Scalable Practical Byzantine Consensus in Wireless Sensor Networks

A report submitted in partial fulfillment of the requirements of the requirements for the Project Part-II

Dual Degree Programme - B. Tech. and M.Tech

in

Computer Science & Engineering

by

Himanshu Goyal
17CS02011

Under the supervision of

Dr. Sudipta Saha

Department of Computer Science and Engineering
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar
Orissa, India - 752050

DECLARATION BY THE SCHOLAR

I certify that:

- The work contained in this thesis is original and has been done by me under the guidance of my supervisor.
- The work has not been submitted to any other Institute for any degree or diploma.
- I have followed the guidelines provided by the Institute in preparing the thesis.
- I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.
- Whenever I have used materials (data, theory and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the reference section.
- In case if something is used and it belongs to you, let author know about it. He will either attribute you or take it out from here.

Himanshu Goyal

CERTIFICATE

This is to certify that the thesis entitled **"Scalable Practical Byzantine Consensus in Wireless Sensor Networks"**, submitted by **Himanshu Goyal** to Indian Institute of Technology Bhubaneswar, is a record of bonafide research work under my supervision and I consider it worthy of consideration for the degree of Dual Degree Programme - B. Tech. and M.Tech of the Institute.

Place: IIT Bhubaneswar
Date: May 13, 2021

Dr. Sudipta Saha
Supervisor
Department of Computer Sc. and Engg.
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar

ACKNOWLEDGEMENT

I would like to express my deep and heartfelt gratitude for my respected guide and mentor, Dr. Sudipta Saha , for his evergreen blessings, valuable advice, support, good wishes and encouragement all alone the journey of this work. Without his undisputed support, this work would not have been true in the light of daylight. I am also grateful to the DSSRG members for assisting me in implementation works and for their valuable comments.

Place: Indian Institute of Technology Bhubaneswar
Date: May 13, 2021

Himanshu Goyal

Abstract

In this current age of the Internet of Things, digital devices form complex interconnections to monitor surroundings. The objects of interest are monitored using sensors, and distributed wireless sensor networks are constructed from numerous sensor nodes. The decentralization of devices poses a significant risk of trust if any critical task is carried out using these devices. The presence of byzantine faulty sensor nodes can potentially degrade the safety property of the system. Several fault tolerance consensus mechanisms in wireless sensor networks (WSNs) were proposed, with a few faulty nodes in a sensor network, most healthy nodes can reach a consensus. Most of them target crash faults as their primary goal as these are more prone to happen but fail to consider the real world scenario of Byzantine Failures. Applying Byzantine Agreement (BA) methods in Wireless Sensor Networks (WSNs) degrades their life dramatically due to significant communication overhead. Most traditional proposed BAs assume that the processors are connected to unlimited energy resources, and the ideal links exist between them. A rough estimation about the required number of exchanged messages for decision in a network with N nodes is $O(N^2)$. Therefore, inefficient energy consumption becomes more challenging as the network grows. In this work, we focus on both reducing the communication cost and overall agreement time to achieve better scalable in distributed sensor network. We achieve so using flooding-based efficient wireless data transmission protocols and guarantees both safety and liveness in a network of $3f + 1$ nodes with at most f failures.

Keywords: Wireless Byzantine Consensus, Network Synchronisation, Fault tolerance, One-to-many, Many-to-Many, etc.

List of Figures

Abbreviations

BA	Byzantine Agreement
WSNs	Wireless Sensor Networks
f	Number of failure nodes in a network of N nodes
CFT	Crash Fault tolerance
PBFT	Practical Byzantine Fault tolerance
O2A	One-to-all
A2A	All-to-all

Contents

CHAPTER 1

Introduction

1.1 Introduction

Byzantine Generals' problem

The concept of Byzantine Fault Tolerance is derived from the Byzantine Generals' problem which was explained in 1982, by Leslie Lamport, Robert Shostak and Marshall Please in a paper[1] at Microsoft Research.

Imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching an agreement. The generals must decide on when to attack the city, but they need a strong majority of their army to attack at the same time. The generals must have an algorithm to guarantee that (a) all loyal generals decide upon the same plan of action, and (b) a small number of traitors cannot cause the loyal generals to adopt a bad plan. The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition (a) regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree upon a reasonable plan.

The process of reaching an agreement through opinion sharing of involved parties in a distributed system is commonly referred to as *Consensus*. At the end of the Consensus protocol participants agree on a single value i.e., a common decision from the set of available initial values. Consensus is challenging to achieve in the presence of failures (node crashes, message losses, network partitions, etc.). It is even proven that Consensus is impossible in a fully asynchronous setting, where at least one node might never be able to communicate.

Consensus protocols differ widely in the types of failure they tolerate. One is *non-Byzantine* (or fail-stop, or crash) failure, in which faulty nodes simply stop responding (and later they will recover). Another is *Byzantine*, in which there are no restrictions to what faulty nodes can do. The former is popular in a distributed settings with permissioned environment(like a data center), whereas the later is similar to the open, decentralized settings like P2P networks. Almost all distributed databases adopt this fail-stop model as they owned and controlled by a single organisation. Byzantine failure models are often made in the context of security, particularly when nodes are compromised, or participants in the network are malicious.

