): Problem statement*. RFC 7554. RFC Editor, May 2015, pp. 1–23. URL: `https://tools.ietf.org/html/rfc7554`.

[27] Fang-Jing Wu, Yu-Fen Kao, and Yu-Chee Tseng. "From wireless sensor networks towards cyber physical systems". In: *Pervasive and Mobile computing* 7.4 (Aug. 2011). Elsevier, pp. 397–413.

[28] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. "Wireless sensor network survey". In: *Computer networks* 52.12 (Aug. 2008). Elsevier, pp. 2292–2330.