Imperial College London

MEng Individual Project

Department of Computing

Imperial College London

Reliable Distributed Consensus for Low-Power Multi-Hop Networks

Author:
Alberto Spina

Supervisors:
Prof. Julie McCann
Dr. Michael Breza

Second Marker:
Dr. Anandha Gopalan

17th June, 2019

Abstract

Wireless Sensor Networks are notorious for their unreliable links. Core distributed computing tasks, such as consensus, are unobtainable with the lack of message delivery guarantees. Synchronous Transmission primitives have been used to make links more reliable, but they all have problems. Glossy is robust, yet suffers from high many-to-many dissemination latency. Chaos has a reduced primitive execution time, but sacrifices strong termination guarantees, and is unlikely to complete under strong network interference.

We propose Hybrid, a new dissemination strategy, to address the problems arising from the use of individual ST primitives. Hybrid interleaves Chaos and Glossy floods to minimise network latency and maximise reliability. Additionally we present WISP and WIMP, Paxos implementations which leverage the guarantees of Hybrid to provide correct and dependable consensus solutions for Wireless Sensor Networks.

This report describes the design of Hybrid, WISP and WIMP, and demonstrates their reliability and resiliency with testbed evaluation under strong interference. The thesis concludes by illustrating how the protocols match, and better the latencies of state-of-the-art implementations, while providing a more robust solution to network-wide voting and consensus.

Acknowledgements

I would like to thank:

- My supervisors, Prof. Julie McCann and Dr. Michael Breza for the incredible amount of help, support and guidance they have provided me with throughout the project.
- My second marker Dr. Anandha Gopalan, for being an exceptional personal tutor, always available to offer assistance during my time at Imperial.
- The Computer Engineering Group at ETH Zürich, especially Romain Jacob, for the guidance and advice provided when learning about Baloo.
- My family, for the unfaltering love and support they have given me during these four university years.
- All my friends, with whom I have shared memorable moments throughout my degree.

Contents

1 Introduction

1.1 Motivation

Network-wide agreement is a key component of a distributed system as it allows participants to share common "objectives", data or knowledge throughout the whole network. Due to their unreliability and high communication costs, consensus has always been considered too expensive of a protocol for wireless sensor networks (WSNs). There exist, though, a number of critical systems which would greatly benefit from the dissemination of trusted values, which a majority of network is guaranteed to agree upon. Distributed configuration management, leader election and node clustering are examples of protocols which require specific network-wide agreement guarantees, which low-power multi-hop networks are unable to provide.

The problem with WSNs is they are notorious for their unreliability. Moreover it has been proven for consensus protocols to not work in a fully asynchronous system in the presence of even a single node failure [16], further impacting their lack of adoption. The solution is to use more robust network dissemination primitives which provide stronger guarantees on correct node reception probabilities.

A new family of robust protocols, exploiting constructive interference and the capture effect to ensure reliability, was introduced in 2011 with Glossy [15]. Utilising back-to-back fast floods, Glossy guarantees network-wide dissemination of packets with above 99.99% probability. To mitigate the high latency impacts incurred when using Glossy's one-to-many synchronous transmissions (ST) to disseminate values from multiple nodes, a new ST primitive, Chaos, introduces efficient all-to-all data sharing by allowing nodes to concurrently aggregate payloads into the flooded packets.

Built on top of Chaos, multi-phase voting protocols and a wireless Paxos [32] implementation were developed between 2017 and 2019 using A^2 Synchotron [2], a synchronous transmission kernel that enables full configuration over Chaos rounds. The lack of precise timing constraints over the Chaos floods, though, hinders the A^2 implementation, which provides low-latency protocol implementations sacrificing fairness and potential system liveliness in the presence of failure or interference.

To pioneer a new generation of more energy aware applications, Baloo [21], a flexible and configurable middleware for ST primitives was published in 2019. Allowing the execution of both Glossy and Chaos-based protocols Baloo enforces a strict time sliced paradigm where protocol run times must be declared in advance. Even though this provides a solution to A^2's unreliable execution lengths, by effectively cutting protocols off after a specific time bound, it potentially hinders the safety properties of the protocols themselves, which might not have been able to terminate correctly.

Therefore, despite the fact that new reliable dissemination primitives have been introduced to low-power multi-hop networks, there currently is no robust, failure tolerant implementation of consensus protocols. Even though experimentations, such as A^2's WPaxos exist, they fail

in providing both robust, scalable and reliable consensus in WSNs.

1.2 Contributions

The project's objective is to make wireless communications more reliable and robust. With these extra reliability guarantees we implement consensus protocols, which are normally too sensitive to unreliable communications. This project presents the following main contributions:

- **Hybrid synchronous transmissions primitive and the XPC library**. We introduce Hybrid, a new ST primitive which minimises network latency and maximises reliability. Hybrid leverages the optimal latency of Chaos floods and optimises them with Glossy's reliable one-to-many communications. To configure and switch multiple ST primitives independently of protocol implementations we introduce XPC. XPC is a novel multi-phase voting library which allows users to easily recreate existing voting protocols within the Baloo middleware matching state-of-the-art latencies and enhancing overall reliability. Hybrid and XPC are discussed in Chapter 3.
- **WISP and the Wireless Part-time Parliament**. We propose WISP, a Paxos implementation for XPC, based on the Hybrid ST primitive. To provide support for majority-voting consensus protocols we present WiPP, the first Wireless Part-time Parliament, an extension of the XPC library tailored for quorum-based voting. WISP uses the Wireless Part-time Parliament to match, and better, the latency and reliability of state-of-the-art protocols, while providing the same guarantees and consensus properties. WISP and WiPP are discussed in Chapter 4.
- **WIMP and Multiple network proposers**. We present WIMP, a reliable Multi-Paxos implementation for XPC, based on the Hybrid ST primitive. The XPC library is extended to allow for multiple proposers. All nodes within the network are allowed to propose values during the execution of any XPC protocol with any ST primitive. WIMP uses multiple proposal to execute multiple back-to-back Paxos rounds for all pending network values. WIMP and multiple network proposers are discussed in Chapter 5.

2 | Background

This chapter aims to introduce the fundamentals of Wireless Sensor Networks and consensus which are the focus for the whole project.

2.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN), which is the system assumption of this thesis, is a network of distributed autonomous devices with sensing and actuating capabilities. Each individual unit is often called a WSN mote. Motes sense the environment and communicate with each other to collect data at central locations.

2.1.1 Motes

A WSN mote (of which the TelosB chip in Figure 2.1 is an example) is made up of the following components:

- A **Microcontroller** (MCU), which is the center of all processing for the WSN mote.
- Multiple **Sensors/Actuators**. Sensors capture an aspect of the state of the environment (i.e. temperature, humidity, light) and actuators enable the mote to perform an action (i.e. LEDs for light, buzzers for sound, motors for movement)
- One **Wireless Communicator or Radio**, a IEEE 802.15.4 [19] compatible transceiver, used by motes to communicate with each other on the network by sending and receiving communications.
- **Battery**. Motes have limited battery life (unless they are able to generate power through other means), and hence are equipped with long-lasting batteries to supply them with power. No battery is present on the TelosB in Figure 2.1 as the are usually externally soldered onto the chip once users decide the target lifetime they wish to provide for their motes.

Motes will switch the radio on and off depending on when they expect to receive communication, to reduce energy consumption. This is called radio duty-cycling.

2.1.2 Contiki-NG

Contiki-NG [31] is an operating system developed for IoT devices with limited resources that facilitates the programming of WSN motes. Available for a multitude of MCU architectures and radio devices, Contiki-NG provides network, sensing and actuation programming capabilities for motes. The internal repository structure is representative of the various components that OS is made out of:

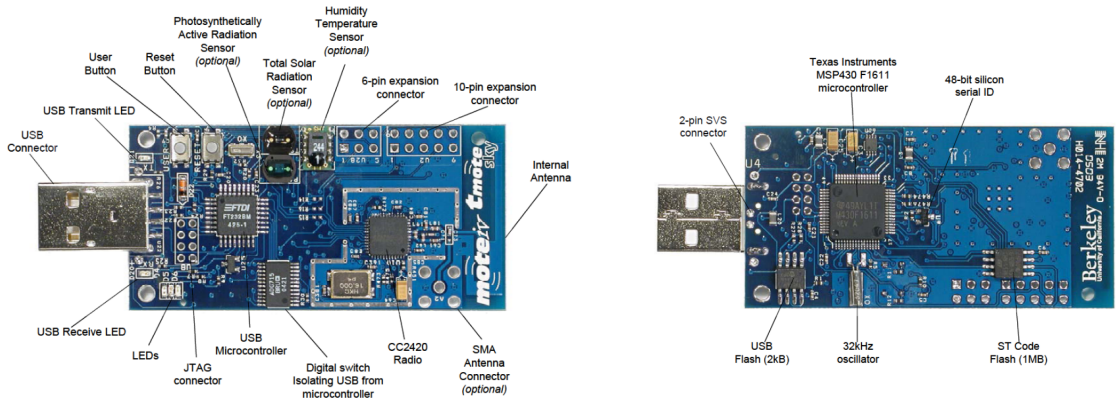


Figure 2.1: Front (left) and back (right) view of the TelosB wireless mote [8].

- **arch**: Contains all the MCU information and has implementations for the various devices present on supported chips.
- **os**: The main Contiki-NG kernel. Provides libraries for sensors/actuators, STL-like functionality (i.e. lists, queues), timers, asynchronous I/O and networking.
- **tools**: A collection of tools for flashing, debugging and simulating Contiki-NG. The main simulation software is Cooja, written in Java.
- **tests**: Adds test programs and unit-testing capabilities to Contiki-NG.
- **examples**: Example programs which can be used as a baseline for the various kernel functionalities.

Contiki-NG supports a wide variety of IoT device architectures, most specifically there are a number of motes which are most common in use, therefore the most supported both in terms of software and hardware:

- TelosB / Tmote Sky: they have an MSP430 microcontroller (one of MSP430x15x, MSP430x16x or MSP430x161x) and a CC2420 transceiver.
- MicaZ: an ATmega128L MCU with a CC2420 radio chip.
- CC2650 SensorTag 2.0: equipped with the CC2650 wireless MCU.

There are numerous other operating systems for IoT devices, such as TinyOS and numerous Contiki versions (2.x, 3.x and NG), and they all have similar event-driven programming models and capabilities [7]. As discussed in Section 2.8 this project builds on top of the Baloo middleware [21], which was written for the Contiki-NG operating system. The OS choice is therefore non-negotiable. Contiki-NG is the most recent and most documented iteration of the Contiki kernel. Code contributions can potentially be pushed to the official GitHub repository and be used by researchers around the globe, making this the most mature IoT-device OS currently in use.

2.1.3 Networking

WSN motes communicate with each other over radio using the IEEE 802.15.4 technical standard (defined for low-rate wireless personal area networks) [19]. The nodes are organized

in topologies based on the spatial distribution of the motes that compose them. Networking in Contiki-NG is slightly different from the traditional OSI-layer model, as the current implementation, called `NETSTACK` is built by four distinct elements:

- Network Layer (`NETSTACK_NETWORK`). Running on top of IEEE 802.15.4 with Time-Slotted Channel Hopping [42] (TSCH), Contiki-NG has a full IPv6 stack. Implemented via uIP [12] (an open-source TCP/IP implementation for 8 or 16-bit microcontrollers), IP, UDP and TCP protocols are available in minimized models. Routing is handled via RPL [41], an IPv6 routing protocol for low-power and lossy networks, which maintains a routing graph built from a root node for the whole network.
- MAC layer (`NETSTACK_MAC`). Designed to prevent packet collisions, Contiki-NG uses Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) [22] to determine if a transmission is occurring at the same time as a node wants to send over the shared medium, backing off when it determines it would cause a packet collision.
- RDC layer (`NETSTACK_RDC`). The Radio Duty Cycling saves energy by powering down the radio transceiver for most of the time according to a periodic schedule (i.e. during a given time period on for 20% and off for 80%). An alternative approach is low-power listening (LPL), supported by Contiki-NG, which allows the RDC to power back on the radio when it senses communication is occurring.
- Radio layer (`NETSTACK_RADIO`). The lowest `NETSTACK` layer handles the specific radio module present on the physical mote.

The maximum communication distance between two motes is limited. Large WSNs may cover an area which is larger than the radio communication range. These are called multi-hop networks, characterised by large network topologies with hundreds of motes, where network packets will go through a number of devices before reaching their final destination. In these multi-hop networks individual motes need to take into account that they must keep their radios powered on for longer periods of time to relay packets broadcast by other nodes. Contiki-NG supports both single-hop and multi-hop networks.

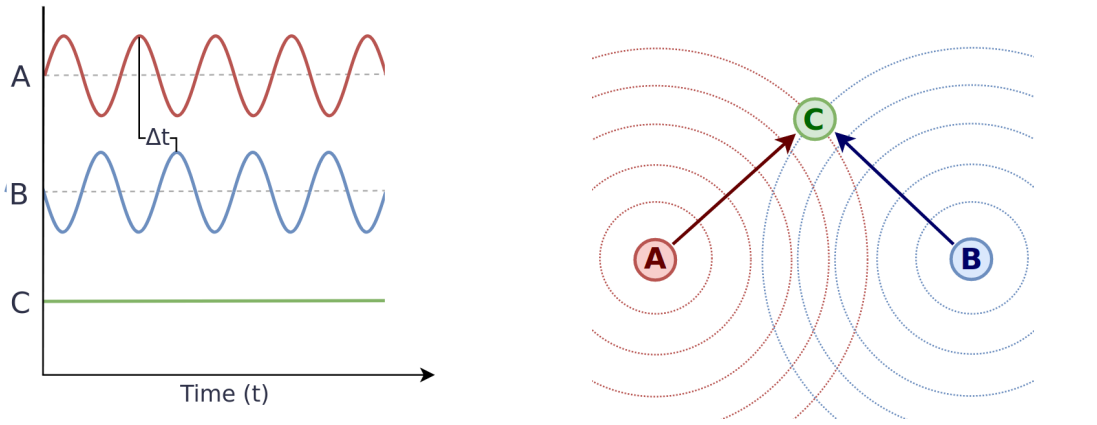
The unreliability of wireless communications is, though, a constant concern. Conflicting broadcasts and lack of constant radio up-time are cause of high packet losses and missed transmissions. A potential solution to unreliable WSN communication has been proposed by using the capture effect and interference.

2.1.4 Capture Effect and Interference

WSNs communicate using radio transceivers. Radio communication is inherently broadcast based; the broadcast of one node will superimpose itself upon the broadcasts of other nodes in close spatial proximity. This phenomenon is known as the superposition principle [43]. It occurs to all electromagnetic waves and is most notable when the frequencies of the colliding waves match. Formally, in the presence of two waves w_a and w_b of the same frequency, generated concurrently from points A and B respectively, which are received at point C, the phase difference $\Delta\phi$ between the two waves is denoted as:

$$\Delta\phi = \frac{2\pi f \Delta d}{c}$$

where $c = 3 \times 10^8\ m/s$ is the speed of light, f is the wave frequency and Δd is the spatial distance between the broadcaster and the receiver. The phase difference is perceived as a time offset Δt between the alignment of the periods of the two waves (see Figure 2.2a).



(a) Superposition of 2 waves at receiver C (b) Dissemination of concurrent broadcasts to C

Figure 2.2: Visualisation of multiple concurrent broadcast superposition at receiver C.

The receiver, though, will only perceive the summation of all input waves together, and for non-zero phase difference values the received signal strengths vary. This phenomenon is called interference and occurs when two broadcasts share the same frequency and generate signals which physically overlap in time and space (see Figure 2.2b). For periodic waves of period 2π there are two main types of interference:

- **Destructive**, which occurs when $\Delta\phi$ e is an odd integer multiple of π. Under destructive interference (see Figure 2.2a) the receiver is unable to decode incoming transmissions as the addition of the individual wave signal strengths is zero.
- **Constructive**, which occurs when $\Delta\phi$ is an integer multiple of 2π. Constructive interference allows receivers to correctly detect the superimposed signals emitted by multiple transmitters. The received signal has greater strength if compared to the individual broadcasts from A or B.

With constructive interference node broadcasts are received more reliably and with greater signal strengths, yet in order to achieve it the time offset Δt between the waves reaching receiver nodes must be close to zero. This is a very difficult problem in practice, which has only recently been approached within WSNs. There are currently two main methods to maximise wave allignment in order to guarantee constructive interference in Wireless Sensor Networks:

- Using **clock synchronization**. By achieving accurate clock synchronization, and being able to predict with certainty all software delays incurred when preparing, transmitting and receiving packets, it is possible to synchronise all the nodes in a network so that the broadcasted waves are aligned and achieve constructive interference on receivers.
- Using the **capture effect**. The capture effect occurs when a wireless module correctly detects a transmission from one transmitter despite there being interference on the channel [45]. This might occur if the original signal is stronger than all others (power capture) or when the broadcast started significantly earlier (delay capture).

Both of these effects are not commonly used in other forms of digital radio communication, and are still considered active research in the WSN community. In 2011 a new protocol, Glossy

[15], has introduced the first dissemination strategy to exploit both the capture effect and clock synchronization, starting a new branch of scientific research (see Section 2.3). Communication robustness, though, comes at a cost: Glossy uses a serial approach which greatly increases protocol latencies. A more recent strategy taken by Chaos [25] (2013) boosts dissemination times but is once again unreliable in the case individual nodes become temporarily unreachable.

Numerous other protocols have been developed to exploit constructive interference. Splash [10], Pando [11], Crystal [20] and Mixer [18] are such examples, each adding a contribution in terms of latency, reliability, or communication efficiency. Throughout this thesis we will focus on analysis Glossy (Section 2.3) and Chaos (Section 2.5), as they have led the most analysed, cited and robust branch of research.

2.2 Consensus

A fundamental primitive of all distributed systems (among which WSNs) is consensus. The problem of consensus consists of reaching network-wide agreement on proposed values in the presence of potentially faulty processes [9]. All (or a majority) of nodes share an identical, agreed-upon, value in a system which has reached consensus. This strong guarantee is applied to protocols which involve leader election, atomic broadcasts and distributed configuration management.

In our model we consider a collection of processes P_i (where $i = 1..N$) communicating via message passing. Communication between processes is assumed to be reliable, though processes themselves can fail. A failed process is said to have crashed; a non-failed process is said to be correct. In order to reach consensus, each process P_i proposes a single value v_i. Processes communicate with each other and then choose one of the proposed variables v_i to set as their decision variable d_i. At this point the process has entered a decided state and is no longer able to change d_i (see Figure 2.3).

Many consensus algorithms exist. For all of them the following properties hold [34]:

- **Termination**. Eventually each correct process decides a value d_i.
- **Agreement**. All correct processes decide on the same value d_i.
- **Integrity**. A process decides at most on one value.
- **Validity**. If a process decides on a value d_i, then d_i has been proposed by some process.

The termination property can also be reformulated as liveliness. Liveliness expresses the eventual termination of a protocol and is independent on the correctness of the decided value. The union of agreement, integrity and validity properties guarantee a safe protocol execution. A protocol's safety property expresses which guarantees are given on the decided value upon termination. Safety properties are ranked strong to weak based on the strictness of their condition (i.e. how many nodes in the network are guaranteed to share the same decided value).

2.2.1 Asynchronous Systems

It is proven that no algorithm can guarantee to reach consensus in an asynchronous system with as little as one process failure; this is known as the impossibility result [16]. This is because in an asynchronous system processes may respond at arbitrary times and a "slow-to-reply" process is indistinguishable from a crashed one. Wireless Sensor Network are examples of distributed asynchronous systems: individual node clocks may be out of sync, messages

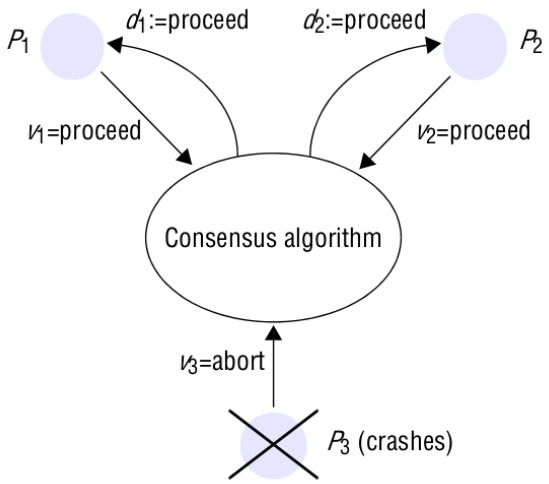


Figure 2.3: Three processes engaged in a consensus algorithm. Processes P_1 and P_2 propose value "proceed". Process P_3 proposed "abort" but then crashes. The two remaining correct processes decide on value "proceed" [9].

can be delayed for arbitrary amount of times and there is no guarantee that a broadcast message will be correctly received by any number of nodes. Yet consensus can be achieved within WSNs, as there are a number of techniques which can be used to work around the impossibility result:

- **Masking faults**. During its execution, each process saves data to persistent storage (which is able to survive failures). Upon a crash, a process is simply restarted and is able to resume its tasks by reading back the data to memory. This means that crashed processes are able to behave similarly to correct ones; occasionally they just take a long time to reply.
- **Failure detectors**. Failure detectors allow all processes taking part in the consensus algorithm to know about a trusted sub-set of processes which are considered to be correct. All processes are initially treated as correct, but during a protocol's execution if replies to periodic heart-beat messages are either missing or incorrect the node will be suspected to be faulty. As failure detectors suspect failed processes, all other processes will discard their messages. If less than $N/2$ processes crash, consensus can still be reached by the sub-set of correct processes.

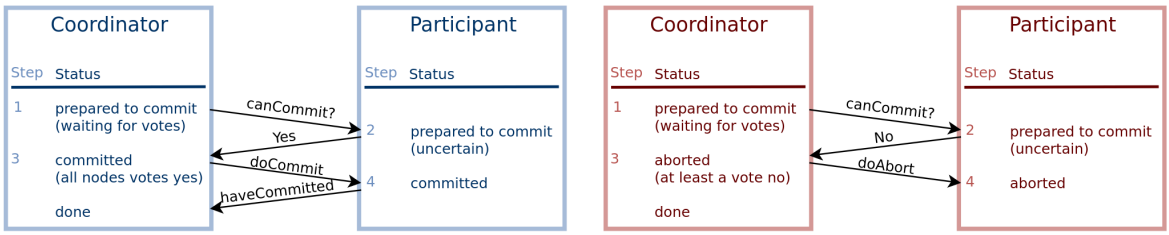
2.2.2 Atomic Commit Protocols

Of the many properties of distributed transactions, we are interested in atomicity. The atomicity property states that when a distributed transaction comes to an end, either all of its actions are carried out, or none of them. Transactions are ended when the client decides to commit or abort them. There are two main atomic commit protocols, which are listed below. Both of them require each transaction to have a designed coordinator, which is aware of all of the nodes participating within the transaction itself.

2.2.2.1 Two-phase commit

The simplest atomic commit protocol is two-phase commit (2PC) [17]. 2PC is a blocking protocol which guarantees that, should all nodes in a network vote to commit a value d_i, upon termination all nodes will have committed the value d_i (safety property). In an asynchronous system, though, the protocol is not guaranteed to terminate (liveliness property) as it may block indefinitely waiting for replies from a few missing nodes. Two-phase commit is initiated by a client asking the coordinator to commit a transaction. There are 5 operations which are allowed by the protocol:

- `bool canCommit(transaction)`. The coordinator asks the participant if it is able to commit the specified transaction. Receiver replies with its vote (*Yes* or *No*).
- `void doCommit(transaction)`. The coordinator tells the participant to commit the given transaction.
- `void doAbort(transaction)`. The coordinator tells the participant to abort the transaction.
- `void haveCommitted(transaction, participant)`. The participant confirms to the coordinator it has committed the given transaction.
- `bool getDecision(transaction)`. The participant asks the coordinator confirmation about a transaction it has voted *Yes* for, but the coordinator did not reply to. Used to recover from coordinator crashes or processing delays.



(a) Global Commit. (b) Global Abort.

Figure 2.4: Coordinator and Participant communication in two-phase commit protocol.

The protocol is then carried out in 2 phases (4 steps):

- **Phase 1** (voting phase):
  1. Coordinator sends a `canCommit` request to each participant.
  2. Upon receiving a `canCommit` request, participants vote (either *Yes* or *No*). Before replying with a *Yes*, participants store all transaction information on permanent storage. If the vote is *No*, the participant aborts the transaction immediately.
- **Phase 2** (completion phase):
  3. The coordinator collects all the votes.
     (a) If all participants voted *Yes*, then a `doCommit` request is sent to each participant.
     (b) With at least one *No* vote, the coordinator sends a `doAbort` request to all participants which voted *Yes*.

4. Upon receiving a `doAbort` request, participants abort the given transaction. If they receive a `doCommit` request they reply to the coordinator with a `haveCommitted` message.

In the best-case scenario, N transactions can be committed in N protocol phases (see Figure 2.4). In this description we assume a fail-restart failure model, where nodes may fail and later resume communications. Due to its weak liveliness property it can take arbitrary amounts of time for the transaction to take place, as no maximal time-limit can be specified for the protocol's execution. Multiple, successive, failures are tolerated via masking faults (§2.2.1) allowing nodes to reboot and resume communication rounds. The problem with two-phase commit is that due to its strong safety guarantees it may block indefinitely, prohibiting applications using it from making progress. This issue is fixed with three-phase commit.

2.2.2.2 Three-phase commit

Two-phase commit is a blocking protocol. Node failures will impact all other participants as the whole network will be waiting the coordinator to make progress. To overcome this, in addition to the lack of termination guarantees, three-phase commit (3PC) [39] is introduced (see Figure 2.5). 3PC provides guaranteed termination (liveliness property) by allowing the coordinator to timeout and abort a transaction. It, though, relaxes the safety property as it is possible for cohort nodes to commit values that were actually aborted due to a timeout by the coordinator. The main decision is to ensure protocol progress at the expense of safety. This is achieved by introducing an additional pre-commit phase between the phases of 2PC:

- **Phase 1** (proposal voting phase): same as 2PC.
- **Phase 2** (pre-commit or abort phase): as in 2PC the coordinator decides whether to commit or abort transactions (participants which timeout or fail to reply are assumed to have aborted the transaction). If a transaction is to be committed a `preCommit` request is sent to each participant. Participants reply with `ACK` messages.
- **Phase 3** (do-commit phase): The coordinator now sends actual `doCommit` requests and the participants commit the transaction. This ensures that no participant has proceeded to commit a transaction if any of the nodes were in an uncertain state.

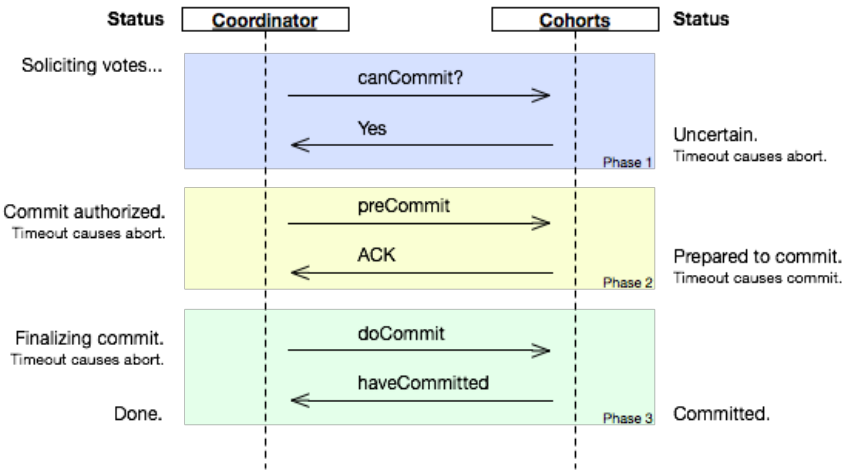


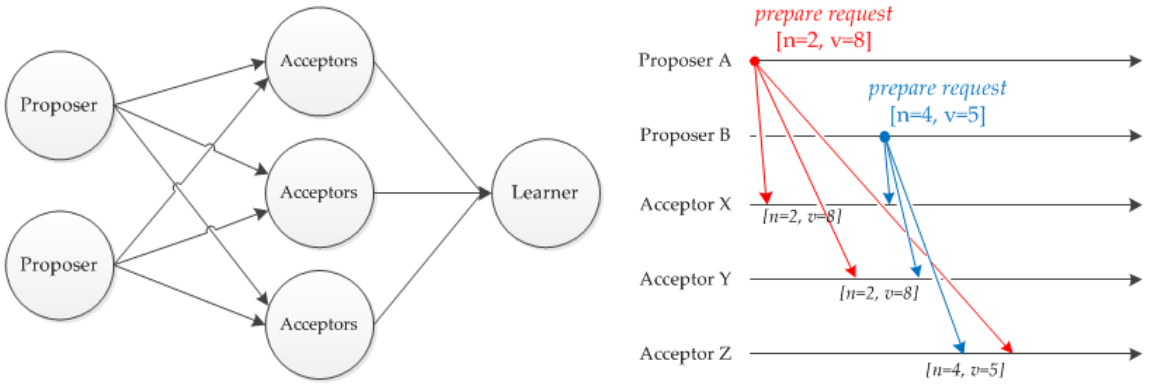
Figure 2.5: Three-phase commit protocol communication [38].

The protocol is non-blocking, individual node failures do not impact the liveliness of the whole network. It removes the possibility for a participant from committing and terminating before the other nodes. This resolves the ambiguity present in 2PC with the `doCommit` message. Coordinator failures can therefore easily be recovered from, without waiting for the existing coordinator to come back online, as a new participant can take over the coordination role and reach a new agreement with the network. Cohort node failures (or delays) will accordingly he handled with timeout commits or aborts as seen in Figure 2.5.

Three-phase commit is unable to recover from a network-partition. The protocol aborts the moment a single node temporarily goes offline or crashes and, should the coordinator have become unreachable for the cohort, a new one would be selected. This is the difference with 2PC's blocking approach, which would wait for the node to come back online. As there would be two live coordinators after a network partition (i.e. the original one and a newly selected one by the partitioned network) there is no way to reconcile the transactions of the two network subsets. Furthermore the protocol requires a minimum of three round trips to complete, potentially introducing high latency between transactions. To address the shortfalls and limitations of three-phase commit our discussion continues to Paxos, a majority-based consensus algorithm.