Consensus protocols are characterized by two properties:

- **Safety:** all non-faulty nodes agree on the same value.
- **Liveness:** the system makes progress, i.e. it does not get stuck in any round of agreement.

1.2 Existing Work

Several solutions to the consensus problem of handling non-byzantine failures have been proposed in the literature [2, 3, 4, 5]. Paxos[3] is one of the first protocols to provide consensus, which further extended to Raft[5]. It is Crash fault (non-Byzantine) fault-tolerant and proven to be correct in handling f failures in the presence of 2f+1 failures. They are widely used in many permissioned deployments, for instance, Google's globally distributed File System [6]and Hyperledger Fabric[7]. It has widely deployed for the distributed environment where traditional desktop/server machines are present. Wireless-Paxos[8] is the

first work to explore the agreement protocol in the domain of resource constraint wireless sensor systems. They focus on Crash faults through optimized PAXOS logic, but they fail to address Byzantine failures. The consensus protocol, Proof-of-work PoW used in Bitcoin [9], and recently Hyperledger Fabric [7] seems to handle Byzantine failures. The idea is straightforward: The network selects randomly at each round a node that proposes a value that the rest of the network adopts. Notably, the probability of a node being selected is proportional to its ability to solve computation puzzles. The system guarantees liveness, i.e., there will be solutions to the puzzles but no safety. PoW[10] is computation bound, and its bottleneck is the overall hash power. However, most IoT devices are battery-powered and limited in bandwidth resources, communication, and computational capabilities. Several other work works [11, 12, 13, 14]tries to provide Blockchain-based services for IoT Devices. They consider both Blockchain and Wireless sensor networks as two non-overlapping networks. Initially, the devices submit their transaction information to the local network by passing it via others. Then these transaction details are forwarded to the on-cloud Blockchain network like Hyperledger Fabric[7] or Ethereum, which runs heavy computation machines for the successful agreement. The separation of IoT devices with an on-cloud blockchain network induces the delay as consensus results come from the cloud. Thus, having a byzantine fault-tolerant agreement in the devices itself can heavily reduce the latency and improve performance.

1.3 Consensus requirement in Wireless Sensor Networks

With the evolution of intelligent sensor technologies, the problem of the network's trustworthiness has increased immensely. These advancements open the door to have both malicious attacks and software errors frequently. Building robust network services that can tolerate a wide range of failure types is a fundamental problem in distributed systems. The most fundamental approach, called Byzantine fault tolerance, helps mask arbitrary failures exhibited by failing nodes. Here, reliably broadcasting messages in a multi-hop network is dealt with, where some nodes may behave maliciously and are likely to fail. Nowadays, many applications in low-power wireless networks need complex coordination

between their peer nodes to carry out the defined specific task. Several examples range from Smart Grid automation to industrial coordination, where consensus in a short period is the foremost need for reliable completion. One typical example from the domain of the Mission-critical system is the swarm of Unmanned Aerial Vehicles (UAVs). All the UAVs must agree on a common destination for better performance, while failing to agree on a common decision can lead to substantial consequences. Thus, it can be realised that agreement is necessary for building reliable decentralised application.

CHAPTER 2

Background and Challenges

2.1 Fault Tolerance

The most crucial problem that distributed consensus algorithms need to solve is ensuring *safety* and *liveness* so that returned consensus results are trustworthy under unknown risks and uncertainties. Naturally, it is assumed that *crash faults* are relatively easy to solve compared to byzantine faults. Algorithms like Paxos[3] and RAFT[5] are typically used to handle crash or non-byzantine faults and thus are commonly known as crash fault tolerance (CFT) algorithms or non-Byzantine fault tolerance algorithms. But if we consider natural world unseen environments, then *Byzantine faults* are more prone to happen. Thus, their occurrence may cause unauthorized changes, have higher complexity, and are more challenging to handle. Algorithms for solving these problems are called Byzantine fault tolerance algorithms. What is the boundary between the two types of fault tolerance algorithms? In what scenarios do these two types of faults occur? These are some important points which always need to be look into while designing both these types of algorithms.

Fisher, Lynch, and Paterson (FLP) Impossibility:-
FLP [15] impossibility means that, in an asynchronous network, no distributed consensus protocol can meet all three Safety, Liveness, and Fault tolrenace properties simultaneously.

In a distributed system, node failures are almost inevitable. Therefore, fault tolerance must be taken into consideration. FLP impossibility means that any consensus protocol can only have either liveness or safety in addition to fault tolerance. In practice, we can often make some sacrifices. For example, we can sacrifice a certain degree of security, which means that the system can reach an agreement quickly, but the agreement is not reliable. We can also offer a certain degree of liveness, which means that the system can go a very reliable agreement, but this process takes too long, or an agreement can never be reached due to relentless debate. Fortunately, many practical scenarios show strong robustness, making it very unlikely for events to invalidate a consensus protocol.

