2021-03

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED LEDGER TO SUPPORT DATA SURVIVABILITY IN AN UNMANNED MULTI-VEHICLE SYSTEM

Pommer, Peter J.

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED LEDGER TO SUPPORT DATA SURVIVABILITY IN AN UNMANNED MULTI-VEHICLE SYSTEM

by

Peter J. Pommer

March 2021

Thesis Advisor: Cynthia E. Irvine
Co-Advisor: Duane T. Davis

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE DESIGN AND IMPLEMENTATION OF A DISTRIBUTED LEDGER TO SUPPORT DATA SURVIVABILITY IN AN UNMANNED MULTI-VEHICLE SYSTEM	**5. FUNDING NUMBERS**
6. AUTHOR(S) Peter J. Pommer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Autonomous vehicle systems, including multi-vehicle systems, are becoming increasingly relevant in military operations. A problem emerges, however, when logging data within these systems. In particular, loss of individual vehicles and inherently lossy and noisy communications environments can result in the loss of important mission data. This thesis presents a novel distributed ledger protocol that can be used to ensure that the data in such a system survives. To test the efficacy of the protocol, we implemented it as a Robot Operating System (ROS) node on the Advanced Robotic Systems Engineering Laboratory (ARSENL) aerial swarm system. Results are presented for implementation tests in the ARSENL software-in-the-loop simulation environment and during live-flight field experiments conducted at Camp Roberts, CA.

14. SUBJECT TERMS distributed ledger, blockchain, unmanned aerial vehicle, UAV, unmanned vehicle system, UVS, Advanced Robotic Systems Engineering Laboratory, ARSENL	**15. NUMBER OF PAGES** 113
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED LEDGER TO SUPPORT DATA SURVIVABILITY IN AN UNMANNED MULTI-VEHICLE SYSTEM

Peter J. Pommer
Civilian, CyberCorps: Scholarship for Service
BS, California State University Monterey Bay, 2018

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2021

Approved by: Cynthia E. Irvine
Advisor

Duane T. Davis
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Autonomous vehicle systems, including multi-vehicle systems, are becoming increasingly relevant in military operations. A problem emerges, however, when logging data within these systems. In particular, loss of individual vehicles and inherently lossy and noisy communications environments can result in the loss of important mission data. This thesis presents a novel distributed ledger protocol that can be used to ensure that the data in such a system survives. To test the efficacy of the protocol, we implemented it as a Robot Operating System (ROS) node on the Advanced Robotic Systems Engineering Laboratory (ARSENL) aerial swarm system. Results are presented for implementation tests in the ARSENL software-in-the-loop simulation environment and during live-flight field experiments conducted at Camp Roberts, CA.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Acronyms and Abbreviations

API	application programming interface
ARSENL	Advanced Robotic Systems Engineering Laboratory
DLP	data ledger protocol
DOD	Department of Defense
MP	Monterey Phoenix
NPS	Naval Postgraduate School
ROS	Robot Operating System
SITL	software-in-the-loop
UAV	unmanned aerial vehicle
UCP	Uniform Chain Protocol
UDP	User Datagram Protocol
UV	unmanned vehicle
UVS	unmanned vehicle system

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

This material is based upon activities supported by the National Science Foundation under Agreement No 1565443. Any opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Thank you, Dr. Cynthia Irvine and Dr. Duane Davis, for being great thesis advisors and always making yourselves available.

I want to offer a special thanks to Nickolas Carter for working with me on this project and staying up late nights helping me write this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Distributed unmanned vehicle systems often work without a centralized authority that directs all of the participants. This distribution provides for scalability and flexibility but presents challenges with regard to event logging and mission analysis. These challenges can be addressed through distributed ledgers that use block chains to store data. In a distributed ledger, data is synchronized across multiple physical locations, and consensus processes are used to ensure the consistent state of the ledger across the entire network [1]. We present a novel protocol to create and maintain such a distributed ledger with improved data survivability in a multi-vehicle system that suffers participant attrition or lossy communications.

1.1 Motivation

This research is inspired by the 2018 *National Defense Strategy* (NDS) which promotes investment in the cyberspace domain. This strategy states, "We will also invest in cyber defense, resilience, and the continued integration of cyber capabilities into the full spectrum of military operations" [2]. The protocol outlined in this thesis addresses the resilience of data generated and recorded during missions of distributed, multi-vehicle, autonomous systems.

1.2 Problem Statement

Data availability in an unmanned vehicle system (UVS) can be compromised or hindered when an individual unmanned vehicle (UV) is lost or destroyed. When data generated by a particular UV is stored on only that UV, the availability of that data is imperiled because the UV must survive the mission and return to its base for the data to be retrieved. If the UV is lost or destroyed during the mission, all of its data perishes with it. In addition, vehicles may experience lossy and sometimes disconnected inter-vehicle communications that further hinder mission data collection and maintenance. Thus, a robust, system-wide method to record data generated by individual agents that is tolerant of vehicle attrition and communication discontinuities is needed. This thesis presents a potential solution and

addresses the following questions:

1. Can a block-chain-based data ledger protocol (DLP) be used to distribute mission data across the system so that individual vehicle losses do not hinder post-mission data consolidation and analysis?
2. Can this DLP account for lossy and occasionally disconnected communications environments?
3. Can this DLP be implemented in a real-world system without impacting the system's functionality?
4. What can empirical analysis of real-world system results tell us about the DLP's performance?

1.3 Scope

This thesis presents a protocol that provides a solution to the problem of availability of generated and recorded sensor data by using a block-chain-based protocol to propagate data to all other UVs in the UVS. The protocol accounts for vehicle loss and accommodates periods of temporary disconnection in inter-vehicle communications. In addition, this thesis describes the implementation and testing of the protocol on the Advanced Robotic Systems Engineering Laboratory (ARSENL) unmanned vehicle system (UVS) platform. Experimental results are empirically analyzed to verify compliance with a set of required characteristics; however, formal mathematical proof of correctness is left to future work.

1.4 Approach

Our approach for developing the protocol began with a study of existing distributed ledgers. We also reviewed well known consensus algorithms, such as the Paxos family of protocols, and researched background information on unmanned aerial vehicle (UAV) systems and challenges associated with inter-vehicle communications. Based on the envisioned target system environment, we then developed a set of required protocol properties. Following the literature review and requirements development, Nickolas Carter, my classmate at Naval Postgraduate School (NPS), and I jointly defined a protocol for use in a distributed, multi-vehicle, autonomous system to improve availability of data in such a system. The protocol was defined as a language-independent and system-independent set of event-handler func-

tions. The implementation was targeted for deployment on the ARSENL swarm system, so I used the Robot Operating System (ROS) Python application programming interface (API) to create new software components. We tested the implementation using a software-in-the-loop (SITL) framework to run on-vehicle software in a realistic physically-based simulation environment. Additionally, I analyzed log files to discover bugs and ensure the protocol executed properly. Finally, we performed live-fly field tests at Camp Roberts, CA, that demonstrated the correctness of the protocol running on ARSENL UAVs.

1.5 Organization

This work is organized into five chapters, the first of which is this introduction. Chapter 2 describes foundational constructs necessary for our protocol and includes discussions of distributed multi-vehicle autonomous systems, consensus in distributed systems, distributed ledgers and block chains, ROS, the ARSENL UVS and on-aircraft architecture, and the SITL simulation software.

Chapter 3, the central element of this work, formally defines the protocol. It begins with a summary, definitions, assumptions and an explanation of how we use a block chain-based approach to achieve our objectives. The chapter also provides a high level explanation of the protocol before delving into a detailed explanation of each event handler. A pseudo-code algorithm and a process flow diagram is provided for each event handler.

Chapter 4 covers the development process of the event handlers required to implement the protocol on the ARSENL platforms. This chapter also describes the testing process, the results of bench tests, and the results of live-fly field experiments. Chapter 5, the conclusion, summarizes the contributions and implications of this research and provides suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter outlines the foundations for this thesis and covers concepts necessary for the development of the protocol. These include distributed multi-vehicle autonomous systems, consensus within a distributed system, distributed ledgers, and block chains. It also introduces technologies used to implement the protocol including ROS, the publish-subscribe mechanism, the ARSENL multi-UAV system, and the SITL simulator.

2.1 Distributed Multi-Vehicle Autonomous Systems

Distributed multi-vehicle autonomous systems, as the name suggests, are systems in which multiple vehicles operate as a distributed system. Given appropriate communication and coordination capabilities, a collection of autonomous vehicles can comprise what is commonly known as a UVS. These systems can incorporate land-, air-, water-, and space-based vehicles. Furthermore, these systems can be homogeneous (i.e., comprised of a single type of vehicle) or heterogeneous (i.e., comprised of dissimilar vehicles that may operate in different domains).

2.1.1 Multi-Vehicle Systems Overview

Distributed multi-vehicle autonomous systems are decentralized by nature. Each vehicle makes its own choices regarding the actions it takes, and there is no centralized control platform piloting every vehicle. Each vehicle also maintains its own sensory data and assessment of the situation. That is, each understands its environment to some degree based on its own observations and information received from other vehicles. Vehicles may share aspects of their awareness with other vehicles in the UVS as required to support collective decision-making and coordination. This local awareness, however, solely informs each vehicle's decisions and actions.

What is commonly referred to as "swarming" is multi-UVS behavior that occurs when relatively simple UVs operate collectively without relying on any form of centralized control or an external infrastructure [3]. One characteristic that differentiates swarms from other

multi-unmanned vehicle systems (UVSs) is that a UV's behavior results from interactions with its local environment and other UVs in the system [4]. That is, swarm participants typically self-organize in a reactive manner using local sensing and local communications. Swarms, therefore, are a subset of distributed multi-vehicle autonomous systems where simple UVs are able to achieve complicated behavior through local interactions.

This research specifically focuses on aerial UVSs in general and swarming systems in particular, but the concepts developed in this work can be applied to any vehicle type.

Distributed multi-vehicle autonomous systems encounter distinct challenges in the course of their operations. These include close range maneuverability ensuring that participating vehicles do not collide with each other [5]; cooperation and task assignment among participants to enable mission planning and mission execution [6]; computational complexity, which is concerned with time and space requirements of the algorithms a vehicle executes [7]; communication implementations that ensure the inter-vehicle network can handle the expected network load [7]; and communication robustness, which is concerned with how much degradation of the communication network is tolerated before vehicle's algorithms fail [8].

2.1.2 Communications within Multi-Vehicle Systems

Communications within multi-vehicle systems is among the most important issues associated with the development and employment of these systems. Limited bandwidth, system dynamicity, and inherently noisy communications channels can impact both the latency and reliability of inter-vehicle communications [9]. These challenges are particularly germane to this research.

It is assumed that system UAVs will not always be within communication range of each other. The swarm might temporarily split into smaller subswarms that diverge to different locations only to subsequently recombine. Alternatively, individual UAVs might detach from the rest of the UAV swarm. These situations result in a temporarily disjoint network where not all UAVs are able to communicate with some subset of, or the entire, UVS. In addition, communications within a large multi-UVS is inherently unreliable, so even messages transmitted within a connected swarm may not be received by all participants [5]. The UAVs with which this research is concerned perform military missions, thus the loss

or destruction of individual UVs can also be expected. Because of this, our protocol must be robust in the face of a disjoint or unreliable network. Further, the protocol must be able to recover and merge disparate mission logs when multiple swarms that have collected different data merge.

2.2 Consensus within Distributed Systems

In the distributed system all participants are expected to maintain consistent information. That is, the system as a whole must achieve some level of agreement, or consensus across the distributed system [10]. As a formal notion, consensus is a process exhibiting the following properties by which distributed nodes reach agreement: [11]

1. All correct agents decide some value.
2. If all agents propose a value, v, then all correct agents will decide v.
3. If an agent decides v, then some agent must have proposed v.
4. All correct agents agree on the same value. (We relax this to "must decide on consistent values" in this thesis.)

The goal of consensus, then, is to bring a system's distributed agents into agreement regarding the system's stored data [12] in support of reliable decision making or record keeping. Consensus in a distributed system can be difficult to achieve, because multiple participants may propose values concurrently, messages may be lost between participants, and some agents may be unavailable to participate in the consensus process.

This thesis set out to develop a protocol which brings a distributed multi-vehicle autonomous system into agreement regarding a series of asynchronously generated events. One consensus algorithm that is closely aligned with the nature of this particular problem is Paxos. Paxos is of interest because it deals with a transactional notion of consensus [10], [13]. That is, Paxos is designed to allow the distributed system to agree on a series of transactions or events proposed by individual agents rather than on a single value to which each agent contributes. The algorithm developed over the course of this research builds on Paxos's notion of transactional consensus.

Paxos was developed to work in an environment with unreliable communications [10], a feature particularly relevant to our work. Participants in Paxos assume some or all of three

roles: [13]

- *Proposers* who propose values to be added;
- *Acceptors* who accept or ignore proposed values; and
- *Learners* who learn of the accepted values.

Paxos uses these three roles to create consensus through the following steps: [13]

1. A *proposer* sends a $prepare$ request with a proposal number n to all *acceptors*.
2. *Acceptors* respond to a $prepare$ request if the proposal number n is greater than any previously accepted request and promises to not accept a smaller n should it receive one in the future, otherwise the *acceptors* ignore the request.
3. The *proposer* sends an $accept$ request if it received responses from the majority of *acceptors*, otherwise the *proposer* starts over with a higher proposal number n.
4. An *acceptor* will adopt an $accept$ request if it has not already agreed to a $prepare$ request with a larger n. When an *acceptor* adopts an $accept$ request it also informs the *learners* of the accepted values.

A number of modifications to the basic Paxos algorithm have been devised over the years to improve efficiency and realize optimizations [13].

For the purpose of the protocol described in this thesis, all participants in a UVS take on all three Paxos roles.

Paxos relies on recognizing a 51 percent majority of *acceptors* to accept a proposal, meaning that the number of participating agents must be known [10]. Unfortunately, UVSs of the sort with which this research deals do not typically foster universal awareness of the number of agents in the system. Because of this, the proposed protocol diverges from Paxos in that it relies on a plurality or local majority of participants' votes to make decisions rather than on a global majority. This difference potentially results in different values being agreed to by disjoint components of the system, a situation for which Paxos also does not account. Our protocol addresses this by allowing for the deconstruction of the record and reordering of previously accepted values.

Due to these differences the proposed protocol must be considered Paxos-like, as consensus can only be obtained in local communications range. System-wide consensus is only

guaranteed in a fully-connected system. Values agreed to by disjoint or partially connected subcomponents are, however, consistent according to formal rules laid out in Chapter 3.

2.3 Distributed Ledgers and Block Chains

A distributed ledger is a formally organized compilation of information that is shared, replicated, and spread among multiple locations [14]. A distributed ledger is typically stored and maintained using a decentralized, peer-to-peer framework wherein all nodes participate equally in the creation and storage of the entire ledger [15].

Distributed ledgers are commonly used to maintain discrete record information, such as financial records [15]. Most documented distributed ledger (and block chain) examples are related to cryptocurrencies and their transaction data [15], but there is nothing preventing the storage of general purpose data.

A distributed ledger is exactly what this thesis aims to design and implement. In particular, the proposed protocol creates a distributed ledger system that achieves local consensus of stored operational event data. One commonly utilized method of accomplishing this is through the generation of a cryptographically linked chain of entries that are duplicated on multiple nodes [15]. This sort of data structure is referred to as a block chain [16].

The key technology behind block chains are cryptographic hash functions. A cryptographic hash function is a non-reversible mathematical function that takes an arbitrarily sized block of data as input and produce fixed length output [17]. The output of a hash function concisely represents the input data. The output is referred to as a hash digest and can be thought of as a "fingerprint" of the input data [18].

A cryptographic hash function, $\mathcal{H}$, has the following formal properties: [19]

- Preimage resistance: it is computationally infeasible to determine the input of a hash function from its output (i.e., given some y where $y = \mathcal{H}(x)$ it is difficult to derive input x from output y).
- Second preimage resistance: with some first input x it is computationally infeasible to find a partner input x' such that the hash of the two inputs are the same (i.e., given some first input x it is difficult to find a partner input x' where $\mathcal{H}(x) = \mathcal{H}(x')$).

- Collision resistance: it is computationally infeasible to find two unique inputs which hash to the same output (i.e., it is difficult to find two distinct inputs, x and x', where $\mathcal{H}(x) = \mathcal{H}(x')$.

A block chain is a data structure comprised of a series of connected data blocks. Each block stores its own data, and blocks are chained together using cryptographic hash digests [16]. Each block is associated with the previous block by inclusion of the previous block's hash digest in its own hash digest computation [20]. Because modification of a block's data would change not only its own hash digest, but that of all successor blocks, modification or corruption of data in any block in the chain is easily detected and isolated. This allows a block chain to provide some notion of integrity of the data stored in the blocks. A simple block chain is depicted in Figure 2.1.

More importantly for our thesis, if the hashes of the top blocks of two or more block chains are equal, then the likelihood that they contain different data is vanishingly small. This observation arises directly from second preimage resistance: if the hashes of two blocks are the same, then the input to the hash function (i.e., all preceding blocks) must be the same for both blocks. This, and how we use the block chain, is discussed in depth in Section 3.4.

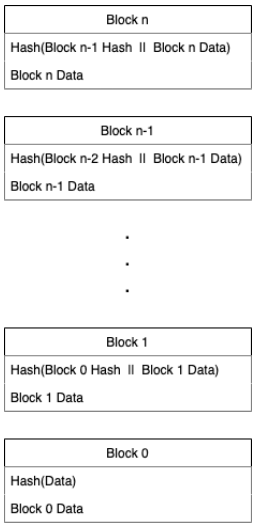

Figure 2.1. A simple block chain example in which blocks store data and a hash digest. Blocks are cryptographically chained together because the previous block's hash digest is concatenated with the current block's data to generate the block hash.

2.4 Robot Operating System (ROS)

This section covers the Robot Operating System (ROS) and its publisher-subscriber inter-process communications model.

2.4.1 ROS Overview

ROS is a middleware system for robotic systems that runs on Linux or Unix. ROS provides a framework and API that allow programmers to write independent plugin-like modules known as ROS nodes [21]. This gives programmers the ability to develop self-contained modules that easily can be integrated into the system, enabled, and disabled. ROS has a large active open source community that has provided a robust set of packages that allows programmers to easily create and run prototypes with minimal boilerplate code [22].

ROS includes formal bindings for the Python and C++ programming languages. The Python binding, referred to as *rospy*, was used in this research. It provides an API for use in developing Python nodes to interact with the rest of the ROS runtime environment [23].

2.4.2 The ROS Inter-Process Communications Model

ROS relies on a publisher-subscriber model for inter-process communication. A publisher-subscriber model is a many-to-many messaging pattern where processes (i.e., ROS nodes) communicate by "publishing" messages to topics to which other processes "subscribe" [24]. A ROS master process globally manages the publisher-subscriber capability through the local network transport layer functionality [24]. The publisher-subscriber model simplifies inter-process communications by isolating processes from one another. A publisher, for instance, is not concerned with what processes are subscribing or how the published data will be used. Similarly, a subscriber is not concerned with the source of the information or how it was generated. The relevant components of the ROS publisher-subscriber implementation are described in more detail in the following sections.

