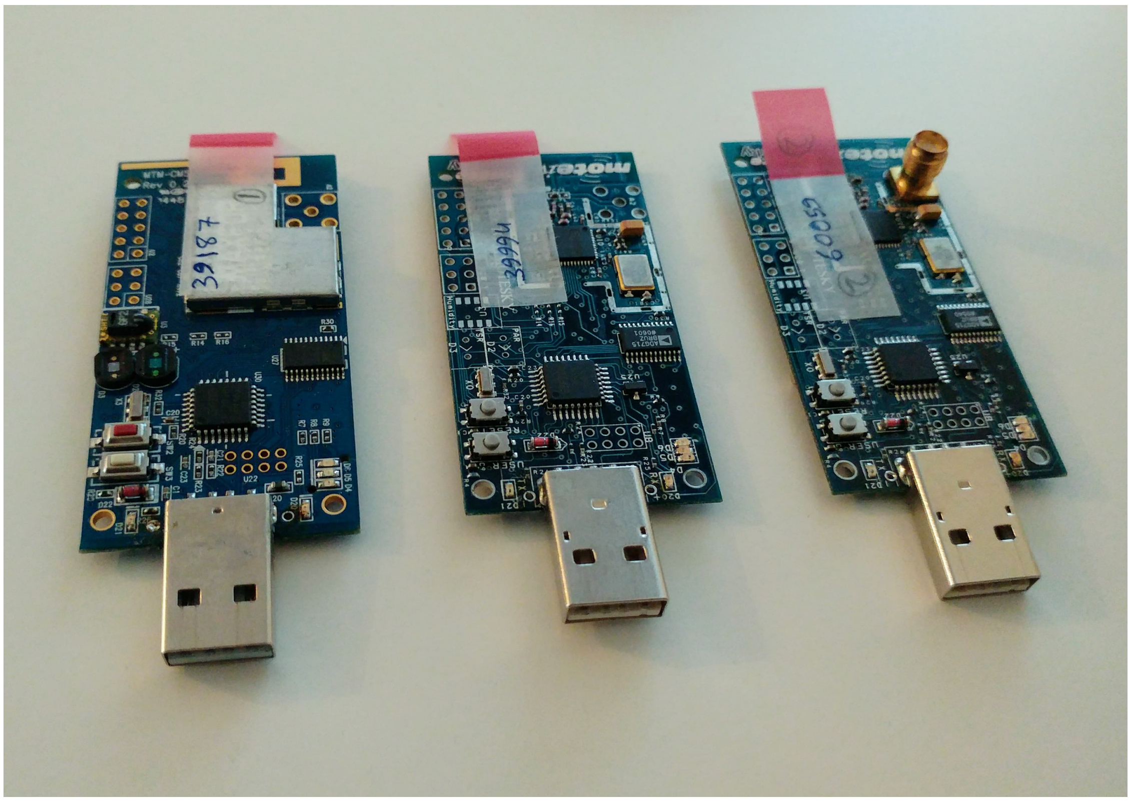


Ultra-Low-Latency Wireless Networking for Mission-Critical Applications

Implementation of a Quorum Protocol on Chaos

Master's thesis in Computer Systems and Networks

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

MASTER'S THESIS 2018

Ultra-Low-Latency Wireless Networking for Mission-Critical Applications

Implementation of a Quorum Protocol on Chaos

BEENISH SHAUKAT

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

Ultra-Low-Latency Wireless Networking for Mission-Critical Applications
Implementation of a Quorum Protocol on Chaos
BEENISH SHAUKAT

Supervisor: Olaf Landsiedel, Computer Science and Engineering
Examiner: Magnus Almgren, Computer Science and Engineering

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

Typeset in $\LaTeX$
Gothenburg, Sweden 2018

Implementation of a quorum protocol on Chaos
BEENISH SHAUKAT
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In a Wireless-sensor network (WSN), low-power sensor devices are connected wirelessly to perform distributed tasks. Such type of networks require low-power wireless networking (LPWN). LPWN provides efficient all-to-all communication and the nodes transmit and receive periodically. Cyber-physical systems (CPSs) are another example of the systems that need LPWN. CPSs also consist of small devices that are connected together to sense the environmental changes, such as heat sensors, motion sensors and light sensors. Both WSN and CPS can also be examples of mission-critical systems. Mission-critical systems require low latency and high reliability.

Usually in a WSN and CPS, the devices have a limited source of energy and they rely on small batteries. As the devices are wirelessly connected, they consume most of their energy in radio-transmissions. Traditional LPWN provides centralized processing, and enables all-to-one and one-to-all communications. In such communications, data is first collected at the sink, the sink processes the data and sends the results to all nodes. Such type of communication schemes consume a lot of energy in sending data towards the sink and receiving back from the sink. Chaos is a new LPWN scheme that allows distributed processing of the data. Chaos provides in-network processing, efficient all-to-all communications and a network-wide agreement using Two-Phase Commit (2PC) and Three-Phase Commit (3PC).

In this thesis, we design and implement a quorum protocol on Chaos. The protocol enables the Single-Writer/Multiple-Reader (SWMR) and Multiple-Writer/Multiple-Reader (MWMR) protocols using a majority quorum configuration. We implement the protocols on the Contiki operating system using Chaos communication primitives. The SWMR/MWMR protocols act as emulators of the shared-memory systems, and provide highly reliable wireless communications with low-latency. The protocols also significantly improve the energy-consumption in LPWN.

Keywords: WSN, CPS, Contiki, LPWN, SWMR, MWMR, Chaos.

Acknowledgements

I, the author of this thesis, would like to take this opportunity to thank the supervisor of this thesis, Olaf Landsiedel at Chalmers University of Technology, for his help and support during this thesis. Also a great thanks to the examiner of this thesis, Magnus Almgren at Chalmers University of Technology, for providing a valuable feedback during this thesis.

Beenish Shaukat, Gothenburg, August 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Low-power wireless networking (LPWN) provides efficient wireless communication and keeps the energy cost low. LPWN is required when devices have a limited energy source. The two most popular examples of LPWN are Wireless-Sensor Networks (WSNs) and Cyber-Physical Systems (CPSs). A WSN can be defined as a network of sensing devices (nodes) that monitor environmental conditions such as temperature, pressure and vibration. A CPS also consists of sensing devices that are used to control and get feedback from the physical environment. Usually, the sensing devices in WSNs and CPSs rely on small batteries and energy efficiency is a big challenge. These devices are equipped with a low-power processor, on-board memory and a radio transceiver. In such networks, most of the energy is consumed by radio transmissions. Therefore, it is essential to minimize the radio transmissions. Both CPS and WSN could be examples of mission critical systems. Many mission critical systems require a reliable wireless communication with low-latency and high availability. However, it is critical to reduce latency and provide efficient wireless communication with high reliability.

1.1 Drawbacks of Traditional Networking

Many low-power wireless applications require data sharing and processing. For example, industrial control systems compute the control-law in a fully distributed fashion, where all sensor nodes share their readings [4]. A typical approach for data sharing in low-power wireless applications is all-to-all communication. A traditional all-to-all communication scheme follows three steps: data collection and aggregation, centralized processing, and dissemination [5, 6, 7]. In the centralized processing scheme, data is collected and aggregated towards a central entity called sink. The sink collects data from all nodes, processes data and disseminates results in the network. A major drawback of this scheme is the presence of single-point of failure; the system fails when the sink fails. Other issues in centralized processing are high power consumption and latency; sending data back and forth towards the sink consumes energy and time.

1.2 Chaos: A new Approach

Chaos [1] is a new protocol stack that overcomes the drawbacks of traditional networking. Chaos provides all-to-all data sharing and enables in-network computations [1]. Compared to other approaches, Chaos provides the ability to perform data aggregation, processing, and dissemination, in parallel. Hence, Chaos improves the energy-efficiency and reduces the latency.

In Chaos, nodes synchronously transmit when they want to share data [1]. Upon receiving, nodes merge received data with their own data by applying a user-defined merge operator. Nodes in the network are distinguished by unique identifiers (IDs). The very first node in a group is named as *initiator* and provides the information for synchronization. In a group, nodes send/receive packets to communicate with each other. Usually, a packet contains a set of flags equivalent to the number of nodes.

Chaos provides several network primitives, where the voting primitive is used to reach an agreement in a group. When a node agrees on a value, it votes for the value by setting its corresponding flag bit to 1. Each time a node receive a packet, it merges the flags and transmits if it learns something new from the last received packet.

In contrast to Glossy [8] and LWB [5], nodes in Chaos synchronously send different data. The nodes transmit when they learn something new from the packet received. When the nodes receive all votes on a value, they repeatedly re-transmit their data and sleep until the next round.

1.3 Thesis Contribution

This thesis work makes the following contributions:

- Design, implementation and evaluation of the SWMR (Single-Writer/Multiple-Reader) register [9], using the majority quorum configurations. The protocol allows only one writer and handles the concurrent read-write conflicts [10].
- Design, implementation and evaluation of the MWMR (Multiple-Writer/Multiple-Reader) register [11], using the majority quorum configurations. The protocol allows multiple writers and handles the read-write and write-write conflicts.

1.4 Thesis Organization

We discuss the LPWN, traditional networking and Chaos in Chapter 1. Chapter 2 provides the background material that helps to fully understand the thesis topic. We present the work relevant to this thesis in Chapter 3. Chapter 4 provides the design choices for SWMR and MWMR protocols. The implementation details of the SWMR and MWMR protocols are presented in Chapter 5. Chapter 6 presents how we evaluate the performance of our SWMR and MWMR algorithms. Discussion on the results is presented in Chapter 7.

Chapter 2

Background

In this chapter, we discuss Contki OS, where we implement our protocols. This chapter provides more details on the Chaos. We also discuss the communication models and the quorum protocol that we implement on Chaos.

2.1 Chaos Communication Primitive

Chaos [1] is the first primitive that provides all-to-all data sharing in LPWN and supports in-network computations. Chaos is built on two fundamental mechanisms: synchronous transmissions and user-defined-merge operator. Chaos propagation policy says, "spread the unknown and suppress the known". The nodes in the network transmit data when they want to share data. Upon receiving, nodes merge the received data with their own data and transmit the merged data in the next slot. The nodes receive the data with high probability, due to the capture effect [2]. The nodes synchronously share the data until all nodes in the network have the same data. The user-defined-merge operator allows users to program the merge operator freely. The nodes only transmit when they learn something new from the last received packet. The goals are to make the communications efficient, increase the reliability and minimize the numbers of synchronous transmissions.

Figure 2.1 shows an example of a *max* round on Chaos. The network consists of three nodes: A, B and C. Node A, B and C can directly communicate with each other, as shown in Figure 2.1 (a). Each node proposes a value and the goal is to determine the maximum value among the nodes. Therefore each node prepares a packet, where the packet consists of a set of progress flags and payload, as shown in Figure 2.1 (b). A specific node (node A in this example), called initiator, starts the transmissions. Node A inserts its value to the packet, sets its flag-bit and transmits the packet. Both B and C receive the packet in Slot 1 and they determine the maximum value using the $max(a, b)$ operator and transmit the merged packets in Slot 2. Due to the capture effect, node A receives B's packet with high probability. Node A merges the flags, updates its value and transmits in Slot 3. Node C learns something new and transmits in Slot 4. Node B does not learn anything new in Slot 3. Hence, node B listens in Slot 4. In Slot 4, both A and B receive the packet form C. In Slot 4, all nodes receive the maximum value and the Chaos round ends.

Nodes may have different processing time due to different conditional statements in the merge operator and the payload size. The time after receiving a packet till the time the micro-controller unit (MCU) starts to serve the interrupt is also variable. However, despite of having a variable processing time and interrupt delay, Chaos ensures the synchronous transmissions by letting the nodes execute the same number of MCU clock cycles. Chaos introduces a wait loop to compensate the variable interrupt delay and processing time, as shown in the Figure 2.2.

Chaos minimizes the radio usage to maximize the energy efficiency. When the nodes complete their tasks, they turn off their radio and sleep until the next round. To make sure everyone is on one page, the nodes aggressively share their final results before they switch to sleep.

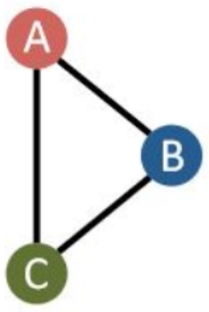
2.2 Communication Models

In distributed systems, failures may occur due to communication failures and processor failures, e.g., crash failures or stop failures. A processor may crash at any point in time. Crash failures are detectable in synchronous systems, e.g., by implementing time-out schemes. However, in asynchronous systems, it is not possible to distinguish between a processor failure and a processor that is just slow to respond. A processor may restart later at some point during a computation. If a processor restart in a well-defined state, the restart is noticeable. The restart is unnoticeable when a processor restarts in the same state it fails. In asynchronous systems, the processors restart time is arbitrary. In synchronous systems, only correct processors proceed with the same clock rate [12].

