TU WIEN Informatics

IoT Implementation and Evaluation of Distributed Consensus Algorithms

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Christoph Echtinger-Sieghart, BSc.
Matrikelnummer 00304130

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Prof. Ulrich Schmid

Wien, 2. Dezember 2021

Christoph Echtinger-Sieghart	Ulrich Schmid

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

IoT Implementation and Evaluation of Distributed Consensus Algorithms

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Christoph Echtinger-Sieghart, BSc.
Registration Number 00304130

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Ulrich Schmid

Vienna, 2nd December, 2021

Christoph Echtinger-Sieghart	Ulrich Schmid

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Christoph Echtinger-Sieghart, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Dezember 2021

Christoph Echtinger-Sieghart

Danksagung

Ich möchte mich bei Prof. Schmid für seine Unterstützung bedanken – seine Ratschläge, seine Anleitung und seine konstruktive Kritik waren immer willkommen.

Ich möchte mich auch bei meinen Eltern, meiner Frau und meinen Kindern bedanken – ihr alle habt dazu beigetragen, dass diese Arbeit geschrieben worden ist.

Acknowledgements

I would like to extend my gratitude to Prof. Schmid for his support – his advice and guidance were always welcome as was his constructive criticism.

I would also like to thank my parents, my wife and my children – you all played a part in me finishing this thesis.

Kurzfassung

Diese Arbeit beschäftigt sich mit Konsensus-Algorithmen, die sich für die Implementierung auf ressourcenbeschränkten Geräten in hochdynamischen drahtlosen Netzwerken eignen – Bedingungen, wie sie in unterschiedlichen Internet of Things (IoT) Umgebungen vorherrschen. Konsensus ist ein klassisches Problem des „Distributed Computing“ – wir besprechen hier sowohl die terminierende als auch die stabilisierende Variante. Diese Arbeit präsentiert zuerst das Systemmodell und die wichtigsten Konzepte einer Klasse von „root component“ basierenden Algorithmen – eingeführt in [SS21] und [WSS19] – die sowohl das terminierende als auch das stabilisierende Konsensus Problem in dynamischen Netzwerken unter „eventually stabilizing message adversaries“ lösen.

Nachdem wir die theoretischen Aspekte der „root component“ basierenden Algorithmen im Detail besprochen haben, geben wir einen kurzen Überblick über den MinMax Konsensus-Algorithmus von [CM21] und bringen einige der wichtigsten Konzepte der „root component“ basierenden Algorithmen in Relation zu zentralen Konzepten des MinMax Algorithmus.

Nach dem theoretischen Abschnitt gibt diese Arbeit eine detaillierte Beschreibung einer Implementierung des „root component“ basierenden Algorithmus und des MinMax Konsensus-Algorithmus. Weiters beschreibt diese Arbeit eine Simulationsumgebung, die entwickelt wurde um die Implementierung der Algorithmen zu unterstützen. Abschließend beschreiben wir die Resultate eines experimentellen Vergleichs des „root component“ basierenden Ansatzes mit dem MinMax Ansatz in Hinblick auf die Stabilisierungszeit der Algorithmen.

Abstract

This thesis is concerned with consensus algorithms suitable for implementation on resource-constrained devices operating in highly dynamic wireless networks – conditions that can be found in various IoT environments. Consensus is a classical problem in the field of distributed computing – we discuss both its terminating and its stabilizing variant. This thesis starts by presenting the system model and core concepts of a class of root component based algorithms – introduced in [SS21] and [WSS19] – that solve both the terminating and the stabilizing consensus problem in dynamic networks under eventually stabilizing message adversaries.

Following the theoretical aspects of the root component based algorithms, we give a brief discussion of the MinMax consensus algorithm by [CM21] and put some central system model concepts of the MinMax algorithm in relation to core concepts of the root component based stabilizing algorithm.

After the theoretical part, this thesis provides a detailed account of an actual implementation of the root component based stabilizing consensus algorithm and the MinMax consensus algorithm. This thesis also discusses the implementation of a simulation environment that was created to aid development. Finally we report on an experimental side-by-side evaluation of the root component based algorithm and the MinMax algorithm with regard to stabilization time.

Contents

CHAPTER 1

Introduction

Everything starts somewhere, although many physicists disagree.

— Terry Pratchett, *"Hogfather", p. 1*

This thesis is concerned with consensus algorithms – both from a theoretical and from a practical viewpoint. [SS21], based on the work of [WSS19], introduced two root component based round-oblivious consensus algorithms – one for terminating consensus, one for stabilizing consensus – operating under related message adversaries.

Initial experiments conducted in [PS16] showed a similar message adversary to be a good match for real world conditions in synchronous Wireless Sensor Networks (WSNs). Since consensus may play a role in emerging IoT systems – which are, in a sense, wireless sensor and actuator networks with resource limited nodes – a practical implementation of these algorithms is of interest. By implementing the algorithms, we expected to gain a deeper understanding of the engineering obstacles one needs to overcome to make use of these recent discoveries in real-world applications. To the best of our knowledge, there is little to no tooling available to aid the developer in implementing such algorithms. As a side product of the implementation, a framework to simulate and test the consensus algorithms was created.

In addition to the implementation work, this thesis provides an experimental side-by-side comparison of the root component based stabilizing algorithm from [SS21] and the MinMax stabilizing algorithm introduced in [CM21] under real world conditions. Viewed from a theoretical standpoint the MinMax stabilizing consensus algorithm reveals that the problem solved by the root component based stabilizing consensus algorithm can be solved for a different adversarial model that does not require nodes with unique identifiers. This thesis also explores the differences in the models and computations behind the MinMax and the root component based stabilizing consensus algorithm.

1.1 The consensus problem

The problem of consensus is a central one in the field of distributed computing. Consensus is important in all scenarios, where independent processors need to agree on a common value [AW04]. Achieving distributed consensus becomes hard under unreliable communication conditions, such as are present in WSNs. These systems need practical implementations of consensus algorithms able to cope with limited resources and unreliable networks.

We will now give a small recap of the consensus problem in two variants – **stabilizing consensus** and **terminating consensus**.

The basic assumptions behind these two variants are the same. Processes are given initial values and are tasked to decide unanimously on one of these values. The validity condition is the same for both variants: processes are only allowed to decide on a value that was the input of some process. The agreement condition states that the decision value has to be chosen unanimously. Terminating and stabilizing consensus differ, in that terminating consensus demands agreement at all points in time and mandates that processes are only allowed to write the output variable once. Under stabilizing consensus, there just has to exist a time t, after which the agreement condition holds, albeit processes are allowed to write the output variable more than once.

We call the set of participating processes $\Pi = \{1, \ldots, N\}$ and for a process $p \in \Pi$, denote the input value by $x_p \in V$ and the output value by $y_p \in V \cup \{\perp\}$, where V is a totally ordered set and $\perp \notin V$ represents "undecided yet".

1.1.1 Stabilizing Consensus

Definition 1.1.1 (Stabilizing consensus)**.** *The problem of stabilizing consensus requires the following conditions to hold:*

Validity $\forall i \in \Pi, \exists j \in \Pi : y_i = x_j$

Eventual Agreement *There is a time t after which* $\forall i, j \in \Pi : y_i = y_j$

The problem of stabilizing consensus requires the conditions **Validity** and **Eventual Agreement** to hold for the set of participating processes Π. Recall that stabilizing consensus allows a process $p \in \Pi$ to write the output value y_p more than once.

1.1.2 Terminating Consensus

Definition 1.1.2 (Terminating consensus)**.** *The problem of terminating consensus requires the following conditions to hold:*

Validity $\forall i \in \Pi, \exists j \in \Pi :$ *if* $y_i \neq \perp$, *then* $y_i = x_j$

Agreement $\forall i, j \in \Pi :$ *if* $y_i \neq \perp, y_j \neq \perp$, *then* $y_i = y_j$

Termination *Every process p eventually writes once to y_p*

The problem of terminating consensus requires the conditions **Validity**, **Agreement** and **Termination** to hold for the set of participating processes Π. In contrast to stabilizing consensus, the condition **Agreement** has to hold at all times and a process $p \in \Pi$ is required to write the output value y_p exactly once.

1.2 Related work

1.2.1 Consensus

The consensus problem comes in variants – terminating consensus, asymptotic consensus, stabilizing consensus. Formal definitions for terminating and stabilizing consensus have been given in Section 1.1. Asymptotic consensus weakens the validity condition and mandates that the participants are only required to asymptotically converge on a common value. This variant of consensus is important in the field of multi-agent systems and used, amongst other things, for sensor data fusion – see [SM03], [OSFM07]. Extensive work on asymptotic consensus and its variants has been done by [LZKS13], [CFN15], [Now15], [CBFN16], [FNS18] and [FN18]. [LZKS13] introduced a novel algorithm able to solve asymptotic consensus with a special focus on resilience against targeted attacks on nodes. The authors further introduced the notion of network-robustness as a graph-theoretic property. [CFN15] investigated the solvability of approximate consensus in dynamic networks and arrived at the result that for approximate consensus to be solvable each round's communication graph has to have a rooted spanning tree. [Now15] investigated the asymptotic consensus problem in a setting where nodes do not necessarily trust their own values and introduced the concept of aperiodic cores to adapt existing convergence results to the loss of self-confidence. [CBFN16] introduced a new technique to enhance existing averaging algorithms, leading to a class of algorithms termed amortized averaging algorithms. The authors further introduced a new algorithm – the amortized midpoint algorithm and showed that its results degrade gracefully even under network model violations. [FNS18] derived tight bounds for contraction rate and time complexity of asymptotic consensus algorithms and showed that the algorithms from [CBFN16] are optimal. [FN18] introduced new algorithms able to solve asymptotic and approximate consensus for agents with multidimensional values, whose time-complexity is not dependent on the size of the value vector.

The problem of clock synchronization can also be viewed as a variety of the asymptotic consensus problem. See Section 1.2.2 for references regarding asymptotic consensus applied to the problem of clock synchronization.

Terminating consensus is considered a classical problem in computer science – especially in the field of distributed computing – and has a history of more than 40 years of research [Fis83]. See [AW04, Chapter 5] for an introduction.

The problem of stabilizing consensus was first introduced in [AFJ06]. Additional work on stabilizing consensus was done in [DGM+11], which described a median-based update rule to solve stabilizing consensus in a non-adversarial setting and showed that using the median-based rule, almost stable consensus can be reached in an adversarial setting. [BCN+16] showed that using a 3-majority based rule, almost stable stabilizing consensus is possible under an adversarial model. Recent progress in stabilizing consensus has been made in the area of directed dynamic networks by [SS21] and [CM21].

Since this thesis is concerned with terminating and stabilizing consensus in directed dynamic networks, we will now direct our attention to theoretical work with a focus on directed dynamic networks. Work was done, amongst others, by [BRS12], [SWS16], [BRS+18], [WSS19], [SS21] and [CM21]. [BRS12] introduced the notion of vertex-stable stable root components in the communication graph and provided and proved correct an algorithm that solves consensus under these assumptions. [SWS16] improved on the previous results regarding consensus algorithms for message adversaries that guarantee a vertex-stable root component. [BRS+18] investigated consensus under a message adversary that allows several vertex-stable root components and in addition took a look at k-set agreement and its impossibility/solvability border. [WSS19] investigated the problem of consensus in dynamic networks under non-oblivious message adversaries and presented and proved a novel consensus algorithm. [SS21] improved the algorithm from [WSS19] to be round-oblivious and introduced a novel stabilizing consensus algorithm. [CM21] introduced the MinMax family of algorithms for solving the problem of stabilizing consensus. For an overview of recent theoretical advances in the field of consensus in directed dynamic networks see [WS19].

Implementation aspects of consensus protocols

When implementing an algorithm using a given technology, it is often advisable to adapt the algorithm to exploit domain-specific characteristics. [LB03] discussed the implementation of a consensus protocol for Controller Area Network (CAN) based systems. The authors discussed scenarios, where a CAN network can become inconsistent, and then developed a consensus protocol that can tolerate these inconsistent scenarios. [Kö13] described a network stack that makes it easy to reach consensus in building automation networks by developing a custom Media Access Control (MAC) layer and a matching routing protocol optimized for the requirements of a consensus protocol. [Lin15] investigated design choices and trade-offs that should be considered when designing (asymptotic) consensus algorithms for the use in WSNs.

[KKRF11] implemented an average consensus algorithm under TinyOS on IRIS motes[1]. [Man13] described the implementation of a multi-hop dynamic consensus algorithm, relying on ZigBee as the wireless transport technology. Routing of messages was implemented using Ad-hoc On-demand Distance Vector (AODV). [Sup18] reported on a

[1] http://www.nr2.ufpr.br/~adc/documentos/iris_datasheet.pdf

bare-metal implementation of a consensus protocol for an ESP8266 microcontroller with RF-communication done in the 433 MHz band.

[ANRE15] described the implementation of a binary consensus algorithm – again using IRIS motes and running TinyOS. The authors performed simulations under TOSSIM and real world experiments with up to 139 nodes. [AEF14] implemented a binary consensus protocol on MoteLab hardware and compared simulation results in TOSSIM with the results obtained from real-world runs.

[PS16] conducted extensive experiments that showed a message adversary, similar to the ones presented in [WSS19] and [SS21], to be a good match for real world connectivity conditions in a WSN. The authors developed and successfully deployed a connectivity monitoring framework for Atmel RZ200 motes running TinyOS and using IEEE 802.15.4 in the 2.4 GHz band for wireless connectivity. The monitoring framework was designed to record the connectivity information per round for each node in the WSN. To facilitate synchronous rounds, global time was established inside the WSN using Flooding Time Synchronization Protocol (FTSP). The collected connectivity data was forwarded to a stationary PC for long term storage and to allow post-processing tools to analyze the captured data and check certain properties. The framework took care to allow the monitoring data to be retrieved in a reliable way (in the face of message loss or network partition) and also had a means to inject errors. Online fault injection was done by sending special fault injection messages to nodes, thereby instructing them to drop all messages from a given list of nodes until further notice. Two different locations were evaluated – an office/lab environment and the roof-top of a university building. Experiments run in both locations (using transmission scheduling to avoid massive packet loss) show a high coverage for a wide range of values for D (the dynamic network depth) for the message adversary $\Diamond\mathsf{STABLE}(D)$ (see [PS16, Definition 3]).

1.2.2 Clock synchronization

Since the algorithms presented in Chapter 3 and Chapter 4 assume a round-based communication scheme, we are going to need some way to establish global time using clock synchronization. Clock synchronization is an important problem and various solutions for different kinds of networks have been proposed over time. The most widely used clock synchronization protocol today is Network Time Protocol (NTP) – see RFC 5905. NTP was designed for traditional (wired) networks like the Internet. It is widely and successfully deployed, but not suitable for application in WSNs, because it ignores key issues like energy efficiency and the dynamics inherent in WSNs, both in network layout and configuration. [ER03] defined the problem of time synchronization in WSNs and discussed in detail why protocols like NTP are not a good fit for WSNs.

Many solutions to the problem of time synchronization in WSNs have been proposed. [EGE02] introduced Reference-Broadcast Synchronization (RBS), where nodes broadcast reference beacons (containing no timestamps) and upon reception of a beacon take a timestamp. All recipients exchange information about their respective reception times

and from this information are able to establish a global time. [GKS03] described the Timing-sync Protocol for Sensor Networks (TPSN). The basic idea behind TPSN is to first elect a reference node and then establish a spanning tree over the network. Nodes then perform pairwise synchronization along the edges of the rooted tree. In the end, all nodes will be synchronized to the clock of the reference node. [MKSL04] introduced FTSP. FTSP can be considered the current de-facto standard for time synchronization in WSNs, indicated by the fact that papers presenting new developments often compare their results to FTSP. FTSP works by dynamically electing a root-node and using timestamped messages, broadcast by the root, in combination with linear regression to predict clock skews. For a more detailed introduction to FTSP see Section 5.3.1. Various improvements to FTSP have been published. [CJM05] described an FTSP implementation specific to ZigBee, which improved FTSP's energy efficiency by using domain-specific knowledge of the ZigBee protocol suite - mainly the master-slave communication mode. [YK14] improved the accuracy of the slow-flooding approach used by FTSP, by utilizing a clock-speed agreement algorithm to keep the estimated errors low – leading to the Flooding with Clock Speed Agreement (FCSA) approach. An approach based on rapid-flooding (in contrast to the slow flooding approach utilized by FTSP) was introduced in [LSW15] and named PulseSync. PulseSync is able to outperform FTSP in some scenarios, but has drawbacks related to network contention due to its rapid-flooding approach.

There are also a variety of asymptotic consensus based time synchronization approaches. [SG07] introduced Average TimeSync (ATS). ATS is fully distributed and does not elect a root node. It employs an average consensus algorithm and does skew compensation and MAC layer timestamping. One drawback of ATS is that it only establishes a logical timebase among the nodes and is not capable of using an external time signal as a source of truth. [HCS+14] described two consensus based approaches to clock synchronization – Maximum Time Synchronization (MTS) and Weighted Maximum Time Synchronization (WMTS) – both based on the idea of max-consensus introduced in [SM03, Equation 7].

The topic of clock synchronization is a vast field and not all approaches are presented here. Both [Fai07] and [SW14] provide a good overview of clock synchronization approaches specific to WSNs. [DB16] discusses general issues of implementation with a focus on existing published implementations of time synchronization protocols.

1.3 Organization and contributions of the thesis

[SS21] introduced the idea of using the root component based technique developed in [WSS19] to solve stabilizing consensus and further improved on the algorithms from [WSS19] by making them round-oblivious. This thesis provides real-world implementations of both root component based algorithms from [SS21] and the MinMax algorithm from [CM21] and a matching simulation environment. In addition it reports on experiments performed to compare the MinMax and root component based stabilizing algorithms under real world conditions with a special focus on stabilization time. This thesis further sets core system model concepts and computations behind the root compo-

nent based stabilizing consensus algorithm and the MinMax based algorithm in relation to each other.

The thesis is conceptually split into two parts – a theoretical and a practical part. The theoretical part comprises Chapters 2 to 4. Chapters 5 to 6 and Appendix A make up the practical part. The theoretical part first introduces the system model, goes on to describe the root component based algorithms and then gives insights into the differences between the root component based algorithms from [SS21] and the MinMax algorithms described in [CM21]. The practical part first discusses the various underlying technologies used and then provides a discussion of the implementation of the root component based algorithms. Last but not least, the simulation environment is introduced and the evaluation results are discussed. Appendix A serves as a small tutorial for future developers to get up to speed with the implementation.

CHAPTER 2

System model

She's a model and she's looking good
— Kraftwerk, *"The Model" (1981)*

This chapter introduces the system model behind the root component based algorithms from [SS21] and describes how the environment, in which the algorithms are trying to reach consensus, essentially behaves. In addition, we will introduce important graph-theoretical concepts, necessary to understand the inner workings of the root component based algorithms. Our description of the system model and the related definitions closely follow the descriptions and definitions from [SS21] and [WSS19].

2.1 Lock-step synchronous rounds

We call the participants trying to reach consensus processes and denote the set of all participating processes with $\Pi = \{1, 2, 3, \ldots, N\}$. We further assume that all processes are fault free and have a unique id, taken from the set Π, but do not necessarily know N.

In our system model, a process' time passes in the form of lock-step synchronous rounds. The algorithms make progress by moving from one round to the next. These steps are synchronized between all active processes. For all active processes $p, q \in \Pi$ it holds that p and q start a round at exactly the same point in time.

All processes start in a passive state. Passive processes can make the transition to being active. A transition from passive to active can only happen when a new round starts. Once a process becomes active, it can never go back to being passive. Our system model behaves differently depending on the consensus algorithm that the processes are executing. For terminating consensus the environment is only able to make all processes do the state transition from passive to active at the same time. When running the stabilizing

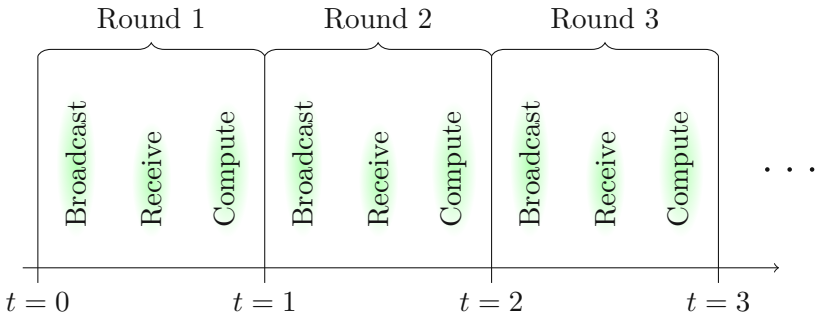


Figure 2.1: Progression of time structured in rounds

consensus algorithm, the environment allows the processes to transition from passive to active at different times. We call the time a process p becomes active a_p and denote with Π_a the set of all active processes.

All messages sent in round r are either received in the same round r or not received at all. Conceptually, rounds are further split into a number of phases. The first phase is the broadcast phase. In this phase all processes broadcast the message they wish to send in the current round. The receive phase follows the broadcast phase and processes now receive and process messages sent by other processes in the preceding broadcast phase. The computation phase is the last phase in a round and processes now execute the algorithm's main computation. Figure 2.1 illustrates how time progresses structured in rounds.

Rounds are numbered starting with 1. We use the special round 0 to refer to the state of a process before execution starts – the process' initial state. When talking about the state of a process in round r, we refer to the state of the process after having finished the computation phase of round r and before starting the broadcast phase of round $r + 1$ – a well-defined state exists only in-between rounds. Since the root component based algorithms are round-oblivious, the actual round number is of no importance to the processes executing the algorithms.

We follow the notational convention to use a superscript for the round number and a subscript to refer to a process. x_p^r refers to the variable x of process p captured at the end of round r.

See Section 5.3 for a detailed description of a possible way to implement processes executing in lock-step synchronous rounds.

2.2 Communication

Processes communicate with each other by sending messages. All communication is broadcast without acknowledgements – there is no way to address a message to a specific process. In contrast to the assumption of fault-free processes, we assume that

communication is unreliable. Communication is unreliable in the sense that messages might get lost and not be delivered at all and that the underlying communication graph changes over time. Note that the concept of lost messages is strong enough to simulate process crashes – a process whose messages are not received by any other process after some time is indistinguishable from a process that has crashed, since messages are the only means of exchanging information. In round r, all the received messages will at most be based on state from round $r-1$, because at the time of broadcasting in round r the round r state is not yet fully available.

A directed graph $G = (\Pi, E)$ captures the static relationship of processes with regard to message reception in a single round. We often use the terms node and process when talking about the vertices in the communication graph. An edge $(p, q) \in E$ encodes the fact that process q is able to receive messages from process p, but not necessarily vice versa. By definition, there are no edges in the communication graph starting or terminating at a passive process. We assume our communication graphs to be reflexive, meaning each active vertex has a loop, for reasons that will become clear when introducing the concept of a compound graph. The self-loops fit well with the concept of message reception, because every process is – in a way – able to receive its own messages. When illustrating concepts by drawing graphs, we will often omit the loops for reasons of clarity. We sometimes sloppily write $(q, p) \in G$, when we actually mean $G = (\Pi, E)$ and $(q, p) \in E$.

To model communication conditions that change over time, we employ the concept of a Directed Dynamic Network (DDN) – for more information regarding DDNs see [KO11]. We are going to represent the execution of a DDN as a sequence of graphs $(G^i)_{i\in\mathbb{N}^+}$. We associate each graph from the DDN with one round and call $G^r = (\Pi, E^r)$ the communication graph of round r with the edge set E^r. We write $(G^r)_{r=a}^{b}$ with $a \leq b$ to refer to a continuous sequence of graphs starting at round a up to round b with a length of $b - a + 1$.

As a notational convention, when talking about sequences of graphs, we generally use the Greek letter σ. We write $\sigma' \subseteq \sigma$ to mean that σ' is a subsequence of the sequence σ. Π_σ is the set of processes that appear in the graphs of sequence σ.

The time-varying topology concept allows us to model the complex communication conditions one might experience in a wireless sensor network. Due to effects like multipath-fading and interference, communications links cannot be assumed to be bidirectional by default. We use DDNs to primarily model changes in the interconnection topology and not so much changes in the set of vertices, albeit the fact that N is generally unknown to the processes also allows the latter.

2.2.1 Unreliable communication and message adversaries

We need a way to specify how "unreliable" our communication is, because without any limitations on the graph topology – e.g. a network, where all nodes are isolated – no interesting problems can be solved. This is where the concept of a message adversary

[AG13] comes into play. A message adversary can be thought of as an all-knowing entity, which is allowed to choose the communication links that are active in a given round r and who tries to hinder the processes from reaching consensus. In other words, a message adversary is allowed to choose the communication graph G^r for every round r. Formally, message adversaries are defined as sets of communication graphs or sets of sequences of communication graphs. Message adversaries cannot have unlimited power, but must adhere to some specification.

Message adversaries come in two flavours – oblivious and non-oblivious. Oblivious message adversaries are allowed to choose the communication graphs at will from a predefined set of admissible communication graphs.

Non-oblivious message adversaries choose the communication graphs from a – possibly infinite – set of graph sequences. Another way to think about non-oblivious message adversaries is to imagine additional constraints on which graphs can be chosen, depending on the round number r. Non-oblivious message adversaries allow us to specify liveness and safety properties on the graph sequences they generate. The algorithms in this thesis are exclusively concerned with non-oblivious message adversaries. Definitions 2.4.4, 2.4.5 and 4.1.6 provide the specifications of the message adversaries relevant to this thesis.

2.3 Graph and DDN concepts

Now that we know how the environment behaves, we will introduce concepts that lead us to an understanding of the basic ideas behind the algorithms.

2.3.1 Root Component

A concept of central importance to the algorithms from [SS21] and [WSS19] is that of a root component.

Definition 2.3.1 (Root component)**.** *A root component R of a directed graph $G = (V, E)$ is a set of vertices, which form a strongly connected component with the additional requirement that $\forall p \in V, \forall q \in R : (p, q) \in E \implies p \in R$ holds.*

Loosely speaking, a root component is a strongly connected component, whose member nodes have no incoming edges with an origin vertex that is not itself part of the root component.

All graphs have at least one root component and at most $|V|$. The set of root components of a graph G is denoted with $\mathsf{Roots}(G)$. We call a graph G a rooted graph, if it has exactly one root component and denote the root component's members with $\mathsf{Root}(G)$. A rooted graph has the useful property that, for every vertex from the root component, there exists a path to every other vertex in the graph. This property will later come in handy, when we reason about information distribution between vertices over a span of rounds.