Message Topics

ROS messages are published to named topics. A topic is a string of text similar to a channel or named bus [24], [25]. A topic supports delivery of the single message type that is specified when the topic is created. Only the nodes that register to a topic with the ROS master process

receive the messages published to that topic [24]. Topics can be organized into complex subgroups using namespaces to facilitate organization according to functionality or nodes.

Topic Subscriptions

A node subscribes to a topic to receive messages published to that topic by notifying the ROS master process [26]. Once the subscription is established, all messages published to the topic will be forwarded to the registering node. Multiple nodes can subscribe to the same topic, in which case messages will be forwarded to all subscribers. Nodes can also subscribe to multiple topics. Message processing within a node is handled by a ROS message handling thread that invokes a callback function. A node specifies the event-handling callback function when it subscribes to the topic [27].

Publishing to a Topic

A node declares its intent to publish to a topic by notifying the ROS master of the topic name and the type of message that will be published [26]. If the topic has not yet been established, it will be created by the ROS master process. If the topic already exists, the registering node will be added to the list of nodes publishing to the topic (an error is generated if the message type does not match what has already been established). Once registered, a node publishes messages to a topic at will, for forwarding to all subscribed nodes.

2.5 Advanced Robotic Systems Engineering Laboratory (ARSENL)

The implementation described in this thesis was developed using the ARSENL UVS hardware and software package.

Formed at the Naval Postgraduate School (NPS), the Advanced Robotic Systems Engineering Laboratory (ARSENL) has provided contributions in the research, development, and implementation of multi-UVSs [8], [28]. ARSENL operates a fleet of UAVs that incorporates fixed-wing and quadrotor UAVs and recently developed ground vehicles (ground vehicles were not included in this research). In prior research, ARSENL has developed unique solutions in the realm of software, networking, and human-swarm interaction [8], [28], [29].

Furthermore, ARSENL has demonstrated the logistical infrastructure and technical acumen necessary to conduct field tests with heterogeneous swarms of up to 50 vehicles [8].

The ARSENL Multi-Vehicle System Architecture

The ARSENL system (architecturally depicted in Figure 2.2) includes both UVs and ground-stations. Ground stations are used for human-swarm interactions and provide operators the ability to control the swarm and monitor its performance during experiments. Air-to-air and air-to-ground communications rely on an 802.11n ad hoc network, and emergency override control is available using a standard RC radio [8].

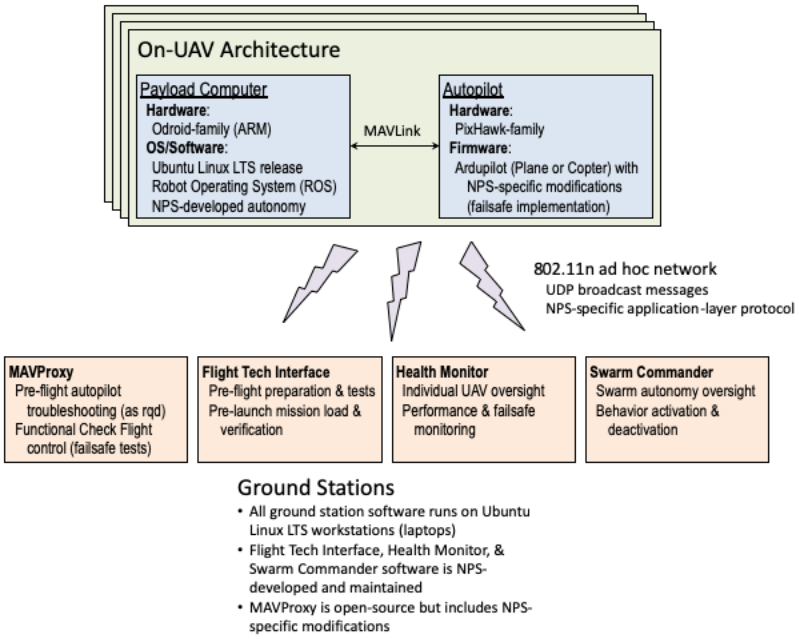

Figure 2.2. The ARSENL UVS architecture provides functionality for monitoring and control of large UV swarms. Source: [29].

Operators utilize ARSENL-developed applications running on Linux workstations to interact with the swarm. Operational interaction with the swarm, once airborne, is separated into two functions: aircraft monitoring and swarm control. Aircraft monitoring is accomplished through the Health Monitor application that provides an interface for monitoring and controlling individual UVs [8]. The Health Monitor is a byproduct of the experimental

nature of ARSENL's research system and is required to meet safety of flight requirements. Swarm control is the task of managing the swarm as a whole and is accomplished through the Swarm Commander application. This application allows the operator to assign UV to swarms and subswarms, to parameterize and initiate swarm behaviors, and to monitor the performance of the swarm as it executes behaviors [8].

The ZephyrII UAV, pictured in Figure 2.3, is ARSENL's primary fixed-wing UAV. The ZephyrII is based on an off the shelf airframe and incorporates commercially available components from the RC hobby community and ARSENL-designed circuitry and 3D-printed parts [8]. ARSENL also employs the Mosquito Hawk quadrotor UAV, pictured in Figure 2.4, for heterogeneous swarms. The Mosquito Hawk was entirely designed and fabricated by ARSENL. Both platforms utilize a PixHawk-family autopilot and an ODroid payload computer on which swarming functionality is implemented.

Figure 2.3. The ARSENL ZephyrII fixed-wing UAV. Source: [8].

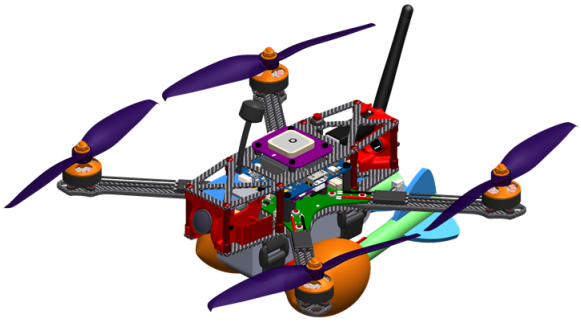

Figure 2.4. The ARSENL Mosquito hawk quadrotor UAV. Source: [29].

UAVs communicate with each other and ground stations via User Datagram Protocol (UDP) broadcast messages transmitted over an 802.11n ad hoc network. Messaging is implemented with an ARSENL-specific application layer protocol. Each UAV transmits state updates (e.g., GPS location, and barometric altitude) at 10 hertz and status updates (e.g., health checks) at two hertz [8]. Ground stations transmit a heartbeat message at one hertz to ensure continuous communications. Asynchronous messages associated with specific swarm behaviors or other functionality can be transmitted by UAVs or ground stations as required.

ARSENL operational experiments consist of multiple stages: pre-flight, ingress, swarm ready, egress, landing, and on-deck [8]. UAVs are first launched and transit through a series of waypoints to a staging location. Upon arrival at the staging waypoint, they transition from the ingress state to a swarm ready state indicating that they are ready for swarm behavior initiation. Once all UAVs reach the swarm ready state, the swarm operator issues commands using the Swarm Commander application to assign UAVs to subswarms and directs subswarms to execute experimental behaviors [8]. Upon completion of the experiment, the swarm operator can direct the entire swarm to land as a group, or the health monitor can take control of and land UAVs individually.

On-Aircraft Architecture

ARSENL's autonomous swarm functionality is implemented as a set of ROS nodes running on each UAV's payload computer. The autonomy package provides commands to the PixHawk autopilot via a serial link. The core of ARSENL's autonomy package as depicted in Figure 2.5 includes the following ROS nodes [8]:

- The *autopilot* node is responsible for communicating with the Pixhawk autopilot, which runs customized ArduPilot software [30], using MAVLink messages [31]. Other ROS nodes access autopilot functionality through message topics published or subscribed to by this node.
- The *network* node is responsible for broadcasting messages to and receiving messages from other UAVs and ground stations over the 802.11n network. Other ROS nodes access the network through message topics published or subscribed to by this node.
- The *swarm_tracker* node is responsible for aggregation and synchronization of position updates received from other system UAVs. Aggregated states are then used during swarm behaviors as required.

- The *swarm_controller* node is responsible for an individual UAV participation in the swarm. Swarm state, swarm behavior activation and deactivation, and vehicle control during behavior execution are all managed by this node. Individual swarm behaviors are incorporated into this node as plugins.
- The *safety* node is responsible for ensuring safety of flight (e.g., preventing inconsistent states, managing failsafe initiation)
- The *collision avoid* node implements deliberative and reactive mechanisms to ensure safe separation between UAVs during the execution of swarm behaviors.

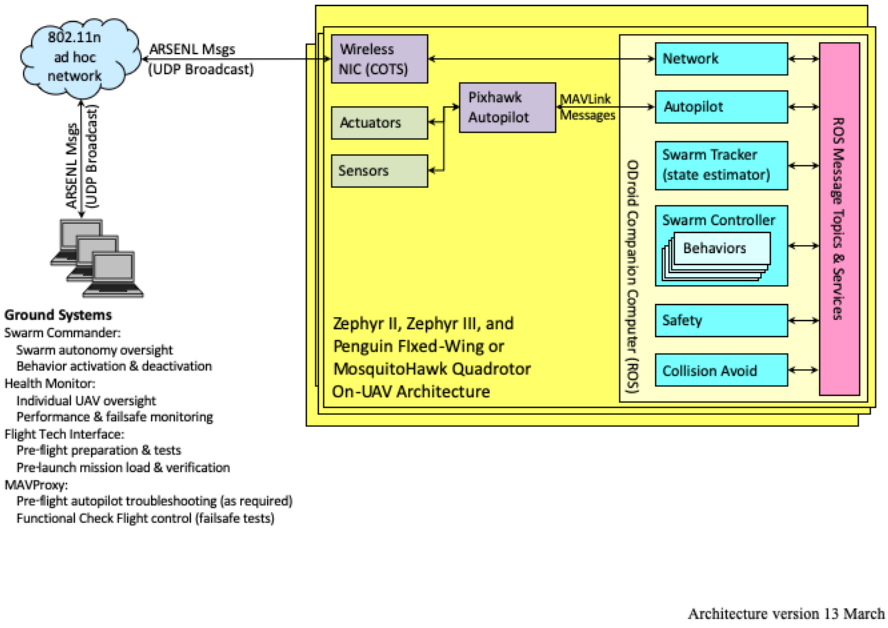

Figure 2.5. The ARSENL on-UAV functionality is implemented as ROS nodes (blue) that utilize the ROS publisher-subscriber mechanism (pink) for inter-node communication. Source: [29].

The protocol implementation documented in this thesis was incorporated into the on-vehicle autonomy architecture. Functionality of the network node was modified by the addition of block chain message topics and network messages that allowed nodes associated with the protocol to interact with the larger system.

Software-in-the-Loop (SITL) Simulation

The open-source ArduPilot software and firmware includes a SITL simulation capability that was leveraged during this research. The SITL simulation environment provides a realistic environment that uses a realistic physically-based vehicle simulation to provide sensor inputs to the autopilot software [30]. This is an important capability that enables thorough testing of the on-vehicle architecture without the overhead of live experimentation. ARSENL-specific improvements to the basic SITL functionality allow testing of all on-board UAV software during multi-vehicle scenarios [28]. In addition, the ARSENL's SITL implementation supports mixed experiments including both live and simulated vehicles.

SITL simulation allows testing of the entire implementation of a UV software package, including interactions between the autopilot and the simulation environment and interactions between vehicles. This testing capability allows developers to test implementations of UV behaviors in a realistic software environment. In particular, this capability enabled the full implementation of the proposed Uniform Chain Protocol (UCP) prior to the first field experiment.

2.6 Chapter Summary

This chapter introduced key concepts and technologies that were used to develop and implement the UCP. Key concepts introduced were distributed multi-vehicle autonomous systems, consensus within a distributed system, and distributed ledgers and block chains. Technologies introduced were ROS and the ARSENL multi-UVS. In Chapter 3, we describe the UCP at both a high level and in detail. The detailed description was directly implemented in the ARSENL on-vehicle architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Uniform Chain Protocol

Nickolas Carter and Peter Pommer, both master's students in the Computer Science program at NPS, co-authored this chapter for their respective theses. This chapter presents an intermediate representation of their jointly developed protocol for a distributed, multi-vehicle UVSs. They have examined the same problem from two different levels of abstraction: Nick in a highly abstract model and Peter in a detailed implementation. [32]

This chapter provides an in-depth description of the Uniform Chain Protocol (UCP). The protocol is designed to ensure the post-mission reliability of UV data from a multi-vehicle UVS that is subject to communications discontinuity and vehicle losses. When properly implemented, the protocol exhibits the following characteristics:

- The protocol is distributed, event driven, and asynchronous. Each event handler is implemented independently on every vehicle, and event handlers can be triggered locally or in response to inter-vehicle messages.
- If a UV's protocol event handlers are in an idle state, then all of its completed `block`s must be committed to its local `block chain`.
- No more than one copy of a particular `block` will be maintained on a UV at any time. A `block` can exist within the locally-maintained `block chain`, within a "waiting to be committed" data structure, or within a data structure associated with the reconcile process.
- No `block` will exist within the block chain or any intermediate data structure that was not generated by a participating UV. This characteristic is partially assured by the UVS's underlying cryptographic system, for now.
- In a fully connected system, all `block`s will eventually be committed to all locally maintained `block chain`s (i.e., blockchains maintained by each UV).
- In a fully connected system where all agents have had the chance to reconcile `block chain`s with each other, one uniform `block chain` will emerge.

3.1 Protocol Summary

The UCP is designed as a set of event handlers, which are executed on each UV in a given UVS. When an event is generated, the UV upon which the event occurred will execute the respective handler for that event. The objective of the UCP is to reliably distribute locally generated information across the UVS to ensure its availability in the event of individual UV losses. Shared data is stored in the form of `block`s that are committed to `block chain`s maintained on each vehicle. `Block`s are committed to the local `block chain`s according to an approval process. Following initiation of the proposal process, associated events will be triggered according to the UCP until all locally maintained `block`s have been committed to the `block chain`, at which point the system will return to an idle state.

Over the course of a mission, one or more UVs may become disconnected from the UVS. If they reconnect at a later time with UVs having `block chain`s that differ, a reconciliation process will eventually be initiated to unify the chains. The reconcile process may occur numerous times throughout a mission. Once a mission is complete, the `block`s contained in the locally maintained `block chain`s can be analyzed in support of mission analysis and reconstruction. The main objective of UCP is to provide the UVS operators access to data that was generated during a mission even if some data-generating UVs do not survive.

3.2 Definitions and Terminology

Throughout this chapter, a number of words and phrases are highlighted with a `teletype` font and are written in camel case in the algorithms. These words and phrases have specific meaning in the context of the UCP and are defined as follows:

`Agent` - This is an individual vehicle in a UVS. For the duration of the chapter, this term is synonymous with the acronym UV.

`Agent ID` - This is a unique identifier for each `agent` in the UVS.

`Block` - This is a data structure that stores data generated by an `agent`'s logging system. It contains the following components:

- `Block data`
- `Block hash`
- `Chain hash`

`Block chain` - This is a locally maintained cryptographic chain that links each

block in the system. A more detailed definition is provided in Section 3.4.

Block data - This term represents the generated data stored in a block. It contains the following components:

- Agent ID of the agent that created the block.
- Event data
- Timestamp at the time the block was created.

Block Hash - This is the hash digest of block data, calculated as

$$BH_n = \mathcal{H}(BD_n) \tag{3.1}$$

where BH_n is the block hash of block n, BD_n is the block data contained in block n and $\mathcal{H}$ is a cryptographic hash function.

Blocks to be committed deque - This is a double-ended queue data structure that stores completed blocks that have not yet been committed to the block chain.

Blocks to be committed deque rlock - This is a re-entrant lock that facilitates safe read and write operations on the blocks to be committed deque.

Chain Hash - This hash digest is stored in a block to cryptographically link it to the previous block. The chain hash is calculated as

$$CH_n = \mathcal{H}(CH_{n-1} \parallel BH_n) \tag{3.2}$$

where $\mathcal{H}$ is a cryptographic hash function, CH_n is the chain hash of the block chain that includes block n, BH_n is the block hash of block n, and CH_{n-1} is the chain hash of the block immediately preceding block n in the block chain.

Chain rlock - This is a re-entrant lock that facilitates safe read and write operations on the local chain.

Current block - This term references the block that is currently being examined or manipulated within an event handler.

Current block being built - This is a data structure that stores locally generated event data that has not yet been incorporated into a block. This data structure is maintained until a "full" block can be generated for addition to the blocks to be committed deque.

Event data - This term refers to loggable data generated by an agent for future storage in the block chain.

Inception block - This is the first block in the block chain. All agents in the UVS share the same inception block.

Local chain - This term is used to reference an agent's locally maintained block chain within an event handler.

Local UVS - This term references the group of agents that are within communications range of the agent when a particular event handler is executed.

Mutual block - This refers to the top block in a mutual chain. The mutual block's chain hash provides verification that all blocks between this block and the inception block are the same.

Mutual chain - This refers to a block chain segment from the inception block to a mutual block that is identical for two or more agents.

Reconcile in progress - This is a Boolean that facilitates atomic operation of the reconcile process (Section 3.5.3). This Boolean stops the agent from responding to external requests.

Reconcile stack - This is a stack data structure that is used to temporarily store blocks removed from the local chain during the reconcile process. These blocks are recommitted to the local chain upon the completion of the reconcile process by the **Reconcile Finalize** event handler.

Re-entrant lock, or **rlock** - This type of lock can be obtained multiple times by a single thread. It must be released the same number of times that it was obtained before another thread can obtain the lock.

Timer - A timer is a computational timing instrument with the following properties:

- The timer counts down from an arbitrary number to zero at a fixed rate.
- The timer can be paused and un-paused.
- The timer generates an event when its count reaches zero.

Top block hash - This is the block hash of the top block on the local chain

Top chain hash - This is the chain hash of the top block on the local chain

3.3 Assumptions

To appropriately scope this research project we made assumptions prior to the design of the UCP:

- The target environment is a distributed system that contains multiple autonomous

vehicles.

- The vehicles operate in a disjoint environment where agents frequently experience network disconnection.
- The system may experience agent casualties. If a casualty occurs, we assume that the casualty will result in the loss of all locally maintained data on that particular agent.
- We have access to a cryptographically secure hash algorithm, which means that we will never encounter a hash collision.
- The platform that the UCP runs on will provide encryption and digital signature services for all network communications and all generated event data. In particular, we assume for now that the underlying cryptographic implementation ensures the authenticity and confidentiality of all data with which the protocol works.
- Byzantine failures are not possible, and all messages are generated and sent in good faith.

3.4 Block Chain Data Structure

In UCP, a block chain data structure is used to ensure that event data is stored in a consistent manner across the UVS. In this section, we describe how this structure is implemented in a way that helps the system reach consensus. By linking data using cryptographic hashes, a block chain can guarantee that if two UVs share the same chain hash in a block, then they share a mutual chain from that block down to the inception block.

In this section, we examine the block chain structure and identify the characteristics that help the UVS attain consensus. The block chain structure is shown in Figure 3.1.

