Analysis and Evaluation of Functionality of RAFT Consensus Algorithm in Internet of Things Domain

Bartosz Czapski

A dissertation submitted in partial fulfilment of the requirements of
Dublin Institute of Technology for the degree of
M.Sc. in Computing (Stream)

16/06/2022

Declaration

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Stream), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed:

Date: June, 16, 2022

Abstract

In this thesis author examines significance of the leader-based consensus and its role in improving reliability of Internet of Things (IoT) devices seen as a broader self-organizing domain. Recent advances in the development of low-cost, single-board, high-performance interconnected devices able to receive and process data from multiple sources led to emergence of IoT which can be perceived as an invisible layer of cooperating sensors, working continuously in the background sensing, measuring, connecting and transmitting huge quantities of environmental data. Once collected, this data is perpetually transmitted either over the internet network or long range, low power, wireless platform (LoRa) to gateway nodes capable of storing, analyzing and processing it before passing events requiring careful consideration upstream, towards server-based data aggregation points. Due to amount of data collected and transmitted non-stop between the IoT nodes, implementation of IoT network presents a significant challenge. Attempts are being made to organize sensing devices, based usually on their location, into clusters that are capable of "intelligent" cooperation and working in unison. Among proposed solutions, achieving consensus between the associated devices motivated the research for this thesis. Building upon previous research, author decided to focus on two leader-based consensus algorithms, Paxos and RAFT, that are taking very similar approach to achieving distributed consensus and are well known and yet not fully implemented in the IoT universe. Although both algorithms are leader-based, Paxos requires all nodes within the cluster to exchange log entries while the election process is taking place. This allows for each node to participate in leadership election, but it increases complexity and network usage. On the contrary, RAFT algorithm requires node's logs to be up to date to be considered for leader

election process and it allows for highly efficient and reliable undertaking. During the research process, author investigated multiple recent works discussing distributed consensus in multi-agent systems (MAS) and attempted to implement RAFT and Paxos algorithms in IoT system to evaluate functionality of RAFT consensus algorithm and its advantages over Paxos algorithm in decreasing packet loss rate and latency in IoT network with LoRa.

Keywords: Raft, Paxos, consensus, IoT, LoRa

Acknowledgments

I would like to thank my supervisor, Dr. Bujar Raufi, for his ongoing support and advise throughout this dissertation.

I would also like to thank my mother Halina, my father Krzysztof, and my sister Karolina, for their constant encouragement and support.

Finally, I would like to say a very special thank you to my girlfriend Federica for enormous support and patience. Without her encouragement and support this dissertation would never have been possible.

Contents

Declaration **I**

Abstract **II**

Acknowledgments **IV**

Contents **V**

List of Figures **VIII**

List of Tables **X**

1 Introduction **1**

1.1 Background 2

1.2 Research Project/problem 3

1.2.1 Distributed System 4

1.2.2 Byzantine vs non-byzantine distributed systems 5

1.2.3 Replicated State Machine 6

1.2.4 Consensus 6

1.2.5 Paxos 7

1.2.6 Raft 12

1.2.7 Paxos vs Raft 15

1.3 Research Objectives 16

1.4 Research Methodologies 18

1.5 Scope and Limitations 19

1.6 Document Outline 19

2 Review of existing literature **21**

2.1 Introduction 21

2.2 Related works 22

2.3 Conclusion 31

3 Experiment design and methodology **33**

3.1 Introduction 33

3.2 Aim of Research 33

3.3 Hypotheses 34

3.4 Design overview 34

3.4.1 IoT networks 35

3.4.2 Hardware components 37

3.4.3 Python Libraries 40

3.4.4 Packet Loss Rate measurement experiment 40

3.4.5 Latency measurement experiment 41

3.4.6 Location 41

3.5 Data Design and Data Capture Overview 42

3.6 Design and Methodology Summary 44

4 Implementation **45**

4.1 Introduction 45

4.2 Required External Libraries 45

4.3 LoRa Transceiver Board 46

4.3.1 MicroPython 46

4.3.2 ESP32 Board Set-Up 46

4.3.3 Configuring SX127X RF chip 47

4.3.4 Communicating with Raspberry Pi 48

4.3.5 Transmitting Data 49

4.3.6 Receiving Data 50

4.3.7 Modification to Transmitter and Receiver required for Latency Measurement . . . 51
4.4 Raspberry Pi with DockerPi Hub . . . 53
4.4.1 Python 3 . . . 53
4.4.2 RPi Operating System . . . 54
4.4.3 RPi Wireless ad hoc network . . . 54
4.4.4 Collecting environmental data with Docker Pi Sensor Hub . . . 55
4.4.5 Creating LoRa packet and sending it via UART . . . 56
4.5 Raspberry Pi Receiver node . . . 63
4.6 Conclusion . . . 64

5 Results, Evaluation and Discussion 65
5.1 Introduction . . . 65
5.2 Research Question . . . 65
5.3 Results . . . 65
5.3.1 Packet Loss Rate measurement experiment . . . 65
5.3.2 Latency measurement experiment . . . 74
5.4 Discussion . . . 77
5.4.1 Results Evaluation . . . 77
5.4.2 Strengths and Limitations . . . 78
5.4.3 Conclusion . . . 79

6 Conclusion 80
6.1 Research Overview . . . 80
6.2 Problem Definition . . . 81
6.3 Design/Experimentation, Evaluation & Results . . . 82
6.4 Contributions and impact . . . 83
6.5 Future Work & recommendations . . . 83

References 85

A Code Snippets 94

List of Figures

1.1 Centralized vs. Distributed system architecture 4
1.2 Replicated State Machine 6
1.3 Paxos Phases 9
1.4 Raft Terms 13
1.5 Raft election process 14

3.1 Network A - 3 nodes 36
3.2 Network B - 5 nodes 36
3.3 Network C - 7 nodes 37
3.4 Raspberry Pi - Model 3A+ 38
3.5 DockerPi Sensor Hub Development Board 39
3.6 ESP32 LoRa Heltec v2 with display 39
3.7 Tx and Rx Nodes Setup 42

4.1 UART Basic Connection Diagram 49

5.1 Outdoor Tx and Rx Nodes Setup 67
5.2 3 Nodes, no node failure 69
5.3 3 Nodes, 1 node failure 70
5.4 5 Nodes, no node failure 71
5.5 5 Nodes, 2 node failures 72
5.6 7 Nodes, no node failures 73
5.7 7 Nodes, 3 node failures 74
5.8 Latency in Network A – 3 nodes 76

5.9 Latency in Network B – 5 nodes 76
5.10 Latency in Network C – 7 nodes 76
5.11 Average latency of all networks (A, B and C) 77

List of Tables

1.1 Summary of the differences between Paxos and Raft 16

3.1 Packet Loss Rate Experiments 35
3.2 Latency Experiments 35
3.3 Packet Loss Rate experiment expected data 43
3.4 Latency experiment expected data 43

4.1 External Libraries 45
4.2 IP addresses of IoT nodes 55

5.1 `p-values` 66
5.2 Indoor control experiments results 66
5.3 Errors in data due to collisions 67
5.4 **nodeName** and **packetNumber** 68
5.5 Packet limit at which nodes would turn off to simulate failure 68
5.6 3 Nodes, no node failure 70
5.7 3 Nodes, 1 node failure 71
5.8 5 Nodes, no node failure 72
5.9 5 Nodes, 2 node failure 72
5.10 7 Nodes, no node failure 73
5.11 7 Nodes, 3 node failure 74
5.12 7 Nodes, 3 node failure 75
5.13 Timestamps for latency experiment 75
5.14 Average latency of all networks (A, B and C) 77

Listings

4.1 Firmware installation on ESP32 board 46
4.2 File transfer with Ampy tool 47
4.3 SX127X configuration 47
4.4 SPI interface initialization 48
4.5 UART setup - MicroPython 49
4.6 Class Sender - MicroPython 49
4.7 Instantiation of Sender obj - MicroPython 50
4.8 Class Receiver - MicroPython 51
4.9 Synchronization with NTP server for the latency experiment - MicroPython 52
4.10 Modification of Sender class for the latency experiment - MicroPython 52
4.11 Wireless ad hoc set up 54
4.12 Sensor data collection - Python3 55
4.13 Control experiment Raspberry Pi code - Python3 57
4.14 Paxos configuration file - Python3 58
4.15 Paxos experiment Raspberry Pi code - Python3 59
4.16 Raft experiment Raspberry Pi code - Python3 61
4.17 Receiver Raspberry Pi code - Python3 63
A.1 Sender code without consensus algorithm implementation- Python3 . . 94
A.2 Raft Sender code - Python3 97
A.3 Paxos Sender code - Python3 101

Chapter 1

Introduction

The goal of this master thesis is to carry out research on the leader-based distributed consensus algorithms, Paxos and Raft, and their role in achieving overall system reliability in the presence of a number of faulty processes or involving multiple unreliable nodes on the Internet of Things (IoT) network. IoT is a network of sensors and devices that can communicate and exchange data with each other through the Internet or long-range, low power, wireless platform (LoRa). As IoT importance increases annually since the first decade of 21st century, it is followed by the increase in the number of devices cooperating in the IoT network and an increase in the complexity of the communication infrastructure required to achieve flawless exchange of the data between the devices (Ismail, 2019). This constant connectivity necessitates a consensus mechanism that, once implemented, enables distinct nodes in an IoT network to agree on the authenticity and validity of transmitted data and increases the reliability of the distributed system (Bhattacharjee et al., 2017). Therefore, this research paper's main focus will be on implementing and maintaining distributed consensus among the IoT devices, and evaluating the performance of proposed algorithms in the presence of increased network latency due to faulty, or unreachable nodes. Additionally, we will attempt to compare the efficiency of both, Paxos and Raft, algorithms in decreasing packet loss rate and improving the system's stability.

1.1 Background

Nowadays distributed systems merge with the Internet of Things (IoT) creating a decentralized system of collaborating Smart Objects (SOs) which sense, store, and interpret information from the surrounding environment and act on their own, cooperate with each other exchange information with other kinds of IoT devices, or client nodes (Shi et al., 2016).

The assumption that IoT network demonstrates properties comparable to distributed systems (ICAR, 2020) opens the door to new solutions and implementations where Smart Objects cooperate together to achieve a common goal, akin to networked devices in the distributed system. Continuous attempts are made to design swarms of sensors that can encompass trillions of different devices capable of dynamically recruiting and disbanding additional resources (Lee et al., 2014), or to implement IoT across the urban areas creating SmartCity where, as Ji (2021) asserts, "centralized control method can neither reflect the spatial topology and physical constraints nor is it suitable for multi-sensor situations" leading, researchers involved, directly towards incorporating IoT into distributed systems paradigm (Xiaoyi et al., 2021).

Although distributed systems seem like a natural approach to the design and implementation of IoT in real-life scenarios (Chamoso et al., 2020), an issue arises when orchestration and synchronization of devices are attempted (Ahmad & Kim, 2020). Consensus is regarded as the primary problem which, once solved, will enable the implementation of a fault-tolerant distributed system (Lamport, 2005). Consensus algorithms enable the collection of machines to cooperate as a coherent group, able to survive the failures of some of its members. This feature warrants consensus algorithms to play a key role in building reliable large-scale distributed systems (Ongaro & Ousterhout, 2014) where the decisions are made by the conglomeration of devices acting as one system or service. Hence, distributed consensus algorithms, once adopted for IoT, will provide mechanism for balanced decision making on the edge nodes, and avoiding losing data from IoT devices in the presence of a number of malfunctioning devices by improving the robustness and reliability of the decision process (Li et al.,

2014).

Multiple authors recently focused on the idea of distributed consensus algorithms, their application in IoT domain (Bof et al., 2017; Fortino et al., 2020; Raghav et al., 2020; Whittaker et al., 2020; Zhang et al., 2020), and implementation of self-organizing real-time systems architecture (Guerrero et al., 2019; Shi et al., 2016; Tošić et al., 2019).

1.2 Research Project/problem

As mentioned by Chien et al. (2015), "there are four major components in application systems with internet-of-things (IoT): sensors, communications, computation and service, where a large amount of data are acquired for ultra-big data analysis to discover the context information and knowledge behind signals. Therefore, it is not feasible to employ centralized solutions on cloud servers, especially in the case of IoT. Thanks to the advances of silicon technology, the cost of computation becomes lower, and it is possible to distribute computation on every node in IoT." This approach leads to the integration and intertwining of both technologies, distributed computing and IoT and requires additional mechanisms focused on improving network throughput and reducing delays between the IoT nodes (Li & He, 2020).

A consensus algorithm is used to attain, in distributed and multi-agent systems, overall system reliability in the presence of a number of faulty processes or involving multiple unreliable nodes. When discussed in the context of IoT, a consensus algorithm is required to achieve fast event ordering, predictable delivery time and minimal packet loss rate in edge computing networks.

Current approaches to consensus between the nodes in IoT domain include modifying existing consensus algorithms: blockchain (Tošić et al., 2019; Wang et al., 2020), cooperative game model (Gulati & Kaur, 2020), proportional-integral-derivative (PID) (Shi & Yang, 2018), developing new algorithm (Castellano, Esposito, & Risso, 2019), or using revised Paxos consensus algorithm (Cachin, 2010; Poirot, Al Nahas, & Landsiedel, 2019).

In this paper, we will research the feasibility of implementing Paxos and Raft consensus algorithms in IoT networks and assess their effectiveness in enabling fault-tolerant distributed IoT. The following sections will introduce particulars relevant to distributed IoT networks and review both algorithms in detail.

1.2.1 Distributed System

A distributed system is a collection of independent devices that can be scattered across different locations, yet able to cooperate with each other and share the computing power of the cluster to accomplish shared objectives.

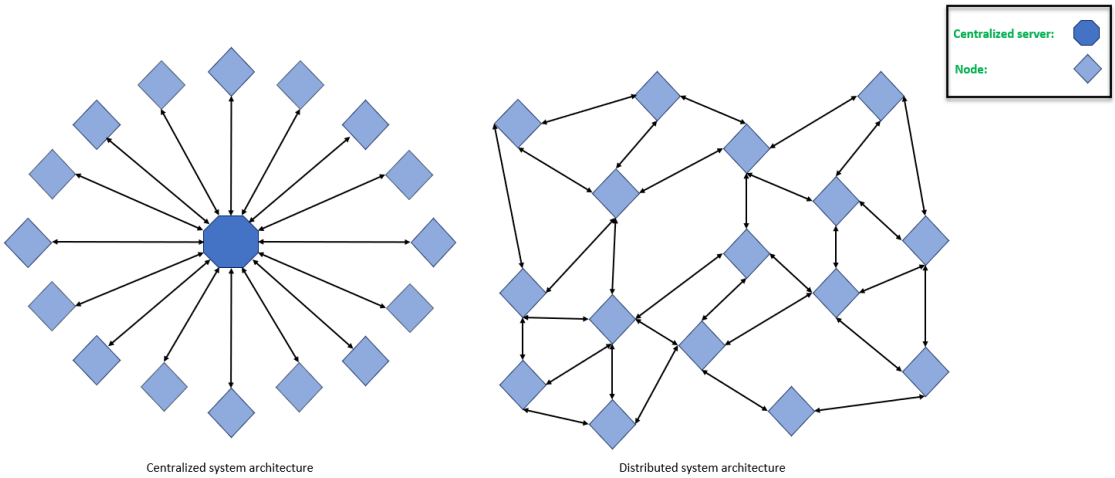


Figure 1.1: Centralized vs. Distributed system architecture
(Shyamala Devi et al., 2019)

For all intents and purposes, distributed system performs as a one interface or service and will appear as a uniform to the client process or end-user. Below properties outline typical characteristics of the distributed system (PUDER et al., 2006):

- Scalability

  Additional processing nodes are easily added to the cluster to improve the computing efficiency of the system.

- Loosely coupled

Nodes are communicating with each other using a standard protocol but have very little direct knowledge of each other.

- Concurrency

  Nodes that create distributed system run simultaneously.

- Heterogeneity

  The nodes and components of a distributed system can be of different types and designs which does not affect its efficiency or scalability.

- Replication

  The distributed system implements information and messaging sharing, ensuring consistency between the nodes affecting in improved reliability and accessibility.

- Availability and fault-tolerance

  Node failure does not lead to a system failure; the remaining nodes will continue to operate and attempt to complete the system's objectives.

The idea of distributed system translates directly to IoT domain (Chien et al., 2015) and its characteristics align with IoT network.

1.2.2 Byzantine vs non-byzantine distributed systems

Lamport et al. (1982) proposed a problem that describes an issue of communication and trust in distributed computing, the Byzantine Generals Problem. It shows how problematic is the achievement of a common objective when the identity and intentions of the collaborators are unknown and possibly hostile. Subsequently, the Byzantine network became synonymous with the system where some of the collaborating nodes are malevolent and unreliable and non-Byzantine became synonymous with the network of trustworthy and reliable nodes.

Both algorithms, Raft and Paxos described and explained further in this paper, are distributed consensus algorithms that are specifically developed for non-Byzantine systems and require modifications to operate in the Byzantine network.

1.2.3 Replicated State Machine

State machine replication (SMR) is a mechanism that is implemented to transform a cluster of unreliable hosts into a single fault-tolerant service that, for the purpose of client devices connecting to it, provides strong consistency and linearizability (Skrzypzcak & Schintke, 2020). Therefore, the correct implementation of SMR allows client machines to treat service as a single system and anticipate expected behaviour. Most importantly, replicated state machine stipulates that the state machine of each node in the distributed system receives exactly the same operations in the exact same order. This requirement can be guaranteed by achieving distributed consensus (István et al., 2016).

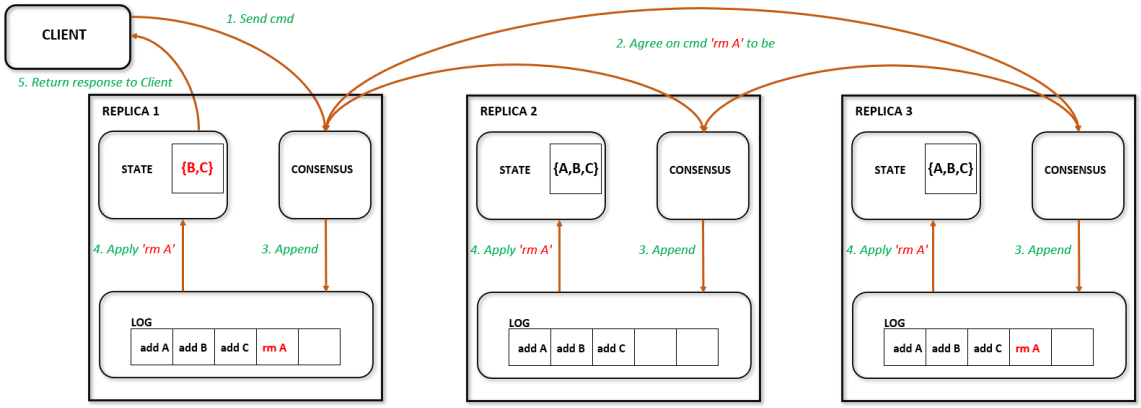


Figure 1.2: Replicated State Machine
(Skrzypzcak & Schintke, 2020)

1.2.4 Consensus

Li et al. (2014) classify IoT "as a network of networks, in which smart "things" are connected to the Internet via heterogeneous access networks and technologies (such as sensor networks, mobile networks, RFID, etc.) to provide services and applications" and similarly Oróstica and Núñez (2019) perceive it as "a sparsely coupled, distributed system of interacting smart objects, or things."

Both depictions of IoT imply that a certain level of autonomy is required for the

devices forming an IoT network to operate and avoid failures. The sheer amount of the nodes coupled with a constant flow of the data require a mechanism that can ensure stability and optimal performance of the system.

One of the proposed solutions to the above problem is distributed consensus, which can be described as a mechanism that allows a group of nodes on an unreliable network to reach an agreement (Kumari & Rohil, 2014). Consensus in distributed systems is necessary to achieve general system reliability in the face of multiple faulty nodes.

1.2.5 Paxos

1.2.5.1 Introduction

Paxos consensus algorithm was developed by Leslie Lamport in 1989 to solve the problem of fault-tolerant distributed systems, where collections of multiple separated devices can deterministically and safely agree on implementing certain actions under required conditions while guaranteeing the system's consistency if the conditions cannot be satisfied (Li, 2020). Often seen as an archetype of non-Byzantian consensus algorithms, Paxos is famous for being difficult to understand and hard to implement on distributed systems. This perception forced the author of the original paper to publish a series of explanatory articles where each step of the original algorithm was discussed and explained in plain English (Lamport, 1998, 2001, 2005).

1.2.5.2 Achieving Consensus

Roles

Paxos perceives distributed system and its behaviour as a collection of roles, where each device can assume more than one role in the cluster. Each node participating in achieving consensus can accept and perform the roles below:

- Client:

  The client is not a member of the distributed system. It issues a request to the cluster and expects a response. Paxos cluster is seen as a single coherent system to the client.