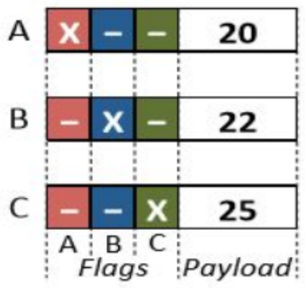
There are two commonly used communication models in the distributed systems: message-passing and shared-memory [9]. In a shared-memory model, processors communicate by reading/writing the shared registers. In a message-passing model, processors are connected through links and exchange messages to communicate.

In a shared-memory system, the presence of shared registers makes the system more robust. If a processor writes a value to a shared register, every future read on this value is always available regardless of processor failures [9]. In message-passing systems, processors rely on the information that is obtained from the messages received. The message passing systems are more prone to failures, for example, the message-loss due to link failures or processor failures can produce inconsistencies in the local-views of different processors. However, it is possible to implement emulators of the shared memory systems in the message-passing systems. Any wait-free protocol that works in a shared-memory system can be implemented as an emulator.

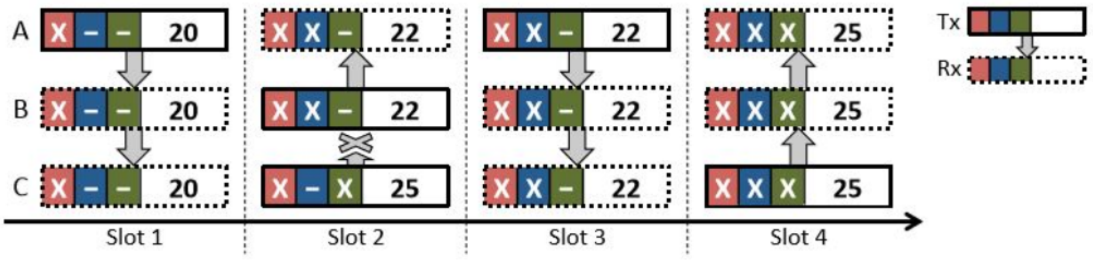
In their work, Attiya, Noy and Dolev [9] present an emulator of a shared-memory model, Single-Write/Multiple-Read atomic register. In the message-passing systems, a global coordination is required to avoid the inconsistencies in the operation. However, the inconsistencies may still occur due to an adversary. An adver-



(a) Node A, B and C are in range of each other.



(b) The nodes add their payload and set their flags.



(c) Node A is initiator and starts the round by transmitting a packet. Upon receiving the packet, node B and C merge the flags and payload and transmit in the next slot. Node A receive the packet from B, merges flags and transmit. Node C learns something new and transmits in Slot 4. In Slot 4, all nodes have the maximum value and the Chaos round ends.

Figure 2.1: An example of the Chaos round, where nodes A, B and C compute the maximum value, from [1].

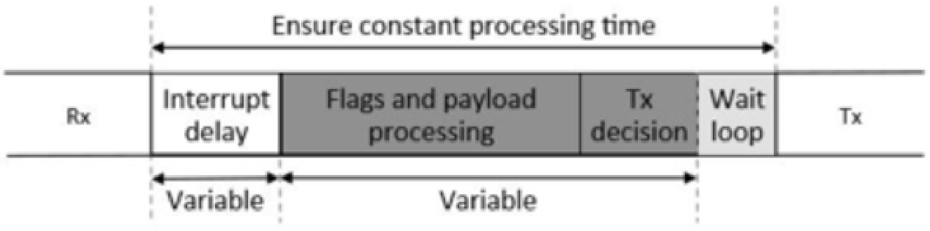


Figure 2.2: Chaos ensures synchronous transmissions; the variable interrupt delay and processing time are compensated by introducing a wait loop, from [1].

sary can split a group into smaller groups, operate them separately and produce inconsistencies. To overcome this problem, the nodes that are disconnected are considered faulty and blocked permanently [9]. A dynamic quorum protocol for the Multiple-writer/multiple-reader (MWMR) registers, is discussed by Nancy and Alexander [11]. Attiya, Noy and Dolev [9] discuss two properties of linearizability in atomic registers:

- A read operation R must return the value that is written by either the most recent write-operation or a concurrent write-operation [9].
- If a read-operation R_1 reads a value from a write-operation W_1, and another read-operation R_2 reads a value from a write-operation W_2, if R_1 precedes R_2 then W_2 does not precede W_1 [9].

2.3 Commit Protocols

Two-Phase Commit (2PC) and Three-Phase Commit (3PC) are two types of commit protocols. Achieving consensus or reaching on an agreement is sometimes crucial in a distributed system. Both the 2PC ans 3PC protocols can be used as consensus protocols[2].

2.3.1 2PC

In 2PC, there exists a single static coordinator and a number of participants called cohort. 2PC consists of two phases: proposal and decision. In the proposal phase, the coordinator broadcasts a proposal to cohort. Each participant of the cohort replies with a vote either yes or no. In the decision phase, based on the votes received, the coordinator decides either to commit or abort. If all participants vote for yes, the coordinator decides to commit otherwise it aborts. Then the coordinator broadcasts its decision to cohort.

2PC is simple and less complicated as compared to 3PC. However, it has a major drawback as being a blocking protocol. If a node fails, other nodes wait for its response and block until the node responds. Recovery schemes can be considered but they cannot handle two or more failing nodes. For example, if both the coordinator and a node fail during the decision phase, other nodes might be in an uncertain state.

2.3.2 3PC

3PC decouples the decision and commit by introducing a pre-commit phase. Therefore, there are three phases in 3PC: proposal, pre-commit and do-commit. The proposal phase is same as in 2PC. If all nodes vote yes, the coordinator decides to move in the pre-commit phase; otherwise, it aborts. When the nodes switch to the pre-commit phase, they acknowledge the coordinator. When the coordinator receives acknowledgements from all nodes, it broadcasts a do-commit and all nodes perform commit.

3PC is non-blocking when there is a single node failure. If a single node fails, rest of the nodes time-out and recover independently. It can handle the failure of multiple nodes and the coordinator. If the coordinator or multiple nodes fail, the nodes can terminate safely using a recovery protocol. However, in case of complex network partitioning, the nodes can still have inconsistencies.

2.4 Quorum Protocol

A quorum is comprised of two non-empty subsets that intersect each other at least at one point, e.g., a quorum Q for N number of nodes is parameterized as $R * N + 1$, where R is the ratio. A minimum ratio that is required to form a quorum is 0.5. For a majority quorum the ratio R varies between 0.5 to 1. In his paper, Thomas [13] presents a quorum-based approach to achieve the concurrency goals in a replicated database. In a quorum-based system, for each operation (read/write), at least a quorum should be ready to respond. It is a non-blocking protocol. For example, in case of processor failures, a reader/writer does not wait for the acknowledgments from all nodes or a specific node. Thus, the protocol provides a reliable, fault-tolerant and highly available service. However, a quorum protocol only works when at least majority of servers are alive or non-faulty.

One drawback of the commit protocols (2PC/3PC) is that they are slow, the coordinator can only proceed with a transaction when it receives votes from all nodes. As compared to 2PC/3PC, in majority quorum protocols, only a majority number of nodes are required to perform a task. Hence, quorum-based algorithms are more efficient and reliable. We discuss two types of atomic registers: Single-Writer/Multiple-Reader (SWMR) and Multiple-Writer/Multiple-Reader (MWMR).

2.4.1 SWMR Register

In an SWMR protocol, only one specific processor can perform the write operations, but more than one processors may read a register. Every read operation must return the value that is written by the most recent preceding write-operation [9]. To avoid the read-write conflicts [10] [14], a write gets preference over the concurrent read [9].

SWMR reader's protocol:
The reader's protocol for SWMR consists of two phases: query and propagation.

Query phase:
- The reader sends a read query to all processors for the latest timestamp.
- Upon receiving the responses from the majority of processors, the reader computes the latest timestamp.

Propagation Phase:
- The reader transmits the results to all processors.
- Upon receiving acknowledgments from a majority of processors, the reader returns the read.

SWMR writer's protocol:
The writer's protocol for SWMR protocol is comprised of four steps as follows:
- The writer computes a new value and increments the timestamp by one.
- The writer then sends the updated value and timestamp to all processors.
- Upon receiving acknowledgments from a majority of processors, the writer acknowledges the user.

2.4.2 MWMR Register

In the MWMR registers, more than one writer exist. So, a register can be accessed by any of the available writers to perform a write operation. However, only one writer can perform a write operation at a time to avoid the write-write conflicts [10] [14].

The reader's protocol for an MWMR register is same as for the SWMR register. However, in the MWMR registers, the writer's protocol requires an additional query phase [11].

MWMR writer's protocol:
In the MWMR registers, the writer does not know about the latest timestamp, therefore the writer performs a read operation before every write.
Query phase:
- The writer sends a read query to all servers for the latest timestamp.
- When the writer receives responses from a majority of processors, it updates its local timestamp with the highest timestamp received.

Propagation Phase:
- The writer computes a new value and increments the timestamp.
- For the concurrent writes, a writer with the highest timestamp and the highest writer's ID gets preference. For example, if there are more than one writers with the same timestamp then the writer with the highest ID performs the write operation.
- The writer transmits the timestamp and value to all processors.
- Upon receiving acknowledgments from a majority of processors, the writer acknowledges the user.

Chapter 3

Related Work

Most of the convention and newer approaches, such as TSCH [15] and Orchestra [16] relies on the routing protocols to provide best-effort low power routing. Achieving end-to-end reliability on a best-effort low power routing protocol is crucial. Some solutions implement the rate-control protocols on top of best-effort protocols to achieve end-to-end reliability. However, these protocols increase latency. Group communication is also challenging. Group communication protocols that rely on the best-effort protocol, do not provide end-to-end reliability.

Most of the research in LPWN, covers only distributed data processing and aggregation. In practice, many conventional approaches are limited to single-hop networking [17], such as JAG [18] that provides reliable agreement between neighboring nodes. Another example of single-hop networking is 6P protocol [19], provides transactions between pairs of neighbors. In this section, we discuss one of the most recent developments in Chaos, Agreement on Air A^2 [2].

3.1 Agreement on Air

A^2 [2] introduces a distributed consensus in low-power devices. A^2 is developed on Synchrotron, a robust kernel that provides synchronous transmissions. A^2 provides a network wide agreement based on commit protocols: two-phase Commit (2PC) and Three Phase Commit (3PC).

3.1.1 Synchrotron

Synchrotron is based on a time-slotted design, the smallest unit of time is a slot. Usually, a slot is few milliseconds long, but long enough that the nodes can receive, process and transmit. A certain number of slots group up to form a round. The nodes should complete their task within a round. Synchrotron provides in-slot energy-saving mechanism, a node can either transmit or receive in one slot, and radio is remained off during the processing time. The radio is turned on only for the transmission or when the node expects to receive data from neighbors.

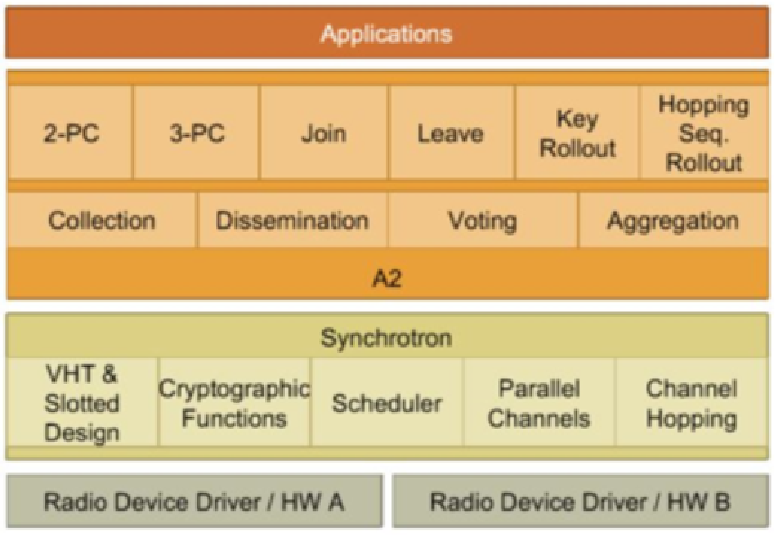


Figure 3.1: Layered Architecture of A^2; Synchrotron provides the base layer services. A^2 relies on the basic Chaos communication primitives and provides a network wide agreement. Figure copied from [2].

A layered architecture of A^2 is shown in the Figure 3.1. A Virtual High-Definition (VTH) timer is used to achieve the synchronization goals. The effect of interference is reduced by introducing the parallel channels. The nodes pick a channel randomly. Each time, the nodes transmit on a different frequency. A scheduler is used to run multiple applications of A^2. It schedules the tasks according to their priorities and deadlines. The cryptographic functions are used to provide hardware-based integrity check.