To further motivate the concept of a root component, we can ask ourselves what kind of properties we want from our communication graphs with regard to the consensus problem? One meaningful answer would be that we want the consensus problem to still be solvable, but we do not want to impose strong guarantees like "strong connectivity" to ensure solvability. Too strong guarantees are problematic, because these guarantees are not easily matched by real-world conditions. Other guarantees like "weak connectivity" are too weak to make consensus solvable in all instances. The concept of a rooted graph lies somewhere between strong and weak connectivity and allows us to still solve the consensus problem – as demonstrated by the algorithms presented in Chapter 3.

Extending the concept of a root component to span a sequence of graphs leads to the definition of a vertex-stable root component.

Definition 2.3.2 (Vertex-stable root component)**.** *If all graphs in a non-empty sequence of graphs have the same root component R, we call the sequence R-rooted and the root component R the vertex-stable root component of the sequence.*

Edges between members of the root component and other vertices are allowed to change, as long as the set of vertices in the root component stays the same and all vertices are reachable. See Figure 2.2 for a graphical explanation of the concepts – the vertices that are part of a root component are colored yellow.

Now that we have introduced the concept of an R-rooted sequence, we can start discussing properties of information propagation over time in a DDN.

2.3.2 Causal past

To introduce the concept of a causal past, we first need to define the compound graph and the in-neighbourhood of a vertex. The causal past will be our tool of choice to describe how information propagates through the DDN over time.

Definition 2.3.3 (Compound graph)**.** *Let $G = (V, E)$ and $G' = (V, E')$ be directed graphs. The graph $G \circ G' = (V, E'')$ with $E'' = \{(p, q) \mid \exists u \in V : (p, u) \in E \wedge (u, q) \in E'\}$ is called the compound graph.*

The compound graph captures the notion of having a direct or indirect connection over time. For an edge to exist in the compound graph, there must be a node u that is able to receive information in the first graph G and is then able to pass on the information in the second graph G'. We assume that communication graphs have self-loops, because these reflexive edges nicely capture the ability of processes to store information and pass it on at a later point in time. Note that $G \subseteq G \circ G'$ and $G' \subseteq G \circ G'$ holds.

Definition 2.3.4 (In-neighbourhood)**.** *Let $G = (V, E)$ be a communication graph. We call the set $\{p \mid (p, q) \in E\}$ the in-neighbourhood of process q and denote it with $\mathsf{In}_q(G)$.*

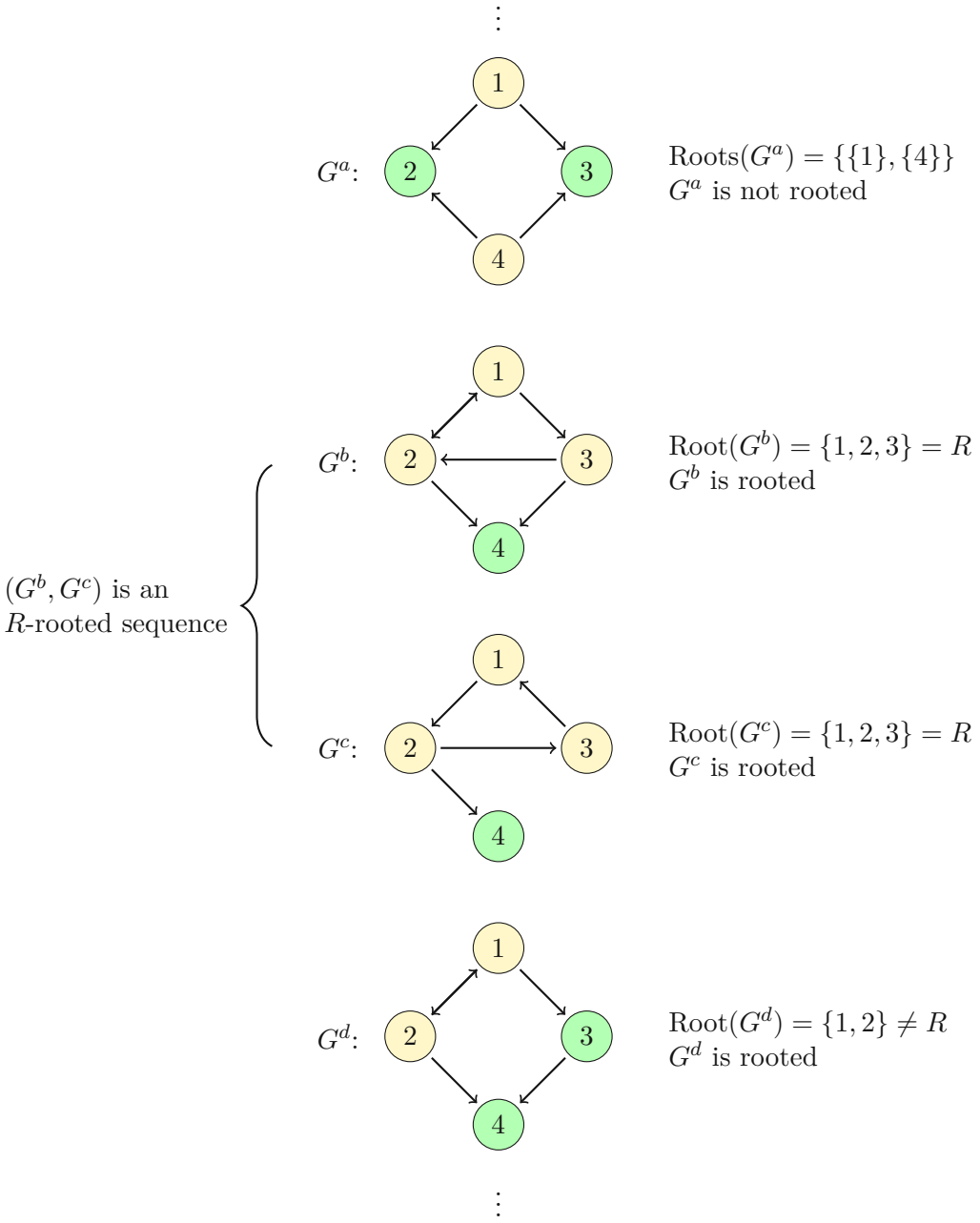


Figure 2.2: Illustration of the graph theoretical concepts *root component* and *R-rootedness*. Vertices that are part of a root component are coloured yellow.

The in-neighbourhood of a process p is the set of all processes that p is able to receive messages from in a round.

Definition 2.3.5 (Causal past)**.** *The causal past of a process p starting at round b and going back to round a is defined as $\mathsf{CP}_p^b(a) = \mathsf{In}_p(G^{a+1} \circ \ldots \circ G^b)$ for $b > a$, otherwise $\mathsf{CP}_p^b(a) = \emptyset$.*

The causal past of a process p from time b down to time a is a set of vertices. All vertices in this set were able to influence the state of p at the end of round b either directly or indirectly with state information no older than from the end of round a. Note that the definition of the causal past builds the compound graph starting with G^{a+1} – this is due to the fact that the round a state is first broadcast in round $a+1$.

If we look at the causal past in the context of a full information protocol – a mode of communication, where all processes broadcast their entire state history, including all the received messages – we can express the meaning of the causal past in a different way. For a process $q \in \mathsf{CP}_p^b(a)$, process p knows the message that q sent in round $a+1$. Knowing the message additionally implies that the causal past $\mathsf{CP}_p^b(a)$ is the set of vertices where process p at time b knows the process' state at round a. Figure 2.3 illustrates the concept of the causal past for the exemplary graph sequence (G^a, G^b, G^c, G^d).

2.3.3 Dynamic Network Depth

Now that we have the tools to model information propagation, we are going to introduce a parameter that captures the speed (in terms of rounds) of information distribution in a DDN in combination with a root component. The dynamic network depth D measures how many rounds it takes, at most, for information from every member of the root component to reach all vertices in the graph. The upper bound for D of $D \leq N-1$ was proven in [BRS+18]. The dynamic network depth is not a static graph concept, but one that makes only sense for R-rooted subsequences of graphs.

Expressing the dynamic network depth in terms of the causal past leads us to the insight that for information from the vertex stable root component R to have reached all nodes, R must be a subset of the causal past of all processes.

Definition 2.3.6 (Dynamic Network Depth)**.** *The dynamic network depth D at round a for a graph sequence that is R-rooted in $(G^r)_{r=a}^{a+D-1}$, is the smallest value of D for which $R \subseteq \mathsf{CP}_p^{a+D-1}(a-1)$ for all processes p holds.*

We will further motivate the concept of the dynamic network depth by giving an example, that also shows that D is not directly related to the maximal path length in the graphs – see Figure 2.4.

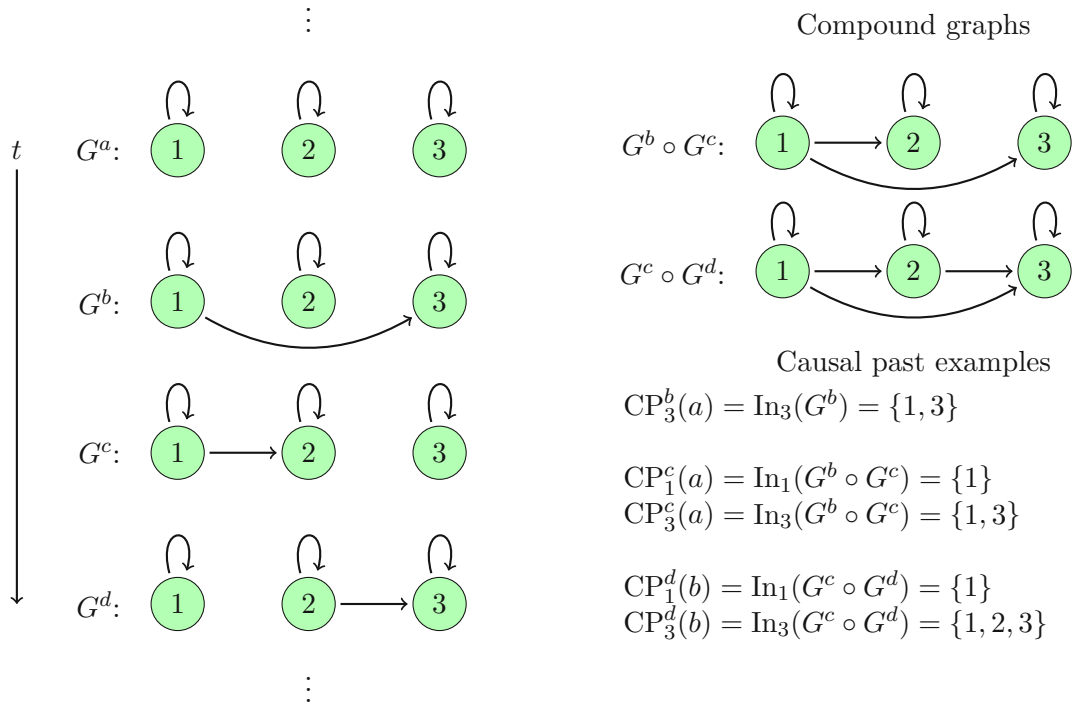


Figure 2.3: Various examples of the causal past operation applied to different nodes from graphs of the sequence (G^a, G^b, G^c, G^d) starting and ending in different rounds. The compound graphs required to calculate the causal past are shown as well.

2.4 Message adversaries

Now all the definitions we need are in place and we can provide the specification for the message adversaries behind the algorithms in Chapter 3. We are going to specify the message adversary $\Diamond\mathsf{STABLE}_{\leq N,D}(D+1)$, used by the terminating consensus algorithm presented in Section 3.3 and the message adversary $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ used by the stabilizing consensus algorithm presented in Section 3.2. The definitions for the message adversaries closely follow the definitions given in [WSS19] and [SS21].

We are going to define three sets of infinite communication graph sequences, where each set describes one important property that we expect from our message adversary. Informally, we want the message adversary to be strong, making consensus not easily solvable, but not so strong as to prevent reaching consensus. As briefly mentioned above, the concept of a root component extends the idea of a weakly connected graph with additional properties that, as Chapter 3 shows, make consensus solvable. Only weakly connected communication graphs are not enough for consensus to be solvable. The fact that certain information can be distributed to all vertices during an R-rooted subsequence of sufficient length, is the main property that the algorithms are going to exploit. It should come as no surprise, that the existence of said R-rooted subsequence is going

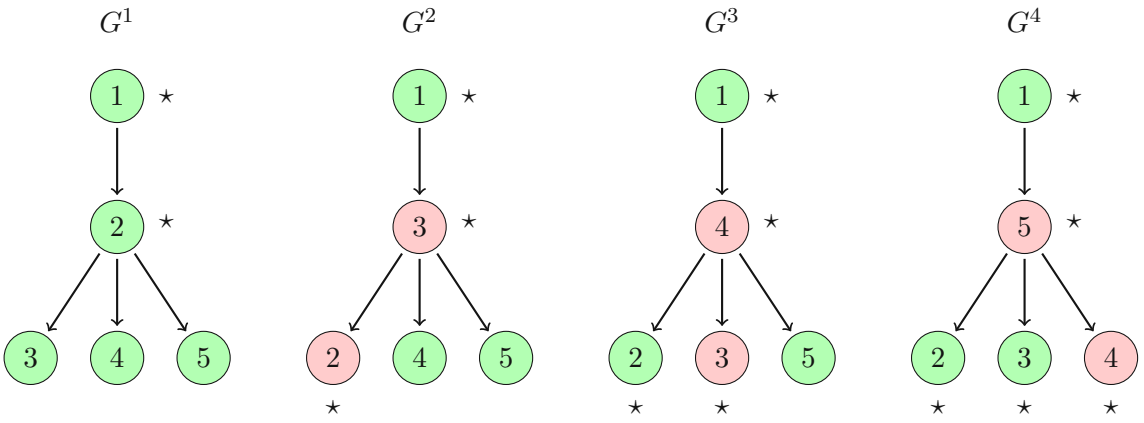


Figure 2.4: The asterisk represents the information that node 1 – the vertex-stable root component of the graph sequence (G^1, G^2, G^3, G^4) – is trying to spread throughout the network. Highlighted nodes switched position between the current and the last graph. Although all paths are of length 2, the dynamic network depth D is 4.

to be an important part of the specification of our message adversaries. Requiring the eventual existence of an R-rooted subsequence can be thought of as a liveness property, in the sense that something good – an R-rooted subsequence in our case – will eventually happen.

Definition 2.4.1 ($\Diamond\mathsf{GOOD}_n(x)$)**.** *Let $r_a = \max\{a_p \mid p \in \Pi\}$ be the earliest round where all processes have become active. The set $\Diamond\mathsf{GOOD}_n(x)$ is the set of all infinite communication graph sequences σ with $|\Pi_\sigma| = n$, that contain an R-rooted subsequence $\sigma' = (G^{r'}, G^{r'+1}, \ldots) \subseteq \sigma$ with $|\sigma'| \geq x$ and $R \subseteq \Pi_{\sigma'}$ and $r' \geq r_a$.*

Under terminating consensus, decisions are irrevocable. Having arbitrarily long sequences of non-rooted communication graphs before the R-rooted subsequence occurs is problematic, because some processes might not receive information for an arbitrarily long time, thereby making terminating consensus impossible. We define an additional set of sequences of communication graphs, where all communication graphs are rooted. Requiring all communication graphs to be rooted can be thought of as a safety property, in the sense that nothing bad – a non-rooted graph in our case – ever happens.

Definition 2.4.2 (ROOTED_n)**.** *The set ROOTED_n is the set of all infinite communication graph sequences σ, where every graph in the sequence is rooted and $|\Pi_\sigma| = n$.*

The last thing we require from our message adversary is that it adheres to a given dynamic network depth. This would not be necessary, if we were content with the upper limit for D, namely $n - 1$. Including this specification in our message adversary allows us to specify smaller upper limits for the dynamic network depth.

Definition 2.4.3 ($\mathsf{DEPTH}_n(D)$)**.** *The set* $\mathsf{DEPTH}_n(D)$ *is the set of all infinite communication graph sequences* σ *with* $|\Pi_\sigma| = n$ *and for all rounds* r *and all subsequences* $\sigma' \subseteq \sigma$*, starting at round* r *with* $|\sigma'| = D$*, we require that if* σ' *is* R*-rooted, then* $R \subseteq \mathsf{CP}_p^{r+D-1}(r-1)$ *for all processes* $p \in \Pi_\sigma$*.*

The set $\mathsf{DEPTH}_n(D)$ enforces that, for all R-rooted subsequences σ' of at least length D, the given parameter D is the dynamic network depth.

Now we can combine the three sets to form the message adversary used by the terminating consensus algorithm:

Definition 2.4.4 ($\Diamond\mathsf{STABLE}_{n,D}(x)$)**.** *The set*

$$\Diamond\mathsf{STABLE}_{n,D}(x) = \mathsf{ROOTED}_n \cap \Diamond\mathsf{GOOD}_n(x) \cap \mathsf{DEPTH}_n(D)$$

is called the eventually stabilizing message adversary with stability period x*.*

We are going to work with an instance of the above message adversary called

$$\Diamond\mathsf{STABLE}_{\leq N,D}(D+1) = \bigcup_{n=2}^{N} \Diamond\mathsf{STABLE}_{n,D}(D+1)$$

where the number of actually participating processes n is $2 \leq n \leq N$. See [WSS19, Theorem 3] for proof why the stability period needs to be at least $D+1$ for a consensus algorithm to exist.

For stabilizing consensus, the message adversary can do without some restrictions. Since processes are allowed to decide multiple times, we do not require that all communication graphs are rooted. We just need the existence of a sufficiently long R-rooted sequence. Note that the definition of $\mathsf{GOOD}_n(x)$ enforces that the R-rooted sequence only occurs after all processes have become active.

Definition 2.4.5 ($\Diamond\mathsf{WEAKSTAB}_n(x, D)$)**.** *The set*

$$\Diamond\mathsf{WEAKSTAB}_n(x, D) = \Diamond\mathsf{GOOD}_n(x) \cap \mathsf{DEPTH}_n(D)$$

is called the weakly stabilizing message adversary.

Again we are going to work with an instance of the above message adversary called

$$\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D) = \bigcup_{n=2}^{N} \Diamond\mathsf{WEAKSTAB}_n(D+1, D)$$

for reasons already mentioned before.

CHAPTER 3

Root component based algorithms

> *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*
>
> — Leslie Lamport *(1987)*[1]

This chapter is concerned with root component based consensus algorithms. The first algorithm we are going to present solves the stabilizing consensus problem under the message adversary $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$. By extending the stabilizing consensus algorithm with functionality to guarantee the termination property mandated by the terminating consensus problem, we will arrive at an algorithm that solves the terminating consensus problem under the message adversary $\Diamond\mathsf{STABLE}_{\leq N,D}(D+1)$.

Both algorithms were first introduced in [SS21]. The terminating consensus algorithm from [SS21] is the round-oblivious improvement of an algorithm first presented in [WSS19].

3.1 Basic ideas

The stabilizing and the terminating consensus algorithm are based on the same ideas. We are going to briefly introduce each idea and refine the discussion when presenting the algorithms in detail.

Full information protocol The algorithms use a full information protocol. A full information protocol mandates that each process broadcasts a copy of its entire state in every round. In our case, the entire state of a process is captured in the variables P, S and A. Note that, because of the way P, S and A are updated, the state of a process includes all the state information it previously received from other processes as well.

[1] https://lamport.azurewebsites.net/pubs/distributed-system.txt

Round obliviousness Instead of using the round counter r to read and write historical state, both algorithms employ an indirect approach. The algorithms refer to the current round r as level 0, the round $r-1$ as level -1, et cetera. All access to the state is done via levels, the algorithms have no need for the round counter r – making them round oblivious.

Under-approximation of graph sequence Using the received messages, each process, over time, builds and refines an under-approximation A of the current graph sequence. More specifically, a process p adds an edge (q, p) to its round r graph approximation A_p for each message received from a process q in round r. Since we operate under a full information protocol, every received message also contains the graph approximation of the sending process. Consequently, each process p merges the received graph approximations into its own. The resulting A_p^r is an under-approximation of the actual round r communication graph G^r, in the sense that only edges that really existed in G^r are present in A_p^r, but not necessarily vice versa.

Detection of vertex stable root component Since both message adversaries guarantee the eventual existence of an R-rooted sequence of graphs of at least length $D+1$, we know, that at some point in time, all processes will have received information from all members of said root component R. This implies that all nodes are able to detect the root component R in their graph approximation A when looking back D rounds in their state history. The nodes can now derive a common decision value using a deterministic function that maps all the proposal values of the processes that make up the root component R to a single value.

Persistence of decision After the R-rooted sequence has occurred, all nodes will have computed the same value as their new proposal value. This effectively prevents vertex stable root components, which occur after the first sufficiently long vertex stable root component, from being able to change the proposal value: new proposal values would be derived from the "old" proposal values, which are now all equal.

Termination The terminating consensus algorithm needs to solve an additional problem – that of termination and deciding only once. The algorithm introduces an additional lock variable and some functions to allow processes to check if the current proposal value is in agreement with that of the other processes. Only after the algorithm has made sure that all processes are in agreement can it safely decide.

3.1.1 Data structures

Since the terminating and the stabilizing consensus algorithm share the same basic ideas, they also operate on the same basic data structures. We use the same data structures for both the terminating and the stabilizing consensus algorithm, even if the stabilizing algorithm does not make use of the lock variable ℓ. This simplifies presentation,

because many helper functions can be shared between the stabilizing and the terminating consensus algorithm.

The variable A is the graph approximation built over time. A is a set of tuples of the form (s, q, p), where s is a level and q and p are processes – in other words: A is a set of edges (q, p) marked with a level s. P is the set of processes the executing process knows about and S is a record of the history of proposal and lock values for all processes. S contains tuples of the form (p, s, x, l), where p is a process, s a level, x a proposal value and l a lock value. Note that only the terminating consensus algorithm actually uses the lock value – the stabilizing consensus algorithm never writes ℓ besides initializing it.

All processes p start out by setting their proposal value x to some initial value $x_p \in V$ with $\perp \notin V$. For stabilizing consensus the proposal value x is also the output value of the algorithm. Processes executing the terminating consensus algorithm have an additional output value called $y_p \in \{\perp\} \cup V$ – this variable can only be written once and is assumed to be initialized to $\perp$.

3.2 Stabilizing consensus

The algorithm that solves stabilizing consensus under message adversary $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ is given in Algorithm 3.2. Algorithm 3.1 lists helper functions used by Algorithm 3.2, and the terminating consensus algorithm, Algorithm 3.4. We are going to briefly discuss the helper functions and then provide a description of the stabilizing consensus algorithm.

Function `x` from Algorithm 3.1 (Line 1) extracts the proposal value of process q at level s from S or returns $\perp$ if no entry exists.

Function `update` from Algorithm 3.1 (Line 6) updates the variables P, S and A with information provided by process q in the form of P_q, S_q and A_q. Because `update` is called after a message has been received from process q in round r, the edge (q, p) can be safely added to A, since after receiving a message from process q in round r, we know that the edge (p, q) must have been present in G^r. Note that the level of the newly added edge is set to 1 instead of 0, because `update2` will later add -1 to every level, leading to the correct level 0.

Function `update2` from Algorithm 3.1 (Line 10) makes using levels instead of absolute round numbers work, by pushing all entries back one level every time it is called. This allows the algorithm to always refer to the current round r as level 0 and to reference past rounds using negative levels.

Function `searchRoot` from Algorithm 3.1 (Line 12) checks if a root component exists at level s in the graph approximation A. To do so, it first extracts the level s communication graph from the graph approximation A and then proceeds to check if some strongly connected component of said graph is a root component. The first root component that can be found is returned. In case the extracted graph is the empty graph, no root component can exist and the empty set is returned.

Algorithm 3.1: Helper functions used by Algorithm 3.2 and Algorithm 3.4

```
1  Function X(q, s):
2      if ∃(q, s, x, ∗) ∈ S then
3          return x
4      else
5          return ⊥

6  Function update(q, P_q, S_q, A_q):
7      P ← P ∪ P_q ∪ {q}
8      S ← S ∪ S_q
9      A ← A ∪ A_q ∪ {(1, q, p)}

10 Function update2():
11     add −1 to s in every tuple (s, i, j) ∈ A and to s and l ≠ ⊥ in every tuple
       (p, s, x, l) ∈ S

12 Function searchRoot(s):
13     V ← {v ∈ P | ∃(s, ∗, v) ∈ A or ∃(s, v, ∗) ∈ A}
14     E ← {(u, v) ∈ P² | ∃(s, u, v) ∈ A}
15     Let SCC(V, E) denote the set of vertex sets of strongly-connected components
       of the graph (V, E)
16     foreach C ∈ SCC(V, E) do
17         if ∄v ∈ V \ C: (v,u) ∈E for some u ∈C then
18             return C
19     return ∅
```

3.2.1 Main algorithm

Algorithm 3.2 is the stabilizing consensus algorithm, first presented in [SS21]. Its implementation is discussed in Chapter 5 and experimental results regarding a comparison with the MinMax algorithm from [CM21] under real-world conditions are presented in Chapter 6. The algorithm makes use of the helper functions presented in Algorithm 3.1.

Every process p executes the same algorithm. After initialization in Lines 1 to 2, Lines 3 to 13 are executed for every round r. In the round r communication phase of the algorithm (Lines 3 to 4) all processes p broadcast their entire state – captured in the variables P, S and A – and receive messages of the form $(\mathsf{P}_q, \mathsf{S}_q, \mathsf{A}_q)$ from all processes q where $(q, p) \in G^r$.

After all messages have been received, the computation phase of the algorithm (Lines 5 to 13) starts. Lines 5 to 7 merge the information from all received messages into the local state and update the level markers for all entries. Lines 8 to 9 try to detect if a possible root component at level $-D$ is different from the root component of the previous level $-D-1$. If such a new root component can be found, the maximum of the

Algorithm 3.2: Stabilizing consensus under $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ for process p

Initialization:
1 $\mathsf{x} \leftarrow x_p, \ell \leftarrow \perp, \mathsf{A} \leftarrow \emptyset, \mathsf{P} \leftarrow \{p\}$
2 $\mathsf{S} \leftarrow \{(p, 0, \mathsf{x}, \ell)\}$

Round r communication, $r \geq 1$:
3 Broadcast $(\mathsf{P}, \mathsf{S}, \mathsf{A})$
4 $\forall q : (q, p) \in G^r$ receive message $(\mathsf{P}_q, \mathsf{S}_q, \mathsf{A}_q)$

Round r computation, $r \geq 1$:
5 **foreach** $(\mathsf{P}_q, \mathsf{S}_q, \mathsf{A}_q)$ *received in round r from q* **do**
6 | `update`$(q, \mathsf{P}_q, \mathsf{S}_q, \mathsf{A}_q)$
7 `update2`()
8 $R \leftarrow$ `searchRoot`$(-D)$
9 **if** $R \neq \emptyset$ **and** $R \neq$ `searchRoot`$(-D-1)$ **then**
10 | $t \leftarrow \max\{\mathtt{X}(q, -D) \mid q \in R\}$
11 | **if** $t \neq \perp$ **then**
12 | | $\mathsf{x} \leftarrow t$
13 $\mathsf{S} \leftarrow \mathsf{S} \cup \{(p, 0, \mathsf{x}, \ell)\}$

proposal values of all members of the root component at level $-D$ is computed in Line 10 and assigned to the current proposal value in Line 12. Keep in mind that the message adversary $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ does not force all communication graphs to be rooted. Should multiple root components exist in a round r communication graph, a random root component is returned. Nonetheless, the eventual existence of an R-rooted communication graph sequence of, at least, length $D+1$ makes the algorithm agree on a single value that cannot be changed afterwards. The computation part ends by updating the state S with current information in Line 13.