2.2.3 Paxos

Paxos [24] is a consensus protocol proposed by Leslie Lamport in 1989 which has become the industry standard for consensus. Paxos solves the problems of safety and liveliness present in 2PC and 3PC by providing efficient majority-based network-wide voting. Paxos divides nodes into three different roles: proposers, acceptors and learners (see Figure 2.6a). Proposers independently propose values they want agreement on. Acceptors receive communication from proposers and independently choose values. When a majority of acceptors agree on a selected value, the information is forwarded to learners, which store the list of decided values. Paxos solves the liveliness problem of 2PC by only needing a quorum of replies in order to proceed to the next phase. It solves the safety problem of 3PC by only querying a majority of acceptors, regardless of who is the leader. Each Paxos proposal has the form `[n, v]`, where `n` is the proposal number and `v` is the proposed value.



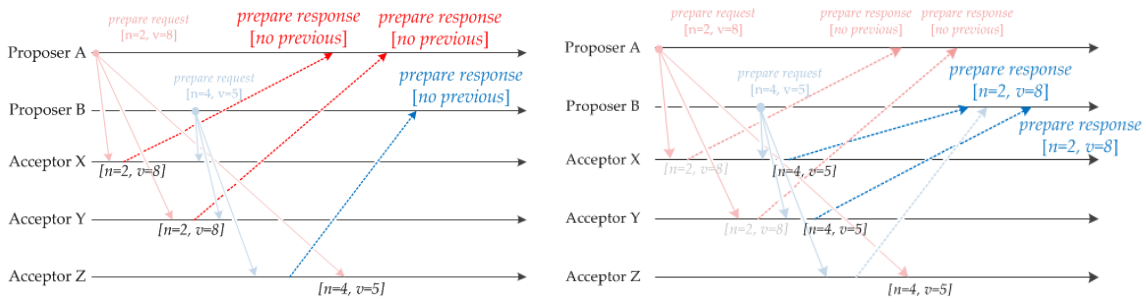
(a) Paxos Overview

(b) Paxos during the prepare phase

Figure 2.6: Paxos diagram and example node communication [28].

Paxos is structured into two rounds:

- **Prepare-Promise**.
  - Each proposer chooses a proposal number `n` and a value `v`. It asynchronously sends the information to the acceptors (Figure 2.6b) in a *prepare request*.
  - Acceptors receive `[n, v]` pairs from proposers. If `n` is greater than their currently stored proposal number, or if they have never seen a proposal number before, they update their stored `[n, v]` pair and reply to acceptors with a *prepare response*. If the proposal number `n` sent by the proposer is lower than the currently stored value it does not send a reply (i.e. acceptor Z does not reply to proposer A in Figure 2.7b).
- **Accept-Accepted**.
  - When a proposer receives a *prepare response* from a majority of acceptors it then issues an *accept request* to the acceptors.
  - Upon receiving an *accept request* if the proposal number `n` is still the highest proposal number seen by the node, the acceptor sends an *accept response* to each learner. A value `v` is declared chosen when learners have received *accept responses* from a majority of of acceptors.



(a) Acceptors replying to proposers (b) Acceptors updating their proposal numbers

Figure 2.7: Complete Paxos prepare phase between two proposers and three acceptors [28].

A number of optimisations have been suggested over a vanilla Paxos implementation. Ideally proposers could agree to vote for leaders. A leader would be able to greatly speed up the algorithm by reducing the proposal number clashes that would require proposers to back off and try proposing again.

There currently exist many implementations of Paxos that make, or depend, upon different system assumptions. In this thesis we present a new variation of Paxos based on the lax guarantees provided by our communication approach. WSNs communications are unreliable, introducing problems for the correct functionality of Paxos. We thus continue our analysis presenting the background of our communication approach, aiming to provide high reliability guarantees to allow for the proposal of a new consensus algorithm.

2.3 Glossy

Proposed to solve WSN unreliable links, Glossy [15] is a novel flooding architecture for Wireless Sensor Networks. It exploits constructive interference of IEEE 802.15.4 radio chips to achieve

fast network flooding and time synchronization. This approach is novel to WSNs, and is still an area of active research. Glossy is able to flood packets within a few milliseconds and achieve a time synchronization error below one microsecond. Glossy's performance is not impacted by node density, making it a good candidate for real-world applications. Glossy's approach to time-synchronization for concurrent broadcasts assign the protocol to the family of Synchronous Transmission (ST) primitives. ST primitives realize energy and time efficient network-wide broadcasts by synchronously transmitting packets from multiple wireless nodes.

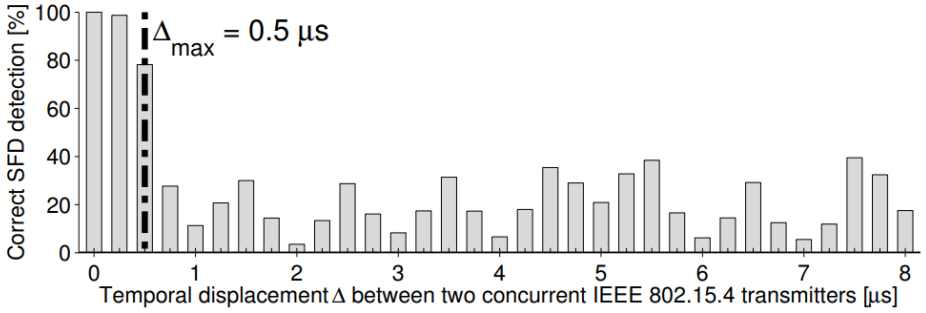


Figure 2.8: High probability of constructive interference of signals for $\Delta_{max} = 0.5\ \mu s$ [15].

2.3.1 Protocol Overview

Glossy uses constructive interference (§2.1.4) to speed up system-wide floods and allow for time synchronization. Given IEEE 802.15.4 signal modulation schemes (which allow digital signals to be encoded in an analogue wave, with redundancy, so that it can be sent over the wireless medium), Matlab simulations have been performed to evaluate maximum-allowable temporal displacement (Δ_{max}) which would still achieve constructive interference. As seen in Figure 2.8, results [15] show that even in the absence of capture-effects, correct detection occurs for $\Delta_{max} = 0.5\mu s$.

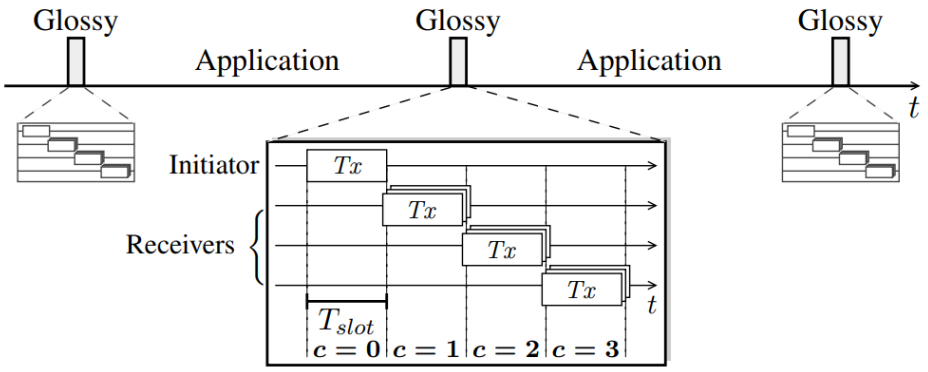


Figure 2.9: Propagation of a Glossy flood [15].

During a Glossy broadcast (Figure 2.9), there are two types of nodes:

- **Initiators**, which are considered as source nodes for each broadcast.
- **Receivers**, which listen for transmissions and re-broadcast them to allow for network propagation.

The entire flooding process is driven by radio events. When a node completes a full packet reception, a re-transmission is triggered so that all of it's neighbours are able to receive the given packet. Receptions and transmissions occur multiple times, propagating down multi-hop topologies.

The Glossy execution model can be expressed as a state-machine with 4 main states nodes can be in:

- **Wait**. Nodes have the radio turned on and are waiting for packets. Upon sensing a transmission, we enter the *receive* state.
- **Receive**. The node keeps receiving the transmission for its full length. If the transmission is incomplete or corrupted, it goes back to the wait state. Otherwise it reaches the *transmit* stage.
- **Transmit**. The MCU immediately re-transmits the incoming message. The relay counter is increased by one. During a glossy flood, nodes can be asked to transmit packets up to N times (to improve overall reliability). If a node has already reached N transmissions it goes to the *off* state. Otherwise it goes back to *wait*.
- **Off**. Upon completion of a glossy flood, the node turns off its radio (to save power).

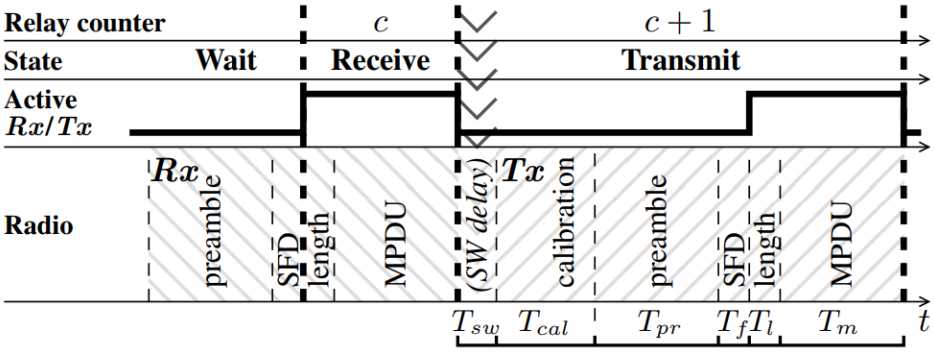


Figure 2.10: Glossy states and timeline model [15].

2.3.2 Time synchronization

As shown in Figure 2.10, all Glossy state transitions depend on the radio hardware (except a brief software delay T_{sw} introduced before the transmission stage). To achieve constructive interference Glossy needs a maximum temporal displacement Δ_{max} of 0.5 μs (Figure 2.8). We take all time delays into account to be able to estimate T_{slot}, the overall slot length of a Glossy transmission. If we denote T_{tx}, the packet transmission time, as:

$$T_{tx} = T_{cal} + T_{pr} + T_f + T_l + T_m$$

We can break down the individual time delays:

- T_{cal} is internally taken by the radio to calibrate its voltage controlled oscillator (VCO).
- T_{pr} is the time taken to generate the preamble of the packet.
- T_f is the time to align the start of frame delimiter (SFD) as specified by the IEEE 802.15.4 standard.

- T_m is the time taken to generate the MAC protocol data unit (i.e. the packet itself carrying the data).

The overall slot time can therefore be approximated as:

$$T_{slot} = T_d + T_{sw} + T_{tx}$$

Where T_d is the radio processing delay introduced by packet reception and T_{sw} is introduced by the MCU having to handle the software interrupt that would cause the transmission. This delay is non deterministic as it is MCU-dependant (i.e. an MSP430 might take between 1 and 6 cycles). To compensate for this Glossy measures the time taken and introduces a number of no operations if the MCU is too fast.

This estimates a finite T_{slot} for a Glossy broadcast, and allows Glossy to be a time synchronization protocol. During a Glossy flood, nodes can measure drift accumulated within T_{slot} transmissions. Based on this calculation they can compensate their clock oscillations to account for the measured drift.

Although fast and reliable, a Glossy-only approach can be slow for all-to-all communication rounds. As transmissions of different packets cannot be interleaved the protocol operates sequentially, new data floods occurs only after the current flood's termination. Furthermore, with the lack of a centralised control node, network participants might accidentally commence concurrent Glossy rounds with conflicting packets. In order to provide structure and organisation for Glossy, making it a viable solution for data gathering and dissemination, our discussion continues with the Low-Power Wireless Bus.

2.4 Low-Power Wireless Bus

Low-Power Wireless Bus (LWB) [14] provides a structured control system for data dissemination with Glossy. Its aim is to turn low-power multi-hop networks into a shared bus, where all nodes can continuously exchange data flows. LWB supports one-to-many, many-to-one and many-to-many traffic; it is topology independent and supports node mobility. It is built on three main ideas:

- Fast **network floods**. LWB is built on top of Glossy (§2.3) and exclusively uses fast network-wide floods for communication.
- **Time-triggered** bus **access**. All nodes using LWB are synchronized (by Glossy) and access the shared bus via a global communication schedule which is recomputed based on traffic demands.
- There is a **central host node**. The host node is in charge of computing the schedule and distributing it to all nodes. The host is able to adjust the schedule to the demands of the network adapting as nodes require to communicate more, or less, data.

2.4.1 Protocol Overview

LWB's protocol operation can be seen in Figure 2.11. It can be mainly split across three different stages. In this example it is assumed that all source nodes (i.e. nodes generating data packets which are sent through the bus back to the host) have an inter-packet-interval (IPI) of $6s$. This means that every 6 seconds nodes generate data which they wish to disseminate to the whole network through LWB.

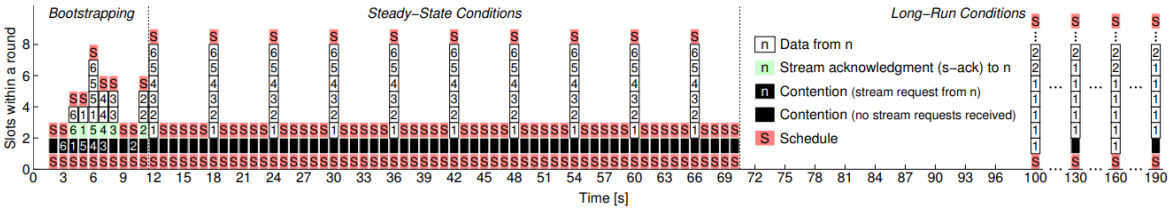


Figure 2.11: LWB protocol in operation through all three states [14].

- **Bootstrapping** ($t = 0 - 11s$). When the protocol is initiated all nodes listen with their radio on. Upon receiving the first schedule, the nodes synchronize with the host and learn about the round period T. The nodes start duty-cycling (§2.1.1) their radio to reduce power consumption. Every node communicates its traffic demands (i.e. its IPI) within the contention slot S which is allocated for every schedule. This introduces loss of communication during the first few rounds (at time $t = 4 - 10s$) as, of all the nodes replying in the contention slot S, the host is only able to detect at most one node's reply due to capture effects (§2.1.4). As the host learns about nodes wishing to join the bus, it acknowledges them by allocating slots on the schedule. By $t = 12s$ the first complete schedule is sent out.
- **Steady-state condition** ($t = 12 - 70s$). Starting from $t = 12s$ all nodes are time-synchronized and the host knows about the node's IPIs. In our case all nodes are aware that the round period T is $1s$ and that they will be receiving a new schedule at every round. Every 6 rounds the host will send a schedule which allocates a data-slot for each of the 6 nodes, which will reply in the allocated slot before the round period is over. Empty schedules at $t = 13 - 17s$ are necessary to allow the host to schedule further transmissions should the IPI of a given node have increased.
- **Long-run condition** ($t = 71 - \infty s$). LWB adapts to the streaming requests on the bus. By time $t = 70s$ the host has learned that the IPI of the nodes is constant. This allows for further optimisation: the period time T is increased from $1s$ to $30s$ and nodes are notified in the schedule. 5 slots are now allocated for each node in the schedule sent at $t = 100s$ to still meet the IPI demands. This allows nodes to have a much longer radio-off period when duty-cycling, effectively saving a lot of energy.

2.4.2 Failure Tolerance

LWB is resilient to failures. Depending on the type of failure it has specific guarantees or recovery mechanisms:

- **Node failures**. If a node fails or disconnects from the network, the host will cease receiving messages from it and eventually adapt the schedule to omit the node's specific data-slot. If the node resumes communication it will use the contention slot to communicate its needs to the host and will be re-added to the schedule.
- **Communication failures**. Even though all communication is handled by Glossy floods, communication failures may happen. Nodes are instructed to not communicate to the host if they missed the schedule for the specific round. They will instead communicate their IPI requirements upon reception of the next schedule in the contention slot.
- **Host failures**. If within a specified time period nodes to not receive communication from the host, it is assumed to have failed. Upon initialisation LWB hard-codes specific

channel-host pairs on all nodes. As the nodes realise the host as failed they will change radio channels and wait for the new host to bootstrap the network.

LWB therefore offers a robust many-to-many Glossy-based dissemination strategy. Using a serial communication style nodes are scheduled to sequentially disseminate their values to the whole network. Even though this approach ensures high reliability, is it heavily inefficient in terms of number of broadcasts and latency. The protocol's execution time grows linearly with the network size, making the implementation of voting protocols too expensive for large topologies.

2.5 Chaos

A building block for Wireless Sensor is the ability to have many-to-many interactions allowing packets to easily be disseminated to all nodes in the network. LWB (§2.4) and other protocols can potentially handle this, but they are inefficient. Such protocols would perform many-to-one data collection, centralised processing, and subsequent one-to-many dissemination. Chaos [25] offers lightweight many-to-many communication, thus parallelising collection, aggregation and dissemination steps using two main mechanisms:

- **Synchronous transmissions**. All nodes synchronously broadcast their data, and receptions are guaranteed due to capture effects (§2.1.4). Nodes merge locally the data and keep re-broadcasting it until all nodes have received data from all other nodes.
- **User-defined merge operators**. Data merging is fully customizable by the user allowing distributed processing of packets from all nodes.

Chaos is a primitive built on top of Glossy (§2.3) to leverage synchronous transmissions. The transformation from a one-to-many to a many-to-many packet sharing protocol, though, introduces two main challenges:

- **Packet merge overhead**. As mentioned previously in §2.3.2, Glossy requires minimal delays to be able to exploit it's fast flooding window. The maximum software delay T_{sw} considered between packet reception and re-transmission is of a few MCU clock cycles. Chaos will instead be performing full merge operations during T_{sw} with the duration of tens of thousands of MCU clock cycles. It could also mean different nodes have different merge execution times (impacting overall synchronization).
- **Constructive interference**. Unlike Glossy packets, Chaos packets are likely different for every node broadcasting them. This means there cannot be constructive interference on receivers and instead we can only rely on the capture effects. Higher number of conflicting packet transmissions severely reduce Packet Reception Rate (PRR) and if packet transmissions exceed $160\mu s$ the capture effect will no longer benefit receivers in decoding messages with high probability.

2.5.1 Protocol Overview

Chaos packets have two main components: a flags section and a payload. The flags are a bitmap containing one bit for every node present on the network. The payload section is data which is exchanged by every node during a Chaos iteration. The Chaos protocol can be seen in action in Figure 2.12, and can be mainly summarised in three sections:

- **Bootstrapping**. Each node has a payload to share with the network. Chaos starts operating when an appointed node, called the initiator sends its prepared packet (flags + payload) to everyone in the network. In our example A broadcasts to neighbours B-F in Slot 1.
- **Aggregation**. Upon receiving data, nodes aggregate the received payload with the payload they have stored using a merge operator (which can be user-defined). Newly received flags are `OR` merged with the existing ones.
  - If the newly merged **flags differ** from the currently stored ones, the node re-broadcasts (i.e. B broadcasts in Slot 2 after receiving data from A).
  - If the newly merged **flags do not differ**, or the reception failed, the node suppresses its transmission and waits. This can be seen by B in Slot 4 after receiving duplicate flags in Slot 3 by A.
- **Termination**. The protocol terminates when all nodes have a full set of all flags and hence no longer transmit (Slot 11). Nodes then enter a brief final-flood stage, where they aggressively disseminates the aggregated packet to the whole network.

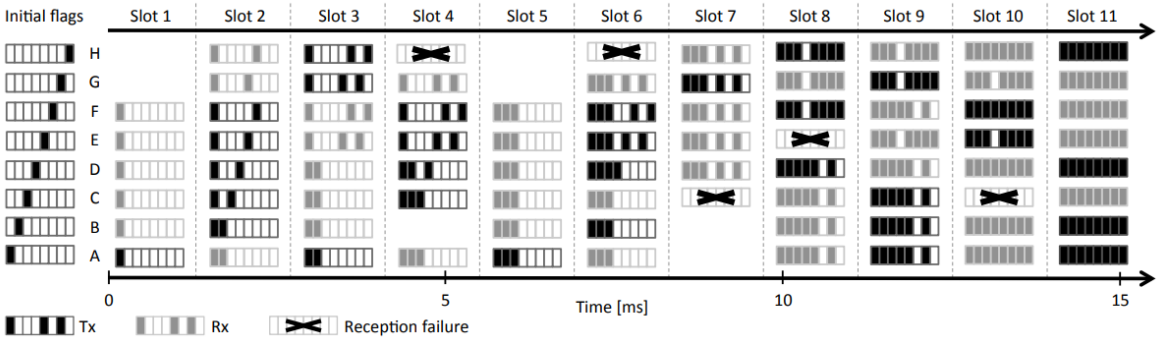


Figure 2.12: Trace of Chaos operating with 8 nodes on a multi-hop network [25].

Aggregation is therefore a key aspect of Chaos. Several simply built-in aggregators are provided (such as *min*/*max* which would allow the nodes to all agree on what is the maximum payload proposed by the network). More advanced aggregation such as *sum* or *average* is possible, though it suffers from the over-counting problem (i.e. accidentally aggregating a node's result more than once).

Varying MCU processing speeds (during aggregation) can also be prevented by setting a minimum processing delay for all nodes. Prior to entering the T_{tx} transmission stage, nodes will busy wait a user-defined number of MCU clock-cycles to make sure not to desynchronize.

Further care is taken to prevent early termination of the protocol (i.e. due to transmission failures which would lock the network in an uncomplete state where nodes have an incomplete flags bitmap). Firstly, to reduce concurrent (and conflicting) transmissions nodes do not broadcast if they received no new information. Secondly if premature termination is suspected (as would happen if during Slot 7 G's broadcast is lost having no nodes receive it), nodes have a timeout mechanism where they will attempt to reinitialise communication until completion.

2.5.2 Practical Applications

The ability to aggregate and disseminate data on the fly, allows Chaos to be applicable to a number of consensus (§2.2) problems:

- **Network-wide agreement**. Any important piece of information necessary to the network can be distributed using Chaos. Encryption keys can be exchanged by all nodes on startup by aggregating them into an array. Aggregators which keep selecting the maximal seen values can be used to agree on which nodes have the highest ID, and therefore should be nominated as leader, etc.
- **Three-phase commit**. Even though this will be greatly expanded by the "A^2: Agreement in the Air" protocols (§2.6), three-phase commit (§2.2.2.2) can be achieved by using the flags section as a counter for the three stages of the protocol, and data is exchanged via the payload. Node failures or missed messages will not advance the flags, prompting for re-transmissions.

Chaos drastically reduces the communication cost and latency for implementations of voting protocols, yet it sacrifices reliability for efficiency. Chaos floods are unable to target specific nodes or request replies form specific network participants. Messages are simply constantly aggregated into the flood's payload and eventually replies from all nodes will reach the initiator mote. The inability to reliably schedule specific participants to reply affects the protocol's correctness in the presence of failure or channel interference. Nodes cannot quickly be suspected of failure, as their replies might simply not have been propagated yet by the floods. Chaos is therefore a potentially unreliable protocol to use for robust failure-tolerant consensus algorithms.

2.6 Agreement in the Air

A^2, Agreement in the Air [2], brings distributed consensus to low-power multi-hop networks. A^2 introduces Synchrotron, a new synchronous transmission kernel, and extending on concepts proposed by Chaos (§2.5) and LWB (§2.4), introduces consensus and network-wide voting primitives.

2.6.1 Synchrotron

Synchrotron is a new synchronous transmission kernel, introduced as a robust lower layer for A^2. Inspired by LWB and Chaos, it implements a number of functionalities:

- **Time-slotted design**. Borrowing from Chaos, Synchrotron operates in time slots which include the time taken for reception, processing and transmission of packets.
- **Synchronization**. Node synchronization is achieved via a virtual high-definition timer [36] (VHT), which uses a combination of low and high-frequency clocks to maintain synchronization in absence of network activity.
- **Frequency Agility**. Within Chaos reception rates degrade quickly in presence of interference. Synchrotron fixes this by borrowing from algorithms such as time-slotted channel hopping (TSCH) [42]: transmissions are spread across multiple channels. Each node chooses one channel to use for transmissions within a given time slot, overall this increases capture effects (§2.1.4) on receivers, and overall the protocol's reliability.
- **Scheduler**. Multiple A^2 applications can coexist and be scheduled independently during the lifetime of a node.

Synchrotron has been used to implement all A^2 protocols described in this section.

2.6.2 Protocol Overview

A^2 extends Chaos introducing a number of network-wide communication primitives:

- **Disseminate, collect and aggregate**. Inherited from Chaos, they allow for many-to-one and one-to-many communication.
- **Vote**. A network-wide voting primitive is introduced allowing nodes to vote against a coordinator's proposal (see §2.6.3).
- **Agreement**. Network-wide agreement protocols such as 2PC and 3PC is introduced (see §2.6.4).
- **Group membership**. A^2 allows for persistent group membership with join and leave capabilities.

2.6.3 Network-wide Voting

A fundamental building block for network agreement (§2.6.4) is a network-wide voting primitive. It allows to distribute a proposed value and collect votes from all participating nodes. A successful voting round ends with the coordinator knowing which votes were cast by which nodes.

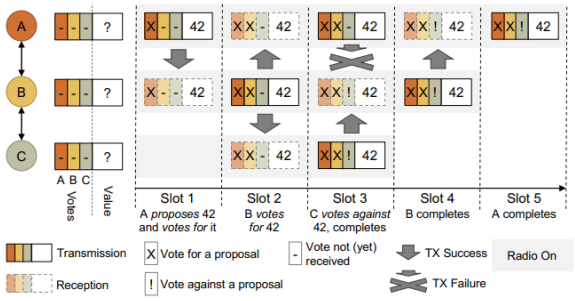


Figure 2.13: Network-wide voting in A^2 over 3 nodes [2].

As can be seen in Figure 2.13, voting is very similar to a one-to-many broadcast in Chaos. The coordinator proposes a given value (as a payload) and the flags section of each packet is used to carry information on the state of the vote (2 bits of information are necessary for each node: `0x00` represents no vote cast, `0x01` is a vote in favour, `0x10` is a vote against. Voting flags get merged at every slot and the aggregated packet is continuously broadcast until completion (i.e. all votes have been cast, and all nodes know about the outcome).

Just as all original Chaos primitives, voting is best-effort as there is no guarantee that all nodes will have received the final aggregated packet containing all the votes.

2.6.4 Two and Three-Phase Commit

Two-phase commit (§ 2.2.2.1) is a simple consensus protocol which can be achieved in A^2 based on top of the network-wide voting primitive (§2.6.3). A sample run can be seen in Figure 2.14.

- **Voting phase**. Nodes use the network-voting primitive illustrated above to vote on a proposed value (as can be seen in Slots 1 - 4 of Figure 2.14). Once the voting stage is complete nodes remain active awaiting the coordinator's decision.

- **Completion phase**. Based on the voting result, the coordinator decides whether to commit or abort the transaction. The coordinator spreads the decision and nodes adopt it.

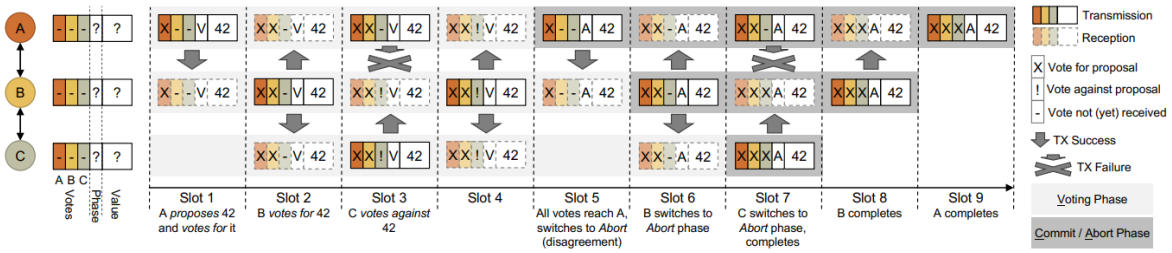


Figure 2.14: Two-phase commit protocol running in A^2 with 3 nodes [2].

2PC is a blocking protocol, meaning that node failures impact the latency of the whole network, possibly preventing termination. Due to this three-phase commit (§ 2.2.2.2) is introduced, which trades liveliness at the expense of safety (i.e. consistency cannot be enforced). It's implementation in A^2 is as follows:

- **Voting phase**. Identical to 2PC.
- **Pre-commit or abort phase**. The decision of the coordinator is disseminated to the whole network. Nodes are not allowed to vote at this stage so they simply lock the resources (in case of pre-commit) and acknowledge the reception of the message by setting their specific flag in the packet. This is identical to how Chaos is able to check if all nodes have received the message.
- **Do-commit phase**. Similarly to the previous stage, the coordinator disseminates the do-commit decision to all nodes.

2.7 WPaxos

The problem of consensus has been studied for decades in traditional wired networks, yet it still remains challenging within the realm of WSNs. Wireless Paxos [32] (or WPaxos for short) brings fault-tolerant consensus to wireless networks by building it within the Synchrotron (§2.6.1) transmission kernel. WPaxos is designed to integrate as closely as possible to the underlying Chaos ST primitive using fast in-network processing and minimal delays between broadcasts.

To be able to successfully execute the Paxos algorithm on a distributed broadcast-only networks such as a WSN testbed, a number of adaptations to Leslie Laport's original algorithm [24] are necessary:

- **Beyond unicast**. Protocols for wired networks are built on top of a one-to-one, unicast dissemination strategy. When broadcasting over the radio, messages are heard by all neighbouring nodes. This multicast strategy can be used to reduce overall protocol latency. Nodes participating within a WPaxos round will constantly aggregate all of the communication that they receive on the network and retransmit the most up-to-date aggregated packet.