3.1.2 Communication Primitives of A^2

Collect, disseminate and aggregate are the basic primitives of Chaos. In addition to these primitives, A^2 also provides the voting primitive. The voting primitive allows the nodes to vote for or against a proposal. Figure 3.2 illustrates the network wide voting in A^2; the initiator starts a round and suggests a value. Upon receiving, the nodes vote for commit/abort, merge the flags and transmit the packet in the next time slot. A node only transmits when it learns something new. Eventually, the initiator receives votes from all nodes. If everyone votes for the commit, the initiator proceeds with the commit, otherwise it aborts. The initiator disseminates the results in the network. Upon receiving, the nodes commit/abort based on initiator's decision. After that the nodes turns off their radio and sleep until the next Chaos round.

3.1.3 Group Membership in A^2

The information regarding synchronization is enclosed in the header of the packet. The nodes wishing to join, start listening the channel. Upon receiving an A^2 packet,

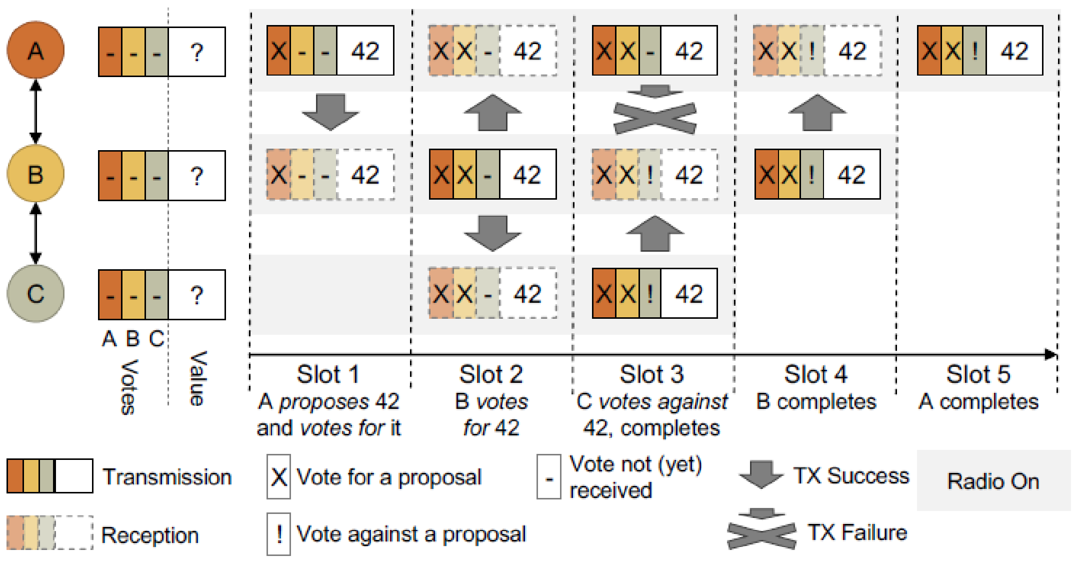


Figure 3.2: Network-wide voting [2]: A initiates the proposal in Slot 1. B votes for the proposal and transmit in Slot 2. C votes against the proposal, C completes and transmit in Slot 3. A and B complete in Slot 5 and 4 respectively. Figure is taken from [2].

the node learns about the next scheduled round, length of the round, slot length and size of the packet. The nodes, set their join flag to 1 and transmit the packet in next time slot. Upon receiving the join requests, the coordinator starts a join operation. The join operation runs in two phases: collect and disseminate. In the collect phase, the new nodes add their IDs to the list, already joined nodes show their participation by setting up their flag. The coordinator monitors the progress flags of already joined nodes, to decide whether to switch or not to the next phase. If all flags are up, the coordinator assigns flag indexes to the new nodes and disseminate the new list of flags in the group. When a node wants to leave the group, the coordinator schedules a leave round. The coordinator removes flag index of the node and disseminates the new list.

3.1.4 Network Wide agreement

A^2 provides network-wide agreement based on 2PC and 3PC protocols. The nodes reach an agreement within a single round. 2PC requires two phases of synchronous transmissions. In first phase, the nodes propose, vote and send feedbacks to the coordinator. In the second phase, the coordinator decides whether to commit or abort, and disseminates the results. Figure 3.3 illustrates an example of a 2PC round. 2PC is a blocking protocol. If a node vote for the commit and coordinator fails, the node blocks until the coordinator recovers and start responding. However, if a node fails before voting, the coordinator aborts after the time-out. If a node votes for no and time-outs, it safely aborts after recovering.

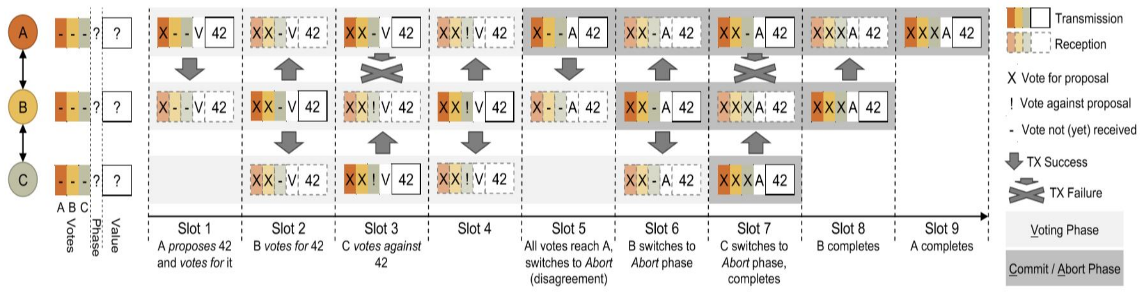


Figure 3.3: Two-phase Commit [2]: in Slot 1, the initiator, node A proposes value 42. Node B agrees on the value and votes for it in Slot 2. Node C disagree and vote against the proposal. In Slot 5, the initiator receives feedback from all nodes and decides to abort the transaction. The initiator's decision propagates through the network, and at the end of second phase, all nodes reach the consensus to abort. Figure is taken from [2].

To overcome the blocking drawback of 2PC, 3PC decouples the proposal-phase from the decision-phase and introduces a pre-commit phase between them. The coordinator proposes a value and collects votes from the cohort. After the proposal phase, the coordinator decides whether to enter the pre-commit phase or abort. If all nodes in a cohort vote for the commit, the coordinator moves to the pre-commit phase. If the coordinator time-outs or if it receives one or more votes against a proposal, it aborts. There is no voting required after the proposal phase; the nodes only send acknowledgments (ACKs) to the coordinator. When the nodes move to the pre-commit phase, they send ACKs to the coordinator. When the coordinator receives acknowledgments from all nodes, it moves to the commit phase. Otherwise, it aborts to avoid inconsistencies. If a node time-outs after receiving a do-commit, it proceeds with commit after recovery. If the coordinator fails after pre-commit phase, the nodes proceed with the commit. After the recovery, the nodes can only proceed with commit if the state before the time-out is pre-commit or commit otherwise they abort.

3.1.5 Results

The performance of A^2 is measured on four metrics: reliability, availability, radio duty cycle and radio on time. Figure 3.4 shows the results of the comparison between A^2, Low Wireless Buss (LWB) and LWB with Forwarder Selection(FS). As compared to Chaos, LWB splits all-to-all interactions into sequential collection, processing, and dissemination phases, thus it is inefficient and produces high latency. In terms of reliability, A^2 achieves the lowest loss rate. However, 2PC and 3PC shows a higher values for latency and duty as compared to Max, Disseminate and Collect, because each phase of 2PC and 3PC involves voting or feedback from the cohort.

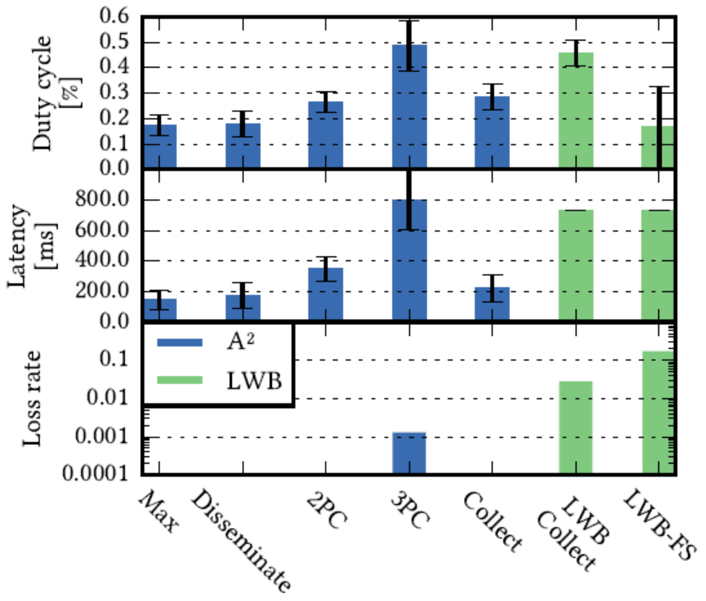


Figure 3.4: Performance comparison between A^2 and LWB : LWB achieves the lowest duty cycle but it is least reliable, from [2].

Chapter 4

Design

In this chapter we present our design choices for the SWMR and MWMR protocols. We begin with discussing the basic communication, followed by the communication primitives. Then we present our quorum configurations. Later in this chapter, we discuss how the SWMR and MWMR quorum protocols work together with Chaos primitives, sections 4.4 and 4.5.

4.1 Synchronous All-to-All Communication

Chaos relies on synchronous all-to-all transmissions and in-network processing. In our quorum algorithms, the nodes synchronously perform the read/write requests and transmit the results to other nodes in the network. A node initiates a read/write request and transmits the request in the network. Upon receiving a read/write request, the nodes process the request and retransmit. Figure 4.1 presents a layered architecture of our design. We develop our algorithm on the Synchrotron in the similar way as A^2. The minimum unit of time in our setup is a slot. A slot is long enough that the nodes can transmit, receive and process a request during one slot.

4.2 Communication Primitive

We adapt the basic communication primitive from Chaos, collect, disseminate and aggregate. For each read/write, the nodes read/write to a majority of nodes, process the read/write requests and disseminate the results in the network. The nodes always share the latest copy of data they have. In addition to Chaos communication primitive, the *vote* primitive is inherited from A^2 [2]. Whenever a node proceeds with a read/write request, it votes for the request. We modify the Vote primitive according to our majority quorum configurations.

In SWMR and MWMR, the nodes start a round with read/write request. When a node wants to start a read/write request, it disseminates the request in the network and collects votes from at least a majority of the nodes. Other nodes, upon receiving the request, decide whether to proceed or discard the request. When a node decides to proceed with a request, it adds its vote, perform the read/write operation and

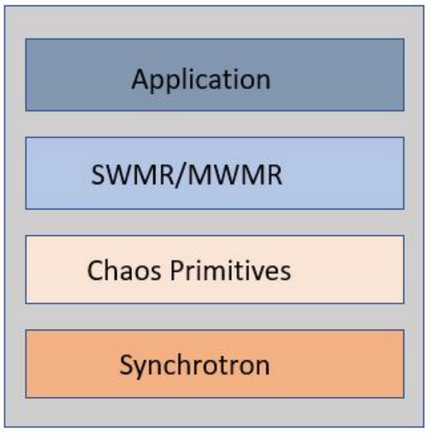


Figure 4.1: Layered architecture of SWMR/MWMR: Synchrotron is the base layer. Chaos primitives are collect, voting, dissemination and aggregation.

transmit the results. When a read/write request achieves at least a majority of votes, results of the request are returned to the user.

4.3 The Majority Quorum Configurations on Chaos

In this thesis, we aim to implement a majority quorum on Chaos. In our majority quorum configurations, a read/write request is considered successful when it achieves at least a majority of votes. The progress flags are used for the voting purpose. Each node has its unique entry in the flags. Whenever a node initiates a read/write request it adds its flag bit and transmits the request. Upon receiving, if the nodes agree to proceed with the request, they set their corresponding flag bit in the flags, merge the flags with received flags and retransmit the request with the resultant flags. The transmission policy of SWMR and MWMR is the same as in Chaos, the nodes transmit whenever they learn something new from the last received request.

The nodes continuously monitor their flags. The nodes move towards the completion procedure when they achieve a majority of flags on a read/write request. In the completion procedure, the nodes perform a final flood before sleep. In the final flood, the nodes aggressively transmit their results a specific number of times. After finishing the final flood, the nodes sleep until next round.

4.4 Mapping of SWMR, Quorum protocol on Chaos

The SWMR protocol allows only one specific writer to initiate the write requests. The same writer performs all write operation, therefore, the writer is aware of the latest timestamp. A basic SWMR protocol is presented in Section 2.4.

Whenever the writer wants to write, it initiates a packet with a WR header, computes a new value, increments the timestamp by 1, sets its flag bit, and transmits.