For a proof see [SS21, Theorem 6].

3.3 Terminating consensus

The algorithm that solves terminating consensus under message adversary $\Diamond\mathsf{STABLE}_{\leq N,D}(D+1)$ is given in Algorithm 3.4. The algorithm was first presented in [SS21] and its implementation is discussed in Chapter 5. The algorithm relies on the helper functions listed in Algorithm 3.1 and introduces new helper functions given in Algorithm 3.3, which we will now briefly discuss.

Function `L` from Algorithm 3.3 (Line 1) is very similar in style to function `X` from Algorithm 3.1 (Line 1). Instead of fetching the proposal value from the state history, `L` fetches the lock value for a given level s and process q.

Algorithm 3.3: Helper functions used by Algorithm 3.4

```
1  Function L(q, s):
2      if ∃(q, s, ∗, l) ∈ S then
3          return l
4      else
5          return ⊥

6  Function latestRefutation(a, b):
7      T ← {i ∈ [a, b] | ∃q ∈ P : L(q, i) = ⊥ or X(q, i) ∉ {⊥, x}}
8      if T ≠ ∅ then
9          return max(T)
10     else
11         return ⊥

12 Function uniqueCandidate(a, b):
13     if ∃k ∈ ℕ : ∀u ∈ P, ∀i ∈ [a, b] : L(u, i) ≠ ⊥ ⟹ X(u, i) = k and
        ∃q ∈ P, ∃j ∈ [a, b] : L(q, j) ≠ ⊥ then
14         return k
15     else
16         return −1

17 Function allGood(a, b, x):
18     return (∀q ∈ P, ∀i ∈ [a, b] : L(q, i) ≠ ⊥ and X(q, i) ∈ {⊥, x})
```

Function `latestRefutation` from Algorithm 3.3 (Line 6) returns the most recent level from the level interval $[a, b]$, where a process q has either not locked (ℓ set to $\perp$) or has locked and proposed a value different from process p's proposal value. The most recent level where this has happened is returned. In case no such level exists, `latestRefutation` returns $\perp$.

Function `uniqueCandidate` from Algorithm 3.3 (Line 12) checks if in the level interval $[a, b]$, there are processes that have locked and in addition, that all the processes that have locked, have proposed the same value. If such a common value can be found it is returned. Otherwise `uniqueCandidate` returns -1.

Function `allGood` from Algorithm 3.3 (Line 17) checks if it is true, that in the level interval $[a, b]$ all processes have locked and, if we know their proposal value, have proposed the same value as process p.

3.3.1 Main algorithm

To make the transition from stabilizing consensus to terminating consensus, the stabilizing consensus algorithm has to be augmented with the additional property of termination, because under terminating consensus, the output value y_p can only be written once. Up

to Line 8, Algorithm 3.4 and Algorithm 3.2 are identical. Both initialize their variables to the same values and handle broadcast and reception of messages in the same way.

Algorithm 3.4: Terminating consensus under $\Diamond\mathsf{STABLE}_{\leq N,D}(D+1)$ for process p

```
   Initialization:
1  x ← x_p, ℓ ← ⊥, A ← ∅, P ← {p}
2  S ← {(p, 0, x, ℓ)}

   Round r communication, r ≥ 1:
3  Broadcast (P, S, A)
4  ∀q : (q, p) ∈ G^r receive message (P_q, S_q, A_q)

   Round r computation, r ≥ 1:
5  foreach (P_q, S_q, A_q) received in round r from q do
6  |  update(q, P_q, S_q, A_q)
7  update2()
8  R ← searchRoot(−D)
9  if R ≠ ∅ and ( ℓ = ⊥ or R ≠ searchRoot(−D − 1) ) then
10 |  x ← max{x(q, −D) | q ∈ R}
11 |  ℓ ← 0
12 else if r > N then
13 |  t ← latestRefutation(−N,−1)
14 |  if t ≠ ⊥ and t ≥ ℓ then
15 |  |  ℓ ← ⊥
16 |  if uniqueCandidate(−N, −1) ≠ −1 then
17 |  |  x ← uniqueCandidate(−N, −1)
18 if r > N(D + 2N), y_p = ⊥, ℓ ≠ ⊥ and allGood(−N(D + 2N), −1, x)
     = True then
19 |  y_p ← x
20 S ← S ∪ {(p, 0, x, ℓ)}
21 if ℓ ≠ ⊥ then
22 |  ℓ ← ℓ − 1
```

Just as Algorithm 3.2, Algorithm 3.4 tries to detect changes in the root component at level $-D$. When a change in root components is detected, in addition to setting the proposal value to the maximum over all proposal values from the root component, Algorithm 3.4 also sets ℓ to the current level 0 in Line 11 – signalling that it has locked.

If Line 9 of Algorithm 3.4 does not evaluate to true, the algorithm has to make sure, that the output value is written only once, and that all the output values are in agreement. When we reach Line 13 of Algorithm 3.4 we know that process p has locked and that no change in the root component was detected. In this case, the terminating consensus algorithm checks if the proposal value of process p is in agreement with the proposal

values of the other processes in the system and tries to align p's proposal value with the rest of the proposal values in Lines 13 to 17. This is done, because some processes might have already detected some other root component and set their proposal value accordingly. Line 14 of Algorithm 3.4 checks if there is a valid refutation of process p's lock value ℓ. A refutation is a level, where another process has either not locked at all or has locked, but set the proposal value to a different value. If the refutation has happened after or in the same level as process p's lock, then p clears its lock status in Line 15. This will make p perform the proposal value update in Line 10 in the next round, even if no change in the root component can be detected.

After checking for a refutation, the algorithm also checks if it can find a unique candidate for p's proposal value in the state history in Line 16 of Algorithm 3.4. If such a unique candidate can be found, the algorithm updates its proposal value to the unique candidate in Line 17, regardless of the fact that it may not have found a refutation. Note that when using the unique candidate the lock variable ℓ is not set, but it might very well be already set.

At the end of each round r computation, the algorithm checks if it can safely decide in Line 18 of Algorithm 3.4. If it is safe to decide, the algorithm decides in Line 19. Last, but not least Lines 20 to 22 update the state S with current information and decrease the lock value to make the level convention work.

For a proof see [SS21, Theorem 1]. This theorem establishes a one-to-one mapping of layers to round numbers – the algorithm using round numbers was already proven in [WSS19, Theorem 4].

CHAPTER 4

MinMax algorithms

[CM21] introduced a new category of algorithms called "MinMax algorithms", able to solve stabilizing consensus in time varying graphs. This chapter gives a short introduction to the system model and dynamic network concepts used by [CM21] and provides a way to translate some of the core ideas to concepts presented in Chapter 2. We will then briefly show that the assumed message adversary is not comparable to the message adversaries introduced in Section 2.4. An implementation of the MinMax consensus algorithm is presented in Chapter 5 and experimental results regarding a comparison with the stabilizing consensus algorithm from [SS21] under real-world conditions are presented in Chapter 6.

4.1 System model

The definitions and concepts presented here closely follow the definitions and concepts introduced in [CM21]. The system model from [CM21] assumes synchronous, round-based operation, fault free processes and a time varying-topology – the same assumptions are made by the system model presented in Chapter 2. The first big difference, is that processes – or agents, as the participants are called – are anonymous, in the sense that they do not have unique IDs.

4.1.1 Time-varying topology

The digraph that represents the round t communication graph is denoted by $\mathbb{G}(t) = (V, E_t)$. This is equivalent to a graph G^t in the notation of the system model from Chapter 2. As is the case in our system model, all communication graphs are assumed to be reflexive.

The sequence of communication graphs is denoted by $\mathbb{G} = (\mathbb{G}(t))_{t\in\mathbb{N}}$ and is called a dynamic graph. Non-empty sets of dynamic graphs are called network models. Similar to our system model, agents start out passive and can once make the transition to

active. $x_u(t)$ is used to refer to variable x belonging to agent u at the end of round t. The in-neighbourhood is denoted by $\text{In}_u(t)$ – the incoming neighbors of u in the communication graph $\mathbb{G}(t)$.

4.1.2 Dynamic graph concepts

Instead of calling $G \circ G'$ the compound graph of the graphs $G = (V, E)$ and $G' = (V, E')$, [CM21] calls the operation the graph product of G and G' and for $t' > t \geq 1$ introduces the notation $\mathbb{G}(t : t') = \mathbb{G}(t) \circ \ldots \circ \mathbb{G}(t')$ for repeated application of the product operation. This notational convention is extended to $\text{In}_u(t : t') = \text{In}_u(\mathbb{G}(t : t'))$ – the in-neighbourhood of a product of consecutive graphs.

This is a concept very similar in nature to the Causal Past – see Definition 2.3.5.

$$\mathsf{CP}_p^b(a) = \text{In}_u(a + 1 : b)$$

The notation $\mathbb{G}(t : \infty)$ is defined to mean

$$\mathbb{G}(t : \infty) = (V, \cup_{t' \geq t} E(\mathbb{G}(t : t')))$$

and this extension of the interval to infinity can also be applied to the in-neighbourhood. $\text{In}_u(t : \infty)$ is defined to mean

$$\text{In}_u(t : \infty) = \cup_{t' \geq t} \text{In}_u(t : t')$$

.

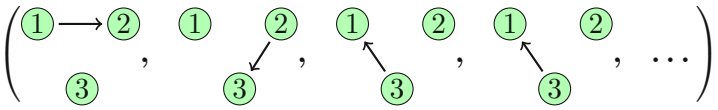
Note that $E(\mathbb{G}(t : t + 1)) \subseteq E(\mathbb{G}(t : t + 2)) \subseteq E(\mathbb{G}(t : t + 3)) \ldots$ holds. The extension of the interval to infinity is just the repeated application of the graph product operation.

Now we can introduce the first major dynamic network concepts – the integral and the limit superior of a dynamic graph.

Definition 4.1.1 (Integral of a dynamic graph)**.** *Let $\mathbb{G}$ be a dynamic graph. Then $\overline{\mathbb{G}}$ is called the integral of the dynamic graph $\mathbb{G}$ and defined by $\overline{\mathbb{G}} = (\overline{\mathbb{G}}(t))_{t \in \mathbb{N}}$ and $\overline{\mathbb{G}}(t) = \mathbb{G}(t : \infty)$.*

Each graph $\overline{\mathbb{G}}(t)$ in the dynamic network $\overline{\mathbb{G}}$, is a graph that arises from the product operation with an interval extended to infinity.

If a dynamic network $\mathbb{H}$ is defined as

$\mathbb{H} =$ 

with the last graph repeated forever, then the integral $\overline{\mathbb{H}}$ of the dynamic graph $\mathbb{H}$ is

$$\overline{\mathbb{H}} = (\mathbb{H}(1:\infty), \mathbb{H}(2:\infty), \mathbb{H}(3:\infty), \mathbb{H}(4:\infty), \ldots)$$

which comes down to

$$\overline{\mathbb{H}} = \Bigg(\begin{array}{ccccccc} 1 & \longrightarrow & 2 & , & 1 & \longleftarrow & 2 \\ & \searrow\swarrow & & & & \nwarrow\swarrow & \end{array} \quad , \quad \begin{array}{ccc} 1 & & 2 \\ & \nwarrow & \\ & 3 & \end{array} \quad , \quad \begin{array}{ccc} 1 & & 2 \\ & \nwarrow & \\ & 3 & \end{array} \quad , \quad \cdots \Bigg)$$

with the last graph repeated forever.

Definition 4.1.2 (Limit superior)**.** *Let $\mathbb{G}$ be a dynamic graph. Then $\mathbb{G}(\infty) = (V, E_\infty)$ is called the limit superior of graph $\mathbb{G}$. E_∞ is defined as*

$$E_\infty = \{(u, v) \in V \times V \mid \forall t, \exists t' \geq t : (u, v) \in E(\mathbb{G}(t'))\}$$

Note that the integral of a dynamic graph is itself a dynamic graph, but the limit superior of a dynamic graph is just a graph. Informally the limit superior is the graph containing all the edges that occur in an infinite number of graphs $\mathbb{G}(t)$ of the dynamic network $\mathbb{G}$.

Using our exemplary dynamic network $\mathbb{H}$, we see that both the limit superior $\mathbb{H}(\infty)$ and the limit superior of the integral $\overline{\mathbb{H}}(\infty)$ come down to

$$\mathbb{H}(\infty) = \overline{\mathbb{H}}(\infty) = \begin{array}{ccc} 1 & & 2 \\ & \nwarrow & \\ & 3 & \end{array}$$

.

Definition 4.1.3 (Root of a graph)**.** *Let $G = (V, E)$ be a digraph. A node $u \in V$ is called a root of the graph G, if there exists a path from u to every node $v \in V$. A node $u \in V$ is called a central root of the graph G, if there exists an edge $(u, v) \in E(G)$ for every $v \in V$.*

A graph with at least one root is called rooted and a dynamic graph $\mathbb{G}$ is called permanently rooted if all graphs $\mathbb{G}(t)$ are rooted. We write $\mathrm{Roots}(G)$ for the set of roots of G and $\mathrm{CRoots}(G)$ for the set of central roots of G.

Lemma 4.1.1. *If a graph G is rooted in the terminology of [CM21], then G is also a graph with exactly one root component according to Definition 2.3.1 and vice-versa.*

Proof. Let $G = (V, E)$ be a rooted graph in the terminology of [CM21]. Then there is a node $u \in V$, where a path exists to all other nodes $v \in V$. Now assume that graph G has more than one root component in the terminology of [SS21]. Because there are no paths between root components this leads to a contradiction and the conclusion that G has exactly one root component.

For the other direction we assume a graph $G = (V, E)$ that has one root component R. By the definition of a root component, we know that for every $u \in R$, there is a path to every other node $v \in V$. All the members of the root component R are roots of G in the terminology of [CM21]. □

Definition 4.1.4 (Kernel of a dynamic network)**.** *Let $\mathbb{G}$ be a dynamic network. Then* $\mathrm{Ker}(\mathbb{G})$ *is called the kernel of $\mathbb{G}$ and defined by*

$$\mathrm{Ker}(G) = \cap_{t \geq 1} \mathrm{CRoots}(\overline{\mathbb{G}}(t))$$

The kernel of a dynamic network $\mathbb{G}$ is the intersection of all the central roots for each of the graphs in the integral of $\mathbb{G}$. Informally, these are the central roots that occur infinitely often in the integral of $\mathbb{G}$.

Definition 4.1.5 (Rooted with delay T)**.** *A dynamic graph $\mathbb{G}$ is called rooted with delay T, if for every $t \in \mathbb{N}$ $\mathbb{G}(t : t + T - 1)$ is rooted. A dynamic graph G is called rooted with bounded delay if it is rooted with delay T for some fixed positive integer T.*

[CM21, Corollary 12] now states that every safe MinMax algorithm solves stabilizing consensus for all dynamic graphs that are rooted with a bounded delay.

Informally, it should be noted that the concept of bounded delay rootedness allows one to reason about information distribution in a graph sequence, similar to the concept of dynamic network depth in combination with a vertex stable root component.

Definition 4.1.6 ($\mathsf{BOUNDEDDELAY}_{\leq N}$)**.** *The message adversary $\mathsf{BOUNDEDDELAY}_{\leq N}$ is the set of a all dynamic networks with $|V| \leq N$, that are rooted with bounded delay.*

4.2 Comparison of message adversaries

To be able to compare the graph sequences generated by $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ with $\mathsf{BOUNDEDDELAY}_{\leq N}$, we must first translate $\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$ into a form that is not dependant on the point in time where processes get activated.

We introduce the message adversary $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ that, instead of generating the vertex stable root component of length $D+1$ only after all processes have become active, generates a vertex stable root component of length $D+1$ infinitely often. This does not influence the correctness of the algorithm. Let s be the smallest round where all processes have become active. All decisions before round s can still be overridden by the first vertex stable root component that starts at or after round s. After the first vertex stable root component at or after round s has occurred, the output value is fixed and cannot be changed by any vertex stable root component that occurs afterwards.

Note that the introduction of $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ also makes it clearer that there is no contradiction to [CM21, Theorem 6], which states the fact that there cannot be an algorithm that solves stabilizing consensus in dynamic graphs with an empty kernel. All

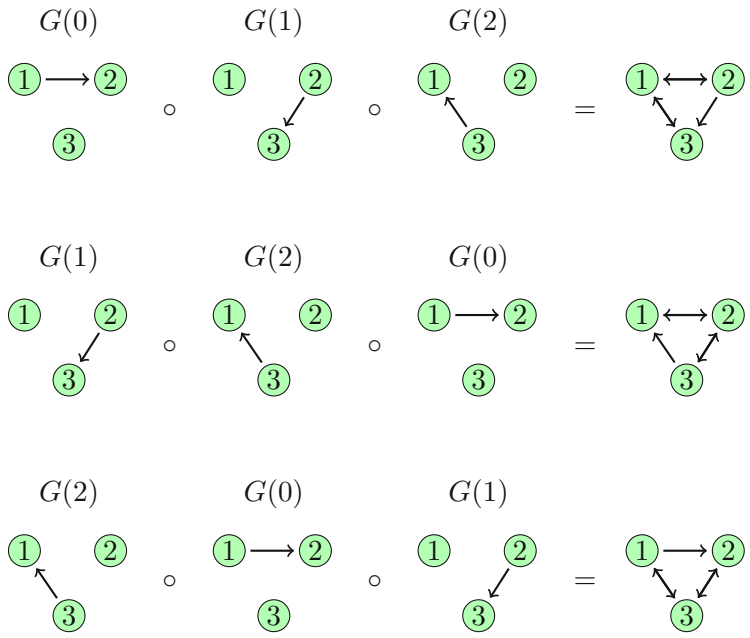
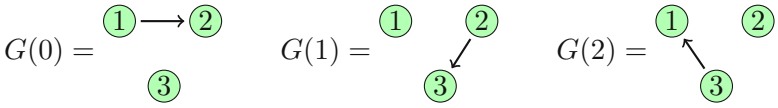


Figure 4.1: G is rooted with bounded delay

the dynamic networks G that $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ generates have $R \subseteq \mathrm{Ker}(G)$, with R being the vertex stable root component that occurs infinitely often.

We will now proceed to show that $\mathsf{BOUNDEDDELAY}_{\leq N}$ and $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ are not directly comparable. To do this, we will construct a dynamic network G that is an element of $\mathsf{BOUNDEDDELAY}_{\leq N}$, but not of $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ and a dynamic network H that is an element of $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ but not of $\mathsf{BOUNDEDDELAY}_{\leq N}$.

Construction of G We define the three graphs



and from those graphs build our dynamic network $G(t) = G(t \bmod 3)$.

None of the graphs in G are rooted. Therefore there does not exist a vertex stable root component of any length and G cannot be generated by $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$.

G is rooted with bounded delay $T = 3$ and is therefore a graph that can be generated by $\mathsf{BOUNDEDDELAY}_{\leq N}$. This can be checked by computing the graph products $G(0) \circ$

$G(1) \circ G(2)$, $G(1) \circ G(2) \circ G(0)$ and $G(2) \circ G(0) \circ G(1)$ and checking if all the results are all rooted. See Figure 4.1 for a visualization of the product operations.

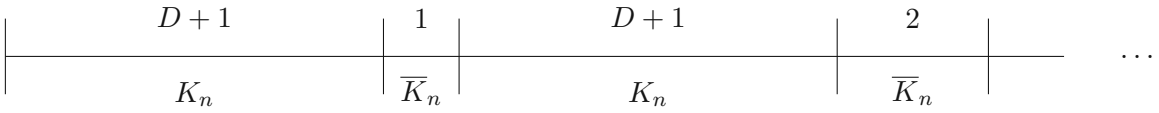


Figure 4.2: Construction of H

Construction of H To construct the dynamic network H with size n, we assume a graph sequence that starts with the complete graph K_n for $D+1$ rounds, followed by the empty graph $\overline{K}_n$ for a duration of 1 round. The empty graph is followed again by the complete graph for $D+1$ rounds, followed by the empty graph for a duration of 2 rounds. This scheme is repeated with increasing durations for the subsequences of empty graphs. The dynamic network H can be generated by $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$, but due to the ever increasing subsequences of empty graphs it is not rooted with bounded delay and can therefore not be generated by $\mathsf{BOUNDEDDELAY}_{\leq N}$. See Figure 4.2 regarding the construction of H.

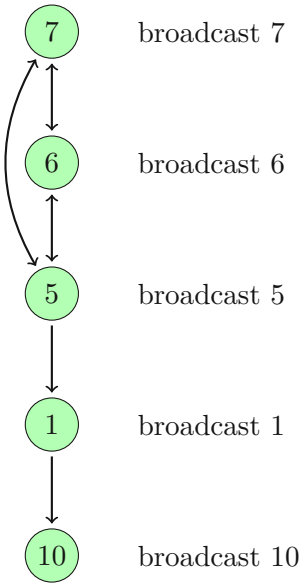
Theorem 4.2.1. $\mathsf{BOUNDEDDELAY}_{\leq N}$ *and* $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ *are not directly comparable.*

Proof. Because of the existence of G, $\mathsf{BOUNDEDDELAY}_{\leq N} \not\subseteq \Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D)$ and by the existence of H, $\Diamond\mathsf{WEAKSTAB}'_{\leq N}(D+1, D) \not\subseteq \mathsf{BOUNDEDDELAY}_{\leq N}$. □

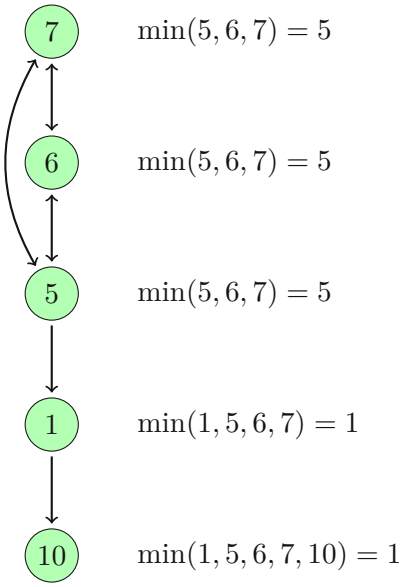
4.3 The algorithm

We are now going to present the main ideas behind the MinMax algorithm, with the purpose of giving an intuitive understanding of this class of algorithms. For more details see [CM21, Chapters 4, 5 and 6]. After discussing the basic ideas we, will present the algorithm optimized for implementation in pseudocode.

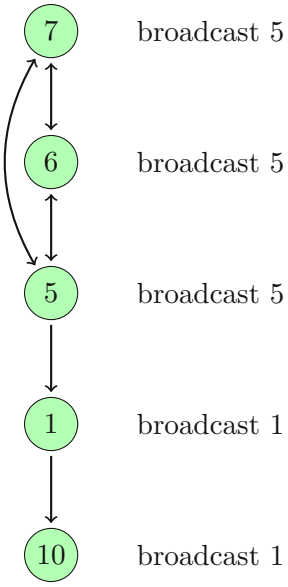
To get an intuitive understanding of why this algorithm works, we take a look at the requirements for the communication graphs. The message adversary $\mathsf{BOUNDEDDELAY}_{\leq N}$ requires that the dynamic network $\mathbb{G}$ is rooted with bounded delay. Having a dynamic graph that is rooted with bounded delay means that the in-neighbourhood of a node u of a compound graph, starting at any round r, of sufficient size must contain nodes from the kernel (every dynamic network that is rooted with bounded delay has a non-empty kernel by definition). In other words: there is an integer s such that for all compound graphs of at least size s every node u has a node from $\mathrm{Ker}(\mathbb{G})$ in its in-neighborhood. Informally, the kernel of a dynamic network is hence the set of nodes that are able to repeatedly reach all nodes in the network.



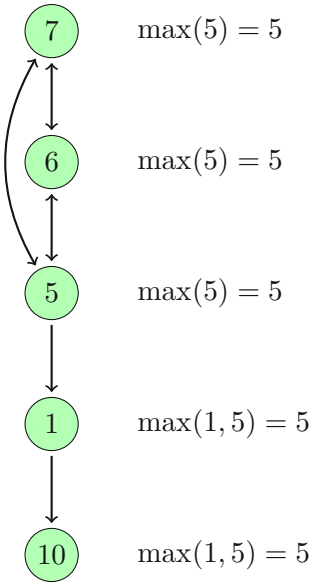
(a) Round 1 – all nodes start to distribute their initial value and repeatedly compute the minimum over all received values.



(b) Round θ – the computed minimum values have stabilized, because all values have been received.



(c) Round $\theta+1$ – the nodes start to distribute the computed minimum values and repeatedly compute the maximum over all received computed minimum values.



(d) Round $\theta+1+\Delta$ – the computed maximum values have stabilized to the same value, because all minimum values have been received.

Figure 4.3: Showing the min and max computation phase of the MinMax algorithm. With a correctly chosen θ the nodes reach consensus.

For the purpose of the presentation, we now assume the existence of a compound graph of sufficient size. We additionally assume a round θ according to the specification below. All nodes start by broadcasting their initial value and by computing the minimum of the received values. If this scheme were to be executed in a dynamic network $\mathbb{G}$ with nodes V and $\mathrm{Ker}(\mathbb{G}) = V$, then all nodes would eventually stabilize to the global minimum over all values. But since we do not require our kernel to equal V, we need additional steps to compute a correct value: Assuming that, when round θ is reached, all nodes within the kernel have stabilized to the same minimum value and all other nodes have stabilized to a possibly different value than the value derived by the kernel members. After round θ, all nodes switch to a different computation scheme: The nodes start broadcasting the computed minimal values in addition to computing the maximum value over the received minimum values. This makes all the nodes agree on a common value over time. Figure 4.3 shows an exemplary graph and gives a graphical explanation of the min and max computation steps.