- Acceptor / Voter:

  Acceptor participates in the maintenance of fault-tolerant distributed storage in the cluster. Change to a state in a Paxos cluster cannot be committed until a quorum, a group of all acceptors, agree on it.

- Proposer:

  The proposer is a node that received a request from the client. It will attempt to persuade a quorum of acceptors to agree upon it.

- Leader:

  A Proposer who had a proposal accepted recently becomes a leader. Paxos requires that there is only one leader serving client requests. Only when the current leader fails, a new one selected and starts communicating with the client.

- Learner:

  Learners are responsible for processing the request from the client, once the proposal was accepted, and sending the result back to the client.

Phases

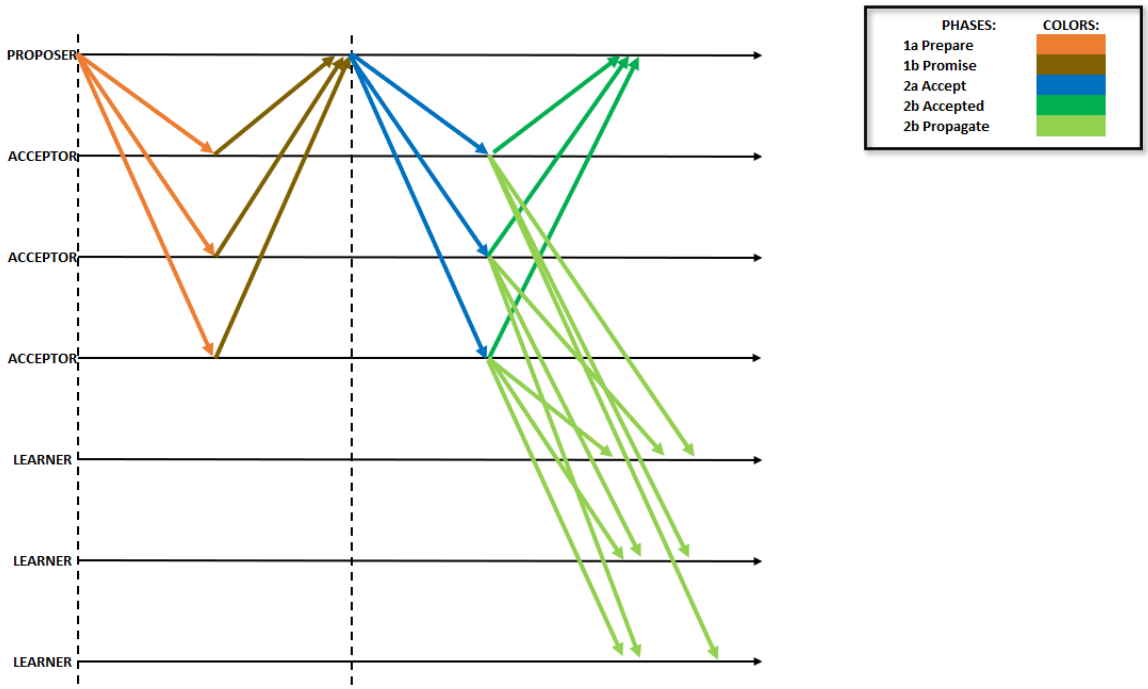


Figure 1.3: Paxos Phases

(Lamport, 2001)

Phase 1:

- 1a Prepare:

  Whenever the proposer in a Paxos cluster receives a request from the client, it sends a proposal message to update the replicated state machine with the new client request to the quorum for acceptance. Proposal message, called prepare request, is in the form of `Prepare(n)` where n is autoincrementing identifier therefore each following prepare request from a given proposer will have a guaranteed unique n higher than the previous one.

- 1b Promise:

  Each of the acceptors in the quorum compares the n-value of the message to the previously accepted "Prepare" messages and if the newly received n-value

of the "Prepare" message is higher, it sends the "Promise" message back to the proposer. If the Acceptor node accepted a different proposal message previously, it must then include the previous proposal n-value(m) and the corresponding value(v), in the "Promise" message returned to the Proposer (`Promise(n, (v, m)`), otherwise, if none were accepted previously it will be set to null (`Promise(n, null)`). This guarantees that the acceptor will not accept any other "Propose" messages with an n-value lower than currently accepted.

If the proposer receives promise messages from the majority of the quorum it attempts Phase 2 of the Paxos consensus algorithm.

Phase 2:

- 2a Accept:

  Once promises from the majority of acceptors are received, the proposer node has to set its own v-value. As acceptors must return any v-values previously accepted, the proposer will set its current "Propose" v-value to the value of the highest proposal number in any of the "Promises" received. Only when all promise messages are empty (`null`), the proposer can set its own v-value. Subsequently, accept message request is sent to the quorum of acceptors (`Accept(n,v)`).

- 2b Accepted:

  Upon receiving `Accept(n,v)` message, acceptors have to accept it unless they issued promise message with n-value higher than accept n-value. `Accepted(n,v)` message is sent to proposer node and learner nodes, and the new value becomes consensus and it's committed to the replicated state machine.

Once the majority of acceptors agrees on it, request is processed by the learner nodes, and the result is returned via proposer node to the client.

1.2.5.3 Electing leader

Paxos does not provide a leader election mechanism. It is advised to implement an algorithm responsible for choosing the leader, as Paxos might reach livelock when there is no progress made by the algorithm due to concurrent proposals from multiple nodes (Howard & Mortier, 2020). There are multiple solutions to leader election, one being the Bully algorithm proposed by Krzyzanowski (2018), where "a node that starts an election sends its server ID to all of its peers. If it gets a response from any peer with a higher ID, it will not be the leader. If all responses have lower IDs than it becomes the leader. If a node receives an election message from a node with a lower-numbered ID, then the node starts its own election. Eventually, the node with the highest ID will be the winner." It is important to note that Paxos' design is fault-tolerant and leader election is not required for achieving consensus, as proposals can be made by any of the Proposer nodes.

1.2.5.4 Failure prevention and recovery

Whenever a failure occurs in the cluster and the leader is unavailable, Paxos consensus algorithm assures that the nodes are won't stop sending proposals leading to the election of a new leader with the highest proposal's n-value. To resolve this issue Paxos might require multiple rounds of proposal messages sent back and forth leading to very inefficient network utilization and algorithmic efficiency.

1.2.5.5 Conclusion

Paxos algorithm attempted to solve consensus problems in non-byzantine distributed systems. It achieves its goal by strictly following two essential properties, as described by Lamport (2001):

- "No value ever reaches consensus without first being proposed and having its proposal accepted."
- "No two distinct values ever reach consensus at the same time."

1.2.6 Raft

1.2.6.1 Introduction

Raft distributed consensus algorithm is the result of the work of Ongaro and Ousterhout (2014) who attempted to create a leader-based consensus protocol that would be able to achieve similar performance and fault tolerance as Paxos but requires less complexity during implementation. Raft's design reflects the need for better understandability of distributed consensus and the steps required to achieve it. Paxos algorithm, often viewed as a predecessor of Raft and a gold standard for distributed consensus algorithms, presents a significant challenge for developers and system designers during system analysis and implementation phases (Ongaro & Ousterhout, 2014). Authors of the Raft protocol listed Paxos complexity and convolution as the main reasons for developing Raft.

1.2.6.2 Achieving consensus

Raft algorithm achieves consensus via an elected leader. Raft cluster is built of exactly one leader and multiple followers, and a node in a cluster can be either leader or follower (in case of unavailability of the leader, the follower becomes a candidate and undergoes the election process). Although all nodes in the distributed system can receive requests from the external clients, only the leader node is allowed to exchange data with the client and all requests received by follower nodes are forwarded to the leader.

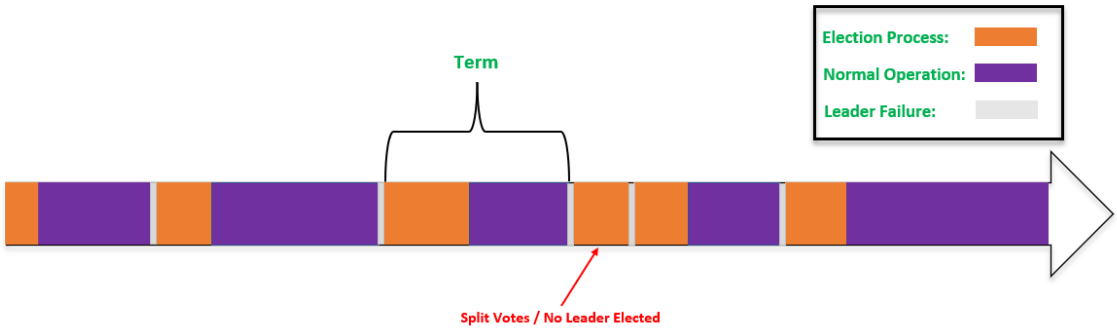


Figure 1.4: Raft Terms

(Ongaro & Ousterhout, 2014)

Term is a time unit used by Raft that can be of random length, and each of the terms is given a consecutive number. Each leader is chosen during the election process that marks the start of a new term. If, during the election, no leader is chosen Raft will restart the election process effectively starting a new term. This step guarantees that during the term there is only one leader in the cluster. It is important to note that terms are used in Raft protocol context as a distributed system's logical clock and, as each node stores the current term number, it allows to detect outdated data on the participating nodes. As mentioned by Ongaro and Ousterhout (2014), "current terms are exchanged whenever servers communicate; if one server's current term is smaller than the others, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to the follower state. If a server receives a request with a stale term number, it rejects the request."

1.2.6.3 Electing leader

Raft utilizes heartbeat messages to implement the leader - followers relationship, where each of the followers expects heartbeat from the leader within the so-called election timeout frame. Each node in the cluster starts as a follower and expects a valid heartbeat message from the leader node. The election process starts when a message is not received within the time limit, otherwise, the election timeout is reset (Ongaro & Ousterhout, 2014).

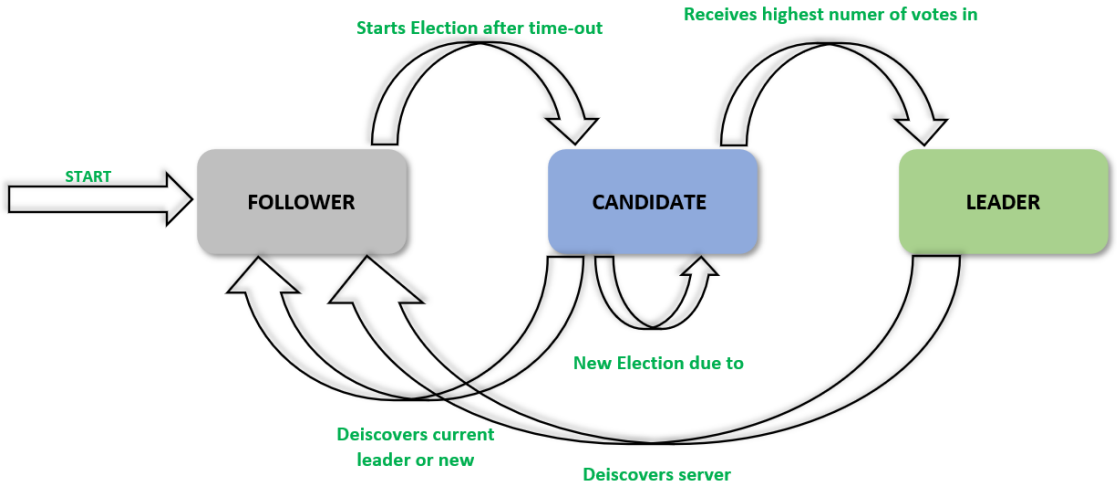


Figure 1.5: Raft election process
(Ongaro & Ousterhout, 2014)

The election starts with a follower transiting to the candidate by incrementing its current term number. This is followed by a `RequestVote` heartbeat message and voting for itself. All follower nodes in the cluster are expected to participate in the election process in parallel, and they remain in the candidate state until one of them wins the election or previously established time passes without selecting the leader. Immediately after the leader is elected, it broadcasts the heartbeat message to the entire cluster which stops the election process.

1.2.6.4 Log replication

The leader node's main responsibility is to communicate with the clients and fulfilling their requests. Every one of those requests is stored within the leader's logs and once the leader updates its log with a new entry, it will send an `AppendEntries` heartbeat message to the followers and expects that this command will be replicated within the cluster. The result is then returned to the originator after followers confirm the append request and the message is applied to the leader's state machine. It is the leader's sole responsibility to update the state machine with new entries and every commit is stored along the term number ensuring its validity. In their paper, Ongaro and Ousterhout

(2014) indicate that Raft is capable of achieving a high level of consistency among the cluster's logs by implementing the properties below:

- "If two entries in different logs have the same index and term, then they store the same command."
- "If two entries in different logs have the same index and term, then the logs are identical in all preceding entries."

1.2.6.5 Failure prevention and recovery

Raft consensus protocol implements a failure prevention and recovery mechanism where, in case of the inconsistency between the leader's and follower's logs, the follower will replicate the leader's entries by overwriting its own. The process of checking and replicating logs is embedded within `AppendEntries` messages and it does not require extra steps from the leader node, as it is automatically executed from the moment of a successful election.

1.2.6.6 Conclusion

Raft has an advantage over the other consensus algorithms, noticeably Paxos, that its understandability is greatly improved (Howard, 2014) and its design is optimized for systems design and architecture (Ongaro & Ousterhout, 2014). Authors of the Raft protocol focused on strong leadership and the leader's decisive role in achieving distributed consensus. As of the date of writing this thesis, there is unsubstantial work on the Raft algorithm being used in distributed IoT architecture for achieving consensus among the edge nodes.

1.2.7 Paxos vs Raft

Both previously described consensus protocols are sharing similar properties and follow similar assumptions, i.e.:

- One of the nodes in the cluster is the leader, who accepts requests from clients and sends log entries to the follower nodes.

- A new entry is committed and once accepted, the state machine is replicated in the cluster.
- A new leader is elected by the majority as soon as the previous leader is unavailable (requires additional implementation in Paxos algorithm).

Most importantly, Paxos and Raft satisfy by design requirement mentioned by Howard and Mortier (2020):

- "Theorem3.1 (StateMachine Safety). If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.
- There is at most one leader per term and a leader will not overwrite its own log so we can prove this by proving the following:

  Theorem 3.2 (Leader Completeness). If an operation op is committed at index i by a leader in term t then all leaders of terms > t will also have operation op at index i."

	Paxos	**Raft**
How does it ensure that each term has at most one leader?	A server s can only be a candidate in a term t if t mod n = s. There will only be one candidate per term so only one leader per term.	A follower can become a candidate in any term. Each follower will only vote for one candidate per term, so only one candidate can get a majority of votes and become the leader.
How does it ensure that a new leader's log contains all committed log entries?	Each RequestVote reply includes the follower's log entries. Once a candidate has received RequestVote responses from a majority of followers, it adds the entries with the highest term to its log.	A vote is granted only if the candidate's log is at least as up-to-date as the followers. This ensures that a candidate only becomes a leader if its log is at least as up-to-date as a majority of followers.
How does it ensure that leaders safely commit log entries from previous terms?	Log entries from previous terms are added to the leader's log with the leader's term. The leader then replicates the log entries as if they were from the leader's term.	The leader replicates the log entries to the other servers without changing the term. The leader cannot consider these entries committed until it has replicated a subsequent log entry from its own term.

Table 1.1: Summary of the differences between Paxos and Raft

1.3 Research Objectives

The aim of this research is to implement, analyse and evaluate the performance of the Raft consensus algorithm on IoT networks in the presence of multiple node failures.

As current research on the topic of consensus problems in IoT domain does not discuss, nor benchmark Raft, the author will attempt to provide performance measurements (latency and packet loss rate) in the presence of different numbers of malfunctioning devices. Additionally, the results of the above experiment will be compared against the results of Paxos algorithm and IoT network without algorithm implemented. Following research objectives would facilitate the achievement of this aim:

- Constructing IoT network of three, five and seven Raspberry PI 3 Model A+ single-board computer and DockerPi Sensor Hub Development Board (EP-0106) and one Raspberry Pi 3 B acting as an IoT gateway.
- Implementing Paxos and Raft consensus algorithms on IoT network.
- Evaluating performance of proposed IoT network in the presence of multiple node failures when:
  - No consensus algorithm is implemented
  - Paxos consensus algorithm is implemented
  - Raft consensus algorithm is implemented
- Analysing the results of the experiments and providing recommendations for consensus implementation in IoT networks.

We will conduct an experiment where original (non-existent) data will be collected, and different consensus algorithms will be compared to confirm if the implementation of Raft consensus algorithm in IoT network can decrease packet loss rate and latency of the network in the presence of malfunctioning edge nodes when compared to Paxos consensus algorithm?

Quantitative (numerical) data from the experiment will be collected and analysed to provide a definitive result. Exhaustive answers will be provided by comparing quantitative data that will test and confirm an existing theory.

1.4 Research Methodologies

Following on different classification of research methodology (Wohlin & Runeson, 2021), author recognizes that this research can be characterized in four distinct ways:

- By type: Primary research

  An experiment will be conducted where original (non-existent) data will be collected. The generation of the data will begin as soon as the IoT nodes are powered up and the sensors start transmitting environmental information over the LoRa to gateway node. During the control experiment Packet Loss Rate (PLR) and latency between the client and edge note, and packet rate will be measured in IoT network. A control experiment will be repeated for each consensus algorithm, Paxos and Raft.

- By objective: Quantitative research

  Quantitative (numerical) data from the experiment will be collected and analysed and provide a definitive result. T-test with paired samples will be used to statistically compare performances of consensus algorithms during both experiments with different conditions.

- By form: Empirical research

  The answers will be provided by comparing quantitative data by statistically comparing all data collected during control experiment algorithms performances during research experiments.

- By reasoning: Deductive research

  Author will test and confirm an existing theory. Proposed hypotheses H1, H2, H3 and H4 will be accepted independently based on the results achieved. Confirmation of hypotheses will confirm the research question.

1.5 Scope and Limitations

The scope of this research is to implement Paxos and Raft consensus algorithms on the IoT network where we will measure the algorithms' performance. As consensus algorithms are necessary for achieving reliability in the distributed network (Lamport et al., 1982), this thesis will attempt to determine if the implementation of consensus algorithms in IoT networks will decrease Packet Loss Rate and latency on the network in the presence of malfunctioning nodes. Additionally, the findings will help to determine if the Raft consensus algorithm achieves better results than Paxos in attaining reliability on IoT networks.

Important limitation of this research is the size of the IoT networks used in the experiments. We will create and execute experiments based on three types of IoT networks, i.e., 3-node, 5-node and 7-node networks. The aforementioned network sizes were chosen based on the Paxos and Raft requirement of a minimum number of nodes to achieve consensus `(N/2 + 1)`. Although either of the proposed networks should achieve consensus in the presence of faulty nodes, this experiment would benefit greatly if the implementation of both algorithms was carried out on a larger scale, both from a network size and geographical location point of view. Additionally, Paxos is allegedly a difficult algorithm to implement on a distributed system (Ongaro & Ousterhout, 2014) which might impede the course of this research.

1.6 Document Outline

This research paper is divided into five chapters where each chapter clearly defines the scope and objectives.

In Chapter One, an outline of the research problem is provided together with an environment where the experiments will be conducted. Additionally, Chapter One describes research objectives and research methodologies utilized and clearly defines scope and limitations of this study.

Chapter Two presents the most up to date research on the topic of consensus in dis-

tributed systems and delivers comprehensive review of the work carried out in the consensus field with clearly defined research gaps.

Chapter Three describes the design of the experiments carried out to answer the research question.

Chapter Four contains the detailed description of each experiment conducted. It also provides the technical specifications of the IoT networks implemented and the snippets of the code with its functionality described in the detail.

Chapter Five discusses results of experiments performed.

Chapter Six is the final chapter that concludes the research work presented in this master thesis.

Chapter 2

Review of existing literature

2.1 Introduction

This section provides an overview of the existing research and literature review that attempts to solve consensus conundrums in the IoT domain.

When mentioning consensus algorithms in the context of IoT, we have to clarify that their task is to enable the collection of machines to cooperate as a coherent group, able to survive the failures of some of its members. This feature warrants consensus algorithms to play a key role in building reliable large-scale distributed systems (Ongaro & Ousterhout, 2014).

As presented further in this section, multiple authors recently focused on the idea of distributed consensus algorithms, their application in IoT domain (Bof et al., 2017; Fortino et al., 2020; Raghav et al., 2020; Whittaker et al., 2020; Zhang et al., 2020), and implementation of self-organizing real-time systems architecture (Guerrero et al., 2019; Tošić et al., 2019; Yi, Hao, Qin, & Li, 2015).

Distributed consensus algorithms, when adopted for IoT, provide a mechanism for balanced decision making on the edge nodes and avoiding losing data from IoT devices in the presence of several malfunctioning devices by improving the robustness and reliability of the decision process (Li et al., 2014).

Consensus algorithms are the forerunner of distributed leadership and are used for

solving problems as diverse as distributed task assignment, distributed estimation and distributed optimisation. The review given below presents broad spectrum of most current works related to the subject of this paper, the consensus in the IoT infrastructure.