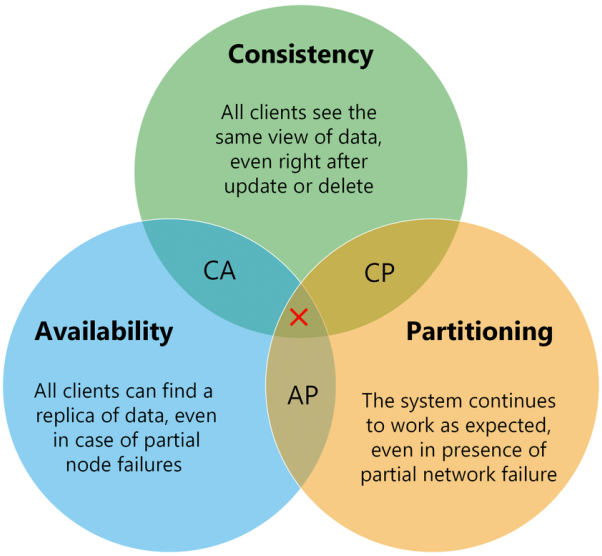

Figure 2.1: *CAP theorem diagram*

2.2 Quorum size

A *quorum* is the minimum number of members of a deliberative assembly (a body that uses parliamentary procedure, such as a legislature) necessary to conduct the business of that group. Given a set of node U, a quorum system is a set of subsets of U *(or quorums)* in which any two quorums intersect. More formally, Q is a quorum system defined as $Q \subseteq P(U) \wedge \forall q1, q2 \in Q[q1 \cap q2 \neq \phi]$. Let N, Q, f be the number of replicas, quorum size and the maximum number of failures. While it is rather easy to derive Q when $N = 2f + 1$ (for non-Byzantine, or $N = 3f + 1$ for Byzantine failures), the following details highlights

more general property of Q, especially with larger N. It shows that going beyond the minimum N for tolerating f may be worse for performance.

Non-Byzantine quorum size

- To ensure liveness, one must be able to make progress (quorum to decide) without waiting for f failed nodes. That is, $Q \leq N - f \leftrightarrow Q \geq \frac{N}{2} + 1$
- To ensure safety, two quorums must intersect. That is, $2Q > N$

It follows that $N < 2Q < 2(N - f)$. It means $f < \frac{N}{2}$, or $N > 2f$. It means the minimum size of N is $2f + 1$, and in this case $Q > \frac{N}{2}$, or $Q \geq f + 1$.

Byzantine quorum size

- To ensure liveness, $Q(N - f)$
- To ensure safety, two quorums must intersect at at least $f + 1$. That is $2Q - f > N \leftrightarrow Q \geq \frac{N+f}{2} + 1$.

It follows that $N < (2Q - f) < 2(N - f) - f < 2N - 3f$. It means $f < \frac{N}{3}$, or $N \geq 3f + 1$. At the minimum size of N, i.e. $N = 3f + 1$, we have $Q \geq 2f + 1$.

2.3 PBFT: Basics

Over 20 years ago, the groundbreaking work on **Practical Byzantine Fault Tolerance (PBFT)** by Miguel Castro and Barbara Liskov [16] had finally demonstrated the practical feasibility of dealing with systematic byzantine (The term Byzantine was coined by Leslie Lamport [1], on the Byzantine Generals Problem, which simply represents a faulty machine that shows arbitrary behavior, thus invalidating or even attacking the protocol itself.) faults in a distributed system, even in face of a very strong adversary. PBFT is a byzantine fault resistant consensus method —is well suited for IoT as it offers low computational power and complexity. PBFT is a practical and improved protocol on BFT that was proposed in [16] and it achieves an order of magnitude improvement in response time over BFT by working in an asynchronous environment [17]. PBFT also achieves a reduction in the complexity level of messages from the exponential level associated with BFT to a polynomial level complexity [18]. PBFT offers significant reduction in energy consumption when compared with well-known Blockchain Consensus Mechanisms such as Proof of Work

(PoW) and Proof of Stake (PoS) [10]. It is also not affected by the centralization and the "nothing at stake" problems associated with PoS. The PBFT protocol provides safety and liveliness when no more than $\lfloor \frac{n-1}{3} \rfloor$ out of the total n replica nodes are faulty [6]. This upper bound on the number of faulty nodes implies that neither operator or software errors nor adversary alterations can cause crash or adverse effect on the system. All the nodes in a PBFT system are ordered sequentially with one node being the Primary node and others considered as backup nodes or replica nodes. All the nodes in a system communicate with each other and reach consensus based on the majority principle. Each PBFT consensus round is called a *view*. Primary node is changed during every view and can be replaced with a protocol called a view change if a certain amount of time has passed without the leading node broadcasting the request. This replica timeout mechanism ensures that the crashed or malicious primary can be detected and that a new view starts by re-electing a new primary node.

As shown in the figure Fig. 2.2, five phases are experienced from the client launching requests to receiving responses.

The five-phases can be understood as follows:

1. **Launch:** The client (client c) initiates the service request m to the cluster.

2. **Pre-prepare:** The leader node (replica 0) verifies the validity of the request message m, assigns the sequence number n to the request m in the view v and broadcasts the assigned pre-prepare message to all the backup nodes (replica 1-3).

3. **Prepare:** The backup nodes verify the validity of the request message m and accept the sequence number n. If a backup node accepts the assignment scheme, it broadcasts the corresponding prepare message to the other nodes. In this phase, all the replicas are required to reach a globally consistent order.

4. **Commit:** Once the assignment agreement message is received from the cluster, all the nodes (primary and secondary nodes) broadcast the commit message to all the other nodes. In this phase, all the replicas have agreed on the order and confirmed the received request.