The problem now is that the value θ is not know beforehand and cannot be precomputed. The paper shows that, by computing θ repeatedly on the fly with certain properties, this scheme can be made working. Loosely speaking, eventually, we need θ to be large enough that all the nodes can compute the minimum value in the first phase and we need it to be small enough that the nodes are able to receive the computed minimum value from members of the kernel and compute the shared maximum.

4.3.1 Practical implementation

Algorithm 4.1 shows the pseudocode for the algorithm. We adapted the form of the pseudocode to fit the presentation of the root component based algorithms.

Algorithm 4.1: Stabilizing consensus under $\mathsf{BOUNDEDDELAY}_{\leq N}$ for process p

Initialization:
1 $\mathsf{x} \leftarrow x_p$, $\mathsf{y} \leftarrow x_p$, $\delta \leftarrow 0$
2 $\mathsf{AGE}[1 \dots |V|] \leftarrow \infty$, $\mathsf{AGE}[\mathsf{x}] \leftarrow 0$

Round r communication, $r \geq 1$:
3 Broadcast AGE
4 $\forall q : (q, p) \in G^r$ receive message AGE_q

Round r computation, $r \geq 1$:
5 **foreach** $\alpha \in V$ **do**
6 $\quad \mathsf{AGE}[\alpha] \leftarrow 1 + \min\{\mathsf{AGE}_q[\alpha] : (q, p) \in G^r\}$
7 $\mathsf{x} \leftarrow \min\{\alpha : \mathsf{AGE}[\alpha] < \infty\}$
8 $\mathsf{AGE}[\mathsf{x}] \leftarrow 0$
9 $\delta \leftarrow \delta(r)$ `// cutoff value, which determines max computation`
10 $\mathsf{y} \leftarrow \max\{\alpha : \mathsf{AGE}[\alpha] \leq \delta\}$

To facilitate a practical implementation, the authors of [CM21] argue that instead of distributing the whole graph for each round and using those graphs to compute the required values, one only needs to broadcast the relevant values using the AGE data structure.

The basic structure of the algorithm is the same as for the root component based algorithms. Lines 1 to 2 initialize all the variables and data structures. The variables x and y are initialized to the starting value. The output value of the algorithm is y. The dictionary AGE contains a mapping of proposal values to counter values. The counter represents the age of a value – how many rounds ago this value was last seen.

Lines 3 to 10 form the body of the algorithm and are executed every round. The communication phase of the algorithm happens in Lines 3 to 4. During the communication phase of round r, the AGE data structure is broadcast and in turn received by all nodes in accordance with the round communication graph G^r.

The computation phase is described in Lines 5 to 10. The set V is the set of all possible consensus values. In Lines 5 to 6, the algorithm determines the minimum age (the latest point in time), where a value was seen by any of the nodes we received a message from. Note that, by definition, a node always receives its own message, so the AGE data structure of the executing process p is also taken into account.

Line 7 is the min step in the algorithm's execution. Each node determines the smallest value it has seen at this point in time, updates the variable x and adds it to the AGE data structure (Line 8). An age of ∞ indicates that the value was not seen yet. The next step is to calculate a cutoff age in Line 9. This cutoff age $\delta(r)$ somewhat corresponds to the θ value in the introductory presentation. The function to calculate δ can be chosen by the implementor, but needs to guarantee that the ranges where the min and the max operations are performed must be ever increasing for $r \to \infty$. A possible function that fulfills these requirements and takes the round counter r as the input is $f(r) = \lfloor \frac{r}{2} \rfloor$. After calculating the cutoff age, the algorithm performs the max step in Line 10. The maximum value of all values younger or equal to δ is taken and assigned to the output variable y.

For a proof of why the MinMax algorithm works in this form, see [CM21, Theorem 16].

CHAPTER 5

Implementation

We think in generalities, but we live in detail.
— Alfred North Whitehead, *"The Education of an Englishman" in The Atlantic Monthly, Vol. 138 (1926), p. 192*

5.1 Overview

The theoretical part of this thesis has – amongst other things – presented variants of the consensus problem and matching algorithms able to solve consensus in their respective system models. In particular, it has given a detailed introduction to a group of root component based algorithms, which solve consensus under a message adversary whose practical relevance is backed up by some experimental data, see [PS16]. The focus of this thesis will now shift towards providing an actual implementation of the aforementioned root component based algorithms and the MinMax algorithm. We will describe challenges faced, solutions found and questions still in need of answering. Our description of the system is going to be from the bottom up. Figure 5.1 provides a map of the involved components and their relationship to each other.

We are going to start by briefly describing the hardware, the operating system and the communication protocols used. We will motivate and explain our design choices by specifying concrete requirements and describing how the selected components meet those requirements.

After introducing the foundations of our implementation, we will focus on the task of providing nodes with a shared notion of time. This notion of global time is necessary to provide one of the pillars upon which the algorithms' system models rest – lock-step synchronous rounds. We will give a detailed description of the time synchronization

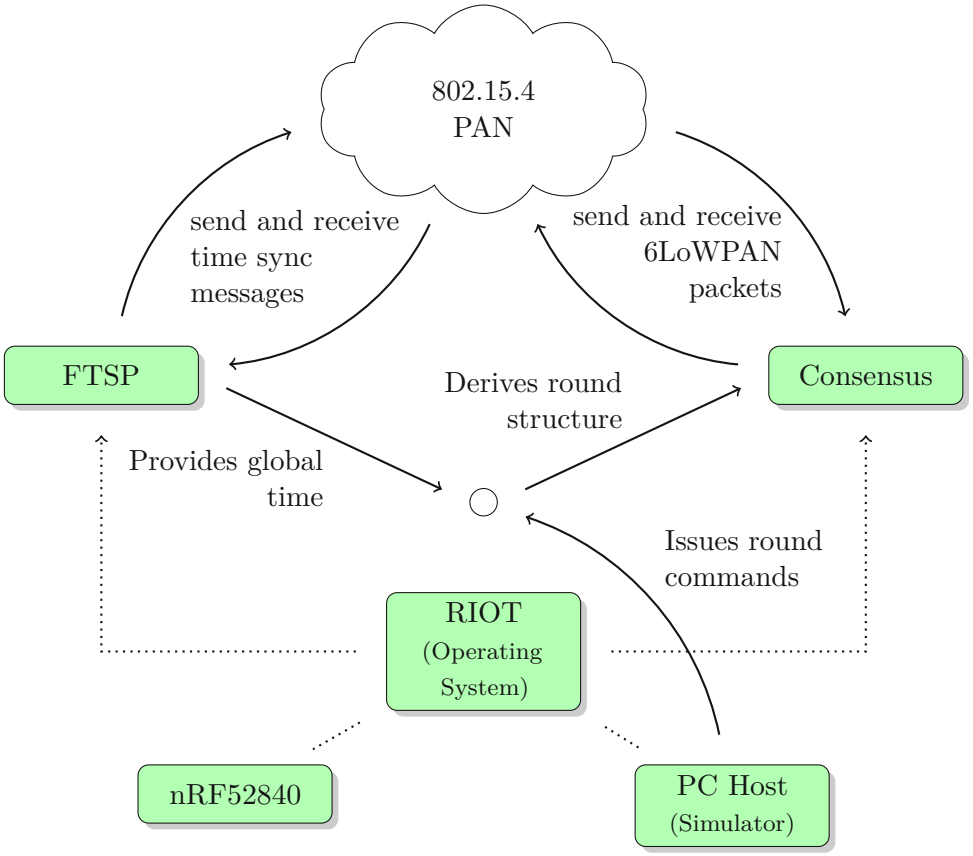


Figure 5.1: System overview and components of the implementation

approach used and describe how to derive lock-step synchronous rounds from a global time. In addition we are going to present some measurement results regarding the achieved synchronization precision.

After we have established a framework that is able to provide communication in lock-step synchronous rounds, we will give a detailed account of the algorithms' implementations and discuss relevant design decisions. A major asset during the implementation of the algorithms was the simultaneously developed simulation environment. Chapter 6 will present key features of the simulator.

5.2 Hardware, Operating System, Networking

The algorithms are described independently of hardware, Operating System (OS) and networking technology. But for an implementation, all these things have to be selected from a multitude of available options. The specific choices have a great deal of influence on the final result. The System-on-Chip (SoC) and its radio determine the physical properties of the communication protocols that can be used, the OS determines the

available abstractions and libraries and the networking stack constrains the available communication primitives. We identified the following constraints and demands that our specific choices needed to satisfy:

Open-source operating system and networking stack Because of the nature of the time synchronization protocol used, access to the source code of the networking stack – especially at the lower levels – and the possibility to easily adapt it, were a major decision point. Because of the number of different concurrent tasks (time synchronization, communication, algorithm computation) the nodes have to perform, support for threading or other concurrency primitives was also essential.

Standardized, low-complexity networking protocols Instead of developing custom solutions, we tried to rely as much as possible on existing standards. The need for a networking stack with relatively low complexity is again motivated by the fact that we most likely are going to have to adapt the stack to fit the needs of the time synchronization protocol. The algorithm's only requirement regarding communication is a means to broadcast information – due to the nature of wireless communication, almost all wireless protocols provide this communication primitive.

Ease of programming and debugging The development board needs to be relatively comfortable to work with in terms of programming and debugging. The ability to power the hardware over USB and a serial-over-USB interface are equally important.

In the end we chose RIOT as the OS and a development board designed around the nRF52840 SoC. The SoC has support for a multitude of communication protocols in the 2.4 GHz band – we chose IEEE 802.15.4 in combination with IPv6 over Low power Wireless Personal Area Network (6LoWPAN) as the basic networking technologies.

5.2.1 nRF52840

The development board used for the implementation is the nRF52840-MDK[1]. This development board features an nRF52840 SoC from Nordic Semiconductor and various peripheral components like a Debug-Access-Port Link (DAPLink) interface for easy programming and debugging. Figure 5.2 shows a picture of the development board with additional peripherals.

The nRF52840 contains a 32 bit ARM Cortex-M4 processor, clocked at a speed of 64 MHz. It is equipped with 1 MB of flash memory, 256 kB of RAM and includes a single precision Floating Point Unit (FPU). The development board provides an additional 8 MB of external flash memory.

The nRF52840 has radio support for Bluetooth Low Energy (BLE), IEEE 802.15.4-2006 and custom proprietary protocols like Gazell (an in-house development of Nordic

[1] https://wiki.makerdiary.com/nrf52840-mdk/

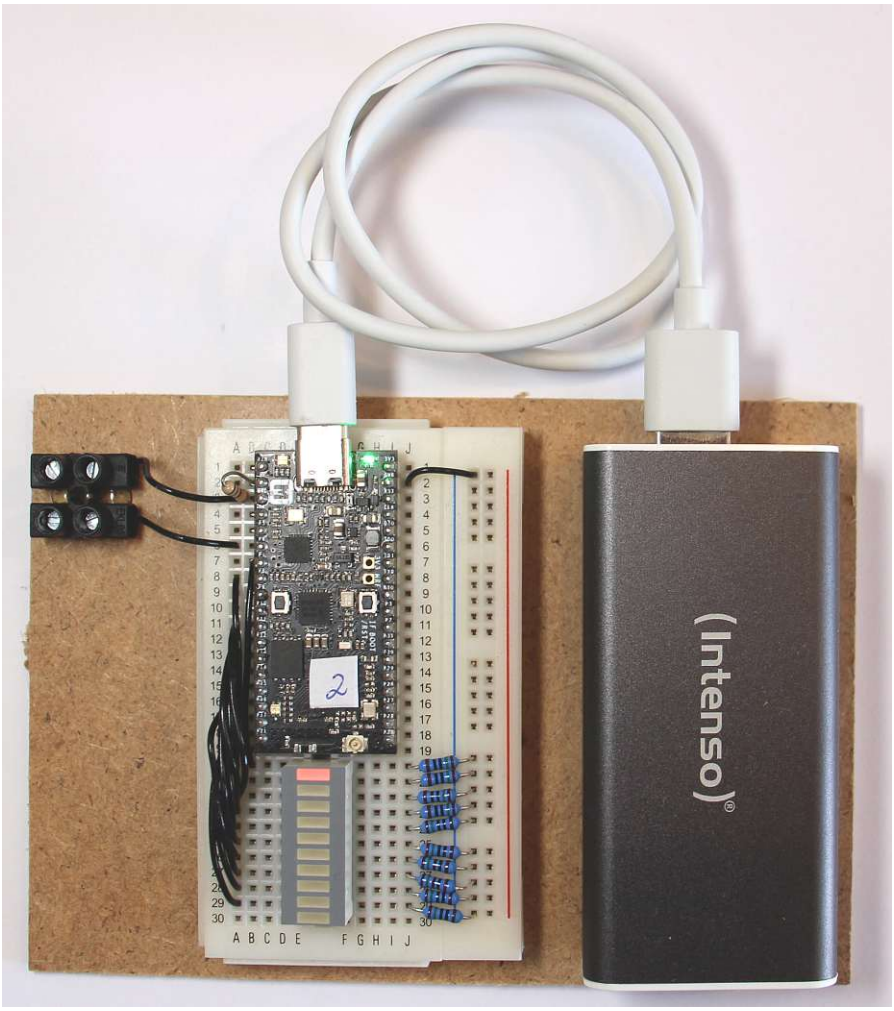


Figure 5.2: Photograph of the development board with additional peripherals

Semiconductor). All supported protocols operate in the 2.4 GHz frequency band. The nRF52840's radio is a typical "Soft-MAC" radio, in that it only implements parts of the MAC layer of IEEE 802.15.4 in hardware. In addition to a complete Physical (PHY) layer implementation, the radio only provides facilities for Cyclic Redundancy Check (CRC) generation and support for channel sensing and energy detection. Data transfer, from and to the radio, is handled via Direct Memory Access (DMA). The nRF52840 includes an EasyDMA module that is used to initiate data transfers to almost all memory mapped peripherals on the SoC. The fact that the radio does not provide a complete IEEE 802.15.4 MAC layer in hardware comes as an advantage, because the time synchronization protocol requires us to obtain and insert timestamps at certain points in the MAC layer. It does not seem very likely that a "Hard-MAC" radio would provide the support required to do this.

5.2.2 RIOT

The OS used in the implementation is RIOT[2]. RIOT is an operating system, next to TinyOS, Contiki-NG and others, targeting the embedded systems market, with a focus on IoT requirements such as low power consumption, low memory and low cost nodes with wireless communication capabilities. RIOT is implemented in C, provides support for threading and Inter-process-communication (IPC) and has a strong academic background. It provides its own networking stack called GeNeRiC network stack (GNRC), but also supports other networking stack implementations like lwIP[3], OpenThread[4] and emb6[5]. Our implementation makes use of GNRC and its 6LoWPAN module for communication in the consensus part. Bare IEEE 802.15.4 frames are used for communication in the time synchronization part. GNRC comes with a high-level POSIX-like socket API, which allows networking code to remain portable over various underlying networking technologies. RIOT has rich support for different SoCs and development boards and its build system has the ability to build executables capable of running under Linux. The native build, in combination with the high-level socket API, enables the developer to run all the code, including networking, on the development machine – our simulation environment makes use of this feature to some extent.

RIOT features a build system that makes it easy to provide functionality encapsulated in modules. The system designer is able to only include modules required for a specific functionality. This helps keep code and memory requirements low. RIOT comes with an extensive test suite and makes it relatively easy to reuse parts of the testing infrastructure for tests in custom modules. All functionality of our implementation is contained in reusable and loosely coupled modules. A comprehensive overview of RIOT can be found in [BGH+18].

5.2.3 Networking

As previously mentioned, IEEE 802.15.4 and 6LoWPAN are our basic networking technologies. Both protocols are well standardized and have good support in RIOT.

802.15.4

IEEE 802.15.4-2003 was released in response to a growing demand for an open, interoperable wireless communication protocol targeted especially at the low-power segment of sensor networks [Ada06].

One of the main reasons for choosing IEEE 802.15.4 over similar technologies like BLE is the relatively low complexity of the core elements of the standard. Since its initial release in 2003, the standard grew considerable in size and more features were added, but the core principles – an explicit focus on low-power networks and low complexity – stayed

[2] https://riot-os.org
[3] https://www.nongnu.org/lwip/2_1_x/index.html
[4] https://openthread.io/
[5] https://github.com/hso-esk/emb6

the same. The 2006 release [IEE06] of the standard – the one supported by our radio – includes four PHY specifications in two frequency bands (2.4 GHz and 868/915 MHz), whereas the current version of the standard – IEEE 802.15.4-2020 [IEE20] – includes around 20 different PHY specifications and additional methods for access arbitration and improved robustness like Time Slotted Channel Hopping (TSCH).

The standard specifies two distinct layers – a PHY and a MAC layer. Northbound of those two layers, the standard takes care not to make any requirements, allowing various higher level protocols like ZigBee or 6LoWPAN to be specified independently.

Even though the standard contains a section, where possible networking topologies are discussed [IEE06, p. 14], it explicitly does not make any assumptions regarding network layout. The formation of the specific topology is delegated to the upper layers. The two topologies mentioned in the standard are star and peer-to-peer topology. The star topology has a central node, which is responsible for passing on frames to the other participants in the network. The peer-to-peer topology, in contrast, allows nodes to communicate freely amongst themselves, without a central authority. The topology we use for our implementation is peer-to-peer – there is no single central authority in our network.

Devices wishing to communicate with each other form a Personal Area Network (PAN). Management functionality, such as PAN membership and other duties like regulating access to the shared medium in a beacon-enabled PAN, fall to a special node termed the PAN coordinator. The standard mandates that each network has one PAN coordinator.

We violate this requirement, by operating our network without a PAN coordinator. We do this because of two reasons: The first and more practical reason is, that all the tasks the PAN coordinator would need to perform, like managing membership, etc. can – in our case – be done at compile time. Since we don't operate a beacon-enabled PAN, we have no need for any of the additional functionality like the superframe structure or data transfer via the coordinator. The channel (default value 26) and the PAN ID (default value 0x0023) are set at compile time. The second reason is that, because all the other parts of the implementation operate without a central and fixed authority – be it for time synchronization or reaching consensus, introducing one into the system seems like bad taste.

PHY layer The PHY layer describes ways to encode and modulate data for transmission over the Radio Frequency (RF) medium. The standard specifies different encoding and modulation schemes, depending on the frequency band and/or the application one has in mind. Since our transceiver is only able to operate in the unlicensed 2.4 GHz band, we will not discuss properties of the other PHYs. IEEE 802.15.4 partitions the 2.4 GHz band into 16 channels (channels 11 to 26), with a bandwidth of 5 MHz per channel. Note that channels 15, 20, 25 and 26 do not overlap with 2.4 GHz 802.11 WiFi channels and are therefore good choices for situations where WiFi interference is to be expected [MET08]. In the 2.4 GHz band, Offset Quadrature Phase-Shift Keying (O-QPSK) is used

as the modulation technique. The addition of Direct Sequence Spread Spectrum (DSSS) technology with 32 chips per symbol makes IEEE 802.15.4 especially robust against small-band interference. IEEE 802.15.4 is specified for a bit rate of 250 kbit/s in the 2.4 GHz band.

Not only is the PHY layer concerned with physical properties and ways to access the RF medium, it must provide additional services like a Clear Channel Assessment (CCA) and an Energy Detection (ED) primitive to the MAC layer. The CCA is used during Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), when trying to arbitrate access to the shared channel. The ED primitive is intended to be used by upper layers to select channels with good conditions or to switch channels during operation. These services can be requested by the MAC layer and in turn by higher layers.

MAC layer The MAC layer is responsible for arbitrating access to the shared channel and contains all the functionality concerned with PAN coordination and addressing.

All devices wishing to participate in an IEEE 802.15.4 network must have a globally unique 64 bit address (an EUI-64 address – see [IEE17] for more information on the topic). When devices are associated with a PAN, the PAN coordinator may hand out 16 bit addresses to the associated devices. These short addresses are intended to reduce the size of the IEEE 802.15.4 frames, but are only valid inside the PAN where they were issued.

Figure 5.3 shows the layout of an IEEE 802.15.4 Data frame, specific to our implementation. The time synchronization protocol and the higher level consensus protocol both use the same underlying frame layout. Note that this is not the definitive layout of an IEEE 802.15.4 Data frame, since IEEE 802.15.4 supports a variable length address header and an additional security header. The standard defines three more types of frames – Beacon, Acknowledgement and MAC Command – but since we do not actively use any of those types, we will not discuss their structure. The exact size and layout of the frame is important when performing time synchronization over IEEE 802.15.4, because to be able to compensate for the transmission delay, one needs to exactly know the number of bits that are transmitted over the air. See Section 5.3.1 for more details. Since IEEE 802.15.4 has a maximum PHY Service Data Unit (PSDU) size of 127 bytes and our header (including the Frame Check Sequence (FCS)) is 17 bytes long we have 110 bytes left for the payload. As one can see in Figure 5.3 the PSDU is further extended with a PHY Protocol Data Unit (PPDU) header. The PPDU header does not count towards the 127 bytes limit and its exact layout is dependant on the particular PHY layer.

The MAC layer has support for acknowledged frames, but neither our time synchronization protocol implementation nor the consensus algorithms make use of acknowledgements. See [SS21, Chapter 6] for a result showing that acknowledged communication does not benefit the task of reaching consensus under stabilizing message adversaries.

IEEE 802.15.4 networks can operate in two distinct modes – beacon enabled and non-beacon enabled. Depending on the mode, concurrent access to the shared medium is

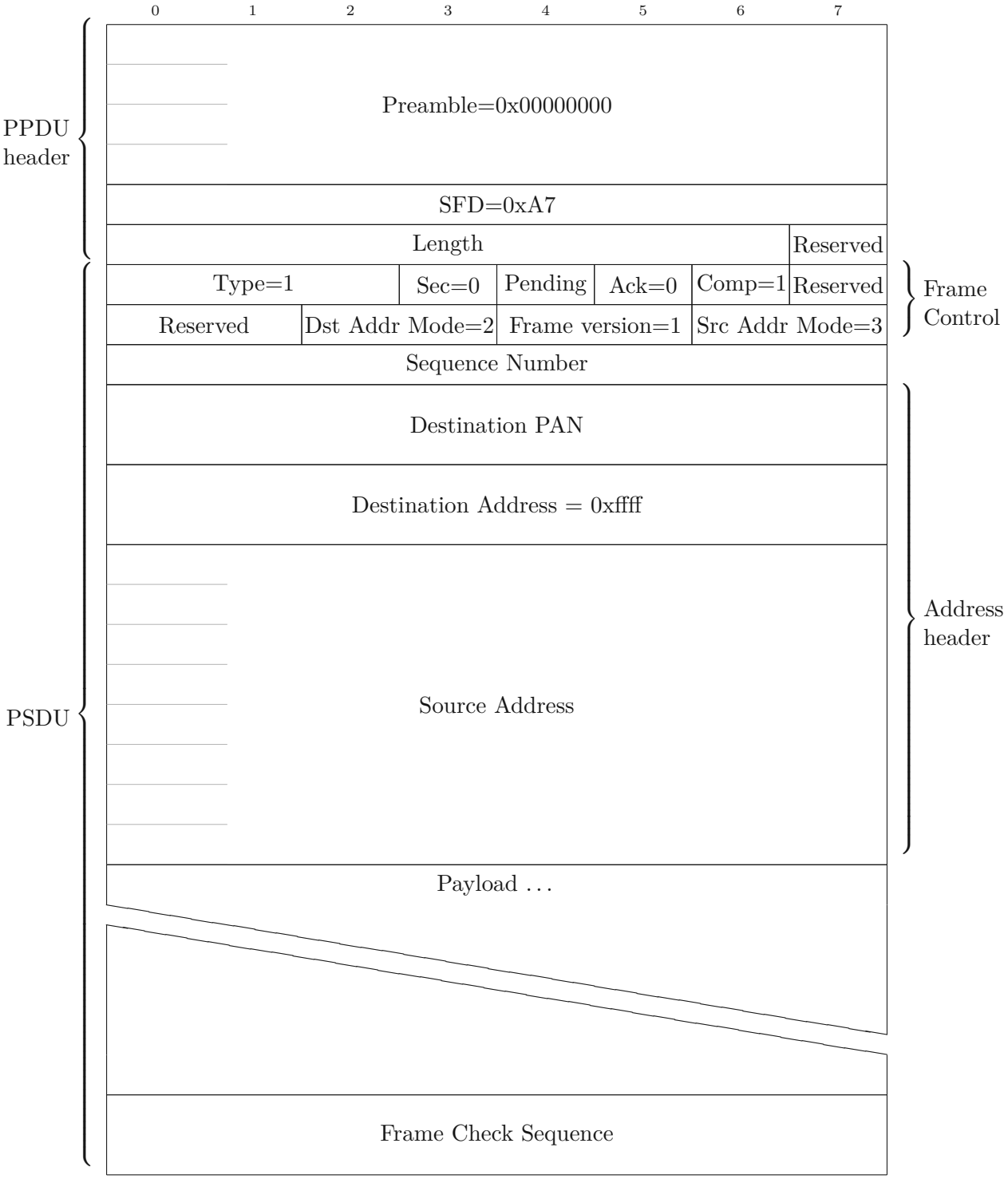


Figure 5.3: The layout of an IEEE 802.15.4 Data frame – specific to our implementation.

handled differently. Networks operating in beacon enabled mode are able to impose a shared structure on the period between two beacons – this period is called a superframe. The superframe structure is a way to split the time between two beacons into 16 equally sized slots. A slot is either reserved for a specific node (called a Guaranteed Time Slot (GTS)) or nodes can contend for access during this slot. The slots where nodes contend for access to the channel are grouped into the Contention Access Period (CAP), followed by the Contention Free Period (CFP) containing the GTSs. In non-beacon enabled mode, all nodes perform a variant of a CSMA/CA algorithm with exponential, random back-off to arbitrate access to the channel. We operate our network in non-beacon enabled mode and derive the synchronization required for the lock-step synchronized rounds from a separate time synchronization protocol.

As we will see later, because of the nature of communication in lock-step synchronized rounds, almost all communication attempts will happen at exactly the same time – leading to massive message loss if no precautions are taken. Since these collisions are very predictable, we chose a different approach than relying on the CSMA/CA algorithm from the specification. For more details see Section 5.4.1.

A more comprehensive overview of IEEE 802.15.4 is given in [Ada06]. The detailed specification document for IEEE 802.15.4-2006 (the exact version supported by our radio) is [IEE06].

6LoWPAN

Time synchronization messages are directly transported over IEEE 802.15.4 frames, because there is no need for any functionality provided by a higher layer in the networking stack. We do not need routing capabilities, because the messages are simply broadcast. Since the time synchronization messages are never going to be bigger than the 110 bytes of available payload space, we do not need any kind of fragmentation support. See Section 5.3.1 for more details on the structure of the time synchronization messages.