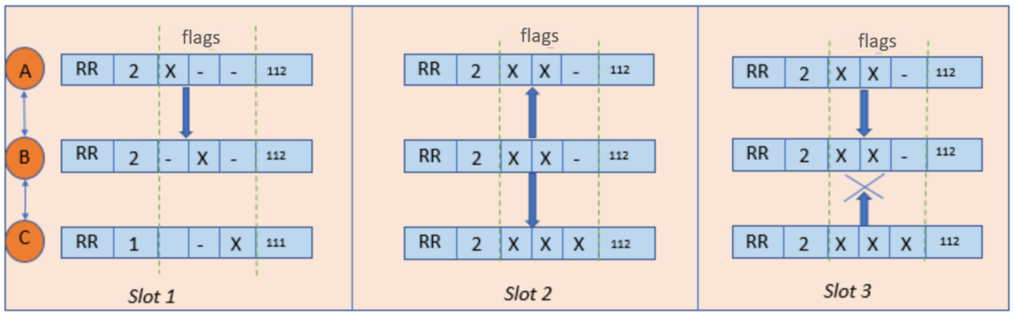


Figure 4.2: An SWMR Read Round: node A initiates an RR packet. In Slot 2, all nodes knows the latest value and time. Node B and C learns something new in Slot 2, therefore, both transmit in Slot 3. Node B transmits in Slot 2, therefore it listens in Slot 3.

Figure 4.6 presents the packet format of SWMR protocol, details on the packet format are given in Section 4.6. Upon receiving a write request, the nodes compare the timestamp of the received packet with their local timestamp. If the timestamp is smaller, the nodes discard the packet. If a node receives a WR packet for the first time, it updates its data and transmits in next possible slot. Otherwise, the nodes just merge the flags. The nodes compare the merged flags with the flags of received packet and transmit if there are differences. Figure 4.3 presents a write round of our SWMR quorum protocol. In Slot 1, node A initiates a write request and transmits to node B. Upon receiving, node B updates its timestamp, value and flags and transmits in Slot 3. In Slot 3, node C receives the packet for the first time, therefore, it updates its data and flags. Node A (the writer), receives the first response on the write in Slot 2 from B and merges the flags. Both A and C transmit in Slot 3. Node B transmits in Slot 2, therefore, it listens in Slot 3.

To read the latest value, the reader initiates a packet with an RR header, its local value and timestamp. Upon receiving the RR packet, the nodes compare the timestamp of the received packet with their local timestamp. If the received timestamp is smaller, the nodes transmit a packet with their local timestamp and value. Figure 4.2 present a read round of the SWMR quorum protocol. Node A starts an RR packet in Slot 1. Upon receiving the packet, node B adds its flag-bit, merges the flags and transmits in next slot. In Slot 2, node A receives back the RR packet it initiated in Slot 1. Node A merges the flags and transmits the merged packet in Slot 3. Node C merges the flags in Slot 2 and transmits in Slot 3. Due to the capture effect [2], node B receives the packet from node A with a higher probability.

Eventually, a read/write request achieves a majority of flags. The time slot, upon which more than half of the flag bits are set, is called the completion slot. Reaching the completion slot, the nodes aggressively transmit their results a specific number of times. After completing the transmissions, the nodes sleep until the next round.

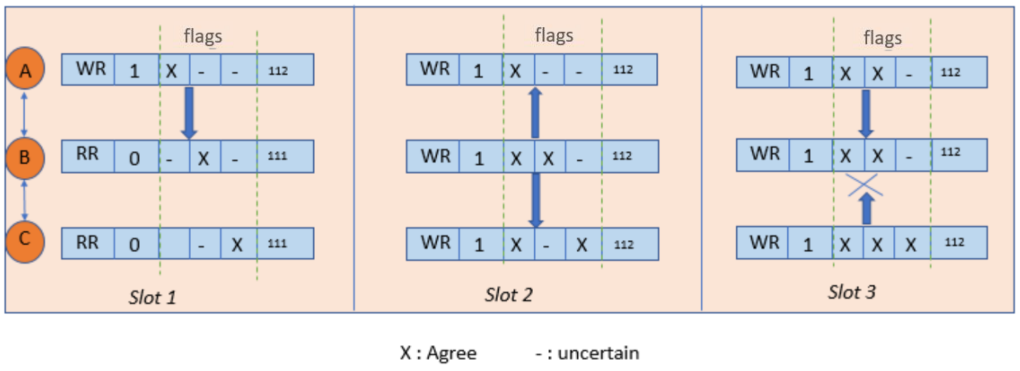


Figure 4.3: An SWMR write round: Node A initiates a WR packet in Slot 1. Upon receiving the WR packet, nodes B and C update their timestamp and value, add their flag bit and transmit the merge packets. All nodes update their data in Slot 2.

If a read/write operation is failed to obtain a majority of flag bits within a round, the algorithm indicates a read/write failure to user.

The condition for atomicity is discussed in Section 2.2 and says if a read occurs concurrently with a write, the read returns the value that is written by the concurrent write. A round may begin with more than one read/write requests. To avoid read-write conflicts the write always gets preference over the reads. Therefore, if one of the requests is the write, the nodes only proceed with the write request. If a node initiates a read-request and it receives a write request later, the node discards its read request and proceeds with the write request.

4.5 Mapping of MWMR, Quorum protocol on Chaos

As compared to SWMR, MWMR allows multiple writers. Any node in the network can initiate the write requests. The read protocol of MWMR is the same as in SWMR but the write protocol requires a read prior to the write. A basic MWMR protocol is presented in Section 2.4. The read protcol of MWMR works exactly like the SWMR's read protocol. Any node in the network can initiate a read/write request. Other nodes upon receiving, merge the information and return the latest version of timestamp and value. However, the MWMR packet also carries the node-id of the sender. Figure 4.7 presents the packet format of our MWMR protocol. Section 4.6 presents the details of different fields of the packet. The purpose of including the node-id in the packet is to resolve the write-write conflicts due to concurrent writes. The read protocol of MWMR does not care about the node-ids.

In the MWMR protocol, when a node wants to perform a write request, first it initiates a read request and runs the MWMR read algorithm. When the node obtains a majority of flags on that read, it switches to the write phase. In the write phase, the node resets its flags, changes the header from RR to WR, increments the timestamp, updates its value and sends the packet to other nodes in the network.

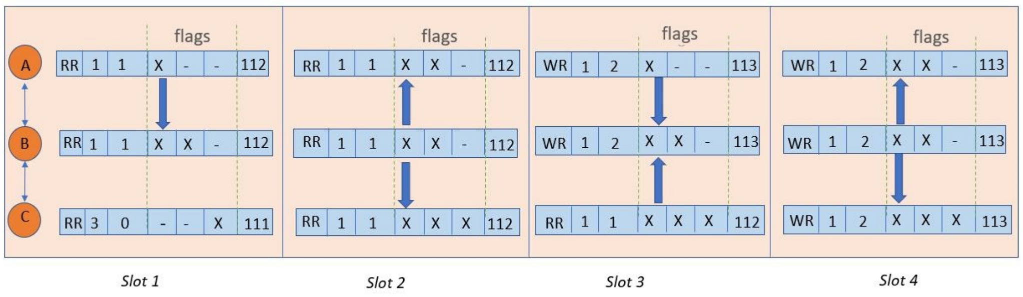


Figure 4.4: An MWMR write round: Node A wants to initiates a WR packet, therefore, it starts an RR packet in Slot 1. Upon receiving the RR packet, nodes B and C update their flag bit and transmit the merge packets. In Slot 2 node A completes the read-prior-to-write and it initiate a WR packet in Slot 3. All nodes update their data in Slot 4 and write completes.

Upon receiving the WR packet, the node update its timestamp and value, merges the flags and retransmits the packet. Figure 4.4 illustrates a write round of our MWMR protocol. Node A wants to perform a write, therefore, the node starts a read request prior to the write. Node A initiates an RR packet in Slot 1. Upon receiving the packet, node B merges the flags and transmit the packet in Slot 2. In Slot 2, node C receives the RR packet, node C adds its flag bit and transmits in Slot 3. Node A receives a majority of flags in Slot 2, hence it moves to the write phase. It computes a new timestamp and value from the received RR packet. Node A initiates a WR packet in Slot 3 and sends it to the node B. Upon receiving the WR packet, node B adds its flag bit and transmit the packet in Slot 4. In Slot 4, all nodes achieve the majority of flags, hence the write request completes.

In the MWMR protocol, one or more nodes may start a write request at the same time that produce the write-write conflicts [10]. The MWMR protocol solves the problem by using the node-ids. If a node receives more than one WR packet with the same timestamp but different node-ids, the node only proceeds with the write request that has the higher node-id. Figure 4.5 presents an example of the concurrent write scenario, both the node B and C wants to write (we suppose the node already have performed a read prior to the write). Node A receives a WR packet from node B in Slot 1. Node A updates its timestamp, value and ID, then it adds its flag bit and retransmit the packet. Due to the capture effect, node B only receives the packet from node C. When node B receives the packet from node C, it realizes that the received packet has a higher ID. Therefore, it discards it packet and proceeds with the packet received. Node B adds its flag bit and transmits the packet in Slot 3. Upon reaching Slot 3, all nodes achieve the majority flags. All node proceed with the same packet initiated by node C.

4.6 Packet Format

We use two different type of packets for SWMR and MWMR protocol. The packets' format of SWMR and MWMR are presented in the figures 4.6 and 4.7. Both packets

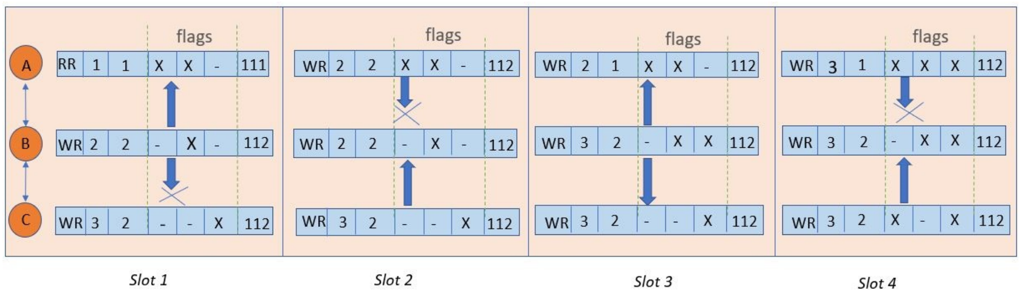


Figure 4.5: Concurrent writes in MWMR: Assuming that both B and C already have performed the read-prior-to-write and they initiate the WR packets in Slot 1 concurrently, node B receives a packet from node C and it sets its flag bit and retransmit the packet. When node A receives the packet, it discards its packet and proceed with the WR packet that initiated by node C. All nodes achieve the majority of flags in Slot 4, hence the write completes

carry a header, timestamp, a set of flag-bits and payload. The header field is 32 bits long and it specifies whether the packet is a read packet or a write packet. The payload field carries a 32 bits long value, written by the writer. An integer number timestamp is attached to each value to keep track of the latest value. The packet also carries a set of flags, the length of flag field depends on the total number of nodes. In addition to these fields, an MWMR packet also carries an ID field. The ID field contains the node-id of the writer.

Payload

The nodes perform all read/write operations on a single object called value. It is an unsigned integer number. In the SWMR protocol, only a specific writer can make changes to the value, therefore, the writer knows the latest value. However, in MWMR there are multiple writers, hence a writer performs a read prior to write.

Header

We specify two types of headers: write-request (WR) and read-request (RR). The header field specifies whether to proceed with a read or a write. When the writer initiates a write, it sets the header of the packet as WR. When other nodes receive that packet, they update their timestamp and value. When a node wants to initiate a read, it sets the header of the packet as RR. Upon receiving an RR packet, the nodes merge the packet and retarnsmit.

Timestamp

The timestamp is an integer number and specifies the version of data, e.g., the highest timestamp indicates the most recent value written by the writer. By looking at the timestamp, the nodes decide whether to accept, merge or discard a packet. When a node receives a packet with a comparatively lower timestamp, it discards the packet. If a node receives a packet with the same timestamp, it merges the

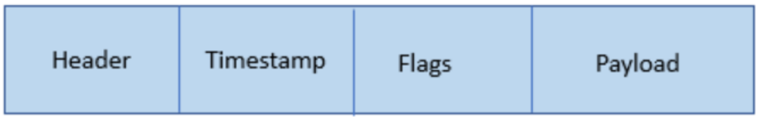


Figure 4.6: The packet format of SWMR: the header of the paket specifies the message type, read or write. The latest packet has the highest timestamp.

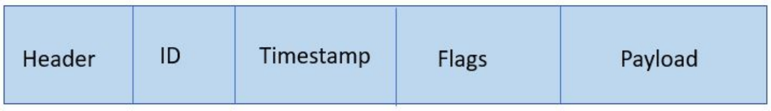


Figure 4.7: The packet format of MWMR: the header, timestamp, flags and payload fields are same as SWMR. ID field of the packet contains the node-id of the writer.

flags. The nodes update their flags and data when they receive a packet with a higher timestamp.