5. **Execute and reply:** After receiving the commit message from the cluster, the nodes execute the request m and send replies to the client. The client awaits the same reply from $f + 1$ different nodes and considers that the request has been

executed successfully. f represents the maximum number of potential faulty nodes in the clusters. That all the nodes directly return messages to the client is also to prevent the primary node from having problems during the request.

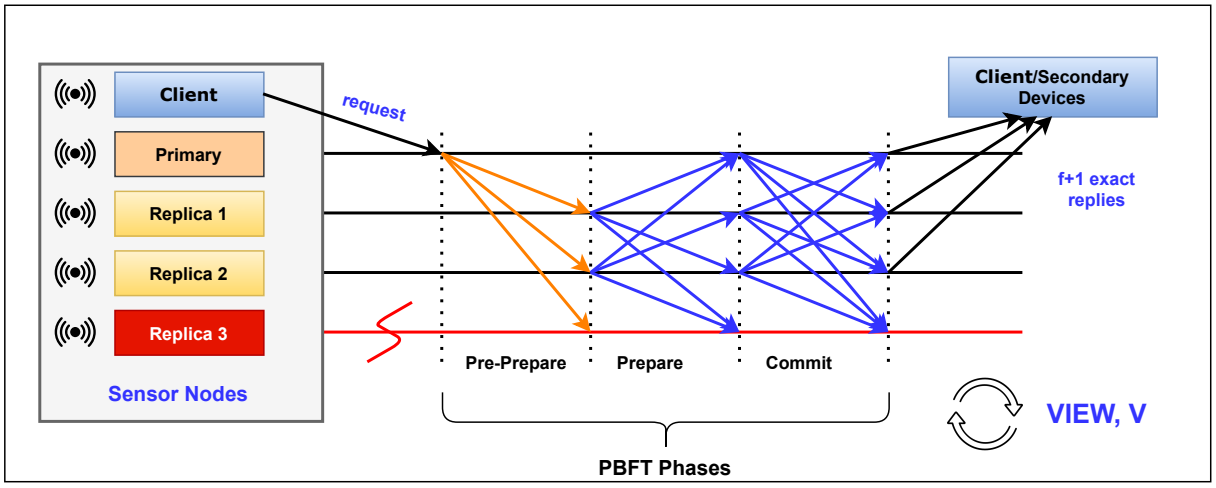

Figure 2.2: *Normal case operation of the Practical Byzantine Fault Tolerance (PBFT) network*

The conventional PBFT network is made up of n replica nodes and it can tolerate a maximum of f faulty replica nodes. The relationship between n and f is defined from [6] as follows

$$f \leq \left\lfloor \frac{n-1}{3} \right\rfloor$$

In other words, the algorithm provides *liveliness* and *safety* as long as not more than $\lfloor \frac{n-1}{3} \rfloor$ replica nodes are faulty. The network will move through various *views* as it progresses and for each view, it selects one of the nodes as the primary node. The nodes can take turns to be the primary node. The primary for the *view* v is denoted by v_p and can be obtained as follows

$$v_p = v(modn)$$

We represent the set of replicas in a view by S and identify each replica by utilizing an integer in $\{0, 1, ..., |S| - 1\}$. Note that there could be more than $3f + 1$ replica nodes, however, the additional replica degrades performance due to the increase in the amount of messaging without an improvement in the level of resilience.

The operational steps of the conventional PBFT, which is illustrated in Fig. 2.2, are as follows:-

1. The client sends a service request to the primary node of the PBFT network. The PBFT network goes through three phases namely, the pre-prepare, prepare and commit.

2. Leader broadcast PRE-PREPARE to followers.

3. Each follower broadcasts PREPARE to other nodes, and waits for $2f + 1$ matching PREPARE messages (including its own).

4. Each follower then broadcasts COMMIT to other nodes, and waits for $2f + 1$ matching COMMIT messages (including its own).

5. Each follower executes the operation, then sends REPLY to the client. The client waits for f+1 replies before returning.

Compared with Paxos and Raft, PBFT can process more problems: In addition to crash faults, it can process Byzantine problems that may cause troubles and unauthorized changes. However, from the perspective of the trade-off policy adopted in PBFT, PBFT is still similar to Paxos and Raft. From the perspective of FLP, PBFT also emphasizes the fault tolerance and safety and weakens the level of liveness.

2.4 Challenges

Crash Fault Tolerance algorithms such as Paxos and Raft can only process machine hardware errors and fails to handle situations where malicious nodes are present. These non-BFT algorithms can only work in highly reliable network environments, for example, an internal network infrastructure. In such closed networks, access requires strict authorization, ensuring that the ownership of specific sites is known and trusted. Such permission helps in eliminating malicious nodes and allows algorithms to work effectively. For example, in a 10-node private network, only ten authorized nodes can access the network. Although some nodes (nodes less than 3) work aggressively and try to change data without authorization, the whole system is trustworthy and secure. In such an approved network, the number of nodes that can malfunction is estimated. When other nodes do badly, their