For the consensus protocol, we – again – do not need any routing facilities, but messages can certainly get bigger than the maximum possible payload size. To transport messages bigger than 110 bytes, we need some kind of fragmentation support. Instead of implementing a custom solution, we rely on already existing technologies. To get fragmentation support, we use 6LoWPAN as an adaption layer for Internet Protocol Version 6 (IPv6) over IEEE 802.15.4. In addition to fragmentation support, this enables usage of the POSIX-like socket API provided by RIOT's GNRC. The benefits of introducing 6LoWPAN more than make up for the small overhead in message size. See Figure 5.4 for an overview of the networking stack.

6LoWPAN conceptually resides between the link layer and the network layer. It allows IPv6 packets to travel over IEEE 802.15.4 links and therefore provides a way to connect IEEE 802.15.4 networks to the Internet. A major challenge in transporting IPv6 over IEEE 802.15.4 is the mismatch in the supported Maximum Transmission Unit (MTU). IPv6 has a minimum MTU of 1280 bytes, but IEEE 802.15.4 has a maximum frame size of

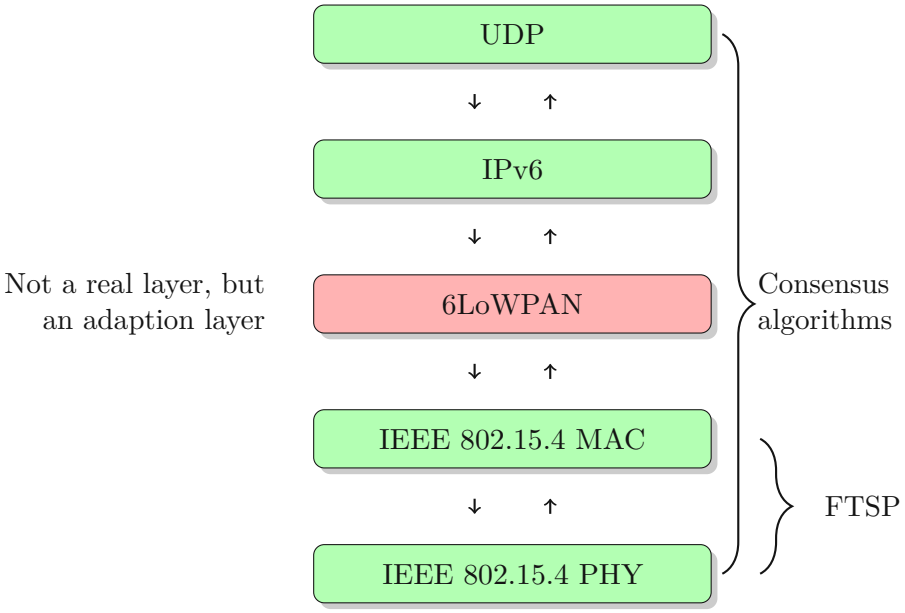


Figure 5.4: Overview of the networking stack

127 bytes. 6LoWPAN – when used in combination with the header compression feature – works as an adaption layer. It does not introduce an additional protocol header that just wraps the IPv6 header, but instead makes use of header compression to represent the IPv6 header in a much more space efficient manner. The space efficiency comes from the fact that it is tightly coupled with IEEE 802.15.4 and can therefore make assumptions about addresses being shared between the link layer and the network layer. Using Stateless Address Autoconfiguration (SLAAC), we are able to derive our IPv6 addresses directly from the IEEE 802.15.4 link layer addresses. In addition to compressing the IPv6 header, 6LoWPAN's header compression is also able to compress details of the User Datagram Protocol (UDP) header. Since communication of all nodes is within the same IPv6 subnet, the 6LoWPAN header is only 2 bytes big in our case. It is interesting to note that RFCs like RFC 7668 "Bluetooth over IPv6" define adaption layers using technology and concepts very similar to 6LoWPAN for other physical layers.

The old adage of "everything over IP, IP over everything" and the resulting IP hourglass design have certainly not lost its relevance. Last but not least, the use of IPv6 and a high-level socket API would make it possible to run the code, without any changes to the networking part on the host system. When running on the host, no 6LoWPAN adaption layer is necessary, because IPv6 is directly transported over Ethernet frames.

Because we are transporting two different types of payload over IEEE 802.15.4, we need a way to distinguish between packets carrying 6LoWPAN and packets carrying time synchronization messages. Since IEEE 802.15.4 does not have an upper protocol field in its header, 6LoWPAN specifies that the first byte of the payload is used as a dispatch field to decide if the data is a 6LoWPAN packet or some other content. We make use

of this feature to mark our time synchronization messages as carrying non 6LoWPAN content – Not A LoWPAN Frame (NALP) content in the terminology of RFC 4944. For more details on 6LoWPAN refer to the current standard documents RFC 4944 and RFC 6282.

5.3 Global time and lock-step rounds

A basic premise of our system model is the existence of lock-step synchronous rounds. To be able to implement the algorithms, we first need to create mechanisms that allow the nodes to execute code within a round structure. For nodes to be able to execute code in lock-step rounds, there must exist a shared notion of time or at least a notion of synchrony.

One viable option would be to use a time signal provided by an external and sufficiently precise time source like Global Positioning System (GPS). Another way to create a notion of common time within the group of participating nodes is to use a time synchronization protocol. This kind of common time must not necessarily have any connection to wall time. We chose to implement a time synchronization protocol to establish a common time base, instead of using an external time source like GPS. Equipping all nodes with GPS receivers, does not seem like a practical approach for low power and low cost IoT systems.

A multitude of time synchronization protocols targeted at WSNs exist - see Section 1.2.2 for an overview. One of the better known protocols is FTSP introduced in [MKSL04]. It is a suitable candidate for our implementation, because of its relatively low complexity both implementation and communication wise. FTSP is explicitly tailored to nodes with little memory and low CPU resources.

To be able to understand the details of the implementation, we must first understand the basic principles behind FTSP. The next section will describe the basic operation of FTSP – afterwards we will delve into the details of our FTSP implementation and we will talk about how we are going to derive a round structure from the shared notion of time.

5.3.1 FTSP

The basic idea behind FTSP is that a dynamically selected node – called the root – broadcasts messages containing a timestamp in regular intervals. The timestamp contained in the broadcast message is called the global timestamp and taken on the brink of sending the broadcast. Nodes, upon reception of a time synchronization message, take another time stamp – dubbed the local timestamp. Using matching pairs of remote and local timestamps, the nodes try to synthesize a time that tracks the root node's time as closely as possible. To facilitate distribution of the time information in a network, where the root node is not able to directly reach all other nodes, each non-root node broadcasts its local estimate of the global time, after having received a sufficient number

of timestamp pairs. A node that has received a sufficient number of time synchronization messages from the same root is said to be synchronized. Sequence numbers in the messages allow the detection of duplicates to avoid processing messages twice.

Precision and delays

The high precision that FTSP is able to achieve – the authors of [MKSL04] report a synchronization accuracy of up to 1.5 µs – is due to two techniques: MAC layer timestamping and skew prediction.

MAC layer timestamping MAC layer timestamping is not so much an algorithmic as an implementation technique. It describes the practice of taking the timestamps as close as possible to the point of actually sending or receiving a time synchronization message. The name "MAC layer timestamping" is a bit of a misnomer, because the ultimate goal is not to take the timestamp in the MAC layer, but – if possible – on the radio driver level. On one hand, MAC layer timestamping violates the layer encapsulation principle and manipulates packet content without going through the appropriate layers of the networking stack. On the other hand, this technique does a good job in minimizing the non-deterministic delays that would otherwise decrease the achievable precision. MAC layer timestamping has its limits, because without special hardware support, the sender side timestamp must always be taken before the bytes containing the timestamp are transmitted and vice-versa for the timestamp taken on reception. There is hardware support for timestamping in some Ethernet Network Interface Cards (NICs), but to our knowledge, none in any of the low cost SoCs targeted at the IoT ecosystem. But even with hardware timestamping support, there is a fundamental limit, as to how accurate time synchronization between nodes can get. [LL84] proved that even in a fully connected network with n nodes without failures and clocks that are not skewed against each other, clocks cannot be synchronized more closely than $\epsilon(1 - \frac{1}{n})$, with ϵ being the always remaining non deterministic delay in message delivery.

Skew prediction and linear regression When it comes to time synchronization, skew (also called offset) and drift are two important parameters that describe the relationship between clocks. We assume a clock to be a function $C(t)$ that maps real time t to the respective clock value. Given two clocks $C_a(t)$ and $C_b(t)$ we call $|C_a(t) - C_b(t)|$ the skew between these clocks. The drift of a clock captures the rate at which it deviates from a given reference clock in a given amount of time. $\frac{C_a(t_1)-C_a(t_0)}{C_b(t_1)-C_b(t_0)}$ would be the drift between clocks $C_a(t)$ and $C_b(t)$ in the interval $[t_0, t_1]$. If clock crystals would oscillate at the exact same frequency, the drift between any two clocks would be exactly one. A single synchronization message would be enough to synchronize a clock to a reference clock. After determining the skew, all nodes could derive the same global time and keep it indefinitely. But in reality, quartz crystals do not oscillate at exactly the same frequency and the time synchronization protocol needs to compensate for this fact. The second technique that helps FTSP achieve high precision is the use of linear regression to make

a prediction on how the local quartz drifts against the root node's quartz. FTSP makes some reasonable assumptions about the behaviour of the quartz crystals or other kinds of oscillators that provide the local time to the node. The local time is assumed to have small local error (when asking for the local time in short succession, only a small change in local time will have happened) and it assumes that the error between different crystals can be modelled closely enough by a linear function, because FTSP is only able to make linear predictions.

The FTSP paper also mentions other techniques like taking multiple timestamps to average over them and determining the radio's byte alignment, but we will not go into detail on those techniques, because they are not applicable to the kind of radio hardware that we have at our disposal.

Root election and resilience

Now that we know the main techniques that allow FTSP to achieve high synchronization precision, we are going to talk about the mechanisms in FTSP that allow dynamic and robust election and reelection of a root node.

The root node is the central source of truth regarding global time. It is dynamically elected during the execution of the algorithm and – should the current root fail – a new root is elected with minimal disturbance to the common time. FTSP's root election mechanism makes the assumption that all nodes possess a unique ID. The core rule of the root election mechanism is that the node with the lowest ID is considered to be the root. To facilitate the election process the nodes follow these steps: Each node i has a variable $root_i$ that contains the current root's ID – as seen from node i's point of view. Every FTSP synchronization message contains $root_i$ and upon reception of a message, all nodes compare the received node ID with their current notion of the root. Should the received message contain a node ID that is bigger than $root_i$, the message is simply ignored. Should the received node ID be smaller than $root_i$, the node assumes that it has seen a new root, updates its local variable and starts – if it has seen enough synchronization messages – rebroadcasting messages from the new root. After some time, only one node is left broadcasting on its own – the newly elected root. Should a node not hear from its current root for a specified amount of time, the node clears the contents of $root_i$ and starts looking for a new root.

To guarantee a smooth transition between different roots, the algorithm takes care to reuse the already established linear regression parameters skew and offset. This allows a continuous global time even when the root changes. The initial waiting period for new nodes that just joined the network enables them to acquire the parameters of the current global time before they might take on the role as the new root node. An interesting consequence of this approach is the fact that the root node might broadcast a global time that is different from the root node's local time.

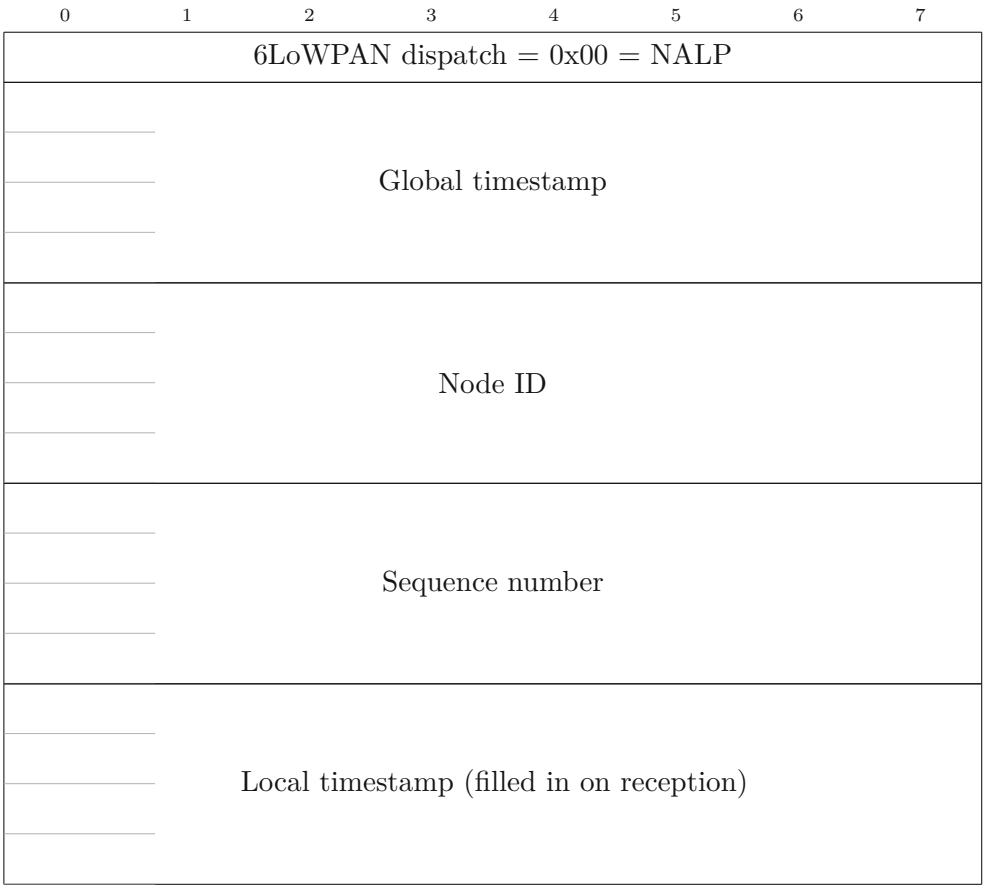


Figure 5.5: Layout of synchronization messages

Implementation

The implementation is split into two threads of control – a sender and a receiver thread. The receiver thread listens for FTSP packets and processes them according to the rules laid out above. The sender thread periodically wakes up, determines the status of the node (is the node synced?, is the node a root?, ...) and sends messages accordingly. Figure 5.5 shows the layout of our synchronization messages. The first byte is used to mark the payload as not being 6LoWPAN content. The global timestamp field is set by the sender, as are the node ID and the sequence number fields. The local timestamp is filled in on reception by the receiver.

MAC-layer timestamping Our implementation takes a slightly different approach to MAC layer timestamping than what Maroti et.al. proposed in [MKSL04]. The radio used in the FTSP paper is byte-oriented, whereas the nRF52840's radio has a DMA interface. The paper describes a MAC timestamping algorithm, where a timestamp is taken after each transmitted byte and the average of these timestamps is used as the timestamp to include in the synchronization message. This approach is not possible for a

radio with a DMA interface, since the whole message is transferred in bulk to the radio via the DMA controller. Since our radio has a "Bitcounter" facility, one could nonetheless get timestamps after each transmitted byte. The bitcounter facility allows the radio to generate an interrupt after a configurable amount of bits have been transmitted or received. But because our radio's interface is DMA-based, we cannot append a timestamp to a message that is already being transmitted by the radio. We could transmit the timestamp in a separate message and somehow fuse the message we used to get the timing information with the message containing the timestamp on the receiver side, but we decided to take another approach. We identified the location in the networking driver, where the DMA transfer is initialized and are taking the timestamp right before the transfer is started. The timestamp on reception is taken in a similar fashion – as soon as the Interrupt Service Routine (ISR) for a received packet is triggered the reception timestamp is taken and appended to the message. The adaption we had to make in the RIOT networking stack is available in `cpu/nrf52/radio/nrf802154/nrf802154_radio.c`.

Linear regression Regarding the implementation of the linear regression logic, care has to be taken to not introduce numerical errors in the floating point computations. To keep the error low, we perform the linear regression not from the local timestamp to the global timestamp, but from the local timestamp to the offset between global and local timestamp. This technique is used in the FTSP implementation available in TinyOS as well. In addition to using the offset instead of the global timestamp, we try to perform all calculations using integer data types where possible – this is especially important when performing additions. To make sure that the floating point error stays within a reasonable bound, we created a test suite generator, which generates linear regression calculation tests for the whole range of the FTSP timestamp. See `consensus/ftsp/tests/ftsp_linear_regression/generate_testcases.py` for the testcase generator.

Since we use a 32 bit unsigned integer for the global timestamp and our unit of time are µs, after approximately 71.58 min the counter will overflow. The choice to use a 32 bit unsigned integer is motivated by the fact that we need to perform floating point calculations. Since our hardware only has support for single precision floating point numbers, using a 64 bit unsigned integer would mean that we would sooner or later leave the range where we can perform meaningful and somewhat precise calculations.

Applying the corrections The major correction we have to apply to the received timestamp is due to the transmission of our packet at a speed of 250 kbit/s. After the timestamp has been taken and the DMA transfer is initiated, the radio will spend considerable time transmitting the bits at the speed mandated by the IEEE 802.15.4 standard. Since we use the offset between the timestamp contained in the message and the timestamp obtained on reception, this offset includes the time it took to transmit the bits over the air and we need to compensate for it. We will ignore the delay introduced by the propagation of the electromagnetic waves, because an electromagnetic wave travels with a speed of about 300 m/µs and assuming a distance of 100 m (which is at the upper

boundary of what is possible with IEEE 802.15.4) the introduced delay would only be around 0.3 µs.

On top of the transmission delay, we observed an almost constant delay of about 75 µs. This delay might capture the DMA setup and transfer times and radio specific delays. We did not investigate the causes of the delay any further, but just included it in the correction logic.

Fault injection Each node can be given a blocklist containing nodes, whose messages should be dropped. This blocklist can be activated and deactivated during runtime. The blocklist functionality allows the controlled injection of faults during the evaluation runs to test the code under different scenarios.

5.3.2 FTSP evaluation

We are now going to briefly describe our synchronization precision measurement setup and present the obtained results. We used two different setups to obtain performance metrics from our FTSP implementation. Our measurement setups differ from the measurement setup in the FTSP paper. In that paper, a special node queries all nodes in the network and the nodes report their timestamps. We don't do accuracy measurements over the air. Our first setup uses an oscilloscope to directly capture the moment of signal change on a GPIO pin. This has the benefit of not needing to implement a separate precision reporting infrastructure – the disadvantage is that this approach will not scale up to a high number of nodes or a setup where the nodes are spaced very far apart, because the nodes need to be connected to the inputs of the oscilloscope.

The second measurement setup consisted of 20 nodes running FTSP and logging various information like the current global timestamp and the node state at predefined points in time to an SD card. These logs were later used to extract the desired performance metrics.

Oscilloscope measurement setup

To be able to measure the accuracy with an oscilloscope, we devised the following scheme that makes use of the lock-step synchronous round infrastructure. We configured the round generation logic to use rounds consisting of 2 phases. At the start of every new phase each node under observation toggles a pin connected via shielded cables to an oscilloscope. The oscilloscope's output is captured on a PC using Virtual Instrument Software Architecture (VISA). A script post-processes the captured traces and determines the time difference between the signal transitions. The script to collect data from the oscilloscope is `consensus/rigolyze.py`. The script to post-process the data is `consensus/ftspalyze.py`.

Measurement run 1 The first measurement run used an FTSP period of 5 s and a phase duration of 8.38 s. The nodes were placed next to each other – simulating optimal

conditions. After some time, a failure of node 2 was simulated and later the recovery of node 2. Some time after node 2 came back online, a root failure was simulated by powering off node 1, making node 2 the new root. Later on, node 1 was powered on again, taking over the role as the root. See Figure 5.6 for a plot of the time differences of the nodes to the current root. The achieved precision ranges from a difference of around 28 µs for node 2, and 8 µs for node 3 down to 3 µs for node 4, depending on the current root node. It is interesting to note that the nodes seem to have a quite constant error, leading to the conclusion that node specific calibration parameters could further increase the achievable performance of our implementation. The counter overflow is handled smoothly, because of the quite high FTSP period.

Measurement run 2 The second measurement run used the same FTSP period of 5 s and again a phase duration of 8.38 s, but this time the nodes were distributed in an apartment in different rooms. The same failure protocol as in run 1 was performed. Figure 5.7 shows a plot of the time differences of the nodes to the current root. We are able to observe a similar behaviour when the nodes are placed further apart and not in direct line of sight. The error seems to be quite constant for each node as well and overflow is handled gracefully.

Measurement run 3 The third measurement run used an FTSP period of 30 s and a phase duration of 2.09 s. The nodes were placed in the same locations as in the second measurement run. This time, no failure protocol was performed. See Figure 5.8 for the time differences of the nodes to the current root. Something interesting can be observed: Even though the period is higher, the node specific errors are now not as pronounced as in the first and second measurement runs, but the extreme values of the time difference error are much higher. This might be due to the increased phase duration and the concomitant increase in measurement error. Due to the high FTSP period, recovering from the counter overflow takes much longer.

SD card measurement setup

20 nodes where placed in close proximity to each other and different blocklist settings were given to the nodes to allow the recreation of different connectivity scenarios. Two scenarios where tested - a regular grid without any blocklist configuration and a kind of line graph that forced all the nodes to have a different root. These experiments where done as a kind of sanity check that the FTSP implementation works as expected.

No fault injection The first measurement run used an FTSP synchronization period of 3 s. Figure 5.9 shows that the biggest synchronization error between all 20 nodes placed in a regular grid stays well below 250 µs in the worst case. The second measurement run increased the FTSP synchronization period to 30 s. This does not have a detrimental effect on the maximal synchronization error - see Figure 5.10. These two measurement runs represent optimal network connectivity conditions.