Flags

Each node has its unique entry in the flags, called flag-bit. If there are n number of nodes then at least n bits are required to represent the n nodes. For example for a network of six number of nodes, a set of 8 flag bits is required. The flags are used to make important decisions such as to assess the confirmations/approvals for a value, or to determine the completion of a read/write.

ID

In the MWMR protocol, two or more writers may initiate a write request with the same timestamp. Therefore, to break the tie between concurrent writes, an MWMR packet carries an ID field that contains the node-id of the writer. When a node receives two or more concurrent write requests with the same timstamp and different writer's IDs, it proceeds the one that have the highest ID.

4.7 Summary of Design

The WSMR and MWMR protocols are designed on the top of Synchrotron together with Chaos and A^2 communication primitives. The reader's protocol in SWMR and MWMR is same. However, the writer's protocol of MWMR consists of two phases, read and write. Due to multiple concurrent writes MWMR is prone to write-write conflicts. Therefore, an ID field is included in the MWMR packet, to distinguish between the concurrent writes.

Chapter 5

Implementation

We implement our algorithms on Contiki OS [20] using the C programming language. We choose the *max* application for the basic settings such as clock rate and slot length. We disable the *join* and enable the static node mapping. In the static node mapping, the number of nodes are fixed and defined by the user before running the algorithms. It is very important to correctly define the exact number of nodes. However, the algorithm are adaptive to different number of nodes and calculates the length of flags and the quorum parameters according to the number of nodes defined by user.

We implement two different applications for the SWMR and MWMR quorum protocols. The basic configurations such as *slot_length* and the length of a *round* are the same for both protocols. However, the packets and implementation of algorithms are different for each protocol. In this chapter we discuss the implementation of the five most important elements of our SWMR and MWMR algorithms:

- Initiating read/write
- Transmission policy
- Concurrent read-write policy
- Concurrent Write-Write policy
- Completion policy

We start by discussing how our SWMR and MWMR algorithms initiate the reads and writes. As the two algorithms have different policies for initiating the reads and writes, therefore, we discuss them separately under Section 5.1. Chaos aims to minimize the number of transmissions to reduce the energy consumption. We follow the same trend set by Chaos. The nodes transmit when they learn something new or when they realize that the neighboring nodes have less information. Section 5.2 presents the details on the implementation of the transmission policy. It is already highlighted in the earlier section of this report that the SWMR and MWMR protocols are prone to read-write conflicts. In addition to read-write conflicts, the MWMR protocol is also prone to write-write conflicts. Sections 5.3 and 5.4 discuss how we implement our solution for these conflicts. The nodes observe three phases, first they initiate read/write requests in the proposal phase, after receiving a majority of flags on a proposal they move to the completion phase. In completion phase, nodes aggressively transmit their results a specific number of times. When the nodes finish the completion phase they move to the sleep phase. Section 5.5 presents how we implement the completion policy for our algorithms.

5.1 Initiating the Read/write

We use a flip-coin method to choose between the read and write. To implement the flip-coin function, we use the *rand*() function to randomly generate a value between 0 and 1.

5.1.1 The Read/Write in SWMR

In the SWMR protocol, only one specific node can act as a writer and the rest of the nodes only read. In our algorithm, the initiator is a single writer. Each round initiator chooses between reading and writing. If the initiator starts a write round, all nodes proceed with the write.

Algorithm 1 Initiating a read/write request in SWMR

```
if Initiator Node then
    Flip Coin
    if Heads then
        Set Header = Write Request
        Timestamp = Local_timestamp +1
        Value = Local_value +1
        Transmit

    else
        Query = Read
        Set Header = Read Request
        Timestamp = Local_timestamp
        Value = Local_value
        Transmit

    end
else
    Choose Read and follow the same steps as mention above for the Read Request.
end
```

5.1.2 The Read/Write in MWMR

As compared to SWMR, there are multiple writers in MWMR. In MWMR, a node starts a round by initiating a read/write with a probability of 0.5. In the beginning of each round there may exist more than one write request but only one write request proceeds in a round. The write procedure of MWMR are different than SWMR.

However, the read procedure is the same as in SWMR. The write procedure follows two phases, read query and write query. In the read query phase, the writer starts a read request and performs a read prior to the write. When the writer achieves a majority of flags on the read request, it moves to the write query phase. In the write query phase, the writer computes a new timestamp and value from the data it received during the read query phase. After computing the timestamp and value, the writer prepares a write packet with the WR header, inserts the computed timestamp and value, adds its flag bit, and transmits the packet.

Algorithm 2 Initiating a read/write request in MWMR

```
if Initial_State then
    Flip Coin
    if Heads then
        Query = Write
        Set Header = Read Request
        Timestamp = Local_timestamp
        Value = Local_value
        ID = node-id
        Transmit
    else
        Query = Read
        Set Header = Read Request
        Timestamp = Local_timestamp
        Value = Local_value
        ID = node-id
        Transmit
    end
end
else if Query = Write and current header == read request and Flags == majority
then
    Set Header = Write Request
    Timestamp = Local_timestamp +1
    Value = Local_value +1
    ID = node-id
    Transmit
end
```

5.2 Transmit and Receive Policy

The nodes can either transmit or receive during a slot. The nodes only transmit, (i) when they learn something new either by themselves or from a packet received, (ii) when they realize that the neighbors know comparatively less. If a node transmits successfully in a slot, the next slot it receives. At the beginning of the algorithm, the initiator transmits, and everybody else receives. If a node fails to transmit, the node retransmits that packet with a retransmit index. If a node fails to receive, the node retransmit with an invalid-receive index. After reaching the completion slot, the nodes aggressively transmit the results a specific number of times and sleep until the next round.

Algorithm 3 Transmit/receive policy in SWMR/MWMR

```
if Received Successfully then
    if Received Timestamp = Local Timestamp then
        Flags = Local_Flags OR Received_Flags
    end
    else if Received Timestamp > Local Timestamp then
        Update Local Information
        Transmit

    end
end
else if Transmitted Successfully then
    Receive In Next Slot
end
else if Transmit Failed then
    Fail_Transmit_Count ++
    if Fail_Transmit_Count = Retransmit_Index then
        Transmit
        Fail_Transmit_Count = 0
    end
end
else if Receive Failed then
    Fail_Receive_Count ++
    if Fail_Receive_Count = Invalid_Receive_Index then
        Transmit Fail_Receive_Count = 0
    end
end
```

5.3 Concurrent read/write

Each node maintains a local version of the information, such as timestamp, header, value, and flags. Upon receiving, the nodes compare the information in the received packet with their local information and decide whether to update, merge or discard. A node may receive a packet with a different header, for example, a node starts with an RR packet and it receives a WR packet. If a node receives a packet with the same header but with a higher timestamp, the node updates its timestamp, value, and flags according to the packet received. When a node receives a packet with a lower timestamp, the node transmits its local version of information, so that the other nodes can catch up. The pseudo code below presents how our algorithm works with the headers and timestamps.

Algorithm 4 Handling the concurrent read-write requests

```
if $Received\_Header \neq Local\_header$ then
    if $Received\_Header = Write\_request$ and $Received\_Timestamp > Local\_Timestamp$
    then
        Header = Received_Header
        Timestamp = Received_Timestamp
        value = Received_Value
        Reset Local_Flags
        Set Flag_bit
        Merge Flags
        Transmit

    end
    else if $(Received\_Header = Read\_request)$ and $(Received\_Timestamp < Local\_Timestamp)$ then
        Transmit

    end
end
else if $Received\_Header = Local\_Header$ then
    if $Received\_Timestamp = Local\_Timestamp$ then
        Set Flag_bit
        Merge Flags
        Transmit
    end
    else if $Received\_Timestamp > Local\_Timestamp$ then
        Timestamp = Received_Timestamp
        value = Received_Value
        Reset Local_Flags
        Set Flag_Bit
        Merge Flags
        Transmit
    end
    else if $Received\_Timestamp < Local\_Timestamp$ then
        Transmit
    end
end
```

5.4 Concurrent Writes

The concurrent writes in MWMR may produce write-write conflicts. The writer's algorithm in MWMR consist of two phases, read query and write query. In the read query phase, all nodes wishing to initiate a write learns the latest time stamp and value. In the write query phase, the nodes compute a new timestamp and value. Then they initiate a write request packet with the newly computed timestamp and value. It is possible that two or more nodes start a write request packet with the same timestamp. This might be problematic if each node (writers) computes a different value but all of them have the same timestamp. In such a situation, the nodes may have inconsistency in their data, e.g, the nodes may have different values but the same timestamps.

The algorithm solves the write-write conflicts by introducing the ID field to the packet. The nodes add their node-id in the ID field of the packet. When more than one WR packets arrive at a node, the node only proceed with the request that have the highest ID.

Algorithm 5 Handling the concurrent write requests

```
if $Received\_Header = Write\_header$ then
    if $Received\_Timestamp = Local\_Timestamp$ then
        if $Received\_ID = Local\_ID$ then
            Set Flag_bit if not set earlier
            Flags = Local_Flags OR Received_Flags
            if Local_Flags $\neq Received\_Flags$
            $Transmit$
            end
            if $Received\_ID > Local\_ID$ then
                Reset Local_Flags
                Set Flag_bit in Local_flags
                Flags = Local_Flags OR Received_Flags
                Transmit
            end
        end
    end
```

5.5 Completion policy

The flags play an essential role, e.g., the nodes get to know about the status of the other nodes. The nodes continuously monitor their flags to determine the completion slot. The nodes reset their flags when they receive a packet with a higher timestamp. Upon reaching the completion slot, the nodes start their final flood. During the aggressive flooding, the nodes repeatedly transmit the results. A node may learn new flags during the final flood because the nodes transmit in one slot and receive in the other slot. So, the nodes transmit, receive, merge the flags and transmit again. Below we show the pseudocodes for the completion and final flood.

Algorithm 6 Completion of a read/write request, final flood and sleep state

```
if Received successfully then
    if Flags_bits > Total_nodes/2 then
        Completion = 1
    end
    if Completion then
        Transmit_count ++
        Transmit
    end
    if Transmit_count ≥ Aggressive_transmit_index then
        Transmit_count = 0
        Completion = 0
        Turn of Radio and Sleep
    end
end
if Transmited successfully then
    Do Receive in next Slot
end
```

5.6 Summary of Implementation

We implement the SWMR and MWMR algorithms on a static node mapping. In SWMR and MWMR, nodes start their round by initiating the read/write requests. However, in SWMR, only the initiator can perform the write operations. The nodes transmit when they learn something new from the last received packet. When a read/write request occurs concurrently, the write always gets preference over the read. In case of concurrent writes, nodes always choose a packet with the highest node-id. Nodes perform an aggressive flood before switching to the sleep phase.

Chapter 6

Evaluation

In this section we evaluate the SWMR and MWMR protocols. We evaluate our algorithms both on a simulator and a testbed. We start by discussing the evaluation setup, then we present a single round of each SWMR and MWMR to provide a detailed view of the inner workings of these protocols. Next, we evaluate the latency and energy consumption in the SWMR and MWMR protocols. Last we evaluate the long-term performance of the SWMR and MWMR protocols.

6.1 Evaluation Setup

We test our algorithms on the Cooja simulator [21] and Flocklab [3] hardware. For each setup, we run tests on a minimum 6 number of nodes. The network topologies for Cooja and Flocklab are slightly different. As compared to Flocklab, the nodes in the Cooja simulator are placed closer to each other. The network topologies for Flocklab and Cooja are presented in Figure 6.1 and 6.2.

In our test setups, the slot length is $\approx 4ms$ and the interval is $\approx 12s$. We run our algorithms on Synchrotron with 2PC/3PC and max. However, we choose a static network policy. We evaluate the performance of our algorithms on the following four metrics.

- **Radio-on-time** estimates the amount of energy consumed by the radio. It measures the total duration of time the radio is on, during a round. It is the total time the radio transmits and receives in a round.
- **Latency** estimates the efficiency of SWMR and MWMR. This is the average time each node takes to complete a read/write request.
- **Radio-duty-cycle** measures energy-efficiency and provides the percentage of time the radio is on during a time interval.
- **End-to-end loss rate** measures the reliability of the protocol. A round is considered reliable when at least majority of nodes contain the latest version of the value.