real identity can be quickly discovered. This indirectly improves network security. But this identification of the malicious nodes is not straightforward as BFT algorithms do not have strict requirements for the network environment. Even if there are malicious nodes, the whole system is safe only if the malicious nodes are no more than 1/3 of the total nodes. However, this presents new challenges. How do you know the number of nodes in the network? What is the ratio of malicious notes to all available nodes in-network? The traditional non-byzantine consensus algorithms PAXOS and Raft uses only two rounds of execution, i.e., One round of data dissemination from Leader to backup nodes and in another round all backup node replies to leader node. Whereas in PBFT, there are three rounds, i.e., Pre-Prepare, Prepare, and, Commit. Pre-Prepare round is similar to PAXOS, i.e., Leader to backup nodes wherein other two rounds in PBFT every node forwards their opinion to all other nodes in the network. These two data broadcasts are quite costly in terms of communication cost compared to non-byzantine consensus algorithms. So, in a network of N nodes, $O(N^2)$number of messages are required for either Prepare and Commit phase. The presence of malicious sensor nodes in the distributed network is also considered in PBFT, which further introduces one more extra phase, VIEW-CHANGE. In the View-change phase, all unsatisfied backup nodes forward a new execution request to all the available nodes, including the next primary node. It eventually introduces extra overhead apart from the usual three phases. There is no discussion about this view-change phase in non-byzantine algorithms. Thus, the communication requirement becomes a real bottleneck when a network grows. This requirement also limits the scalability of the distributed network of tiny sensor nodes.

CHAPTER 3

Scalable Byzantine Consensus for WSNs

3.1 Wireless Communication Primitives

Low-power wireless networks are broadcast oriented networks where each transmission can be received by all neighboring nodes. Executing unicast-based schemes in wireless networks usually induces higher costs, especially in multi-hop networks. Moreover, multi-hop networks can help in knowing neighbouring data more efficiently. Due to the broadcast nature of wireless communications, concurrent transmissions of nearby nodes inherently interfere with each other. However, when such transmissions are precisely timed, one of them can be received, nonetheless. We briefly introduce the concept of capture effect[19] and one-to-many communication primitive i.e. *Glossy*[20], and then present our self developed *Minicast(All-to-All)*[21] communication primitive used in this work extensively.

Capture effect[19]:- Nodes overhearing concurrent transmissions of different data can receive the strongest signal under certain conditions; this is known as the capture effect[19]. To achieve capture at the receiver with IEEE 802.15.4 radios, the strongest signal must arrive no later than 160 s after the first signal and be 3 dBm stronger than the sum of all other signals.

One-to-all:- Approaches that enable a single sender to transmit packets to all other nodes in the network belong to this class; relevant examples include network flooding protocols that send single packets. These are also known as single data dissemination protocols. **Glossy[20]** provides network flooding and optionally time synchronization. The single sender in Glossy is called initiator. It starts the flooding process by transmitting the packet, while all other nodes in the network have their radio in receive mode. Provided propagation delays are negligible, the one-hop neighbors of the initiator receive the packet all at the same time. Glossy then makes these nodes retransmit the received packet so that the retransmission delay is constant and equal for all involved devices. This way, the one-hop neighbors of the initiator that received the packet retransmit it at nearly the same time. This enables the two-hop neighbors of the initiator to receive and retransmit the packet, and so on. Glossy floods are confined to reserved time slots, where no other tasks (application, operating system, etc.) are allowed to execute on a node. This avoids interference on shared resources, which would cause unpredictable jitter. Due to its operation, Glossy also enables nodes to time-synchronize with the initiator.

All-to-all:- Protocols in this class enable all nodes in a network to share data with each other, forming the basis for network-wide agreement or for computing a network-wide aggregate. Once after the completion of this type of protocols all the nodes will be able to receive data of all other nodes. Here, we use our self-developed protocol **Minicast**[21] which achieves reliable and efficient all-to-all data dissemination in just few microseconds. In Minicast[21], a designated node starts an all-to-all interaction by transmitting a chain of packets whose length is equal to number of nodes(N) in the network. Initially, this designated node puts its own data in the assigned slot in this chain. When a node receives a message, it combines its own data with the received data chain, then transmits the combined data synchronously with other nodes. To avoid unnecessary transmissions, nodes are only allowed to transmit if they have new information to share. Whether there is new information to share is determined using flags inside each message. The all-to-all protocols also provides the functionality of many-to-many data dissemination where several designated nodes are required to exchange their data among themselves.

3.2 Contribution

This work makes the following contributions:

- Achieving original version of Practical Byzantine Fault consensus[16] using one-to-all(*Glossy*)[20] and all-to-all(*Minicast*)[21] data dissemination for low-power wireless networks.
- We present *Scalable Byzantine Consensus*, a new flavour specifically designed to address the challenges of low-power wireless networks, through optimised use of data communication protocols.

3.3 Approach

The wireless PBFT network is made up of IoT devices or nodes. A client is an IoT device that makes a transaction or exchange information/record with other IoT devices referred to as replica nodes. When the header node in the network receives the client-IoT request, it starts the *three phases* of the PBFT consensus network namely, *pre-prepare*, *prepare* and *commit*[16]. In the pre-prepare, the primary node multicast a pre-prepare message to other nodes (generally referred to as replica nodes) in the PBFT network while nodes communicate with each other in the prepare and commit phases. The transaction are then recorded on the local storage once consensus is reached in the consensus network. The PRE-PREPARE phase, where the header node forwards the incoming client request to rest of the nodes can be easily mapped to one-to-all data dissemination round as shown in Fig. 3.1. Furthermore, the next two rounds i.e. Prepare and Commit involves every node to know the opinion of remaining nodes. These both also can be reduced to an execution of Minicast(all-to-all) data dissemination protocol independently as shown in Fig. 3.2. As shown in section 2.2, the minimum quorum size required to achieve byzantine consensus is atleast $2f+1$ identical responses. We consider this a potential opportunity for improvement to enhance scalability. Thus, instead of executing Minicast fully to achieve all-to-all data dissemination, we stop its execution after certain iterations with an aim that all nodes receive atleast $2f+1$ responses instead of waiting replies from all N nodes. This optimisation further reduces the communication cost and also reduces the overall time for achieving consensus.