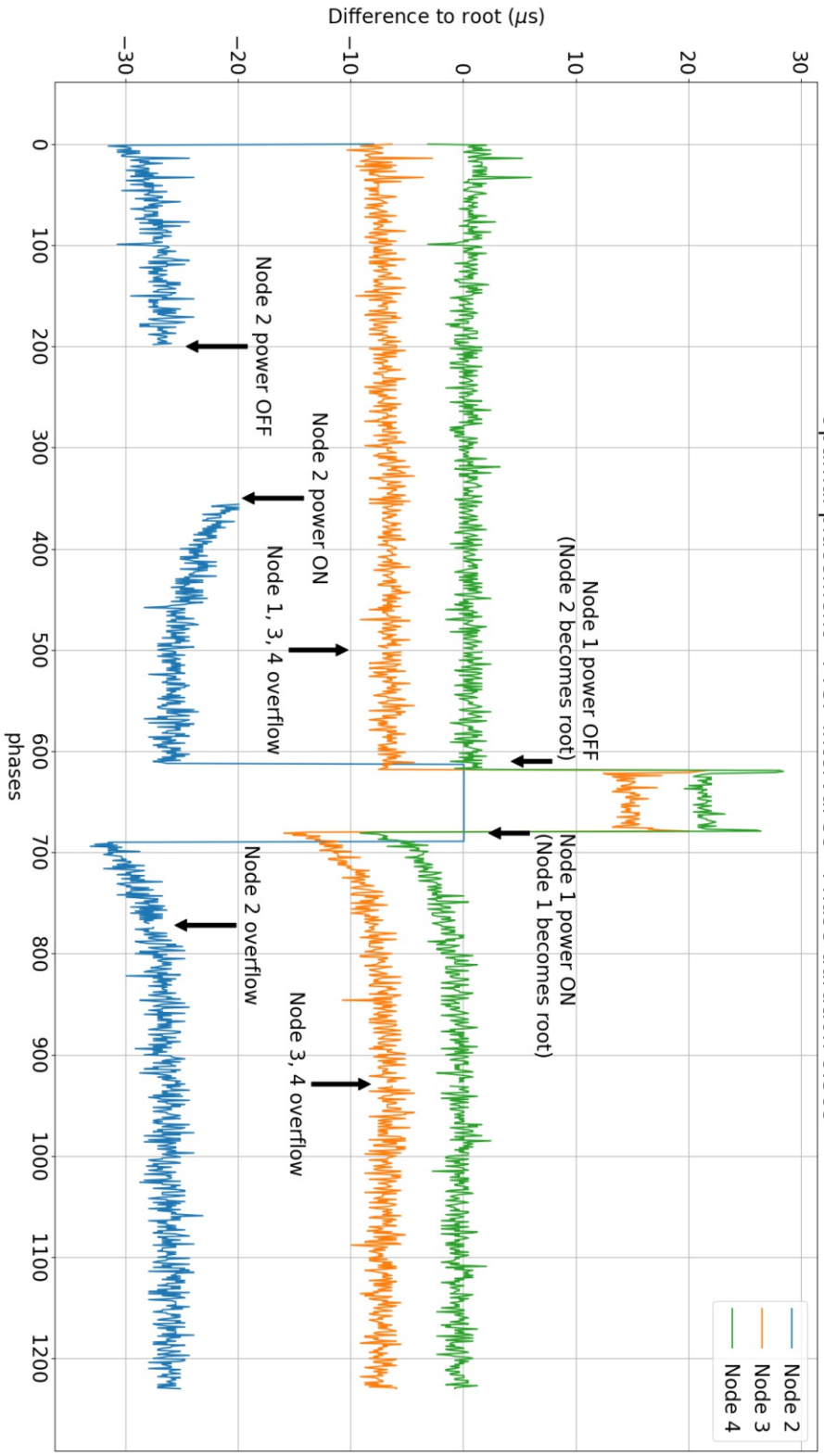


Figure 5.6: FTSP precision measurements – run 1 – oscilloscope setup

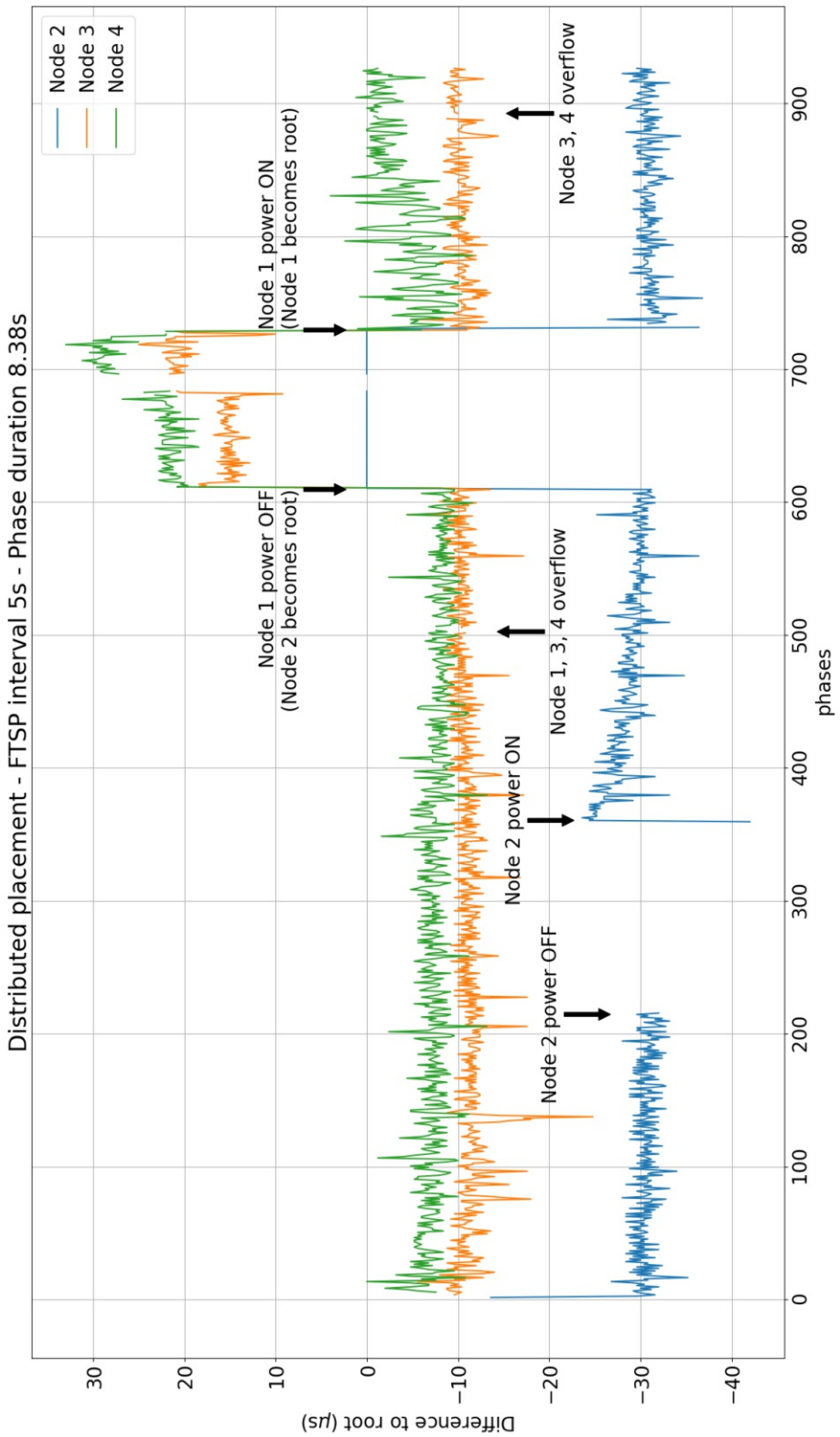


Figure 5.7: FTSP precision measurements – run 2 – oscilloscope setup

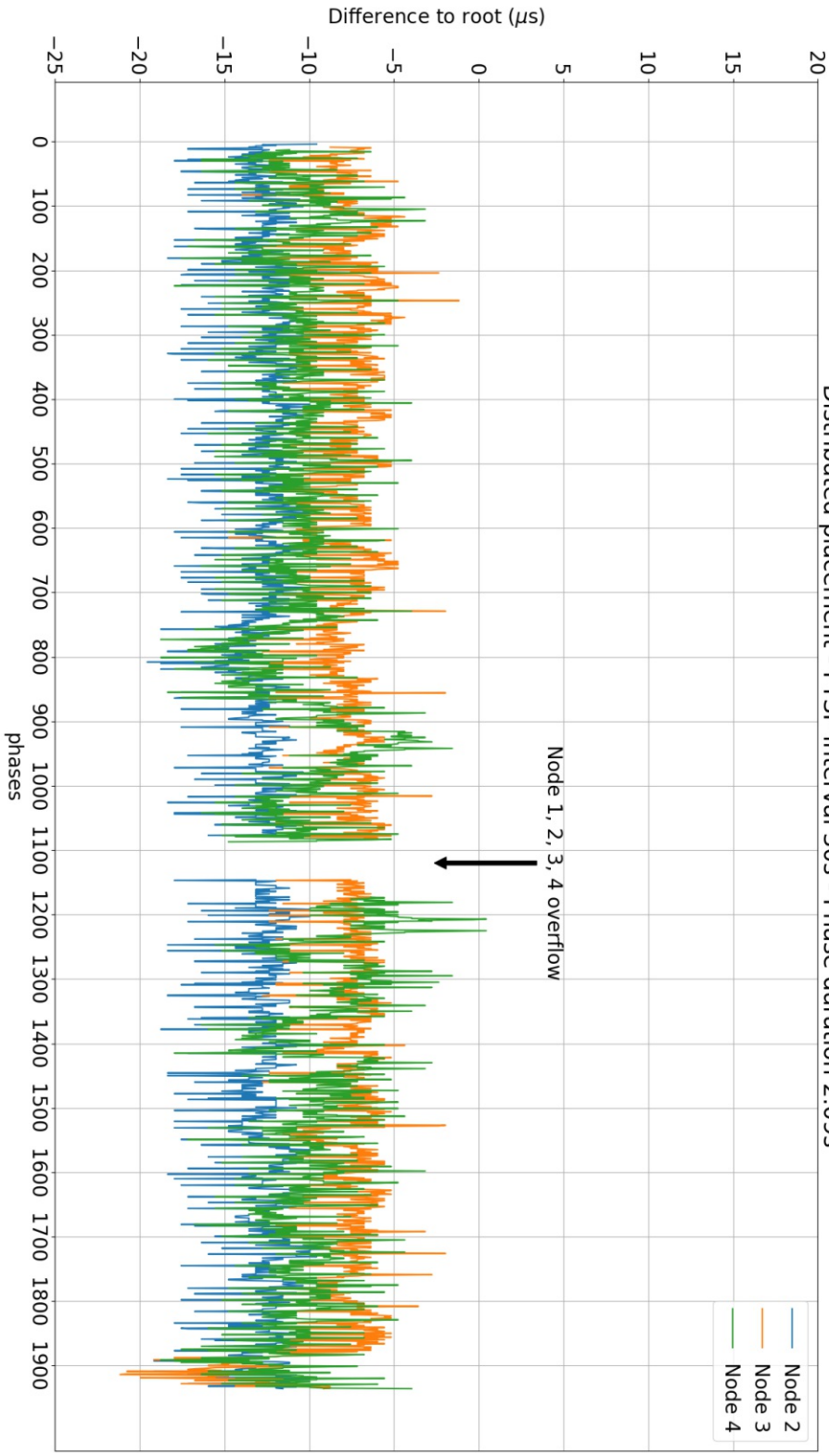


Figure 5.8: FTSP precision measurements – run 3 – oscilloscope setup

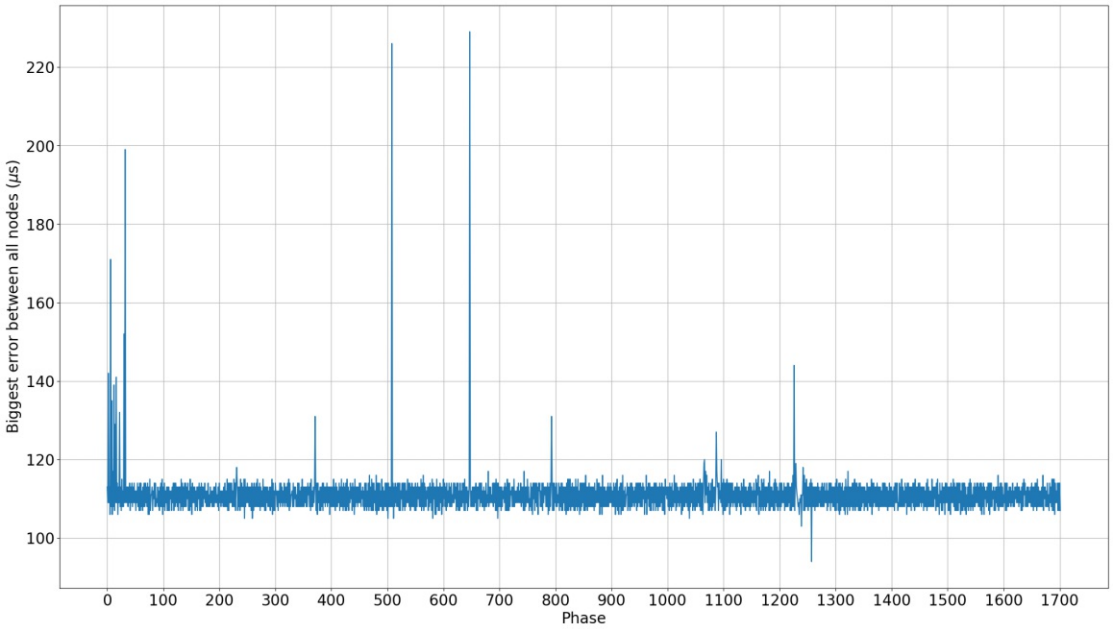


Figure 5.9: FTSP precision measurements – run 1 – SD card setup. The biggest error between all 20 nodes placed in a regular grid with an FTSP synchronization period of 3 s and no blocklist settings.

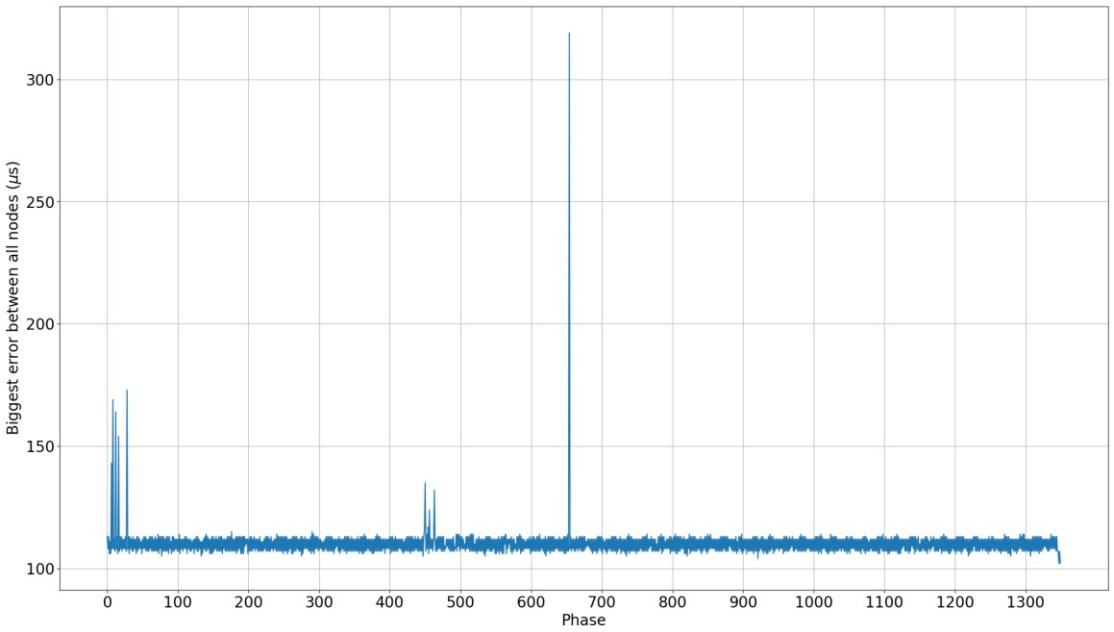


Figure 5.10: FTSP precision measurements – run 2 – SD card setup. The biggest error between all 20 nodes placed in a regular grid with an FTSP synchronization period of 30 s and no blocklist settings.

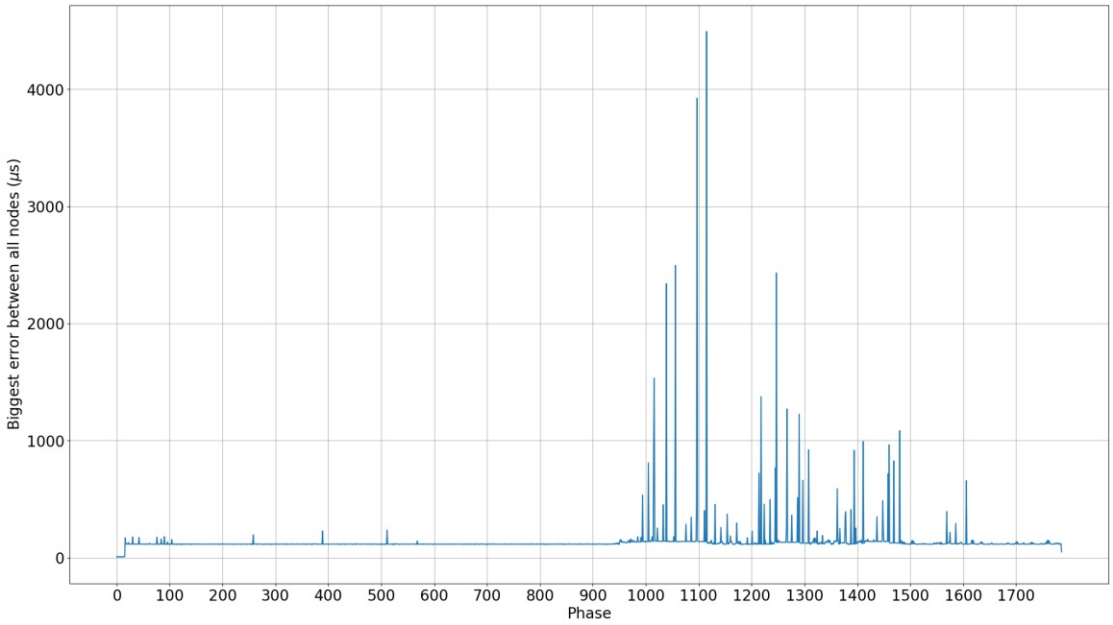


Figure 5.11: The biggest error between all 20 nodes placed in a regular grid with an FTSP synchronization period of 3 s and a blocklist activated after 30 min that forces a line graph with respect to the visible root nodes.

With fault injection To evaluate the behaviour of our FTSP implementation under non-optimal conditions we ran a third experiment with an FTSP synchronization period of 3 s and error injection starting after 30 min that forced each node to see a different FTSP root node. Figure 5.11 shows that even in the worst case the maximal synchronization error between all nodes stays below 4.3 ms. This is well below the required phase duration of 2.09 s used in the experimental evaluation of the consensus algorithms - see Section 6.2.

5.3.3 Lock-step synchronous rounds

Now that we are able to robustly establish a common time base between the nodes, we can take the next step and create an infrastructure to execute code in a lock-step fashion. The basic idea behind our approach is to define points in time where new rounds or phases start, and have all nodes – using global time – determine their distance to the next starting point and sleep until then.

Because our system model dictates that the lock-step synchronous rounds consist of separate non-overlapping phases, the basic unit of synchronization in our implementation is the phase. A certain number of phases makes up a round. At first, the nodes try to synchronize to a round boundary, and after that they will always synchronize to the next phase boundary.

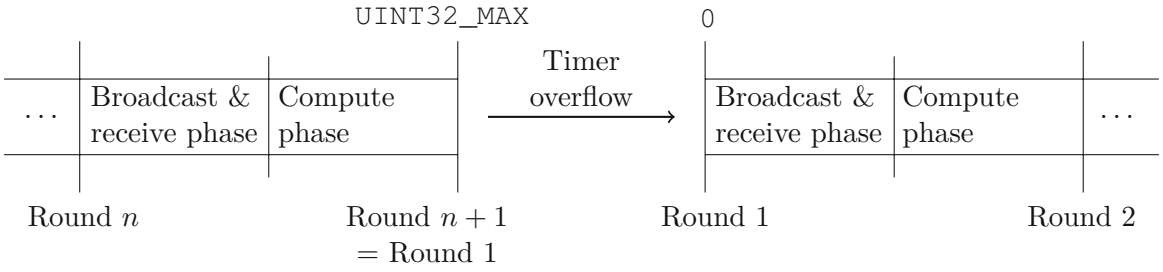


Figure 5.12: Phase and round structure of global time. To map a timestamp to a phase after global time has overflowed, we need the duration of a round in µs to divide `UINT32_MAX` + 1.

Properties of rounds and phases with respect to global time

The algorithms presented in Chapter 3 assume rounds to be structured into three phases: broadcast, receive and compute. This is mostly done to have a nice conceptual structure, but this structure cannot necessarily exist in that exact way in an actual implementation. What we need to guarantee is that all messages are broadcast and received before the nodes start the compute phase. When doing time based structuring of the global time into rounds and phases, one must take into account that global time will eventually wrap around. Note that we must be able to uniquely identify the phase a timestamp belongs to, because the receiver part of the consensus implementation has support for a mode where it does not synchronize itself to a round structure. Instead, on the reception of a message, it determines the current broadcast phase start and end points and checks if the message contains a timestamp that falls into this range. In other words: The receiver must be able to determine if a timestamp belongs to a broadcast phase or not. This is necessary to prevent spurious messages from being received in this mode and thereby enhances the resilience of the system. Therefore, we need the following property to hold if we want to handle global time wrap-around reliably: The duration of a round in µs must divide `UINT32_MAX` + 1, which is the maximum timestamp value (in µs) + 1. This allows us to uniquely identify which phase a timestamp belongs to – even after wrap-around. See Figure 5.12 for an illustrated explanation of the global time wrap-around and its effect on the round/phase structure.

As already mentioned, the algorithms assume a round that consists of three phases, but the receive phase naturally happens at the same time as the broadcast phase, because as soon as nodes start to broadcast other nodes will receive data. This means that the broadcast and receive phase happen at the same time and we are only left with the compute phase. Therefore, a round in our implementation consists of a broadcast phase, where all messages are broadcast and received to a temporary variable and a compute phase, where the received messages stored in the temporary variable are integrated into the main data structure and the computations are performed.

5.4 Consensus Algorithms

All the necessary parts are now in place to shift our focus to the implementation of the algorithms as a whole. We implemented the terminating and stabilizing variants of the root component based algorithms and the MinMax stabilizing consensus algorithm.

The implementation naturally decomposes into various modules. The most important ones are listed below. Each module corresponds to a directory in our source tree.

`ftsp`	Responsible for providing a global time
`ddn`	Contains all the root component based data structures and graph algorithms
`consensus`	The basic consensus machinery shared by all algorithms
`consensus_stabilizing`	The implementation of the root component based stabilizing consensus algorithm
`consensus_terminating`	The implementation of the root component based terminating consensus algorithm
`consensus_minmax`	The implementation of the MinMax based stabilizing consensus algorithm

For a more detailed overview of the filesystem layout see Appendix A.

5.4.1 System overview

The `consensus` module is split into two threads of control. A receiver thread, responsible for the reception and processing of network packets and a computation thread, which is in charge of both broadcasting the node's state and executing the algorithm's main computations. See Figure 5.13 for a system overview.

The receiver thread is not synchronized to the round structure and constantly listens for incoming messages. To check if a received message was sent in the current broadcast phase, the receiver either compares the timestamp embedded in the message with its own global time or uses the global timebase to determine if a message belongs to the receive phase without looking at the message timestamp embedded in the consensus message. The define `CNS_IGNORE_MSG_TIMESTAMP` switches between the different modes. If a message does not belong to the current broadcast phase, the message is dropped. See Section 5.3.3 for a discussion on the properties of the round/phase structure that need to hold, to enable the receiver thread to reliably compute the current broadcast phase boundaries. We do not merge the received messages directly into the main data structure, but keep a temporary variable, because we want to make sure that the reception of a message does not change the state of a node that has not yet sent its broadcast message.

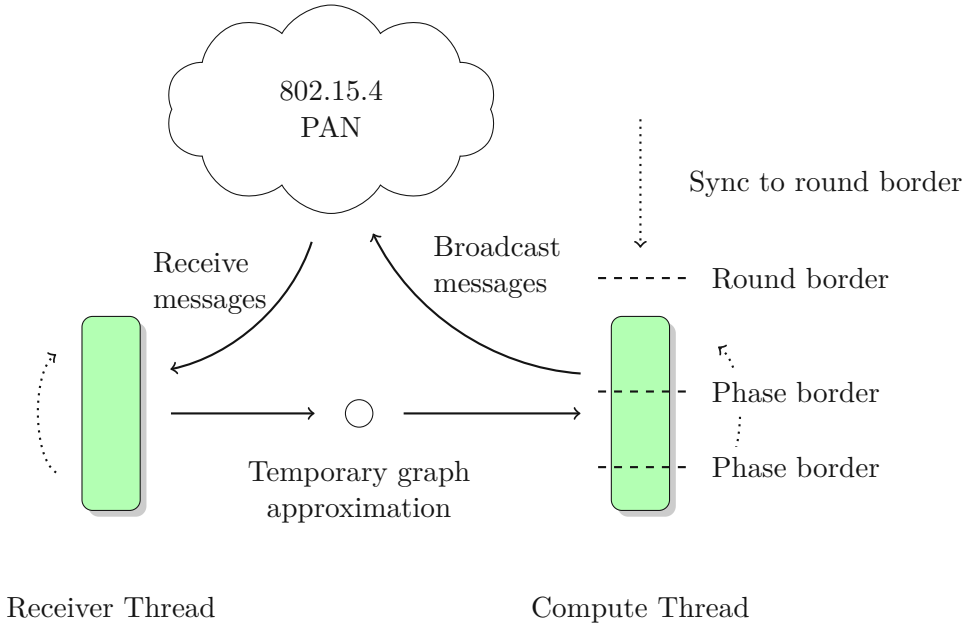


Figure 5.13: System overview of the consensus part

The temporary variable is merged into the main data structure at the beginning of the computation phase. Since the temporary variable is accessed by both receiver and computation thread, a lock controls access to the data structure.

The second thread in the system is the computation thread. In contrast to the receiver thread, the computation thread is synchronized to the round structure. After synchronizing to the round structure, the computation thread first broadcasts the current state in the broadcast phase. After entering the computation phase, the thread merges the received state information from the other processes into its main data structure, then clears the temporary variable and finally takes the necessary computation steps, depending on the variant of consensus implemented. When broadcasting messages in the computation thread, we run into the problem that all nodes are precisely synchronized and would attempt to broadcast messages at exactly the same time. We could leave the channel arbitration to the MAC layer to perform the CSMA/CA algorithm mandated by the IEEE 802.15.4 standard. Because of the nature of the exponential back-off algorithm, however, this would introduce non-determinism into the system and make estimation of the necessary phase duration more difficult. Instead we chose a simpler approach – all nodes must wait $(\texttt{node\_id} - 1) * t_{msg}$, before broadcasting the local state. t_{msg} is the time it takes to transmit the biggest consensus message. Due to the nature of the data structures and our knowledge of N the maximum size of a consensus message is known at runtime. See Section 5.4.2 for more details on the encoding scheme and on pruning the graph approximation. Using this scheme enables us to find an upper bound for the duration of a phase when we know the number of nodes involved. This means that

for algorithms that require lock-step synchronous rounds (and are working in a shared medium – like the RF medium) the implementation needs to have at least a knowledge of N to determine the maximum round/phase duration – even if the algorithms themselves do not require this kind of knowledge.

Our implementation has the additional feature of being able to execute two consensus algorithms at the same time. This allows us to evaluate certain properties of different consensus algorithms under exactly the same real world conditions. When two algorithms are run in parallel their messages are fused together, to make the communication conditions exactly the same for both algorithms.

5.4.2 Implementation details

First we are going to discuss some global implementation details and then we will highlight various aspects of the root component based algorithms and the MinMax based algorithm.

Choosing an appropriate round duration The phase duration needs to allow for enough time for all messages to be sent one after the other. It takes 802.15.4 30 µs to transmit one byte, because the nominal speed of transmission is 250 kbit/s. The maximum message size is dependant on the number of nodes taking part in the algorithm and the different algorithms being executed.

Node IDs The boards have factory set unique identification registers that we map into the range 1 to N. This mapping happens in the module `node_id`. When running the code in the simulator the node ID is passed as a commandline argument.

Debugging and tests The ATUSB 802.15.4 dongle[6], suitable for sniffing IEEE 802.15.4 with a recent Linux PC, was extremely helpful. See `consensus/scripts/sniff.sh` for a script to put the dongle into promiscuous mode and start sniffing on a given channel. This dongle in combination with Wireshark[7] was a powerful tool to debug various network related problems with relative ease.

All modules come with an extensive unit test suite contained in the `tests` subdirectory of the respective module. The `consensus/scripts` directory contains scripts to run various kinds of automated tests. See Appendix A for more detailed instructions.

Consensus messages The consensus messages sent over the air are rather simple. They only contain the node ID, a timestamp – not used when executing with `CNS_IGNORE_MSG_TIMESTAMP` – and the serialized local state of the node. Figure 5.14 shows the layout of the consensus messages.

[6] http://shop.sysmocom.de/products/atusb
[7] https://www.wireshark.org

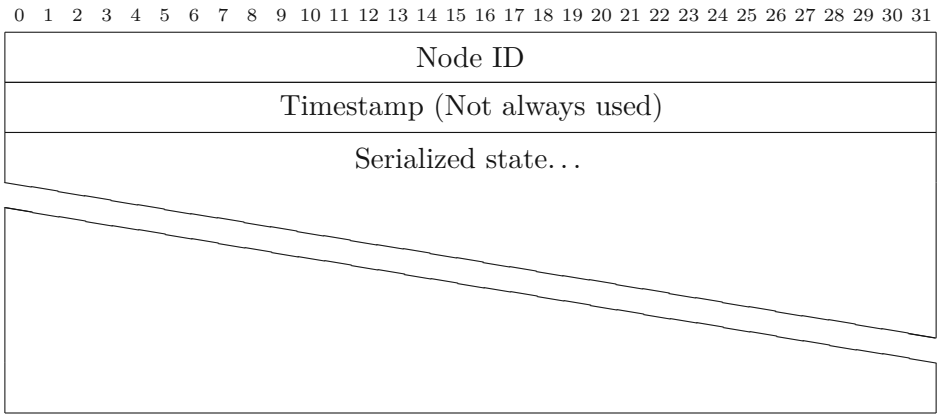


Figure 5.14: Layout of the consensus message

Root component based algorithms

Data structures In the algorithm's pseudocode, the graph approximation A, the state S and the set of known processes P are kept separate. In our implementation, we decided to merge all three data structures into one. We store the necessary state information with the vertex representing the node in the graph. This has the benefit that we only have one central data structure – the graph approximation (`ddn_t` in the source code) – to perform our operations on. Merging the data structures does not have an impact on the correctness of the algorithm, because the algorithm's correctness arguments make no assumptions with regard to the separateness of the data structures.