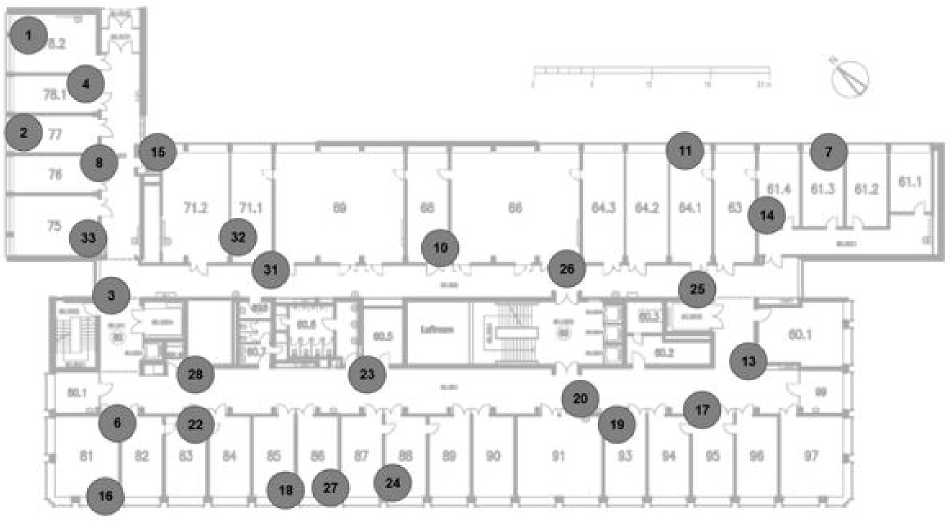


Figure 6.1: Network topology for Flocklab, all nodes do not have direct communication with each other [3].

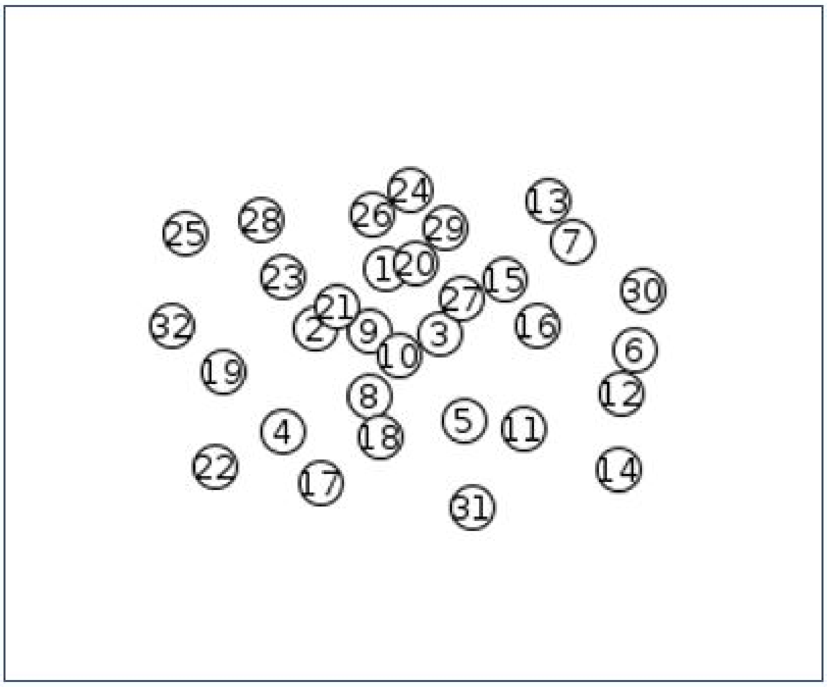


Figure 6.2: Network topology for 32 nodes in Cooja. The nodes are in range of each other.

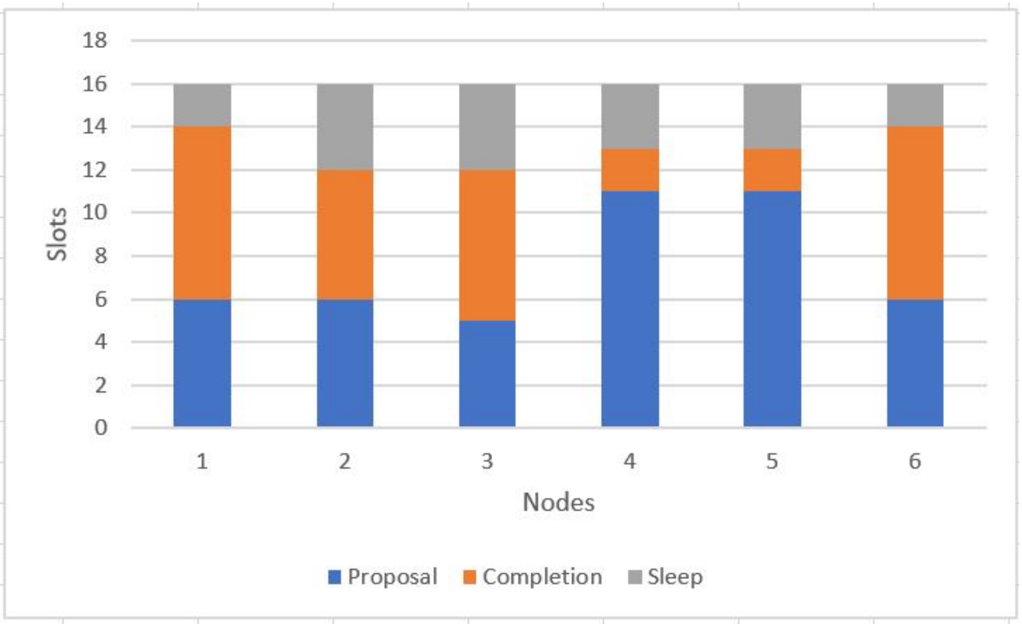


Figure 6.3: Nodes activity during a single SWMR round, for 6 nodes in Cooja. Node 3 is initiator and starts the round with a read/write request.

6.2 Quorum in Action

In the first evaluation, we monitor the activity of each node during a representative round, both for SWMR and MWMR.

Scenario: We divide the activity of each node into three phases, proposal, completion and sleep. The proposal phase starts when a node initiates a read/write request. The completion phase presents two activities: completion of a read/write and aggressive final flood of the results. The sleep phase starts after the nodes finish with the aggressive transmissions. We run our algorithms for SWMR and MWMR in Cooja and Flocklab on at most 32 nodes.

6.2.1 SWMR Round

Figure 6.3 presents a network of 6 nodes and shows the activity of each node during an SWMR round. Node 3 is the initiator and starts the round by initiating a read/write request. All nodes obtain majority of flags after at most 11 slots. The aggressive final flood takes at most 8 slots. After ≈ 14 slots, all nodes move to the sleep phase. Figure 6.4 shows the average activity of the nodes during an SWMR round. We observe that the proposal phase of the 24-node network is longer than the 6-node network.

6.2.2 MWMR Round

Figure 6.6 presents a network of 6 nodes and shows the activity of each node during an MWMR round. Nodes 1,2 and 6 start the round by initiating a write request.

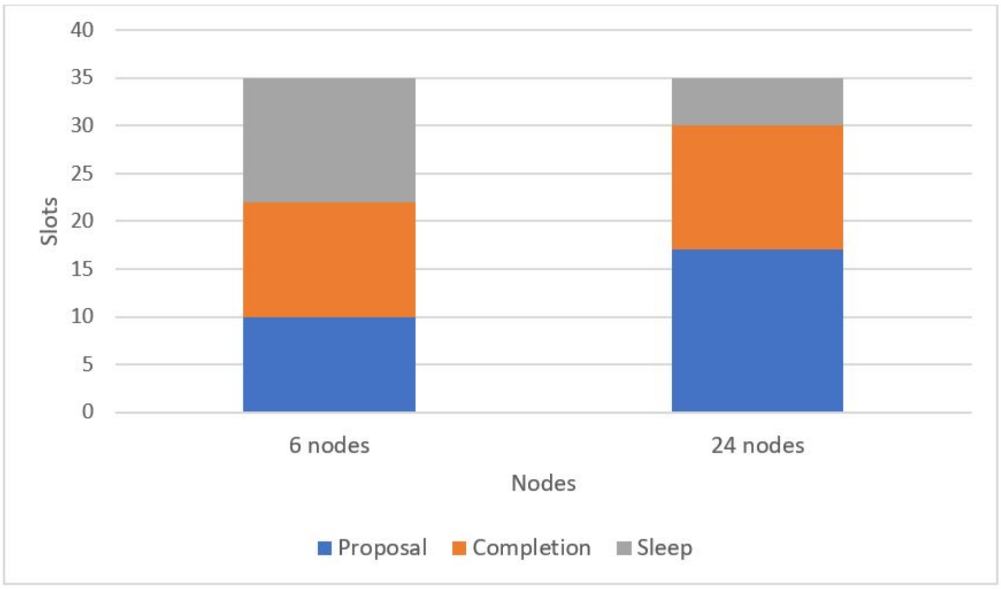


Figure 6.4: Average activity of the nodes in SWMR for a 6-node network and 24-node network.

Other nodes start their round by initiating a read request. The results show that that the nodes take at most 14 slots in the proposal phase. In the proposal phase, the nodes initiate different read/write requests, converges to one write request and obtain a majority of flags on that request. The nodes take at least 30 slots on average to finish their task and begin the sleep phase. Figure 6.5 present an average activity of nodes during 30 rounds in MWMR. The figure also compares the average activity of a 6-node network and 25-node network. The results show that the network of 25 nodes takes more time slots to finish the tasks.

6.3 Latency

This section provides the evaluation results for the latency in SWMR and MWMR protocols.

Scenario To estimate the latency, we setup different test scenarios, both for SWMR and MWMR, and measure the number of slots that are required to complete a read/write request. Chaos provides a facility to measure the total number of slots on a current time instance. It starts counting the slots from the beginning of a round and stores the current number of slots in a variable. We run our algorithms on Flocklab and Cooja, for at least 30 rounds.

6.3.1 Latency in SWMR

Table 6.1 shows the latency results for the SWMR protocol. The results show that the latency increases when the number of nodes increases. For example, in a network of 6 nodes, at least 4 nodes are required to complete a quorum for a read/write

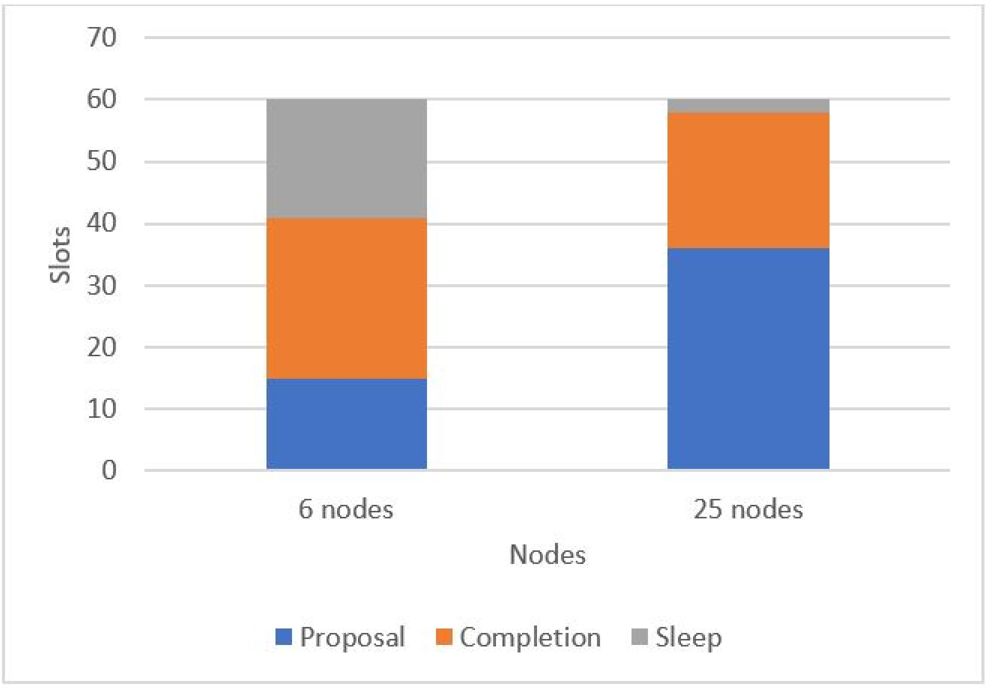


Figure 6.5: Average activity of nodes in MWMR for the networks of 6 and 25 nodes.