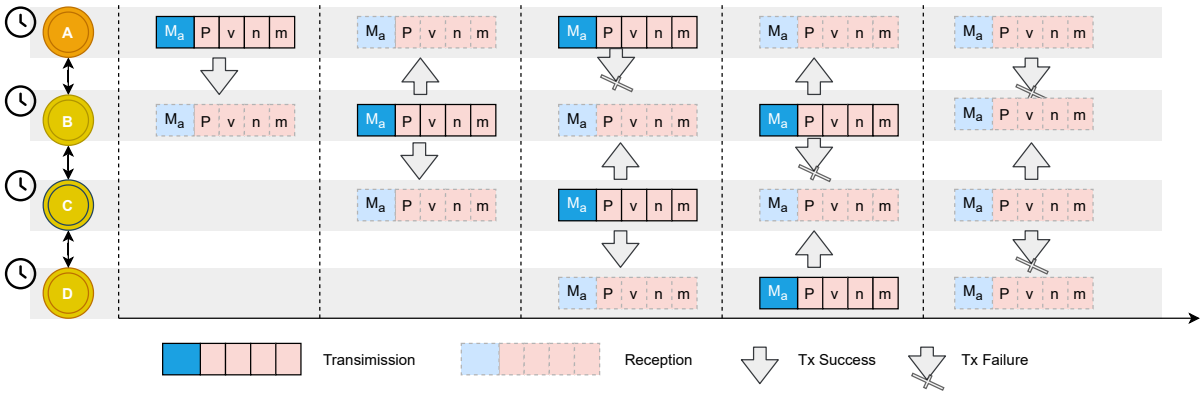

Figure 3.1: *Execution of PRE-PREPARE Phase using One-to-all Dissemination Glossy primitive. Here, in Slot 1 the primary/initiator node forwards the Pre-Prepare*$< v, n, m >$ *request to all neighbours. During the remaining slots every node either transmit or receive the data shared from either neighbours or primary node.*

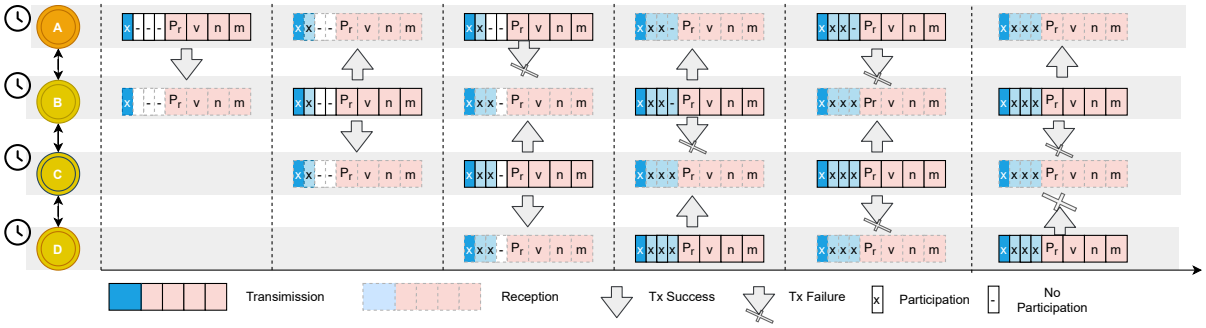

Figure 3.2: *Execution of COMMIT/PREPARE phase using All-to-All data dissemination protocol. Here, during each slot every node participates in either the data sharing or reception. There is a pre-defined slot in packet chain for every node which can be used to transmit their own data. To avoid unnecessary transmissions, nodes are only allowed to transmit if they have new information to share. After the completion of round every node will get to know about the data of all other nodes available in the network.*

CHAPTER 4

Experiments and Results

Simulation Environment:-

We setup our distributed Wireless sensor network(WSNs) environment using ns-3 network simulator. ns-3 easily supports loading of firmware files, and has some tools for monitoring stack, setting breakpoints, and profiling. We use ultra low power IEEE 802.15.4 compliant wireless sensor module based on a TI MSP430 and Chipcon CC2420 radio. It has 10k SRAM, 48k Flash and 1024KB Serial storage. The firmware for these sensors nodes was developed locally and further uploaded to each node. We coded the Byzantine Fault tolerance(BFT) logic described in Fig. 2.2 using above described communication primitives in C-Language. The behaviour of these protocols can be understood from Fig. 3.1 and Fig. 3.2. The following results are obtained with multiple experiments by varying both the number of sensor nodes and their respective overall coverage area.