2.2 Related works

Tošić et al. (2019) propose a decentralized self-balancing architecture that implements a consensus algorithm intending to improve the usage of all IoT devices on the network and resilience to disconnection while providing increased privacy and security. Authors focus on minimising, and hopefully avoiding race conditions when migrating applications that execute on IoT devices. Their solution amalgamates data structure based on blockchain technology with a consensus algorithm using leader election. While authors do not mention a specific consensus algorithm, they claim that any leader election-based consensus algorithm can be used interchangeably in their proposed design. Their solution requires the Docker platform running on an IoT node, increasing the resources and computing power needed for the IoT device. Docker requirement adds additional constraints on the described IoT network architecture, as it excludes traditional IoT sensors, which lack computing power and memory to ensure flawless Docker environment execution. Hence, according to Tošić et al. (2019) Blockchain-based decentralized self-balancing architecture for the web of things is mostly suited for more powerful IoT Edge nodes rather than IoT sensors. Moreover, in their work, authors focus on a small number of devices and miss the opportunity to test their architecture with different consensus algorithms against multiple use cases.

On the other hand, Li et al. (2014) focus on the distributed consensus decision making (CDM) for services over IoT. Authors design a distributed consensus algorithm with the goal of improving decision making at the edge nodes of IoT, with a focus on the global consensus of multiple services. Their approach to the architecture and design of the IoT network is based on the service-oriented architecture (SOA) where each IoT node exposes resources as services, easily accessible by external systems. In author's

words, "service-oriented IoT can thus control, manage, and interact with the real world by means of 'services,' which enable bi-directional user-to-object information exchange and interaction" (Li et al., 2014). Existing challenges to the implementation of service-oriented IoT are clearly identified and consequently solved during the course of research. As a result, Li et al. (2014) propose partitioning of the IoT network where each partition creates a cluster with a leading node. Each of the clusters is responsible for achieving local consensus and once it is established, the leading node exchanges consensus with other leading nodes achieving global consensus. The authors conclude that the results of their research prove the proposed solution's effectiveness and performance. As per the authors' suggestion, more focus should be given to research on collaborative methods for the realization of information exchange and resource allocation in an IoT environment.

Similar attempts to develop asynchronous Distributed Resource AssiGnment and OrchestratioN (DRAGON) algorithms are presented in the paper authored by Castellano et al. (2019). Although DRAGON is not specifically designed to address IoT consensus mechanism, its goal is to provide a solution to the `NP-hard Problem` in regard of resource allocation in edge computing. Authors attempt to maximize the optimization strategy of the applications demanding resources from shared edge infrastructure, i.e., distributed content delivery and caching or Internet of Things, by implementing their algorithm, as an agent, on each application competing for resources. The voting and election process is held on the node level, and once the application residing on the node is elected, it receives the required resources. Ultimately, the goal of the research is to allow multiple applications to coexist in an environment with shared infrastructure and limited resources. Authors do not specifically focus on IoT domain, and yet their approach and algorithm should not require a multitude of modifications to be adapted to the IoT network. Castellano et al. (2019) test the proposed algorithm within the scope of two use cases focusing on its convergence and performance properties only and miss the opportunity to implement DRAGON across multiple different domains.

Although Hao, Yi, and Li (2018) focus their research on the edge computing networks, in a similar fashion to Castellano et al. (2019), their approach aims to resolve fast event

ordering issue for large-scale delay-sensitive distributed applications. Authors propose a novel consensus protocol, EdgeCons, that implements Paxos algorithm. Its instances run on the edge nodes and it is using recently running history of the consensus process to distribute the leadership of the instances. EdgeCons additionally uses the cloud infrastructure behind the edge network to establish a reliable backend and as a result, warrants the continuance of the consensus process. Similarly, to DRAGON algorithm, EdgeCons is a promising consensus that can be applied in IoT domain, and although primary results are very promising authors have to design proper monitoring of the system. Additionally, the timeframe of recovery of EdgeCons "leadership share" can take an interminable amount of time.

Gulati and Kaur (2020) present a different approach, to previous authors, in their pursuit of implementing Weighted Voting Game (WVG) model for conflict resolution in argumentation enabled social IoT networks. As Social IoT (SIoT), or "speaking and hearing" IoT sensors, are very recent ideas, one of the most pressing issues within this type of IoT network is consensus among the nodes. The design of SIoT is based on human interactions and as this allows for faster and more flexible connections between the nodes, it creates an issue of negotiating the existing resources or achieving an agreement to resolve the conflict. Gulati and Kaur's algorithm is based on game theory due to its multidisciplinary recognition in conflict resolution and decision making. Although the results of the experiment show over 35% improvement in throughput in the SIoT network, the authors simulate and evaluate the proposed model's performance within one use case scenario. Finally, the authors notice that implementing the model in a multi-agent environment to enable agreement among conflicting agents is very promising for future work.

The concept of SIoT is discussed in Yogesh, Patel, and Patel (2022) paper as well, where the authors examine the phenomenon of convergence in IoT networks under heavy data load from sensing devices. The assumption of heterogeneity of devices connected to SIoT network, made by authors, implies that certain steps of friendship establishment will have to be completed for the nodes to create SIoT. Authors however do not propose specific consensus algorithm to establish and manage social relation-

ships in IoT, they conclude that implementing one should be the future research goal instead.

Oróstica and Núñez (2019) discuss distributed multi-cast algorithms for robust average consensus over the internet of things environments. Authors implement their algorithm in the transport layer where they make use of UDP protocol and IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN). By using multi-cast and UDP, Orostica and Nunez make it clear that their focus is on low resource consumption and speed rather than reliability. Interestingly, their research is based on an assessment involving 280 agents and multiple experiments conducted over the public network spanning Paris, Saclay, Grenoble and Strasbourg. The scale of their experiment with the use of 6LoWPAN makes this research quite significant from the point of view of the author of this paper, as it's conducted in an environment close to real-life conditions with the use of technologies favoured by the author. The proposed algorithm is tested and compared along with the Push-Sum-Based algorithm introduced by Bof et al. (2017), the robust Average Consensus algorithm (ra-AC), that implements asynchronous communication between the nodes on the unreliable network systems but in contrast to Orostica and Nunez algorithm, ra-AC was tested in simulated scenarios. Orostica and N´u˜nez conclude that their algorithm outperforms Bof et al. algorithm and achieves average consensus over IoT network but they miss the opportunity to compare it with additional consensus algorithms.

Bahreini and Grosu (2017) present different approach to solve the consensus problem among the distributed agents. Authors discuss the problem of distributed placement algorithm for ad-hoc optimization in Mobile Edge Computing (MEC). In their work, Bahreini and Grosu develop Mixed Integer Linear Program (MILP) algorithm for solving the online version of the placement problem, based on an iterative matching process. MILP achieved good performance during experiments and although results presented in the paper are very promising, authors unfortunately did not attempt to implement proposed algorithm in 'real-life' multiple scenarios and focused on online single use case simulation only.

It is worth mentioning the work of Cui, Yang, and Liu (2020) on positive edge consen-

sus of nodal networks, where the authors presented a consensus algorithm addressing the issue of consensus in complex networks. It is not directly related to IoT environments but has the potential as "computationally cheaper, especially for large-scale systems".

Another interesting approach not directly related to the consensus mechanism is the work of Rocha and Brandão (2019). Authors propose a successful model of scalable architecture for discovering and managing a group of IoT devices with an aim to design and implement Multiagent Architecture for Taming e-Devices (MATe) which attempts to solve discovery capabilities on IoT networks. Whereas their model is successful, it does not take into account the impact of devices' energy consumption or the increase in number of nodes connected.

Salimitari, Chatterjee, and Fallah (2020) paper focuses on resource-constrained IoT devices and networks. Authors recognize the importance of consensus mechanisms that enable IoT nodes to operate and reach agreements without the influence or input of the central controller. Although they recognize that Blockchain-based consensus algorithms require enormous computational power, Salimitari et al. attempt to adapt known Blockchain technology in IoT networks and compare multiple blockchain-based consensus methods. In their research paper, the authors set out 24 consensus protocols and determine their suitability for IoT domain based on, in particular, computing, network overhead and storage overhead. Salimitari et al. arrive at the conclusion that only 3 consensus protocols are wholly suitable for the IoT domain, and 10 are partially suitable out of the group of 24 tested protocols. In the pool of tested protocols, RAFT is one of the protocols that are deemed to be partially suitable, unfortunately authors omit RAFT's network overhead and point out it's overreliance on the voting process. During the process of testing and comparing different consensus mechanisms, Salimitari et al. did not achieve implementation satisfying all challenges listed in the paper and they conclude that although they were able to address some of issues with the resource constrained IoT networks, the best approach might require redesign and development of the new consensus algorithms.

Despite IoT not being the main subject of his research, Gramoli (2020) touches upon

the topic of consensus in the Blockchain context. In his research, the author adapts the Byzantine Fault (BFT) consensus algorithm for blockchain technology and distributed systems. The algorithm, practical BFT (pBFT), developed by Gramoli pursue the goal of achieving consensus among the Blockchain nodes in respect of adding a unique block of transactions without creating an unsolvable conundrum. The author summarizes existing approaches to the consensus issue, listing their advantages and disadvantages. Gramoli meticulously dissects the algorithm and investigates it whenever an advantage is found while improving his own work. One of the qualities important for Gramoli, is the security of the consensus algorithm. Significant part of his work is focused on the security of the Blockchain transactions and possible ways of improving it while implementing consensus solutions. Granoli's algorithm achieves a clear consensus in each round of operation and cannot be manipulated. As previously mentioned in Salimitari et al. (2020) work, Blockchain consensus algorithms might be difficult to implement in IoT domain due to their high resource requirements. Nonetheless, Gramoli's paper is an interesting study of the strengths and weaknesses of consensus protocols.

Qin, Ma, Shi, and Wang (2017) present a survey of advances in Multi-Agent Systems (MAS) consensus. The authors discuss multiple approaches to solving distributed coordination in the extensive context of MAS, with the application of consensus algorithms being one of the chosen solutions. Although Quin et al. admit that the consensus protocols in the MAS environment are mostly related to systems and controls, those can be easily transferred into other domains, including IoT. The paper's main focus is on a control perspective rather than decentralised consensus. The authors point out that the "survey is far from an exhaustive literature review, and there may still be some crucial results missing in the study due to space limitation ".

Similarly, Zhang, Liu, Wang, Liu, and Ferrese (2015) and Yang, Wu, Sun, and Lian (2016) propose distributed algorithms that are utilising MAS properties (consensus and invariant summation of state variables) for the Economic Dispatch Problem (EDP). Zhang et al. approach focuses on finding a distributed control solution Unfortunately, the proposed algorithm does not optimise transmission losses. On the other hand, Yang et al. work aims at an algorithm able to achieve consensus in the minimum

possible time. Both attempts are not strictly venturing into the IoT area, but their results could have interesting applications for IoT architecture.

Mondal and Tsourdos (2020) discuss another approach to achieving consensus in the Multi-Agent Systems by using the Genetic Algorithm (GA) technique to achieve an optimal adjacency matrix. Their main focus is on solving the optimal network topology problem in MAS that aids in lowering energy consumption during consensus negotiations. The authors concluded that their research could be extended by including more agents in the experimental network by changing the dimension of the adjacency matrix.

Similarly, Fortino et al. (2020) work focuses on MAS but with an IoT-centric approach. In their paper, the authors view IoT infrastructure as an association of software agents, constantly collaborating and competing to process data and complete compound tasks. During the research, Fortino et al. noticed that this approach to organising IoT infrastructure favours a more aggressive approach than anticipated. Therefore, the Friendship and Group Formation (FGF) algorithm, based on a meritocratic mechanism, is developed with organisation and improvement of IoT objects collaboration in mind. FGF draws from the Game Theory, similarly to Gulati and Kaur (2020), sharing the idea of social constructs able to form friendships and groups to achieve consensus by creating social capital. In other words, FFG achieves trust and consensus by rewarding competency and honesty among software agents, forming a community within the IoT infrastructure. Authors demonstrate the positive effect of the FGF algorithm on the IoT domain where agents form groups based on a meritocratic mechanism. However, they could extend their research by implementing additional use cases with multiple agents cooperating

Al-Doghman, Chaczko, and Brookes (2018) introduce consensus techniques that provide complete information about network status by implementing one case study using the consensus aggregation within Fog environments. Authors implement Bayesian analysis and Markov Chain techniques to amplify and improve the quality of the information exchanged between nodes in the IoT network. Their work analyses and tests the impact of consensus algorithms on data aggregation, which translates Quality-of-

Service (QoS) directly. Al-Doghman et al. propose two groups of algorithms. The first one focuses on detecting abnormal nodes using short-term analysis (Bayesian analysis). The second group implements algorithms to detect unusual activity patterns using long-term analysis (Markov chain). Models for both groups are developed during the Calibration Phase preceding the experiment. As the amount of data created in the IoT network is significant, the authors conclude that implementing intelligent aggregation of data using proposed consensus algorithms can remarkably improve data quality sent over the IoT network. Additionally, Al-Doghman et al. note that when increasing the number of nodes in the IoT network in conjunction with Bayesian analysis, QoS increased significantly.

Abdelwahab, Hamdaoui, Guizani, and Znati (2015) developed two algorithms for distributed sensing resource discovery in IoT with their main focus on gossip policy. Authors formulate the idea of a Cloud of Things for Sensing as a Service (CoTSasS), or a global architecture based on cloud computing and IoT sensors that is highly scalable and managed by the cloud interface. CoTSasS implements flow of data gathered by sensors as tasks that can be defined by the user. One of the algorithms proposed, Sensing Resource Discovery (SRD), utilises gossip algorithm, in a manner similar to Fortino et al. and Gulati and Kaur, where software agent casts received a task to the neighbouring nodes using the gossip protocol. Although this paper does not discuss the implementation of consensus protocols, it is essential to note that it researches in-depth remote discovery, which is a vital part of achieving consensus among distributed nodes.

Similarly to Tošić et al. and Salimitari et al., Raghav et al. (2020) propose Blockchain-based proof of elapsed work and luck (PoEWAL) consensus algorithm for satisfying security requirements (e.g., integrity, authentication, and availability) on IoT devices. Their algorithm addresses high energy consumption and high computational requirements of similar consensus protocols, e.g., proof of work, proof of stake, or proof of authority. Although PoEWAL was compared with other consensus algorithms regarding consensus time, energy consumption, and network latency, the authors missed the opportunity to make it more complete by not including scalability, computing require-

ments and decentralisation features.

Tree-chain, a fast and scalable consensus protocol based on blockchain's hash function output, is another attempt at implementing blockchain-based consensus algorithms in the IoT domain. Dorri and Jurdak (2020) recognise IoT restrictions in regard to computing capabilities, the time required to achieve consensus and energy limitations, etc., that create a barrier to implementing traditional blockchain consensus algorithms. To amend those issues while allowing for achieving fast and reliable consensus, the authors propose Tree-chain, an algorithm that randomises validators for committing transactions using hash function output. Validators are responsible for executing transactions on the IoT network to blocks, and Tree-chain, in contrast to existing Blockchain protocols, minimises the computational power required for the leadership process. In conclusion, Dorri and Jurdak notice that Tree-chain considerably reduces computational requirements for achieving consensus and commitment latency.

Whittaker et al. (2020) compare multiple consensus protocols (e.g., Paxos and Raft) and implement them in the IoT universe. Their work is a comprehensive study on implementing distributed consensus algorithm in the IoT domain. The authors present multiple flavours of the Paxos algorithm, listing their pros and cons along with the RAFT algorithm. Whittaker et al. research is focused on applying proactive reconfiguration in elastic systems while achieving distributed consensus. They design and contribute two algorithms based on Paxos, i.e., Matchmaker Paxos and Matchmaker MultiPaxos, that achieve reconfiguration of the nodes without performance degradation.

Howard (2014) considers the RAFT algorithm's characteristics and its presumed simplicity compared to Paxos. Authors meticulously dissect RAFT and its internal workings and attempt to replicate results achieved in Ongaro and Ousterhout (2014) paper and propose additional improvements and optimisations to the protocol. The authors achieved their goal and were able to prove that Raft is understandable and easily implementable algorithm. Unfortunately, Howard limited their research to a network simulation framework and missed the opportunity to implement it in the distributed system over an unreliable network. The authors did not attempt to measure packet

loss or test cluster scaling in dynamic network topologies, which could significantly improve their findings. However, their results align with the general consensus that the Raft algorithm is an understandable and efficient distributed consensus algorithm.

Similarly, Howard and Mortier (2020) attempt to establish which of the algorithms, Paxos or RAFT, is more suitable for achieving consensus in a distributed environment. Their research focuses on the state machine replication problem, which heavily relies upon multiple independent agents reaching consensus and cooperating without failure. Howard and Mortier chose both algorithms based on their leader-based approach to solving the problem of distributed consensus. Paxos and RAFT are tested and judged during the research based on multiple criteria, e.g., handling leader failures, safety, understandability or efficiency, and a complete evaluation of both is presented and described in detail. Notably, the authors notice that although both algorithms give a similar level of complexity that may affect their undesirability, RAFT excels in leader-election and significantly outpaces Paxos in that matter. Research presented by Howard and Mortier is an interesting approach to solving consensus problems. However it's lacking in the real-life scenario use-cases and missing the chance of implementing both algorithms in any distributed environment, be it MAS or IoT.

2.3 Conclusion

Although there is an extensive body of research in consensus algorithms and their application for IoT architecture, the author of this paper notices that not all avenues are adequately explored. One joint gap that emerges in the research is the lack of the 'real-life' scenarios (Bahreini & Grosu, 2017; Gulati & Kaur, 2020) and an inadequate number of use cases tested (Castellano et al., 2019; Fortino et al., 2020; Mondal & Tsourdos, 2020; Tošić et al., 2019). Another critical issue is the absence of proper monitoring (Hao et al., 2018) and missed opportunity to discuss consensus with nodes scalability in mind (Li et al., 2014; Raghav et al., 2020; Rocha & Brandão, 2019). While some of the papers discussed provide a good comparison of multiple algorithms, most fall short from the comprehensiveness perspective (Abdelwahab et al., 2015;

Oróstica & Núñez, 2019; Salimitari et al., 2020). One avenue of particular interest is the implementation of the Raft consensus algorithm to minimise latency and packet loss on IoT networks in the face of multiple node failures and contrast the results with those of Paxos and pBFT algorithms (Gramoli, 2020; Whittaker et al., 2020).

Chapter 3

Experiment design and methodology

3.1 Introduction

This section presents an overview of the experiments and components required to conduct them, it also outlines methodologies used to verify and analyse the results of the experiments. Following subsections will cover the aim of research, the hypothesis, the design and implementation of the experiment and data output design. This chapter will conclude with design conclusions.

3.2 Aim of Research

The objective of this research is to implement, analyse and evaluate performance of the Raft consensus algorithm on IoT network in the presence of multiple node failures. As current research on the topic of consensus problems in IoT domain does not discuss, nor benchmark Raft, the author will attempt to provide performance measurements (packet loss rate and latency) in the presence of several malfunctioning devices. Additionally, the results of the above experiment will be compared against the results of the Paxos algorithm.

3.3 Hypotheses

Null hypothesis H0:

Implementation of the Raft consensus algorithm will not improve the packet loss rate or latency of the IoT network in the presence of faulty nodes.

Alternative hypothesis H1:

Implementing the Raft consensus algorithm will decrease the packet loss rate in the IoT network.

Alternative hypothesis H2:

Implementing the Raft consensus algorithm will improve the latency of the IoT network in the presence of faulty nodes.

Alternative hypothesis H3:

Implementing the Raft consensus algorithm will decrease the packet loss rate in the IoT network in the presence of faulty nodes.

Alternative hypothesis H4:

Implementation of the Raft consensus algorithm will outperform Paxos consensus algorithm.

3.4 Design overview

Hypotheses for this research require focussing on two aspects of data transmission in IoT networks, i.e., packet loss rate and latency. To collect data required for confirming (or rejecting) the above hypotheses, the author will perform two experiments divided into sub-experiments that will generate sets of numerical data.

The first experiment will focus on measuring packet loss rate in IoT networks with LoRa connectivity, and will simulate conditions within the networks in the presence of faulty IoT nodes. During this experiment author plans on performing additional sub-experiments with varying conditions and different network set ups described further below. First experiment aims to collect packets sent from each IoT node and compare

their number with packets received by the IoT Gateway node where calculated packet loss ratio (ratio of the number of lost packets to the total number of sent packets) will provide sufficient evidence to prove or disprove hypotheses.

During the second experiment, focused on measuring latency, each packet will be timestamped when send from transmitter node and another timestamp will be added upon receival in receiver node. Both experiments will generate numerical data, with additional timestamps added in second experiment. Similarly, to experiment measuring packet loss, author will repeat all sub-experiments with different network set ups and scenarios.