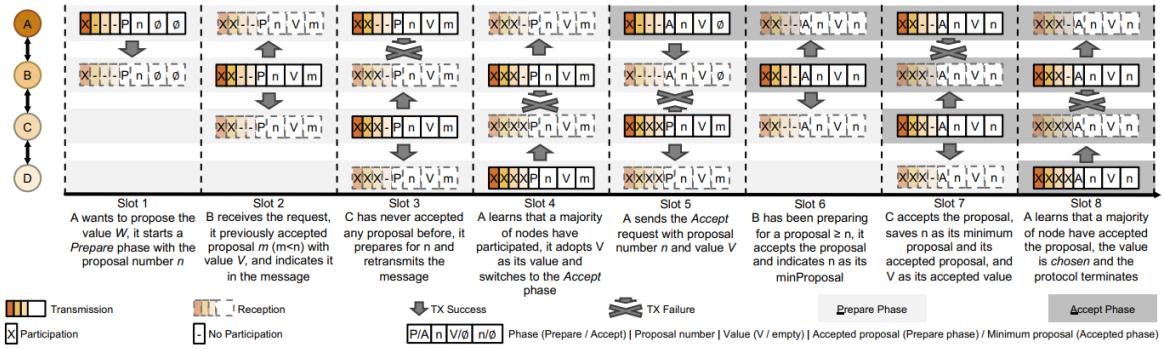


Figure 2.15: WPaxos execution with 4 nodes. Node A is the initial proposer [32].

- **Proposers and acceptors**. All nodes act as acceptors during a Wireless Paxos round and nodes can optionally act as proposers. Should a node be a proposer all proposer logic will be executed before the acceptor one.

Additionally there are a couple characteristics of the original Paxos implementation that map into the Wireless Paxos model:

- **Proposal cohabitation**. Multiple proposers may be active at any time within the network. Proposals are ordered using unique monotonically increasing ballot numbers and acceptors will only rebroadcast the proposal they received with the highest ballot number (to prevent the network form stalling).
- **Phase cohabitation**. Paxos is a two-phase protocol (i.e. "prepare-promise" and "accept-accepted") and both phases might coexist during any Chaos round. To reduce the number of messages a heuristic is in place to ensure nodes will keep transmitting a newer phase even if they receive older phase information. Furthermore nodes will reduce their transmission rate should they see prepare-phase packets which already contain a majority of replies; they will resume back to a normal rate once they detect the proposer has initiated the subsequent accept-phase.

Wireless Paxos introduces the idea of reliable distributed consensus to wireless networks. Multiple nodes are able to proposes values to the network and eventually one of these values will be chosen and agreed upon. It therefore satisfies all Paxos safety conditions [23]:

1. Only a value that has been proposed may be chosen,
2. Only a single value is chosen, and
3. A process never learns that a value has been chosen unless it actually has been.

Furthermore it also ensures that the liveliness condition is preserved: eventually (in absence of Byzantine failures) a value will be chosen by the protocol. Reliability, though, is hindered by bad network conditions. In the presence of interference or channel noise WPaxos' Chaos floods are unable to reliably reach all network nodes, increasing the protocol's latency and impacting its robustness.

2.7.1 Wireless Multi-Paxos

Multi-Paxos [44] is a common extension of Paxos which introduces the ability for the network to agree on a sequence of values by executing multiple consensus rounds. Many rounds can

be executed in parallel or sequentially, potentially allowing for proposers to merge many concurrent proposals into one single packet. Together with WPaxos a novel Wireless Multi-Paxos solution was proposed based on four main design rationales:

1. **Bounded memory**. In its original implementation an unbounded number of ballots is allowed to exist at any time. To mitigate the problems potentially arising due to limited memory available on the motes, log and packet buffer sizes are fixed and ballots may be aggregated by proposing nodes during the protocol's execution.
2. **Prepare-phase specifics**. Proposers are constantly required to learn the transaction outcome of all previous rounds to prevent inconsistencies. This is impossible due to the packet size constraints. An iterative process used, allowing proposers to gradually learn all values already committed by the network.
3. **Message ordering**. Ballot numbers must be universal and monotonically increasing across all proposers. To guarantee uniqueness the lower digits of each proposal number (PN) carry over the ID of the proposer which generated it.
4. **Long lasting leader**. For fast execution of multiple Paxos rounds there cannot be a constant fight for ballot numbers across multiple proposers. Nodes with the highest ballot are acknowledged as leaders by the network for lasting periods of time. This way values are voted upon quickly and efficiently by the network.

Overall Wireless Multi-Paxos introduces quick and efficient multi-proposer consensus to wireless multi-hop networks, while suffering from the same limitations of WPaxos: unreliability and poor performance under non-optimal network conditions. The existing codebase is currently developed for Synchrotron within the Contiki 2.7 Operating System.

2.8 Baloo

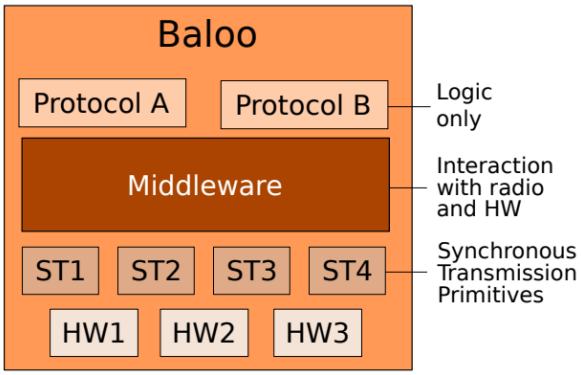
Synchronous transmissions (ST) popularised by Glossy are a highly reliable and energy efficient method of communicating in low-power multi-hop networks. A number of ST-primitives (i.e. protocols that execute any-to-all broadcasts in bounded time) exist, such as Glossy or Chaos, though they heavily rely on low-level control of timers and radio events. Designing a network-stack over an ST-primitive can therefore become a challenging and time-consuming task. Baloo [21] aims to simplify the process by introducing a middleware for synchronous transmissions with the following guarantees:

- Expose a well-defined interface enabling run-time control of ST-primitives by the network layer.
- Allow for implementation of a wide variety of network layer protocols.
- Network layers should be able to use multiple ST-primitives, potentially switching them at runtime.
- The middleware should not impact the time synchronization requirements of ST-primitives.

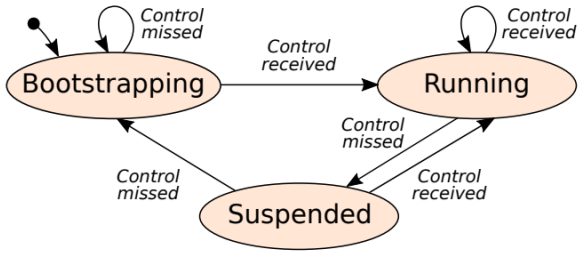
Overall Baloo provides an extensible middleware which has already working implementations for algorithms such as LWB (§2.4). It is being developed within the new Contiki-NG codebase (§2.1.2) and has a strong documentation effort underway. This enables this state-of-the-art project to be used as a strong baseline for the implementation of more complex algorithms such as A^2 (§2.6).

2.8.1 Middleware Overview

Implementing ST-primitives such as Glossy and Chaos, Baloo is designed to work in Time Division Multiple Access (TDMA) [37] execution rounds. All vital protocol information will be sent to the network by a central node on the first slot of each communication round. It is designed to provide a middleware which abstracts low-level cumbersome operations to the higher-level protocols (see Figure 2.16a). The middleware is in charge of all timers and radio operations. Control information can be provided by the protocol within callback functions exposed by Baloo. The entirety of the protocol logic is implemented within the callback functions (§2.8.1.1).



(a) Overview of Baloo's design [21].



(b) State machine of Baloo's middleware [21].

Figure 2.16: Baloo components: design overview and middleware state-machine

The middleware has a minimal state-machine (see Figure 2.16b) which consists of three different states (as defined below). Baloo is driven by control packets which are sent at the beginning of each round. The packets contain schedule information (i.e. how to execute the current communication round, and when to wake up for the next round), and configuration information (i.e. slot length and retransmission count). Nodes that have successfully received and decoded control packets can transmit during the subsequent allocated data slots.

- **Bootstrapping**. When booted a node waits to receive a control packet. Upon reception it can enter the running state for the round specified in the control packet itself.
- **Running**. When running a node is able to participate in the network for all protocol operations. Should a node miss a control packet it enters the suspended state.
- **Suspended**. Suspended nodes will quickly resume their running state if they are able to receive subsequent control packets. Should they sense that multiple other control packets have been missed, they return to the bootstrapping state; the node is likely out of sync and requires brand new fresh copies of schedule and configuration information.

With the abstraction introduced by the middleware (Figure 2.17) it is possible to separate the protocol implementation with the lower level manipulation of data packets, data transfers and timing model. The fixed Baloo middleware allows users to have simple and portable protocol implementations which uniquely communicate with the middleware. The underlying ST primitives (such as Glossy or Chaos) may be changed without impacting the soundness of the protocols themselves.

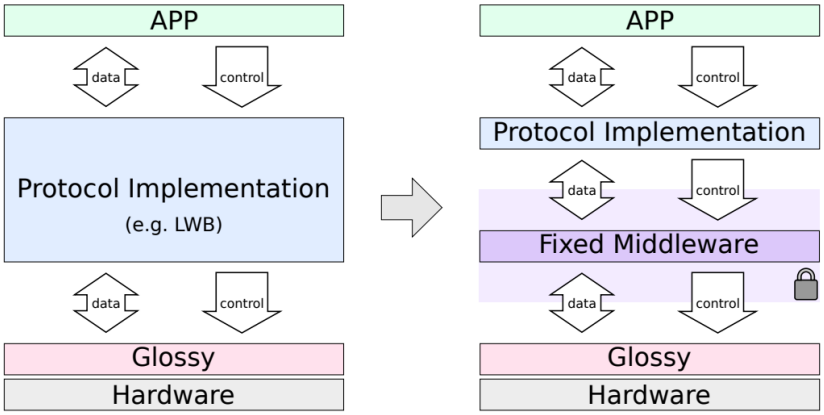


Figure 2.17: Protocol-level abstraction achieved by using Baloo's middleware [5].

2.8.1.1 Callback functions

During the running stage, nodes expose callback functions which can be used to implement high-level protocols over the Baloo middleware. There are five different callbacks which can be seen in operation in Figure 2.18.

- `on_control_slot_post()`. At the end of a control slot, it can be used to process the information present in the control packet.
- `on_slot_pre()`. Executed before each data slot, allows to send a custom payload to the middleware.
- `on_slot_post()`. Executed at the end of each data slot, allows to process the received payload.
- `on_round_finished()`. Executed at the end of each round allows to execute more time-demanding processing or state management.
- `on_bootstrap_timeout()`. Executed when a node fails to bootstrap (i.e. does not receive any control packets). The protocol might require nodes to go to sleep for some time to save on energy consumption.

Arbitrary code can be run in the Baloo defined callbacks. This might impact maximum synchronization delays required by the various ST primitives. Baloo solves this by allowing

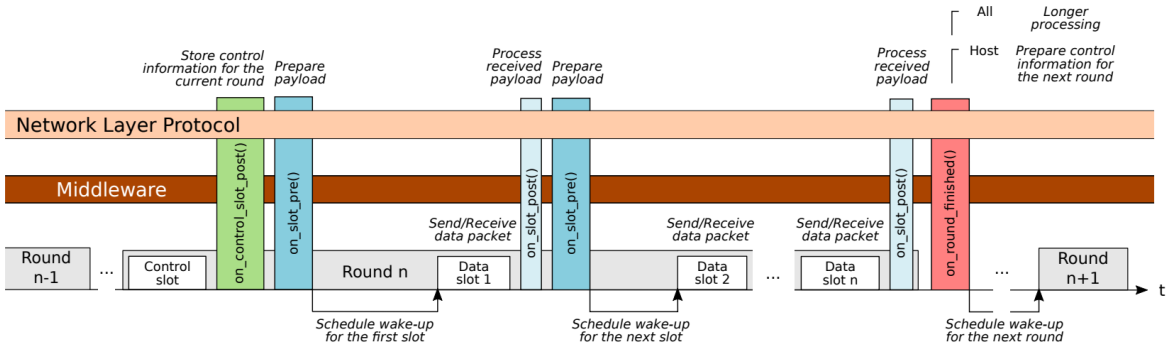
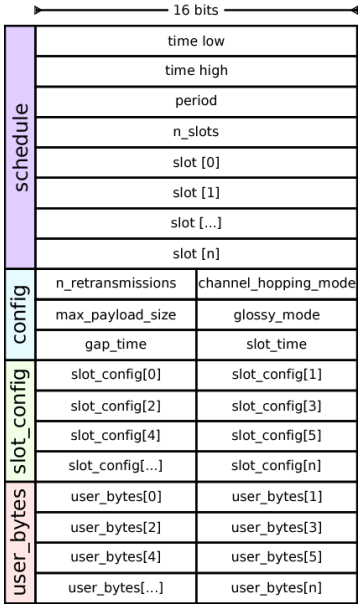


Figure 2.18: Protocol running on top of Baloo middleware [21].

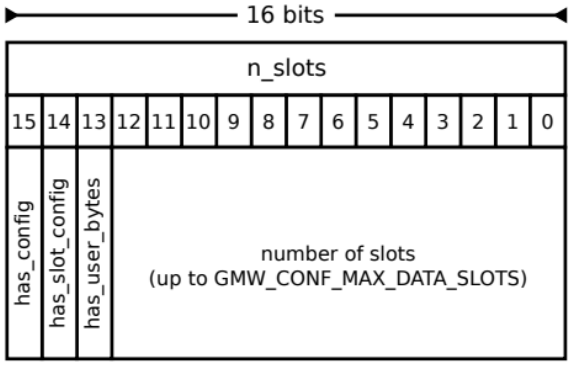
the user to configure a parameter, called `gap_time`, that sets an upper-bound for callback execution time.

2.8.1.2 Control packet

The control packet (Figure 2.19a) is at the core of Baloo's synchronous design. It is sent by the global host at the beginning of each round and it contains all the information required by all round participants to transmit and duty cycle correctly.



(a) Baloo control packet [5]



(b) Configurable control packet parameters [5]

Figure 2.19: Full Baloo control packet structure including optional configuration slots.

It is split into 4 main sections:

- `schedule`: The only compulsory part of the control packet, holds all the information required by the nodes to complete the next round. It holds the global time measure and all slot assignment and scheduling. The schedule is allowed to vary for each new round allowing the order of the scheduled nodes to change. Slots can be configured to be "standard" with one specific node assigned, or "contention" where any node willing to broadcast to the whole network may do so. In case more than one node broadcasts within a contention slot, due to the capture effect, at least one of the replies will be heard by the global host.
- `config`: The configuration section holds middleware configuration parameters which may vary at runtime. Most importantly the underlying ST primitive (i.e. Glossy or Chaos) could be changed, and together with them the slot lengths and retransmission numbers. This section is optional and if not present all nodes will use the globally configured defaults.
- `slot_config`: Further (optional) per-slot customisation can be achieved with this section of the packet. If a specific slot must deviate from the configured slot length and retransmission numbers, it can be customised in this section.

- `user_bytes`: This final optional section of the control packet allows the global host to send arbitrary data to all the nodes within the control packet. It can be used by protocols to configure the behaviour of the network during the next round.

When any of the 3 optional sections of the control packet (`config`, `slot_config` or `user_bytes`) are used, their presence must be declared by setting specific bits of the `n_slots` section of the `schedule` (see Figure 2.19b).

2.8.2 Practical Applications

Baloo allows protocols to use multiple ST-primitives. It currently supports a Glossy and Chaos and can likely be easily extended to support newer primitives such as Mixer [18]. Other than the concepts already discussed it additionally supports detection of interference, potential for an advanced state machine and starvation protection for nodes. Baloo's advanced scheduling functionality allows it to currently have implementations for state-of-the-art algorithms such as LWB, Crystal [20] and Sleeping Beauty [35].

Baloo is developed on top of a Contiki-NG fork, allowing for easy extensibility within a fast growing and popular operating system. It is well documented and aims to finally bridge together academic protocol implementations with real-world reproducible results. It is important to note that although Baloo allows for protocols to switch ST primitives, to the best of our knowledge this feature was only every used once as a EWSN 2019 Dependability Competition submission [29]. Data dissemination rounds were executed via Chaos and acknowledgements were gathered in subsequent phases using Glossy.

In this thesis we propose a new reliable approach to using ST primitives by switching between Chaos and Glossy floods during the dissemination and gathering of the same data. This new primitive, Hybrid, whose implementation is made possible by Baloo, requires numerous additional configurations alongside the middleware. This project therefore builds on top and extends Baloo to introduce a novel, reliable, multi-phase voting library: XPC.

2.9 Current Experimental Methodology

WSN protocols can be assessed in a variety of ways. Their evaluation may consider multiple metrics including packet loss, latency, throughput or even clock synchronization. To aid this investigation and analysis it is common for protocols to be firstly developed and debugged within a simulation environment, and subsequently be thoroughly tested on actual hardware within a testbed.

As not all software stacks fully support hardware emulation (for simulation purposes), certain libraries, such as Baloo's middleware, cannot be tested out on Contiki's simulation environment (Cooja [13]). Such code must therefore be tested out on real hardware, and can potentially be more cumbersome to debug, test and analyse.

2.9.1 Testbeds

Simulations provide very good approximations of expected broadcast outcomes and general network behaviour, though when dealing with physical phenomena such as the capture effect it is important to thoroughly test the behaviour on actual hardware. The isolated simulation environment might not model environmental factors, such as potential channel interference, possibly yielding inaccurate results if not backed up by real-world experiments. Given that it becomes very expensive and time-consuming to distribute binaries and test them "in-house"

for large topologies, in addition to the possibility of error or inaccuracies in simulations, researchers have developed a number of WSN testbeds all across the globe [40].

Most commonly, WSN testbeds are large mote deployments at academic institutions and are mainly accessed by researchers. Ideally testbeds want to achieve some isolation or shielding from outside interference. Deployments in indoor locations achieve this, allowing for higher result replicability and easier maintenance. The majority of testbeds have the same characteristics:

- **Known topology**. Testbed deployments have a fixed known topology. This allows for specific experiments which might prefer to target particular network configurations.
- **Specific devices**. Academic projects tend to target different types of motes and architectures as they have different instruction-sets and might allow for varying programming flexibility.
- **Backend server**. The server manages the scheduling of the tasks to the nodes at deploy time, and handles and centralizes all of the logging done by the motes during the experiment.
- **GUI**. Graphical User Interfaces are usually in place to allow for easy deployment to the motes and task scheduling.

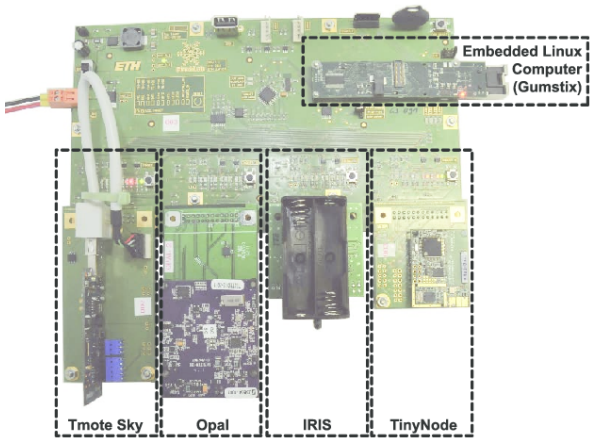


Figure 2.20: TelosB node (referenced as Tmote Sky) attached to a FlockLab observer [26].

Testbeds are being updated constantly. Embedded hardware is prone to failure, and older motes tend to get replaced to make room for newer ones. For this reason it is important to choose testbeds with a fairly large number of motes which can be extensively used throughout the length of the project. Furthermore, in order to guarantee the scientific robustness of gathered results, when comparing against existing protocol implementations, data should be gathered using the same testbed. Minimal variations in the network condition allow results to be comparable and for protocol enhancements to be clearly visible.

2.9.1.1 Indrya2

Indrya2 [4] is a WSN testbed deployed at the National University of Singapore. Indrya2 features 74 TelosB motes and 28 CC2650 SensorTags. TelosB motes are the most commonly used platform in papers and academic research. They are equipped with a MSP430F1611 Microcontroller, and a IEEE 802.15.4-compatible CC2420 radio.

2.9.1.2 FlockLab

FlockLab [27] is a WSN testbed based in Zurich, Switzerland. It consists of 27 observer nodes (Figure 2.20) connected with each other over LAN (Figure 2.21). An individual FlockLab observer node has numerous sensor nodes attached to it, among which a TelosB (also known as Tmote Sky). Similarly to Indrya2 it provides an online web-interface with monitoring capabilities and on-line performance metrics. Focklab offers logging capabilities for GPIO (General Purpose Input/Output) pins, which are used as LEDs, together with all the data which is written to the Serial Port. All output is timestamped to allow for easy comparison across different network nodes. Even though Flocklab also supports chips such as Dual Processing Platform (DPP), the TelosB mote is the target platform used throughout this thesis.

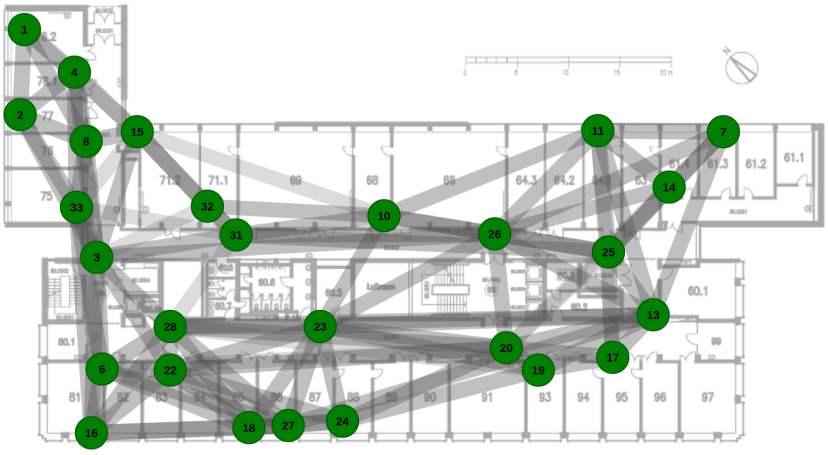


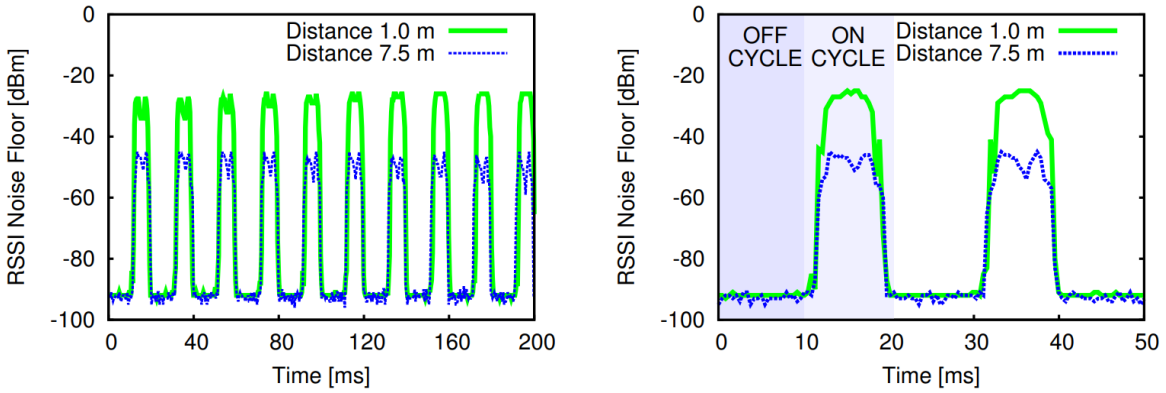
Figure 2.21: Flocklab connectivity map with 27 nodes [27].

2.9.2 Modelling Interference

When executing tests on a remote testbed there are a number of factors which could affect the repeatability of gathered results. Amongst these the most prominent is interference, or radio noise, occurring on the same channels used for 802.15.4 network transmissions. As most testbeds are deployed within office buildings they offer little to no control over the testing environment: multiple researchers will be using the same frequencies, causing packet collisions and missed broadcasts. Additionally WiFi and microwave ovens share the 2.4 GHz spectrum used by 802.15.4, causing additional disruption. A method used to ensure testing reliability and repeatability is to artificially inject interference patterns into the network, covering the background channel noise with a stronger, distinct, interfering wave. Furthermore the background noise can be analysed, as the additional traffic during day-time hours causes additional disruption to the medium, if compared to night-time. Analysing a protocol's performance with different background-noise levels and under different levels of injected interference provides thorough insight in it's robustness and reliability.

This is the approach taken by JamLab [6]. It provides low-cost flexible testbed infrastructure to allow the repeatable generation of a wide range of interference patterns. When using JamLab a number of nodes in the network will be designated as "jammer" motes, thus being removed from the list of protocol participants. JamLab nodes can model three different interference sources:

- **WiFi**. Using a probabilistic approach, JamLab can emulate the interference impact of WiFi communications occurring on the 2.4 GHz ISM band. It has models for saturated (i.e. constant data streaming) and non-saturated (i.e. sporadic broadcasts) traffic sources.
- **Bluetooth**. More complex than WiFi, Bluetooth is allowed to perform Adaptive Frequency Hopping (AFH) to combat interference on devices. Hops occur 1600 times a second and cannot be predicted deterministically. Similarly to the non-saturated WiFi transmissions, JamLab emulates the effects of Bluetooth broadcasts with a probabilistic model.
- **Microwave Oven**. Well known kitchen appliances, microwave ovens heat food by usually emitting particles at 2.45 GHz frequencies. Being close to the 2.4 GHz band it is possible for these particles to interfere with 802.15.4 transmissions (with a higher impact on channels 20-26 [6]). Microwave ovens follow a simple on/off transmission mechanic, which is easier to model for interference purposes (see Figure 2.22).



(a) Zoom Out.

(b) Zoom In.

Figure 2.22: Temporal characteristics of JamLab microwave oven interference [6].

JamLab offers a simple and efficient solution to reliably injecting interference during protocol evaluations. It does, though, come at a topology cost. A number of network nodes must be removed from the main protocol pool and assigned a jamming-only task, this can potentially hinder comparisons with existing papers, as the number of nodes used to schedule the tests would differ. Additionally it is important to configure a "sufficient" number of interfering nodes within the network. As all interference is executed via the mote's radio transceivers, the signal will decay over distance meaning that multiple JamLab nodes must be used to replicate the effects of a single WiFi stream or microwave oven.

2.10 Current Challenges

This chapter has introduced much of the background for Wireless Sensor Networks, constructive interference and consensus. Table 2.1 summarizes all the problems which arise with state-of-the-art literature, together with the solutions which have been provided so far. A number of issues, such as reliability and the adoption of consensus-based protocols, remain currently unsolved.

Section	Technology	Problem	Our Solutions
§ 2.1.3	WSN broadcasts	Unreliable reception and transmission guarantees.	Capture effect and constructive interference.
§ 2.1.4	Capture effect and interference	Difficult to achieve in orderly network-wide fashion.	Glossy-based protocols to provide synchronous network-wide flooding.
§ 2.2.1	Consensus in WSNs	Unreliable links could stall protocols indefinitely.	Reliable links via Glossy-based protocols **(P1)**.
§ 2.2.2.1	Two-phase commit	Blocking protocol cannot make progress with node failures.	Three-phase commit, which sacrifices safety for liveliness.
§ 2.2.2.2	Three-phase commit	Unable to recover from a network partition.	Majority-based consensus algorithms such as Paxos.
§ 2.3	Glossy	Requires structured schedule to prevent colliding floods.	Low-power wireless bus uses a global scheduling host.
§ 2.4	Low-power Wireless Bus	Slow for many-to-many communication rounds over large networks.	Chaos protocol performs aggregation during round dissemination.
§ 2.5	Chaos	Cannot reliably retrieve data from individual nodes.	Requires a new low-latency and reliable ST primitive **(P2)**.
§ 2.6	Agreement in the air	Only implements 2PC and 3PC.	WPaxos extension introduces consensus.
§ 2.7	WPaxos	Interference has a big impact on reliability.	Must use a new low-latency and reliable ST primitive **(P3)**.
§ 2.8	Baloo	Requires a lot of configuration to switch ST primitive during a phase.	All configuration code to be handled by a new multi-phase voting library **(P4)**.
§ 2.9	Experimental Methodology	Determining true protocol reliability under interference.	Reliability must be tested using JamLab for WiFi and microwave oven interference **(P5)**.

Table 2.1: Summary of problems addressed in Chapter 2 together with the problems **P1**-**P5** which remain unsolved in the current literature.

The problems present in the current literature (**P1**-**P5**) are highlighted in Table 2.1. They can be reformulated into the following challenges which this thesis aims to solve:

C1 WSN communications must be low latency and reliable. A standalone use of Glossy and Chaos ST primitives is insufficient. We need to propose a new ST primitive which is able to provide the robustness of Glossy together with the performance of Chaos. This challenge encapsulates **P2**.

C2 Protocols must be able to easily switch between ST primitives. Protocol implementations

must not rely on a specific dissemination strategy and must be able to run with different ST primitives to allow for reliability and latency comparisons. An extension on Baloo needs to be developed to address this. With the creation of this new library we address **P4**.

C3 Consensus protocols must run reliably on WSNs. Existing approaches to consensus need to use a new ST primitive to provide strong liveliness and safety guarantees. This challenge encapsulates **P1** and **P3**.

C4 Protocol reliability has to be tested with JamLab to allow for replicability. By using JamLab to inject interference during our evaluations we address **P5**.

This thesis addresses **C1** and **C2** by introducing Hybrid (a new reliable ST primitive) and XPC (an extension to Baloo to allow for switchable ST primitives) in Chapter 3. Consensus protocols WISP and WIMP (two new flavours of WSN Paxos) further address **C3** in Chapters 4 and 5 respectively. All protocol and ST implementations are tested with JamLab for reliability (**C4**), and compared to state-of-the-art protocol implementations in Chapter 6.