3.4.1 Inception Block

In the UCP, we define inception as the "beginning of the chain." The block chain contains one inception block, which must be the first block in the block chain. The inception block is depicted in Figure 3.1. It possesses the following properties:

- It is loaded onto each agent in the UVS prior to the commencement of operations.
- Each vehicle in the UVS has the exact same inception block.
- It is the only block in the block chain that is created before any event data is generated.

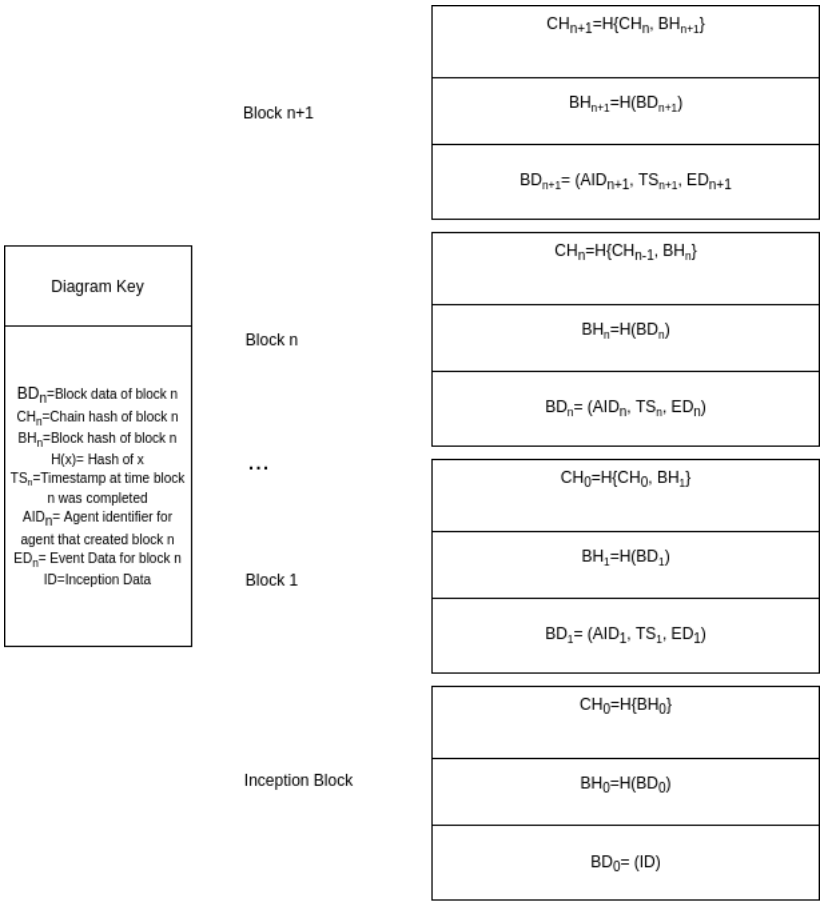

Figure 3.1. This figure is an illustration of the `block chain` data structure.

The `inception block` has the following components:

1. `Block data` - BD_0 - The data contained in this `block` is predefined. The UVS administrator determines the data in BD_0. This data is represented with ID (initial data) in Figure 3.1.
2. `Block hash` - BH_0 - The `block hash` of the `inception block` is a hash of BD_0. It is computed by the following equation:

$$BH_0 = \mathcal{H}(BD_0) \tag{3.3}$$

3. `Chain hash` - CH_0 - The `block hash` is utilized to create the `chain hash`. For the `inception block`, and only the `inception block`, BH_0 is hashed to create CH_0

according to the following equation:

$$CH_0 = \mathcal{H}(BH_0) \tag{3.4}$$

3.4.2 Subsequent Blocks

Each subsequent `block` in the UCP `block chain` has the same components which differ slightly from the corresponding components of the `inception block`. Depicted as `block` 1, `block` n, and `block` $n+1$ in Figure 3.1, these `block`s include the following information:

1. `Block data` - BD_n - This component is a concatenation of the `block`'s three data fields:
   (a) `Agent ID` - AID_n - of the UV that generated the data.
   (b) A timestamp of when the `block` was completed.
   (c) `Event data` - ED_n - in the block.

   To ensure that no two `block`s have the same `block data`, the `Agent ID` of the `agent` that generated each `block` and the time at which it was generated are included in the `block data`. The timestamp component is also used as a tiebreaker in the **Reconcile** (Section 3.5.3) process. The `block data` is computed as follows:

$$BD_n = (AID_n \parallel TS_n \parallel ED_n) \tag{3.5}$$

2. `Block hash` - BH_n - This is the hash of the `block data` of `block` n (BD_n) as calculated by Equation 3.1
3. `Chain hash` - CH_n - This is the hash of the `chain hash` of the preceding block concatenated with the `block hash` of the `current block` as calculated by Equation 3.2. The collision resistance property of the cryptographic hash function ensures that if two `chain hashes` match for a `block` n in two or more distinct `local chain`s, then a `mutual chain` exists from the `inception block` to `block` n.

3.5 High-Level Overview

To present UCP, we start with an abstract overview of the events and sequences that construct the protocol. In Section 3.6, we provide an in-depth description of the event handlers that form UCP. The abstracted components of UCP will be referenced in *italics* for the remainder

of this chapter. These components are: *Generate a Block*, *Commit a Block*, *Reconcile Phase 1*, *Reconcile Phase 2*, *Reconcile Phase 3*, and *Reconcile Finalize*. Each of these components provides functionality that integrates to form the UCP. This high level overview is depicted in Figure 3.2.

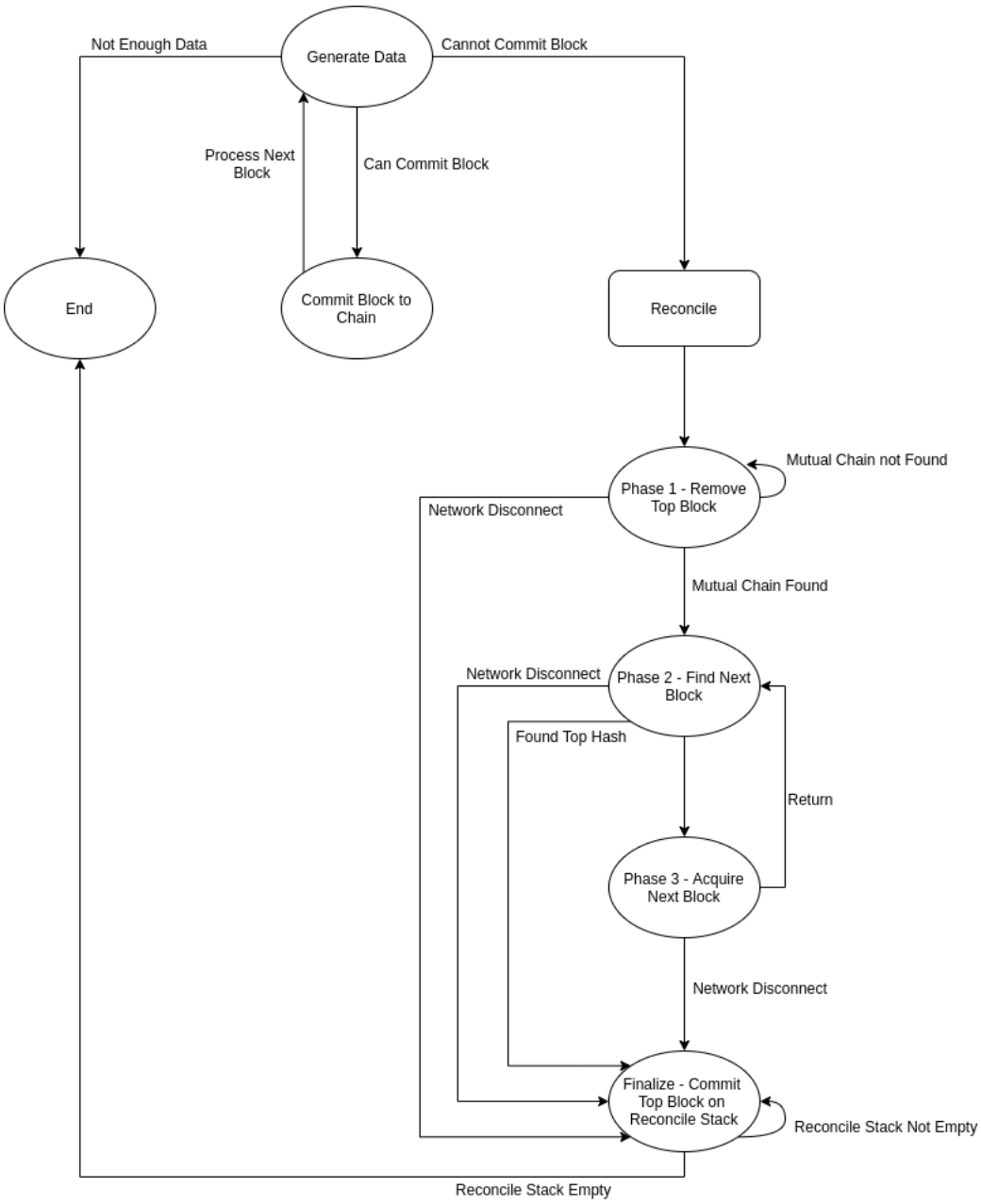

Figure 3.2. An abstracted version of UCP.

3.5.1 Generate a Block

A requirement for this protocol is that it must provide a mechanism for adding data to the system. In this component of the algorithm, data is received from an `agent's` internal system. The data is added to the `current block being built`. Once added the algorithm determines whether or not the `current block being built` contains enough data to commit to the rest of the UVS. The amount of data required to commit a `block` is implementation specific. If the `current block being built` contains enough data, the algorithm attempts to commit it by invoking the *Commit Block* component of the protocol. Otherwise, the algorithm maintains the `current block being built` in its updated form and waits for more data to be generated.

3.5.2 Commit Block

Once a full `block` has been generated and submitted for commit, it must be distributed among `agents` in the system. To do this, this the *Commit Block* component initiates a vote whereby the committing `agent` requests permission to commit a `block` to the `local UVS` by broadcasting the `block hash` and its current `top chain hash`. The `agents` in the `local UVS` individually respond with either an approval or disapproval vote. If the received `top chain hash` matches the receiving `agent`'s `top chain hash`, then the vote shall be for approval. Otherwise, the vote shall be for disapproval. If the majority of the votes are for approval, then the `block` is broadcast to the UVS for commitment to each `agent`'s respective `block chain`. If the majority of votes are not for approval, then the *Reconcile* component is called.

3.5.3 Reconcile

The Reconcile process is initiated to bring an `agent`'s `block chain` back into agreement with the `local UVS`. The process is split into four distinct phases. The first phase finds a `mutual chain` with the `local UVS`. In the second and third phase, the protocol builds upon the `mutual chain` that was established in phase one. In the final phase everything left in the `reconcile stack` (i.e., all `block`s that are not included in the extended `mutual chain`) is committed to the `block chain` and broadcast to the `local UVS` for committing to each `agent`s `block chain`s.

Reconcile Phase 1

In *Reconcile Phase 1* the agent removes blocks from its block chain until a mutual chain is found. *Reconcile Phase 1* takes the following steps:

1. Broadcast a request for votes. These votes are used to determine if the broadcasting agent's top chain hash is contained in the block chain of the agents that received the request. After the vote has been conducted, move to step 2.
2. Tally the votes. If the majority of responses indicate that the top chain hash is not included anywhere in their block chains, move to step 3. If the majority of responses indicate that their block chains include that top chain hash, then the mutual chain upon which to build has been identified, and the agent can move on to *Reconcile Phase 2*. If no votes were received, move on to *Reconcile Finalize*.
3. Remove the top block from the block chain, push it onto the reconcile stack, and repeat step one.

Reconcile Phase 2

In the second and third phases, the protocol builds upon the mutual chain that was established in *Reconcile Phase 1*. In *Reconcile Phase 2*, the agent performs the following steps:

1. Broadcast a vote request to the local UVS for the block on top of the mutual block. Receiving agents will respond with the block hash of the block that is on top of the mutual block in their local block chains. When the vote is complete, move to step 2.
2. Tally the votes. If no responses are received, proceed to *Reconcile Finalize*. If at least one response is received, the block with the most votes is selected to be added to the mutual chain. Proceed to *Reconcile Phase 3* to acquire this block. If the majority response is that the top chain hash is the mutual block, proceed to *Reconcile Finalize*.

Reconcile Phase 3

By the time *Reconcile Phase 3* starts, the system has identified a block to add to the mutual chain by its block hash, but may need to acquire the actual block. This process

is conducted through the following steps:

1. Check the `reconcile stack` to see if the block to be added is present. Proceed to step 2 if the block is present or step 3 if it is not.
2. Remove the `block` from the `reconcile stack` and go to step 4.
3. Broadcast a request for the selected `block` to the `local UVS` and wait for a response carrying the requested `block`. If no response is received, proceed to *Reconcile Finalize*. If a response is received, go to step 4.
4. Commit the `block` to the `block chain`. The newly committed `block` is now considered the `mutual block` and the updated `block chain` is now considered the `mutual chain`. Proceed to *Reconcile Phase 2* to continue the building process.

Reconcile Finalize

In *Reconcile Finalize*, all of the `blocks` that were added to the `reconcile stack` are committed to the `block chain`. The process for this component is as follows:

1. If the `reconcile stack` is empty, then the *Reconcile Finalize* component terminates. Otherwise, proceed to step 2.
2. Pop a `block` off of the `reconcile stack` and proceed to step 3.
3. Commit the `block` to the local `block chain` and broadcast a message to the `local UVS` to commit the `block`. Repeat step 1.

3.6 Detailed Chain Protocol

Figure 3.3 shows the key for symbols used in subsequent diagrams referenced throughout this section. Each event handler is represented using a separate figure. Figures consists of a box with a dotted arrow to signify the entry point of the function. The box consists of the function name, how the function was triggered, a short description of the event, and a list of input data as arguments to that function. There are also network messages broadcast to the `local UVS` which are represented by a cloud. Locks, which are locked and unlocked to facilitate safe concurrency, are represented by a lock image. Blue parallelograms with yellow arrows represent triggering of a local event in an `agent`. Ellipses represent execution of atomic steps. Finally, diamonds represent conditional evaluations (i.e., true or false) with event handling diverging to different branches based on the results.

Each event handler is represented using a separate figure. In the figures, we identify how the event is triggered, provide a short description of the event handler, list what data the handler receives as input, and describe the operations performed within the handler.

While examining the protocol, keep in mind that these event handlers are running on each `agent` in the system. Handlers are triggered by locally generated events, and in response to received network messages as indicated.

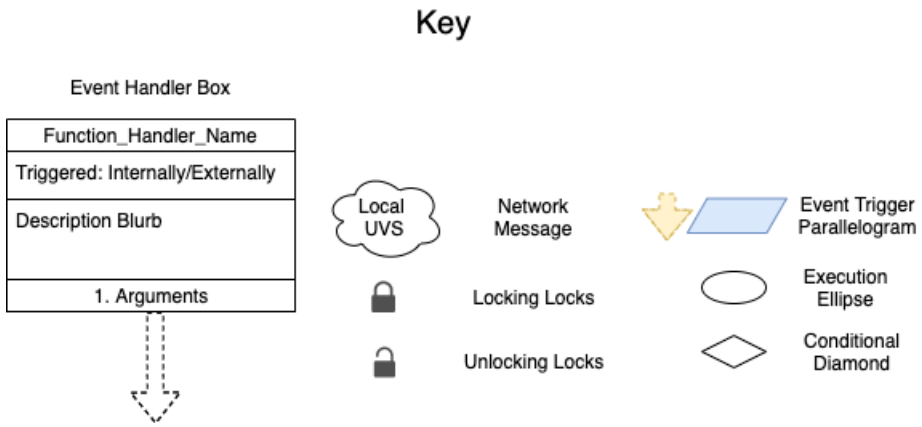

Figure 3.3. Event handler diagram and symbol key for use throughout this section.

Network Messages

Throughout this chapter, there are types of network messages identified in the event handlers. These messages are broadcast to the `local UVS` and handled according to their message type. Likewise, the type of the message determines which event handler parses the message and performs action based on the contents of the message. The message types are defined in the algorithm code and diagrams.

3.6.1 Incorporation of Event Data into Blocks

Build Data Block Event

The **Build Data Block** event handler is triggered internally when a vehicle's logging system generates `event data` to be incorporated into the `block chain`. The purpose of this event handler is to collect and store `event data` in the `current block being built`

until enough `event data` has been generated for the `current block being built` to be considered "full". Once full it is added to the to the `blocks to be committed deque` and a **Process Next Deque Block** event is triggered. The processing of this event is described by Algorithm 1 and depicted graphically in Figure 3.4.

Algorithm 1 Build Data Block Event Handler

$currentBlock.data \leftarrow currentBlock.data + event$
if $Size_{currentBlock} + Size_{maxEvent} \geq Size_{maxBlock}$ **then**
    Obtain *blocksToBeCommittedDequeRLock*
    $currentBlock.blockHash \leftarrow \mathcal{H}(currentBlock.data)$
    Push $currentBlock$ onto bottom of *blocksToBeCommittedDeque*
    Trigger **Process Next Deque Block** event
    Release *blocksToBeCommittedDequeRLock*
end if

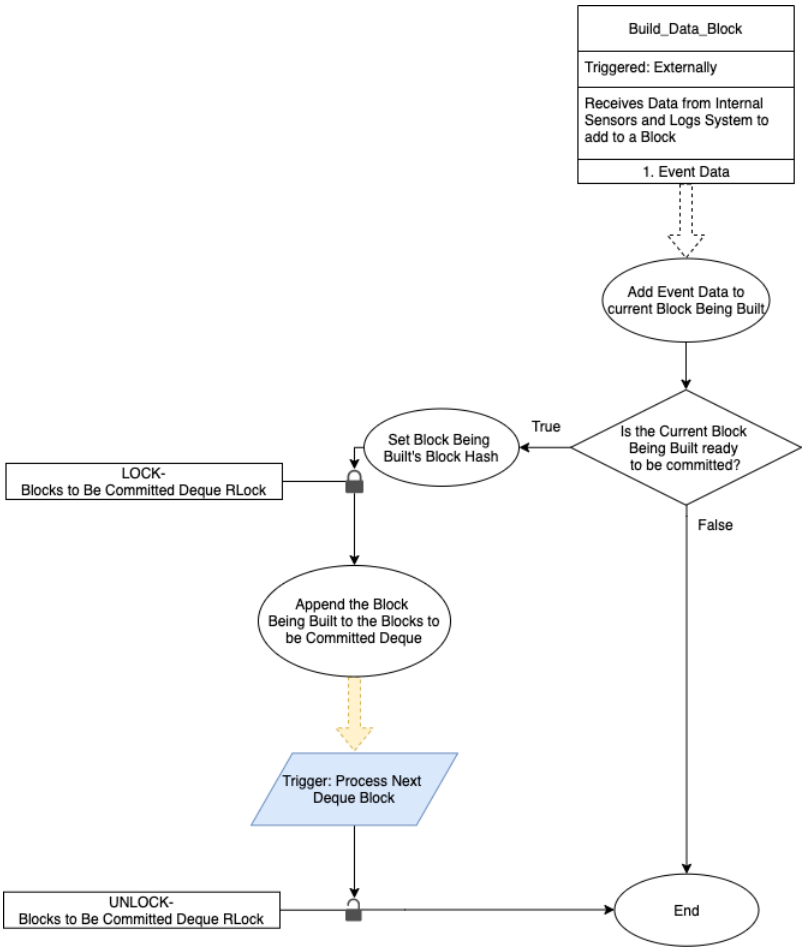

Figure 3.4. The **Build Data Block** event handler is invoked when the system generates loggable data for incorporation into the `block chain`.