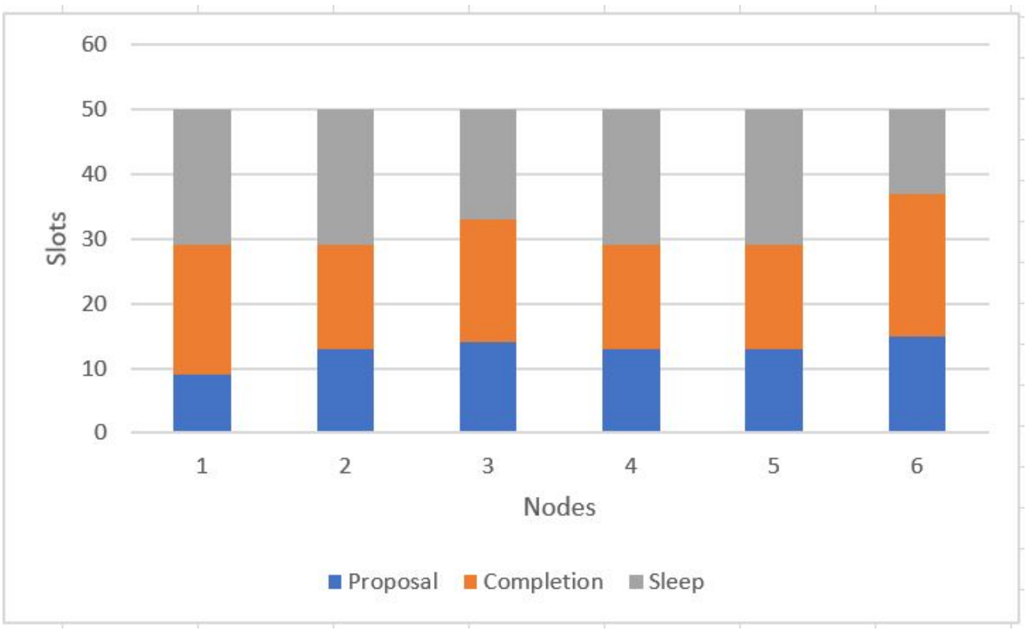


Figure 6.6: Nodes activity during a single MWMR round, for 6 nodes in Cooja.

Platform	Cooja		Flocklab	
Nodes	6	32	6	24
Rounds	80	35	60	60
Latency (Slots)	7	17	10	17
Standard deviation	1.5	2.7	4.1	5.1

Table 6.1: Test results for latency in SWMR. Latency is presented in the number of slots, a slot is ≈4ms.

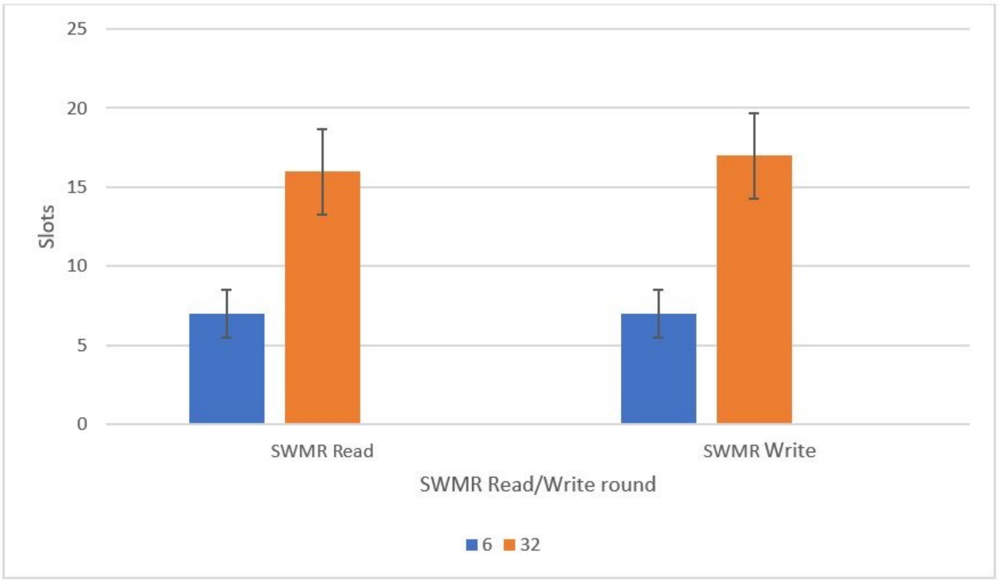


Figure 6.7: Latency comparison between the reads and writes in the SWMR protocol.

request. On the other hand, in the 32-node network, the quorum size increases approximately four times. Hence a node needs at least 17 flags upp to complete a read/write request. As compared to Cooja, Flocklab shows higher number of slots for latency. This is because of different network topologies and nodes inter-connectivity. Figure 6.7 shows the latency results for the read and write rounds explicitly. The results show that both the reads and writes take almost equal number of slots. However, we notice the network size affects the latency as we explain above.

6.3.2 Latency in MWMR

Table 6.2 shows the latency results for the MWMR protocol. The results from Cooja and Flocklab show that the highest average latency is 27 slots for a 24-node network in Flocklab. Figure 6.8 show the latency results for the read and write rounds in MWMR. The figure shows that the latency results of write rounds are twice higher the read rounds. This is because for each write request the nodes need to perform a read prior to write. Therefor, the write requests run in two phases, the nodes take first ≈ 15 slots to read and further ≈ 15 slots to perform the write, hence the nodes take ≈ 30 slots to complete a write request.

Platform	Cooja		Flocklab	
Nodes	6	32	6	24
Rounds	60	35	60	60
Latency (Slots)	13	25	15	27
Standard deviation	5.01	11.09	7.1	11.2

Table 6.2: Test results for latency in MWMR. Latency is presented in the number of slots, a slot is ≈4ms.

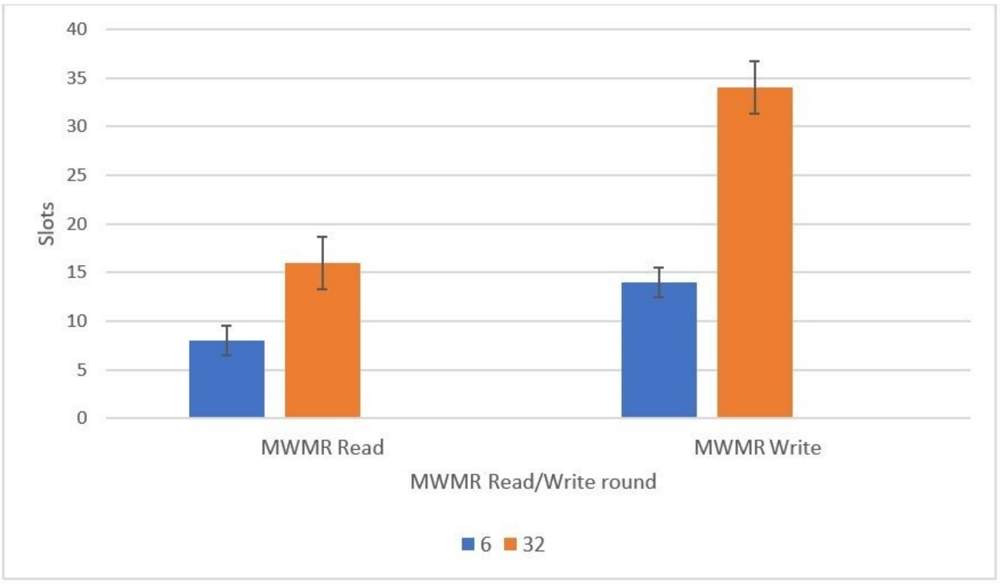


Figure 6.8: Latency comparison between the reads and writes in MWMR protocol.

6.4 Energy Consumption

In this section, we evaluate the energy consumption in SWMR and MWMR.

Scenario We estimate the energy efficiency by measuring the energy consumed by the radios. The energy consumed by the radio is estimated on two metrics: radio-on-time and radio-duty-cycle. To estimate the radio-on-time, we count the number of slots from the beginning of a round till a node sleeps. We utilize the facility provided by Chaos to measure the radio-duty-cycle.

6.4.1 Energy Consumption in SWMR

Table 6.3 presents the results for average radio-on-time in SWMR. The results show that the maximum radio-on time is 30 slots. Figure 6.9 presents the results for % duty-cycle in Flocklab and Cooja. The results show that the highest duty-cycle is 0.016 % for the 32-node network in Cooja.

Platform	Cooja		Flocklab	
Nodes	6	32	6	24
Rounds	80	35	60	60
Radio-on time (Slots)	20	29	23	30
Standard deviation	0.06	0.11	4.1	5.1

Table 6.3: Radio-on time in SWMR. Slot length: $4ms$. Flocklab shows the highest radio-on time of 30 slots.

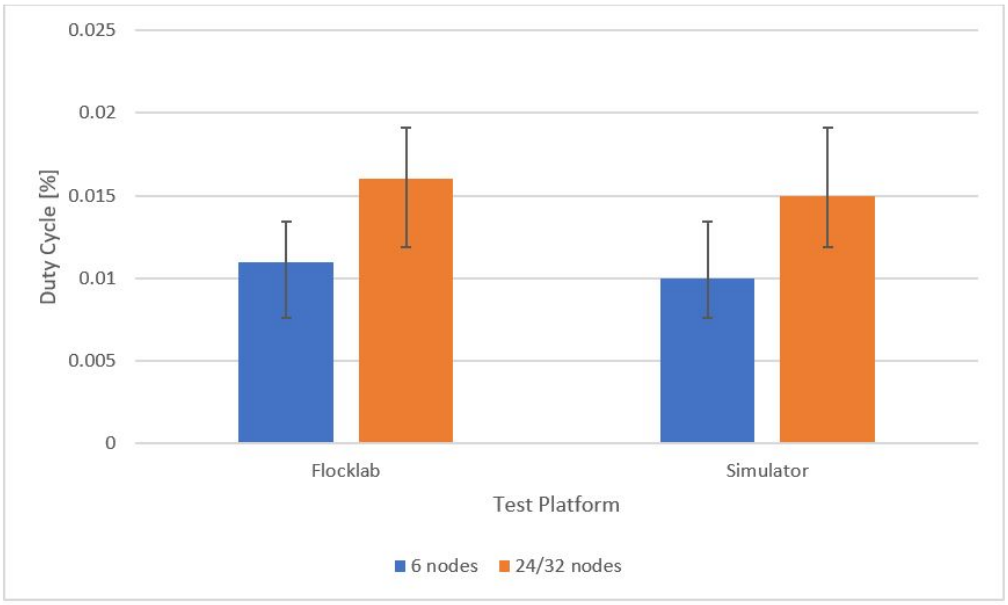


Figure 6.9: Energy consumption as % duty-cycle in SWMR. The highest duty-cycle is 0.016 % for an interval of 12 seconds in a network of 24 nods in Flocklab.

Platform	Cooja		Flocklab	
Nodes	6	32	6	24
Rounds	80	35	60	60
Radio-on time (Slots)	35	50	36	52
Standard deviation	7.3	9.3	9.7	10.1

Table 6.4: Radio-on time in MWMR. Slot length: $4ms$. Flocklab shows the highest radio-on time of $\approx$ 50 slots.

6.4.2 Energy Consumption in MWMR

Table 6.4 presents the results for radio-on-time in MWMR. The results show that the highest radio-on-time is $\approx$ 50 slots. Figure 6.10 shows the results of average duty-cycle in the MWMR protocol. The results show that the highest duty-cycle is 0.03 % for 24 nodes in Flocklab.

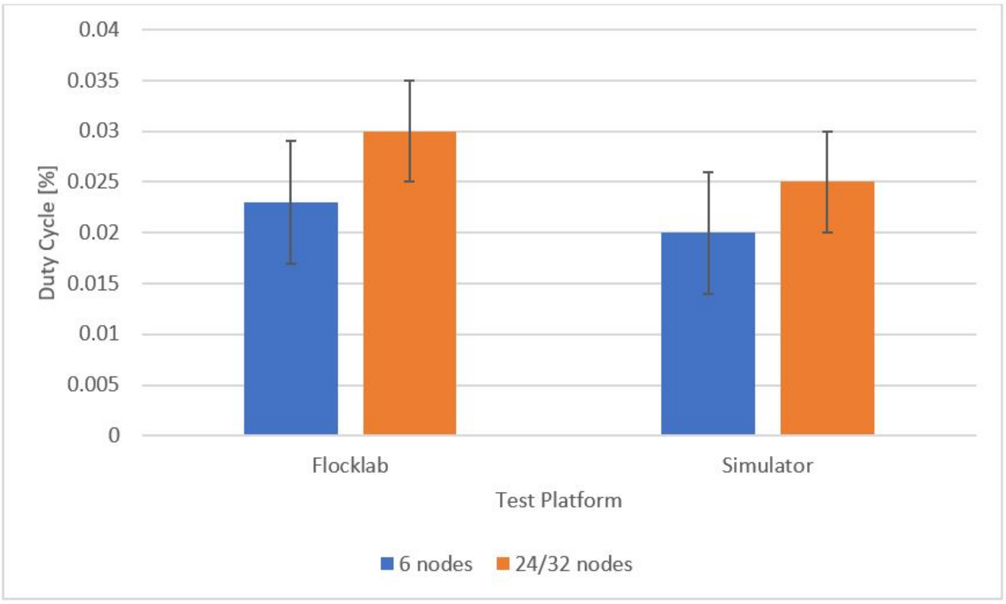


Figure 6.10: Energy consumption as % duty-cycle in MWMR. The highest duty-cycle is 0.03 % for an interval of 12 seconds in a network of 24 nods in Flocklab.

Application		Cooja		Flocklab	
Nodes	Total	6	32	6	24
	Alive	4	29	4	21
To Completion		4	29	4	21
Losses		0	0	0	0

Table 6.5: Reliability in SWMR protocols: the results show that the protocols are highly reliable with 0 loss-rate.

6.5 Long-term Performance

In this section we evaluate the reliability of our algorithms.