3 | Hybrid ST Primitive and XPC

We introduce Hybrid, a new synchronous transmissions network dissemination strategy aimed at minimising latency and maximising reliability. Hybrid leverages the optimal Chaos flooding latency and optimises it with Glossy's reliability. In order to allow for configurable ST primitives independent of protocol implementations we introduce XPC alongside Baloo. XPC is a novel, highly configurable, multi-phase voting library which allows us to easily recreate existing voting protocols within the Baloo middleware, matching state-of-the-art latencies and enhancing overall reliability. By implementing the Hybrid ST primitive and XPC we address and solve **C1** and **C2** respectively.

Building up to the implementation of Hybrid, we will first commence in Section 3.1 by analysing the interactions and configurations required on top of Baloo to implement multi-phase voting protocols. This motivates the necessity for XPC, a standalone library implementation which places itself alongside Baloo's middleware. Section 3.2 studies the behaviour of 2PC and 3PC voting protocols with existing ST primitives. Glossy and Chaos are analysed in Sections 3.2.1 and 3.2.2 respectively. Section 3.2.3 culminates our analysis by introducing Hybrid, a new dissemination strategy with stronger timing and robustness guarantees than both Glossy and Chaos.

3.1 XPC fundamentals

XPC is a library which simplifies the use of different synchronous transmission (ST) primitives for multi-phase voting protocols built on top of Baloo. With XPC, protocols may interchangeably use any ST primitive during their execution, allowing for the implementation of Hybrid. XPC is structured into four main components:

1. **Application**: XPC abstracts all Baloo code away from the application. Network-wide voting is a service which is executed every round and the application may interact with it by either proposing a value for the next round, or by inspecting the outcome of the previously executed round (i.e. which value was committed or aborted by the network). The application layer is fully customizable by the user, as arbitrary code can be executed by all nodes. The only requirement is for the node to poll upon completion. By polling Baloo will duty-cycle the node and execute all XPC code prior to the application being preempted once again. This allows for a strong separation of application and protocol implementations, enabling nodes within a network to use XPC as a service for reliable dissemination of values agreed upon by the whole network.
2. **Protocol implementation**: XPC allow protocols to be implemented on top of a minimal Baloo-like structure (see Figure 3.1). Protocols are only required to handle internal state-machine transitions and packet pre and post processing. Network dissemination

and primitive-specific tuning is handled via specific API calls to other layers of the XPC stack.

3. **Common code**: All handling of packet buffers, message parsing, and all retransmission policies have been packaged into a "common" section of XPC's implementation to allow the protocol code to be as simple and readable as possible. Within the codebase the section is referred to as `xpc-common`.
4. **ST primitives**: Different ST primitives have completely different message packet structures. Baloo's control packet `schedule` and `config` sections will also differ. Protocol implementations specify which ST primitive to use to exchange messages each round, and the primitive-specific code will handle generation of the appropriate packet, dissemination during the round and aggregation of the results. XPC has a primitive-independent API to configure network-layer buffers. All primitive-specific code is re-run each round allowing the user to potentially switch between different ST primitives within the same protocol.

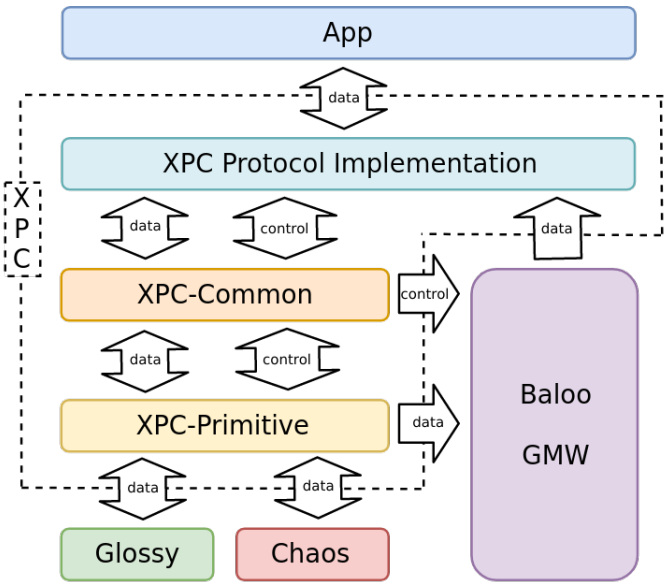


Figure 3.1: Layered overview of all XPC components. XPC lives alongside Baloo's implementation processing all data incoming from the application and managing the control structure.

3.1.1 Adjustments for Voting Protocols

To implement voting protocols we used and configured Baloo in the following ways:

1. **Single Initiator**. Baloo relies on the presence of a global host. This node is in charge of bootstrapping the network and sending control packets at the beginning of each flood. For our 2PC and 3PC implementations this node is in charge of the protocol and of the initiator's state machine. It can be argued this simplifies the implemented protocols to single-initiator ones. Multiple-initiation, though, could simply be achieved by having the nodes communicate the values to the host first, delegating all initiation privileges to the global node.

2. **Retransmissions for Reliability**. As stated in the background, WSN links are very unreliable and packets may be lost due interference or environmental conditions. Even though Glossy is an incredibly reliable protocol, dealing with large networks will always involve missing packets. To mitigate this issue, our protocols will implement retransmission procedures that allow the execution of a given phase more than once, should there be missing replies from a part of the network. This measure greatly increases reliability (as we will see in Section 3.2).
3. **Additional Final Round**. Baloo relies on a tightly timed transmission middleware. At the beginning of each round nodes receive a control packet which contains exact information on their radio-on times. This decision fixes the configuration for the current round and new, updated, radio duty-cycle times can only be communicated with additional control packets. With our retransmission policy, though, at the beginning of a Baloo round, we are unsure if our protocol will terminate, or if, instead, a "retransmission" round has to be scheduled right after due to missing replies. This forces us to introduce an extra, final, and empty round at the end of each protocol run. As can be seen in Figure 3.2 during rounds 1 to N the host sends out a control packet C expecting all nodes in the cohort to reply during their scheduled slots. If the protocol implementation is able to terminate after round N, a final ending round (denoted as E) has to be scheduled, where the global host communicates to all nodes the protocol run is over. All nodes will therefore turn off their radios and allow their applications to access the result of the protocol's execution.

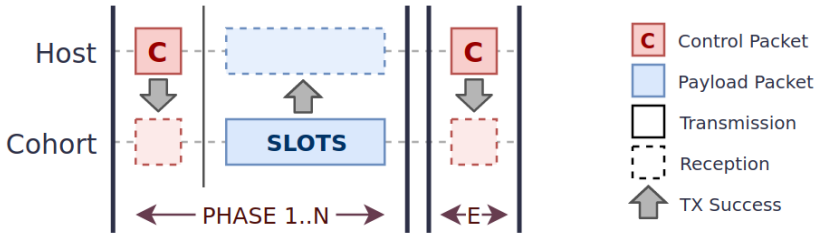


Figure 3.2: Example N-Phase protocol ported to Baloo's round structure.

In order to support these adjustments, there are a number of common Baloo configurations that will be used by all protocols, regardless of their underlying ST primitive:

- `schedule.period`. All protocols will share the same overall period. This is the length of time allocated for the execution of the application after a successful iteration of the protocol. This can be configured to adapt the protocol runs to the needs of the top-level application.
- `user_bytes`. Due to the nature of the single global initiator, all protocol information necessary for a given round will be disseminated within the control packet itself. The host assigns two sections of the optional `user_bytes` configuration parameter: the first holds the message sent by the host to all nodes in the cohort, the second holds the value currently proposed by the host.

All above Baloo configurations, together with any additional primitive-specific adjustments are necessary to enable the development of quick, simple, and clean protocol implementations. The XPC layered library is written to take care of this unique configuration management, handling all communication to GMW on behalf of the application.

3.1.2 Initiator and Participant

Building alongside Baloo, all XPC protocols are implemented with one node acting as an initiator and all other nodes acting as participants. Even though all `xpc-common` and `xpc-primitive` code is independent to such a division, these two different node roles have slightly different implementations for the various Baloo callback functions. We analyse such a distinction by viewing high-level pseudo-code for an XPC initiator node (see Algorithm 1) and an XPC participant node (see Algorithm 2). Within the pseudo-code we denote as parameters to the callback functions all variables present within the global state of the node that will be modified during the execution of the function itself. All variables are passed by reference and updates using the ← operator modify the internal state of the object referenced by the given variable.

Algorithm 1 XPC Initiator Pseudocode

```
1:  function ON_ROUND_FINISHED(state, message, retr_cnt, reply_num)
2:      retr_cnt ← retr_cnt + 1
3:      n_slots = XPC_COHORT_NODES - reply_num
4:      if retr_cnt > XPC_TIMEOUT_RETRANSMISSIONS then
5:          state = XPC_ABORT_STATE
6:      else if reply_num == XPC_COHORT_NODES then
7:          STATE_TRANSITION(state)
8:          n_slots = XPC_COHORT_NODES
9:          retr_cnt ← 0
10:     PREPARE_MESSAGE(state, message)
11:     Prepare the Control Packet with the message and n_slots
12:
13: function ON_SLOT_POST(state, message, reply_num, node_id)
14:     if we have not yet received a reply from node_id while in this state then
15:         PROCESS_MESSAGE(state, message, reply_num)
```

The XPC initiator (usually the Baloo host) is in charge of the protocol's progress and therefore determines the contents of the control packets for each round (Algorithm 1). Control packets must always be generated one round in advance, hence the `on_round_finished` callback executed at the end of round t must generate the new control packet which will be sent to initiate round $t + 1$.

When generating a control packet XPC will first determine how many nodes must be scheduled to reply within the given round (set using the `n_slots` field on line 3). It then determines if it has retransmitted more than the retransmission limit (line 4). If all nodes have replied during the previous round (lines 6-9), the initiator's state is updated to execute the next protocol phase. The control packet is then prepared with the correct `message` which should be sent by the protocol when at the given state (lines 10-11).

All other protocol logic is executed within the `on_slot_post` callback. The initiator processes replies from all of the nodes and determines if the correct nodes have replied and how each reply impacts the initiator's state machine. Implementations for `state_transition(...)`, `prepare_message(...)` and `process_message(...)` are protocol dependent and will differ for each different voting protocol implemented using XPC.

Nodes participating within an XPC round (see Algorithm 2) will parse the control packet and attempt to execute a state transition based on the new information sent by the host

Algorithm 2 XPC Participant Pseudocode

```
1: function ON_CONTROL_SLOT_POST(state, control, message, retr_cnt)
2:     retr_cnt ← retr_cnt + 1
3:     if retr_cnt > XPC_TIMEOUT_RETRANSMISSIONS then
4:         state = XPC_ABORT_STATE
5:     if state == XPC_ABORT_STATE then
6:         message ← XPC_DO_ABORT
7:     prev_state = state
8:     STATE_TRANSITION(state, control, message)
9:     if prev_state ≠ state then
10:        retr_cnt ← 0
11:
12: function ON_SLOT_PRE(state, message, node_id)
13:    if it is node_id's turn to transmit then
14:        Send the message using the correct primitive
```

(line 8). Similar to the XPC initiator, all participant nodes are allowed to timeout if the information sent by the host does not cause a change in their state after a set amount of retransmissions (lines 2-4, 7, 9-10). This is to prevent a possible deadlock conditions. A host may miss too many replies from a participant and time-out. The current round is aborted and then a new value is proposed. Due to interference in the network a participant may miss a control packet containing timeout-abort information and may infinitely wait for a specific message from the host, stalling the whole network for multiple rounds. Upon reaching an `ABORT_STATE` due to a timeout, participants show the same behaviour as if they received a `DO_ABORT` message within the control packet. Similarly to the initiator code, implementations for `state_transition(...)` are protocol dependent and will be specified for each protocol implemented via XPC.

XPC therefore offers an API for stateless primitive manipulation. Individual protocols are mainly left with the task of handling their internal state machine on top of a simplified minimal Baloo callback structure. We therefore meet and solve **C2** presented in Section 2.10. In the next section we provide and explain two voting protocols implemented with XPC.

3.1.3 Example Voting Protocols in XPC

We provide reference implementations for Two-Phase commit (§ 2.2.2.1) and Three-Phase commit (§ 2.2.2.2) protocols to illustrate the functionality of XPC. This further allows XPC to match the existing literature (such as A^2 Synchrotron), providing a common point of comparison for all introduced reliability and robustness enhancements.

3.1.3.1 Two-Phase Commit

We provide a reference implementation of 2PC to illustrate XPC. Two-Phase commit can be translated very neatly to XPC's time-sliced paradigm. Assuming that all the nodes in the network intend to commit, the phase structure is as follows:

- **Phase 1** (see Figure 3.3) contains the `CAN_COMMIT` message in the Control packet, sent by the host to all the nodes in the cohort. Nodes willing to commit the value reply with `VOTE_YES`, otherwise they reply with `VOTE_NO`.

- **Phase 2** begins when either all nodes have voted yes, or at least one node has voted no. `DO_COMMIT` or `DO_ABORT` messages are conditionally sent in the control packet. Requests to commit are acknowledged by nodes with `HAVE_COMMITTED` messages.

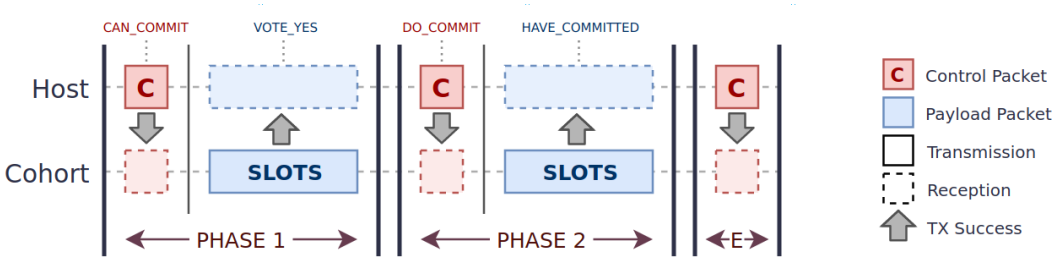


Figure 3.3: 2PC Protocol in Baloo with 1 retransmission.

The above breakdown assumes replies are always heard from all nodes. In the case of missing replies, the host is given a maximum number of retransmissions (`XPC_TIMEOUT_RETRANSMISSIONS`) to attempt to reach stragglers within each phase. If these nodes remain

Algorithm 3 2PC Initiator

```
1: function STATE_TRANSITION(s)
2:     switch s do
3:         case XPC_INIT_STATE
4:             s ← XPC_READY_STATE
5:         case XPC_READY_STATE
6:             s ← XPC_COMMIT_STATE
7:         case XPC_COMMIT_STATE
8:             Commit value
9:             s ← XPC_INIT_STATE
10:        case XPC_ABORT_STATE
11:            Abort value
12:            s ← XPC_INIT_STATE
13:
14: function PREPARE_MESSAGE(s, m)
15:    switch s do
16:        case XPC_INIT_STATE
17:            m ← XPC_EMPTY_MESSAGE
18:        case XPC_READY_STATE
19:            m ← XPC_CAN_COMMIT
20:        case XPC_COMMIT_STATE
21:            m ← XPC_DO_COMMIT
22:        case XPC_ABORT_STATE
23:            m ← XPC_DO_ABORT
24:
25: function PROCESS_MESSAGE(s, m, r)
26:    switch s do
27:        case XPC_READY_STATE
28:            if m == XPC_VOTE_YES then
29:                r ← r + 1
30:            if m == XPC_VOTE_NO then
31:                s ← XPC_ABORT_STATE
32:        case XPC_COMMIT_STATE
33:            if m == XPC_HAVE_COMMITTED then
34:                r ← r + 1
```

Algorithm 4 2PC Participant

```
1: function STATE_TRANSITION(s, c, m)
2:     if s == XPC_INIT_STATE then
3:         s ← XPC_READY_STATE
4:     switch c do
5:         case XPC_CAN_COMMIT
6:             if s == XPC_READY_STATE then
7:                 if agree then
8:                     m ← XPC_VOTE_YES
9:                 if disagree then
10:                    m ← XPC_VOTE_NO
11:                s ← XPC_COMMIT_STATE
12:        case XPC_DO_COMMIT
13:            if s == XPC_COMMIT_STATE then
14:                Commit value
15:                m ← XPC_HAVE_COMMITTED
16:                s ← XPC_INIT_STATE
17:        case XPC_DO_ABORT
18:            Abort value
19:            s ← XPC_INIT_STATE
```

unresponsive after the maximum number of retransmissions is reached, the host times-out. Upon timeout the host switches to a `DO_ABORT` phase.

Building on top of XPC, initiators and participants will have custom implementations for the state and messaging processing functions. The logic behind state transitions and generated messages can be found in the Algorithm 3 and 4 code listings. Most notably the initiator switches to an abort state upon receiving the first `VOTE_NO` (Algorithm 3 lines 30-31) and participants do not need to acknowledge `DO_ABORT` messages (Algorithm 4 lines 17-19).

3.1.3.2 Three-Phase Commit

We present a reference implementation of 3PC written with XPC. Building on top of 2PC, Three-Phase commit aims to give additional guarantees on the correct termination of the protocol attempting to reduce the number of inconsistencies. The biggest change can be seen as the addition of "timeout commits". Rather than always aborting, if all nodes have acknowledged their intention to commit, but do not explicitly reply to a `DO_COMMIT` message, upon timing out the initiator will commit the value, hopefully granting the network higher resilience to communication failures.

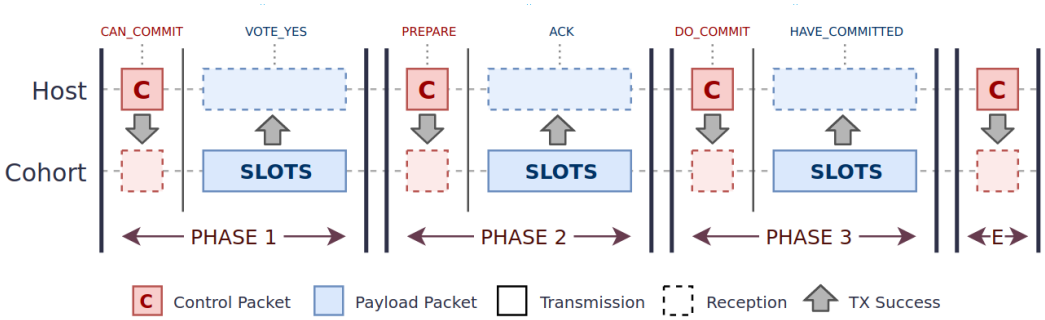


Figure 3.4: 3PC Protocol in Baloo with 1 retransmission.

Very similarly to Two-Phase commit, 3PC also maps to the Baloo paradigm as follows (Figure 3.4):

- **Phase 1**. Identical to 2PC.
- **Phase 2**. Upon receiving a `VOTE_YES` from all nodes, the host initiates the prepare stage with a `PREPARE` message which has to be `ACK`ed by all the nodes.
- **Phase 3**. When all nodes have `ACK`ed their commit preparations, the initiator sends a `DO_COMMIT` message which will be acknowledged by nodes with `HAVE_COMMITTED` replies. If the host does not receive `HAVE_COMMITTED` messages from all nodes it will timeout but identify the transaction as a commit regardless.

As outlined in the description of the different protocol phases, 3PC initiator (Algorithm 5) and participant (Algorithm 6) pseudo-code is very similar to that of 2PC. Both implementations simply introduce an additional stage between the `READY_STATE` and the `COMMIT_STATE` with its corresponding messages being exchanged.

3.1.3.3 Multi-Phase Protocol Properties and Guarantees

2PC and 3PC are implemented for XPC and may use both Glossy and Chaos ST primitives. All combinations are correct (i.e. nodes commit upon the whole cohort voting yes, and abort

Algorithm 5 3PC Initiator

```
1: function STATE_TRANSITION(s)
2:     switch s do
3:         case XPC_READY_STATE
4:             s ← XPC_PREPARE_STATE
5:         case XPC_PREPARE_STATE
6:             s ← XPC_COMMIT_STATE
7:         other cases same as 2PC
8:
9: function PREPARE_MESSAGE(s, m)
10:    switch s do
11:        case XPC_PREPARE_STATE
12:            m ← XPC_PRE_COMMIT
13:        other cases same as 2PC
14:
15: function PROCESS_MESSAGE(s, m, r)
16:    switch s do
17:        case XPC_PREPARE_STATE
18:            if m == XPC_ACK then
19:                r ← r + 1
20:        other cases same as 2PC
```

Algorithm 6 3PC Participant

```
1: function STATE_TRANSITION(s, c, m)
2:     if s == XPC_INIT_STATE then
3:         s ← XPC_READY_STATE
4:     switch c do
5:         case XPC_CAN_COMMIT
6:             if s == XPC_READY_STATE then
7:                 if agree then
8:                     m ← XPC_VOTE_YES
9:                 if disagree then
10:                    m ← XPC_VOTE_NO
11:                s ← XPC_PREPARE_STATE
12:        case XPC_PRE_COMMIT
13:            if s == XPC_PREPARE_STATE then
14:                m ← XPC_ACK
15:                s ← XPC_COMMIT_STATE
16:        other cases same as 2PC
```

when at least one node votes against) and have different reliability and latency guarantees. An important measure for reliability is transaction outcome: it analyses a situation where all nodes in the network always intended to commit, checking how often the protocol was able to terminate correctly committing the value. The aim is to match the results of the A^2 (§2.6) implementation: 100%.

In XPC's 2PC implementation, if the host has committed a value, all other nodes must have also committed the same value, though if the host as aborted a value, the cohort might be in an inconsistent state. We can formalise this behaviour with the following properties:

- **Liveliness**: 2PC is a blocking protocol, simply meaning that if a node stops replying no progress can be made. Within XPC, though, the protocol is forced to eventually timeout in such a case, and thus "complete". This way we are sacrificing safety for liveliness, ensuring the protocol always completes for all nodes, yet not necessarily maintaining consistency across the whole network.
- **Safety**: If the initiator commits then all nodes must have committed. If the initiator aborts, though, there might be nodes which have incorrectly committed. This can occur if the host sends out a `DO_COMMIT` request to all nodes, but never receives `HAVE_COMMITTED` replies from a few. In this case the host sends a `DO_ABORT` message, some nodes might have already committed and are prohibited by 2PC to switch to an abort, creating potential inconsistency in the network.

With 3PC the initiator is allowed to make a decision to commit a value regardless of cohort-nodes failing, meaning that to a much greater degree, if compared to 2PC, overall protocol liveliness is valued at the expense of safety. In 3PC if the host commits there could be failed nodes in the network that have never committed, and if the host aborts due to a timeout on the final stage (similarly to 2PC) the cohort could be in an inconsistent state.

In the next section we present 2PC and 3PC again to show how they perform with the different communication primitives provided by XPC.

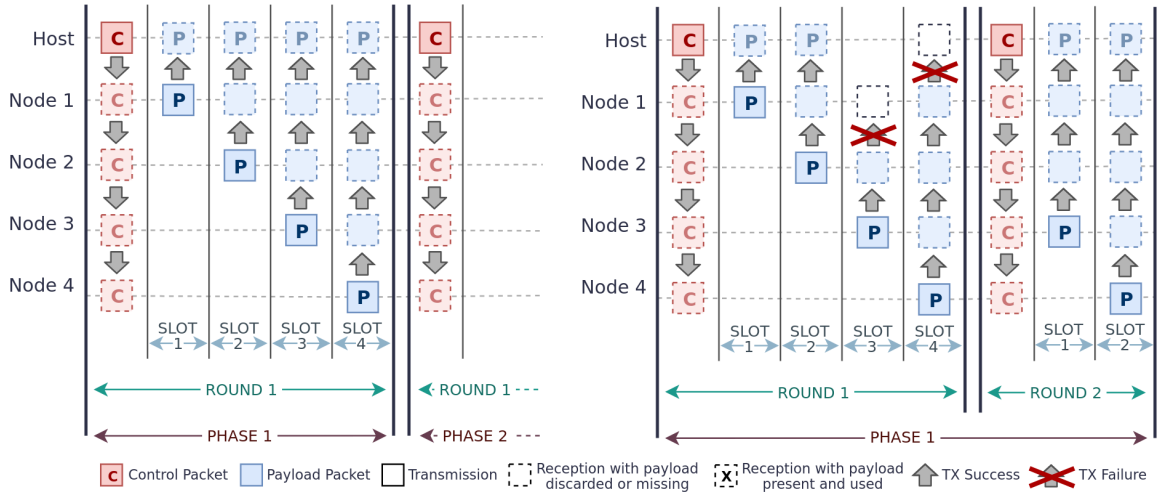
3.2 Hybrid: Fast and Reliable Synchronous Transmissions

No optimal, fast and reliable algorithm can be constructed with a vanilla use of ST primitives within Baloo. With XPC we have introduced the possibility to compare the latency and reliability of identical protocol implementations with different underlying ST primitives. Sections 3.2.1 and 3.2.2 analyse the issues (in terms of performance and robustness) which arise when executing multi-phase voting protocols with Glossy and Chaos.

In order to address **C1**, fixing the scalability, latency and reliability problems encountered by the two primitives individually, Section 3.2.3 introduces Hybrid. Hybrid is a completely new approach to Glossy-based flooding protocols which leverages the optimal Chaos latency and optimises it with Glossy's reliability. Hybrid takes full advantage of XPC's capabilities by switching among different ST primitives during the execution of a protocol phase. This approach was not possible before the introduction of Baloo, and was only enabled by the implementation of XPC.

3.2.1 Glossy Protocols

The simplest primitive to use with XPC protocols is Glossy (§2.3). XPC with Glossy uses a time-sliced data dissemination approach. Given a network of k nodes (where one is the Baloo host), each round the `schedule.n_slots` field is set to $k-1$, meaning that all nodes, which are not the global host, are scheduled to communicate in a given `schedule.slot` (see Figure 3.5a). All nodes will receive round information from the host within the control packet, and they will reply to the host by broadcasting during their allocated scheduled slot. The `payload` exchanged during each round contains the messages sent by each node as a reply to the host. In Glossy's case this is one byte of data (containing one of the messages determined by the `process_message` callback).



(a) Glossy round execution (b) Glossy retransmission round

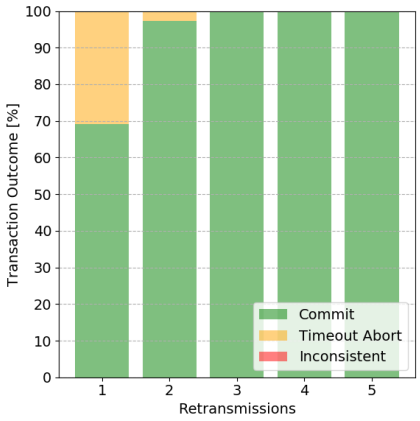
Figure 3.5: Execution of Glossy rounds with XPC. When nodes do not reply in a given slot they are scheduled to retransmit in the subsequent round during the same phase.

Regarding their retransmission policy, Glossy-based protocols simply keep track of nodes that do not reply in their scheduled slots and schedule them to retransmit in the next round. This retransmission round will have a shorter length, as replies would only be missing for a

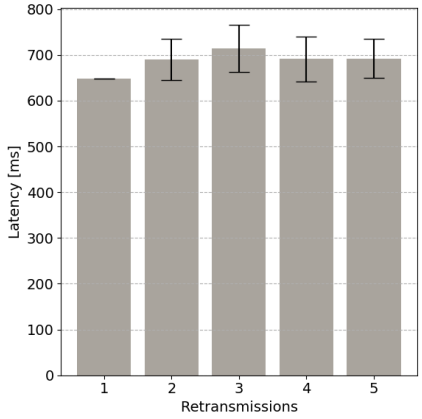
subset of the whole network. This is illustrated in Figure 3.5b, node 3 and node 4 were unable to successfully send their reply to the host node during the first round of transmissions. Therefore the retransmission round (i.e. round 2) contains only two entries in the `schedule.slot` section to specifically allow the nodes with missing replies to broadcast them.

3.2.1.1 Two-Phase Commit with Glossy

Two-Phase Commit is a simple protocol which benefits greatly from a time-sliced reliable dissemination primitive such as Glossy's. The protocol is unable to reliably reach all nodes in one round for each phase. The introduction of retransmissions greatly boosts the overall reliability, quickly reaching 100% (Figure 3.6a).



(a) 2PC-Glossy transaction outcome



(b) 2PC-Glossy retransmission latency

Figure 3.6: Evaluation of transaction outcome and latency for 2PC-Glossy in FlockLab.

Retransmissions don't have a heavy impact on the overall latency of the protocol (which, as outlined by Figure 3.6b, is below 1 second for the 27 nodes present in FlockLab). This is likely due to the protocol being forced to timeout due to missing replies from a couple of nodes. Rescheduling specifically these few nodes to send a reply in the next round (which therefore has a much shorter duration), solves the problem. It is important to note that the first communication round counts as a "retransmission", meaning that if only 1 retransmission round is scheduled the protocol will never attempt to communicate with nodes which have missed the first control packet.

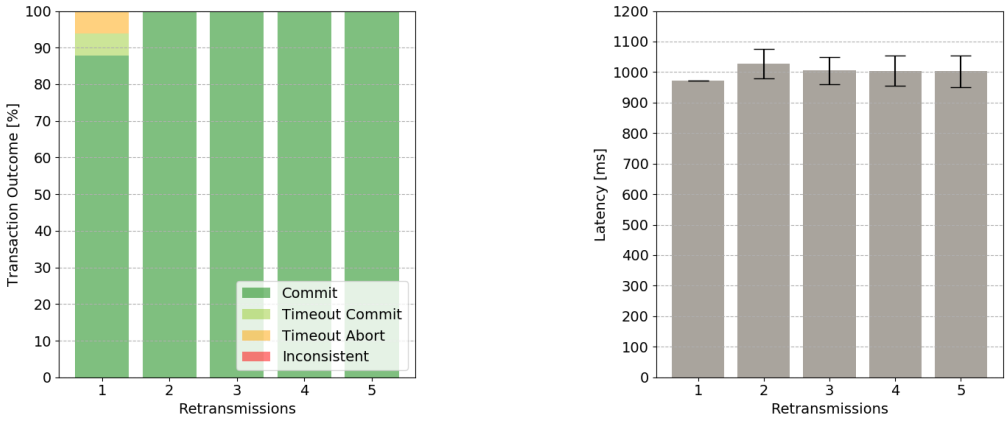
When increasing the number of retransmissions, the protocol is no longer ever inconsistent (i.e. when part of the nodes have committed the values, and part of the nodes have timed-out and aborted). This is because when timing out the global host must then send an abort message to all nodes in the subsequent round. Nodes are not required to acknowledge abort statements, and if the control packet is lost, a node might never be notified the value it has committed actually has to be aborted, therefore leading to an inconsistent state. By increasing the reliability of the protocol, timeouts occur less often, increasing the consistency of commits and aborts.