3.6.2 The Commit Process

Process Next Deque Block Event

The **Process Next Deque Block** event is triggered internally by multiple event handlers: **Build Data Block**, **Vote Timer Expiration**, and **Reconcile Finalize**. The purpose of this event handler is to initiate the voting process among the `local UVS` to add the first `block` from the `blocks to be committed deque` to the `block chain`. The processing of this event is described by Algorithm 2 and depicted graphically in Figure 3.5.

Algorithm 2 Process Next Deque Block Event Handler

if *reconcileLock* is unlocked and *votingTimer* is not active **then**
  Obtain the *blocksToBeCommittedDequeRLock*
  if *blocksToBeCommittedDeque* is not empty **then**
    Clear *voteCount*
    $voteCount[\text{“}yes\text{”}] \leftarrow 1$
    Obtain the *chainRLock*
    $topChainHashAtInitiation \leftarrow topChainhash$
    $candidateBlock \leftarrow blocksToBeCommittedDeque.front$
    $blockAddRequest.blockHash \leftarrow candidateBlock.blockHash$
    $blockAddRequest.chainHash \leftarrow topChainHash$
    Start the *votingTimer*
    Network send **Block Add Request**
    Release the *chainRLock*
  end if
  Release the *blocksToBeCommittedDequeRLock*
end if

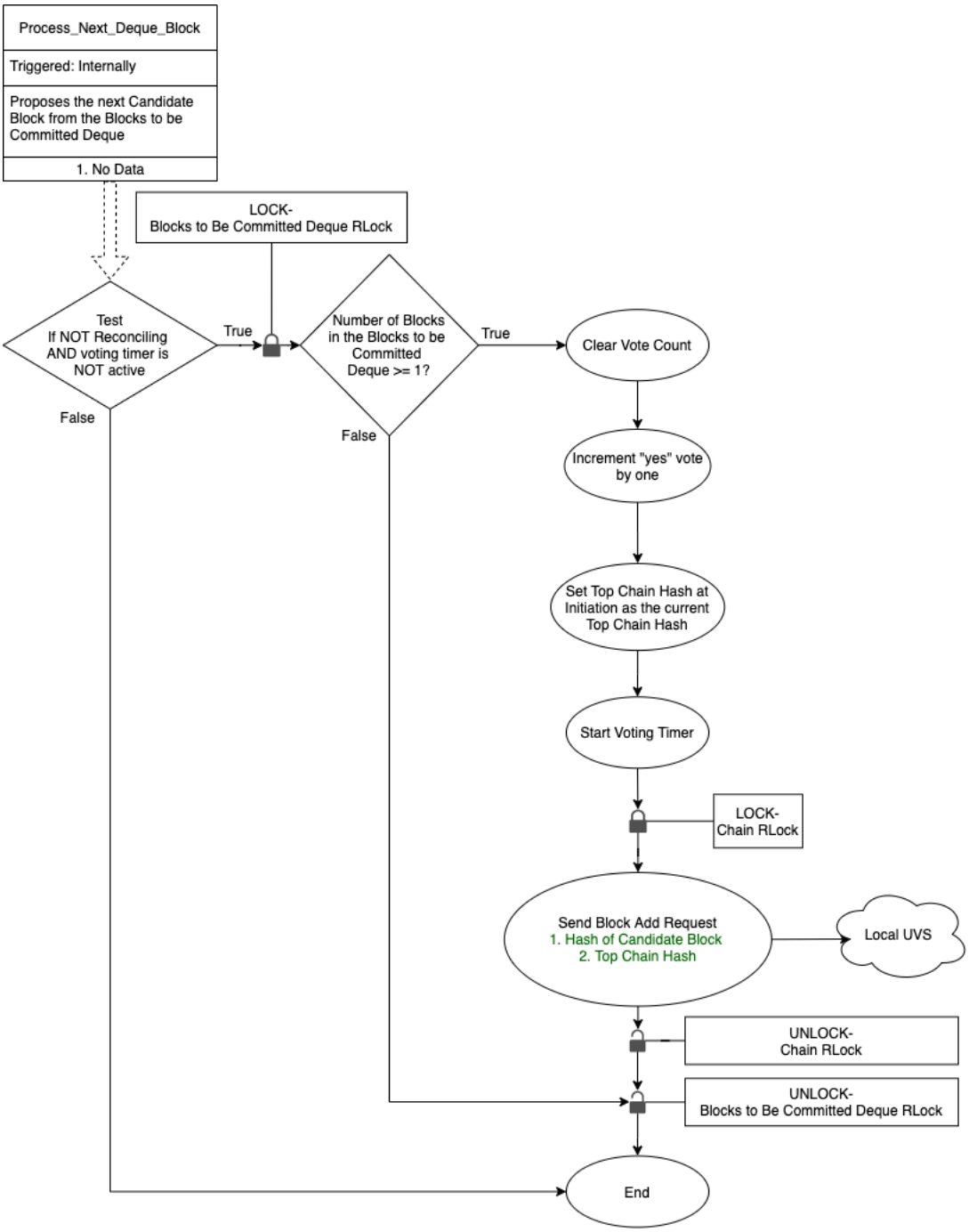

Figure 3.5. The **Process Next Deque Block** event handler is initiated internally by multiple event handlers to initiate the process by which a new `block` is proposed for addition to the `block chain`.

Receive Request Add Block Event

The **Receive Request Add Block** event is triggered externally by receipt of a message indicating that another `agent` is requesting to add a `block` to the `block chain`. The triggering message includes the `block hash` of the `block` that is being proposed and the requesting `agent`'s `top chain hash`. If the local `top chain hash` matches the requesting `agent`'s `top chain hash`, the event handler will generate and transmit an "approve" response for the proposed `block hash`. Otherwise, it will generate and transmit a "disapprove" response. The processing of this event is described by Algorithm 3 and is depicted graphically in Figure 3.6.

Algorithm 3 Receive Request Add Block Event Handler

if not *reconcileInProgress* **then**
    Obtain the *chainRLock*
    $blockAddResponse.blockHash \leftarrow message.blockHash$
    if $topChainHash == message.chainHash$ **then**
        $blockAddResponse.vote \leftarrow$ "$approve$"
    else
        $blockAddResponse.vote \leftarrow$ "$disapprove$"
    end if
    Network send **Block Add Response**
    Release the *chainRLock*
end if

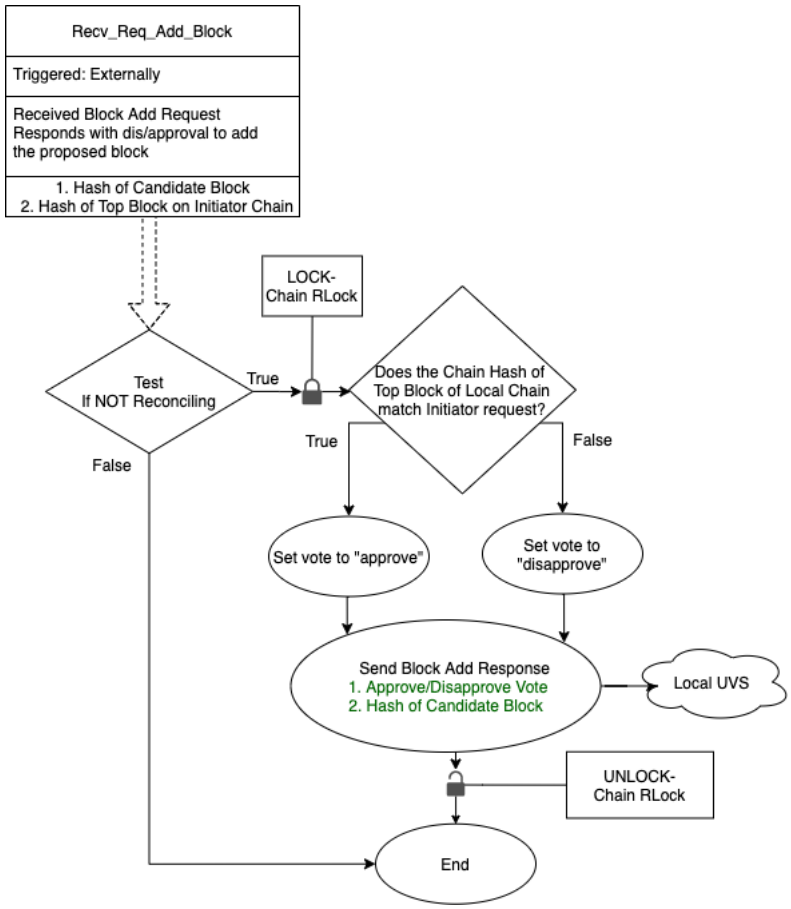

Figure 3.6. The **Receive Request Add Block** event handler is invoked upon receipt of a proposal from another `agent` to add a `block` to the `block chain`.

Receive Vote Add Block Event

The **Receive Vote Add Block** event is triggered externally when a response to a block add request message is received from another `agent`. The purpose of this handler is to parse and tally votes for a previously proposed addition. Because communication is broadcast to every `agent` and because each `agent` executes the handler for every event, the `agent` receiving the response must determine if the response was for it. If so, the `agent` will increment internal vote tally counters; otherwise it will ignore the response. The processing of this event is described by Algorithm 4 and is depicted graphically in Figure 3.7.

Algorithm 4 Receive Vote Add Block Event Handler

```
if votingTimer is active and
     message.blockHash == candidateBlock.blockHash then
    if message.vote == "approve" then
        voteCount["yes"] += 1
    else
        voteCount["no"] += 1
    end if
end if
```

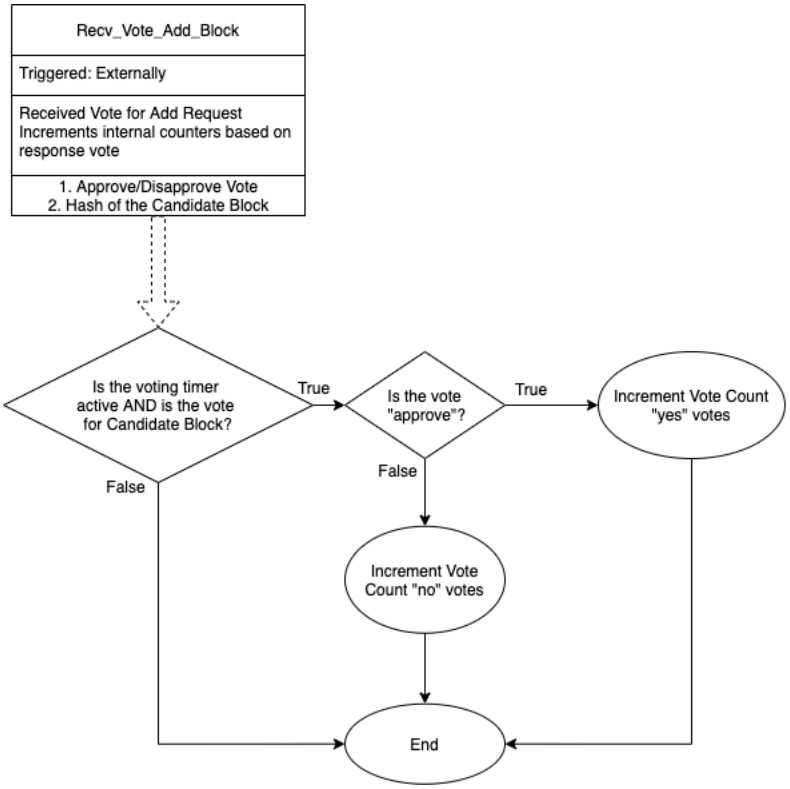

Figure 3.7. The **Receive Vote Add Block** event handler is triggered upon receipt of a vote for a previously proposed block addition and tallies "approve" and "disapprove" votes for later use.

Vote Timer Expiration Event

The **Vote Timer Expiration** event is triggered internally when the vote timer expires. (Recall that the vote timer was started by the **Process Next Deque Block** event handler.)

The purpose of this event handler is to determine if the request to add a `block` was "approved" or "disapproved" by a majority of the responding `agent`s. Depending on the voting results the `agent` will do one of three things:

1. Trigger a **Reconcile Begin** event if the majority of the votes were "disapprove".
2. Commit the `block` to the local `block chain`, broadcast the `block` for other `agent`s to commit to their `block chain`s, and trigger a **Process Next Deque Block** event if a majority of the votes were "approve".
3. Trigger a **Process next Deque Block** without committing the proposed `block` if the `block chain` changed while the vote tally process was running.

The processing of this event is described by Algorithm 5 and is depicted graphically in Figure 3.8.

Algorithm 5 Vote Timer Expiration Event Handler

if not *reconcileInProgress* **then**
  if $voteCount[\text{“}yes\text{”}] > voteCount[\text{“}no\text{”}]$ **then**
    Obtain the *blocksToBeCommittedDequeRLock*
    Obtain the *chainRLock*
    if *topChainHash* == *topChainHashAtInitiation* **then**
      $approvedBlock \leftarrow blocksToBeCommitted.front$
      $hashData \leftarrow (topChainHash + approvedBlock.blockHash)$
      $approvedBlock.chainHash \leftarrow \mathcal{H}(hashData)$
      $commitBlock.block \leftarrow approvedBlock$
      Commit $approvedBlock$ to the $blockChain$
      Network send **Commit Block**
      Remove $blocksToBeCommitted.front$
      Trigger **Process Next Deque Block** event
    end if
    Release the *chainRlock*
    Release the *blocksToBeCommittedDequeRLock*
  else
    Trigger **Reconcile Begin** event
  end if
end if

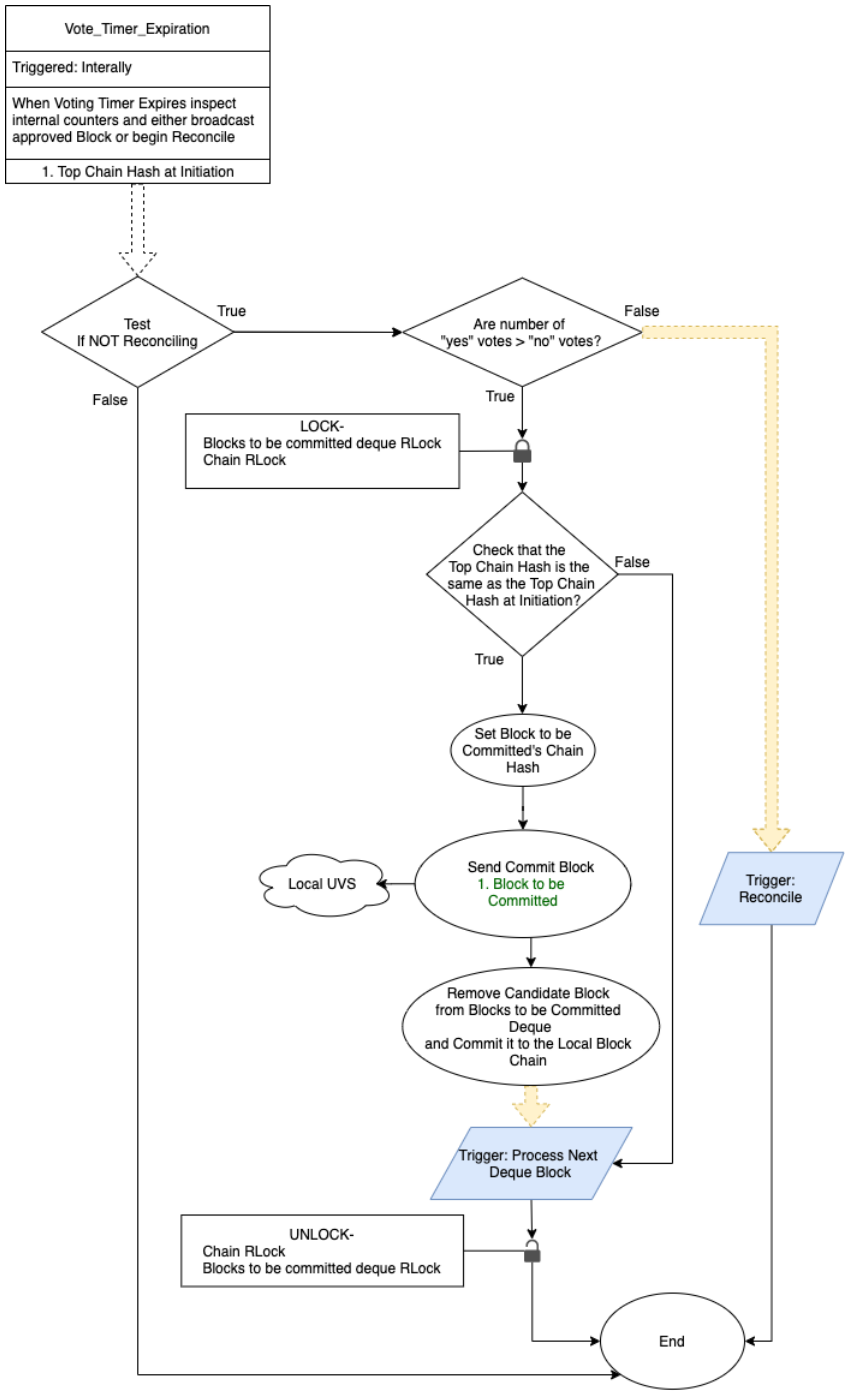

Figure 3.8. The **Vote Timer Expiration** event handler uses the results of the block add vote to determine whether to commit the proposed `block` to the `block chain` or trigger a **Reconcile Begin** event.

Receive Commit Block Event

The **Receive Commit Block** event is triggered externally by receipt of a block add message from another `agent`. The purpose of this event handler is to commit the received `block` to the local `block chain`. The processing of this event is described by Algorithm 6 and is depicted graphically in Figure 3.9.