The DDN is implemented as a linked list of graphs. Graphs themselves are implemented as sparse adjacency matrices – again using linked lists. The main graph theoretical algorithm used to find strongly connected components, necessary to identify roots, is Tarjan's algorithm. See the function `ddn_get_sccs` in `consensus/ddn/algorithm.c` for the implementation.

Serialized state The basic idea behind the state encoding is to represent each node u of round s as a suitably sized bitfield and set bit $v-1$ if there is an edge going from u to v in round s. See the functions `ddn_encode` and `ddn_decode` in `consensus/ddn/ddn.c` for the implementation of the state encoding and decoding.

Pruning the graph approximation To make the implementation practical, the graph approximation has to be pruned at a certain depth. Examining both algorithm's pseudocode, we note that we do not go more than $D+1$ rounds into the past in the case of stabilizing consensus. For terminating consensus we do not go more than $N(D+2N)$ rounds into the past. This allows us to prune the approximation at the respective depth.

Relative round numbers The implementation uses relative round numbers from 0 upwards, with 0 being the current round. The algorithm encodes relative round numbers a negative numbers, describing how far back "in time" a round is. The choice to use non-negative numbers comes natural in the implementation, because the round numbers can be thought of as an index into an array of graph approximations.

MinMax based algorithm

Data structures The main data structure of the MinMax algorithm is `age_t`, a linked list of objects storing an age for a value. Encoding this linked list for transport over the air is done by flattening the list to a compact array. All functions operating on `age_t` can be found in `consensus/consensus_minmax/age.c`.

Cut-off function The special value δ, used to determine the cutoff point between the minimum and the maximum operation is calculated based on the current round number as $\delta(r) = \lfloor \frac{r}{2} \rfloor$.

5.5 Future Extensions

We have now presented a bottom-up view of the implementation and are now going to briefly discuss possible future extensions to various parts of the system.

Global time It would be interesting to investigate how good a fit the superframe structure of IEEE 802.15.4 is to implementing lock-step synchronous rounds. One could either directly use the GTSs or derive synchronization from the beacons. When using the GTSs and equating one superframe with one round, the limit of 16 slots per superframe limits the number of nodes that could participate severely. A solution might be to make a round span more than one superframe. The beacons could provide a basic synchronization point, but they do not provide any of the features we expect from a dedicated time synchronization protocol. The beacon frame does not contain an explicit time stamp field (although the optional user-defined beacon payload could be used for this) and there is no robustness against failing PAN coordinators. The issue of PAN coordinator (re-)election could be handled on a higher layer – see [CCA10] for a possible solution.

FTSP Future improvements to the FTSP implementation could be to increase the robustness during overflow and an investigation if node specific calibration parameters can further increase the performance. In particular, an approaching overflow could be detected and a separate linear regression table used to store values until it contains enough entries to be used as the main linear regression table.

Security Since our implementation's focus is not on security, we do not go into details regarding the various security features that both 6LoWPAN and IEEE 802.15.4 offer. In production IoT systems, security is of the utmost importance, because of the often sensitive

nature of these kind of systems. To add message encryption to our implementation two possible avenues exist. Encryption can either be added on the IEEE 802.15.4 level – the standard specifies symmetric key encryption with a key provided by some upper layer using Advanced Encryption Standard (AES) in Counter with CBC-MAC (CCM*) mode – or by using Datagram Transport Layer Security (DTLS) defined by RFC 6347 on the application level. The SoC we chose could cope with stricter security requirements, because it has hardware support for 128-bit AES in the form of a dedicated co-processor. This co-processor would be well suited to encrypt and decrypt packets on the fly, without any additional burden on the main CPU.

CHAPTER 6

Simulation and Experiments

This was the gist of the notice. It said "The Guide is definitive. Reality is frequently inaccurate."
— Douglas Adams, *"The Restaurant at the End of the Universe"*

Of all the debugging and testing tools at hand, the custom developed simulation environment was the most valuable tool during the implementation of the algorithms. The simulator executes multiple instances of the code that would otherwise run on the resource limited IoT boards on a host PC – encapsulated as separate processes. The simulation environment creates a virtual communication network between the processes that changes over time according to a given graph specification. The simulator allows the user to programmatically step through the progression of rounds, without any connection to the passing of real time. It is further possible to inspect – through instrumentation – the internal and external state of the participating virtualized IoT nodes.

The simulator can compare the output and state of a given implementation for an IoT node with the output and state of an implementation contained in the simulator. The ability to compare the C implementations of the algorithms with implementations using a different technology allowed us to gain further confidence in the correctness of the C code.

6.1 Simulation environment

The simulator was first written to simplify debugging of the root component based algorithm implementation. Programs that heavily interact with their environment are inherently difficult to debug, because of the sometimes fleeting nature of physical reality. It might be possible to halt the CPU and inspect the registers using a debugger, but it is

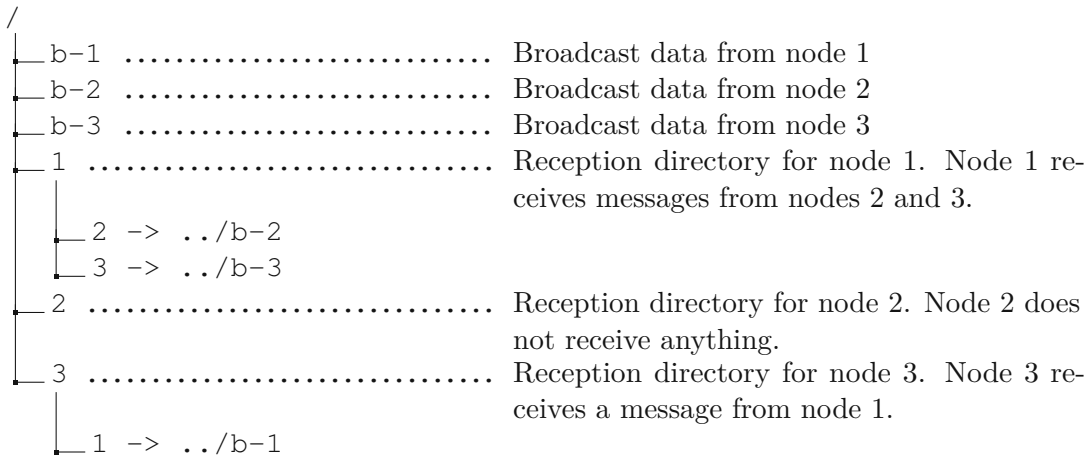


Figure 6.1: Exemplary filesystem layout that provides the broadcast data and describes the communication graph for one round for nodes 1, 2 and 3. Shown are the broadcast files `b-1`, `b-2` and `b-3` as well as the reception directories `1`, `2` and `3` with the symlinks describing the current communication graph.

certainly not possible to precisely control the communication graph or stop time during the transmission of data over the air. Using a simulator allowed us to exert much more fine grained control over the network graph than would ever be possible in the real world.

Multiple options present themselves when implementing a simulation environment. It is entirely possible to provide an environment where only the abstract concepts of the system are simulated, providing full control over all aspects of the simulation. On the other hand, one can try to run as much production code as possible in the simulation environment, thereby giving up some control over the environment and likely loosing performance, but gaining a more realistic basis for simulation. We choose a middle ground by abstracting away the time synchronization and round/phase calculations, but more or less keeping the other code paths.

The simulator takes the C code targeted at the nRF52840 hardware and compiles it for Linux running on ordinary PC hardware. When the C code is compiled for use under the simulator, some code paths are changed by setting simulator specific defines. All code paths concerned with the computational aspects of the algorithms stay the same. The first big difference when running under the simulator is the fact that the round/phase structure is not established by running a time synchronization protocol. The start of a new round or phase is signaled by the simulator to the node processes using POSIX message queues. The second big difference when running the code under the simulator is how messages are exchanged. Networking under the simulator is mapped to operations on files. Each node gets assigned a special file, used to write all the data it wants to broadcast in the current round. The simulator then – according to the given current network graph – creates symlinks from the broadcast files to files in a special reception directory separate for each node. Note that the symlinks are cleared and recreated at

Message adversary	Dynamic network generator
$\mathsf{BOUNDEDDELAY}_{\leq N}$	`BoundedDelay`
$\Diamond\mathsf{WEAKSTAB}_{\leq N}(D+1, D)$	`EventuallyWeakStab`
$\Diamond\mathsf{STABLE}_{\leq N,D}(D+1)$	`EventuallyStable`

Table 6.1: Available dynamic network generators in the simulator and the matching message adversary. All listed Python classes are defined in the module `simulator.message_adversaries`

the start of each round. See Figure 6.1 for a more detailed description of the filesystem networking setup.

Using the filesystem has a number of advantages over using the hosts networking stack. The first advantage is that the filesystem allows for easy inspection by various commandline tools and is suitable for taking quick snapshots for later analysis – especially of broadcast data. The second advantage is that we can avoid all the timing assumptions that the host's networking stack implementation certainly makes and that would be difficult to handle reliably inside the simulator. A third advantage of using the filesystem is the fact, that the 6LoWPAN implementation has an upper limit for the number of bytes it can transport. Since we do not need to go through the 6LoWPAN stack, the simulator is able to simulate a higher number of nodes than would be supported by an unpatched 6LoWPAN implementation. For more details on this topic see Section 6.2.

After using the simulator to debug various issues, the need to automatically check if the algorithm computed correct results arose. We extended the simulator with facilities to programmatically describe the expected state of the algorithm for each node and round. This programmatic description is then used to compare the C code's internal state after every round with the programmatically described state. This approach is useful for testing specific key points, but quickly becomes cumbersome for larger values of N and larger numbers of simulated rounds. Instead of manually describing the expected state, we additionally implemented the stabilizing root component algorithm as well as the MinMax algorithm in Python. Afterwards we extended the simulator with facilities to compare the expected state of the Python implementation with the state of the C implementation when executing both implementations using the same parameters and dynamic network.

There are two possible ways to provide the dynamic network for a simulation run. Using the Python `networkx`[1] package, the user of the simulator can programmatically describe the communication graph for each round. This approach only lends itself to small checks for specific properties and becomes quickly long-winded. A second problem with the manual approach is that the dynamic networks do not include any variation or randomness, making it even more cumbersome to write different scenarios. To solve this problem the simulator was extended with dynamic network generators that are able

[1] https://networkx.org/ A popular Python graph library

to create dynamic networks of a given length and according to a message adversary specification. The dynamic network generators can be used in two different modes: seeded and unseeded mode. When the generator is given a seed it will always create the same dynamic network. In unseeded mode the dynamic network will be different for every execution. See Table 6.1 for a list of dynamic network generators included in the simulator.

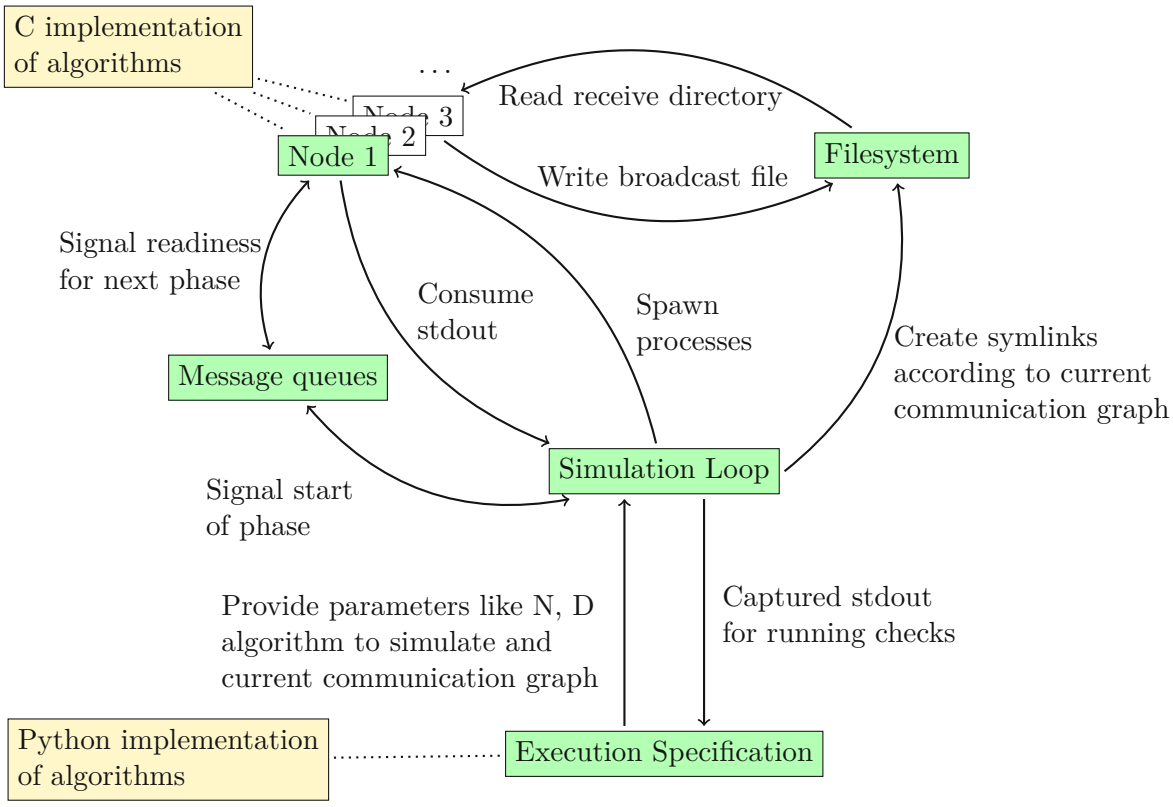


Figure 6.2: The flow of control in the simulator. The execution specification provides information to the simulation loop and in turn gets information about the nodes' internal state to run the automated checks. The simulated nodes and the simulation loop both access the filesystem to facilitate communication between the nodes.

Flow of control

The core of the simulator consists of the simulation loop. The simulator first spawns the specified number of processes, where each process represents a node. Since the simulator does not need to establish a global time base, it disables the FTSP module. Instead of using the global time base to determine the start of the next round, the simulator uses POSIX messages queues to control the round structure of the node processes. The simulator waits for all nodes to report that they are ready to start a new phase and then

issues a command to start the next phase. The nodes listen for instructions from the simulator and start execution of the next phase when the command to do so arrives. During execution of a phase, the simulator consumes the standard output of the node processes. The structured messages on the standard output are a representation of the current internal state of the node and are parsed by the simulator and used for comparison with either the manual or the automatic checks. After a round is complete, all checks scheduled for this round are performed and then the simulation loop advances to the next round. See Figure 6.2 for a description of the flow of control in the simulator.

Execution specification

Writing executions is the main way of interacting with the simulator. Executions are written using the Python programming language and must be included in the `simulator.executions` module. Executions are specific runs of an algorithm, with a given environment and graph sequence. When writing the executions the programmer either specifies checks manually or – using automatic mode – lets the simulator generate the checks using a separate implementation of the algorithm under test.

```
class LateJoinExecution(AutoExecutionMixin, BaseExecution):
    N = 6
    D = N - 1
    algorithm_class = StabilizingConsensus

    @property
    def graphs(self):
        g = nx.DiGraph()
        g.add_edge(1, 2)
        g.add_edge(2, 3)
        g.add_edge(3, 4)
        g.add_edge(4, 1)
        g.add_node(5)
        g.add_node(6)

        # Now nodes 5 and 6 join
        h = nx.DiGraph()
        h.add_edge(1, 2)
        h.add_edge(2, 3)
        h.add_edge(3, 4)
        h.add_edge(4, 5)
        h.add_edge(5, 6)
        h.add_edge(6, 1)

        return [g] * 40 + [h] * 40
```

Listing 6.1: An execution written in Python with an explicitly specified graph sequence and automatic comparison against the simulator provided implementation `StabilizingConsensus` of the root component based stabilizing consensus algorithm.

In manual mode, the developer programmatically describes the expected state at a certain point in time, and the simulator checks that the actual and expected states are equal. The `AutoExecutionMixin` enables automatic check generation depending on the specified algorithm class using the `algorithm_class` attribute.

The `graphs` property must return a list of graphs – one for each round of the execution. Graph sequences can be auto-generated according to a message adversary specification or specified explicitly. See Listing 6.1 for an exemplary execution using an explicit graph sequence. The code shown in Listing 6.2 makes use of the dynamic network generator `BoundedDelay` to generate a dynamic network that follows the constraints given by $\mathsf{BOUNDEDDELAY}_{\leq N}$.

```
class RandomExecution(AutoExecutionMixin, BaseExecution):
    """An execution using random graphs without an initial seed.

    The list of graphs generated will be different for each call.
    """

    N = 16
    algorithm_class = MinMax
    T = 5

    @property
    def graphs(self):
        ma = BoundedDelay(self.T, self.N, 60, edge_p=0.1)
        return list(ma)
```

Listing 6.2: An execution written in Python with a generated graph sequence according to a message adversary specification and automatic comparison against the simulator provided implementation `MinMax` of the MinMax based stabilizing consensus algorithm.

6.2 Experiments

After completing the implementation we decided to run some experiments. The purpose of our experiments was to gather data on the stabilization times of the root component based stabilizing algorithm and the MinMax algorithm. This data would allow us to make predictions on which algorithm stabilizes faster under real world conditions. To gather this data we extended our implementation with the ability to run two different consensus algorithms in parallel. Using this feature we were able to directly compare the real-world performance of the consensus algorithms. It is important to note, that broadcast messages from both algorithms get combined into one message that is either fully received or completely lost. Using this combined message approach guarantees that the communication graph is exactly the same for both algorithm implementations during an experimental run.

We additionally extended both algorithms to write a log of their execution. This data

was later collected and analyzed to determine the respective stabilization times. Since both algorithms solve the stabilizing consensus problem, there is no point in time where the algorithms irrevocably decide on a value. We defined the point of stabilization as the round after which all nodes agree on a common value until the end of the run.

The experiments were performed using 14 nodes. Increasing the number of nodes further would – when running the stabilizing and the MinMax algorithm in parallel – cause our consensus messages to exceed the MTU of 1280 bytes for 6LoWPAN defined by RFC 4944. Future experiments could circumvent this limitation by patching the 6LoWPAN implementation of RIOT to allow messages bigger than 1280 bytes.

The experiments ran over the course of several hours on the third floor (approx. 500 m^2) of an office building in Vienna. The nodes were evenly spread around, but several metal structures like an elevator shaft and other IT equipment prevented direct line of sight communication between the nodes. We performed 8 experimental runs, where each run was 30 min long.

Run No.	Stabilization time – MinMax	Stabilization time – root component based
1	51	35
2	47	37
3	43	33
4	49	30
5	55	30
6	45	32
7	47	18
8	55	39

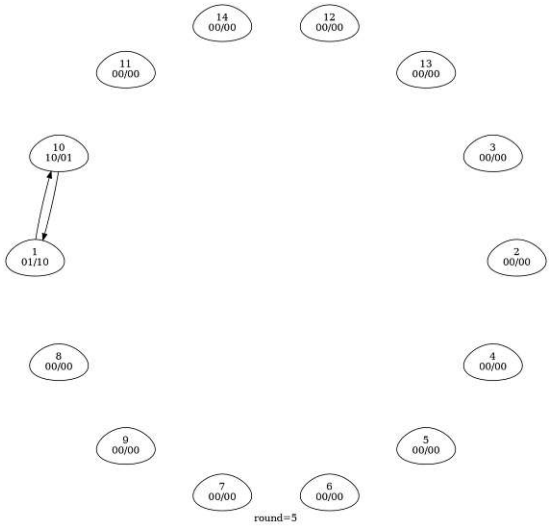
Table 6.2: Stabilization times in rounds for the direct comparison of the MinMax and the root component based stabilizing consensus algorithms.

To be able to run the experiments the minimum phase duration needed to be calculated first. For a message in size shortly below 1280 bytes, 14 nodes and the nominal transmission speed of 250 kbit/s of 6LoWPAN, we arrived at a minimum phase duration of about 2.09 s. Note that this value takes into account the properties that were described in Section 5.3.3.

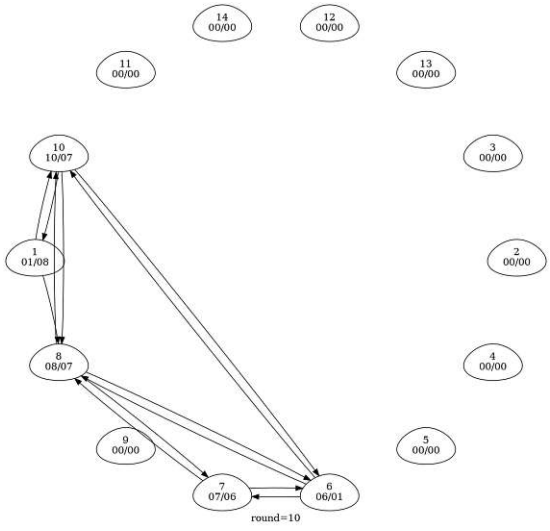
All nodes logged to SD cards that were collected for analysis after all 8 runs finished. The logs from the SD cards allowed us to reconstruct the connection graphs for each round in all the runs. Figure 6.3 shows renderings of some of the communication graphs of run 1. Shown are every fifth round's graph until both algorithms have stabilized. The top values inside the circles are the node IDs, the left bottom values are the output of the root component based algorithm, the right bottom values the output of the MinMax algorithm. Animations of communication graphs for all 8 runs can be found under https://thesis.echtinger-sieghart.at.

See Table 6.2 for the different stabilization times obtained per run. The experimental

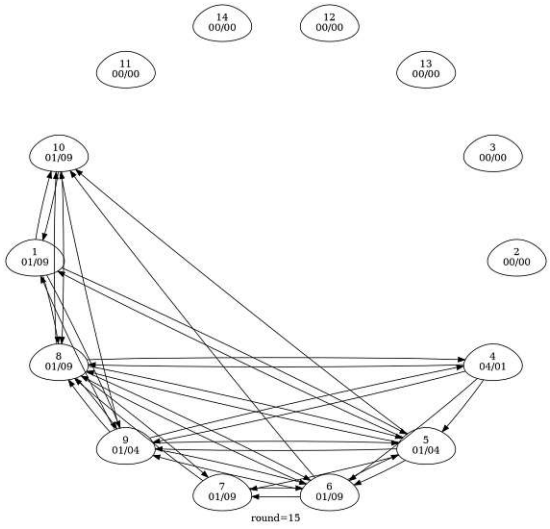
evaluation confirmed our suspicion that the root component based consensus algorithm stabilizes faster than the MinMax consensus algorithm under real-world conditions.