Sometimes Baloo makes a slight timing mistake when preempting the application to execute the middleware operations (such as listening for control packets). In these rare cases a participant's radio is turned on too late and the very first control packet for the first round can be missed. When using only one retransmission, if the first control packet is missed, the

full communication round will be missed by the node. This will cause the node to be unable to either commit or abort the proposed value (and thus causing the protocol to timeout).

3.2.1.2 Three-Phase Commit with Glossy

The introduction of "timeout commits" doesn't play a huge role in 3PC-Glossy's correctness. 3PC-Glossy (Figure 3.7a) has a very similar reliability if compared to 2PC-Glossy (Figure 3.6a). The most significant impact, though, can be seen in latency (Figure 3.7b). As expected 3PC has one more phase, and thus a higher latency if compared to 2PC when tested on FlockLab. This latency means that a Glossy-based consensus protocol would be very slow.



(a) 3PC-Glossy transaction outcome

(b) 3PC-Glossy retransmission latency

Figure 3.7: Evaluation of transaction outcome and latency for 3PC-Glossy in FlockLab.

3.2.2 Chaos Protocols

Chaos (§2.5) has different timing requirements to Glossy, and is harder to configure to match Baloo's (and thus XPC's) precisely timed and synchronous paradigm. Chaos is a ST primitive which terminates dissemination rounds once a reply has been heard from all nodes. Baloo requires a fixed execution time upper time bound for each round. Therefore, the `schedule.period` section of the control packet is set to a mutable value which allows nodes enough time to reply to the Chaos initiator. Missed nodes can be reached via retransmissions.

Only one `schedule.n_slots` is set, during which all nodes communicate simultaneously until the end of the round. This is due to Baloo's middleware, which schedules primitives for execution and terminates the execution upon primitive "completion". Glossy completes once it has flooded/broadcast its packet enough times, whereas Chaos completes once its packet has been acknowledged/seen by all nodes in the network. The `payload` attached to the packet is a bitmap where each node will aggregate the message it wishes to send back to the host. Votes are encoded as one bit for every node (i.e. technically two bits if we consider that each payload carries around a bitmap flag to keep track which nodes have cast their vote and which nodes are still missing).

If compared to Glossy, Chaos prioritises latency over reliability. As can be seen in Figure 3.8 Chaos floods occur in a "best-effort" fashion, where rounds should hopefully be long enough to reach all nodes in the network and aggregate replies from them.

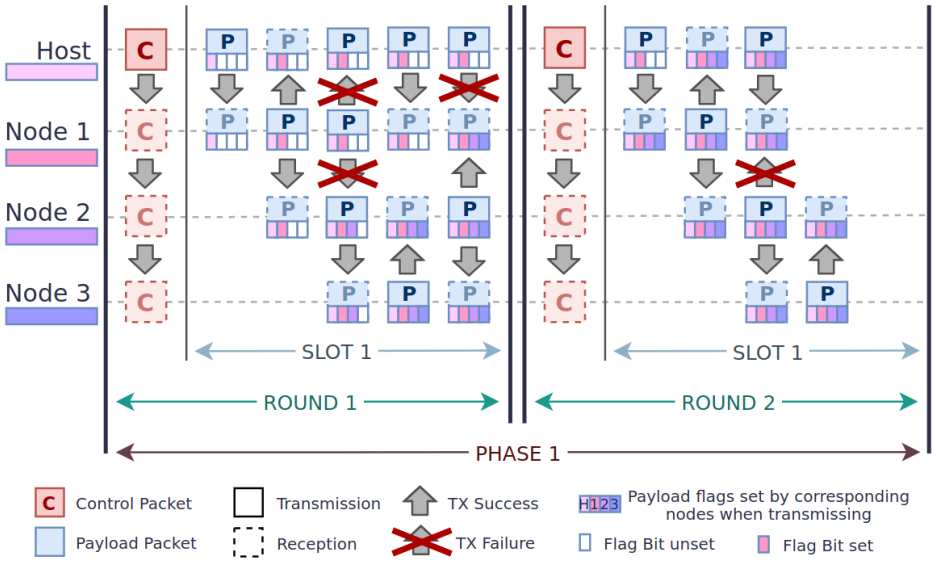


Figure 3.8: Execution of an XPC Chaos round with 1 retransmission. As nodes aggregate their vote into the payload they set their bits into the packet's flags bit-field.

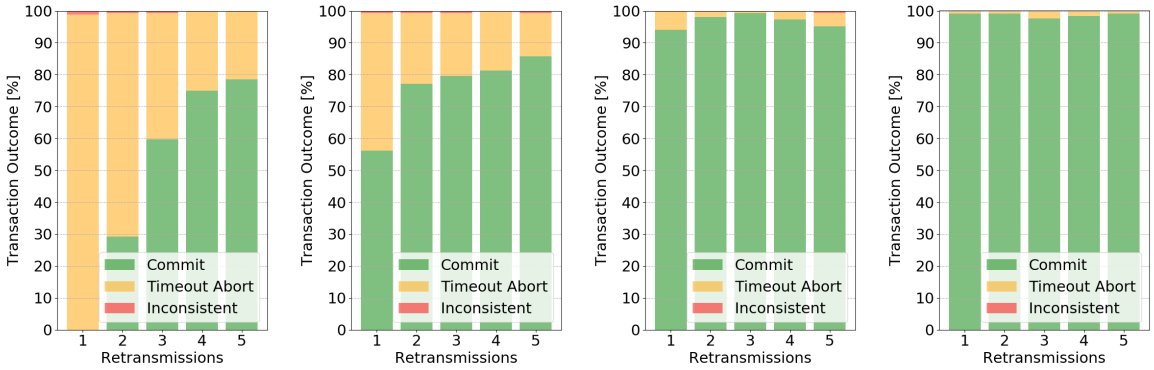
When requiring retransmissions due to missing replies, Chaos is unable to directly target a specific node in the network. It is also unaware of how many iterations the protocol will require for the missing packets to reach the host. The current policy within XPC is to schedule a new round of the same length instructing all nodes to keep using the `payload` of the previous round (Figure 3.8). The full payload is carried over to the new round and communication ceases when no node is seeing new information in the packets being broadcast (as occurs in Figure 3.8 after the fourth broadcast during the second round). This way the Chaos flood is resumed exactly from where it was last stopped by XPC, and it is given extra time so that it will terminate. This is not guaranteed.

3.2.2.1 Two-Phase Commit with Chaos

The biggest challenge encountered when using 2PC together with the Chaos ST primitive is allocating a correct timing value to the slot size. If compared to Glossy, Chaos has more of a local-gossip approach, where the message is constantly broadcast to all neighbours, which receive it, aggregate their own message into the payload, and rebroadcast it themselves. Hence two main problems can occur with varying slot sizes:

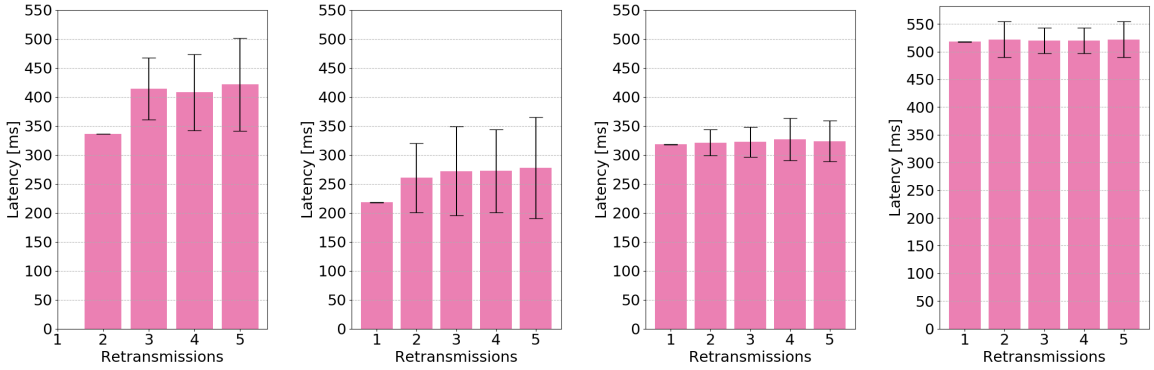
- **Under-allocation**. If the slot size is too small there is simply not enough time for the nodes the furthest away from the host to send their reply across, broadcast after broadcast. Differently from Glossy, there is no one-to-many communication and nodes need to rely on their one-hop neighbours to disseminate the message across. Even with varying levels of retransmissions, Chaos slots which are too small might not allow the host's message to be received by the whole network.
- **Over-allocation**. Bigger slot sizes end up having a big latency impact. Even though ideally one would schedule as big slots as possible, in practice it is best to try and minimise overall protocol execution time as much as possible. Faster protocols lead to less energy consumption in radio broadcasts.

Therefore the experimentation of 2PC-Chaos had to take into account two possible variables: number of retransmissions and Chaos slot length. Practical experimentation (see Figure 3.9) validated the assumptions: bigger slot lengths (i.e. 100ms or 200ms) can reliably get 100% reliability, very similarly to Glossy.



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.9: Consensus outcome of 2PC-Chaos with varying slot lengths in FlockLab.



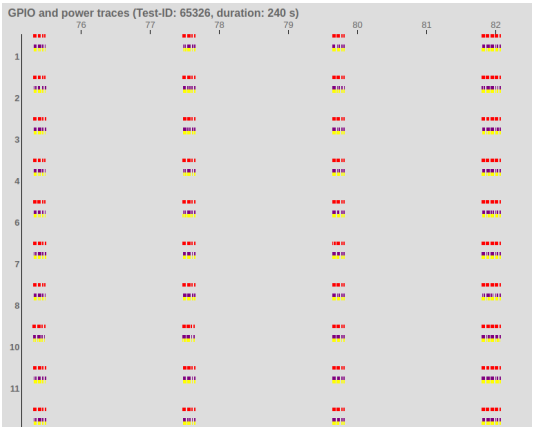
(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.10: Latency of 2PC-Chaos with varying slot lengths in FlockLab.

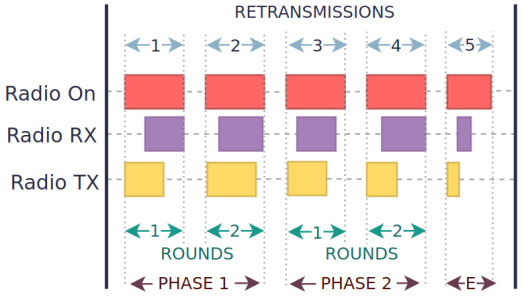
The biggest difference from 2PC-Glossy, though, is the overall latency of the protocol (Figure 3.10). Above 95% reliability can be achieved with 100ms long Chaos floods and overall 325ms latency. Close-to 100% reliability can instead be obtained with 200ms floods with a latency of 525ms (around 40% less than 2PC-Glossy when run on FlockLab). It can be observed that with longer Chaos flood lengths, retransmission numbers do not have a big impact on overall latency. This is likely due to the fact that retransmissions occur on very few occasions, as the protocol already starts off with a high reliability from slot 1.

The results show that 50ms slots (Figure 3.10b) have a lower latency than 25ms slots (Figure 3.10a). We can see the answer in the FlockLab GPIO (General Purpose Input/Output pins, in this case used as LED lights) traces (Figures 3.11 and 3.12). When executing tests on FlockLab the red LED is set whenever the radio is turned on, the purple LED is set upon reception of radio messages and the yellow LED is set whenever a radio message is broadcast. LED setting has little time overhead and increased accuracy (if compared to writing outputs to serial) and thus offers a much less disruptive way to analyse packet exchanges and

overall synchronization of the network during a protocol flood. Flocklab also offers a web visualiser that allows the inspection of these GPIO outputs. As Glossy based protocols keep synchronising the network, all of the logs will have matching timestamps across the various nodes in the network, and therefore can be compared.



(a) Flocklab Radio LED tracing

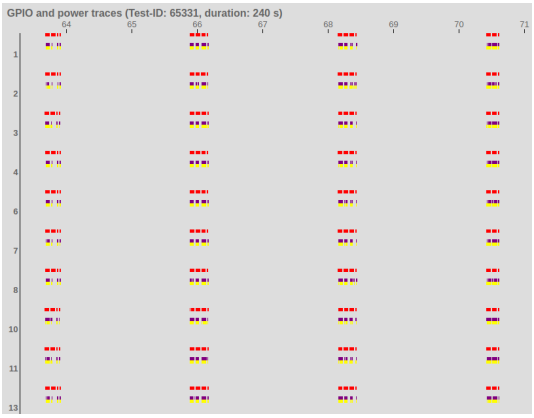


(b) Breakdown of Chaos retransmissions

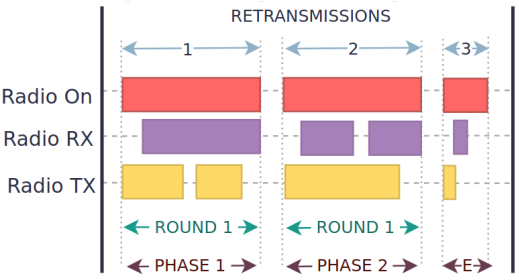
Figure 3.11: Breakdown of FlockLab Radio LED tracing for 2PC-Chaos with 25ms slots.

A snapshot of the execution of 2PC-Chaos with 25ms slots and 2 retransmissions can be seen in Figure 3.11a. This analysis is carried out over 2 retransmissions as the 1 retransmission implementation is never able to commit (see Figure 3.9a); in particular, of the 4 protocol executions present in the image (i.e. the four column of LED traces), only the right-most was able to commit (and can be seen enlarged in Figure 3.11b). All other executions ended up timing out and aborting due to missing replies.

From the breakdown we can see that a successful commit actually takes 5 retransmissions: 2 retransmissions for Phase 1, 2 retransmissions for Phase 2, and a final retransmission to communicate the end of the 2PC round to the network. When committing the protocol is always using 2 rounds for each communication phase, which is a sign of a too small communication slot: not all nodes can be easily reached, and retransmissions are thus constantly required.



(a) Flocklab Radio LED tracing



(b) Breakdown of Chaos retransmissions

Figure 3.12: Breakdown of FlockLab Radio LED tracing for 2PC-Chaos with 50ms slots.

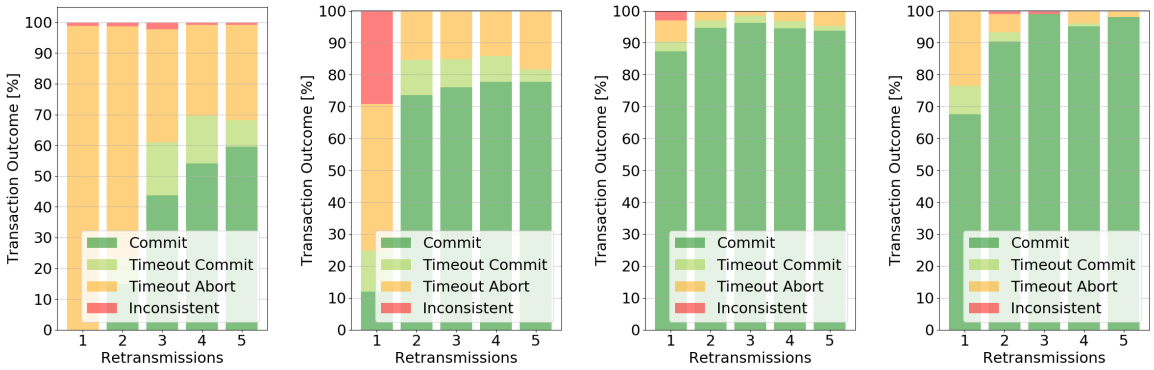
When analysing 2PC-Chaos with 50ms slots we see a completely different picture. In this

case the FlockLab GPIO trace (see Figure 3.12a) shows 3 successful commit runs (the left-most ones), and a timeout abort run (right-most). When looking into any of the successful commit executions (Figure 3.12b), it shows that with 50ms slots, Chaos tends to execute 3 rounds before committing (if compared to the 5 overall rounds with 25ms slots). And the overhead cost of scheduling additional rounds together with the in-between gap times (necessary to allow callback functions to execute and to prevent network traffic clashes), is actually outweighed by having slightly longer slot times. The overall impact of having to send additional control packets to schedule more rounds means that 1 retransmission 2PC-chaos with 100ms slots has higher reliability and slightly lower latency that any number of retransmissions with 25ms slots.

3.2.2.2 Three-Phase Commit with Chaos

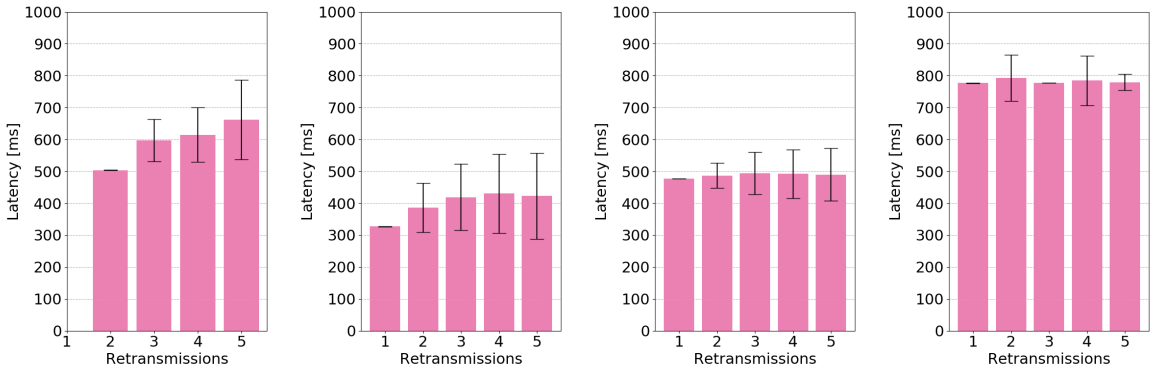
Overall 3PC-Chaos suffers from the same reliability and latency concerns as 3PC-Glossy. From a reliability perspective (Figure 3.13), 3PC performs worse than its two-phase counterpart even once we consider the "timeout commit" feature. This is likely due to additional Chaos rounds being more highly impacted by stragglers that compromise the full protocol run. Similarly to 2PC-Chaos, though, close to 100% reliability can be achieved with 200ms transmission slots.

As expected, latency is affected by the introduction of an additional communication round.



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.13: Consensus outcome of 3PC-Chaos with varying slot lengths in FlockLab.



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.14: Latency of 3PC-Chaos with varying slot lengths in FlockLab.

Overall latencies, though, remain much lower than their counterpart when using Glossy (Figure 3.14). Chaos is therefore a viable ST primitive that helps minimise the overall protocol execution time, yet unfortunately it simply lacks the sufficient reliability to be a robust long-term voting solution, or the basis for the implementation of a consensus protocol for low-power multi-hop networks.

3.2.3 Hybrid

In this section we introduce a novel communication architecture that solves the latency issues of Glossy and the reliability issues of Chaos. Having examined Glossy and Chaos implementations of two and three-phase commit protocols within XPC, it can be said that both approaches have a number of advantages, but also quite a few drawbacks.

- **Glossy**-based approaches are incredibly reliable across retransmissions, yet latency suffers greatly when scaled across large networks. This is because inherently every single node within the network has to be have its own communication slot scheduled in order to reply back to XPC's host node. Furthermore the size of the control packet grows linearly with the size of the network. Within the `schedule.slots` there needs to be room to allocate at least every node once. Glossy-based protocols can be visualised as a "serial" execution of tasks, where nodes are requested to transmit in an orderly-fashion one after the other.
- **Chaos**-based algorithms have much better latency, yet they suffer from reliability. It is very hard to guarantee that all nodes within the network will be able to reply during a round. Nodes cannot be reached individually and retransmission rounds therefore need to have much greater lengths. Furthermore the size of the `payload` exchanged by the nodes during each round grows linearly with the size of the network, as it needs to contain one bit of information from each node. Chaos-based protocols can be visualised as a completely "paralllel" execution of tasks, where all nodes keep trying to transmit to all others until the protocol is terminated or all nodes have received replies from all participants in the network.

Hybrid introduces a novel multi-ST primitive retransmission policy that leverages Baloo's middleware support for fully configurable flooding rounds. The first retransmission of each phase is executed using a Chaos flood (see Figure 3.15). Based on the flood's outcome, starting from the second retransmission, Glossy is used. This allows the host to receive most replies during the first Chaos round and thereafter uses Glossy to quickly retrieve information from all straggler nodes which were unable to broadcast during the first slot. Up to a maximum of

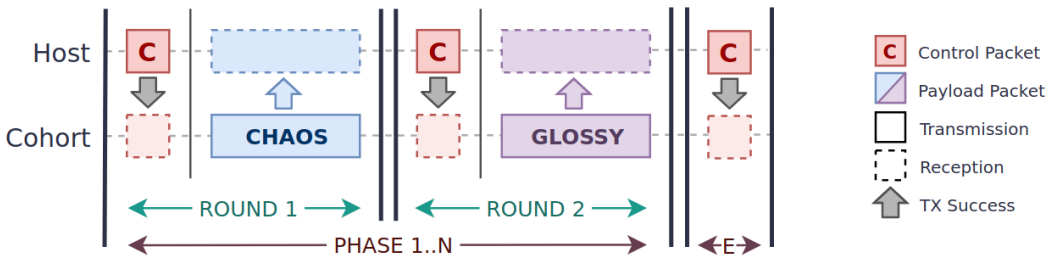


Figure 3.15: Overview of Hybrid execution across multiple phases. Each phase starts with a Chaos dissemination round, followed by a variable number of glossy floods.

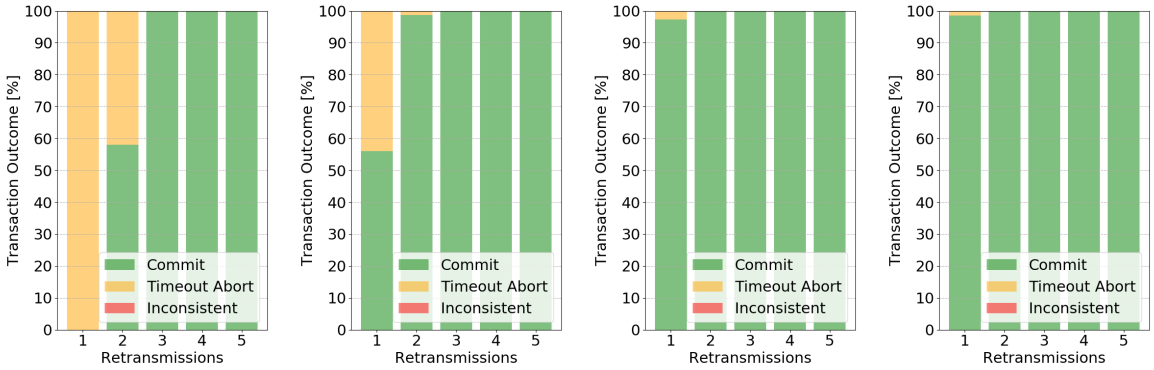
`XPC_TIMEOUT_RETRANSMISSIONS` − 1 Glossy floods may be scheduled, after which, if still no reply is heard from every node, the protocol times-out.

Hybrid solves all reliability problems experienced by Chaos when forced to fit a specific timing model such as Baloo's. It is also able to achieve close-to-Chaos latency, overall providing a completely new, fast and reliable means of executing wireless communication. Hybrid offers a practical solution to the scalability of ST primitives: regardless of network density and regardless of number of nodes it enables the creation of quick, efficient and reliable protocols that reach every single available mote. Hybrid therefore meets and solves **C1** presented in Section 2.10.

3.2.3.1 Hybrid Two-Phase Commit

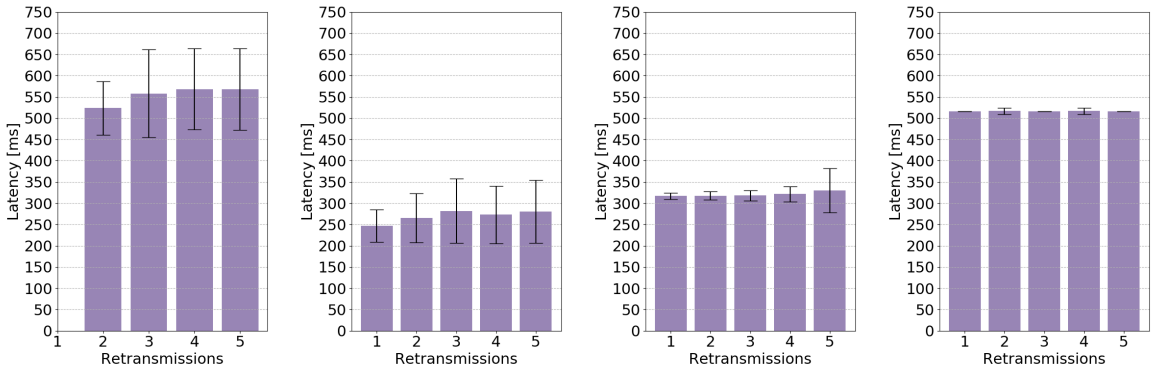
As XPC offers a complete separation of all ST primitive code from the protocol implementation, "hybrid" based approaches only differ in the parameters set within the project's configuration file. It is therefore important to configure the two individual ST primitives correctly to ensure the lowest possible latency with the highest reliability.

The protocol could, in fact, suffer from both drawbacks of Glossy and Chaos. If the initial Chaos slot does not communicate with enough nodes then too many replies will be required during the subsequent Glossy rounds, severely impacting the overall protocol latency. If the



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.16: Consensus outcome of 2PC-Hybrid with varying slot lengths in FlockLab.



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.17: Latency of 2PC-Hybrid with varying slot lengths in FlockLab.

initial Chaos round is too large we likely might be wasting time waiting for pending replies from very few nodes; probably a shortened Chaos slot, and subsequent Glossy retransmissions, would have retrieved the replies from these stragglers faster.

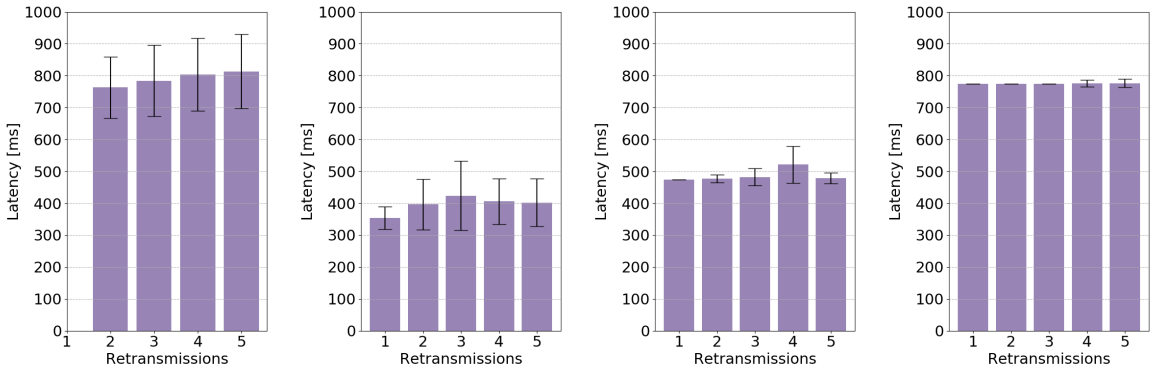
Hence when testing Hybrid, in order to guarantee an as-close-as-possible performance to Chaos, we tested it using the same Chaos slot sizes. These slot sizes are only used to determine the length of the first round of each protocol phase, after which Glossy uses its own, fixed, slot lengths. What we can see when analysing Figure 3.16 is that, just as expected, the Glossy retransmissions are able to reliably bump up the protocol reliability to 100%. The latency (Figure 3.17) is also minimally affected (if compared to 2PC-Chaos). Being able to converge this quickly to 100% correct transaction outcome allows nodes to transmit minimal number of messages: fewer messages imply big energy savings and it is always important to ensure that motes save on using as many communication rounds as possible to ensure maximal energy efficiency.

3.2.3.2 Hybrid Three-Phase Commit

Due to its additional third phase, 3PC has already proven itself to slightly fall short on reliability (if compared to its 2PC counterpart) both with Glossy and Chaos implementations. Unsurprisingly our 3PC-Hybrid implementation does also require additional retransmissions

(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.18: Consensus outcome of 3PC-Hybrid with varying slot lengths in FlockLab.



(a) 25ms slots (b) 50ms slots (c) 100ms slots (d) 200ms slots

Figure 3.19: Latency of 3PC-Hybrid with varying slot lengths in FlockLab.

to reach 100% reliability (Figure 3.18).

Latency (see Figure 3.19) is also once again comparable with 3PC-Chaos, offering a much needed speedup over the Glossy implementation. Overall this data highlights how our novel Hybrid approach to scheduling ST primitives can potentially generate a whole new family of fast reliable voting protocols built on top of XPC and Baloo.

3.3 Next Steps

Chapter 3 has introduced Hybrid, a novel, efficient and reliable ST primitive. The implementation of Hybrid relies on the introduction XPC, a new library which simplifies the use of different ST primitives for the creation of multi-phase voting protocols built on top of Baloo.

Our analysis began with XPC. Section 3.1 outlines its 3 fundamental aspects: protocol implementation, common code and ST primitive interface. Similarly to Baloo, XPC has split initiator and network-participant sections in the code. We have analysed the difference in the callback structure and state-machines of the two different kinds of nodes, providing examples of how two-phase and three-phase commit protocols can be translated to XPC's time-sliced paradigm.