Algorithm 6 Receive Commit Block Event Handler

if not *reconcileInProgress* **then**
    Obtain the *chainRLock*
    if $message.block$ succeeds $topChainHash$ **then**
        Commit $message.block$ to the $blockChain$
    end if
    Release the *chainRLock*
end if

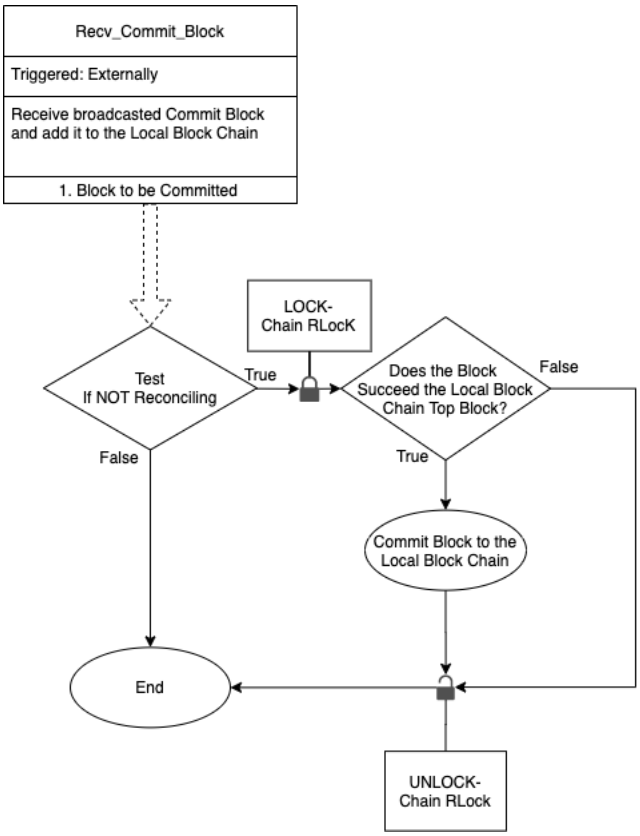

Figure 3.9. The **Receive Commit Block** event handler can be triggered by receipt of a block add message from another `agent` or locally by the **Vote Timer Expiration** or **Reconcile Finalize** event handlers. This event handler formally commits a block to the local `block chain` if appropriate.

3.6.3 Reconcile Process

The reconcile process is utilized to create consensus to ensure a `mutual chain` among the `local UVS` as `block`s are added. This also ensures that progress is made toward dispersing `event data` to each `agent` in the system.

Reconcile Begin Event

The **Reconcile Begin** event is triggered internally by two event handlers: **Vote Timer Expiration** and **Receive Request Next in Sequence**. This is the event handler that is referenced when reconcile is triggered in the diagrams. The purpose of this event handler

is to initiate the reconcile process. It also sets an internal Boolean variable, `reconcile in progress`, that will inhibit the `agent`'s participation in other `agents`' commit and reconcile processes while the local `block chain` is in the process of being altered. The processing of this event is described by Algorithm 7 and is depicted graphically in Figure 3.10.

Algorithm 7 Reconcile Begin Event Handler

if not *reconcileInProgress* **then**
  $reconcileInProgress \leftarrow True$
  Trigger **Reconcile Phase 1** event
end if

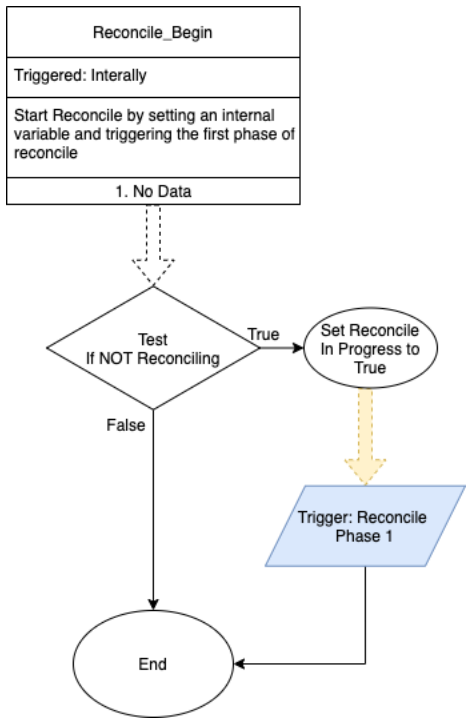

Figure 3.10. The **Reconcile Begin** event handler is triggered by the **Vote Timer Expiration** and **Receive Request Next in Sequence** events and initiates the reconcile process.

Reconcile Phase 1 Event

Per the high-level overview, the purpose of phase 1 is to remove `blocks` from the local `block chain` and store them in the `reconcile stack` until a `mutual chain` is found

with a majority of the `local UVS`.

The **Reconcile Phase 1** event is triggered internally by two event handlers: **Reconcile Begin** and **Reconcile Phase 1 Timer Expiration**. It issues a request to the `local UVS` asking if their `block chain`s contain this `agent`'s `top chain hash`. The processing of this event is described by Algorithm 8 and is depicted graphically in Figure 3.11.

Algorithm 8 Reconcile Phase 1 Event Handler

Clear *reconcileVoteCount*
$chainContainQuery.chainHash \leftarrow topChainHash$
Network send **Chain Contain Query**
Start *phaseOneTimer*

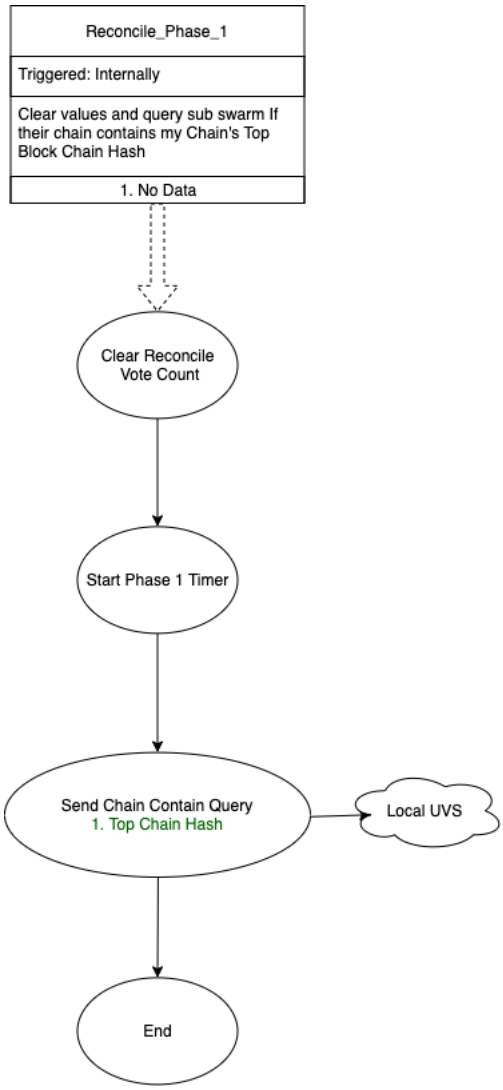

Figure 3.11. The **Reconcile Phase 1** event handler is triggered by the **Reconcile Begin** and **Reconcile Phase 1 Timer Expiration** event handlers and initiates the process of identifying a `mutual chain` upon which to build.

Receive Request Chain Contain Event

The **Receive Request Chain Contain** event is triggered externally by receipt of a chain contain query message from another `agent` asking whether or not the local `block chain` contains a particular `chain hash`. This event handler will respond the to the request with

a “yes” if the local `block chain` contains the queried `chain hash` and “no” otherwise. The processing of this event is described by Algorithm 10 and is depicted graphically in Figure 3.13.

Algorithm 9 Receive Request Chain Contain Event Handler

```
if not reconcileInProgress then
    chainContainResponse.chainHash ← message.chainHash
    Obtain the chainRLock
    if localChain contains message.chainHash then
        chainContainResponse.vote ← "yes"
    else
        chainContainResponse.vote ← "no"
    end if
    Network send Chain Contain Response
    Release the chainRLock
end if
```

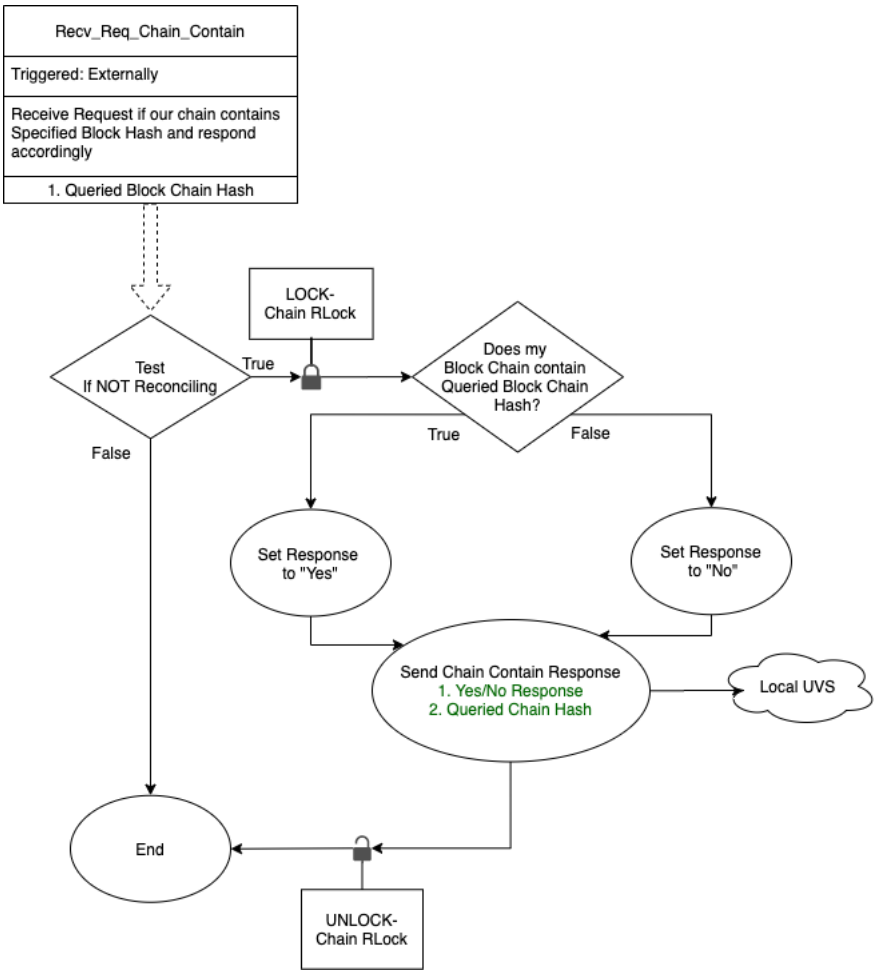

Figure 3.12. The **Receive Request Chain Contain** event handler is triggered by receipt of a message from another `agent` asking whether or not a particular `chain hash` is contained in the local `block chain`.

Receive Phase 1 Response Event

The **Receive Phase 1 Response** event is triggered externally by receipt of a message responding to a chain contain query message. The purpose of this event handler is to collect responses for use in determining whether or not the current `block chain` can be used as a `mutual chain` upon which to build. The processing of this event is described by Algorithm 10 and is depicted graphically in Figure 3.13.

Algorithm 10 Receive Phase 1 Response Event Handler

```
if phaseOneTimer is active and message.chainHash == topChainHash then
    if message.vote == "yes" then
        reconcileVoteCount["yes"] += 1
    else
        reconcileVoteCount["no"] += 1
    end if
end if
```

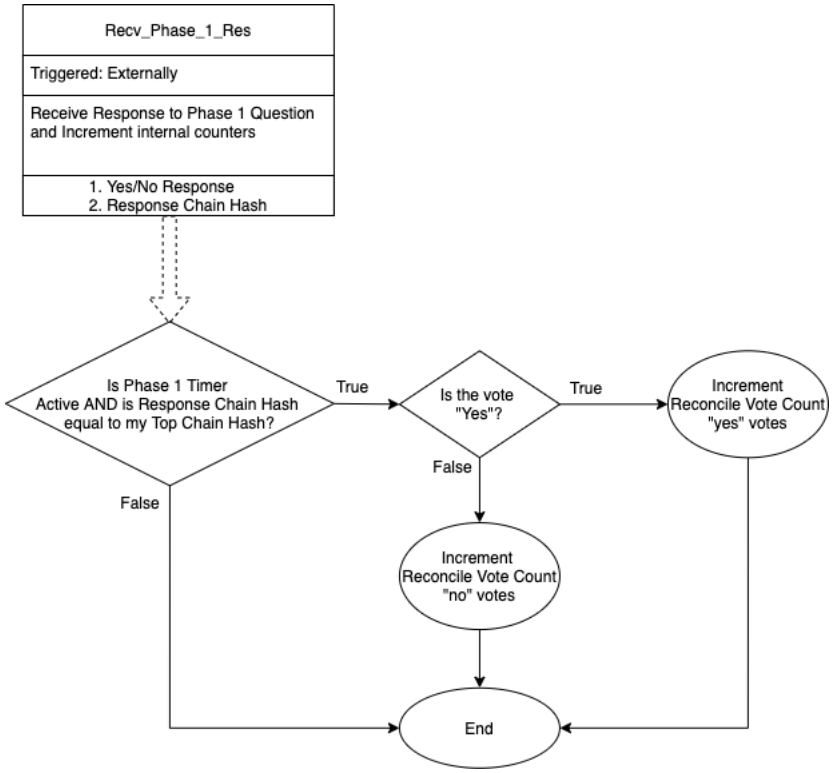

Figure 3.13. The Receive Phase 1 Response event handler is triggered by the receipt of a response to a Chain Contain Query message.

Reconcile Phase 1 Timer Expiration Event

The **Reconcile Phase 1 Timer Expiration** event is triggered internally upon expiration of the `phase one timer` that was started by the **Reconcile Phase 1** event handler. The purpose of this event handler is to tally the chain contain response results. One of the following courses of action will be initiated based on the results:

1. If no responses were received, then the `agent` is not in communication with any other `agent` and a **Reconcile Finalize** event is triggered.
2. If the majority of the responses are "no" then the top `block` in the `agent`'s local `block chain` `block chain` is removed and added to the to `reconcile stack`, and a **Reconcile Phase 1** event is triggered.
3. If the majority of the responses are "yes" then the `mutual chain` has been found and a **Reconcile Phase 2** event is triggered.

The processing of this event is described by Algorithm 11 and is depicted graphically in Figure 3.14.

Algorithm 11 Reconcile Phase 1 Timer Expiration Event Handler

if $reconcileVoteCount[\text{“}yes\text{”}] + reconcileVoteCount[\text{“}no\text{”}] == 0$ **then**
    Trigger **Reconcile Finalize** event
else if $reconcileVoteCount[\text{“}yes\text{”}] < reconcileVoteCount[\text{“}no\text{”}]$ **then**
    $temp \leftarrow localChain.pop()$
    $reconcileStack.push(temp)$
    Trigger **Reconcile Phase 1** event
else
    Trigger **Reconcile Phase 2** event
end if

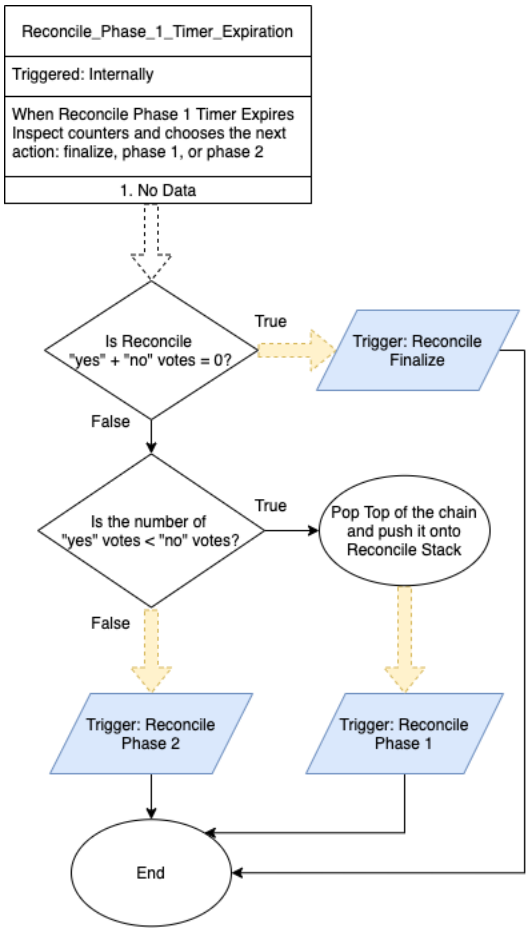

Figure 3.14. Upon expiration of the **Reconcile Phase 1 Timer**, responses are tallied to determine whether or not a `mutual chain` has been identified.

Reconcile Phase 2 Event

The **Reconcile Phase 2** event is triggered by the **Reconcile Phase 1 Timer Expiration** event handler when a `mutual chain` has been identified. Recall from Section 3.5.3 that phase 2 is where the `agent` builds upon the `mutual chain` to align its `local chain` with that of the `local UVS`. In phase 2, `agents` within the `local UVS` provide the reconciling `agent` with `blocks` from their `block chains` to establish consensus.

The event handlers associated with phase 2 rely on a dictionary or map data structure associating a counter and time stamp with each candidate `block hash` rather than a simple "yes"/"no" counter. This data structure facilitates determination of the correct `block` to add

to the `block chain` given multiple options and allows for the use of the time stamp as a tiebreaker (i.e., the `block` with the oldest time stamp is chosen).

This event handler is triggered internally after the `agent` has found a `mutual chain` with the majority of the `local UVS` The purpose of this event handler is to send the phase 2 request to the `local UVS` and clear counters for the expected responses. The broadcast request asks each `agent` in the `local UVS` to reply with the `block hash` of its `block` that succeeds the requesting `agent`'s `top chain hash`. The processing of this event is described by Algorithm 12 and is depicted graphically in Figure 3.15.

Algorithm 12 Reconcile Phase 2 Event Handler

$reconcileQuestionResponseCount \leftarrow 0$
Clear $reconcileResponseBlockDictionary$
$chainHashSuccessorQuerry.chainHash \leftarrow topChainHash$
Start $phaseTwoTimer$
Network send **Chain Hash Successor Query**

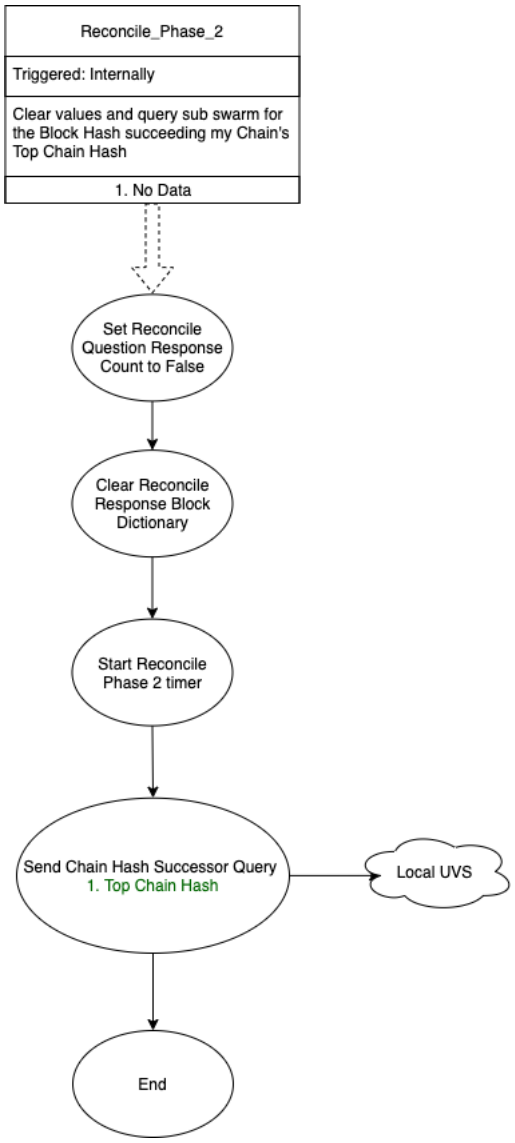

Figure 3.15. The **Reconcile Phase 2** event handler is initiated by the **Reconcile Phase 1 Timer Expiration** event handler to initiate the process of adding blocks to the `mutual chain`.