Packet Loss Rate Experiments						
Algorithm	**Experiment 1**	**Experiment 2**	**Experiment 3**	**Experiment 4**	**Experiment 5**	**Experiment 6**
Raft	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail
Paxos	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail
No Algorithm	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail

Table 3.1: Packet Loss Rate Experiments

Latency Experiments						
Algorithm	**Experiment 1**	**Experiment 2**	**Experiment 3**	**Experiment 4**	**Experiment 5**	**Experiment 6**
Raft	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail
Paxos	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail
No Algorithm	3 nodes	3 nodes, 1 fail	5 nodes	5 nodes, 2 fail	7 nodes	7 nodes, 3 fail

Table 3.2: Latency Experiments

Building on the work of Blenn and Kuipers (2017), we will implement a limit of 1000 packets to be sent from transmitter nodes to the receiver node during both experiments, with additional decreased number of packets set to simulate nodes' failures.

3.4.1 IoT networks

Test use cases are planned to be implemented during the first and second control experiment where three networks are created: A, B and C.

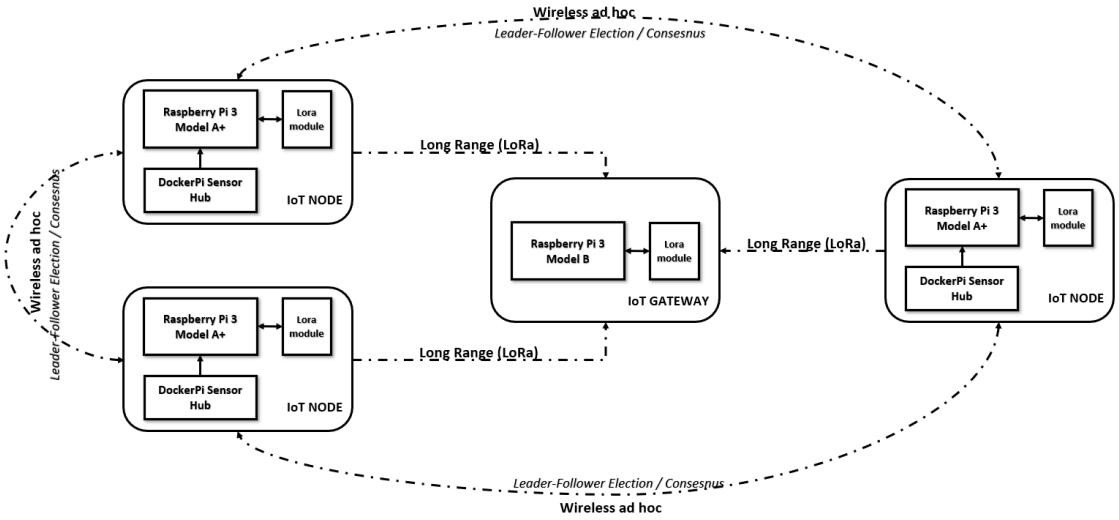


Figure 3.1: Network A - 3 nodes

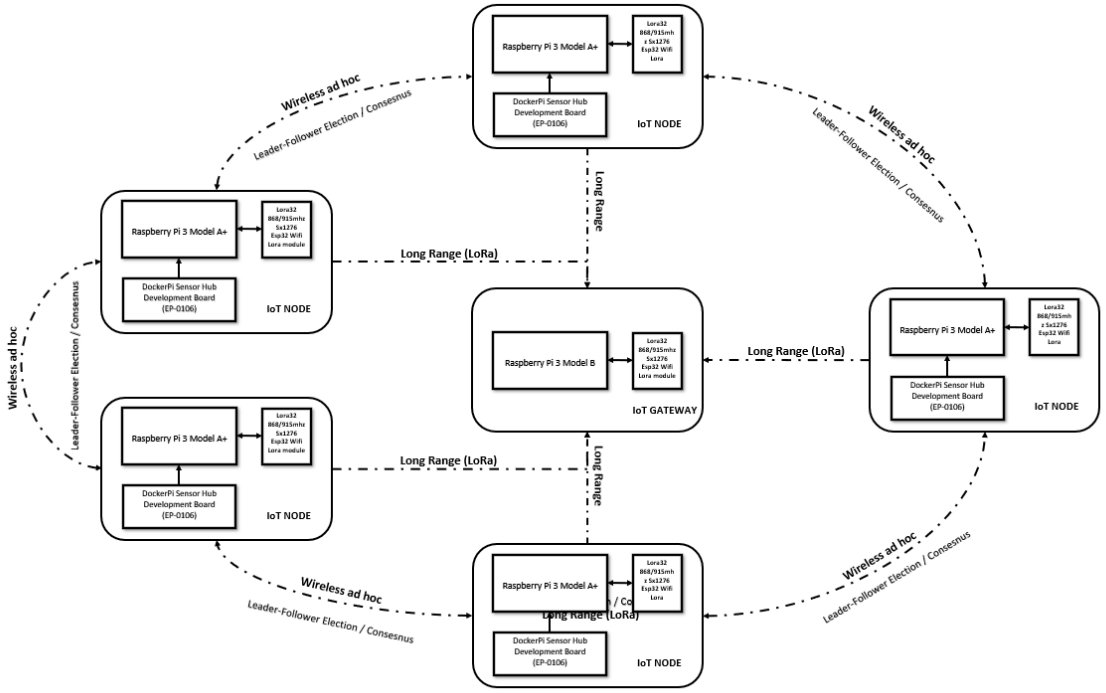


Figure 3.2: Network B - 5 nodes

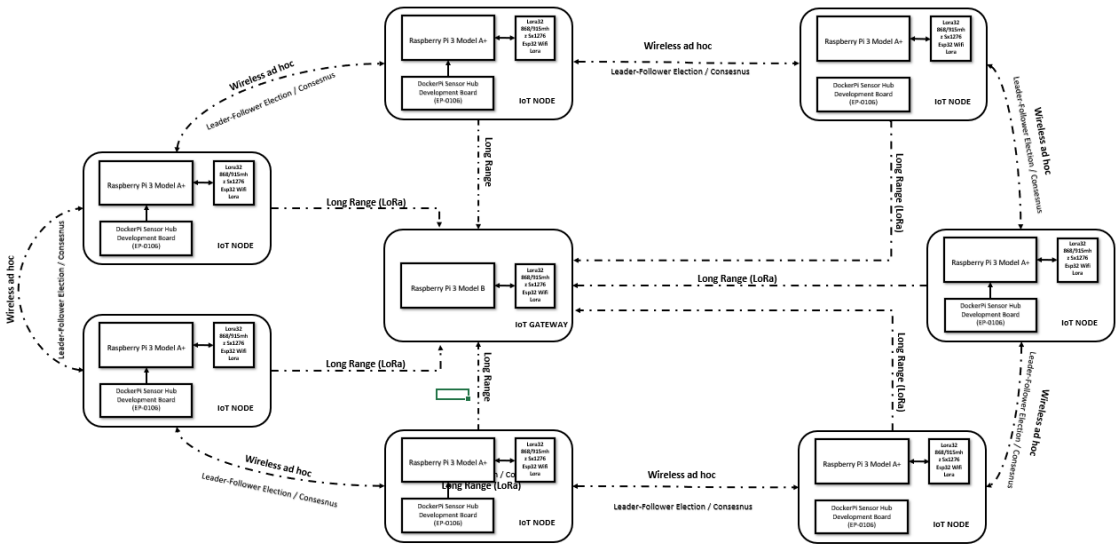


Figure 3.3: Network C - 7 nodes

Network A will consist of three IoT nodes, Network B of five IoT nodes, and Network C of seven IoT nodes. An odd number of nodes is chosen specifically due to the Paxos and Raft requirement of a minimum number of nodes to achieve consensus `(N/2 + 1)`.

3.4.2 Hardware components

LoRa is a low-power wide-area network protocol that uses the physical layer of the seven-layer OSI model. The physical layer defines the means of transmitting bits over a physical data link connecting network nodes. LoRa uses unlicensed radio frequencies, including 433 MHz and 868 MHz in the European Union (Saelens et al., 2019). LoRa enables long-range transmissions with low power consumption, making it a good option for connecting devices in far-reaching environments (Chaudhari & Borkar, 2020). As this experiment requires significant amount of hardware components, below section lists each group with detailed specifications.

3.4.2.1 IoT nodes

Raspberry Pi

Third-generation Raspberry Pi 3 Model A+ single-board computer (SBC) will serve as an IoT node. It features a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @1.4GHz quad-core processor System on a Chip(SoC), 512MB LPDDR2 SDRAM, 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN and Bluetooth 4.2/BLE, Extended 40-pin GPIO header, Single USB 2.0 ports and Micro SD port.

Figure 3.4: Raspberry Pi - Model 3A+

Sensor

Sensor attached to each IoT node is DockerPi Sensor Hub Development Board (EP-0106). It integrates various environmental sensors: temperature sensors, humidity sensors, air pressure sensors, lighting, and thermal imaging sensors. Both, RPP and DockerPi sensor, will be connected using MB102 Breadboard with 3.3V 5V MB102 Breadboard Power Supply Module.

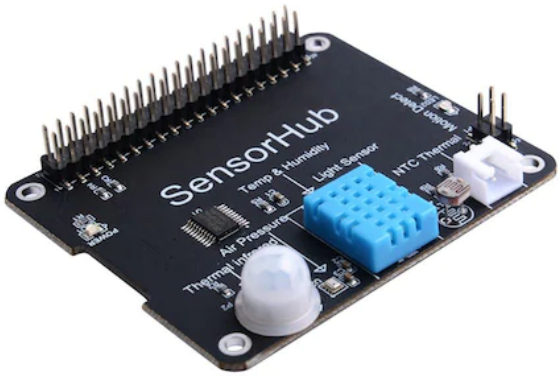


Figure 3.5: DockerPi Sensor Hub Development Board

WiFi LoRa ESP32

Connectivity between edge nodes and gateway will be established with Heltec Lora32 868/915mhz Sx1276 Esp32 WiFi Lora module. WiFi LoRa 32 is an IoT development board produced by Heltec Automation. It is a highly integrated product based on ESP32 microcontroller with 8MB SPI Flash and 529Kb SRAM with SX127x radio chip onboard. Additionally, it features Wi-Fi, BLE and provides full support for LoRa frequencies.

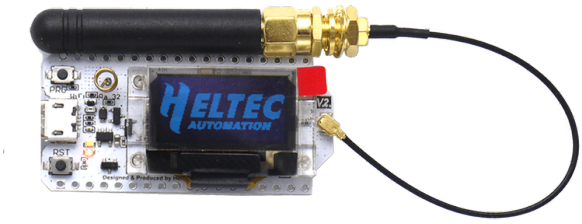


Figure 3.6: ESP32 LoRa Heltec v2 with display

3.4.3 Python Libraries

The author of this paper chooses Python programming language to implement Raft and Paxos algorithms. Python is widely used, and well-documented programming language. Raspberry Pi SBC supports it natively. Additionally, Python libraries provide an implementation of both consensus algorithms. Paxos implementation is based on Cocagne (2013) "composable paxos" implementation that provides an algorithmically correct Paxos implementation that can be used in networked applications. Raft implementation used Ozinov (2022) "PySyncObj" module that provides the ability to replicate application data between multiple servers.

3.4.4 Packet Loss Rate measurement experiment

Network A – 3 nodes

First sub-experiment proposed will measure packet loss rate within IoT network of three nodes. The experiment will be repeated for each of the algorithms, RAFT and Paxos, and additionally for control where no consensus algorithm will be implemented on participating nodes. Packets sent will be recorded on the transmitter and on the receiver node as well and once completed, data will be collected and stored for evaluation and analysis. During the second sub-experiment, fail of one node will be introduced and consequently data will be collected after remaining nodes finished transmitting.

Network B – 5 nodes

Similar steps will be repeated during sub-experiment three, where five IoT nodes will attempt to record and send packets to the receiver node. The receiver node will record packets that arrived and store them for analysis. These steps shall be repeated for sub-experiment four with the introduction of two faulty nodes stopping transmission independently with different intervals. The results of both experiments will be stored for analysis. Both sub-experiments will be repeated for each of the algorithms and for control without consensus algorithm implemented.

Network C – 7 nodes

During sub-experiment six, an IoT network of seven nodes will create data sets by sending packets to the transmitter with RAFT and Paxos algorithms separately implemented and additionally without algorithm. In the course of sub-experiment seven, failures of three nodes will be introduced and data recorded similarly. Once all data from Packet Loss Rate experiment is collected, it will be analysed in data analysis tool. At this point, the author will be able to draw conclusions on whether the alternative hypothesis H2 and H3 can be accepted or rejected.

3.4.5 Latency measurement experiment

The latency measurement experiment will repeat sub-experiments from the Packet Loss Rate experiment for each IoT network:

- Network A – 3 nodes
- Network B – 5 nodes
- Network C – 7 nodes

Additionally, to data collected in the similar way to previous experiment, two timestamps will be added to each packet sent and received. Once having recorded both timestamps, the time it took for data to be transferred between its original source and its destination will be measured, and the author will be able to, either confirm or reject alternative hypothesis H2.

3.4.6 Location

All the above experiments will be carried out in an indoor environment similarly to Kurji, Al-Nakkash, and Hussein (2021). A utility room of 8 meters long and 7 meters wide, where transmitters will be fixed in the corners.

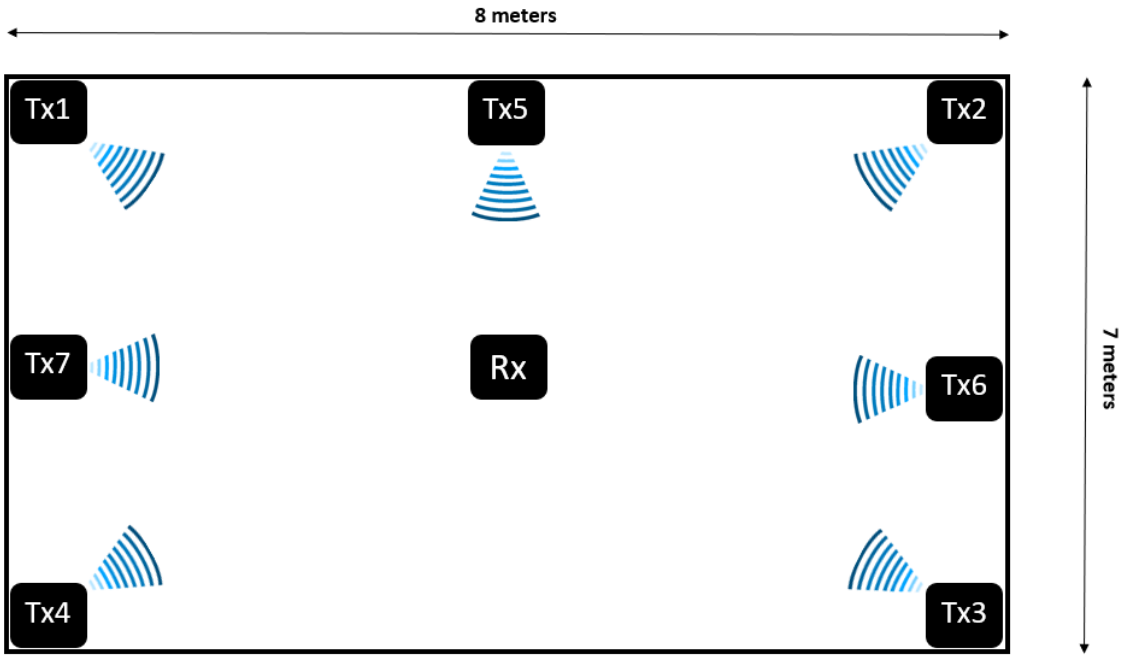


Figure 3.7: Tx and Rx Nodes Setup

Each transmitter will be positioned approximately 1.5 meter above the floor with receiver positioned centrally at around 2 meters height. Below figure depicts the location of transmitter nodes (Tx) and receiver node (Rx) for Packet Loss Rate and Latency measurements experiments.

3.5 Data Design and Data Capture Overview

During both experiments original (non-existent) quantitative (numerical) data will be collected and analysed to confirm or reject H1, H2, H3 and H4.

nodeName	packetNumber	temperature	brightness	humidity	onBoardTemp
1	1	21	27	48	22
1	2	21	27	48	22
1	3	21	27	48	22
1	4	21	27	48	22
1	5	21	27	48	22
1	6	21	27	48	22
1	7	21	26	48	22
1	8	21	25	48	22
1	9	21	26	48	22

Table 3.3: Packet Loss Rate experiment expected data

Above data will be recorded on each IoT node (sent packets) and their aggregation is recorded on the receiver node (received packets). The first variable listed in the table, **nodeName**, represents IoT transmitter node and it will be used to assign **packetNumber**, variable number two, to each of the transmitters. This will allow identifying packets received and associating them with the correct IoT sender node. Variables three through six will be readings of DockerPi Sensor and their use simulates IoT network deployed in a commercial setting (Blenn & Kuipers, 2017).

nodeName	packetNumber	temperature	brightness	humidity	onBoardTemp	sent	received
5	1	19	52	53	18	706344457702	706344457832
6	1	18	178	54	17	706344457914	706344458685
7	1	19	85	49	18	706344459905	706344460639
1	5	19	172	53	18	706344461316	706344461752
2	3	19	121	50	18	706344462983	706344463669
3	2	19	92	57	18	706344464859	706344465688
5	2	19	47	53	18	706344468220	706344468353
6	2	17	185	54	18	706344468432	706344469196
7	2	18	91	49	18	706344470423	706344471157

Table 3.4: Latency experiment expected data

Variable seven, **sent**, represents timestamp and will be recorded on the transmitter side for the purpose of the latency measurement experiment. This will be followed by variable eight, **received**, another timestamp on the receiver's node side. All data will be saved in CSV files on transmitter and receiver nodes.

3.6 Design and Methodology Summary

This section presented an overview of the design and methodology used in the thesis. An overview of the experiment was given, followed by sections that discussed the specifics of the design and methodology. Additionally, data output and capture were discussed, with a detailed evaluation. The next chapter will provide detailed implementation of the experiment.

Chapter 4

Implementation

4.1 Introduction

This section will build on the previous one by delving deeper into the implementation specifics of experiments and the code required to perform them. The following subsections will be divided based on each component required to execute the test cases.

4.2 Required External Libraries

The below table represents MicroPython and Python 3 external libraries and tools required to conduct experiments for the purpose of this thesis.

Library / Module / Tool	Language
esptool	Python 3
uPyLoRaWAN	MicroPython
adafruit-ampy	Python 3
pysyncobj	Python 3
composable_paxos	Python 3

Table 4.1: External Libraries

4.3 LoRa Transceiver Board

4.3.1 MicroPython

WiFi LoRa 32 IoT development board carry the ESP32, a single 2.4 GHz Wi-Fi-and-Bluetooth combo chip, designed for smartwear, mobile and Internet-of-Things (IoT) applications. The ESP32 supports different programming languages, among others is MicroPython, an implementation of Python 3.4 programming language created specifically for microcontrollers. MicroPython, similarly to Python, is easy to use and its simplified syntax provides more emphasis on natural language. A very important difference between both programming languages is that MicroPython does not come with full standard library but it includes libraries that provide access to low-level hardware like GPIOs. Author decided on using MicroPython for programming of ESP32 on WiFi LoRa 32 IoT development board due to his familiarity with Python and widely available documentation and support.

4.3.2 ESP32 Board Set-Up

Heltec board with ESP32 chip used in these experiments provides dual functionality; it can function as transmitter or receiver. As it is shipped from the producent with Andruino Development Environment preinstalled, it required erasing of the entire flash and deploying MicroPython firmware. This was achieved by connecting the board via USB to the Ubuntu 20.04 OS and using the esptool.py to erase the flash and install new firmware.

```
#erase flash
sudo python3 esptool.py --chip esp32 --port /dev/ttyUSB1 erase_flash
#install firmware
sudo python3 esptool.py --chip esp32 --port /dev/ttyUSB1 write_flash -
    ↪ z 0x1000 esp32-20220117-v1.18.bin
```

Listing 4.1: Firmware installation on ESP32 board

Once the board was flashed and MicroPython firmware installed, it required code for the SX127X RF chip and additional helper files to enable LoRa communication. Using ampy tool all required files were transferred onto ESP32 board

```
#transfer files required for enabling LoRa
ampy --port /dev/ttyUSB1 --baud 115200 put config.py
ampy --port /dev/ttyUSB1 --baud 115200 put config_lora.py
ampy --port /dev/ttyUSB1 --baud 115200 put sender.py
ampy --port /dev/ttyUSB1 --baud 115200 put receiver.py
ampy --port /dev/ttyUSB1 --baud 115200 put ssd1306_i2c.py
ampy --port /dev/ttyUSB1 --baud 115200 put ssd1306.py
ampy --port /dev/ttyUSB1 --baud 115200 put sx127x.py
ampy --port /dev/ttyUSB1 --baud 115200 put main.py
ampy --port /dev/ttyUSB1 --baud 115200 put oledsetter.py
```

Listing 4.2: File transfer with Ampy tool

Once files were transferred, board required reset to run main.py with program code.