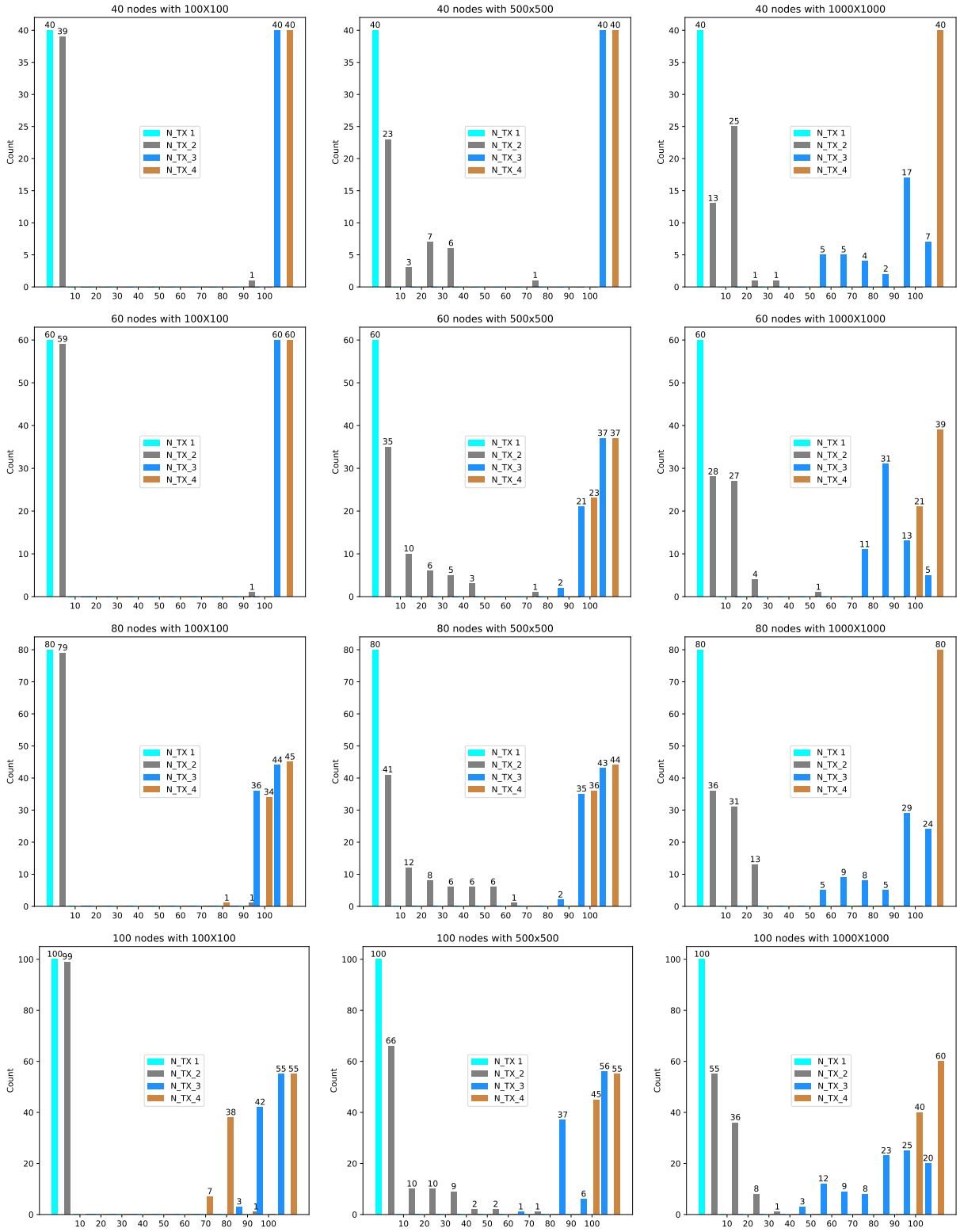

Figure 4.1: *Execution of Minicast(all-to-all) data dissemination protocol by varying N_TX(a parameter which controls the number of required packet transmission/reception count) and overall area in m^2. X-axis shows the overall packet percentage while Y-axis measures the count of the nodes falling into respective packet percentage. E.g. In top left figure of* 40 *nodes spreads across an area of* 100×100 m^2*, the blue bar shows all the* 40 *nodes have received data from all remaining nodes with N_TX=3.*