Receive Request Next in Sequence Event

The **Receive Request Next in Sequence** event is triggered externally by receipt of a `chain Hash` successor query message by which another `agent` is requesting the `block hash` of the `block` immediately succeeding the requesting `agent`'s local `top chain hash`. The

purpose of this event handler is to search the local `block chain` for the requested `chain hash` and to respond accordingly based on whether or not it is found.

This event handler has the additional task of initiating the **Reconcile Begin** process if the requested `chain hash` was not found in the `block chain`. If this happens then it indicates that this `agent`'s local `block chain` has diverged from the majority `mutual chain` and needs to be reconciled.

The processing of this event is described by Algorithm 13 and is depicted graphically in Figure 3.16.

Algorithm 13 Receive Request Next in Sequence Event Handler

```
if not reconcileInProgress then
    Obtain the chainRLock
    if blockChain contains message.chainHash then
        nextInSequenceResponse.chainHash ← message.chainHash
        if message.chainHash == topChainHash then
            nextInSequenceResponse.block ← "Is Top Of Chain"
            nextInSequenceResponse.timestamp ← blockChain.top.timestamp
        else
            nextBlock ← blockChain.getNext(message.chainHash)
            nextInSequenceResponse.blockHash ← nextBlock.blockHash
            nextInSequenceResponse.timestamp ← nextBlock.timestamp
        end if
        Network send Next in Sequence Response
    else
        Trigger Reconcile Begin event
    end if
    Release the chainRLock
end if
```

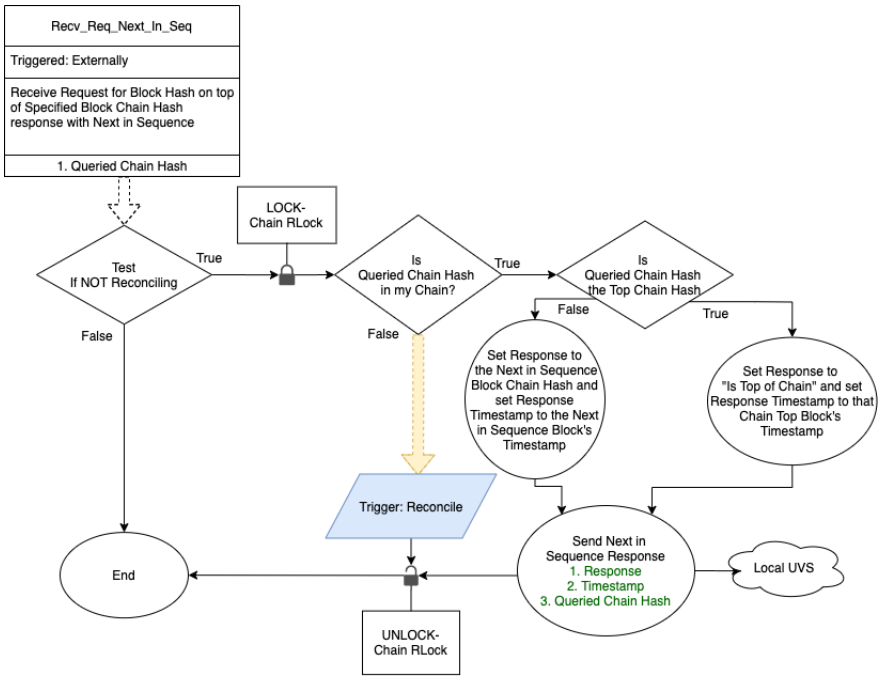

Figure 3.16. The **Receive Request Next in Sequence** event handler is triggered by receipt of a `chain hash` successor query from another `agent`. It responds to the query accordingly and initiates a local reconcile process as required.

Receive Response Next in Sequence Event Handler

The **Receive Response Next in Sequence** event is triggered externally by receipt of a message responding to a previously transmitted `chain hash` successor query. It maintains the vote counter and dictionary objects as responses are received so that the correct `block` can be selected for addition to the `mutual chain`. The processing of this event is described by Algorithm 14 and is depicted graphically in Figure 3.17.

Algorithm 14 Receive Response Next in Sequence Event Handler

if $phaseTwoTimer$ is active and $message.chainHash == topChainHash$ **then**
  if $message.blockHash$ not in $reconcileResponseDictionary$ **then**
    $counterRecord \leftarrow reconcileResponseDictionary[message.blockHash]$
    $counterRecord.count \leftarrow 1$
    $counterRecord.timestamp \leftarrow message.timestamp$
  else
    $reconcileResponseDictionary[message.blockHash].count$ += 1
  end if
  $reconcileResponseCount \leftarrow True$
end if

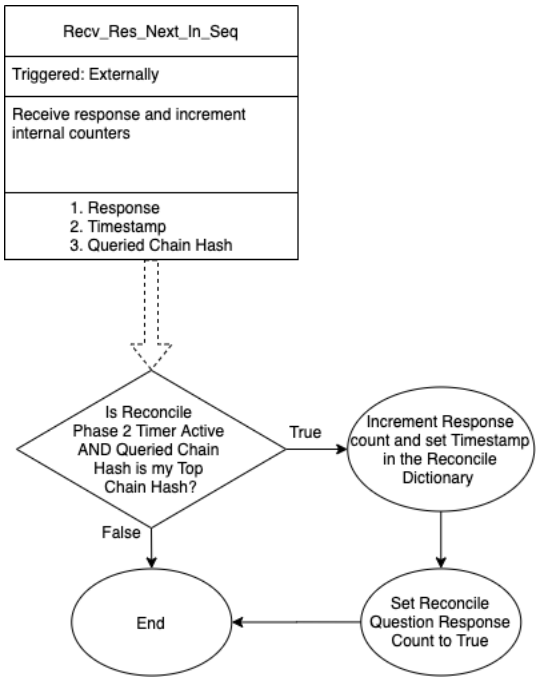

Figure 3.17. The **Receive Response Next in Sequence** event handler is triggered by receipt of a message responding to a previously transmitted `chain hash` successor query and maintains counters facilitating correct extension `mutual chain`.

Reconcile Phase 2 Timer Expiration Event

The **Reconcile Phase 2 Timer Expiration** event is triggered upon expiration of the `phase 2 timer` that was started by the **Reconcile Phase 2** event handler. Upon being triggered,

this event handler chooses the majority response from the dictionary. In the case of a tie, the response with the oldest time stamp is chosen. If no responses were received or the majority response was the special "Is Top of Chain" indicator, then the majority `mutual chain` has been completed and a **Reconcile Finalize** event is triggered. Otherwise, a **Reconcile Phase 3** event is triggered to acquire a copy of the `block` associated with the majority response. The processing of this event is described by Algorithm 15 and is depicted graphically in Figure 3.18.

Algorithm 15 Reconcile Phase 2 Timer Expiration Event Handler

if $reconcileResponseCount == False$ **then**
    Trigger **Reconcile Finalize** event
else
    if $majorityResponse$ from $reconcileResponseDictionary$ is a tie **then**
        $majorityResponse \leftarrow blockHash$ with oldest $timestamp$
    else
        $majorityResponse \leftarrow majorityResponse\ blockHash$
    end if
    if $majorityResponse ==$ *"Is Top of Chain"* **then**
        Trigger **Reconcile Finalize** event
    else
        $majorityResponseBlockHash \leftarrow majorityResponse$
        Trigger **Reconcile Phase 3** event
    end if
end if

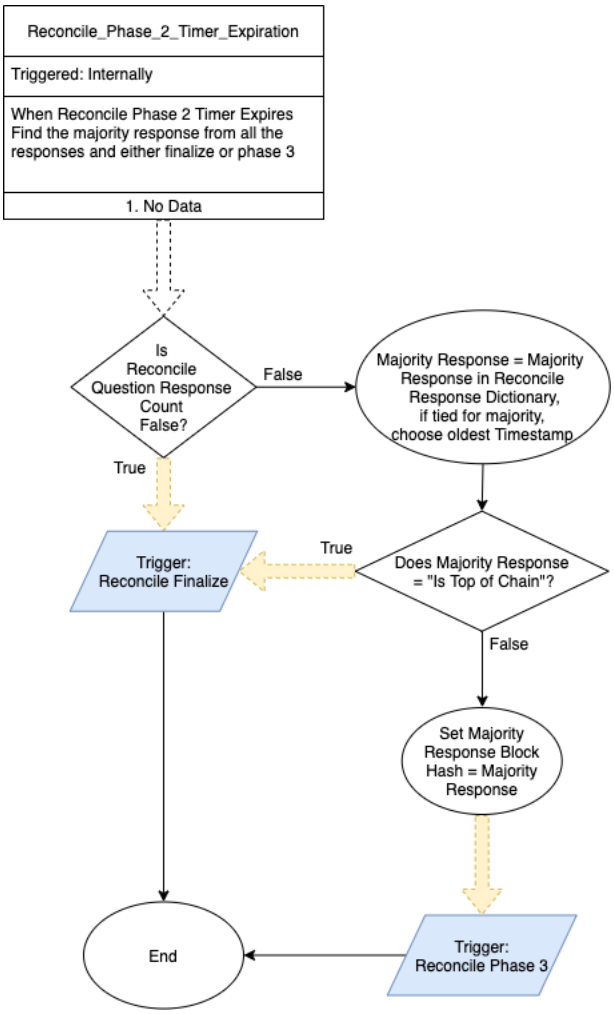

Figure 3.18. The **Reconcile Phase 2 Timer Expiration** event is triggered upon expiration of the `phase 2 timer`. It determines the `block hash` associated with the local majority `mutual chain`.

Reconcile Phase 3 Event

The **Reconcile Phase 3** event is triggered internally by the **Reconcile Phase 2 Timer Expiration** event handler once it has identified the `block hash` of the next `block` to be added to the `block chain`. Recall that the point of phase 3 is to obtain the actual `block` associated with the majority `block hash`. The event handler first searches for the `block hash` in the `reconcile stack` (where it may be located if was previously committed to the `block chain` but was removed earlier in the reconcile process). If the `block` is

available locally, it is removed from the `reconcile stack` and committed to the `block chain`. If not, a request for the `block` is broadcast to the `local UVS` and the `phase 3 timer` is started. The processing of this event is described by Algorithm 16 and is depicted graphically in Figure 3.19.

Algorithm 16 Reconcile Phase 3 Event Handler

if $majorityResponseBlockHash$ is in the $reconcileStack$ **then**
  $block \leftarrow reconcileStack.remove(majorityResponseBlockHash)$
  Obtain the $chainRLock$
  $hashData \leftarrow (topChainHash + block.blockHash)$
  $block.chainHash \leftarrow \mathcal{H}(hashData)$
  Commit $block$ to the $blockChain$
  Release the $chainRLock$
else
  $requestForBlock.blockHash \leftarrow majorityResponseBlockHash$
  Start $reconcilePhaseThreeTimer$
  Network send **Request for Block**
end if

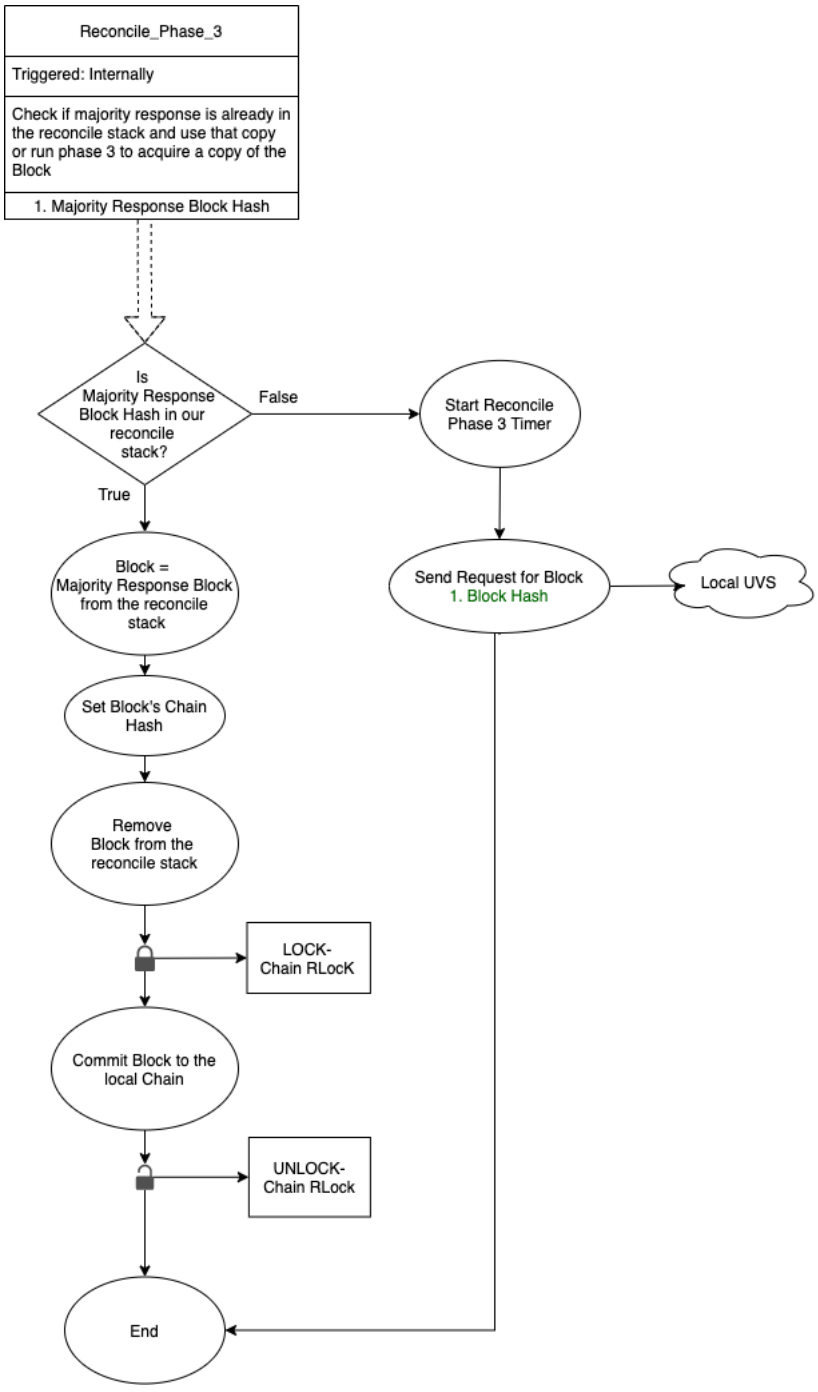

Figure 3.19. The **Reconcile Phase 3** event is triggered internally by the **Reconcile Phase 2 Timer Expiration** event handler and is responsible for initiating request for the `block` associated with the majority `block hash`.

Receive Request Block for Hash Event

The **Receive Request Block for Hash** event is triggered externally when a request is received from another `agent` for the `block` associated with a particular `block hash`. The purpose of this event handler is to search the local `block chain` for the requested `block` and to send it to the requesting `agent` if it is found. The processing of this event is described by Algorithm 17 and is depicted graphically in Figure 3.20.

Algorithm 17 Receive Request Block for Hash Event Handler

```
if not reconcileInProgress then
    Obtain the chainRLock
    if message.blockHash in blockChain then
        block ← blockChain.copy(message.blockHash)
        Network Send Block Response
    end if
    Release the chainRLock
end if
```

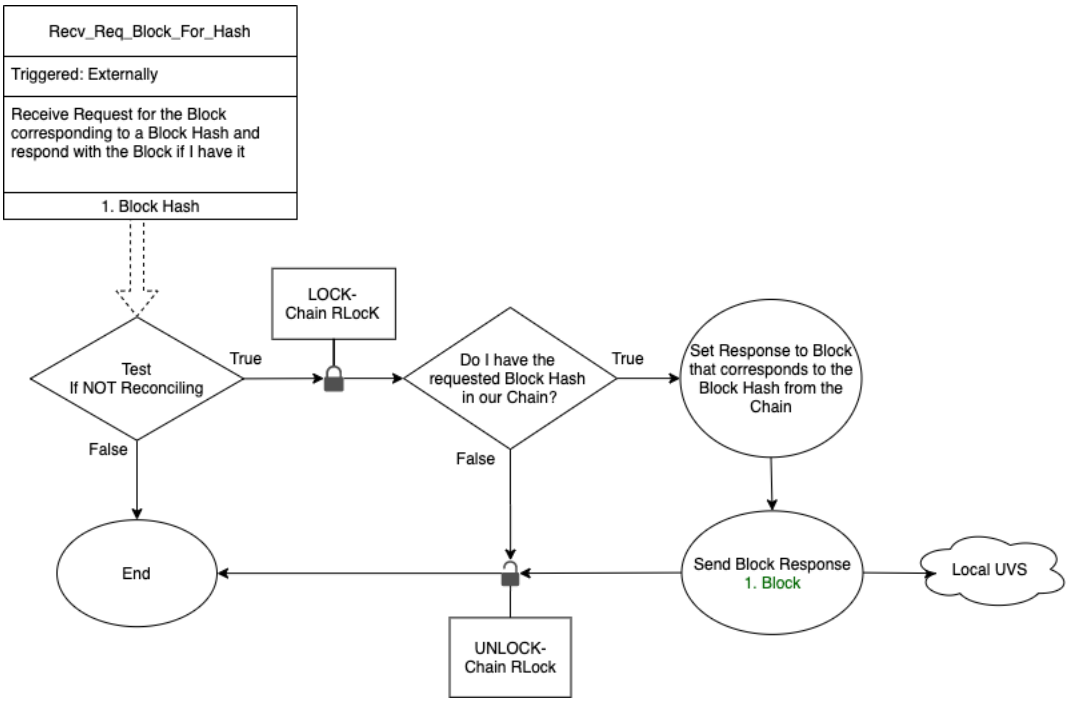

Figure 3.20. The **Receive Request Block for Hash** event handler is triggered by an external request for a `block` associated with a particular `block hash`. The `agent` sends `block` to the requesting `agent` if it is present in the local `block chain`.

Receive Response Block for Hash Event

The **Receive Response Block for Hash** event is triggered externally when a response to a `block` is received in response for a previously broadcast request for `block` is received. The purpose of this event handler is two-fold. First, it must determine whether or not the received `block` was in response to a request made by this `agent`, and if so, commit it to the `block chain`. Second, it manages the `phase 3 timer` by pausing and restarting it or terminating it. The processing of this event is described by Algorithm 18 and depicted graphically in Figure 3.21.