4.3.3 Configuring SX127X RF chip

Below configuration was chosen for SX127x based on the work of Pötsch and Hammer (2019) and Kurji et al. (2021). Spread Factor (SF), bandwidth(BW) and transmission power (Tx) were specifically adopted due to the short distance between the transmitters and the receiver.

```
#configuration required for initialization of SX127x
default_parameters = {
    'frequency': 868E6,
    'tx_power_level': 2,
    'signal_bandwidth': 125E3,
    'spreading_factor': 8,
    'coding_rate': 5,
    'preamble_length': 8,
```

```
    'implicit_header': False,
    'sync_word': 0x12,
    'enable_CRC': False,
    'invert_IQ': False
}
```

Listing 4.3: SX127X configuration

SX1276x communicates with ESP32 board using Serial Peripheral Interface (SPI) and once SPI is instantiated RF chip is initialized.

```
#instantiate SPI interface
board_spi = SPI(baudrate = 115200,
       polarity = 0, phase = 0, bits = 8, firstbit = SPI.MSB,
       sck = Pin(device_config['sck'], Pin.OUT, Pin.PULL_DOWN),
       mosi = Pin(device_config['mosi'], Pin.OUT, Pin.PULL_UP),
       miso = Pin(device_config['miso'], Pin.IN, Pin.PULL_UP))
#initialize RF chip
lora = SX127x(device_spi, pins=device_config, parameters=
   ↪ lora_parameters)
```

Listing 4.4: SPI interface initialization

4.3.4 Communicating with Raspberry Pi

Universal Asynchronous Receiver/Transmitter. (UART) was chosen as the communication medium between ESP32 and RPi. UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed and requires three wires to enable transmission: a transmit wire, a receive wire, and an electrical ground wire (Barry & Crowley, 2012).

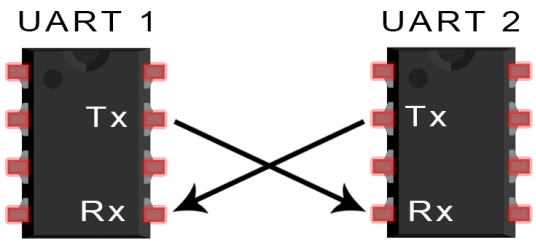


Figure 4.1: UART Basic Connection Diagram

(Campbell, 2020)

Both devices, Heltec board and Raspberry Pi, are equipped with a set of GPIOs that allows for straight forward implementation of UART

```
#import UART module from standard library
from machine import UART
#instantiate UART obj
uart = UART(2, 115200)
#initialize UART
uart.init(115200, bits=8, parity=None, stop=1, tx=17, rx=13)
```

Listing 4.5: UART setup - MicroPython

4.3.5 Transmitting Data

In order to transmit data, Sender object was instantiated in the main.py on reboot of the ESP32 board. This procedure was repeated on each board designated as transmitter.

```
class Sender:
    #constructor
    def __init__(self, lora):
        self.lora = lora
    def send(self, msg):
        self.lora.blink_led()
```

```
        oled = OledSetter.setOled()
        role = "TRANSMITTER"
        #create payload
        payload = '{p}'.format(p=msg)
        OledSetter.displayOled(oled, role, payload)
        #send payload
        self.lora.println(payload)
```

Listing 4.6: Class Sender - MicroPython

An infinite(while) loop was used for the Sender object to constantly read data from the UART interface and send (**send()**) it via LoRa and UART interface was used to retrieve messages from Raspberry Pi.

```
#instantiate Sender obj
sender = Sender(lora)
#infinite loop
while True:
    #read UART
    msg = uart.readline()
    if msg:
        #send payload
        sender.send(msg)
```

Listing 4.7: Instantiation of Sender obj - MicroPython

Once the ESP32 board was restarted, it automatically accepted any payload sent using the UART interface, smaller than 256 bytes, from Raspberry Pi and immediately sent via LoRa in the ether.

4.3.6 Receiving Data

The receiver object was created in the main.py similarly to the Sender object with the infinite(while) loop in the receive method responsible for reading LoRa packets.

Listing 4.8: Class Receiver - MicroPython

```
class Receiver:
    #constructor
    def __init__(self, lora):
        self.lora = lora


    def receive(lora):
        oled = OledSetter.setOled()
        role = "RECEIVER"
        #infinite loop
        while True:
            if lora.received_packet():
                lora.blink_led()
                #save payload
                payload = lora.read_payload()
                #send payload to UART
                print(payload)
                OledSetter.displayOled(oled, role, payload)
```

Heltec LoRa ESP32 board configured as a receiver was constantly accepting packets sent over via LoRa and pushing them using UART interface to Raspberry Pi.

4.3.7 Modification to Transmitter and Receiver required for Latency Measurement

Latency measurement experiment proved to be a challenge due to synchronization issues with internal clocks on both, Raspberry Pi and Heltec LoRa ESP32 boards without access to internet (Hernández-Solana, Pérez-Díaz-De-Cerio, García-Lozano, Bardají, & Valenzuela, 2020). To perform Latency measurement, the author decided on using ESP32 boards WiFi module to synchronize the onboard clock with the Network Time Protocol (NTP) servers to achieve timewise synchronization among the

transmitters and receiver boards. Once connected to WiFi on start-up, each node would synchronize its internal time with the local NTP server.

```
#connect to WiFi
wlan = network.WLAN(network.STA_IF)
wlan.active(True)
wlan.connect('******', '*******')

#sleep until connected to WiFi
while not wlan.isconnected():
    time.sleep(2)
#synchronize the time of the board
ntptime.settime()
```

Listing 4.9: Synchronization with NTP server for the latency experiment - MicroPython

For the purpose of this thesis, latency was measured as the difference between timestamps((**TS**); `Elapsed Time = TS received - TS sent` . To achieve the most accurate results, timestamp was generated on packet sent, in Sender object, and another timestamp was generated on packet received, in Receiver object. Both timestamps were outputted to CSV file on Raspberry Pi to allow for later analysis.

```
#modification to while loop for Sender obj
while True:
    msg = uart.readline()
    if msg:
        #create timestamp in milliseconds
        n = time.time_ns() // 1000000
        sender.send(msg, n)

# modification to while loop for Receiver obj
while True:
        if lora.received_packet():
```

```
        #create timestamp in milliseconds
        n = time.time_ns() // 1000000
        lora.blink_led()
        p = lora.read_payload()
        #save payload with the timestamps
        payload = '{m} - {i}'.format(m=p, i=n)
        #send payload to UART
        print(payload)
        OledSetter.displayOled(oled, role, payload)
```

Listing 4.10: Modification of Sender class for the latency experiment - MicroPython

4.4 Raspberry Pi with DockerPi Hub

4.4.1 Python 3

Python 3 is a general-purpose interpreted, interactive, object-oriented, and high-level programming language which source code is available under the GNU General Public License. It is a mature language with a large set of comprehensive multipurpose libraries, frameworks and extensions that is actively developed by Python Software Foundation. During the 30 years of existence, Python went through three major revisions and its current version 3.x is not fully backward-compatible with previous iterations. Python's design was based on concept of modularity, where additional functionality can be added by importing modules instead of building it into the core of the language. The author decided to implement experiments in Python 3 due to its extensibility and the Raspberry Pi operating system being shipped with preinstalled Python 3 onboard. Additionally, as mentioned in the previous subsection, MicroPython is a lighter version of Python 3.4.

4.4.2 RPi Operating System

Raspberry Pi OS (64-bit) v.11, Debian-based operating system, was deployed and running on each Raspberry Pi.

4.4.3 RPi Wireless ad hoc network

Each Raspberry Pi participating in experiments was assigned a static IP address that enabled it to create a part of a decentralized wireless ad hoc network. To allow connectivity between RPis acting as IoT nodes below changes were made to `/etc/network/interfaces` file which contains network interface configuration information for Debian Linux.

```
#include files from /etc/network/interfaces.d
source-directory /etc/network/interfaces.d
#the loopback network interface
auto lo
iface lo inet loopback
#the primary network interface
iface eth0 inet dhcp
auto wlan0
iface wlan0 inet static
       #assign unique IP address
       address 192.168.1.1
       netmask 255.255.255.0
       wireless-channel 4
       wireless-essid RPIMESH
       wireless-mode ad-hoc
```

Listing 4.11: Wireless ad hoc set up

The table below illustrates the unique IP addresses assigned to each IoT node in ad hoc network:

Node Number	IP Address
1	192.168.1.1
2	192.168.1.2
3	192.168.1.3
4	192.168.1.4
5	192.168.1.5
6	192.168.1.6
7	192.168.1.7

Table 4.2: IP addresses of IoT nodes

Once Raspberry Pi was restarted, it automatically joined the wireless ad hoc network and was able to communicate with other nodes in the network.

4.4.4 Collecting environmental data with Docker Pi Sensor Hub

DockerPi Sensor Hub Development Board (EP-0106) was attached to each RPi and together it worked as an IoT node. Sensor Hub was collecting environmental data using various sensors such as: temperature sensors, humidity sensors, air pressure sensors, lighting, and thermal imaging sensors. For the purpose of this thesis author decided to collect four sensor readings (i.e., temperature, brightness, humidity and on board temperature) to send them in LoRa packets.

```
def createData():
    #set registers
    DEVICE_BUS = 1
    DEVICE_ADDR = 0x17
    #set registers
    TEMP_REG = 0x01
    LIGHT_REG_L = 0x02
    LIGHT_REG_H = 0x03
    STATUS_REG = 0x04
    ON_BOARD_TEMP_REG = 0x05
```

```
ON_BOARD_HUMIDITY_REG = 0x06
ON_BOARD_SENSOR_ERROR = 0x07
BMP280_TEMP_REG = 0x08
BMP280_PRESSURE_REG_L = 0x09
BMP280_PRESSURE_REG_M = 0x0A
BMP280_PRESSURE_REG_H = 0x0B
BMP280_STATUS = 0x0C
HUMAN_DETECT = 0x0D

#initialize System Management Bus
bus = smbus.SMBus(DEVICE_BUS)

#container for sensors readings
aReceiveBuf = []
aReceiveBuf.append(0x00)

#read registers
for i in range(TEMP_REG,HUMAN_DETECT + 1):
    aReceiveBuf.append(bus.read_byte_data(DEVICE_ADDR, i))

#return temperature, brightness, humidity, on board temp
return aReceiveBuf[TEMP_REG], (aReceiveBuf[LIGHT_REG_H] << 8 |
    ↪ aReceiveBuf[LIGHT_REG_L]), aReceiveBuf[ON_BOARD_HUMIDITY_REG
    ↪ ], aReceiveBuf[ON_BOARD_TEMP_REG]
```

Listing 4.12: Sensor data collection - Python3

4.4.5 Creating LoRa packet and sending it via UART

Below subsections will illustrate implementation of experiments from chapter three of this thesis. These experiments were divided in three sub experiments based on

the algorithm (or lack thereof) used implemented for scheduling purposes. Therefore, control experiments were implemented without any consensus algorithm and were followed by Paxos and Raft consensus algorithms implementations. As the timestamps required for the latency measurement experiment were not added to the packets during program execution on Raspberry Pi, these variables are not presented in the code.

Control Experiment

Listing 4.13 shows how the data is collected from the environmental sensors send to a LoRa ESP32 module and outputted to the CSV file with additional information, i.e., node number and packet number. As per the experiment's assumptions, the counter variable was initialized and the maximum value was set to 1000 to allow for an exact number of packets to be sent. Every single packet had a unique consecutive number in the range of 1 to 1000 assigned to allow for identification and analysis after the experiments were concluded.

```
def main():
    #set UART serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #declare node name
    node = 1
    #initialize packet counter
    count = 1
    #declare header for csv file
    header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
        ↪ 'humidity', 'onBoardTemp']

    #create csv file
    with open('sensor-4_data.csv', 'w', encoding='UTF8', newline='') as
        ↪  f:
        #writer object
        writer = csv.writer(f)
```

```
        #write the header to the csv file
        writer.writerow(header)


        #loop
        while True:
            #2 sec interval
            time.sleep(2)
            #read sensor hub data
            temp, light, humidity, onBoardTemp = createData()
            #create data
            data = [node, count, temp, light, humidity, onBoardTemp]
            #send packet over UART to LoRa board
            s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light,
                ↪ humidity, onBoardTemp))
            #save packet to the csv file
        writer.writerow(data)
            #increment packet counter
            count += 1
```

Listing 4.13: Control experiment Raspberry Pi code - Python3

During the control experiments each IoT node, consisting of RPi, sensor hub and LoRa ESP32 board, was collecting and sending data with two seconds intervals and no other mechanism was implemented to prevent packet's collisions (Zorbas et al., 2021) or to improve scheduling (Rajab et al., 2021).

Paxos Experiment

Listing 4.14 represents configuration file required to specify nodes of the Paxos group with the IP addresses and ports. This implementation of Paxos require separate files for preserving state of the Paxos group as well.

```
# IP, Port Number
```

```
peers = dict(1=(192.168.1.1,12345),2=(192.168.1.2,12345),3=(
    ↪ 192.168.1.3,12345))

#state files for crash recovery
state_files = dict(1='/tmp/1.json', 2='/tmp/2.json', 3='/tmp/3.json')
```

Listing 4.14: Paxos configuration file - Python3

Scheduler in the form of counter was implemented based on consensus and value of the counter was propagated between the nodes participating in the experiment. Once value of the counter was equal to the node number, the packet was sent.

```
def main():
    #declare node name
    node = 1
    filename = 'sensor-1_data_raft.csv'
    #initialize Server obj
    serv = Server(node)
    #set serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #initialize packet counter
    count = 1
    #node counter
    counter = 1
    #declare header for csv file
    header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
        ↪ 'humidity', 'onBoardTemp']

    #create csv file
    with open(filename, 'w', encoding='UTF8', newline='') as f:
        #writer object
        writer = csv.writer(f)
```

```
        #write the header to the csv file
        writer.writerow(header)
        #loop
        while count <= 1000:
            #2 sec interval
            time.sleep(2)
            #check if this node is master node
            if serv.get_master() == node:
                if counter <= 3:
                    #increment counter
                    counter+=1
                #initialize Client object and propose new value
                ClientProtocol(node, counter)
                else:
                    #restart counter
                    counter = 1
                #initialize Client object and propose new value
                ClientProtocol(node, counter)
        #if the value of the counter matches node number, sent packet
            ↪
            if counter == node:
                #read sensor hub data
                temp, light, humidity, onBoardTemp = o.createData()
                #write data to csv file
                data = [node, count, temp, light, humidity, onBoardTemp]
                writer.writerow(data)
                #send packet over lora
                s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light
                    ↪ , humidity, onBoardTemp))
            #increment packet count
```

```
            count+=1
```

Listing 4.15: Paxos experiment Raspberry Pi code - Python3

Raft Experiment

In a similar manner to Paxos algorithm, Raft was implemented with synchronized counter that allowed for scheduling of the packets sent to only one node at the time minimizing collisions of the packets. Two seconds interval was unchanged as per assumption of the control experiment.

```
def main():
    #this node IP
    my_ip = "192.168.1.1"
    #initialize RaftObj obj
    o = RaftObj('192.168.1.1:12345', ['192.168.1.2:12345', '
        ↪ 192.168.1.3:12345'])
    old_value = -1
    filename = 'sensor-1_data_raft.csv'
    #set serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #declare node name
    node = 1
    #initialize packet counter
    count = 1
    #temp, light, humidity, onBoardTemp = createData()
    #print(node, count, temp, light, humidity, pressure)
    #declare header for csv file
    header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
        ↪ 'humidity', 'onBoardTemp']

    #create csv file
```

```
with open(filename, 'w', encoding='UTF8', newline='') as f:
    #writer object
    writer = csv.writer(f)
    #write the header to the csv file
    writer.writerow(header)
    #loop
    while count <= 1000:
        #2 sec interval
        time.sleep(2)
        #check if counter is initialized
        if o.getCounter() != old_value:
            old_value = o.getCounter()
        #check if this node is leader node
        if str(o._getLeader()).split(":", 1)[0] == my_ip:
            if o.getCounter() <= 3:
                #increment counter
                o.addValue(1)
            else:
                #restart counter
                o.setCounter(1)
        if o.getCounter() == node:
            #read sensor hub data
            temp, light, humidity, onBoardTemp = o.createData()
            #write data to csv file
            data = [node, count, temp, light, humidity, onBoardTemp]
            writer.writerow(data)
            #send packet over lora
            s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light
                ↪ , humidity, onBoardTemp))
        #increment packet count
```

```
            count+=1
```

Listing 4.16: Raft experiment Raspberry Pi code - Python3

All packets sent during each experiment were outputted to the csv file after forwarding them via UART to the LoRA ESP32 board. The addition of synchronized counter in Paxos and Raft experiments, for the purpose of this thesis functioning as a scheduler, allowed to introduce an order in which packets were sent over LoRa.

Additionally, as per experiments' requirements, failures of the nodes were carried through changing the packet counter range for nodes 1, 2 and 3 to 300, 500 and 700 correspondingly (dependant on the use case).

4.5 Raspberry Pi Receiver node

Raspberry Pi acted as the gate node for the purpose of this thesis and, in the similar manner to transmitter nodes, was connected to receiver LoRA ESP32 board via UART. Python script was used to constantly receive any data on the serial port and to output it to csv file. Due to the format of the incoming data some additional string manipulation was required before writing fields to the file. The csv file contained received packets from each node.

```
#set serial connection with ESP32
s=serial.Serial('/dev/ttyS0',115200)
#exclude pattern
pattern = "free"
#declare header for csv file
header = ['nodeName', 'packetNumber', 'temperature', 'brightness', '
    ↪ humidity', 'onBoardTemp']
#create csv file
with open('/home/bart/msc_python/data.csv', 'w', encoding='UTF8') as f
    ↪ :
    #writer object
```

```
writer = csv.writer(f)
#write the header to the csv file
writer.writerow(header)
#loop
while True:
    #read input from UART
    data = s.readline()
    #do not process if pattern exists
    if data and not re.search(pattern, data.decode()):
        #decode from bytestream
        d = data.decode()
    #cleanup
        final_data = d.replace('"','').replace("b","").replace("'",
            ↪ "").replace("\r","").replace("\n","").replace("-","","
            ↪ ).replace("␣","")
        #write data to the file
    writer.writerow(final_data.split(","))
```

Listing 4.17: Receiver Raspberry Pi code - Python3

4.6 Conclusion

In this chapter author covered in depth the implementations of the experiments with detailed code examples and discussed the setup of each of the components required to perform them. The next section will discuss the results obtained from the experiments.

Chapter 5

Results, Evaluation and Discussion

5.1 Introduction

This section discusses the results obtained from running the experiments described in detail in Chapters Three and Four. Results from the Packet Loss Rate and Latency implementations will be presented and compared. This Chapter will also evaluate the results, with some discussion taking place.

5.2 Research Question

Can implementation of Raft consensus algorithm in IoT network decrease packet loss rate and improve latency on the network in the presence of malfunctioning edge nodes when compared to Paxos consensus algorithm?

5.3 Results

5.3.1 Packet Loss Rate measurement experiment

Test Statistics

T-test with paired-samples was used to statistically compare performance of consensus algorithms during experiments with different conditions. Author attempted to

calculate `p-value` to either confirm or reject null hypothesis. `P-value` indicates if the difference between two paired samples is statistically significant. As level of statistical significance is denoted by the value between 0 and 1, higher `p-value` indicate that null hypothesis should be accepted and rejected in case of smaller `p-value`:

- `p-value` higher than 0.05 is not statistically significant (`p-value` $>$ `0.05`)
- `p-value` less than 0.05 is statistically significant (`p-value` $\leq$ `0.05`)

	packets	Raft vs No Algorithm	Raft vs Paxos
0	totalPacketsReceived	0.009620	0.018597
1	goodPacketsReceived	0.009574	0.059302

Table 5.1: `p-values`

In the above table the p-value clearly demonstrates that the null hypothesis can be rejected for IoT network with no consensus algorithm implemented. P-value for Paxos implementation signifies that null hypothesis should be accepted. These results of t-test are expected, as both consensus algorithms introduce order to LoRa transitions hence decreasing Packet Loss Rate.

Experiments' evaluation

Attempted indoor control experiments, with no consensus algorithm implemented, delivered results where packet loss rate was notably high when compared to author's expectations.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
0	7_nodes	2366	2648	7000	33.8000	37.8286	66.2000	62.1714	noAlg
1	7_nodes_3-f	1770	1967	5500	32.1818	35.7636	67.8182	64.2364	noAlg
2	5_nodes	2147	2381	5000	42.9400	47.6200	57.0600	52.3800	noAlg
3	5_nodes_2-f	2011	2016	3800	52.9211	53.0526	47.0789	46.9474	noAlg
4	3_nodes	1879	1926	3000	62.6333	64.2000	37.3667	35.8000	noAlg
5	3_nodes_1-f	1996	2002	2500	79.8400	80.0800	20.1600	19.9200	noAlg

Table 5.2: Indoor control experiments results

After further research, author realized that, as per Liang, Zhao, and Wang (2020) "the packet loss rate in the outdoor under line-of-sight environment was much smaller

than that in the indoor environment". Following on these findings, all experiments were redesigned and relocated outdoors. Locations of the nodes were adjusted due to topography of the outdoor space.