Two main protocol families stem out of XPC, differing mainly on the underlying ST primitive used. Section 3.2.1 analyses Glossy-based protocols and their linear time-sliced approach to ensuring reliability across retransmissions. Section 3.2.2 investigates Chaos showing how it is still possible to have close-to 100% transaction outcome with much faster, yet less orderly, local-gossip flooding and dissemination. There are though problems in both approaches either in terms of latency or robustness.

The analysis concludes in Section 3.2.3 with our first contribution: Hybrid. Hybrid is introduced as a novel multi-ST primitive protocol which is able to leverage the fast flooding of Chaos and the high reliability of Glossy. With the implementation of Hybrid and XPC we adress and solve **C1** and **C2**.

Evaluation of Hybrid-based protocols, together with comparison with state-of-the-art implementations, can be found in Section 6. Yielding very promising results the next step is to build a new family of voting protocols which can fully utilise the scalability and latency benefits of this new hybrid use of ST primitives: the Wireless Part-time Parliament.

4 | WISP and the Wireless Part-time Parliament

This chapter introduces WISP, a Paxos implementation for XPC based on the Hybrid ST primitive. To support Paxos-based protocols we present WiPP, the first Wireless Part-time Parliament, an extension of the XPC library tailored for majority-based protocols and applications. To the best of our knowledge there is currently no other library or project in literature which supports this level of voting configurability, while guaranteeing reliable dissemination and Chaos-like latency.

In 2019 WPaxos [32] introduced the first majority-based consensus primitive based on top of Chaos to the realm of WSNs. Borrowing many concepts from Leslie Lamport's Paxos [24], today's most used consensus protocol, WPaxos allows a proposer to globally disseminate a value after a majority of nodes have accepted to commit it. Implemented using A^2, WPaxos relies on Chaos floods to disseminate values and is unable to reliably reach the whole network in the presence of node failures or interference. WISP uses the Wireless Part-time Parliament, together with Hybrid dissemination rounds, to match, and better, the latency and reliability of WPaxos, while providing the same safety guarantees and consensus properties. By using WiPP and Hybrid, the implementation of WISP addresses **C3**.

Our discussion begins in Section 4.1 with an overview of the Wireless Part-time Parliament's internal layered structure. We analyse the implementation of majority voting and global dissemination, which are necessary to support consensus protocols. Section 4.2 then introduces WISP, a new Paxos-based protocol implemented within XPC's stack.

4.1 WiPP fundamentals

The Wireless Part-Time Parliament is a new XPC-based protocol which enables the network to execute majority voting. Rather than requiring a reply from each member of the cohort (operation executed by universal-voting protocols such as 2PC and 3PC) majority-voting rounds are allowed to proceed to the next phase once $\lfloor 50\% + 1 \rfloor$ of the cohort nodes have replied. Wireless multi-hop networks benefit greatly from majority-based protocols in the following ways:

1. **Resiliency to failures**. The failure of a single node in a standard 2PC implementation causes the whole protocol to block and be unable to make progress. A majority-based protocol with a quorum of $\lfloor 50\% + 1 \rfloor$ could not only make progress with up to f node failures, but it would also constantly be able to guarantee consistency. When querying the network for a protocol's result there will always be a majority of (alive) nodes which would agree on what values have been agreed upon and committed.

2. **Lower latencies**. When implementing protocols for wireless multi-hop networks, latency is important. The sooner nodes are able to complete a protocol round, the quicker they are able to resume their sensing operations. Protocols should always aim to have minimal impact in the duty-cycling of an individual mote.
3. **Fewer exchanged packets**. A final consideration is energy efficiency. The higher the number of messages which keep being exchanged and flooded by the network, the more energy will be consumed. WSN motes are bound by a limited battery supply which often determines their overall lifetime; the fewer times a node needs to use its radio, the longer it will be able to participate in the network.

4.1.1 WiPP alongside XPC

The Wireless Part-time Parliament has been implemented to complement XPC's layered structure and extend it with new functionality. By introducing WiPP we bring three new additions to XPC (Figure 4.1), all these features being backwards-compatible with existing protocols and application implementations:

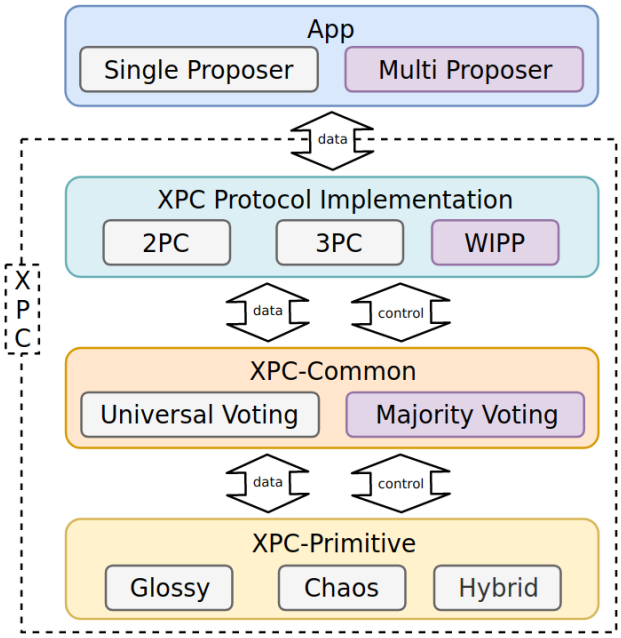


Figure 4.1: Overview of the contributions introduced by WiPP to the XPC library. WiPP adds optional majority round voting and multiple proposing nodes.

- **Multiple Proposers**. The applications using XPC may now initiate round proposals from any node in the network. When using the single proposer option only the XPC host is able to propose values at each protocol run. With multi-initiation any arbitrary number of nodes may propose values and all values will be voted sequentially. The multi proposer feature is transparent to the protocol layer, meaning that no state-machine or code alteration needs to be performed in order to enable it. A more in depth analysis on multiple proposers can be found in Section 5.1.
- **Majority Voting**. Shifting from 2PC's and 3PC's universal voting strategy (where a reply must be heard from all nodes in the cohort in order to proceed), majority voting

allows a round to complete when a quorum of nodes have given the same reply. As nodes are only allowed to reply in a binary yes/no fashion this guarantees our rounds will eventually terminate.

- **WiPP Protocol**. The Wireless Part-time Parliament is offered as a protocol which can be used by Contiki applications, similarly to two and three-phase commit. Being built within XPC's stack it can be configured to use any ST primitive and proposer methodology.

4.1.2 Global Dissemination

WiPP features reliable global dissemination of committed values, additionally to offering a fast and flexible solution to majority-style voting on wireless multi-hop networks. A hurdle blocking the wide-spread adoption of majority protocols in the WSN community is the lack of a globally-coherent state upon completion. All nodes which are not part of the quorum may be potentially unaware of the committed value, and ignore the protocol round entirely. This is not a problem for wired networks. Each node will simply attempt to communicate with the cohort and eventually a majority of nodes will provide the committed value. Such an operation is incredibly energy-expensive for a wireless low-power node, and does not fit with Baloo's synchronous approach.

WiPP introduces a global dissemination round at the end of its execution in order to minimise individual mote power usage and to mitigate the problem of having nodes autonomously decide whether or not to query the network as whole to determine the outcome of a transaction. This approach is not new to the WSN field, as WPaxos [32] also uses a global Chaos flood of committed values to ensure the whole network constantly shares the same common state. With the addition of the global dissemination all nodes will always be aware of the outcome of each voting round for each proposed value. This strong guarantee allows WiPP to not only be used for quick majority-based voting rounds, but also for reliable distribution of the transaction outcomes.

4.1.3 Majority voting for ST primitives

We analyse the behaviour of the underlying ST primitives (and XPC) to determine any additional adjustments which must be made when transitioning from universal-voting to majority-based voting.

- **Chaos** requires no adjustments. Chaos floods see no difference during their execution as there is no method to deterministically predict how many nodes will be contacted and will reply during a round. What changes is the "end" condition for a Chaos round as it now requires only $\lfloor 50\% + 1 \rfloor$ of the nodes to have performed the same operation (i.e. either a commit or an abort) before switching to the next protocol phase.
- **Glossy** requires further engineering. Due to its time-slotted nature, the schedule at the beginning of a Glossy round has to carry information regarding which, and how many nodes will be broadcasting their replies. There is a problem; at the beginning of a Glossy round the protocol is unsure of how many nodes should be scheduled to broadcast. This is because only a quorum of the nodes is required to reply with the same message, and it is not necessary to hear back from every node in the cohort. We refer to this problem as the Retransmission Cohort Problem (RCP).

With Glossy the Retransmission Cohort Problem can be expressed in terms of round allocation size. This problem can be analysed in the case of a simple protocol which proposes a value and proceeds to the next stage based upon a majority-reply from the network. Each node is allowed to reply in favour (`VOTE_YES`) or against (`VOTE_NO`) the proposed value. Once the host has received a majority of replies for either message it proceeds to the next stage. RCP consists in knowing what is the minimal number of nodes the host should schedule to reply within a round to reach a quorum of replies in the least amount of time and most energy-efficient way possible.

A simple example of RCP can be theoretically analysed for a network the same size as Flocklab's with 27 nodes and a quorum set at 14 votes. At the very first round a conservative scheduler might decide to randomly pick 14 nodes in the network and ask for their vote against a proposed value. Assuming each node has a 70% chance of agreeing with a value proposed by the host (hence issuing a `VOTE_YES` reply, and a 30% chance of replying with a `VOTE_NO`), out of the 14 expected replies, the host will, on average, receive 10 in favour and 4 against. In this average case the quorum is not reached and a new communication round has to occur (which will cause a higher energy cost for the nodes and an overall higher protocol latency).

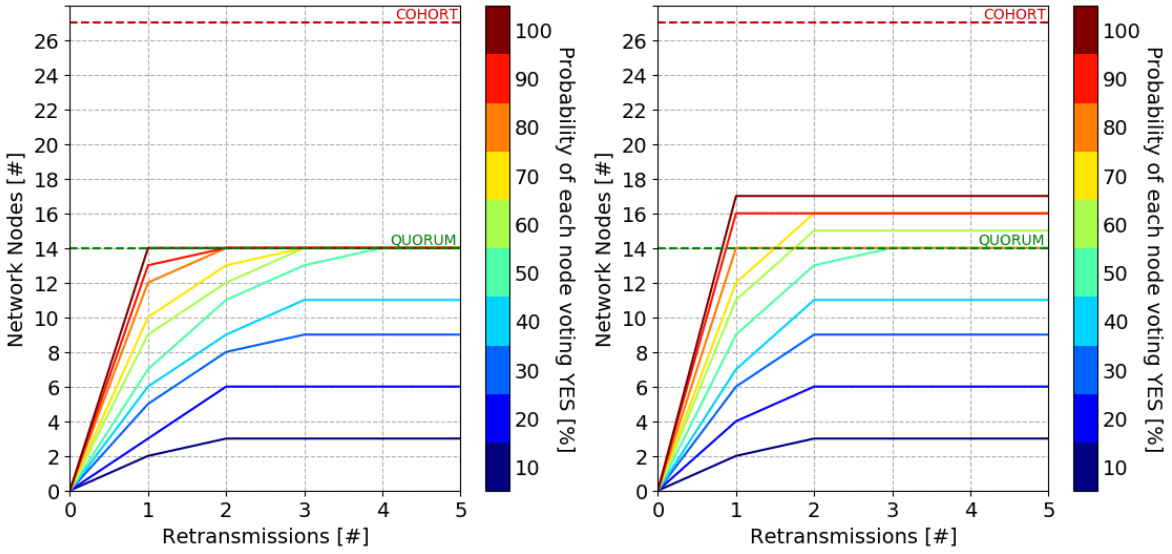
The scheduler now determines how many additional nodes should be contacted in the second round, given that it already has received a specific number of replies in favor and against. As the quorum is set at 14 nodes, an attempt to minimise the number of nodes to ask causes the host to schedule 4 random nodes (of the remaining 13) to reply (as 10 replies in favor were received). With a 70% `VOTE_YES` rate such round is likely to only receive 3 replies in favor on average, forcing the scheduler to execute another additional round (following the law of diminishing returns). A conservative scheduler can therefore risk to keep scheduling additional rounds with a decreasing numbers of scheduled nodes per round. This will increase protocol latency as a new control packet has to be sent out each time.

Algorithm 7 Determines number of `n_slots` scheduled in the next Glossy round

```
1:  function XPC_PROCESS_MAJORITY_VOTE(votes_yes, votes_no)
2:      max_n ←MIN(votes_yes, votes_no)
3:      if votes_yes > QUORUM then
4:          Proceed to next state and return
5:      else if votes_yes > QUORUM then
6:          Switch to abort state and return
7:      else
8:          n_slots ← (QUORUM − max_n) + ΔQ
9:          n_slots ←MIN(XPC_COHORT_NODES − (votes_yes + votes_no), n_slots)
10:         Schedule a new majority round with n_slots scheduled nodes
```

The Wireless Part-time Parliament solves the Retransmission Cohort Problem by introducing ΔQ, a configurable parameter used when scheduling replies from nodes during a Glossy-based round. When determining the `n_slots` parameter of an XPC majority round (see Algorithm 7) the protocol will normally attempt to use the minimal number of nodes that can be scheduled in the optimistic view of receiving the same reply from every contacted node. By setting ΔQ to a non-zero value, additional nodes will be scheduled during the next round. To ensure that nodes are not queried redundantly for replies, the algorithm stops once it either reaches the quorum or each of the nodes in the cohort has replied (and therefore can make no further progress).

The introduction of ΔQ to solve the Retransmission Cohort Problem offers a trade-off



(a) $\Delta Q = 0$ (b) $\Delta Q = 3$

Figure 4.2: Comparison of expected number of replies across retransmissions for Glossy floods on FlockLab with varying probability of nodes committing.

between single-round latency and overall number of retransmissions. A comparison simulation, based on numerical analysis, is shown in Figures 4.2a and 4.2b. We analyse the impact of using a ΔQ of 0 or 3 in the case where nodes in the network have varying probabilities of issuing a `VOTE_YES` reply.

Figure 4.2 visualises the impact a larger ΔQ has in speeding up the convergence towards the network quorum ($\lfloor 50\% + 1 \rfloor$ of the 27 nodes) by reducing the number of required retransmissions. If the network has an overall high `VOTE_YES` rate it does mean we are likely to overshoot the quorum (as occurs in the 100%-90% plots in Figure 4.2b), yet overall it reduces the amount of retransmissions required for potentially unstable networks (80%-50% plots). Each retransmission has a fixed cost as it needs to reliably flood a new schedule. A focal point of the analysis of protocols using the Wireless Part-time Parliament is to determine what impact varying amounts of ΔQ have on the overall protocol latency.

4.2 WISP: WiPP Simple Paxos

An efficient implementation of Paxos was introduced to low-power wireless networks by WPaxos in March 2019 [32]. It mapped the algorithm to Synchrotron's continuous Chaos flooding mechanism, allowing all nodes in the network to be hypothetical proposers during a WPaxos commit round (§ 2.7). Relying exclusively on Chaos floods, WPaxos is unable to reliably reach the whole network in the presence of node failures or interference as single nodes cannot be individually scheduled to reply. With our newly introduced Wireless Part-time Parliament within XPC's Baloo-based stack we introduce WiPP Simple Paxos (WISP for short), a new take on consensus-based commit protocols. WISP is able to match, and better, the latency and reliability of WPaxos, extending it's functionality by including complete network-wide voting capabilities.

Lamport's Paxos is a two-phase protocol (§ 2.2.3). During the first phase (prepare-promise) proposers send ballot numbers to all acceptors searching for the highest proposal number

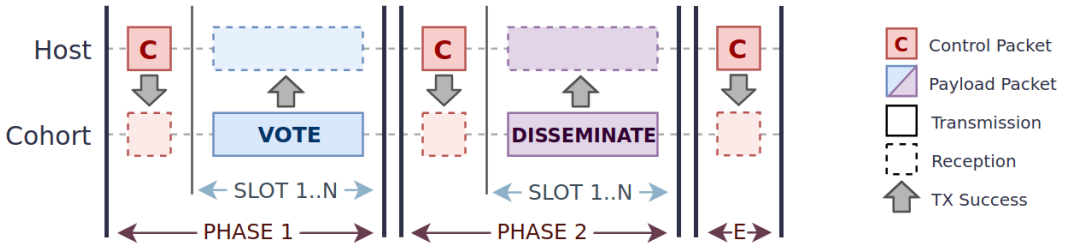


Figure 4.3: Overview of the two phase structure of a WISP proposal

(PN). When a proposer receives acknowledgement that its PN is higher than any other PN it switches to phase two (accept-accepted) where it tries to make a majority of nodes commit it's proposed value (PV). WPaxos adopts this approach and adds a third phase, a global dissemination phase of the proposed value, at the end. WISP leverages the presence of the global Baloo host to simplify WPaxos' phase structure to maintain two phases (Figure 4.3):

1. **Voting phase**. Baloo requires a host to be present at all times in the network. The host is in charge of generating every round's control packets and guarantees the synchronous timing structure of the middleware. WISP leverages the presence of the global host to propose all values, treating it as a global leader. As all values are proposed by a single node there is no need to use proposal numbers during rounds: the presence of the leader guarantees that no more than a single value will be proposed at once, and additionally provides a global ordering to all values proposed to the network. Needing no PNs, WISP removes the prepare-promise phase of Paxos. Additionally the second phase is modified such that nodes can vote on each value in a binary yes/no fashion, rather than being only able to accept (and thus commit) the proposed values. The protocol only requires a majority of nodes to agree before continuing to the next phase.
2. **Dissemination phase**. The host initiates a second phase to disseminate the transaction's result in order to guarantee that all nodes in the network are aware of the outcome of the voting phase. Dissemination is reliable and must be acknowledged by all network nodes.

The novelty of WISP is in its majority-based voting. In WPaxos nodes are only able to reject proposal numbers that are not monotonically increasing with the highest currently stored PN. With WISP participants can vote on the actual transaction content, potentially rejecting proposals they do not wish to see committed.

Even though WISP removes the prepare-promise phase, it is still a Paxos algorithm as it maintains the safety and liveliness properties described in Section 2.2. Termination, agreement, integrity and validity are all satisfied due to the presence of the global leader which guarantees a global ordering for all proposed values. Due to the global dissemination round all transactions will be locally ordered on each individual node. We therefore maintain the correctness of Paxos, providing an efficient implementation for low-power multi-hop networks.

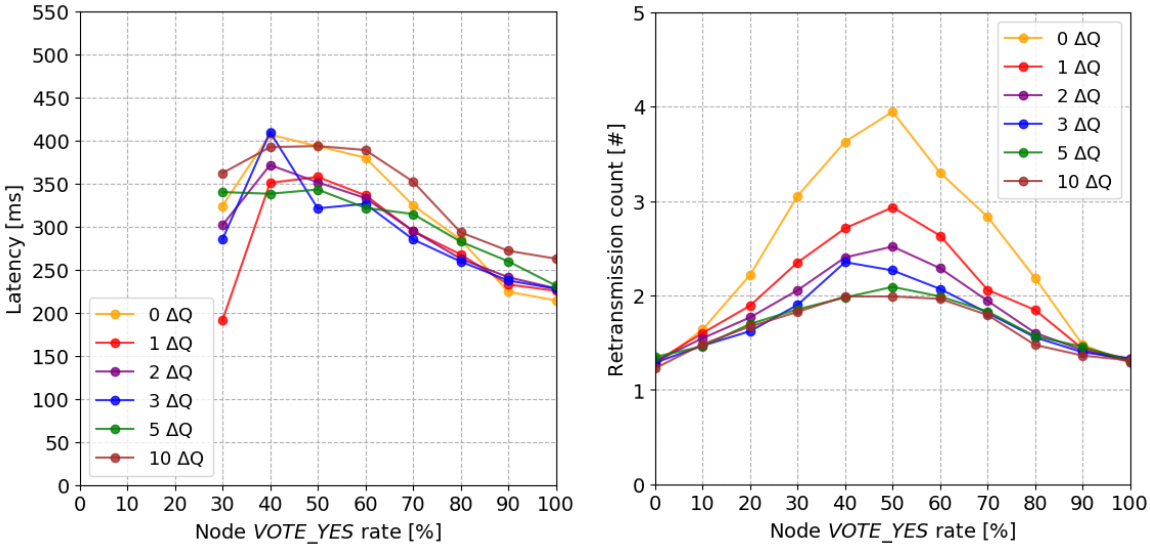
Our WISP implementation is a vanilla adaptation of the Wireless Part-time Parliament. It leverages the WiPP implementation written for XPC together with the single proposer feature and hybrid ST primitive (§ 3.2.3). This allows us to have efficient and quick majority voting rounds (with Chaos), reliably reaching all stragglers with the Glossy retransmissions.

4.2.1 ΔQ and M-Slot parameters

WISP is a Wireless Part-time Parliament protocol built especially for the Hybrid ST primitive. Glossy or Chaos-only approaches are supported and configurable, but lack robustness, reliability and low latencies. The voting and the dissemination round commence with a Chaos flood and then continue with Glossy rounds. The majority round can modify both Chaos and Glossy to guarantee protocol speedups and faster convergence towards the quorum. WISP focuses mainly on two parameters:

- **ΔQ size**. Introduced and analysed theoretically in Section 4.1.3, ΔQ determines the number of additional nodes scheduled for retransmission. Large ΔQ values ensure faster convergence during Glossy floods, yet will overall impact the protocol's latency.
- **M-Slot size**. Chaos rounds were originally designed to reach all nodes in the network. Majority voting, instead, has to only reach a quorum. Majority Chaos rounds can be configured to have an arbitrary "M-Slot" length, rather than sticking to the default 25, 50 or 100ms sizes for Chaos slots. Large M-Slot values will grant no latency benefit, but it is possible to find a local point of minima where a given M-Slot size reaches enough nodes to guarantee replies from the quorum.

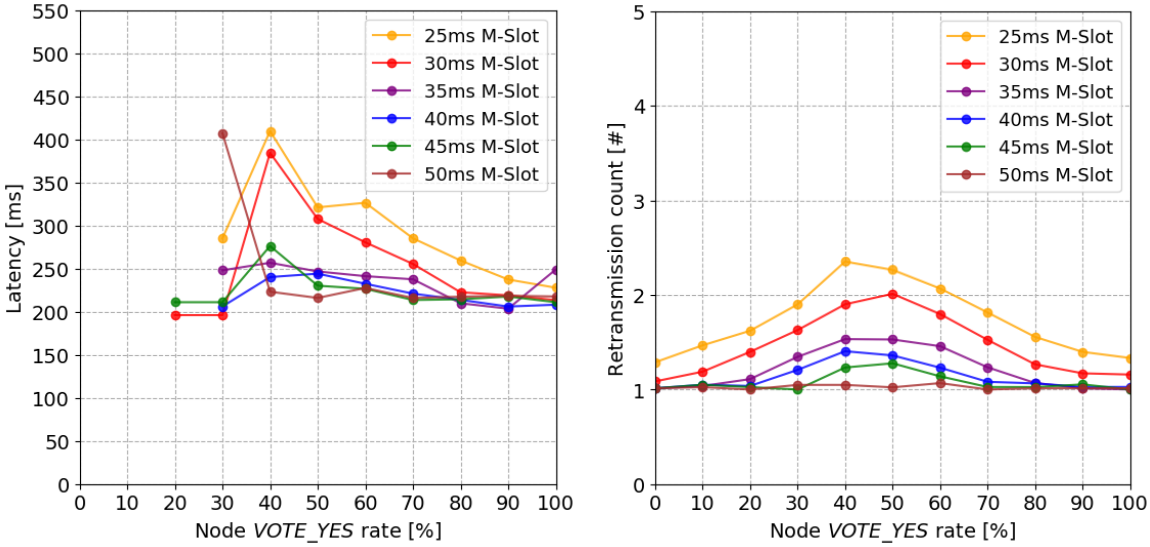
Determining the optimal values for ΔQ and M-Slot is dependent on the network node density and overall width. We apply the theoretical analysis from Section 4.1.3 to show the effects of ΔQ and M-Slot in an empirical example. Figures 4.4 and 4.5 show the overall impact different values of ΔQ and M-Slot have on protocol latency and the number of majority round retransmissions. As we increase the size of ΔQ, the number of retransmissions required for protocol convergence during the majority round decreases. This is also seen for the most uncertain network conditions where only 50% of the nodes are likely to vote in favour of a proposed value. Latency, though, lies at an optimal point between too little (i.e. 0) and too large (i.e. 10) values of ΔQ.



(a) Average commit latency

(b) Average number of retransmissions

Figure 4.4: Evaluation of WISP with different ΔQ values and 25ms M-Slot size on FlockLab.



(a) Average commit latency

(b) Average number of retransmissions

Figure 4.5: Evaluation of WISP with different M-Slot lengths with ΔQ of 3 on FlockLab.

Figures 4.5a and 4.5b show the latency and retransmission count for different values of `VOTE_YES` rates. The `VOTE_YES` rate is the probability that a node will vote yes for each proposed value. The probabilities are uniformly distributed and independent, and each point is the average of 100 runs. The `VOTE_YES` rate was introduced to mimic hypothetical network scenarios where varying amounts of nodes wish to commit the values flooded by the global host.

For probabilities greater than 50% the network is quite stable and is likely to commit all values. Below the 50% threshold the data for commit latency becomes really unstable because fewer data-points are available. An empirical analysis of results shows that in the case where nodes have a 30% `VOTE_YES` rate, the network is able to commit in less than 5% of protocol runs. With a 20% `VOTE_YES` rate, commit rates are below 1%. Intuitively this occurs because with `VOTE_YES` rates above 50%, on average each round, above 50% of the replies from nodes will be yes, meaning that the quorum is constantly reached. With a 30% `VOTE_YES` rate on average only 8 of the 27 Flocklab nodes would wish to commit a proposed value. It is thus incredibly unlikely for the network to reach the 14-node quorum.

Varying both ΔQ and M-Slot lengths generates bell-shaped curves for retransmission counts (Figures 4.4b and 4.5b). The further away the network is from abort (0% `VOTE_YES` rate) or commit (100% `VOTE_YES` rate) certainty, the higher the number of retransmissions that have to be executed in order to reach a quorum of nodes with the same reply.

Obtaining optimal values for ΔQ and M-Slot is non-trivial and topology dependent. While testing we found that 35ms M-Slot values coupled with a ΔQ of 3 was able to reliably yield the lowest latencies and highest transaction outcome across retransmissions (see Figure 4.6). Any application of WISP will, though, have to ensure that appropriate tuning of these parameters is performed in order to maximise the protocol's performance.

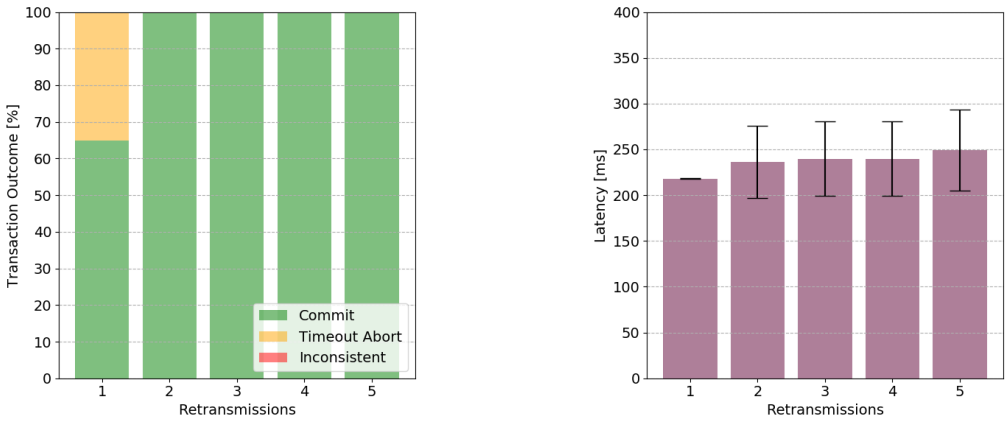
WISP thus extends the idea of consensus in low-power wireless networks to a configurable majority voting paradigm. As there is no needs for individual proposers to fight for proposal numbers (as all transactions are executed via the global host), WISP is able to obtain minimal latency (Figure 4.6b) while guaranteeing perfect reliability starting from 2 retransmissions

(Figure 4.6a). By using the Hybrid ST primitive, WISP offers a reliable solution to consensus in WSNs, meeting and solving **C3** presented in Section 2.10.

4.2.2 Applications

Out of the box WISP is configured to behave as a majority-based parliament. Similar-to-Paxos guarantees can be obtained if all nodes have a 100% approval rate of proposed values. As all protocols use XPC, all voting code is abstracted to an application-defined callback function. Any arbitrary code can therefore be executed to determine whether a node wishes to vote in favor or against or a specific proposed value. This opens up for a number of potential uses:

- **Configuration management**. A common configuration management task is to reliably disseminate channel allocations, in channel hopping communication schemes. It is important for the network to hop to channels which guarantee a benefit to a majority of nodes, and not only to the new channel's proposer. WISP excels at such forms of configuration management, where nodes are allowed to vote before blindly accepting a new setup. Overall the network will constantly be more stable and more resilient to failures or congestion.
- **Leader election**. ST primitives can be used to greatly decrease the latency of all leader election protocols, and this is a topic of recent academic interest [3]. WISP allows applications to implement fully-configurable election rounds where proposed leaders can be determined based on metadata stored within the node itself. This could include routing information held by the protocol to exclude leaders it knows it has poor link reliability to.
- **Clustering**. Dense networks suffer from taking longer time to converge during Chaos floods and a number of approaches such as local clustering have been proposed [30]. WISP-based applications can be tailored for this functionality, having nodes perform leader-election rounds to determine cluster heads and then partition the large network into individual sub-clusters.
- **Failure free commits**. What really makes WISP shine, though, is its ability to commit in the presence of failures. Not all nodes need to be present during a round's execution



(a) WISP transaction outcome

(b) WISP retransmission latency

Figure 4.6: Evaluation of transaction outcome and latency for WISP in FlockLab. 100 runs per data-point were executed with 35ms M-Slot values and a ΔQ of 3.

and only a quorum of the total cohort has to reply to the host with the same message. WISP is therefore inherently failure tolerant. If nodes are expected to commit on all rounds, it means that potentially up to $50\% - 1$ nodes could be allowed to fail with the protocol still being able to terminate. Networks with lossy links or potentially misbehaving nodes could greatly benefit from such a guarantee where an individual straggler mote does not delay or actually timeout the whole network's protocol execution.