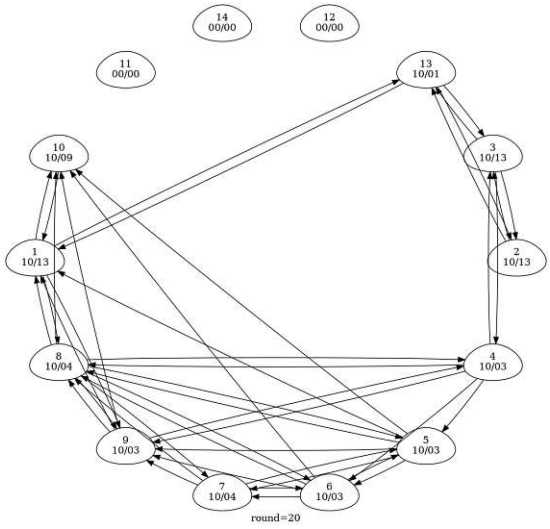
(a) Run 1, communication graph of round 5



(b) Run 1, communication graph of round 10

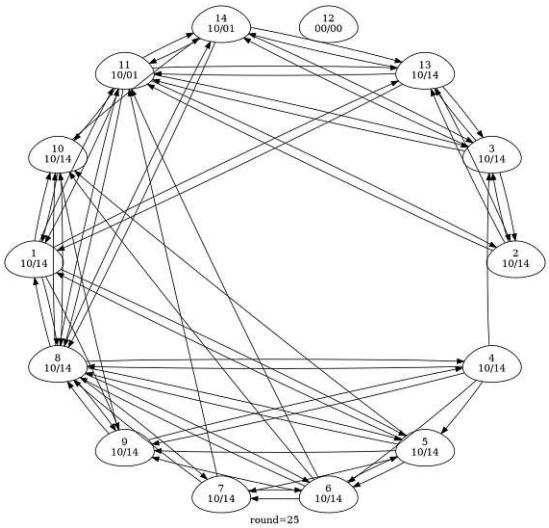


(c) Run 1, communication graph of round 15

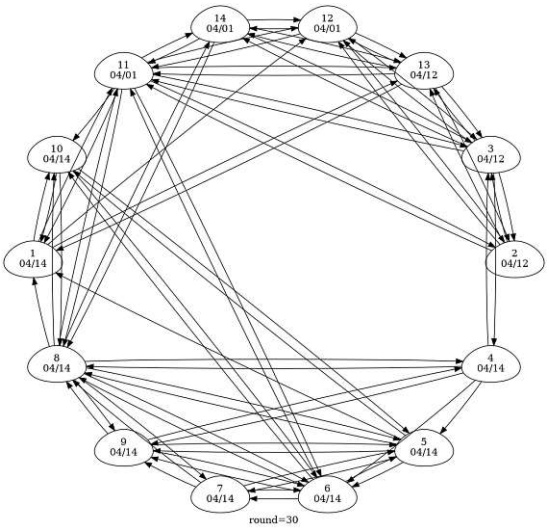


(d) Run 1, communication graph of round 20

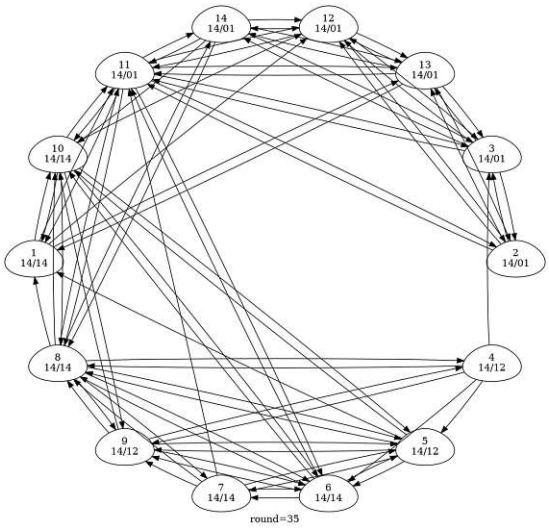
Figure 6.3: Communication graphs for run 1 for every fifth round until stabilization of both algorithms. The top values are the node IDs. The bottom left values are the output of the root component based algorithm, the bottom right values the output of the MinMax algorithm. (1/3)



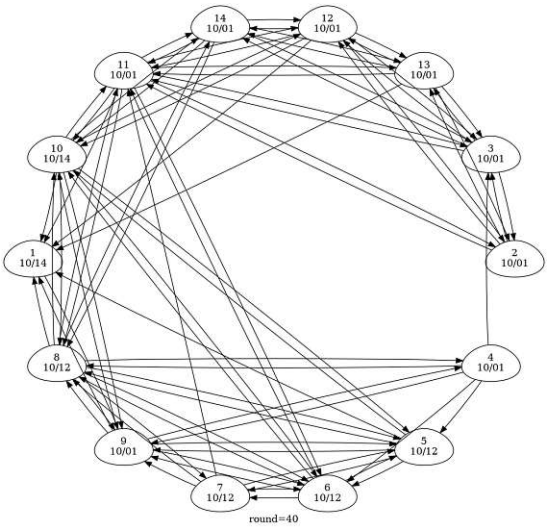
(e) Run 1, communication graph of round 25



(f) Run 1, communication graph of round 30

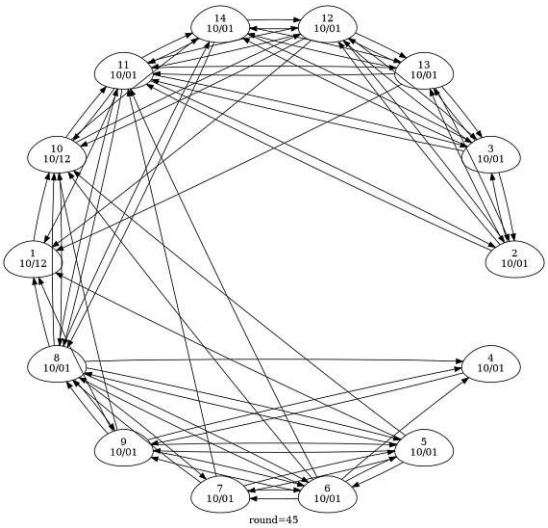


(g) Run 1, communication graph of round 35

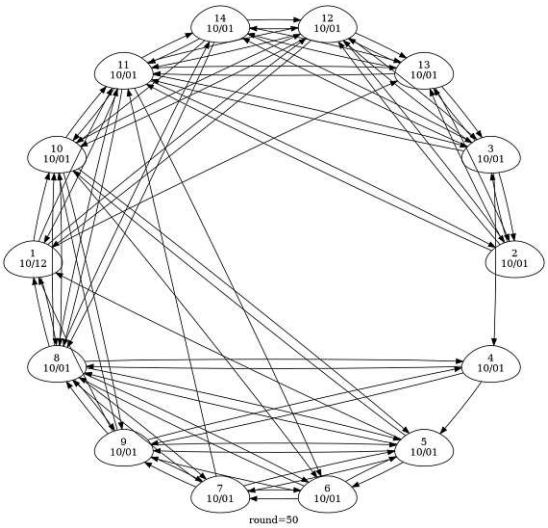


(h) Run 1, communication graph of round 40

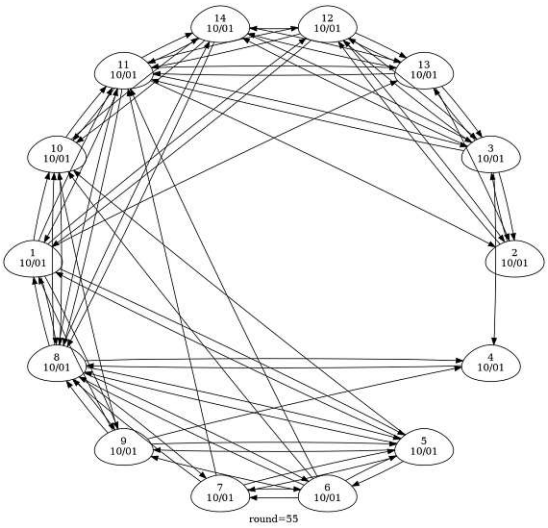
Figure 6.3: Communication graphs for run 1 for every fifth round until stabilization of both algorithms. The top values are the node IDs. The bottom left values are the output of the root component based algorithm, the bottom right values the output of the MinMax algorithm. (2/3)



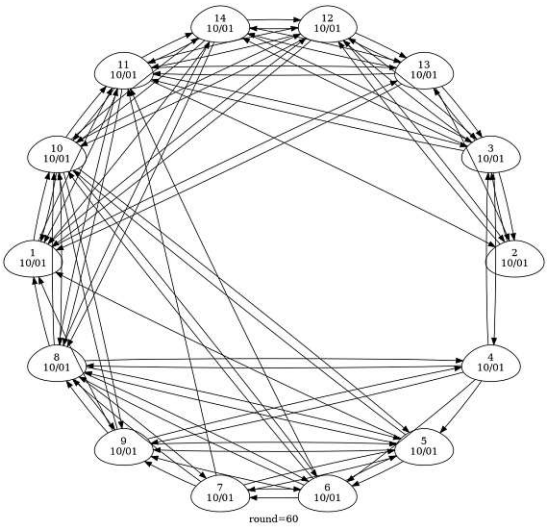
(i) Run 1, communication graph of round 45



(j) Run 1, communication graph of round 50



(k) Run 1, communication graph of round 55



(l) Run 1, communication graph of round 60

Figure 6.3: Communication graphs for run 1 for every fifth round until stabilization of both algorithms. The top values are the node IDs. The bottom left values are the output of the root component based algorithm, the bottom right values the output of the MinMax algorithm. (3/3)

CHAPTER 7

Conclusion

"Not even time to finish my cake?"

NO. THERE IS NO MORE TIME, EVEN FOR CAKE. FOR YOU, THE CAKE IS OVER. YOU HAVE REACHED THE END OF CAKE.
— Terry Pratchett *"Night Watch"*

We conclude with a short recap of this thesis, highlighting the major parts. We started with a literature overview that put its focus on consensus algorithms in the context of WSNs, time synchronization algorithms and respective implementation techniques. We also introduced the terminating and the stabilizing consensus problem. Chapter 2 talked about the system model and the core concepts necessary to understand a class of root component based consensus algorithms introduced by [SS21] and [WSS19].

Chapter 3 presented, in detail, the aforementioned root component based algorithms. After discussing the root component based approach to solving consensus, we shifted our focus to a completely different approach: The MinMax algorithm introduced by [CM21] and presented in Chapter 4. We explained the system model and the core concepts behind the MinMax based approach to stabilizing consensus. We further put some concepts of the root component based system model into relation with concepts from the MinMax system model and showed that the message adversaries behind the root component based algorithm and the MinMax algorithm are not comparable.

After the in-depth introduction to the theoretical underpinnings, we presented in Chapter 5 the steps we took to arrive at a working implementation. We provided practical and theoretical insights regarding an implementation of FTSP, showed a way to derive the required round structure from the global time base and presented data structures and implementation techniques suitable to implementing the root component based stabilizing algorithm and the MinMax algorithm.

Last but not least, Chapter 6 introduced the simulation environment that was developed during the implementation of the consensus algorithms and gave a short tutorial on how to interact with the simulator. The thesis is rounded off by a description of a series of experiments we undertook regarding a comparison of the stabilization times of the root component based stabilizing consensus algorithm and the MinMax consensus algorithm. As can be seen from looking at the source code, the implementation of the root component based algorithms is much more involved than the implementation of the MinMax based algorithm. Our experimental results hint at the fact that the higher complexity in implementation pays off in terms of a faster stabilization time.

APPENDIX A

Getting started

A learning experience is one of those things that says,
"You know that thing you just did? Don't do that."
— Douglas Adams *"The Salmon of Doubt"*

This appendix serves as a small guide to the practical side of the implementation. We will explain the directory layout and go into detail regarding usage of the build system and the simulation environment. We conclude the appendix with a list of common tasks and corresponding commands.

A.1 Source code and directory layout

The source code for the implementation can be found in the git repository `https://git.sr.ht/~chrstph/consensus`. Figure A.1 gives an overview of the directory layout of the project – note that only the most important directories and files are described.

The main part of the implementation was written in C99 with GNU C Extensions. The GNU C Extensions are used to define variable length arrays in structs – see the comments in `consensus/ddn/include/ddn.h` for more details. The simulator was implemented in Python 3.8 and most supporting scripts are either plain bash or also Python 3.8.

All commands given in this appendix must be executed in the directory `consensus` and all exemplary command invocations are given relative to this directory.

```
/
├── RIOT/ ..................... RIOT OS source tree with custom changes
└── consensus/
    ├── consensus/
    │   └── consensus.c ....... Code shared by the three consensus implementations
    ├── consensus_terminating/
    │   └── terminating.c ..... Root component based terminating consensus implementation
    ├── consensus_stabilizing/
    │   └── stabilizing.c ..... Root component based stabilizing consensus implementation
    ├── consensus_minmax/
    │   ├── minmax.c .......... MinMax based stabilizing consensus implementation
    │   └── age.c ............. Data structures and helper functions for the MinMax algorithm
    ├── ddn/
    │   ├── algorithm.c ....... Tarjan's algorithm and various related functions
    │   └── ddn.c ............. DDN implementation and helper functions used by the root component based consensus implementations
    ├── docker/ ............... Definitions of the various docker containers
    ├── ftsp/
    │   └── ftsp.c ............ FTSP implementation
    ├── main.c ................ The main entry point
    ├── Makefile .............. The main Makefile
    ├── node_id/
    │   └── node_id.c ......... Maps physical SoC IDs to node IDs
    ├── parameters/
    │   └── parameters.c ...... Allows setting the parameters D and N
    ├── riotmake.sh ........... Dockerized build/flash/debug
    ├── scripts/ .............. Various helper scripts
    ├── simulator/ ............ Implementation of the simulator
    └── simulator.sh .......... Start script for the simulator
```

Figure A.1: Directory layout of the project

A.2 Build system

A.2.1 Node IDs

To be able to use your board with the build system, the serial number of the board's programmer needs to be configured in `consensus/scripts/lib.sh`. Our build system currently has support for nRF52840-MDK and nRF52840 DK boards, but it is easy to add support for other boards as long as RIOT has support for the board[1]. If you are running a recent Linux distribution with `systemd-udev`, the easiest way to get the serial number of your board's programmer is to check the directory `/dev/serial/by-id` after plugging the board in.

In addition to adding the programmer's serial to `consensus/scripts/lib.sh`, the SoC's internal ID needs to be added to the array `device_ids[]` defined in `consensus/node_id/node_id.c`. To obtain the internal ID of your SoC, you may use the function `node_id_print_device_id()` defined in `consensus/node_id/node_id.c`. The mapping of SoC ID to array index is used to provide the boards with the necessary unique node IDs. This allows us to later address a specific board when running `consensus/riotmake.sh` in combination with the `CNID` environment variable.

A.2.2 Multiple algorithms

Since the implementation supports the parallel execution of two different consensus algorithms we need a way to control which algorithms should be active. The environment variables `CONSENSUS_ALGO_A` and `CONSENSUS_ALGO_B` can be set to `stabilizing`, `terminating`, `minmax` and `dummy`. These variables specify which consensus algorithms will be built and run on the board. The special value `dummy` is there to disable an algorithm slot.

A.2.3 Docker

The whole project is configured to be built inside a Docker[2] container. This makes it possible to get started without installing any dependencies with the exception of Docker. The starting point for all interactions with the build system is the `consensus/riotmake.sh` script. To build all the required Docker images use `consensus/scripts/build-images.sh`.

If the use of Docker is not an option, the project can be built without Docker by directly calling `make`. When using the plain `make` command, make sure to set the `BOARD` and `CNID` environment variables accordingly.

[1] https://doc.riot-os.org/group__boards.html
[2] https://www.docker.com

A.3 Simulator

The simulator is also run inside a Docker container. See consensus/simulator.sh for more details. The implementation of the simulator is contained in in the directory consensus/simulator. The C define SIMULATOR_PRESENT is used in the source code to switch between sections of code that only make sense in the simulator or that only make sense when running on the actual board. The main differences between running in the simulator and running on real hardware, is the way that the round/phase structure is enforced and the way messages are sent and received. On real hardware FTSP is used, whereas inside the simulator the main process uses POSIX Message Queues to signal start and completion of new rounds and phases. In addition to that, dumping the internal state is disabled when running on real hardware. The simulator expects the binaries to be already compiled with the correct flags – the script consensus/simulator.sh takes care of that.

A.4 Common tasks

A.4.1 Basics

List all available Makefile targets.

```
./riotmake.sh help
```

Compile the source code for the board with node ID 2.

```
CNID=2 ./riotmake.sh
```

Flash the board identified by node ID 3 with the root component based stabilizing algorithm in algorithm slot A and the MinMax algorithm in algorithm slot B.

```
CNID=3 CONSENSUS_ALGO_A=stabilizing CONSENSUS_ALGO_B=minmax \
    ./riotmake.sh flash
```

Connect a debugger to the board identified by node ID 1.

```
./riotmake.sh debug
```

Connect a terminal to the serial output of the board identified by node ID 4.

```
CNID=4 ./riotmake.sh term
```

A.4.2 Tests

Run all tests included with the project. Note, that to run the C tests, a board with node ID 1 needs to be connected to the system, because some tests are run directly on the target hardware.

```
./scripts/check.sh
```

To run only a specific set of tests use one of the following commands:

```
./scripts/run-simulator-tests.sh
./scripts/run-c-tests.sh
./scripts/run-code-quality.sh
```

Run only a subset of the C tests.

```
./riotmake.sh -C ftsp/tests/ftsp_table_ops flash test
```

Run only a subset of the simulator tests.

```
./scripts/run-simulator-tests.sh -k SOME_FUNC_OR_CLASS
```

A.4.3 Running the simulator

Run all simulator executions for the terminating consensus algorithm.

```
./simulator.sh terminating
```

Run all simulator executions for the stabilizing consensus algorithm, whose names contains the string “CompleteCycle”.

```
./simulator.sh stabilizing -k CompleteCycle
```

Continually run random executions for the stabilizing consensus algorithm.

```
while ./simulator.sh stabilizing -k RandomExecution; do :; done
```

A.4.4 Various

Enter the docker container used by the build system.

```
./scripts/riotshell.sh
```

Build or rebuild all docker images.

```
./scripts/build-images.sh
```

Setup the ATUSB 802.15.4 dongle to sniff IEEE 802.15.4.

```
./scripts/sniff.sh
```

List of Figures

List of Algorithms

Glossary

Contiki-NG An IoT operating system. 41

EasyDMA A Nordic Semiconductor DMA module. 40

emb6 A networking stack. 41

IEEE 802.15.4 A low-level communication protocol. 5, 39–46, 51, 52, 61, 62, 64, 65, 84, 85

lwIP A networking stack. 41

nRF52840 An SoC manufactured by Nordic Semiconductors. 38–40, 50, 68

OpenThread A networking stack. 41

RIOT An IoT operating system. 39, 41, 45, 51, 73, 81

TinyOS An IoT operating system. 4, 5, 41, 51

TOSSIM A simulator for TinyOS wireless sensor networks. 5

ZigBee A network level protocol defined to work over IEEE 802.15.4. 4, 6, 42

Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Network. 39, 41, 42, 45–47, 50, 64, 69, 73

AES Advanced Encryption Standard. 65

AODV Ad-hoc On-demand Distance Vector. 4

ATS Average TimeSync. 6

BLE Bluetooth Low Energy. 39, 41

CAN Controller Area Network. 4

CAP Contention Access Period. 45

CCA Clear Channel Assessment. 43

CCM* Counter with CBC-MAC. 65

CFP Contention Free Period. 45

CRC Cyclic Redundancy Check. 40

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance. 43, 45, 61

DAPLink Debug-Access-Port Link. 39

DDN Directed Dynamic Network. 11, 13, 15, 63, 80

DMA Direct Memory Access. 40, 50–52

DSSS Direct Sequence Spread Spectrum. 43

DTLS Datagram Transport Layer Security. 65

ED Energy Detection. 43

FCS Frame Check Sequence. 43

FCSA Flooding with Clock Speed Agreement. 6

FPU Floating Point Unit. 39

FTSP Flooding Time Synchronization Protocol. 5, 6, 47–53, 57, 58, 64, 70, 77, 80, 82, 85, 86

GNRC GeNeRiC network stack. 41, 45

GPS Global Positioning System. 47

GTS Guaranteed Time Slot. 45, 64

IoT Internet of Things. xi, xiii, 1, 41, 47, 48, 64, 67

IPC Inter-process-communication. 41

IPv6 Internet Protocol Version 6. 45, 46

ISR Interrupt Service Routine. 51

MAC Media Access Control. 4, 6, 40, 42, 43, 48, 50, 61

MTS Maximum Time Synchronization. 6

MTU Maximum Transmission Unit. 45, 73

NALP Not A LoWPAN Frame. 47, 50

NIC Network Interface Card. 48

NTP Network Time Protocol. 5

O-QPSK Offset Quadrature Phase-Shift Keying. 42

OS Operating System. 38, 39, 41

PAN Personal Area Network. 42, 43, 64

PHY Physical. 40, 42, 43

PPDU PHY Protocol Data Unit. 43

PSDU PHY Service Data Unit. 43

RBS Reference-Broadcast Synchronization. 5

RF Radio Frequency. 42, 43

SLAAC Stateless Address Autoconfiguration. 46

SoC System-on-Chip. 38–41, 48, 65, 80, 81

TPSN Timing-sync Protocol for Sensor Networks. 6

TSCH Time Slotted Channel Hopping. 42

UDP User Datagram Protocol. 46

VISA Virtual Instrument Software Architecture. 52

WMTS Weighted Maximum Time Synchronization. 6

WSN Wireless Sensor Network. 1, 2, 4–6, 47, 77

Bibliography

[Ada06] J. T. Adams. An introduction to IEEE STD 802.15.4. In *2006 IEEE Aerospace Conference*, pages 8 pp.–, 2006.

[AEF14] Abderrazak Abdaoui and Tarek M. El-Fouly. Tossim and distributed binary consensus algorithm in wireless sensor networks. *Journal of Network and Computer Applications*, 41:451–458, 2014.

[AFJ06] Dana Angluin, Michael J. Fischer, and Hong Jiang. Stabilizing consensus in mobile networks. In Phillip B. Gibbons, Tarek F. Abdelzaher, James Aspnes, and Ramesh R. Rao, editors, *Distributed Computing in Sensor Systems, Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006, Proceedings*, volume 4026 of *Lecture Notes in Computer Science*, pages 37–50. Springer, 2006.

[AG13] Yehuda Afek and Eli Gafni. Asynchrony from synchrony. In Davide Frey, Michel Raynal, Saswati Sarkar, Rudrapatna K. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking, 14th International Conference, ICDCN 2013, Mumbai, India, January 3-6, 2013. Proceedings*, volume 7730 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2013.

[ANRE15] Noor Al-Nakhala, Ryan Riley, and Tarek Elfouly. Distributed algorithms in wireless sensor networks: An approach for applying binary consensus in a real testbed. *Computer Networks*, 79:30–38, 2015.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[BCN+16] Luca Becchetti, Andrea Clementi, Emanuele Natale, Francesco Pasquale, and Luca Trevisan. Stabilizing consensus with many opinions. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 620–635. SIAM, 2016.

[BGH+18] Emmanuel Baccelli, Cenk Gündoan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and

Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018.

[BRS12] Martin Biely, Peter Robinson, and Ulrich Schmid. Agreement in directed dynamic networks. In Guy Even and Magnús M. Halldórsson, editors, *Structural Information and Communication Complexity - 19th International Colloquium, SIROCCO 2012, Reykjavik, Iceland, June 30-July 2, 2012, Revised Selected Papers*, volume 7355 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2012.

[BRS+18] Martin Biely, Peter Robinson, Ulrich Schmid, Manfred Schwarz, and Kyrill Winkler. Gracefully degrading consensus and k-set agreement in directed dynamic networks. *Theoretical Computer Science*, 726:41–77, 2018.

[CBFN16] Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Fast, robust, quantizable approximate consensus. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 137:1–137:14, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[CCA10] Emanuele Cipollone, Francesca Cuomo, and Anna Abbagnale. A distributed procedure for IEEE 802.15. 4 PAN coordinator election in emergency scenarios. In *The Internet of Things*, pages 39–48. Springer, 2010.

[CFN15] Bernadette Charron-Bost, Matthias Függer, and Thomas Nowak. Approximate consensus in highly dynamic networks: The role of averaging algorithms. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2015.

[CJM05] D. Cox, E. Jovanov, and A. Milenkovic. Time synchronization for zigbee networks. In *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory, 2005. SSST '05.*, pages 135–138, 2005.

[CM21] Bernadette Charron-Bost and Shlomo Moran. Minmax algorithms for stabilizing consensus. *Distributed Comput.*, 34(3):195–206, 2021.

[DB16] Djamel Djenouri and Miloud Bagaa. Synchronization protocols and implementation issues in wireless sensor networks: A review. *IEEE Systems Journal*, 10(2):617–627, 2016.

[DGM+11] Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing consensus with the power of two choices. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 149–158, 2011.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[ER03] Jeremy Elson and Kay Römer. Wireless sensor networks: A new regime for time synchronization. *ACM SIGCOMM Computer Communication Review*, 33(1):149–154, 2003.

[Fai07] Ya R Faizulkhakov. Time synchronization methods for wireless sensor networks: A survey. *Programming and Computer Software*, 33(4):214–226, 2007.

[Fis83] Michael J Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *International conference on fundamentals of computation theory*, pages 127–140. Springer, 1983.

[FN18] Matthias Függer and Thomas Nowak. Fast multidimensional asymptotic and approximate consensus. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, volume 121 of *LIPIcs*, pages 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[FNS18] Matthias Függer, Thomas Nowak, and Manfred Schwarz. Tight bounds for asymptotic and approximate consensus. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 325–334, 2018.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 138–149, 2003.

[HCS+14] Jianping He, Peng Cheng, Ling Shi, Jiming Chen, and Youxian Sun. Time synchronization in wsns: A maximum-value-based consensus approach. *IEEE Transactions on Automatic Control*, 59(3):660–675, 2014.

[IEE06] IEEE Standard for Information technology – Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pages 1–320, 2006.

[IEE17] Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID). *pdf, Tech. Rep., August 2017.* https://standards.ieee.org/content/dam/

ieee-standards/standards/web/documents/tutorials/eui.pdf, 2017.

[IEE20] IEEE Standard for Information technology – Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (WPANs). *IEEE Std 802.15.4-2020 (Revision of IEEE Std 802.15.4-2015)*, pages 1–800, 2020.

[KKRF11] Jozef Kenyeres, Martin Kenyeres, Markus Rupp, and Peter Farkas. WSN implementation of the average consensus algorithm. In *17th European Wireless 2011-Sustainable Wireless Technologies*, pages 1–8. VDE, 2011.

[KO11] Fabian Kuhn and Rotem Oshman. Dynamic networks: Models and algorithms. *SIGACT News*, 42(1):82–96, March 2011.

[Kö13] Andreas Köpke. *Engineering a communication protocol stack to support consensus in sensor networks*. PhD thesis, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, 2013.

[LB03] G.M.A. Lima and A. Burns. A consensus protocol for can-based systems. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 420–429, 2003.

[Lin15] Christopher Lindberg. *Consensus Trade-offs in Wireless Sensor Networks*. PhD thesis, Chalmers Tekniska Hogskola (Sweden), 2015.

[LL84] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and control*, 62(2-3):190–204, 1984.

[LSW15] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Pulsesync: An efficient and scalable clock synchronization protocol. *IEEE/ACM Transactions on Networking*, 23(3):717–727, 2015.

[LZKS13] Heath J. LeBlanc, Haotian Zhang, Xenofon Koutsoukos, and Shreyas Sundaram. Resilient asymptotic consensus in robust networks. *IEEE Journal on Selected Areas in Communications*, 31(4):766–781, 2013.

[Man13] Sabato Manfredi. Design of a multi-hop dynamic consensus algorithm over wireless sensor networks. *Control Engineering Practice*, 21(4):381–394, 2013.

[MET08] Razvan Musaloiu-E and Andreas Terzis. Minimising the effect of wifi interference in 802.15. 4 wireless sensor networks. *International Journal of Sensor Networks*, 3(1):43–54, 2008.

[MKSL04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, 2004.

[Now15] Thomas Nowak. Asymptotic consensus without self-confidence. In *2015 54th IEEE Conference on Decision and Control (CDC)*, pages 4133–4138, 2015.

[OSFM07] Reza Olfati-Saber, J Alex Fax, and Richard M Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, 2007.

[PS16] Daniel Pfleger and Ulrich Schmid. A framework for connectivity monitoring in wireless sensor networks. In *Proceedings 10th International Conference on Sensor Technlogies and Applications (SENSORCOMM 2016)*, pages 40–48, 2016.

[SG07] Luca Schenato and Giovanni Gamba. A distributed consensus protocol for clock synchronization in wireless sensor network. In *2007 46th IEEE Conference on Decision and Control*, pages 2289–2294, 2007.

[SM03] Reza Olfati Saber and Richard M Murray. Consensus protocols for networks of dynamic agents. In *Proceedings of the 2003 American Control Conference, 2003.*, volume 2, pages 951–956. IEEE, 2003.

[SS21] Ulrich Schmid and Manfred Schwarz. Invited Paper: Round-Oblivious Consensus in Dynamic Networks under Stabilizing Message Adversaries. *To appear in proceedings SSS'21*, 2021.

[Sup18] Gaguk Suprianto. Implementation of distributed consensus algorithms for wireless sensor network using nodemcu esp8266. In *2018 Electrical Power, Electronics, Communications, Controls and Informatics Seminar (EECCIS)*, pages 192–196, 2018.

[SW14] M. A. Sarvghadi and T. Wan. Overview of time synchronization protocols in wireless sensor networks. In *2014 2nd International Conference on Electronic Design (ICED)*, pages 204–209, 2014.

[SWS16] Manfred Schwarz, Kyrill Winkler, and Ulrich Schmid. Fast consensus under eventually stabilizing message adversaries. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, pages 1–10, 2016.

[WS19] Kyrill Winkler and Ulrich Schmid. An overview of recent results for consensus in directed dynamic networks. *Bulletin of EATCS*, 2(128), 2019.

[WSS19] Kyrill Winkler, Manfred Schwarz, and Ulrich Schmid. Consensus in rooted dynamic networks with short-lived stability. *Distributed Computing*, 32(5):443–458, 2019.

[YK14] Kasim Sinan Yildirim and Aylin Kantarci. Time synchronization based on slow-flooding in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):244–253, 2014.