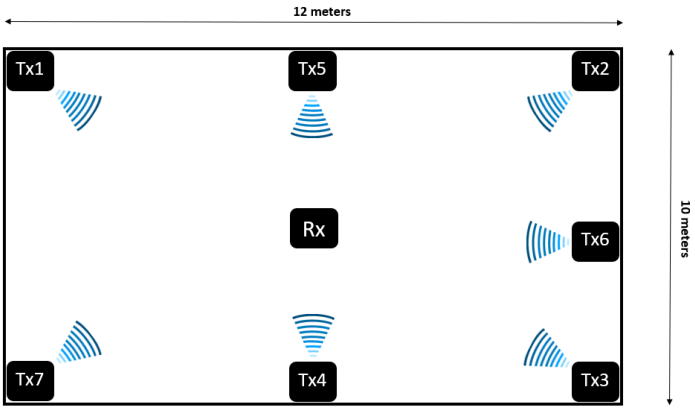


Figure 5.1: Outdoor Tx and Rx Nodes Setup

Additionally due to the collisions of the LoRa packets (Ferre, 2017), the data received required additional steps to remove corrupted and incorrectly formatted data from the data set. Example in the table below shows corrupted and incomplete data in dataset created on the receiver side.

	nodeName	packetNumber	temperature	brightness	humidity	onBoardTemp
0	\\x00	nan	nan	nan	nan	nan
1	1	1	18.0000	388.0000	54.0000	17
2	1	2	17.0000	389.0000	54.0000	17
3	2	1	17.0000	1555.0000	53.0000	16
4	1	3	17.0000	384.0000	54.0000	17
5	\\x8e\\x0cppL\\x0c\\x0c|\\x92\\x0cr\\xe0\\x0cp\\x9e\\xfc	nan	nan	nan	nan	nan
6	1	4	17.0000	260.0000	54.0000	17
7	2	2	17.0000	1547.0000	53.0000	16
8	3	2	18.0000	91.0000	58.0000	17
9	1	5	17.0000	256.0000	54.0000	17
10	2	3	17.0000	65535.0000	53.0000	16

Table 5.3: Errors in data due to collisions

Once data cleansing process was completed, data set contained only numerical values assigned to six columns of which column **nodeName** and column **packetNumber** were analysed for the purpose of the Packet Loss Rate experiment.

	nodeName	packetNumber	temperature	brightness	humidity	onBoardTemp
1	1	1	18.0000	388.0000	54.0000	17
2	1	2	17.0000	389.0000	54.0000	17
3	2	1	17.0000	1555.0000	53.0000	16
4	1	3	17.0000	384.0000	54.0000	17
6	1	4	17.0000	260.0000	54.0000	17
7	2	2	17.0000	1547.0000	53.0000	16
8	3	2	18.0000	91.0000	58.0000	17
9	1	5	17.0000	256.0000	54.0000	17
10	2	3	17.0000	65535.0000	53.0000	16

Table 5.4: **nodeName** and **packetNumber**

Below subsections will present the measurement of the frequencies of each node in the dataset representing packets received. As each node transmitted exactly 1000 packets, the packet loss rate was measured by multiplying number of packets sent by the number of nodes participating in the experiment. Numbers are represented in three columns per each experiment:

- **goodPacketsReceived** representing data after cleaning of the dataset
- **totalPacketsReceived** representing all rows in the dataset
- **packetsExpected** representing number of expected packets

To simulate failing nodes during experiments, below configuration was applied. Once the transmitter node reached pre-set limit of packets to be sent, it would automatically switch off:

	Experiment	1 fail	2 fail	3 fail
0	7 nodes with 3 failures	300	500	700
1	5 nodes with 2 failures	500	700	NaN
2	3 nodes with 1 failure	500	NaN	NaN

Table 5.5: Packet limit at which nodes would turn off to simulate failure

By removing randomness from the experiments requiring node failures, author attempted to introduce predictable number of packets expected.

Network A – 3 Nodes

During the first sub-experiment author measured the packet loss rate within IoT network of three nodes. The experiment was repeated for each of the consensus algorithms, RAFT and Paxos, and additionally for control where no consensus algorithm was implemented on participating nodes. Each of the transmitter nodes recorded 1000 environmental readings and attempted to dispatch them via LoRa medium. All packets sent were recorded on the transmitter and on the receiver node as well, in csv format, to allow for further evaluation and analysis in Jupyter Notebook.

In the second sub-experiment, the failure of one node was introduced to measure the impact on LoRa traffic and packet loss rate. Similarly, all packets sent and received were recorded and evaluated in Jupyter Notebook during the experiment evaluation phase.

3 Nodes, no node failure

Figure 5.2 shows the number of packets received during each experiment, i.e., Raft implementation, Paxos implementation and no algorithm. In this experiment author expected 3000 thousand packets to be recorded in dataset on the receiver node. It can be seen from the figure that both implementations of consensus algorithm, Raft and Paxos, significantly decrease packet loss rate.

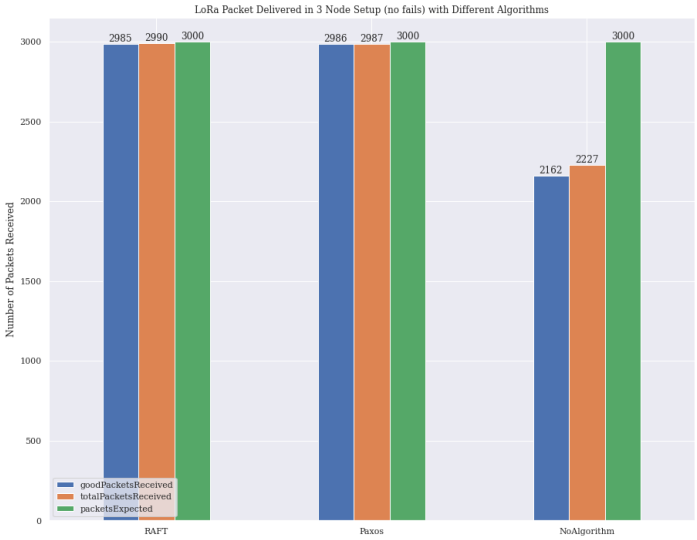


Figure 5.2: 3 Nodes, no node failure

Both, IoT network with Raft and Paxos, achieved over 99% of the packets delivered successfully and their performance was better than the IoT network without any consensus implementation.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
12	3_nodes	2985	2990	3000	99.50	99.67	0.50	0.33	RAFT
13	3_nodes	2986	2987	3000	99.53	99.57	0.47	0.43	Paxos
14	3_nodes	2162	2227	3000	72.07	74.23	27.93	25.77	noAlg

Table 5.6: 3 Nodes, no node failure

3 Nodes, 1 node failure

During experiment with three nodes and one failure simulated at 500 packets, we can see from Figure 5.3 that it very closely follows Figure 5.2 in respect to Raft algorithm. Paxos implementation result, although not as efficient as Raft implementation, is showing a significant advantage in decreasing packet loss rate when compared to no consensus implementation as well.

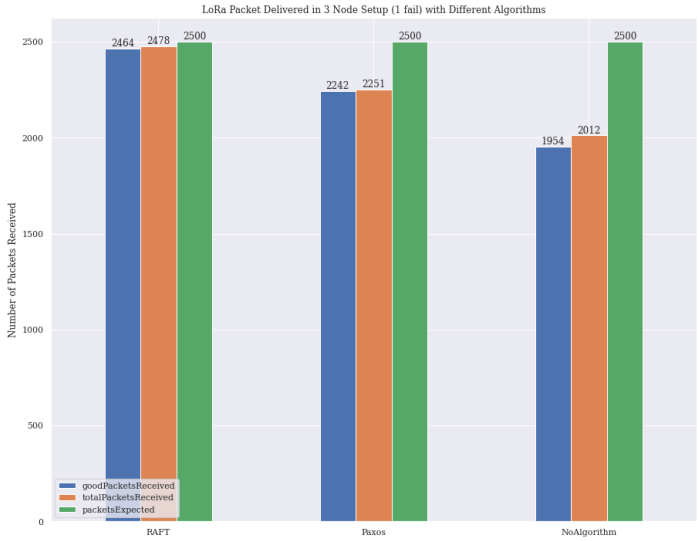


Figure 5.3: 3 Nodes, 1 node failure

As can be seen from Table 5.7, Raft consensus algorithm implementation decreases packet loss rate when compared to no implementation in control experiment.

Table 5.7: 3 Nodes, 1 node failure

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
15	3_nodes_1-f	2464	2478	2500	98.56	99.12	1.44	0.88	RAFT
16	3_nodes_1-f	2242	2251	2500	89.68	90.04	10.32	9.96	Paxos
17	3_nodes_1-f	1954	2012	2500	78.16	80.48	21.84	19.52	noAlg

Network B – 5 Nodes

Similar steps were repeated during sub-experiment three, where five IoT nodes attempted to record and send packets to the receiver node. The receiver node recorded packets that arrived and stored them for analysis. These steps were repeated for sub-experiment four with the introduction of two faulty nodes stopping transmission independently with different intervals. Results of both experiments were stored for analysis in a csv file on transmitter and receiver nodes. Both sub-experiments were repeated for each of the algorithms and for control without a consensus algorithm implemented. **5 Nodes, no node failure**

Figure 5.4 represents number of packets received in experiments with 5 nodes. Once again, it shows that general trend of Raft algorithm decreasing packet loss rate is upheld.

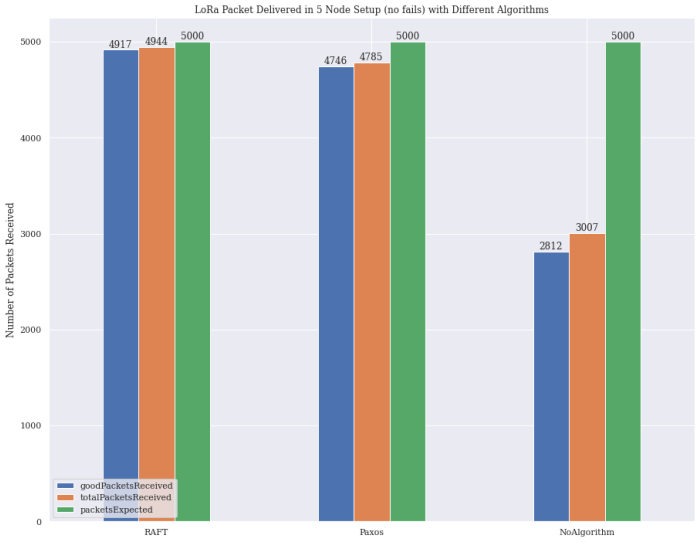


Figure 5.4: 5 Nodes, no node failure

During experiments IoT network with Raft implementation achieved over 98% of packet delivery rate, followed closely by Paxos implementation.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
6	5_nodes	4917	4944	5000	98.34	98.88	1.66	1.12	RAFT
7	5_nodes	4746	4785	5000	94.92	95.70	5.08	4.30	Paxos
8	5_nodes	2812	3007	5000	56.24	60.14	43.76	39.86	noAlg

Table 5.8: 5 Nodes, no node failure

5 Nodes, 2 node failures

Similarly, when 2 failures were introduced, IoT network with Raft implementation significantly outperforms IoT network without consensus algorithm implementation. Figure 5.5 shows packets delivered during experiments with both algorithms implemented and control experiment.

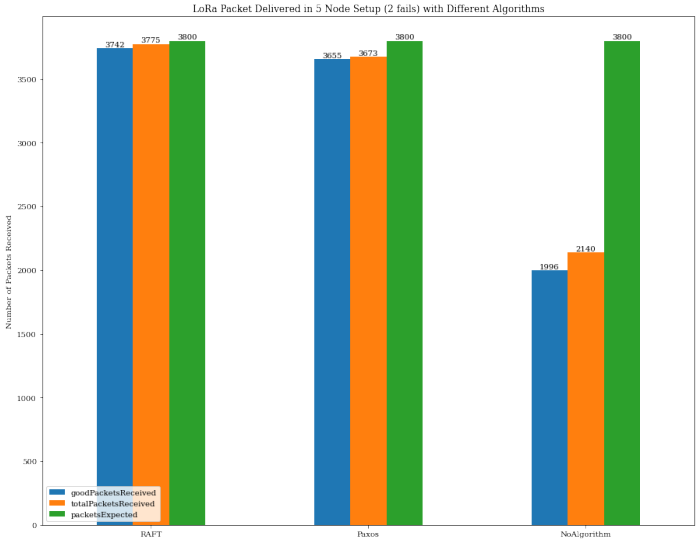


Figure 5.5: 5 Nodes, 2 node failures

During this experiment both consensus algorithms outperformed IoT network without algorithm implementation, as can be seen in Figure 5.9.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
9	5_nodes_2-f	3742	3775	3800	98.47	99.34	1.53	0.66	RAFT
10	5_nodes_2-f	3655	3673	3800	96.18	96.66	3.82	3.34	Paxos
11	5_nodes_2-f	1996	2140	3800	52.53	56.32	47.47	43.68	noAlg

Table 5.9: 5 Nodes, 2 node failure

Network C – 7 nodes

During sub-experiment six, IoT network of seven nodes created data sets by sending

packets to transmitter with Raft and Paxos algorithms separately implemented and additionally without algorithm. During sub-experiment seven, failures of three nodes were introduced and data were recorded in CSV format, in similar manner to the previous experiments.

Once all data from the Packet Loss Rate experiment was collected, the author analysed it in Jupyter Notebook tool and at this point, the author was able to draw conclusions on whether the alternative hypotheses H2 and H3 can be accepted or rejected. **7 Nodes, no node failure**

Figure 5.6 shows results of the experiments with 7 nodes where both IoT networks with implemented algorithms show clear advantage in respect to decreasing packet loss rate in comparison to IoT network from control experiment.

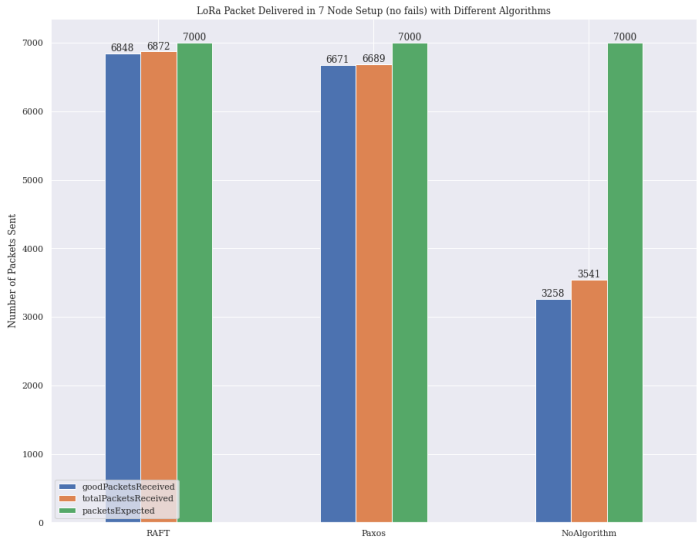


Figure 5.6: 7 Nodes, no node failures

Likewise, when the percentage of packets delivered is shown in Table **??** it demonstrates the advantage of implementing a consensus algorithm in IoT network. Both algorithms deliver a packet loss rate of less than 5%.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
0	7_nodes	6848	6872	7000	97.83	98.17	2.17	1.83	RAFT
1	7_nodes	6671	6689	7000	95.30	95.56	4.70	4.44	Paxos
2	7_nodes	3258	3541	7000	46.54	50.59	53.46	49.41	noAlg

Table 5.10: 7 Nodes, no node failure

7 Nodes, 3 node failures

Figure 5.7 represents the results of the final sub-experiments in Packet Loss Rate measurement experiment.

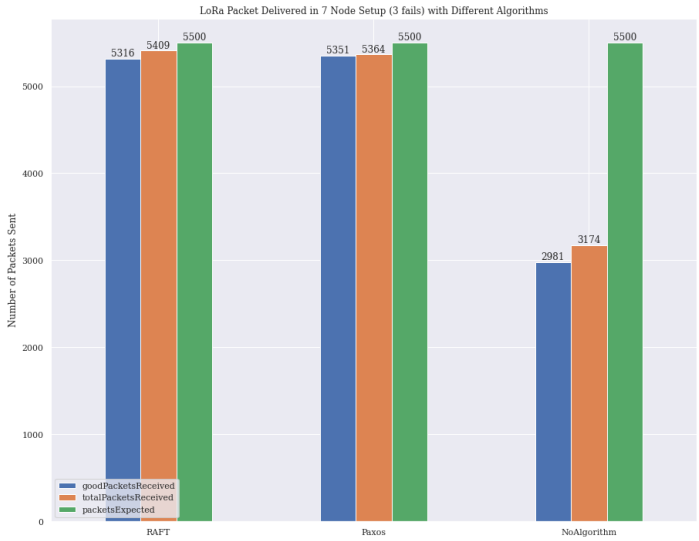


Figure 5.7: 7 Nodes, 3 node failures

Following on similar trend shown in previous experiments, both consensus algorithms provide significant advantage in respect to delivering decrease in packet loss rate in IoT network.

	test	goodPacketsReceived	totalPacketsReceived	packetsExpected	%goodPacketsReceived/packetsExpected	%totalPacketsReceived/packetsExpected	%goodPacketLossRate	%totalPacketLossRate	algorithm
3	7_nodes_3-f	5316	5409	5500	96.65	98.35	3.35	1.65	RAFT
4	7_nodes_3-f	5351	5364	5500	97.29	97.53	2.71	2.47	Paxos
5	7_nodes_3-f	2981	3174	5500	54.20	57.71	45.80	42.29	noAlg

Table 5.11: 7 Nodes, 3 node failure

5.3.2 Latency measurement experiment

Latency measurement experiment repeated all sub-experiments from Packet Loss Rate experiment for each IoT network:

- Network A – 3 nodes
- Network B – 5 nodes
- Network C – 7 nodes

Additionally, to data collected in the similar way to previous experiment, two timestamps were added to each packet send and received. Once having recorded both timestamps, the time it took for data to be transferred between its original source and its destination was calculated.

Experiments' evaluation

Datasets retrieved from Receiver node required additional cleaning due to number of empty or corrupted rows.

	nodeName	packetNumber	temperature	brightness	humidity	onBoardTemp	sent	received
0	2	2	21	404	38	22	706621708234	706621708751.000000
1	nan	706621709426	706621709878	nan	nan	nan	nan	nan
2	2	4	21	401	38	22	706621710949	706621711461.000000
3	2	5	22	400	38	22	706621712958	706621713473.000000
4	3	10	2	400	38	22	706621714165	706621714686.000000
5	1	8	21	421	39	22	706621715050	706621715347.000000

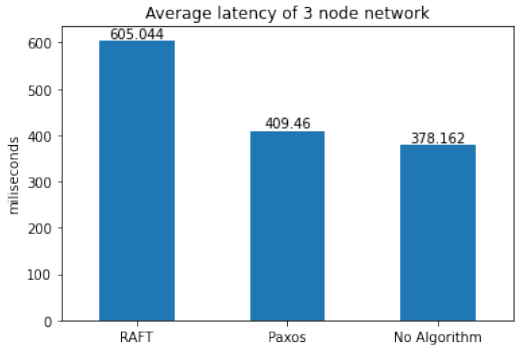
Table 5.12: 7 Nodes, 3 node failure

As latency, or delay is calculated as the difference between received timestamp and sent timestamp, the author decided to discard all columns bar two: **sent** and **received**.

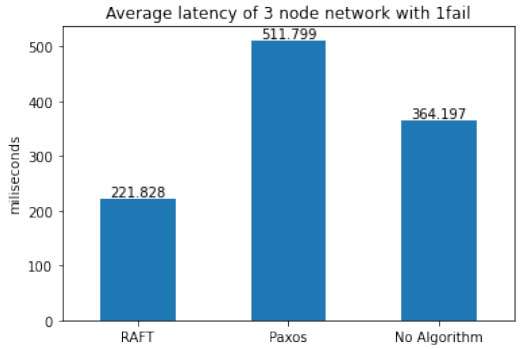
	sent	received
0	706621708234	706621708751.0
2	706621710949	706621711461.0
3	706621712958	706621713473.0
4	706621714165	706621714686.0
5	706621715050	706621715347.0
...	...	...
1809	706623217339	706623217822.0
1810	706623218856	706623218540.0
1811	706623220864	706623220553.0
1812	706623221871	706623221556.0
1813	706623223779	706623223474.0

Table 5.13: Timestamps for latency experiment

Below figures represent latency measurement taken during repeated experiments where timestamp was concatenated on transmitter node and additional timestamp was added when packet was received on receiver IoT node.