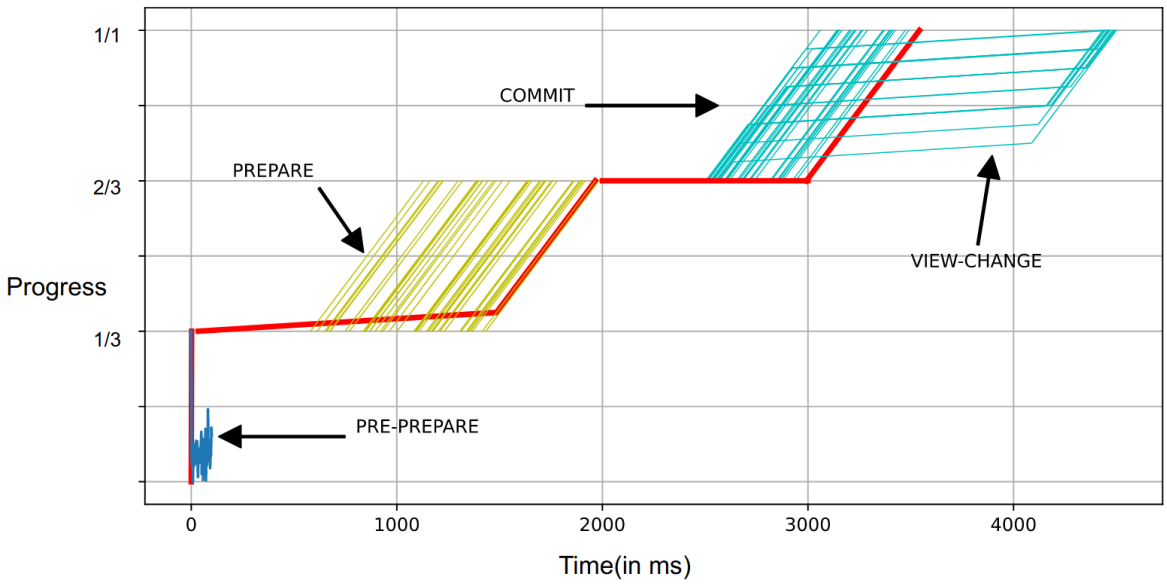

Figure 4.2: *Overall progress of* 50 *nodes for achieving byzantine fault tolerant consensus on a single value. The PRE-PREPARE uses one-to-all whereas remaining both PREPARE and COMMIT phase uses Minicast(all-to-all) as data communication primitives.*

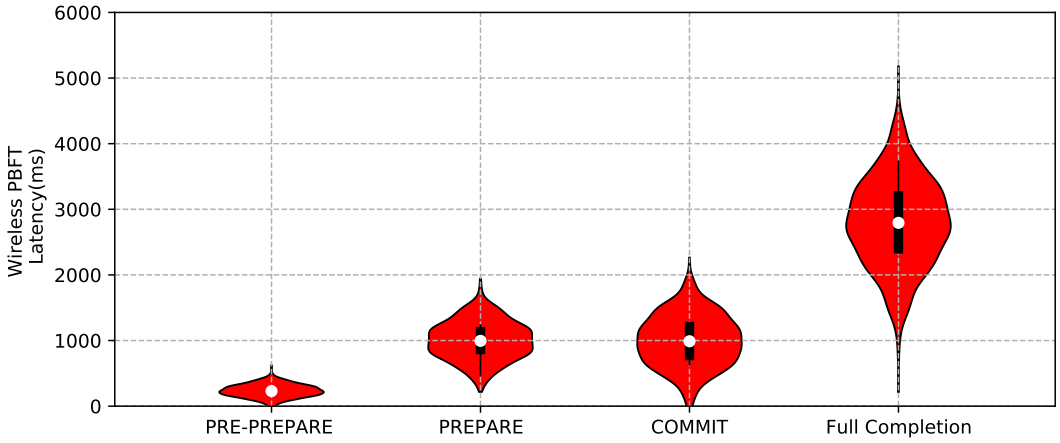

Figure 4.3: *Average time duration of each consensus phase and overall completion time(in milliseconds)*

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, p. 382–401, July 1982.

[2] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, p. 133–169, May 1998.

[3] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[4] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, October 2006.

[5] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (USA), p. 305–320, USENIX Association, 2014.

[6] J. C. Corbett, J. Dean, and E. et al., "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, Aug. 2013.

[7] E. Androulaki, A. Barger, and B. et al., "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[8] V. Poirot, B. Al Nahas, and O. Landsiedel, "Paxos made wireless: Consensus in the air," in *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, EWSN '19, (USA), p. 1–12, Junction Publishing, 2019.

[9] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.

[10] "Blockchain challenges and opportunities: A survey," *Int. J. Web Grid Serv.*, vol. 14, p. 352–375, Jan. 2018.

[11] O. Novo, "Scalable access management in iot using blockchain: A performance evaluation," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4694–4701, 2019.

[12] E. Schiller, E. Esati, S. R. Niya, and B. Stiller, "Blockchain on msp430 with ieee 802.15.4," in *2020 IEEE 45th Conference on Local Computer Networks (LCN)*, pp. 345–348, 2020.

[13] C. Profentzas, M. Almgren, and O. Landsiedel, "Iotlogblock: Recording off-line transactions of low-power iot devices using a blockchain," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, pp. 414–421, 2019.

[14] C. Profentzas, M. Almgren, and O. Landsiedel, "Tinyevm: Off-chain smart contracts on low-power iot devices," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 507–518, 2020.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, p. 374–382, Apr. 1985.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999* (M. I. Seltzer and P. J. Leach, eds.), pp. 173–186, USENIX Association, 1999.

[17] N. Chondros, K. Kokordelis, and M. Roussopoulos, "On the practicality of 'practical' byzantine fault tolerance," *CoRR*, vol. abs/1110.4854, 2011.

[18] L. Zhang and Q. Li, "Research on consensus efficiency based on practical byzantine fault tolerance," in *2018 10th International Conference on Modelling, Identification and Control (ICMIC)*, pp. 1–6, 2018.

[19] K. Leentvaar and J. Flint, "The capture effect in fm receivers," *IEEE Transactions on Communications*, vol. 24, no. 5, pp. 531–539, 1976.

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

[21] S. Saha, O. Landsiedel, and M. C. Chan, "Efficient many-to-many data sharing using synchronous transmission and tdma," in *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pp. 19–26, 2017.