4.3 Next Steps

This chapter has introduced WISP together with the Wireless Part-time Parliament, an extension of the XPC library which allows for network-wide majority voting of individual values.

WiPP is provided as a protocol for XPC and may be used by any application layer which should require it. Section 4.1 goes through the fundamentals of WiPP covering techniques used to ensure low latency and fast protocol convergence such as ΔQ, M-Slots and the final global dissemination round.

A vanilla implementation of the Wireless Part-time Parliament is WISP, an enhanced Paxos adaptation for low-power multi-hop networks. Being able to match the latency of state-of-the-art protocols such as WPaxos, WISP (analysed in Section 4.2), opens up XPC for a variety of reliable real-world applications. Solving **C3**, WISP provides a solution for WSN tasks which require reliable consensus, such as leader election, configuration management and local group clustering. Evaluation of the WISP protocol, in terms of latency and reliability, together with comparisons to the WPaxos implementation, can be found in Section 6.

To complete XPC's support for network-wide consensus the next step is to allow for any node in the cohort to propose values to the global leader. The next chapter extends XPC with multiple initiator functionality and introduces WIMP, a WiPP extension for Multi-Paxos.

5 | WIMP and Multiple Network Proposers

This chapter presents WIMP, a Multi-Paxos implementation for XPC based on the Hybrid ST primitive. To enable all nodes in the network to propose values to the Baloo host we extend XPC to support multiple proposers. Multiple proposal is independent of protocol implementation and can be used by any protocol together with any ST primitive supported by XPC. WIMP uses multiple proposers, together with the Hybrid ST primitive and the Wireless Part-time Parliament to provide a reliable, low latency Multi-Paxos implementation.

Section 5.1 introduces the multiple proposer feature. It analyses how the use of contention slots, together with minimal additions to the payload of primitives used for dissemination, guarantees proposal liveliness and safety. Section 5.2 then introduces WIMP, analysing its implementation within XPC.

5.1 Multiple Proposers

In this section we present an extension to WiPP and XPC to allow for multiple proposers. Multiple proposers means that any node within the network is allowed to propose during a round with the guarantee that all proposed values will eventually be voted upon, and their respective transaction outcome globally disseminated. Due to the presence of the global host, most protocols developed for Baloo leverage the leader to execute all value proposals. This is a valid and energy-efficient approach, minimising the number of rounds required for each proposals, therefore maximising the mote's duty-cycling times.

There are a number of potential applications, where values may be generated from any node in the network and be voted upon to determine global acceptance. Protocols such as LWB [14] allow for this many-to-many sharing of payloads. The recent WPaxos [32] implementation does also supports multiple values being proposed concurrently within the network, with one of them being eventually committed and disseminated to all nodes.

Multiple proposer is included in XPC, and is subject to the following guarantees:

1. **All values will be proposed**. If the application layer for any node wishes to propose a value before a protocol run, said value will eventually be proposed during the subsequent protocol run.
2. **All values will be voted upon**. Once a value is proposed, it will be voted on by at least a quorum of nodes.
3. **Protocol latency increases inversely to inter packet arrival rate**. The packet generation rate can be modelled in terms on an hypothetical per-node IPI (inter packet interval). Given a network of N nodes, in order to obtain an average of one value being

proposed each round, motes must have an IPI of N rounds (i.e. generating a packet every N rounds). As the IPI decreases for each node (therefore generating values to be proposed more frequently), the overall latency of each protocol run will increase as it must express and disseminate a vote for each of these values.

5.1.1 Technical Overview

In order to guarantee an implementation of multi proposers which is transparent to the protocol layer, most of the scheduling has to occur behind the scenes leveraging the existing round structure. To accommodate multiple proposers XPC uses contention slots. During a contention slot all nodes that are willing to propose a value to the network will broadcast the value at the same time. The broadcast uses Glossy floods. Due to the capture effect there is a high probability that at least one of these replies will be reliably heard by the XPC host (even though all nodes willing to propose will broadcast their value during the same contention slot). Upon receiving a value from the network the selected XPC protocol is executed throughout all of its phases (as outlined in Figure 5.1).

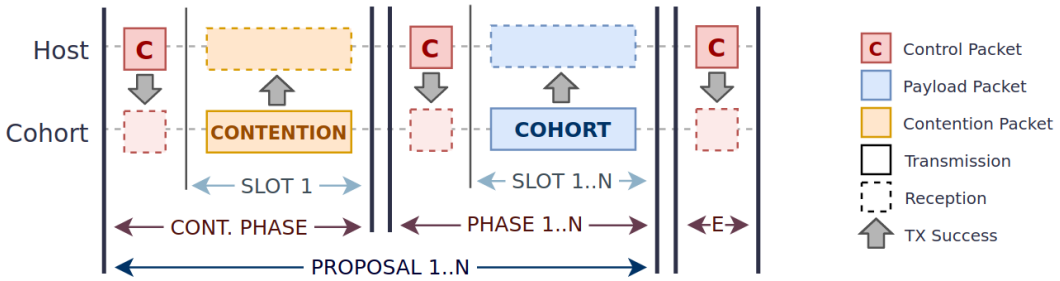


Figure 5.1: Overview of how XPC schedules contention slots (preceding all protocol phases) to enable multiple values to be proposed and voted within the same protocol run.

When multiple proposers are initiated, XPC schedules a contention slot as the first phase of each protocol run. If the host node wishes to initiate it is given precedence and will override the round structure by removing the contention slot and directly proposing its value first to the network. If the host, though, does not initiate, contention rounds are bound by the following rules:

1. **Determining proposed value**. Due to the capture effect if two or more nodes in the network flood a value during the contention slot, one of them will be correctly received by the host with a high probability. If a value is heard by the host it will initiate the protocol from Phase 1 proposing the newly retrieved value. If no value is heard the host switches to the final end round (denoted as E in Figure 5.1).
2. **Scheduling a new contention slot**. During each round XPC keeps track of a pending proposal pit, PPB for short, which allows the library to determine if there are other nodes in the network which are willing to propose (more detail in Section 5.1.2). If at any point during a protocol run PPB is set, then there must be at least one node in the cohort which wishes to propose a new value. In this case the protocol does not terminate and instead a new proposal phase is initiated by scheduling a new contention slot allowing for the node(s) to broadcast their proposals.
3. **End of protocol execution**. If during any contention slot either no value is heard, or the pending proposal bit was never set by a node during a protocol round, then XPC

switches to the final end round to prompt all network nodes to terminate the protocol's execution and run the application code. Upon terminating, XPC constructs the packet that will initiate the next round. When multiple proposers are enabled it will always contain a contention slot, this will, though, be overridden in case the host itself is willing to propose a value.

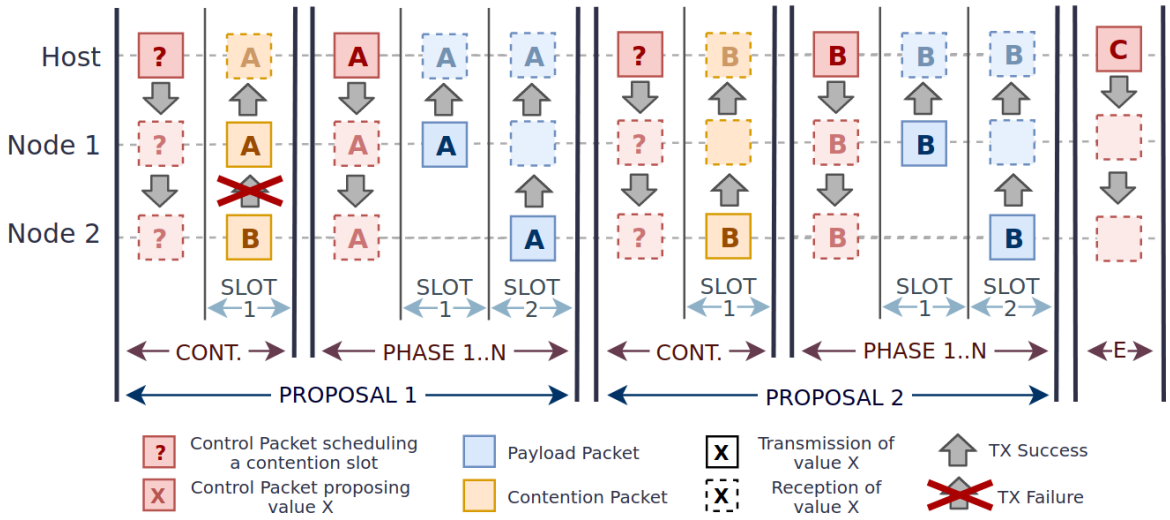


Figure 5.2: Example execution of an N-Phase protocol with multiple proposers. Values A and B, from nodes 1 and 2 respectively, are proposed and voted upon by the network.

The multi proposer logic can be seen in action in Figure 5.2. As the host node does not have a value to propose during this run, XPC schedules a contention slot to begin the first proposal round. Both node 1 and node 2 will attempt to broadcast their values (A and B). Due to the capture effect the host will only pick up one of them, in this case value A. As the N-Phase protocol is run with value A the host picks up that the pending proposal bit (PPB) has been set, and therefore, after having committed A, schedules a new contention slot during which it receives a new value to propose: B. During the protocol's execution with value B no node manifests the intention of proposing again over the network. XPC therefore schedules the end round and terminates.

5.1.2 PPB and PVB

XPC's multiple proposer implementation uses a reactive method. The host will propose any value that it receives during a contention slot and must take additional measures to determine if there are pending packets in the network and if the values which are being proposed were actually originating from a mote in our cohort. For these purposes we introduce two additional bits of information which are appended to each payload exchanged by the protocol:

- **Pending Proposal Bit (PPB)**. This bit carries information of whether there is at least one node in the network which has a value which is pending to be proposed. Whenever a node is set to broadcast it checks if it has a pending value and sets the PPB accordingly. The PPB is not owned by any node (as during Chaos slots there is no way of determining which node has set it), and it is only a means for the host to determine if a contention slot must be scheduled at the end of each round or not. Due to the fact that the PPB

determines the scheduling of the end round, and thus protocol termination, it can be seen as a liveliness property guarantee: eventually the PPB will not be set and the protocol will terminate.

- **Proposed Value Bit (PVB)**. This bit carries information of whether the currently proposed value was actually proposed by a node in the network. Potential node failures could cause a node to propose a value during a contention slot but no longer be active when the value is ready to be committed. PVB adds this sense of ownership to proposals and allows the host to actually abort values if no node acknowledges it has proposed it. As PVB guarantees node acknowledgement of proposed values it enforces a safety property: the network will only accept values proposed by a live node in its cohort.

Both PPB and PVB rely on all nodes broadcasting a reply to the host at some point during the protocol's execution. This means that any protocol using multiple proposers is allowed to have any number of majority voting rounds but must have at least one universal voting or acknowledgement phase. In the specific Wireless Part-time Parliament case, the reliable global dissemination round requires acknowledgements from all nodes in the network and is able to guarantee that PPB is set correctly.

WiPP is unable to guarantee PVB correctness. As the global dissemination round occurs after a majority of nodes have agreed to commit the value, it is possible that none of these nodes set the proposed value bit. Once the global dissemination round begins it is too late to abort a Paxos transaction (as it would break the safety guarantee, having a majority of nodes already committed). PVB information may be obtained too late for WiPP to abort a transactions corresponding to invalid values. Should correctness of proposed values be a concern, an optional `XPC_PROPOSAL_GUARANTEE` feature can be enabled within the project configuration. With proposal guarantee XPC protocols are prohibited from executing a commit before they have certainty that the PVB has been set. This means that majority rounds may take longer than expected as more than a majority of nodes could be contacted in order to reach the one mote which proposed the current value. Enabling the safety checking of proposal guarantee can therefore slightly impact latency of majority-based protocols such as WiPP, while having no significant effect on 2PC or 3PC.

The pending proposal and proposed value bits allow XPC to have a reliable reactive multiple proposer implementation. Both bits are packed within the underlying primitive implementations. They add two extra bits to the Chaos payload, and have no impact on the size of Glossy packets.

5.2 WIMP: WiPP Multi Paxos

With the introduction of multiple proposers the Wireless Part-time Parliament can be used to generate a simple Multi-Paxos implementation. We propose WIMP, an XPC-based protocol able to execute multiple back-to-back WiPP rounds, constantly disseminating all partial transaction outcomes.

WiPP Multi Paxos (WIMP for short) schedules a contention slot at the beginning of each proposal round (see Figure 5.3). The contention slot is enabled by the multi proposer functionality of XPC and thus follows the previously mentioned safety and liveliness guarantees: the protocol will terminate once all nodes have proposed their values (liveliness) and no value will be committed by the network which wasn't proposed by a node in the cohort (safety). Individual WiPP rounds are run for each proposed value, allowing each proposal to be voted upon, committed and disseminated individually to the network. The global Baloo host is also

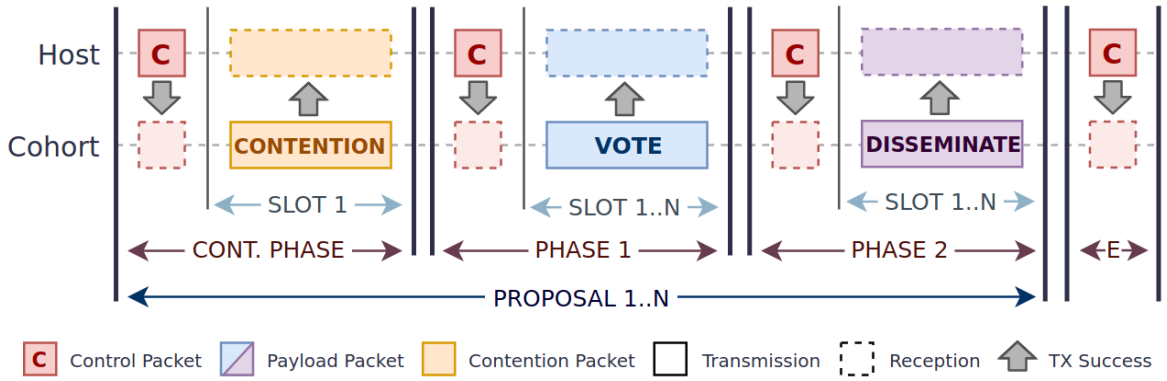


Figure 5.3: Overview of WIMP's phase structure. All proposal rounds are initiated by a contention slot followed by majority voting and global dissemination rounds.

allowed to initiate itself, in which case the first proposal round commences directly by flooding the host's proposal to the whole network during the voting phase.

WIMP differs from Wireless Multi-Paxos (§ 2.7.1) due to the underlying ST primitive used and the multiple proposer approach. Wireless Multi-Paxos uses Chaos floods to execute all data dissemination. WIMP utilises the more reliable Hybrid primitive introduced in Section 3.2.3. Additionally while in Wireless Multi-Paxos value proposals to the leader can occur at any time, potentially conflicting with protocol phases concurrently being executed, WIMP uses the structured multi-proposer strategy defined in Section 5.1.

5.2.1 Paxos optimisations

WIMP introduces a number of optimiations to Paxos with two main aims. On the one hand low-power wireless networks are incredibly memory constraint and must ensure that the smallest possible buffers are allocated and used as efficiently as possible during execution. On the other hand we need to utilise the network structure provided by XPC and Baloo. Currently Baloo uses a global leader to disseminate control packets each round; we already leverage the leader to be in charge of contention slot scheduling and processing, and could potentially abstract further protocol-based guarantees, namely:

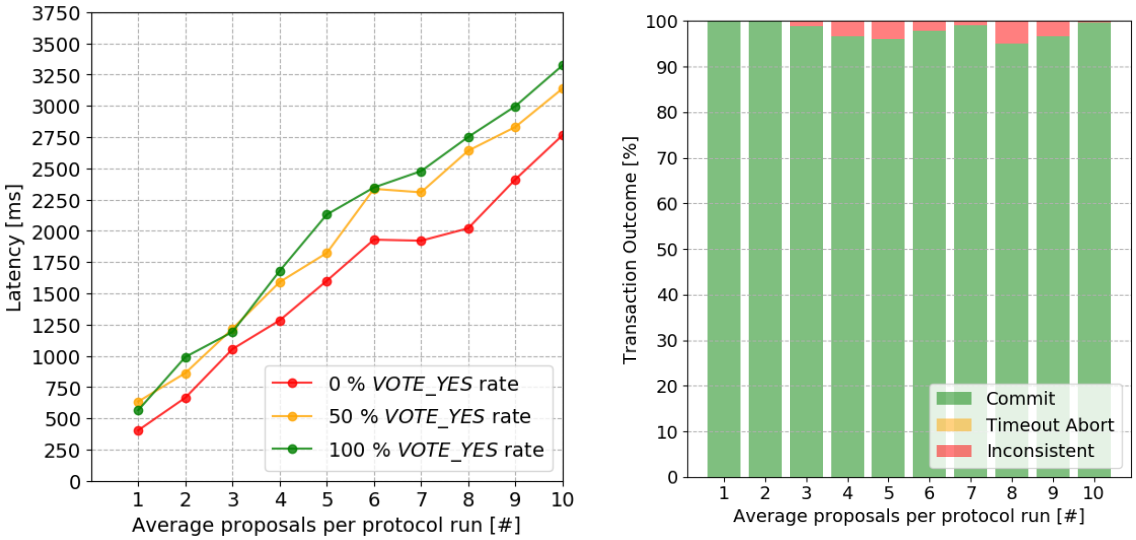
1. **Long lasting leader**. Using the Baloo host as a global leader WIMP removes the need for ballot numbers as there will never be two nodes proposing to the network at the same time (as proposals can only be executed with the control packet). Proposal rounds will therefore be executed back-to-back as long as there are pending values to be proposed by cohort nodes.
2. **Message ordering**. The global host provides a global ordering guarantee of transactions. Due to the global dissemination round all transactions will have a local ordering on each individual node. As the host only picks one value at a time to be broadcast after a contention slot, this stronger requirement ensures that all network nodes constantly share the same transaction history. If a node is part of the network it must reply during global dissemination rounds (as otherwise the host executes a timeout) and therefore will have to execute commit and abort operations in the same order as all other members of the cohort.
3. **Bounded memory**. The infinite memory requirement for Multi-Paxos is relaxed as all

wireless nodes are bound by a finite amount of memory. Due to the global dissemination of transaction outcomes and lack of proposal numbers, nodes do not need to worry about inconsistent logs. Values are only consistently committed once all nodes have acknowledged them during the dissemination round. No node needs to account for missing proposed value transaction outcomes.

4. **Prepare-phase specifics**. In Multi-Paxos new nodes have to learn all past transactions of the network before accepting new values. WIMP nodes learn outcomes of all transactions as the network progresses. As dynamic group membership is not yet supported (§ 7.3), new nodes are not allowed to join the network after the protocol has begun to execute, all participants will constantly be kept up to date with the protocol's execution.

5.2.2 Multiple Proposer Impact

As long as there are nodes willing to propose, WIMP will keep scheduling contention rounds followed by full WISP runs for each value. WIMP latency is therefore expected to scale linearly with the average number of concurrent proposers. WIMP pays an additional cost for having multiple initiation, increasing the fixed latency cost for the execution of each protocol run. As with all WiPP-based protocols, the cost of a commit round is also expected to be higher than that of an abort, as transaction commits must be globally disseminated and acknowledged by all cohort nodes.



(a) WIMP protocol latency.

(b) Transaction outcome for 100% `VOTE_YES` rates.

Figure 5.4: Analysis of WIMP's behaviour with increasing node IPIs.

These assumptions are verified through an empirical analysis of WIMP's execution. Figure 5.4a shows a clear linear correlation between average number of concurrent proposals and overall protocol latency. Once again we analyse the protocol performance with different `VOTE_YES` rates. As the number of proposers increases we see that latency grows linearly for varying node `VOTE_YES` rates. Protocol reliability is slightly affected by higher IPIs, while still being able to consistently commit $> 95\%$ of transactions. Unfortunately WSN nodes are bound by strict memory requirement meaning that our current experimentations are limited by buffer overflows (we cannot store more than 8MBytes of data on each node, and we can only flush

to serial when the application is being executed as otherwise the synchronous Glossy floods could be delayed); this can cause certain values to occasionally go missing due to lack of log space on larger IPIs (Figure 5.4b). Overall the protocol reliably reaches all nodes willing to transmit and executes WISP rounds for all proposals. WIMP is therefore a viable Wireless Multi-Paxos alternative built on top of XPC's reliable dissemination structure.

5.3 Next Steps

This chapter has introduced WIMP together with support for multiple network proposers for XPC. Using contention slots any node in the network is able to propose values to the global host. All proposed values are guaranteed to eventually be voted upon by the cohort. XPC's multiple proposer implementation is independent from the underlying protocol or ST primitive used. This allows for XPC users to fully customise the library, tailoring it for their specific uses and requirements.

The multiple proposer implementation is showcased together with the Wireless Part-time Parliament. Implementing Multi-Paxos for WSNs, WIMP uses multiple initiation to show how multiple values can be reliably disseminated during subsequent proposal rounds. Section 5.2 analyses WIMP's implementation together with it's latency and reliability.

The next step is to finally see all XPC protocols in action. The evaluation in Section 6 will consist of an analysis of XPC's reliability by benchmarking its protocols in various failure and interference scenarios. The results, together with the overall protocol latencies, will be compared to state-of-the-art implementations published in 2017 (A^2) and 2019 (WPaxos).

6 | Evaluation

In this section we present experimental results validating the claims made about XPC, WISP and WIMP. Our analysis will be executed in terms of reliability, latency and broadcast efficiency.

Section 6.1 addresses **C4** by analysing the behaviour of protocols when tested under increasing levels of controlled interference. The aim is to evaluate the high reliability guarantees of XPC, investigating how interference impacts the various universal and majority voting protocol phases. In Section 6.2 we compare XPC and WiPP to existing, state-of-the-art, implementations based on A^2's synchronous transmission kernel. This includes comparison with the original 2017 2PC/3PC implementation [2], and the WPaxos extension proposed in 2019 [32]. The analysis is concluded in Section 6.3 with a brief discussion of the challenges addressed and solved by this report.

6.1 Interference Analysis

We analyse the behaviour of XPC under varying amounts of radio interference. Interference causes nodes to miss broadcasted packets, and potentially desynchronize from the network (causing transmission slots to be missed); protocol reliability must therefore be analysed in the presence of network interference.

We evaluate XPC using the reliability and latency metrics. The analysis focuses on Hybrid based protocols (see Section 3.2.3). Four main protocols are selected as targets for evaluation under interference:

- **2PC-Hybrid**. Composed of two phases: a universal voting round and final global dissemination.
- **3PC-Hybrid**.Composed of three phases: two universal voting rounds and final global dissemination.
- **WISP**. Composed of a majority voting round and final global dissemination.
- **WIMP**. Composed of a contention slot to allow for network-wide proposals, followed by a majority voting round and final global dissemination.

All protocols share a global dissemination round where, if a reply isn't heard by all network nodes, the transaction is aborted. All protocols utilise the Hybrid ST primitive, meaning that broadcasts will commence with a network-wide Chaos flood and will be followed by Glossy retransmissions.

6.1.1 Experimental Setup

When testing with interference the protocols are run on a subset of the Flocklab testbed. 22 (out of the possible 27) nodes are used as the cohort and the 5 remaining nodes are used as jammers. An overview of the individual configuration parameters used is present in Table 6.1. Most notably, all nodes in the network vote in favor of all proposed values (100% agreement rate) and each protocol phase is allowed up to 10 retransmissions before timing out and aborting.

Parameters		**Used by Protocols**			
Name	**Value**	**2PC**	**3PC**	**WISP**	**WIMP**
Network Nodes	22	X	X	X	X
Max Retransmmissions	10	X	X	X	X
Chaos Round Length	50ms	X	X	X	X
Agreement Rate	100%	X	X	X	X
M-Slot Length	35ms			X	X
Δ Q Size	3			X	X
Concurrent Initiators	1				X

Table 6.1: XPC configuration for Flocklab interference tests.

To address **C4** (presented in Section 2.10) we generate reliable and accurate interference patterns using JamLab [6], a customizable off-the-shelf interference generation library for WSN motes. The following interference patterns (as discussed in Section 2.9.2) are used to perform comparisons:

1. **Low Interference** is determined by background noise on the FlockLab testbed during night-time hours (9pm-6am). It models an ideal network deployment with few external conflicting broadcasts occurring on the channels used by the protocol. When analysed, protocols should give their best performance under these conditions, both in terms of speed and in terms of reliability.
2. **High Interference** is determined by background noise on the FlockLab testbed during day-time hours (7am-8pm). It provides an estimate of average real-world conditions in order to set realistic expectations for the protocols being tested. In this case interference is not controlled.
3. **WiFi Interference** is generated by JamLab and emulates the interference of non-saturated WiFi file transfers and radio streaming.
4. **Microwave Interference** is generated by JamLab and emulates the periodic interference caused by microwave ovens over 802.15.4 transmission channels.

Both types of JamLab injected interference were executed during night-time hours. This was done to minimise the influence of the multitude of different devices broadcasting during office hours. During testing up to 5 nodes could be configured as jammers. None were activated during high/low interference measurements. Our FlockLab interference analysis can therefore be expressed with the following metrics:

- **Interference Model**. Low, High, WiFi and Microwave interference models were tested and evaluated individually for each protocol.
- **Average Reliability**. Protocol reliability measures the rate at which all nodes in the network commit the proposed transaction consistently. If even just one node times out or aborts, the reliability is scored as zero for the given round.
- **Latency**. Latency measures the overall time from the beginning of an XPC transaction (i.e. when the Application layer is preempted) until the application is resumed at the end of the XPC round. As the interference increases, overall protocol latency is expected to rise to account for nodes being scheduled for retransmission.
- **Chaos Round Coverage**. The Chaos Coverage metric analyses what percentage of network nodes was reached, on average, during the first chaos dissemination of each phase. Universal and majority voting rounds differ in the lengths of the initial Chaos flood for each protocol phase. Majority phases (**MP**) have M-Slot length floods, while all other phases (**P1**, **P2**, **P3**) and the dissemination phase (**DP**) use the default Chaos Round Length configuration.
- **Average Number of Retransmissions**. This metric analyses the number of retransmissions required for a protocol to switch to a subsequent stage. **P1**, **P2**, **P3** and **DP** phases must receive replies from every node in the cohort before switching, whereas **MP**, the majority phase, is able to proceed when $\lfloor 50\% + 1 \rfloor$ of the nodes are reached (12 out of 22 nodes in the case of the reduced FlockLab network).

6.1.2 Simple Interference Models

A full execution of all proposed XPC protocols, evaluated under the four interference models outlined above can be seen in Table 6.2. In order to obtain a baseline useful for future comparisons, the Table reports WiFi and Microwave interference values in the presence of one network jammer.

The data presented in Table 6.2 can be analysed for each of the proposed metrics:

- **Reliability**. All protocols across all interference models achieve a 100% correct transaction outcome. This data backs up and validates XPC's strong reliability claims, as, not only is the correct functionality of the protocols maintained in ideal (Low Interference) and normal (High Interference) network conditions, but it is also able to sustain interference specifically injected to cause packet loss and broadcast conflicts.
- **Latency**. As the interference models increase their disturbance over the channel, protocol latencies increase linearly. With the stronger WiFi and Microwave jamming, majority voting protocols such as WISP and WIMP require replies only from a cohort of nodes during the voting stage, their latencies increase at a slower rate if compared to universal voting approaches such as 3PC's.
- **Chaos Coverage**. The full potential of the new Hybrid ST primitive can really be seen when analysing the percentage of the network reached during the initial Chaos floods. Being able to reach over 90% of nodes during the first 50ms of each phase is crucial for time efficiency purposes; subsequently switching to reliable Glossy broadcasts is then able to compromise for Chaos' unpredictable termination time and lack of reliable detection of straggler nodes.

Low Interference	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	100.00	222.95	**P1**: 98.51 **P2**: 99.27	**P1**: 1.10 **P2**: 1.07
3PC-Hybrid	100.00	333.72	**P1**: 98.74 **P2**: 98.99 **P3**: 99.63	**P1**: 1.11 **P2**: 1.12 **P3**: 1.01
WISP	100.00	215.87	**MP**: 98.49 **DP**: 99.46	**MP**: 1.00 **DP**: 1.09
WIMP	100.00	329.25	**MP**: 98.58 **DP**: 98.47	**MP**: 1.00 **DP**: 1.15
High Interference	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	100.00	304.19	**P1**: 92.83 **P2**: 88.38	**P1**: 1.52 **P2**: 1.59
3PC-Hybrid	100.00	432.14	**P1**: 94.81 **P2**: 94.38 **P3**: 94.39	**P1**: 1.40 **P2**: 1.52 **P3**: 1.51
WISP	100.00	241.43	**MP**: 93.40 **DP**: 97.41	**MP**: 1.01 **DP**: 1.47
WIMP	100.00	350.25	**MP**: 93.61 **DP**: 92.57	**MP**: 1.00 **DP**: 1.54
Wifi Interference	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	100.00	383.07	**P1**: 92.19 **P2**: 90.54	**P1**: 2.29 **P2**: 2.19
3PC-Hybrid	100.00	606.20	**P1**: 91.32 **P2**: 92.92 **P3**: 89.58	**P1**: 2.26 **P2**: 1.96 **P3**: 2.31
WISP	100.00	270.93	**MP**: 87.79 **DP**: 91.98	**MP**: 1.05 **DP**: 1.44
WIMP	100.00	360.88	**MP**: 86.09 **DP**: 91.98	**MP**: 1.04 **DP**: 1.70
Microwave	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	100.00	396.06	**P1**: 92.97 **P2**: 91.82	**P1**: 2.07 **P2**: 2.17
3PC-Hybrid	100.00	586.67	**P1**: 92.09 **P2**: 91.58 **P3**: 92.30	**P1**: 1.95 **P2**: 2.12 **P3**: 1.95
WISP	100.00	342.53	**MP**: 78.90 **DP**: 91.14	**MP**: 1.15 **DP**: 2.23
WIMP	100.00	377.31	**MP**: 86.22 **DP**: 91.69	**MP**: 1.01 **DP**: 1.66

Table 6.2: Comparison of XPC protocol implementations across varying interference levels.