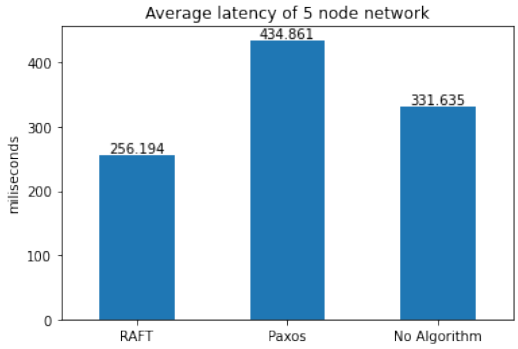


(a) 3 Nodes, no node failure latency

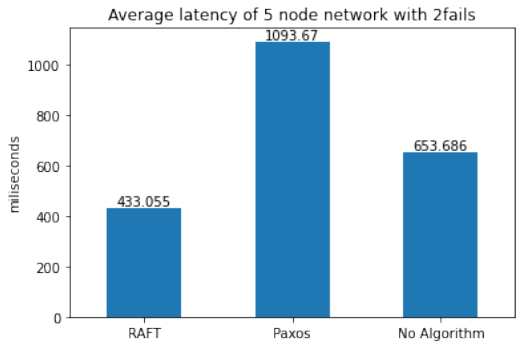


(b) 3 Nodes, 1 node failure latency

Figure 5.8: Latency in Network A – 3 nodes

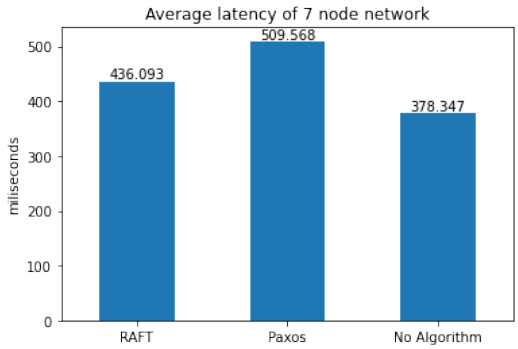


(a) 5 Nodes, no node failure latency

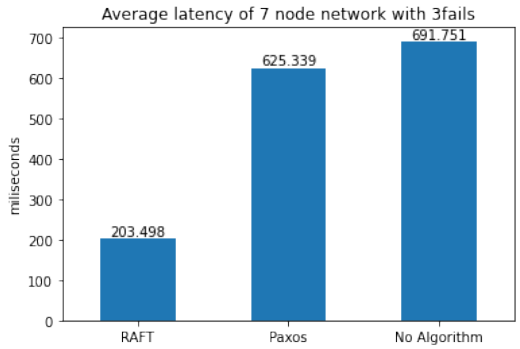


(b) 5 Nodes, 2 node failure latency

Figure 5.9: Latency in Network B – 5 nodes



(a) 7 Nodes, no node failure latency



(b) 7 Nodes, 3 node failure latency

Figure 5.10: Latency in Network C – 7 nodes

Additionally, Figure 5.11 shows each latency value plotted on the same diagram

with the results comparable to the ones achieved by Pötsch and Hammer (2019) in their work on end-to-end latency of LoRaWAN.

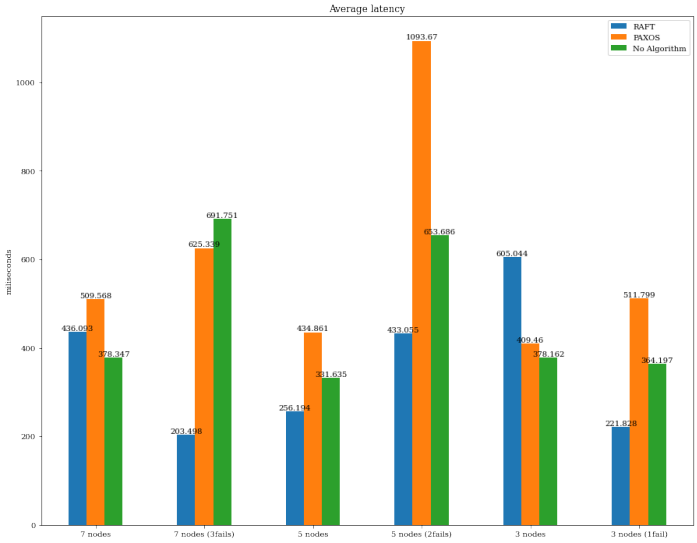


Figure 5.11: Average latency of all networks (A, B and C)

As demonstrated in previous figures, IoT networks with Raft implementation have an advantage (in most cases) in regards to decreasing latency on IoT networks, over IoT networks with either Paxos implementation or no consensus algorithm at all.

	Experiment	RAFT	PAXOS	No Algorithm
0	7 nodes	436.093431	509.567698	378.346520
1	7 nodes (3fails)	203.498192	625.339420	691.750815
2	5 nodes	256.193596	434.860850	331.635032
3	5 nodes (2fails)	433.054533	1093.673450	653.685699
4	3 nodes	605.044012	409.459693	378.161590
5	3 nodes (1fail)	221.827970	511.799328	364.197263

Table 5.14: Average latency of all networks (A, B and C)

5.4 Discussion

5.4.1 Results Evaluation

From the results in Packet Loss Rate experiment section, author can clearly see that there is an advantage in implementing Raft consensus algorithm on IoT networks to

decrease possible packet loss rate. Not surprisingly, Paxos consensus algorithm implementation delivers similar results as Raft algorithm. Both networks with consensus algorithms implemented scheduling mechanisms that used synchronized counter to introduce order in the packets' expedition. Once possibility of packet collisions was minimized by allowing only one node to send packets at given time, this significantly reduces lost packets. Packet loss experiments focused on deliverability of the message and in all test cases, their results proved that scheduling by implementing consensus algorithm has potentially huge benefits. This partially answers the research question related to decreasing packet loss rate by implementing Raft on IoT networks by proving that Raft implementation can potentially deliver huge benefits for LoRa operators.

In terms of decreasing latency in IoT network with the help of Raft consensus algorithm, the author has to admit that, although results clearly point to the advantage of Raft over Paxos and IoT networks without consensus algorithm, there are other aspects that were not taken into consideration during the experiment. While there might be some concerns regarding the outcome of the latency experiments, the author is inclined to assume that the research question has been answered. Additionally, to answer research question, results clearly provide enough proof to reject null hypothesis H0 and confirm hypothesis H1 and H3. Based on the results from the latency experiment, author can confirm as well hypothesis H2, unfortunately, there is not enough evidence to confirm hypothesis H4. While Raft implementation outperformed Paxos implementation in some of the experiments, the differences were inconclusive.

5.4.2 Strengths and Limitations

The key strength of this thesis is the consistency of the Packet Loss Rate experiments that clearly answer the research question in regards to the possible benefits of Raft algorithm implementation in IoT networks and its impact on packet loss rate. In proving that Raft is of relevance from the perspective of IoT, author believes that this opens new possible research directions and allows for the discussion on implementing consensus algorithms in IoT networks to take place.

The main limitation of this research is the physical environment where the experiments took place. The author believes that LoRa is the future of IoT communication and, with this in mind, recognizes that this research would benefit from additional experiments performed on a larger scale. LoRa is able to achieve reliable communication over a distance of up to 7 kilometres, and as such it would benefit future research if the experiments could be conducted over the longer distances with tens or possibly hundreds of nodes. Additionally, latency experiment should have to consider in the future the impact of weather conditions (Lavdas, Zacharioudakis, Khalifeh, & Zinonos, 2021) and the clock drift (Zorbas et al., 2021) on the time measurement.

Very importantly as well, author believes that future research should focus on implementing Raft consensus algorithm directly on LoRa ESP32 board. Currently boards used in these experiments have 8MB SPI Flash memory with 520KB of SRAM, these specs should allow for Python 3 library used by author to be adapted to MicroPython and run directly on the board as an embedded program.

5.4.3 Conclusion

This section covered the results acquired during the experiments described in depth in chapters three and four of this thesis. Each of the presented results were subdivided into six test cases based on the type of the experiment performed. Commentary was provided on these results, with some notable trends highlighted. The research question was answered in this chapter and the alternate hypotheses H1, H2 and H3 were accepted. Hypothesis H4 was rejected.

The next Chapter will conclude this thesis, with an overview of the results accomplished during both experiments, evaluation of the results obtained, and additionally proposed future work will be discussed.

Chapter 6

Conclusion

This section outlines the entire thesis, reviewing the research objective and answers the research question. Finally, the contributions, future work, impact and recommendations are presented and discussed.

6.1 Research Overview

The goal of this thesis was to provide empirical evidence that implementation of Raft consensus algorithm in IoT network can decrease packet loss rate and improve networks latency. In Chapter One author introduced the research topic and some basic overview of the research subject. Chapter Two presented existing research into the topic of consensus algorithms implementation in IoT domain, and identified gaps in the literature around Raft consensus algorithm implementation, noticing that no existing literature focused on scheduling LoRa packets on IoT nodes with the help of Raft algorithm, or any other consensus algorithm. This gap was used as a foundation of this research topic. In chapters Three and Four, the author presented the experiment design and implementation. Additionally, details were provided on how to run the experiments and how to obtain the resulting datasets. Finally, Chapter Five concluded with an overview of the results from both experiments and discussion on trends identified within the results dataset.

6.2 Problem Definition

In this thesis, the author aimed to prove that implementation of Raft consensus algorithm can improve communication between IoT nodes and the IoT gateway when using LoRa as the communication medium. Author formed the following research question: "Can implementation of Raft consensus algorithm in IoT network decrease packet loss rate and improve latency on the network in the presence of malfunctioning edge nodes when compared to Paxos consensus algorithm?" From this research question the following research objectives were formed:

1. Create three IoT networks consisting of three, five and seven nodes.
2. Implement Raft and Paxos on each network independently.
3. Generate dataset from LoRa transmissions based on varying inputs with Raft consensus algorithm implemented.
4. Generate dataset from LoRa transmissions based on varying inputs with Paxos consensus algorithm implemented.
5. Generate dataset from LoRa transmissions based on varying inputs without consensus algorithm implemented.
6. Compare datasets to identify packet loss rate and latency in IoT networks for each consensus algorithm and control experiment.
7. Identify limiting factors, points of interest in the data and identify future research

Null hypothesis H0:

Based on the `p-value` below the threshold of `0.05` Null hypothesis (H0) can be rejected.

Alternative hypothesis H1:

The packet loss rate is significantly lower on the IoT networks with implemented Raft consensus algorithm when compared to the IoT network with Paxos algorithm and

IoT network without consensus algorithm implemented.

Alternative hypothesis H2:

The latency of IoT networks with Raft implementation, in the presence of faulty nodes, is considerably lower when compared to the IoT networks with Paxos, and without consensus algorithm implemented.

Alternative hypothesis H3:

The IoT networks with Raft implementation show notable decrease in the packet loss rate in the presence of faulty nodes.

Alternative hypothesis H4:

Neither of the experiments provided enough evidence to confirm hypothesis H4, therefore, the author rejects H4.

6.3 Design/Experimentation, Evaluation & Results

For the purpose of this thesis author created three IoT networks, network of three IoT nodes, network of five IoT nodes and network of seven IoT nodes. Six test cases were identified, three simulated each of the IoT networks without running without the failure and three simulating failure of different number of nodes. For each test case author implemented Raft consensus, Paxos algorithms and ran control experiments where no consensus algorithm was implemented. Results from each tests were collected and saved to CSV file for further analysis. For the purpose of the experiments, data was collected on transmitter nodes and cumulative data from each node was collected on transmitter node. This allowed to cross-examine packets sent with packets received to calculate packet loss ratio. Additional data fields were added to the dataset to allow for recording of the timestamps required for the latency experiment. All datasets were cleaned and processed in Jupyter Notebook with Python 3 Pandas library. Once reviewed, results of the experiments clearly shown that scheduler based on Raft consensus algorithm provides an advantage in decreasing packet loss rate and latency over the no algorithm implementation. Results did not show Raft to have a

significant advantage over Paxos implementation though. Author can therefore confirm that null hypothesis can be rejected and alternative hypotheses H1, H2 and H3 can be confirmed. There is no sufficient evidence to confirm alternative hypothesis H4, hence author decided to reject it.

6.4 Contributions and impact

The main goal of this thesis was to build on work presented in Chapter Two and fill in the gap identified. In this thesis author focused on packet loss rate and latency in IoT networks where scheduler based on Raft and Paxos was compared against the IoT network without scheduling. The result clearly showed that implementing consensus-based scheduling for LoRa transmissions in IoT network has significant advantage over the network without consensus-based scheduler. In Chapters Three and Four design and implementation of the experiments were presented with details and steps required to create IoT network with LoRa capabilities were clearly presented. One of the biggest challenges in executing the experiments was the connectivity of LoRa board and Raspberry Pi and implementation of ad hoc network. Author hopes that this thesis will simplify steps required for setting up the IoT network for future researchers into the topic of consensus in IoT domain.

6.5 Future Work & recommendations

- Author believes that this research could be extended by scaling up the IoT network itself and testing it in the larger area where LoRa capabilities could be clearly shown. Implementation of Raft algorithm-based scheduler across greater number of nodes could show additional benefits in regards to avoiding packet collisions and improving packet loss rate.

- Implementing consensus algorithm embedding it directly on the LoRa ESP32 board could benefit further research by lowering the hardware complexity.

- While researching latency topic in LoRa based communication, it would greatly benefit future researchers to focus on clock-drift in LoRa ESP32 boards.

References

Abdelwahab, S., Hamdaoui, B., Guizani, M., & Znati, T. (2015). Cloud of things for sensing as a service: Sensing resource discovery and virtualization. In *2015 ieee global communications conference (globecom)* (p. 1-7). doi: 10.1109/GLOCOM .2015.7417252

Ahmad, S., & Kim, D. (2020). A multi-device multi-tasks management and orchestration architecture for the design of enterprise iot applications. *Future Generation Computer Systems*, *106*, 482-500. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0167739X19318035` doi: https://doi.org/10.1016/j.future.2019.11.030

Al-Doghman, F., Chaczko, Z., & Brookes, W. (2018). Adaptive consensus-based aggregation for edge computing. In *2018 26th international conference on systems engineering (icseng)* (p. 1-8). doi: 10.1109/ICSENG.2018.8638200

Bahreini, T., & Grosu, D. (2017). Efficient placement of multi-component applications in edge computing systems. In *Proceedings of the second acm/ieee symposium on edge computing.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3132211.3134454

Barry, P., & Crowley, P. (2012). Chapter 4 - embedded platform architecture. In *Modern embedded computing. designing connected, pervasive, media-rich systems* (p. 41-97). Elsevier Inc. doi: https://doi.org/10.1016/C2011-0-05083-4

Bhattacharjee, S., Salimitari, M., Chatterjee, M., Kwiat, K., & Kamhoua, C. (2017). Preserving data integrity in iot networks under opportunistic data manipulation. In *2017 ieee 15th intl conf on dependable, autonomic and secure computing, 15th intl conf on pervasive intelligence and computing, 3rd*

intl conf on big data intelligence and computing and cyber science and technology congress(dasc/picom/datacom/cyberscitech) (p. 446-453). doi: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.87

Blenn, N., & Kuipers, F. (2017). *Lorawan in the wild: Measurements from the things network.* `https://arxiv.org/abs/1706.03086`. arXiv. doi: 10.48550/ARXIV.1706.03086

Bof, N., Carli, R., & Schenato, L. (2017). Average consensus with asynchronous updates and unreliable communication. *IFAC-PapersOnLine*, *50*(1), 601 - 606. (20th IFAC World Congress) doi: https://doi.org/10.1016/j.ifacol.2017.08.093

Cachin, C. (2010). *IBM Research. Yet Another Visit to Paxos.* `https://dominoweb.draco.res.ibm.com/5233d5f926b64f2a8525766b00383ec9.html`. (Accessed: 2022-03-12)

Campbell, S. (2020). *Circuit Basics. Basics of UART communication.* `https://www.circuitbasics.com/basics-uart-communication`. (Accessed: 2022-05-08)

Castellano, G., Esposito, F., & Risso, F. (2019). A distributed orchestration algorithm for edge computing resources with guarantees. In *Ieee infocom 2019 - ieee conference on computer communications* (p. 2548-2556). doi: 10.1109/INFOCOM.2019.8737532

Chamoso, P., González-Briones, A., De La Prieta, F., Venyagamoorthy, G. K., & Corchado, J. M. (2020). Smart city as a distributed platform: Toward a system for citizen-oriented management. *Computer Communications*, *152*, 323-332. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0140366419321152` doi: https://doi.org/10.1016/j.comcom.2020.01.059

Chaudhari, B., & Borkar, S. (2020). 2 - design considerations and network architectures for low-power wide-area networks. In B. S. Chaudhari & M. Zennaro (Eds.), *Lpwan technologies for iot and m2m applications* (p. 15-35). Academic Press. Retrieved from `https://www.sciencedirect.com/science/article/pii/B9780128188804000028` doi: https://doi.org/10.1016/B978-0-12-818880-4.00002-8

Chien, S.-Y., Chan, W.-K., Tseng, Y.-H., Lee, C.-H., Somayazulu, V. S., & Chen, Y.-K. (2015). Distributed computing in iot: System-on-a-chip for smart cameras as an example. In *The 20th asia and south pacific design automation conference* (p. 130-135). doi: 10.1109/ASPDAC.2015.7058993

Cocagne, T. (2013). *GitHub. composable paxos Python 3 Library.* `https://github.com/cocagne/multi-paxos-example`. (Accessed: 2022-03-08)

Cui, Y., Yang, N., & Liu, J. J. (2020). A novel approach for positive edge consensus of nodal networks. *Journal of the Franklin Institute*, *357*(7), 4349 - 4362. doi: https://doi.org/10.1016/j.jfranklin.2020.02.054

Dorri, A., & Jurdak, R. (2020). *Researchgate Tree-Chain: A Fast Lightweight Consensus Algorithm for IoT Applications.* `https://www.researchgate.net/publication/341507295_Tree-Chain_A_Fast_Lightweight_Consensus_Algorithm_for_IoT_Applications/`. (Accessed: 2022-03-12)

Ferre, G. (2017, 08). Collision and packet loss analysis in a lorawan network. In (p. 2586-2590). doi: 10.23919/EUSIPCO.2017.8081678

Fortino, G., Fotia, L., Messina, F., Rosaci, D., & Sarné, G. (2020). A meritocratic trust-based group formation in an iot environment for smart cities. *Future Generation Computer Systems*, *108*, 34 - 45. doi: https://doi.org/10.1016/j.future.2020.02.035

Gramoli, V. (2020). From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems*, *107*, 760 - 769. doi: https://doi.org/10.1016/j.future.2017.09.023

Guerrero, C., Lera, I., & Juiz, C. (2019). Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. *Future Generation Computer Systems*, *97*, 131 - 144. doi: https://doi.org/10.1016/j.future.2019.02.056

Gulati, N., & Kaur, P. D. (2020). A game theoretic approach for conflict resolution in argumentation enabled social iot networks. *Ad Hoc Networks*, *107*, 102222. doi: https://doi.org/10.1016/j.adhoc.2020.102222

Hao, Z., Yi, S., & Li, Q. (2018). Edgecons: Achieving efficient consensus in edge

computing networks. In *Hotedge*.

Hernández-Solana, , Pérez-Díaz-De-Cerio, D., García-Lozano, M., Bardají, A. V., & Valenzuela, J.-L. (2020). Bluetooth mesh analysis, issues, and challenges. *IEEE Access*, *8*, 53784-53800. doi: 10.1109/ACCESS.2020.2980795

Howard, H. (2014, July). *ARC: Analysis of Raft Consensus* (Tech. Rep. No. UCAM-CL-TR-857). University of Cambridge, Computer Laboratory. Retrieved from `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-857.pdf`

Howard, H., & Mortier, R. (2020). Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th workshop on principles and practice of consistency for distributed data.* New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3380787.3393681`

ICAR. (2020). *ICAR Distributed Systems and Internet of Things. Institute for high performance computing and networking.* `https://www.icar.cnr.it/en/sistemi-distribuiti-e-internet-delle-cose/`. (Accessed: 2022-02-14)

Ismail, Y. (2019). Introductory chapter: Internet of things (iot) importance and its applications. In *Internet of things (iot) for automated and smart applications.* IntechOpen. doi: https://doi.org/10.5772/intechopen.90022

István, Z., Sidler, D., Alonso, G., & Vukolic, M. (2016, March). Consensus in a box: Inexpensive coordination in hardware. In *13th usenix symposium on networked systems design and implementation (nsdi 16)* (pp. 425–438). Santa Clara, CA: USENIX Association. Retrieved from `https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan`

Ji, Y. (2021). Application of fault detection using distributed sensors in smart cities. *Physical Communication*, *46*, 101182. Retrieved from `https://www.sciencedirect.com/science/article/pii/S1874490720302597` doi: https://doi.org/10.1016/j.phycom.2020.101182

Krzyzanowski, P. (2018). *Rutgers University Understanding Paxos. Asynchronous Fault-Tolerant Consensus.* `https://people.cs.rutgers.edu/~pxk/417/notes/paxos.html`. (Accessed: 2022-02-12)

Kumari, S., & Rohil, H. (2014, 06). Taxonomy of distributed consensus algorithms. *International Journal of Computer Science and Information Technologies*, *5*, 3921- 3923.