Algorithm 18 Receive Response Block for Hash Event

if $phaseThreeTimer$ is active **then**
  Pause $phaseThreeTimer$
  if $message.block.blockHash == majorityResponseBlockHash$ **then**
    Stop $phaseThreeTimer$
    Obtain the $chainRLock$
    $hashData \leftarrow (topChainHash + message.block.blockHash)$
    $message.block.chainHash \leftarrow \mathcal{H}(hashData)$
    Commit $message.block$ to the $blockChain$
    Release the $chainRLock$
    $majorityResponseBlockHash \leftarrow Null$
    Trigger **Reconcile Phase 2** event
  else
    Resume $phaseThreeTimer$
  end if
end if

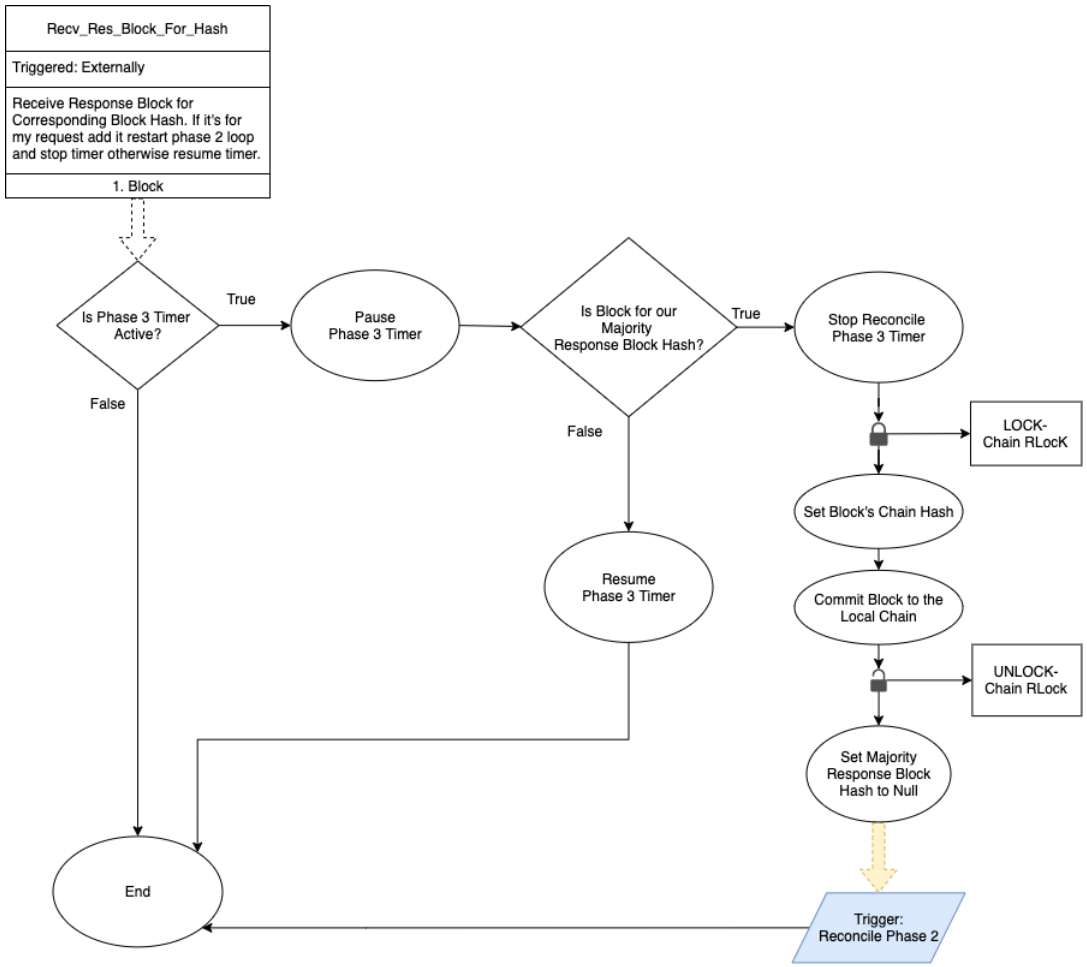

Figure 3.21. The **Receive Response Block for Hash** event handler is triggered by receipt of a response to a previously transmitted request for `block`. If the contained `block` is the one that was requested, it is added to the local `block chain`.

Reconcile Phase 3 Timer Expiration Event

The **Reconcile Phase 3 Timer Expiration** event is triggered upon expiration of the `phase 3 timer` that was started by the **Reconcile Phase 3** event handler. If a valid response is received, the `phase 3 timer` is terminated by the **Receive Response Block for Hash** event handler, so the `phase 3 timer` will only expire if no valid responses were received. This event handler, therefore, simply concludes the reconcile process by triggering a **Reconcile Finalize** event. The processing of this event is described by Algorithm 19 and is depicted

graphically in Figure 3.22.

Algorithm 19 Reconcile Phase 3 Timer Expiration Event Handler

$majorityResponseBlockHash \leftarrow Null$
Trigger **Reconcile Finalize** event

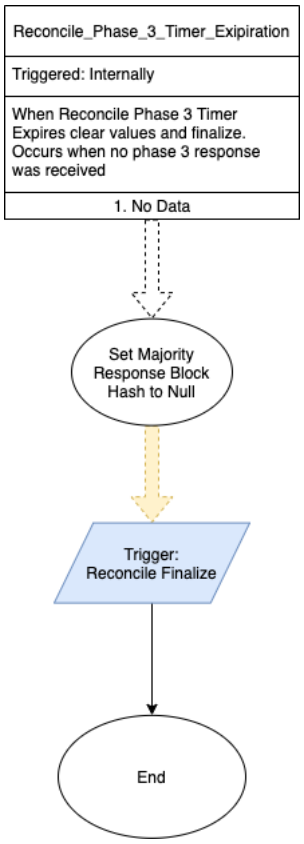

Figure 3.22. The **Reconcile Phase 3 Timer Expiration** event handler is only triggered when no valid responses are received to a request for `block` message. When this occurs, **Reconcile Phase 3** is terminated by triggering a **Reconcile Finalize** event.

Reconcile Finalize Event

The **Reconcile Finalize** event is triggered internally by the **Reconcile Phase 1 Timer Expiration** event handler, the **Reconcile Phase 2 Timer Expiration** event handler, or the **Reconcile Phase 3 Timer Expiration** event handler. Its purpose is to recommit `blocks`

remaining in the `reconcile stack` after completion of the reconcile process back to the local `block chain`. In addition to recommitting them to the local `block chain`, the event handler broadcasts them to the `local UVS` for other `agents` so that they can be added to their respective `block chains`. Once the `reconcile stack` has been cleared, the event handler unsets the `reconcile in progress` flag and triggers a **Process Next Deque Block** so that any blocks that have been generated during the reconcile process can be dealt with. The processing of this event is described by Algorithm 20 and depicted graphically in Figure 3.23.

Algorithm 20 Reconcile Finalize Event Handler

Obtain the $chainRLock$
while $reconcileStack$ not empty **do**
    $commitBlock \leftarrow reconcileStack.pop()$
    $hashData \leftarrow (topChainHash + commitBlock.blockHash)$
    $commitBlock.chainHash \leftarrow \mathcal{H}(hashData)$
    Commit $commitBlock$ to $localChain$
    Network Send **Commit Block**
end while
Release the $chainRLock$
$reconcileInProgress \leftarrow False$
Trigger **Process Next Deque Block** event

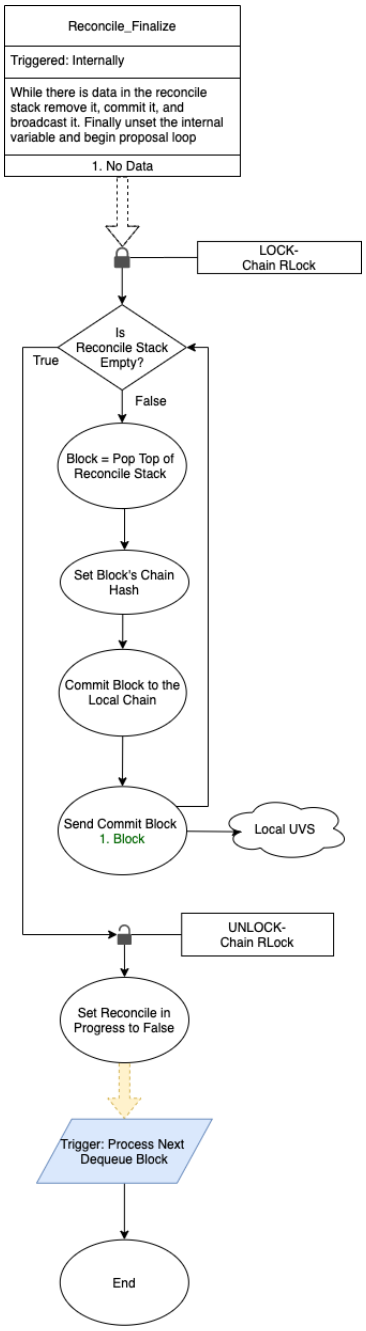

Figure 3.23. The **Reconcile Finalize** event handler can triggered by any of the reconcile phase `timer` expiration events or when the maximum `local UVS mutual chain` has been generated to recommit `blocks` remaining in the `reconcile stack` to the `blockchain`.

3.7 Chapter Summary

In this chapter, we presented the novel Uniform Chain Protocol (UCP). We started the chapter by presenting a brief summary of the protocol in Section 3.1. Next, we introduced and defined the terminology that we used in the context of the UCP. After defining the terminology, we asserted all of the assumptions that we made in designing the UCP. Next, we described the `block chain` data structure that we implemented in the UCP and provided a high-level overview of the UCP. Finally, we described the 20 events that comprise the UCP and provided detailed the event-handling algorithms for each.

In Chapter 4 of the Pommer thesis, an exemplar implementation of the UCP on the ARSENL multi-UAV system is described. In Chapter 4 of the Carter thesis, the UCP is examined for correctness utilizing the Monterey Phoenix (MP) behavior modeling tool.

CHAPTER 4:
Implementation and Results

This chapter covers the implementation and testing of the UCP, as described in Chapter 3. Following a brief overview of the implementation, a discussion of the development process, implementation-specific details, developmental SITL testing, field testing, and issues that surfaced during testing are provided. The chapter concludes with a discussion of results obtained during experiments with the fully-implemented protocol.

4.1 Implementation Overview

This section introduces the target system and components of the UCP implementation, including the Python classes involved and details about the events generated.

The exemplar UCP was implemented within the ARSENL on-vehicle system described in Section 2.5. As described in the Chapter 2, the onboard system runs on an ODroid payload computer running a version of Ubuntu Linux. The ODroid is a single-board computer powered by an Advanced RISC Machine processor. The ARSENL system, and thus the exemplar UCP implementation, is also compatible with the SITL simulation environment described in Section 2.5.

The UCP implementation was developed to run as a ROS node and is written in Python2. Four Python classes were developed to support all UCP functionality. A `PauseableTimer` class executes a function upon expiration and has the ability to be started, paused, resumed, and canceled. `Block` and `BlockChain` classes are used to encode individual data `block`s and the `block chain` itself, respectively. Finally, a `BlockchainBridge` class provides all UCP functionality and serves as a bridge between the block chain implementation and the ARSENL ROS nodes.

UCP event handlers are implemented as methods of the `BlockchainBridge` class. Externally triggered event handlers are implemented as callbacks for ROS message topics to which the node subscribes. Internally triggered event handlers, such as those associated with transitions to the next phase in the reconcile process, are also invoked using callbacks

for ROS message topics to which the node both publishes and subscribes. Timer expiration event handlers are invoked by the 'timer_finished' method of the `PauseableTimer`.

Event handlers that transmit messages over the network do so by publishing to a ROS topic to which the network node subscribes. The network node then generates and transmits the ARSENL message. The network node also receives UCP messages from the network and publishes ROS messages to the appropriate topics for event handler processing.

In addition, each network message includes the UAV's `agent ID` (i.e., the $aircraft\_id$ of the sending UAV). The purpose of the `agent ID` is to allow detection of duplicate messages. Consider for example, if two UAVs were executing phase 2 of the reconcile process[1] and they both requested information for the same hash, they should both receive the exact same responses. If there was no way to ignore responses that have already been received, each of the UAVs might double count the responses. This would not be an issue if the network conditions and timing of the two UAVs requests were identical, as their final result would still be the correct just doubled; however, we cannot depend on this, so some answers would be counted twice, while others would be counted once. The duplicate responses could skew the results toward an incorrect answer.

Loggable events to be added to the block chain are generated by a separate ROS node that publishes event data to a message topic to trigger the **Build Data Block** event handler. These events are randomly generated approximately every 30 seconds. The event generation algorithm is described by Algorithm 21.

Algorithm 21 Generation of Loggable Events

$MEAN\_TIME\_BTWN\_EVENTS \leftarrow 30.0$
while *running* **do**
  if $random\_between(0, 1) <= 1/MEAN\_TIME\_BTWN\_EVENTS$ **then**
    $generate\_event()$
  end if
end while

A loggable event consists of three components: a timestamp indicating when the event was generated; an event ID, and event data. Because the actual event data has no impact on the

[1]Phases and their relationships to the UCP reconcile process are discussed in detail in Section 4.2.

UCP, event IDs and data are randomly generated for the exemplar implementation. Event IDs are random integers between 1 and 254, and event data consists of a random sequence of 32, 64, 128, or 256 bytes.

An individual `block` is comprised of the following: a timestamp, an `agent ID` indicating the `agent` that generated the `block`, a series of logged events, a `chain hash`, and a `block hash`. A `block` is configured to store no more than 1024 bytes of data. This maximum `block` size was chosen to allow encoding of a `block` in a single UDP packet. This limit was incorporated to decrease the risk of receiving incomplete `block`s.

4.2 Development

Code development was conducted in two stages. The first stage focused on the event handlers associated with generating and committing `block`s (Sections 3.6.1 and 3.6.2). The event handlers associated with the reconcile process (Section 3.6.3) were implemented in the second stage.

Testing was conducted in three stages. The first stage assumed lossless (i.e., no packet loss) network connectivity. These tests eliminated the requirement for reconciliation and allowed testing of the commit block process in isolation. The second stage used the same lossless network, but probabilistically ignored commit messages in the **Receive Commit Block** event handler. These tests forced divergences among the local `block chain`s in order to trigger the reconcile process. The final stage of testing was with lossy-communications where every packet had a probabilistic chance of being ignored. These tests were used to verify the robustness of the entire UCP in a realistic communications environment. It should be noted that these tests did not deterministically test protocol performance in all reconciliation scenarios, but the MP testing documented in [32] provides broader protocol verification.

Aside from the **Build Data Block**, **Receive Commit Block**, **Reconcile Begin**, and **Reconcile Finalize** events, each set of the event handlers follow a "*send request*, *receive request* and *send response*, *receive response*, *timer expires*" pattern. Event handlers for this sequential pattern are highly interdependent, which directly impacted the development process. For example, the reliance of one event handler on output from another made development of

the *receive response* event handler before the *send response* event handler infeasible.

Voting Processes and Timers

Voting is utilized by both the event generation and commit process event handlers and reconcile process event handlers. Voting itself relies on timers that trigger vote tallying and response events upon expiration. The voting process aligns with the "*send request*, *receive request* and *send response*, *receive response*, *timer expires*" pattern. The event handler performing the *send request* is also responsible for starting a timer that controls how long the UAV will wait for responses from the `local UVS`. While the timer is active, UAVs within the `local UVS` perform the *receive request* and *send response* steps. The UAV that initiated the request will *receive response*s until the *timer expires* at which point the results are tallied and acted on.

The amount of time these timers should wait before expiring is somewhat arbitrary. Satisfactory algorithm performance, however, is reliant on choosing a "good" time. If a time is too short, a timer might expire before all responses are received. On the other hand, using a time that is too long results in wasted time during which the protocol waits unnecessarily. The exemplar implementation discussed here used a wait time of 10 seconds for the commit request timer and a wait time of 5 seconds for each reconcile phase timer were chosen. These values proved long enough to facilitate debugging and understanding the UVS performance in real-time without being so long as to bog down performance.

Commit Block Process Implementation

The commit block process begins with the receipt of a generated event that results in the generation of a new `block` as is described in Section 3.6.2. The `block`'s timestamp, which is equal to that of the most recent event in the `block`, is the only addition to the implementation beyond what is described in Chapter 3. This timestamp is used in **Reconcile Phase 3** as a tie breaker.

Because the commit block process (i.e., the **Process Next Deque Block**, **Receive Request Add Block**, **Receive Vote Add Block**, and **Vote Timer Expiration** events) aligns with the "*send request*, *receive request* and *send response*, *receive response*, *timer expires*" pattern, the sequence was implemented as a unit. Then the **Receive Commit Block** event handler

was implemented, as it relied on the entire previous sequence to be triggered.

Debugging of the commit sequence was difficult since all these events are interdependent and interact via the ROS publisher-subscriber framework. This made isolating the functions to perform test-driven development or unit-tests impossible. Rather, full implementation of the commit process was required before significant testing could be conducted. Once all of the event handlers were in place, it was possible to simulate multiple-UAVs in the SITL environment and debug the entire series of event handlers at once.

The next debugging difficulty was interpreting what was occurring on each UAV. Because of the parallel nature of the protocol and threading within the ROS infrastructure, it was not feasible to use a Python debugger to step through the code as it executed. Therefore, debugging was conducted by analyzing detailed messages saved to log files. These messages included when events were triggered, which event-handler branches were taken, and what values were generated or received. These log files were meticulously inspected by hand to discern what occurred during a test and to identify the states that caused the code to execute as it did. The tedious nature of this debugging precluded simulating more than four UAVs at a time, as manually following the log trace of more drones, even for a 10 minute run, was incredibly cumbersome.

To test the commit process implementation, the SITL environment was set up to provide perfect network communications (i.e., communication with no network traffic loss). Given perfect communications, there should be no missed commit requests, and all `agent`s should be able to complete all commits. Thus, we expected to see one system-wide `common chain` in which all `block`s were committed to all local `block chain`s in the order in which they were generated. That is, in fact, what we observed after completing the debugging process for this development stage.

Reconcile Process Implementation

The reconcile process as described in Section 3.6.3 consists of **Reconcile Begin**, the three phases of reconcile, and **Reconcile Finalize** events. The **Reconcile Begin** event is triggered when an `agent` determines that its local `block chain` has diverged from that of the `local UVS`. This event handler is largely required for "housekeeping" purposes and is responsible for setting the `reconcileInProgress` Boolean to true prior to initiating the rest of the

reconcile process. To ensure a race condition does not occur while checking and setting this variable, a general purpose lock was used. Functionally, the lock is used as an atomic Boolean to enforce mutual exclusion of read (i.e., test the lock) and write (i.e., set the lock) operations. Development of the three reconcile phases was similar to development of the commit process as each phase follows the "*send request*, *receive request* and *send response*, *receive response*, *timer expires*" pattern.

Reconcile phase 1 includes the **Reconcile Phase 1**, **Receive Request Chain Contain**, **Receive Phase 1 Response**, and **Reconcile Phase 1 Timer Expiration** events. These event handlers were implemented as a unit as each event handler is dependent on the previous handler in the sequence. Similarly, reconcile phase 2 includes the **Reconcile Phase 2**, **Receive Request Next in Sequence**, **Receive Response Next in Sequence**, and **Reconcile Phase 2 Timer Expiration** events and was implemented as a unit. Finally, reconcile phase 3, which consists of the **Reconcile Phase 3**, **Receive Request Block for Hash**, **Receive Response Block for Hash**, and **Reconcile Phase 3 Timer Expiration** events was also implemented as a unit.