Scenario To estimate the reliability, we run each application in the presence of node failures. We run SWMR/MWMR over several rounds and measure the end-to-end loss rate.

6.5.1 Reliability in SWMR/MWMR

Tables 6.5 and 6.6 present the results for end-to-end loss rate in the SWMR and MWMR protocols. The results shows that both the SWMR and MWMR protocols are highly reliable and able to tolerate node-failures.

Application		Cooja		Flocklab	
Nodes	Total	6	32	6	24
	Alive	4	29	4	21
To Completion		4	29	4	21
Losses		0	0	0	0

Table 6.6: Reliability in MWMR protocols: the results show that the protocols are highly reliable with 0 loss-rate.

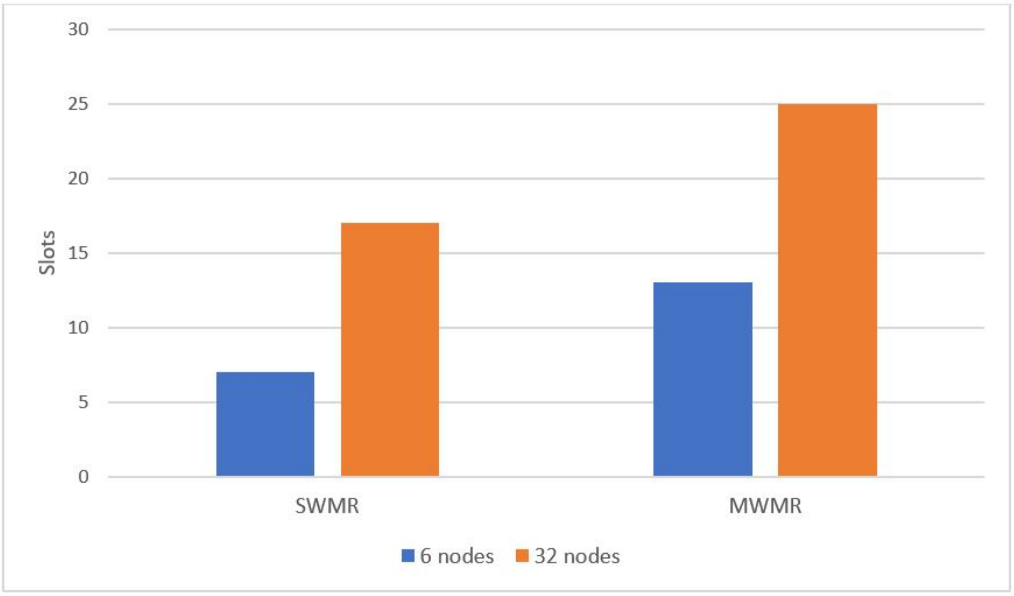


Figure 6.11: Latency comparison between SWMR and MWMR.

6.6 Discussion of Results

In this section we discuss the results of our evaluation. We start by discussing the impact of network size on the results. Then we compare the results of SWMR and MWMR. Lastly we compare the protocols to the state of the art.

6.6.1 Impact of Network Size

We have seen in Chapter 6 that latency increases when the number of nodes increases. Figure 6.11 shows the latency results for SWMR and MWMR. The results show that the latency of a 6-node network in SWMR is 7 slots and it is 17 slots for a 32-node network. The results show that latency increases two times if the network size increase five times. The same trend is followed by the MWMR protocol. Figure 6.11 shows that the latency of a 6 node network in MWMR is 13 slots and it increases approximately two times, 25 slots.

The size of network also affect the radio-on-time and energy consumption. Figure 6.12 shows the impact of network size on energy consumption. The figure shows that radio-on-time for a 6-node network in SWMR is 20 slots and it is 29 slots for

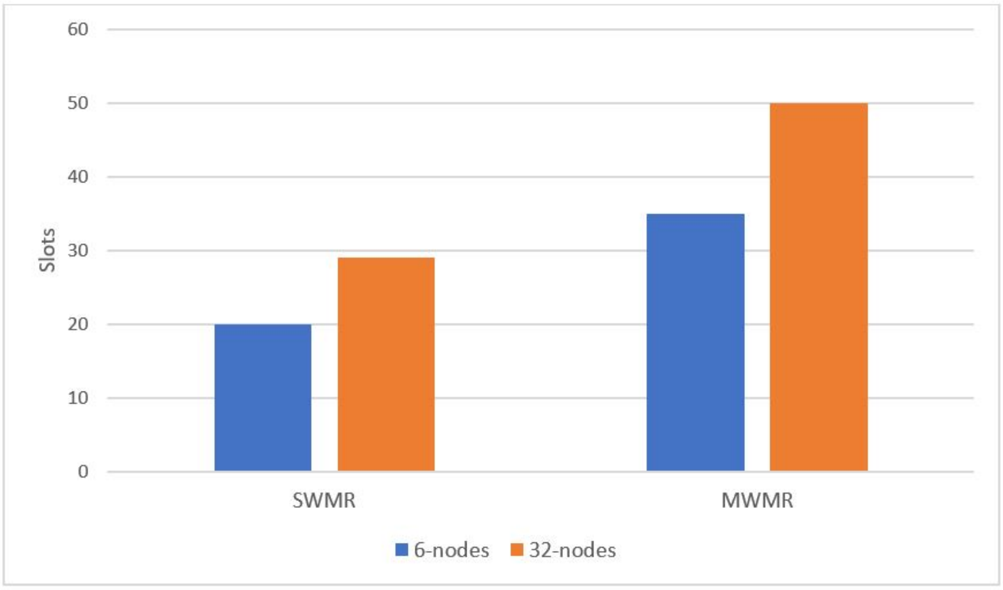


Figure 6.12: Radio-on-time comparison between SWMR and MWMR.

a 32-nodes network. However, the impact of network size is smaller in radio-on-time as compared to latency. By observing the radio-on-time results of SWMR and MWMR, we conclude that the radio-on-time increases 0.5 times when the size of the network increases 5 times.

6.6.2 Comparison between SWMR and MWMR

Figure 6.11 shows the latency results for SWMR and MWMR. The results show that the latency and energy consumption in MWMR are higher than SWMR. This is because the write protocol in MWMR runs in two phases, read query phase and write phase. Figure 6.8 presents a comparison between the reads and writes in MWMR. The results show the latency for the writes is almost two times higher than the reads, in MWMR. Due to the two phase write protocol, energy-consumption in MWMR is also higher than SWMR, as shown in Figure 6.12. However, the read protocol of SWMR and MWMR shows similar results, as shown in Figure 6.7 and Figure 6.8.

6.6.3 Comparison with the State of the Art

We compare the performance of SWMR and MWMR with other Chaos applications, such as max, 2PC and 3PC. We compare the performance on latency, energy consumption and reliability. Figure 3.4 shows the results of duty-cycle, latency and loss rate in max, 2PC, 3PC and LWB [2]. Max shows a minimum latency of $\approx 100\ ms$. The SWMR and MWMR protocols show the latency of 68-100 ms. Max, Disseminate and LWB-FS have lower duty cycle when comparing to 2PC, 3PC, Collect and LWB-Collect, ≈ 0.12. Radio duty-cycle for the SWMR and MWMR protocols is 0.015% to 0.03%. Figure 3.4 shows that Max, Disseminate, 2PC and Collect are highly reliable with zero loss rate. Both the SWMR and MWMR protocols are highly reliable and show zero loss rate (Section 6.5). Hence, the comparison of the results shows that both SWMR and MWMR have the lowest latency, duty-cycle and loss-rate.

Chapter 7

Conclusion and Future Work

Chaos provides efficient all-to-all sharing of data in low-power wireless devices. As compared to traditional networking, Chaos enables in-network processing in an efficient and reliable way. One of the recent development in Chaos is A^2 [2]. A^2 is implemented on Synchrotron, it provides network-wide agreement and enables 2PC and 3PC protocols on Chaos. In this thesis, we focus on a less demanding problem than agreement which is implementation of the shared memory emulator on the message passing system [9]. Therefore, we implement a quorum protocol on Chaos.

We design and implement the SWMR and MWMR, quorum protocols on Chaos. SWMR allows only one specific writer to perform the write operations. The results of SWMR show latency of $\approx 68ms$ (1 slot is $\approx$4ms), and radio duty-cycle of 0.015% for the period of 12 seconds. Compared to SWMR, MWMR allows multiple writers, any node in the network can initiate a write request. Write protocol of MWMR consists of two phases, read query and write. Due to two phase write-protocol, MWMR shows comparatively higher values of latency and radio duty-cycle than SWMR, 108ms and 0.03%. However, both SWMR and MWMR show the lowest values of latency and radio duty-cycle when comparing to other Chaos applications.

We implemented a majority quorum which is the simplest form of quorum. However, we suggest to implement a more complex type of quorum for the future work, dynamic quorum protocol [11]. In this thesis, SWMR and MWMR can handle only one single object (*value*). However, it is possible to handle multiple objects by including the object IDs in the packet [22] [23].

7.1 Ethics and Sustainability

Energy consumption has a direct or indirect impact on our environment because most of the energy produced, does not originated from Eco-friendly sources. It is a fact that information and communication technologies (ICT) industry consumes 6% of the energy worldwide, and the wireless communication industry consumes a major part of it [24]. ICT industry generates $0.3\% - 0.4\%$ of global CO_2 and other greenhouse gases [24] which directly correlate with the global warming phenomenon and air pollution. These amounts are expected to grow even more in the years to come. Hence, it is very important to reduce the energy consumption and emphasis more on the energy efficient communications. In this thesis, we focus a lot on lowering the energy consumption in the LPWN. Energy efficient communications in LPWN not only increase the life of a network device but it is also important for the sake of a healthy environment.

Bibliography

[1] O. Landsiedel, F. Ferrari, and M. Zimmerling, "Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale," *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, pp. 1–14, November 2013.

[2] O. Landsiedel, S. Duquennoy, and B. A. Nahas, "Network-wide consensus utilizing the capture effect in low-power wireless networks," *15th ACM Conference on Embedded Networked Sensor Systems*, pp. 1–14, November 2017.

[3] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, "Flocklab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems," *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, pp. 153–166, 2013.

[4] M. Pajic, S. Sundaram, G. J. Pappas, and R. Mangharam, "The wireless control network: A new approach for control over networks," *IEEE Transactions on Automatic Control*, vol. 56, pp. 2305–2318, October 2011.

[5] F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele, "Low-power Wireless Bus," *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pp. 1–14, 2012.

[6] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pp. 1–14, 2009.

[7] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," *Proceeedings of the Second European Workshop on Wireless Sensor Networks*, pp. 121–132, 2005.

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

[9] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the Association For Computing Machinery (JACM)*, vol. 42, pp. 124–142, Jan 1995.

[10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: a highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[11] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," *In Proceedings of the 27th Inter-*

national Symposium on Fault-Tolerant Computing system(FTCS), pp. 272–281, 1997.

[12] C. Georgiou and A. A. Shvartsman, "Cooperative task-oriented computing: Algorithms and complexity," *Synthesis Lectures on Distributed Computing Theory*, vol. 2, no. 2, p. 167, 2011.

[13] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, pp. 180—-209, 1979.

[14] B. Kemme and G. Alonso, "A suite of database replication protocols based on group communication primitives," pp. 156–163, 1998.

[15] S. Duquennoy, A. Elsts, A. Nahas, and G. Oikonomou, "TSCH and 6TiSCH for Contiki: Challenges, design and evaluation," *DCOSS 2017 - 13th International Conference on Distributed Computing in Sensor Systems*, pp. 1–8, 2017.

[16] S. Duquennoy, B. Al Nahas, O. Landsiedel, and T. Watteyne, "Orchestra: Robust mesh networks through autonomously scheduled TSCH," *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pp. 337–350, 2015.

[17] M. Demirbas, O. Soysal, and M. Hussain, "TRANSACT: A transactional framework for programming wireless sensor/actor networks," *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, pp. 295–306, 2008.

[18] C. A. Boano, M. A. Zúñiga, K. Römer, and T. Voigt, "Jag: Reliable and predictable wireless agreement under external radio interference," *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 315–326, 2012.

[19] Q.Wang, X.Vilajosana, and T.Watteyne, "IETF draft-ietf-6tisch-6top protocol-04, WiP," 2017.

[20] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[21] F. Österlind, "A sensor network simulator for the Contiki OS," *In: SICS Research Report*, 2006.

[22] M. Herlihy, "A quorum-consensus replication method for abstract data types," *ACM Trans. Comput. Syst.*, vol. 4, no. 1, pp. 32–53, 1986.

[23] A. Kumar, "Performance analysis of a hierarchical quorum consensus algorithm for replicated objects," *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 378–385, 1990.

[24] T. Chen, Y. Yang, H. Zhang, H. Kim, and K. Horneman, "Network energy saving technologies for green wireless access networks," *IEEE Wireless Communications*, vol. 18, Oct 2011.