Kurji, A. S., Al-Nakkash, A. H., & Hussein, O. A. (2021, jun). LORA in a campus: Reliability and stability testing. *IOP Conference Series: Materials Science and Engineering*, *1105*(1), 012034. Retrieved from https://doi.org/10.1088/1757-899x/1105/1/012034 doi: 10.1088/1757-899x/1105/1/012034

Lamport, L. (1998, May). The part-time parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60]..* Retrieved from https://www.microsoft.com/en-us/research/publication/part-time-parliament/ (ACM SIGOPS Hall of Fame Award in 2012)

Lamport, L. (2001, December). Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, 51-58. Retrieved from https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

Lamport, L. (2005). *Microsoft Generalized Consensus and Paxos.* https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/. (Accessed: 2022-02-24)

Lamport, L., Shostak, R., & Pease, M. (1982, jul). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, *4*(3), 382–401. Retrieved from https://doi.org/10.1145/357172.357176 doi: 10.1145/357172.357176

Lavdas, S., Zacharioudakis, L., Khalifeh, A., & Zinonos, Z. (2021). The effect of temperature and humidity on indoor lora propagation model. In *2021 17th international conference on distributed computing in sensor systems (dcoss)* (p. 374-379). doi: 10.1109/DCOSS52077.2021.00066

Lee, E. A., Hartmann, B., Kubiatowicz, J., Simunic Rosing, T., Wawrzynek, J., Wessel, D., ... Rowe, A. (2014). The swarm at the edge of the cloud. *IEEE Design Test*, *31*(3), 8-20. doi: 10.1109/MDAT.2014.2314600

Li, J. (2020). Resource optimization scheduling and allocation for hierarchical distributed cloud service system in smart city. *Future Generation Computer Systems*, *107*, 247-256. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167739X1932028X doi: https://doi.org/10.1016/j.future.2019.12.040

Li, S., Oikonomou, G., Tryfonas, T., Chen, T. M., & Xu, L. D. (2014). A distributed consensus algorithm for decision making in service-oriented internet of things. *IEEE Transactions on Industrial Informatics*, *10*(2), 1461-1468. doi: 10.1109/TII.2014.2306331

Li, W., & He, M. (2020). Imp raft: a consensus algorithm based on raft and storage compression consensus for iot scenario. *The Journal of China Universities of Posts and Telecommunications*, *27*(3), 53-61.

Liang, R., Zhao, L., & Wang, P. (2020, 07). Performance evaluations of lora wireless communication in building environments. *Sensors*, *20*, 3828. doi: 10.3390/s20143828

Mondal, S., & Tsourdos, A. (2020). Optimal topology for consensus using genetic algorithm. *Neurocomputing*, *404*, 41 - 49. doi: https://doi.org/10.1016/j.neucom.2020.04.107

Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 usenix conference on usenix annual technical conference* (p. 305–320). USA: USENIX Association.

Oróstica, B., & Núñez, F. (2019). A multi-cast algorithm for robust average consensus over internet of things environments. *Computer Communications*, *140-141*, 15 - 22. doi: https://doi.org/10.1016/j.comcom.2019.04.007

Ozinov, F. (2022). *GitHub. PySyncObj Python 3 Library.* https://github.com/bakwc/PySyncObj. (Accessed: 2022-03-05)

Poirot, V., Al Nahas, B., & Landsiedel, O. (2019). Paxos made wireless: Consensus in the air. In *Proceedings of the 2019 international conference on embedded wireless systems and networks* (p. 1–12). USA: Junction Publishing.

PUDER, A., RÖMER, K., & PILHOFER, F. (2006). Chapter 2 - basic concepts. In

A. PUDER, K. RÖMER, & F. PILHOFER (Eds.), *Distributed systems architecture* (p. 7-31). San Francisco: Morgan Kaufmann. Retrieved from https://www.sciencedirect.com/science/article/pii/B9781558606487500036 doi: https://doi.org/10.1016/B978-155860648-7/50003-6

Pötsch, A., & Hammer, F. (2019). Towards end-to-end latency of lorawan: Experimental analysis and iiot applicability. In *2019 15th ieee international workshop on factory communication systems (wfcs)* (p. 1-4). doi: 10.1109/WFCS.2019.8758033

Qin, J., Ma, Q., Shi, Y., & Wang, L. (2017). Recent advances in consensus of multi-agent systems: A brief survey. *IEEE Transactions on Industrial Electronics*, *64*(6), 4972-4983. doi: 10.1109/TIE.2016.2636810

Raghav, Andola, N., Venkatesan, S., & Verma, S. (2020). Poewal: A lightweight consensus mechanism for blockchain in iot. *Pervasive and Mobile Computing*, *69*, 101291. doi: https://doi.org/10.1016/j.pmcj.2020.101291

Rajab, H., Cinkler, T., & Bouguera, T. (2021, 04). Iot scheduling for higher throughput and lower transmission power. *Wireless Networks*, *27*. doi: 10.1007/s11276-020-02307-1

Rocha, V., & Brandão, A. A. F. (2019, aug). A scalable multiagent architecture for monitoring iot devices. *J. Netw. Comput. Appl.*, *139*(C), 1–14. Retrieved from https://doi.org/10.1016/j.jnca.2019.04.017 doi: 10.1016/j.jnca.2019.04.017

Saelens, M., Hoebeke, J., Shahid, A., & Poorter, E. D. (2019). Impact of EU duty cycle and transmission power limitations for sub-GHz LPWAN SRDs: an overview and future challenges. *EURASIP Journal on Wireless Communications and Networking*, *2019*. doi: https://doi.org/10.1186/s13638-019-1502-5

Salimitari, M., Chatterjee, M., & Fallah, Y. P. (2020). A survey on consensus methods in blockchain for resource-constrained iot networks. *Internet of Things*, *11*, 100212. doi: https://doi.org/10.1016/j.iot.2020.100212

Shi, C.-X., & Yang, G.-H. (2018). Robust consensus control for a class of multi-agent systems via distributed pid algorithm and weighted edge dynamics. *Applied*

Mathematics and Computation, *316*, 73 - 88. doi: https://doi.org/10.1016/j.amc.2017.07.069

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, *3*(5), 637-646. doi: 10.1109/JIOT.2016.2579198

Shyamala Devi, M., Suguna, R., Joshi, A. S., & Bagate, R. A. (2019). Design of iot blockchain based smart agriculture for enlightening safety and security. In A. K. Somani, S. Ramakrishna, A. Chaudhary, C. Choudhary, & B. Agarwal (Eds.), *Emerging technologies in computer engineering: Microservices in big data analytics* (pp. 7–19). Singapore: Springer Singapore.

Skrzypzcak, J., & Schintke, F. (2020). Towards log-less, fine-granular state machine replication. *Datenbank-Spektrum*, *20*(3), 231-241. doi: https://doi.org/10.1007/s13222-020-00358-4

Tošić, A., Vičič, J., & Mrissa, M. (2019). A blockchain-based decentralized self-balancing architecture for the web of things. In T. Welzer et al. (Eds.), *New trends in databases and information systems* (pp. 325–336). Cham: Springer International Publishing.

Wang, E. K., Sun, R., Chen, C.-M., Liang, Z., Kumari, S., & Khurram Khan, M. (2020). Proof of x-repute blockchain consensus protocol for iot systems. *Computers 'I&' Security*, *95*, 101871. doi: https://doi.org/10.1016/j.cose.2020.101871

Whittaker, M., Giridharan, N., Szekeres, A., Hellerstein, J. M., Howard, H., Nawab, F., & Stoica, I. (2020). *Matchmaker paxos: A reconfigurable consensus protocol [technical report].* doi: https://arxiv.org/abs/2007.09468

Wohlin, C., & Runeson, P. (2021). Guiding the selection of research methodology in industry–academia collaboration in software engineering. *Information and Software Technology*, *140*, 106678. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0950584921001361` doi: https://doi.org/10.1016/j.infsof.2021.106678

Xiaoyi, Z., Dongling, W., Yuming, Z., Manokaran, K. B., & Benny Antony, A. (2021). Iot driven framework based efficient green energy management in

smart cities using multi-objective distributed dispatching algorithm. *Environmental Impact Assessment Review*, *88*, 106567. Retrieved from https://www.sciencedirect.com/science/article/pii/S0195925521000172 doi: https://doi.org/10.1016/j.eiar.2021.106567

Yang, T., Wu, D., Sun, Y., & Lian, J. (2016). Minimum-time consensus-based approach for power system applications. *IEEE Transactions on Industrial Electronics*, *63*(2), 1318-1328. doi: 10.1109/TIE.2015.2504050

Yi, S., Hao, Z., Qin, Z., & Li, Q. (2015). Fog computing: Platform and applications. In *2015 third ieee workshop on hot topics in web systems and technologies (hotweb)* (p. 73-78). doi: 10.1109/HotWeb.2015.22

Yogesh, M., Patel, S., & Patel, Y. (2022). Social iot. *Indian Scientific Journal Of Research In Engineering And Management*, *06*.

Zhang, W., Liu, W., Wang, X., Liu, L., & Ferrese, F. (2015). Online optimal generation control based on constrained distributed gradient algorithm. *IEEE Transactions on Power Systems*, *30*(1), 35-45. doi: 10.1109/TPWRS.2014.2319315

Zhang, W., Wu, Z., Han, G., Feng, Y., & Shu, L. (2020). Ldc: A lightweight data consensus algorithm based on the blockchain for the industrial internet of things for smart city applications. *Future Generation Computer Systems*, *108*, 574 - 582. doi: https://doi.org/10.1016/j.future.2020.03.009

Zorbas, D., Caillouet, C., Abdelfadeel Hassan, K., & Pesch, D. (2021). Optimal data collection time in lora networks—a time-slotted approach. *Sensors*, *21*(4). Retrieved from https://www.mdpi.com/1424-8220/21/4/1193 doi: 10.3390/s21041193

Appendix A

Code Snippets

The results of the experiments and the code are publicly available on a GitHub Repository https://github.com/bartoszczapski/msc_thesis

Listing A.1 represents Python 3 code of Sender node implementation in control experiment (no consensus algorithm implemented).

```
#!/usr/bin/python3
import serial
import time
import smbus
import csv


def createData():
    DEVICE_BUS = 1
    DEVICE_ADDR = 0x17


    TEMP_REG = 0x01
    LIGHT_REG_L = 0x02
    LIGHT_REG_H = 0x03
    STATUS_REG = 0x04
    ON_BOARD_TEMP_REG = 0x05
    ON_BOARD_HUMIDITY_REG = 0x06
```

```
    ON_BOARD_SENSOR_ERROR = 0x07
    BMP280_TEMP_REG = 0x08
    BMP280_PRESSURE_REG_L = 0x09
    BMP280_PRESSURE_REG_M = 0x0A
    BMP280_PRESSURE_REG_H = 0x0B
    BMP280_STATUS = 0x0C
    HUMAN_DETECT = 0x0D

    bus = smbus.SMBus(DEVICE_BUS)

    aReceiveBuf = []

    aReceiveBuf.append(0x00)

    for i in range(TEMP_REG,HUMAN_DETECT + 1):
        aReceiveBuf.append(bus.read_byte_data(DEVICE_ADDR, i))

    #return str(aReceiveBuf[TEMP_REG]), str((aReceiveBuf[LIGHT_REG_H]
    ↪  << 8 | aReceiveBuf[LIGHT_REG_L])), str(aReceiveBuf[
    ↪ ON_BOARD_HUMIDITY_REG]), str((aReceiveBuf[
    ↪ BMP280_PRESSURE_REG_L] | aReceiveBuf[BMP280_PRESSURE_REG_M]
    ↪  << 8 | aReceiveBuf[BMP280_PRESSURE_REG_H] << 16))
    return aReceiveBuf[TEMP_REG], (aReceiveBuf[LIGHT_REG_H] << 8 |
    ↪ aReceiveBuf[LIGHT_REG_L]), aReceiveBuf[ON_BOARD_HUMIDITY_REG
    ↪ ], aReceiveBuf[ON_BOARD_TEMP_REG]

def main():
    #set UART serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #declare node name
```

```
node = 4
#initialize packet counter
count = 1
#temp, light, humidity, onBoardTemp = createData()
#print(node, count, temp, light, humidity, pressure)
#declare header for csv file
header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
    ↪ 'humidity', 'onBoardTemp']

#create csv file
with open('sensor-4_data.csv', 'w', encoding='UTF8', newline='') as
    ↪  f:
    #writer object
    writer = csv.writer(f)
    #write the header to the csv file
    writer.writerow(header)

    #loop
    while True:
        #read sensor hub data
        temp, light, humidity, onBoardTemp = createData()
        #write data to csv file
        data = [node, count, temp, light, humidity, onBoardTemp]
        #send packet over lora
        s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light,
            ↪ humidity, onBoardTemp))
        #save packet to the csv file
                writer.writerow(data)
        #print(count)
        count += 1
```

```
            time.sleep(2)


if __name__ == '__main__':
    main()
    os.system('sudo␣poweroff')
```

Listing A.1: Sender code without consensus algorithm implementation- Python3

Listing A.2 shows Python 3 code for Raft node on Raspberry Pi.

```
#!/usr/bin/python3


from pysyncobj import *
import serial
import time
import smbus
import csv
import os

class RaftObj(SyncObj):

    def __init__(self, selfNode, otherNodes):
        super(TestObj, self).__init__(selfNode,otherNodes)
        self.__counter = 1

    @replicated
    def setCounter(self, value):
        self.__counter = value
        return self.__counter

    @replicated
    def addValue(self, value):
```

```
        self.__counter += value
        return self.__counter

    def getCounter(self):
        return self.__counter

    def createData(self):

        DEVICE_BUS = 1
        DEVICE_ADDR = 0x17

        TEMP_REG = 0x01
        LIGHT_REG_L = 0x02
        LIGHT_REG_H = 0x03
        STATUS_REG = 0x04
        ON_BOARD_TEMP_REG = 0x05
        ON_BOARD_HUMIDITY_REG = 0x06
        ON_BOARD_SENSOR_ERROR = 0x07
        BMP280_TEMP_REG = 0x08
        BMP280_PRESSURE_REG_L = 0x09
        BMP280_PRESSURE_REG_M = 0x0A
        BMP280_PRESSURE_REG_H = 0x0B
        BMP280_STATUS = 0x0C
        HUMAN_DETECT = 0x0D

        bus = smbus.SMBus(DEVICE_BUS)

        aReceiveBuf = []

        aReceiveBuf.append(0x00)
```

```
        for i in range(TEMP_REG,HUMAN_DETECT + 1):
            aReceiveBuf.append(bus.read_byte_data(DEVICE_ADDR, i))

        #return str(aReceiveBuf[TEMP_REG]), str((aReceiveBuf[
            ↪ LIGHT_REG_H] << 8 | aReceiveBuf[LIGHT_REG_L])), str(
            ↪ aReceiveBuf[ON_BOARD_HUMIDITY_REG]), str((aReceiveBuf[
            ↪ BMP280_PRESSURE_REG_L] | aReceiveBuf[
            ↪ BMP280_PRESSURE_REG_M] << 8 | aReceiveBuf[
            ↪ BMP280_PRESSURE_REG_H] << 16))
        return aReceiveBuf[TEMP_REG], (aReceiveBuf[LIGHT_REG_H] << 8 |
            ↪ aReceiveBuf[LIGHT_REG_L]), aReceiveBuf[
            ↪ ON_BOARD_HUMIDITY_REG], aReceiveBuf[ON_BOARD_TEMP_REG]

def main():
    my_ip = "192.168.1.1"
    o = RaftObj('192.168.1.1:12345', ['192.168.1.2:12345', '
        ↪ 192.168.1.3:12345'])
    old_value = -1
    filename = 'sensor-1_data_raft.csv'
    #set serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #declare node name
    node = 1
    #initialize packet counter
    count = 1
    #temp, light, humidity, onBoardTemp = createData()
    #print(node, count, temp, light, humidity, pressure)
    #declare header for csv file
    header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
```

```
    ↪ 'humidity', 'onBoardTemp']

#create csv file
with open(filename, 'w', encoding='UTF8', newline='') as f:
    #writer object
    writer = csv.writer(f)
    #write the header to the csv file
    writer.writerow(header)
    #loop
    while count <= 1000:
        time.sleep(2)
        if o.getCounter() != old_value:
            old_value = o.getCounter()
        if str(o._getLeader()).split(":", 1)[0] == my_ip:
            if o.getCounter() <= 3:
                o.addValue(1)
            else:
                o.setCounter(1)
        if o.getCounter() == node:
            #read sensor hub data
            temp, light, humidity, onBoardTemp = o.createData()
            #write data to csv file
            data = [node, count, temp, light, humidity, onBoardTemp]
            writer.writerow(data)
            #send packet over lora
            s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light
                ↪ , humidity, onBoardTemp))
            #print("COUNTER: %d" % o.getCounter())

        count+=1
```

```
        f.flush()
        f.close()


if __name__ == '__main__':
    main()
    os.system('sudo␣poweroff')
```

Listing A.2: Raft Sender code - Python3

Listing A.3 shows Python 3 code for Paxos node on Raspberry Pi.

```
#!/usr/bin/python3


from client import ClientProtocol
from server import Server
from composable_paxos import *
from master_strategy import *
from messenger import *
from replicated_value import *
from resolution_strategy import *
from sync_strategy import *
import config
import serial
import time
import smbus
import csv
import os


def createData(self):


        DEVICE_BUS = 1
        DEVICE_ADDR = 0x17
```

```
TEMP_REG = 0x01
LIGHT_REG_L = 0x02
LIGHT_REG_H = 0x03
STATUS_REG = 0x04
ON_BOARD_TEMP_REG = 0x05
ON_BOARD_HUMIDITY_REG = 0x06
ON_BOARD_SENSOR_ERROR = 0x07
BMP280_TEMP_REG = 0x08
BMP280_PRESSURE_REG_L = 0x09
BMP280_PRESSURE_REG_M = 0x0A
BMP280_PRESSURE_REG_H = 0x0B
BMP280_STATUS = 0x0C
HUMAN_DETECT = 0x0D

bus = smbus.SMBus(DEVICE_BUS)

aReceiveBuf = []

aReceiveBuf.append(0x00)

for i in range(TEMP_REG,HUMAN_DETECT + 1):
        aReceiveBuf.append(bus.read_byte_data(DEVICE_ADDR, i))

#return str(aReceiveBuf[TEMP_REG]), str((aReceiveBuf[
    ↪ LIGHT_REG_H] << 8 | aReceiveBuf[LIGHT_REG_L])), str(
    ↪ aReceiveBuf[ON_BOARD_HUMIDITY_REG]), str((aReceiveBuf[
    ↪ BMP280_PRESSURE_REG_L] | aReceiveBuf[
    ↪ BMP280_PRESSURE_REG_M] << 8 | aReceiveBuf[
    ↪ BMP280_PRESSURE_REG_H] << 16))
```

```
        return aReceiveBuf[TEMP_REG], (aReceiveBuf[LIGHT_REG_H] << 8 |
            ↪ aReceiveBuf[LIGHT_REG_L]), aReceiveBuf[
            ↪ ON_BOARD_HUMIDITY_REG], aReceiveBuf[ON_BOARD_TEMP_REG]


def main():
        #declare node name
    node = 1
    filename = 'sensor-1_data_raft.csv'
    #initialize Server obj
        serv = Server(node)
    #set serial connection with ESP32
    s = serial.Serial('/dev/ttyS0', 115200)
    #initialize packet counter
    count = 1
        #node counter
        counter = 1
    #declare header for csv file
    header = ['nodeName', 'packetNumber', 'temperature', 'brightness',
        ↪ 'humidity', 'onBoardTemp']


    #create csv file
    with open(filename, 'w', encoding='UTF8', newline='') as f:
        #writer object
        writer = csv.writer(f)
        #write the header to the csv file
        writer.writerow(header)
        #loop
        while count <= 1000:
                    #2 sec interval
            time.sleep(2)
```

```
        #check if this node is master node
        if serv.get_master() == node:
            if counter <= 3:
                #increment counter
                counter+=1
                                #initialize Client object and
                                    ↪ propose new value
                                ClientProtocol(node, counter)
            else:
                                #restart counter
                counter = 1
                                #initialize Client object and
                                    ↪ propose new value
                                ClientProtocol(node, counter)
                    #if the value of the counter matches node number
                        ↪ , sent packet
        if counter == node:
            #read sensor hub data
            temp, light, humidity, onBoardTemp = o.createData()
            #write data to csv file
            data = [node, count, temp, light, humidity, onBoardTemp]
            writer.writerow(data)
            #send packet over lora
            s.write(b'%d,%d,%d,%d,%d,%d' % (node, count, temp, light
                ↪ , humidity, onBoardTemp))
            #print("COUNTER: %d" % o.getCounter())
        #increment packet count
        count+=1
    f.flush()
    f.close()
```

```
if __name__ == '__main__':
    main()
    os.system('sudo␣poweroff')
```

Listing A.3: Paxos Sender code - Python3