- **Average Retransmissions**. Similar to the analysis with latency and Chaos round coverage, the average number of phase retransmissions reflects the intensity of the channel's interference. As the interference increases all global dissemination and universal voting rounds require more retransmissions to ensure 100% protocol reliability. An exception are majority phases which, on average, reach a sufficiently large cohort of nodes to require few sporadic retransmissions (none in the case of low interference).

6.1.3 Multi-node Interference

To further evaluate the reliability of XPC-based protocols, our analysis extends the simple interference model to consider multiple nodes jamming the network. We will focus on microwave oven interference, as it generated the highest latencies in the case of 1 JamLab node (in Table 6.2). By increasing the levels of microwave oven interference we are therefore more likely to impact the robustness of the protocol. Comparisons are carried out between 2PC and WISP, which are respectively representative of XPC's universal-voting and majority-voting approaches. Table 6.3 reports the results of executions with up to 5 interfering nodes.

Microwave (2 Nodes)	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	100.00	1027.18	**P1**: 82.91 **P2**: 81.54	**P1**: 4.64 **P2**: 5.04
WISP	98.33	625.37	**MP**: 56.49 **DP**: 82.85	**MP**: 1.57 **DP**: 4.52
Microwave (3 Nodes)	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	95.92	1025.86	**P1**: 80.26 **P2**: 81.43	**P1**: 4.95 **P2**: 5.11
WISP	92.86	756.73	**MP**: 47.00 **DP**: 81.37	**MP**: 1.97 **DP**: 5.22
Microwave (4 Nodes)	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	57.14	1221.30	**P1**: 79.59 **P2**: 79.92	**P1**: 6.72 **P2**: 6.11
WISP	59.18	1024.20	**MP**: 42.06 **DP**: 78.83	**MP**: 2.27 **DP**: 6.91
Microwave (5 Nodes)	**Reliability (%)**	**Latency (ms)**	**Chaos Coverage (%)**	**Avg. Retr.**
2PC-Hybrid	47.22	1291.68	**P1**: 75.51 **P2**: 79.37	**P1**: 7.85 **P2**: 7.50
WISP	59.09	1178.84	**MP**: 41.67 **DP**: 73.88	**MP**: 2.83 **DP**: 7.67

Table 6.3: 2PC and WISP analysis with increasing levels of JamLab microwave interference.

As shown in Table 6.3 high levels of interference end up impacting the protocol's reliability. With 4 interfering nodes 2PC-Hybrid and WISP are correctly committing only 60% of transaction, timing out due to missing replies in other cases. A decrease in reliability also affects latency and average retransmissions. In an attempt to reach all straggler nodes the protocols reach the maximum number of retransmissions (10 for the execution of these tests) and timeout during the execution of global voting or dissemination rounds (**P1**, **P2** and **DP**).

Overall network performance, though, does not reflect the status of the individual nodes. Figure 6.1 visualises the reliability of each Flocklab node when executing WISP with 5 interfering nodes. It identifies how the 59% reliability of WISP is mainly due to the lack of reliability of an individual node: mote 18. Even though WISP has majority-based voting, the global dissemination round requires an acknowledgement from each network participant, meaning that a single straggler node can cause the protocol to timeout.

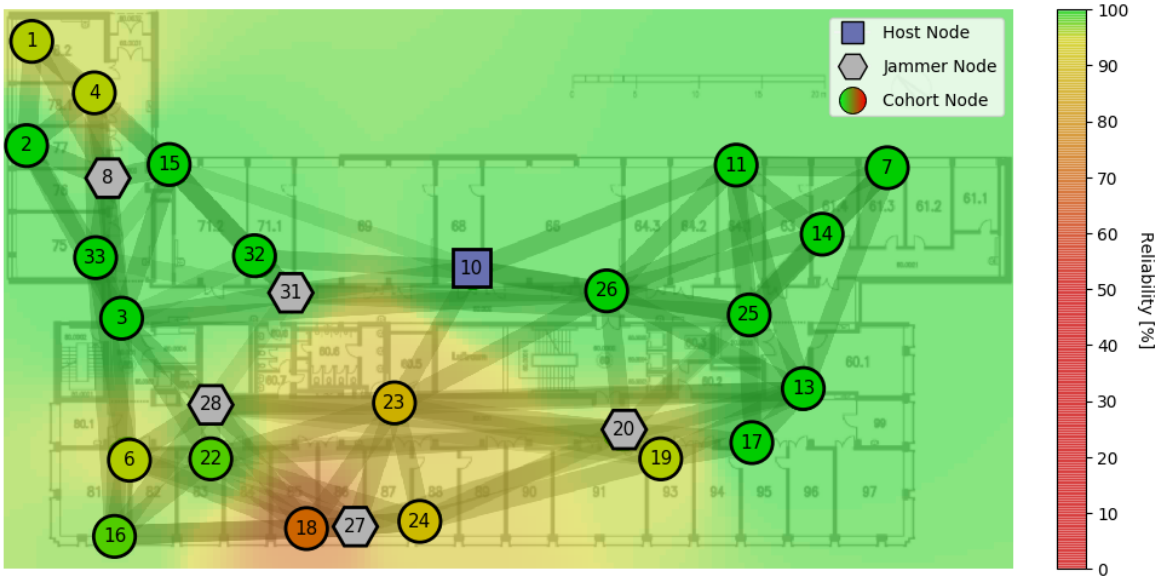


Figure 6.1: Visualisation of the reliability of each node during WISP protocol executions with 5 Microwave oven JamLab nodes on FlockLab.

The impact of individual node failures is also reflected in the Chaos coverage and average retransmission metrics. Even under heavy interference the first Chaos round reaches more than 70% of the network, yet the average number of retransmissions is incredibly high. This is due to as a small part of the network (as Figure 6.1 suggests, potentially even one node) which is constantly unreachable and blocks the protocol from making progress. Dynamic group membership (§ 7.3) would greatly increase the reliability of XPC protocols under interference. Being able to exclude straggler nodes from the network, would prevent individual nodes to cause network timeouts.

Ideally a reliability-under-interference interference comparison should be executed with A^2. Unfortunately the official WPaxos repository [33] does build when cloned from Git (requiring multiple adjustments to the codebase for compilation), and the A^2 source-code [1] has a tightly coupled codebase. The number of alterations necessary to execute JamLab alongside a cohort of 22 Flocklab nodes may cause alterations to the main protocol logic, impacting the truthfulness of the results. Sadly this is typical of research in this area and is a result of working with bleeding edge unstable research code.

This Section has analysed the performance of XPC under increasingly stronger interference injected using JamLab. This addresses **C4** presented in Section 2.10 by providing reliable results reproducible by the research community. Being unable to analyse A^2 under failure, the evaluation continues by comparing results of XPC protocols with those presented in the A^2 and WPaxos publications.

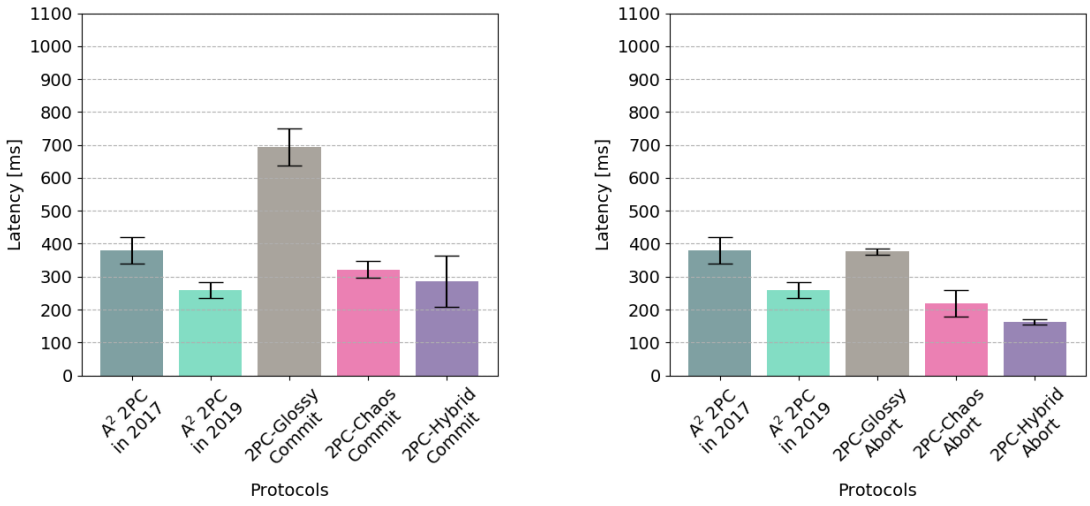
6.2 Comparison with A² implementations

We compare the latency of XPC to the execution times reported by A² in 2017 [2] and 2019 (in the WPaxos paper [32]). Our analysis covers all protocols implemented within XPC across all ST primitives. 2PC and 3PC both support Glossy, Chaos and Hybrid implementations, whereas the Wireless Part-time Parliament is evaluated through WISP and WIMP (which use the Hybrid ST primitive). Tests are executed on Flocklab using all 27 nodes.

We show how protocols using XPC reach, and better, the latency of their A² equivalents. The Hybrid ST primitive, which has been proven to be reliable in Section 3.2.3, constantly outperforms Glossy and Chaos in terms of latency. The A² implementations do not disclose if their data is measured with commit-only transactions, or if aborted transactions were considered as well. In our analysis of XPC we present both sets of data side by side. Transactions abort if at least one node votes against a proposed value during the voting stage. Aborted transactions have lower completion latencies. If aborted transactions are considered, they impact measurements of upper-bound protocol execution times.

6.2.1 Two and Three-Phase Commit

Hybrid outperforms Glossy and Chaos for 2PC implementations (see Figure 6.2). 2PC-Hybrid is able to match the 2019 A² latencies for commit-only transactions (Figure 6.2a), and provides a significant speedup in the case of network-wide aborts (Figure 6.2b).

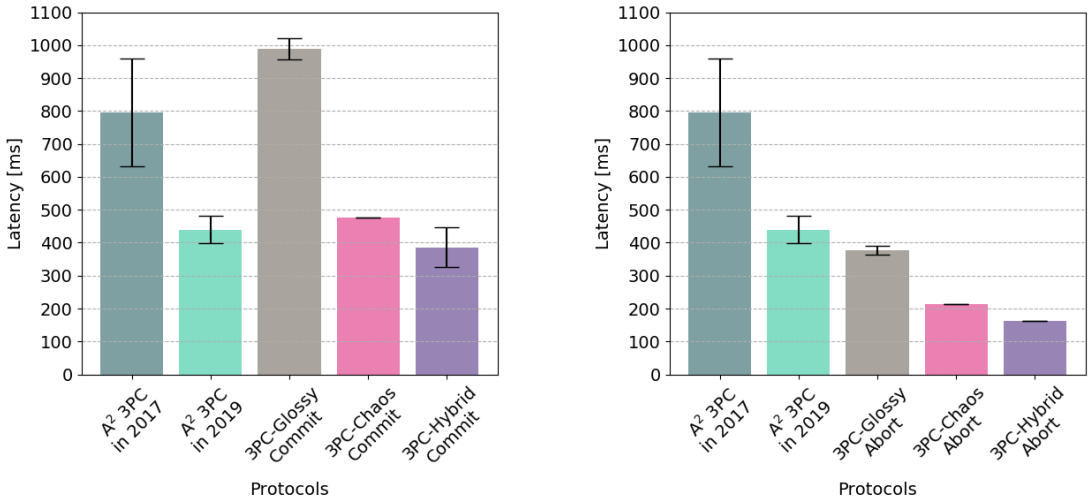


(a) 2PC Commit Comparison

(b) 2PC Abort Comparison

Figure 6.2: Comparison with A² implementations over all XPC versions of 2PC.

Hybrid has also the lowest latency among all ST primitives for 3PC (see Figure 6.3). Similarly to 2PC, 3PC-Hybrid matches A²'s 2019 implementation for transaction commits (Figure 6.3a) and provides a significant improvement over aborts (Figure 6.3b). This validates the claim that the Hybrid ST primitive fulfills **C1**. Section 6.1 proves the resiliency and robustness to interference, and this section confirms its low latency. **C2** is additionally validated both by the comparisons executed in this section (which certify that an identical protocol implementation can use multiple ST primitives), and by the correct execution of Hybrid while switching between Chaos and Glossy rounds.

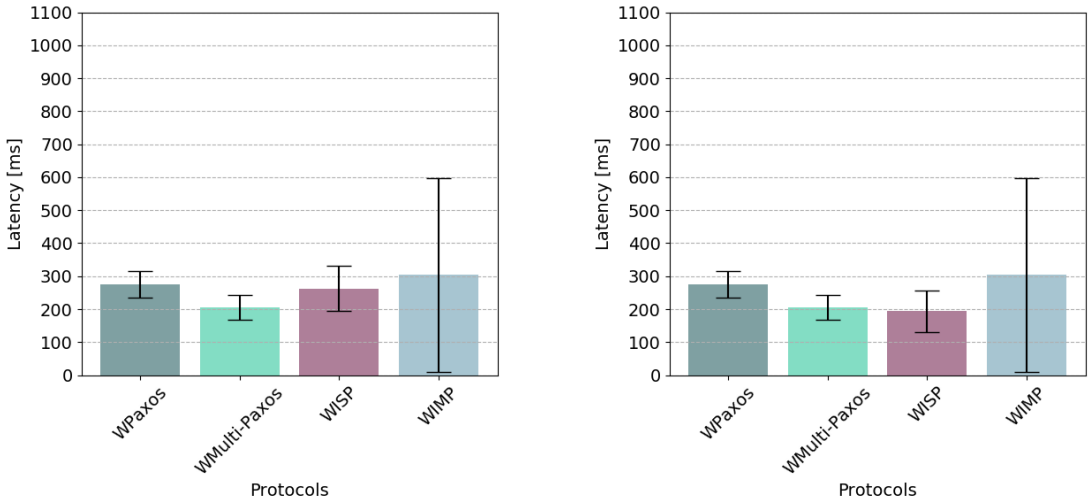


(a) 3PC Commit Comparison (b) 3PC Abort Comparison

Figure 6.3: Comparison with A^2 implementations over all XPC versions of 3PC.

6.2.2 WPaxos and WiPP

WISP matches the latency of WPaxos for commit workloads (Figure 6.4a) and provides a speedup during the execution of aborts (Figure 6.4b). WIMP is impacted by the addition of the contention slot for multiple proposal, being slightly outperformed by both WISP and WMulti-Paxos implementations. The probabilistic approach to network proposers (i.e. an average of one value is proposed each WIMP round) also causes high variance in protocol execution times. It is possible for WIMP to either execute 2 back-to-back proposals, or even have empty rounds, while still maintaining an average of one value per protocol execution. Furthermore it is harder to determine the exact speedup of WIMP aborts (Figure 6.4b) as both commit and abort transactions may be executed during the same protocol run, contributing



(a) WiPP and WPaxos Commit Comparison (b) WiPP and WPaxos Abort Comparison

Figure 6.4: Comparison with A^2's WPaxos implementations of Simple and Multi-Paxos.

in different proportions to the cumulative latency measured by the application.

The reliable implementation of consensus protocols WISP and WIMP, which are guaranteed to terminate with high reliability under interference, validates and fulfills **C3**. The use of majority-based voting, together with the Hybrid ST primitive, provides robust consensus guarantees for Wireless Sensor Networks.

6.3 Conclusion

In this chapter we have validated the reliability, latency and broadcast efficiency claims made in this thesis about XPC, WISP and WIMP. The new Hybrid ST primitive has been proven to be robust and resilient to high levels of network interference. Our 2PC, 3PC, WISP and WIMP implementations are able to match the latency of state-of-the-art protocols such as A^2, while providing the same correctness guarantees and liveliness properties. We have also addressed all challenges (**C1**-**C4**) proposed in Section 2.10, showing how the they are solved by either our XPC protocols, or by our repeatable testing methodology.

7 | Conclusion and Future Work

7.1 Conclusion and Contributions

Wireless Sensor Networks links are unreliable. The lack of message delivery guarantees in WSNs makes adoption of consensus protocols impractical, as they do not work in fully asynchronous systems. Synchronous Transmission primitives such as Glossy or Chaos do not provide both the low-latency and high-reliability necessary to improve the robustness of communication. High levels of network interference further impact the reliability of individually used ST primitives. In Section 2.10 we identified four challenges (**C1**-**C4**) which remain unsolved in the current literature, and address the problems present in state-of-the-art publications.

In Chapter 3 we introduced Hybrid, a new approach to Synchronous Transmissions which leverages the optimal latency of Chaos and optimises it with Glossy's reliability. Hybrid's low-latency and robust dissemination guarantees solves the reliability challenges expressed by **C1**. To allow for the Hybrid ST primitive implementation we developed XPC. XPC is a novel, highly configurable, multi-phase voting library which extends Baloo by enabling protocols to easily switch between ST primitives during their execution. The implementation of XPC addresses and solves **C2**.

In Chapter 4 we proposed WISP, a Paxos-based consensus protocol for Wireless Sensor Networks which uses the Hybrid ST primitive. To support majority voting protocols XPC was extended with the Wireless Part-time Parliament. WISP uses WiPP, together with Hybrid dissemination rounds, to provide a correct and safe solution for distributed consensus. By relying on the reliability of Hybrid, WISP fulfills the requirements set by **C3**.

In Chapter 5 we presented WIMP, a reliable Multi-Paxos implementation which uses the Hybrid ST primitive and supports network-wide value proposal. To allow for multiple nodes to propose values during the execution of the protocol we extend XPC to support multiple proposers. WIMP executes back-to-back consensus rounds to vote on all values submitted by the cohort. This extends the guarantees of **C3** by providing strong liveliness and safety guarantees to all network proposals.

We evaluated Hybrid, WISP and WIMP in Chapter 6, to validate the reliability and latency claims made in this thesis. All testing was performed on Flocklab and injected interference was generated with JamLab. This satisfied **C4** by providing reproducible results which can be compared to current research publications. The Hybrid ST primitive is shown to be robust and reliable under interference across all protocols. Additionally our XPC implementations of 2PC, 3PC, WISP and WIMP match the latencies of state of the art protocols in literature: A^2 and WPaxos.

In this thesis we made 3 main contributions to the WSN community: Hybrid, WISP and WIMP. Our implementations address challenges which are currently unsolved in the literature (**C1**-**C4**), and our code is packaged into XPC, an all-in-one compact library with working examples for each proposed feature. We aim to submit this work for publication to IEEE

INFOCOM 2020 and keep maintaining and extending its functionality over the years.

7.2 Limitations

In this thesis we have made three main contributions to the WSN community. In this section we address the limitations of our work that would have been address if we had more time.

- **Formal Analysis**. Our protocols lack a formal analysis of their implementations. We could have used formal methods such as process algebra (π-calculus), abstract state machine (ASM), or high level languages such as TLA+ to model our concurrent distributed system. This would allow us to provide a formal specification and formally check the correctness of our proposals.
- **Comparison with $\mathbf{A^2}$**. The A^2 codebase is tightly coupled and hard to customise. This prevented us from being able to execute interference comparison tests using JamLab, as we had to rely on the published latency and reliability results. Being able to compare Chaos-based WPaxos with Hybrid-based WISP would have given great insight on the reliability and robustness improvements provided by the Hybrid ST primitive.
- **Cross-testbed evaluation**. Online testbeds are changing and TelosB hardware is being removed. Currently the single largest openly-accessible deployment of TelosB motes is Indrya2 (with 74 motes), but constant hardware failures and testbed unavailability prevent it from being a reliable means of evaluation. The results of this thesis have therefore all been gathered on FlockLab with 27 nodes.

7.3 Future Work

Designing new features for WSN protocols takes significant investigation work. They require knowledge of the hardware and may need alterations to the underlying kernel used by the motes. All modifications must be thoroughly evaluated in simulation and run on testbeds, to prevent impacts on existing features. In this chapter, we present extensions to XPC we have designed, but not been able to implement, due to time constraints.

Group Membership

Consensus protocols require knowledge of the cohort which is targeted during communication rounds. Due to interference certain communication links may become unreliable making certain nodes unreachable. To prevent protocols from stalling during global dissemination rounds, as unreachable nodes prevent them from making progress, we propose group membership. With group membership the network's cohort is dynamic. Straggler nodes will be removed form the cohort when the round leader detects they no longer reliably reply. Nodes removed from the cohort will no longer have assigned slots or payload sections to send their replies in. To be readmitted into the cohort special contention slots will be occasionally scheduled by the host, during which excluded nodes may propose themselves for readmission. After a number of protocol runs, once the node is deemed to be reliable again, it will be reintroduced as part of the cohort.

Dynamic Host

Currently the Baloo host is static and defined at compile time by the applications using XPC. This can hinder the reliability of protocols, as all progress is stalled when the host becomes unreachable or fails. With Baloo support for dynamic host allocation, XPC may execute leader election rounds to change the node allocated as host depending on the network's needs. Should the original host become unreachable, individual nodes may self-elect themselves as leaders, and eventually one of them will be acknowledged by a majority of the cohort resuming the protocol execution.

Impact of Configuration Parameters on Reliability under Interference

When evaluating XPC protocols we have chosen default values for its numerous configuration parameters. Varying maximum retransmission values, Chaos and M-Slot round lengths or ΔQ sizes will have different impacts on different testbeds (as node density and network sizes vary). To achieve optimal latency and reliability under interference we should thoroughly assess the impact of each individual configuration parameter on the transaction outcome and overall execution time. Being able to empirically analyse the impact of each configuration parameter across varying testbed topologies may be used to train new algorithms to provide optimal configurations for new unseen networks.

XPC as a Service

Applications currently using XPC are currently bound to adopting one protocol at a time. WISP protocol runs cannot, for example, be followed up by the execution of 2PC. To mitigate this, XPC could be offered as a protocol scheduling service. The host application is in charge of choosing protocols for each round. When required to vote, individual nodes will be aware of the value proposed by the host as well as the protocol being used, making decisions accordingly. Providing XPC as a service would allow for protocols, such as group membership or leader election, to be interleaved with the execution of XPC voting rounds, providing stronger reliability and safety guarantees.

Contribution to Contiki-NG and Baloo on GitHub

Once published we aim to contribute XPC to the open source Baloo and Contiki-NG GitHub repositories. By making our code open source, pushing into an actively developed IoT kernel, we can determine which features should be prioritised due to increased interest or demand from the community. We believe that the Hybrid ST primitive, and all protocols present within XPC, will be important enablers for a new class of reliable Internet of Things applications.

Bibliography

[1] Beshr Al Nahas. *A2-Synchotron: Network-wide Consensus Utilizing the Capture Effect in Low-power Wireless Networks*. `https://github.com/iot-chalmers/a2-synchrotron`. (Visited on Jan. 25, 2019).

[2] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. "Network-wide Consensus Utilizing the Capture Effect in Low-power Wireless Networks". In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. SenSys '17. ACM, 2017, 1:1–1:14.

[3] Beshr Al Nahas, Simon Duquennoy, and Olaf Landsiedel. "Poster: Network Bootstrapping and Leader Election Utilizing the Capture EEect in Low-power Wireless Networks". In: *ACM SenSys 2017-15th ACM International Conference on Embedded Networked Sensor Systems*. 2017, pp. 1–2.

[4] Paramasiven Appavoo et al. "Indriya2: A Heterogeneous Wireless Sensor Network (WSN) Testbed". In: 2019, pp. 3–19.

[5] Jonas Bächli. "Creating a Flexible Middleware for Low-Power Flooding Protocols". MA thesis. ETH Zürich, 2018.

[6] C. A. Boano et al. "JamLab: Augmenting sensornet testbeds with realistic and controlled interference generation". In: *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*. Apr. 2011, pp. 175–186.

[7] Thang Chien, Hung Nguyen Chan, and Nguyen Thanh. "A comparative study on operating system for Wireless Sensor Networks". In: *Advanced Computer Science and Information System (ICACSIS)*. Jan. 2011.

[8] Moteiv Corporation. *Tmote Sky: Low Power Wireless Sensor Module*. `https://fccid.io/TOQTMOTESKY/User-Manual/Users-Manual-Revised-613136`. (Visited on Jan. 13, 2019).

[9] G. Coulouris et al. *Distributed Systems: Concepts and Design*. 5th ed. Addison-Wesley Publishing Company, 2011. Chap. 15 and 17.

[10] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong. "Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 269–282.

[11] Wan Du et al. "Pando: Fountain-Enabled Fast Data Dissemination With Constructive Interference". In: *IEEE/ACM Trans. Netw.* 25.2 (Apr. 2017), pp. 820–833.

[12] Adam Dunkels. "uIP - A Free Small TCP/IP Stack". 2002. URL: `https://github.com/adamdunkels/uip` (visited on Jan. 25, 2019).

[13] Joakim Eriksson et al. "COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks". In: *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*. Simutools '09. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, 27:1–27:7.

[14] Federico Ferrari et al. "Low-power Wireless Bus". In: *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*. SenSys '12. ACM, 2012, pp. 1–14.

[15] F. Ferrari et al. "Efficient network flooding and time synchronization with Glossy". In: *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*. 2011, pp. 73–84.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* (1985), pp. 374–382.

[17] Jim Gray and Leslie Lamport. "Consensus on Transaction Commit". In: *ACM Trans. Database Syst.* 31.1 (Mar. 2006), pp. 133–160.

[18] Carsten Herrmann, Fabian Mager, and Marco Zimmerling. "Mixer: Efficient Many-to-All Broadcast in Dynamic Wireless Mesh Networks". In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. SenSys '18. Shenzhen, China: ACM, 2018, pp. 145–158.

[19] "IEEE Standard for Low-Rate Wireless Networks". In: *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)* (2016), pp. 1–709.

[20] Timofei Istomin et al. "Data Prediction + Synchronous Transmissions = Ultra-low Power Wireless Sensor Networks". In: *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems*. SenSys '16. Stanford, CA, USA: ACM, 2016, pp. 83–95.

[21] Romain Jacob et al. "Synchronous Transmissions made easy: Design your network stack with Baloo". In: *16th International Conference on Embedded Wireless Systems and Networks (EWSN 2019)*. 2019.

[22] Wen-Kuang Kuo and C. -. J. Kuo. "Enhanced backoff scheme in CSMA/CA for IEEE 802.11". In: *2003 IEEE 58th Vehicular Technology Conference. VTC 2003-Fall.* 2003.

[23] Leslie Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pp. 51–58.

[24] Leslie Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169.

[25] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. "Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale". In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. SenSys '13. 2013.

[26] Roman Lim et al. "Distributed and synchronized measurements with FlockLab". In: (2012).

[27] R. Lim et al. "FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems". In: *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '13. 2013, pp. 153–165.

[28] Angus Macdonald. "Paxos by Example". 2012. URL: `https://angus.nyc/2012/paxos-by-example` (visited on Jan. 25, 2019).

[29] Jan Mueller et al. "Competition: Keep It Simple, Let Flooding Shine." In: *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*. EWSN '19. Beijing, China: Junction Publishing, 2019, pp. 294–295.

[30] Mattias Nilsen and André Samuelsson. "Bringing Order to Chaos". MA thesis. Chalmers University of Technology, 2018.

[31] George Oikonomou et al. *Contiki-NG: The OS for Next Generation IoT Devices*. `https://github.com/contiki-ng/contiki-ng`. (Visited on Jan. 25, 2019).

[32] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. "Paxos Made Wireless: Consensus in the Air". In: *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*. EWSN '19. 2019, pp. 1–12.

[33] Valentin Poirot, Beshr Al Nahas, and Olaf Landsiedel. *Paxos Made Wireless: Consensus in the Air*. `https://github.com/iot-chalmers/wireless-paxos`. (Visited on May 29, 2019).

[34] Michel Raynal. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool Publishers, 2010.

[35] C. Sarkar et al. "Sleeping Beauty: Efficient Communication for Node Scheduling". In: *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems*. MASS '16. 2016, pp. 56–64.

[36] Thomas Schmid, Prabal Dutta, and Mani B. Srivastava. "High-Resolution, Low-Power Time Synchronization an Oxymoron No More". In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '10. 2010, pp. 151–161.

[37] Tadahiro Sekimoto and Jo Puente. "A satellite time-division multiple-access experiment". In: *IEEE Transactions on Communication Technology* 16.4 (1968), pp. 581–588.

[38] Dale Skeen. "Nonblocking Commit Protocols". In: *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. SIGMOD '81. 1981, pp. 133–142.

[39] Dale Skeen. "Nonblocking commit protocols". In: *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. ACM. 1981, pp. 133–142.

[40] L. P. Steyn and G. P. Hancke. "A survey of Wireless Sensor Network testbeds". In: *IEEE Africon '11*. 2011.

[41] Ed. T. Winter. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. 2012. URL: `https://tools.ietf.org/html/rfc6550` (visited on Jan. 25, 2019).

[42] R. Tavakoli et al. "Enhanced Time-Slotted Channel Hopping in WSNs Using Non-intrusive Channel-Quality Estimation". In: *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*. MASS '15. 2015, pp. 217–225.

[43] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication*. New York, NY, USA: Cambridge University Press, 2005. ISBN: 0-5218-4527-0.

[44] Robbert Van Renesse and Deniz Altinbuken. "Paxos Made Moderately Complex". In: *ACM Comput. Surv.* (Feb. 2015).

[45] K. Whitehouse et al. "Exploiting the capture effect for collision detection and recovery". In: *The Second IEEE Workshop on Embedded Networked Sensors, 2005. EmNetS-II.* May 2005.