The reconcile process concludes with the **Reconcile Finalize** event that is triggered once the longest possible `mutual chain` is constructed, or no responses are received for a request. This event handler is responsible for committing any `block`s remaining in the `reconcile stack` to the `block chain` before setting the `reconcileInProgress` Boolean to false. This process includes broadcasting each `block` to the `local UVS` so that it can be committed to their local `block chain`s as well. A pause that is not depicted in the Chapter 3 event handler description was incorporated to provide UAVs time to receive, parse, and commit `block`s as they are received.

To trigger the reconcile process for SITL testing, the **Receive Commit Block** event handler was temporarily modified to probabilistically ignore received commit messages. This forced divergence among the UAVs' `block chain`s. Aside from ignoring these messages, the SITL environment was configured to provide perfect network communications. As a result, no messages associated with the reconcile process itself were dropped.

As expected, during experiments we observed that some commit block messages were ignored, and the reconcile process commenced once an `agent` discovered the resulting divergence (i.e., during a subsequent commit process). Furthermore, the reconciling `agent`

continued through the process as anticipated, and the local `block chain` was in agreement with that of the `local UVS` after the **Reconcile Finalize** event handler concluded.

It should be noted that this testing phase did not fully vet the reconcile process. In particular, forced divergences were detected after a single `block` was missed, and the perfect communications allowed the discrepancy to be quickly rectified. This testing phase did, however, verify that the reconciliation process was correctly implemented in general.

The biggest challenge associated with debugging the reconcile process was that it was not possible to deterministically execute certain code paths. For example, in the SITL simulation environment it was impossible to force a tie in the **Reconcile Phase 2** voting process. The inability to induce large `block chain` divergences also made it impossible to perform in-depth testing.

Testing with Lossy-Communications

With a complete implementation of the UCP, testing in a lossy-communications environment (i.e., there is network traffic loss) was next. The ARSENL codebase provides a lossy-communication branch intended specifically for simulating dropped network packets in the SITL environment. When executing from this branch, arbitrary network messages are probabilistically ignored rather than being forwarded to the autonomy package. Testing in SITL with this branch provided the opportunity to not only test the robustness of the protocol, but also to execute code paths that were not executed under perfect communications conditions.

This phase of testing was performed with varying probabilities of packet loss: 0.00, 0.10, 0.15, 0.25, 0.30, 0.50, 0.75, and 1.00 probability of loss. Of course, loss probabilities of 0.00 and 1.00 equate to perfect and completely disconnected network environments respectively. To maintain consistency, each experiment ran for 20 minutes and included five simulated UAVs (these values were chosen to facilitate manual analysis of the results). After 20 minutes, event generation was stopped and the UAVs were allowed to finish completing any commit and reconcile processes. Once the system entered an idle state, the simulation was stopped, and the generated logs and `block chain`s were stored for analysis.

The majority of implementation bugs found during this testing phase were syntax errors

(e.g., misspelled variables and method calls). Such errors do not emerge until the runtime system attempts to read or call the non-existent identifier and generates an exception. One logical error that was identified resulted from incorrect implementation of a reconcile process event handler in which a message was published to the incorrect ROS topic and the wrong event was triggered as a result.

Due to the probabilistic nature of this round of tests, bugs frequently required many test runs to manifest. Similarly, bug fixes could not be verified until the scenario that presented the bug in the first place was duplicated. Moreover, since probabilistic tests cannot assure the execution of all code paths, the existence of additional bugs or protocol shortcomings cannot be ruled out. In addition, this phase of testing did not simulate the formation of disjoint networks forming and their subsequent merging (the ARSENL system does not currently provide for this scenario).

Live-Flight Field Testing

Following SITL testing, the UCP was tested during live flight field experiments conducted at Camp Roberts, CA in November 2020. Two tests were conducted with ARSENL ZephyrII fixed-wing and Mosquito Hawk quadrotor aircraft. In each test the protocol was allowed to run for approximately 30 minutes while the swarm executed various behaviors. Unlike SITL testing with lossy-communications, the field tests did not simultaneously terminate event generation for the entire swarm. Rather, each UAV stopped generating local events upon landing, so events were being generated within the system until the last plane was on deck. Thus, although all UAVs remained powered for some time after landing, completion of all reconcile processes was not verified before they were shut down. Also, due to the limited airspace available and the safety-of-flight requirement for continuous communication between ground stations and each UAV, it was not possible to force disjoint communication networks. At the conclusion of the tests, all `block chain`s and generated logs were downloaded for analysis.

During these experiments, we noted that `block chain`s were successfully saved correctly. However, after analyzing the generated log files, a number of implementation mistakes were discovered:

1. A file path error was identified that resulted in premature termination of the commit

process. This bug was corrected prior to the second experiment.
2. A syntax error arose when the majority response for reconcile phase 2 was adjudicated in cases where the vote was tied.
3. A logic error in the implementation of the *PauseableTimer* was found in which the expiration callback function was being invoked incorrectly.
4. A logical error in the **Receive Response Block for Hash** event handler resulted in a failure to recalculate the `chain hash`.

All of these errors were associated with the implementation and not with the UCP itself. Following the field testing, all errors were corrected, and, following their implementation, a final round of experiments was conducted in the SITL environment. The results are documented in Section 4.3.

4.3 Experimental Results

To verify that the final version of our implementation behaves correctly and to provide empirical evidence of the protocol's correctness, we conducted multiple experiments in both the SITL environment and the field. As a reminder, the following protocol requirements were laid out in Chapter 3:

1. The protocol is distributed, event driven, and asynchronous. Each event handler is implemented independently on every vehicle, and event handlers can be triggered locally or in response to inter-vehicle messages.
2. If a UV's protocol event handlers are in an idle state, then all of its completed `block`s must be committed to its local `block chain`.
3. No more than one copy of a particular `block` will be maintained on a UV at any time. A `block` can exist within the locally-maintained `block chain`, within a "waiting to be committed" data structure, or within a data structure associated with the reconcile process.
4. No `block` will exist within the block chain or any intermediate data structure that was not generated by a participating UV. This characteristic is partially assured by the UVS's underlying cryptographic system, for now.
5. In a fully connected system, all `block`s will eventually be committed to all locally maintained `block chain`s (i.e., blockchains maintained by each UV).

6. In a fully connected system where all agents have had the chance to reconcile block chains with each other, one uniform block chain will emerge.

Of these characteristics, 1 and 4 are trivially demonstrated by the protocol description itself. That is, the implementation is evidently distributed, event driven, and asynchronous; and blocks only enter the system through execution of the **Build Data Block** event handler (as described, the protocol relies on the underlying cryptographic implementation to assure the validity of the events to be logged). Characteristics 2, 3, 5, and 6 were confirmed by examination of the local block chains after completion of each experiment's final reconcile process.

To verify characteristic 2, local logs and block chains were analyzed to ensure that all committed blocks, whether locally generated or received from another agent, were present in the block chain. To verify characteristic 3, block chains were analyzed to ensure that only one copy of a block was present, and logs were analyzed to ensure that the blocks to be committed deque and reconcile stack data structures were empty. Characteristics 5 and 6 were verified during the SITL testing with no packet loss. In these tests, we observed that a single mutual chain emerged.

The remainder of this section discusses the results from each set of tests. As discussed in Section 4.2, each test was conducted with five UAVs and was allowed to run for 20 minutes before block generation was terminated. All UAV were then allowed to settle into an idle state (i. e., all commit and reconcile processes were allowed to complete). Logs and block chains were preserved following each experiment for analysis purposes.

Figures are provided to depict typical block chains that resulted from each group of tests. Block chains are depicted using tree representations of the system-wide results where branches indicate divergences among the local block chains. The diagrams should be read from bottom to top. The bottom box, labeled "inception block", represents the inception block as described in Section 3.4.1. Each subsequent box represents a generated block. Each box is labeled with two numbers separated by a period. The first number is the agent ID of the generating UAV, and the second is the ordinal number of the block for that agent. Boxes are also colored coded according to the agent ID. Cryptographic links between blocks are represented by connecting arrows. Finally, the dotted lines and ovals at the tree leaf nodes indicate the agent IDs of the UAVs on which that block chain was

observed. In some examples long, branchless `block chain` segments are abbreviated with ellipses to save space.

Perfect Communications Testing

The UCP implementation was first simulated with the probability of packet loss set to zero. As discussed in Section 4.2, perfect communications testing demonstrated that the commit block process was correctly implemented. As expected, a single `block chain` emerged. Log analysis indicated that none of the UAVs entered the reconcile process during this test, which was as expected. Since commits were universally received without fail, the results from subsequent `block` commit voting processes were always adjudicated as "approved." In cases where the `local chain` was altered while voting was in progress (i.e., a `block` was added), the voting process was restarted, as intended, until the `block` was successfully committed. A graph of the final `block chain` from this test is shown in Figure 4.1.

None of characteristics 2, 3, 5, and 6 was violated in these tests. Satisfaction of characteristics 2 and 3 was verified by noting that the `blocks to be committed deque` and `reconcile stack` data structures were empty, and that no `block` was in the `block chain` more than once. More importantly, characteristics 5 and 6 were only demonstrable in these tests, as they were the only tests with a fully connected communications system. Satisfaction of characteristic 6 was verified by the agreement of all local `block chain`s (i.e., a single system-wide `mutual chain`). That the `mutual chain` included exactly one copy of every `block` generated over the course of the experiment provided verification that characteristic 5 was satisfied as well.

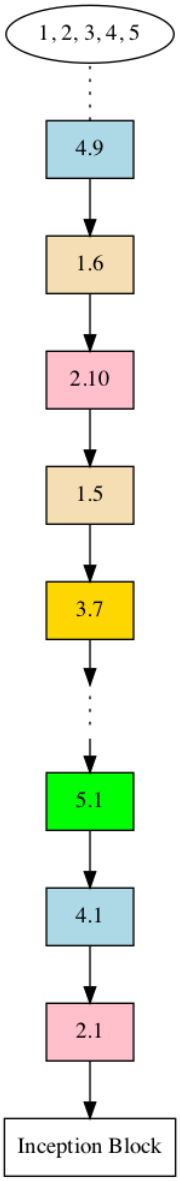

Figure 4.1. Typical final `block chain` from the no-packet-loss tests in which a single chain emerged.

Missed Commit Testing

Reconcile process functionality was first tested by probabilistically ignoring commit messages to induce divergences among the local `block chains`. As expected, during these tests

a single block chain emerged and the UAVs correctly completed the reconcile process. The reconcile process successfully brought the local UVS block chains into agreement with one small exception. In the example depicted in Figure 4.2, multiple successful reconciliations were conducted by system agents. At the conclusion of the experiment, however, agent 2 missed the commit for block "4.11", agent 4's final commit (the commit voting process was already in progress when event generation was terminated). Since no more blocks were generated, agent 2 settled into an idle state with its local block chain differing from the rest of the local UVS.

Characteristics 2 and 3 were not violated in these tests. The satisfaction of both was verified by confirmation that the blocks to be committed deque and reconcile stack data structures were empty and that no block was present in any local block chain more than once. Every block that was generated over the course of the experiment was determined to be present in at least one local block chain. Although characteristics 5 and 6 were not applicable because communications were not fully connected, a system-wide mutual chain did emerge with the exception of block 4.11. The lack of a full mutual chain does not violate any of the protocol's required characteristics and would be easily rectified were agent 2 to initiate a reconcile process.

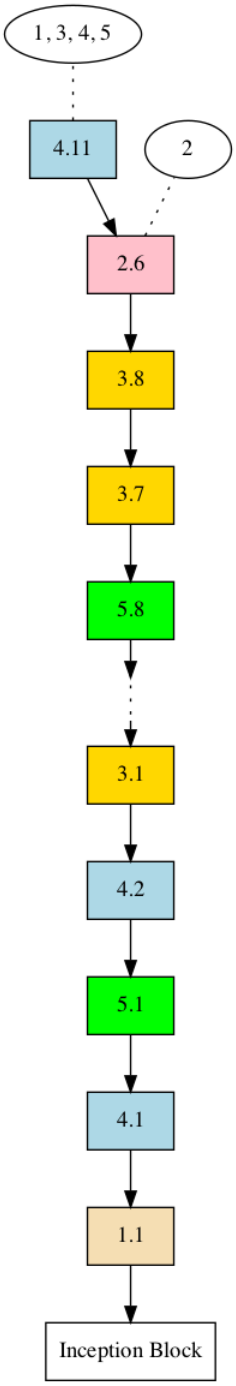

Figure 4.2. The final `block chain` from a random commit packet loss test in which a single chain emerged.

Lossy Communications Testing

Multiple tests were performed with the final UCP implementation in the SITL environment to simulate various levels of packet loss. This section discusses results of those tests to include identification of probabilities that demarcated noticeable changes in the resulting `block chain`s. These tests were intended to test the robustness of the UCP implementation in lossy-communications environments.

One of the main objectives of these tests was to observe the implementation's compliance with characteristics 2 and 3. That neither was violated in any of the lossy-communications tests was verified by two observations. First, the `blocks to be committed deque` and `reconcile stack` data structures were empty, and, second, no `block` was present in any local `block chain` more than once. In addition, it was noted that every `block` that was generated over the course of each experiment was present in at least one local `block chain`.

Strictly speaking, characteristics 5 and 6 were not relevant in these tests, because the UVS communications system was not fully connected. It is worth noting, however, that in tests with a probability of loss of 0.15, a uniform `block chain` did emerge. This was in contrast to tests with larger probabilities of loss, where a uniform `block chain` did not emerge.

All tests at and below 0.15 probability of packet loss settled into an idle state similar to the missed commit test; except for a few differences among the final `block`s, a uniform `block chain` emerged across the UVS. For the 0.15 tests. In the example depicted in Figure 4.3, `agent` 1's local `block chain` diverged due to timing. `Agent` 1 was close to the end of phase 2 of the reconcile process; however, before its phase 2 timer expired, `agent` 5 broadcast a `block` for commit. Even though `agent` 1 received the commit message, the fact that its commit process was still in progress prevented it from performing the new commit. `Agent` 1 never initiated another reconcile process, and was therefore never able to add `block` 5.7 to its `block chain`.

Increasing the probability of packet loss caused the `mutual chain` between all of the UAVs to shrink. In addition, the divergence of the `block chain`s was magnified as the difference between the `mutual chain` and the individual UAV's `block chain`s grew. This seemed to manifest because, given the increasing probabilities of packet loss, it was much more difficult for a UAV to align its local `block chain` with the rest of the UVS when its chain grew in length beyond the `mutual chain`.

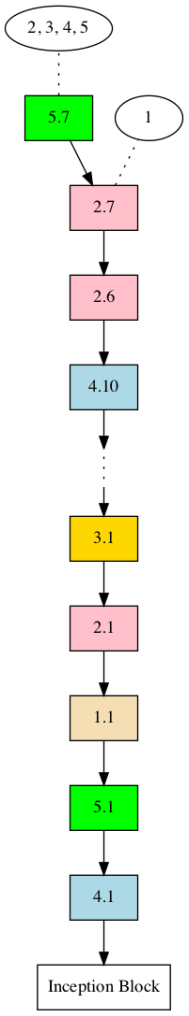

Figure 4.3. The final `block chain` from a typical 15 percent packet loss test in which a single `block chain` emerged.

This divergence became noticeable around 0.30 probability of packet loss and became more accentuated as the probability of packet loss increased. A graph of a typical final `block chain` from these tests is shown in Figure 4.4.

Interestingly, Figure 4.4 shows additional expected UCP behavior that was not observed in lower-loss-rate tests: the existence of `mutual chain`s among subsets of the UVS beyond the system-wide `mutual chain`, and `block`s being shared beyond the system-wide `mutual chain`. Both of these characteristics are byproducts of the reconcile process' reliance on a

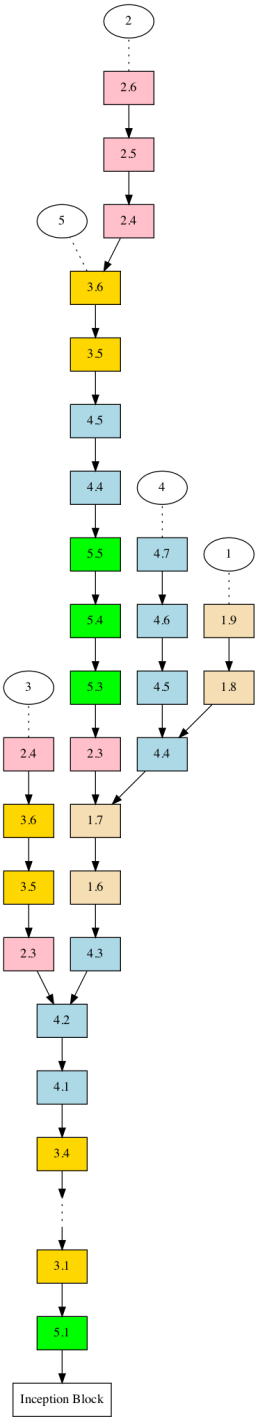

Figure 4.4. The final `block chain` from a 30 percent packet loss test in which multiple chains emerged.

local, rather than a system-wide, majority. Both are intentional and demonstrate the ability of the protocol to propagate reliable information across the system in lossy-communications systems in which full consensus cannot be achieved.

Figure 4.4 provides multiple examples of the existence of mutual chains among individual UAVs beyond the system-wide mutual chain. agent 3 has no mutual chain beyond block 4.2 (the system-wide mutual chain); however, agents 1, 2, 4, and 5 share a mutual chain through block 1.7, agents 1 and 4 share a mutual chain through block 4.4, and agents 2 and 5 share a mutual chain through block 3.6.

Figure 4.4 also presents multiple examples of the presence of shared blocks beyond the end of UAV pairs' mutual chain. Blocks 3.5 and 3.6 occur in the block chains of both agent 3 and agent 5 beyond their mutual chain. These blocks were passed from agent 3 to agent 2, and eventually from agent 2 to agent 5. Agent 3 did not reconcile directly with agent 5, but agent 3's blocks were indirectly propagated when agent 2 reconciled with agent 3 and later with agent 5. This also occurred when blocks 4.4 and 4.5 were shared between agents 4 and 5 and were subsequently propagated from agent 5 to agent 2.

Once the probability of packet loss reached around 0.50, it became difficult for any mutual chain to form. At this level of loss, thrashing was observed because agents received "disapprove" votes for practically every proposal. Early in the simulation a few blocks are propagated to other UAVs, but once the divergence was significant, it became impossible to perform the entire reconcile process. Typical results associated with this phenomenon are shown in Figure 4.5. The block chains, however, did not violate any protocol requirements, and characteristics 5 and 6 imply that a system-wide mutual chain would be obtained through the reconcile process if perfect communications were established.

When the probability of packet loss reaches 1.00 the system can be thought of as fully disconnected. No reconciliation processes are initiated because each agent receives only its own "approve" response to each commit request. Each local block chain, therefore, contains only the locally generated blocks as shown in Figure 4.6 where the mutual chain includes only the inception block.

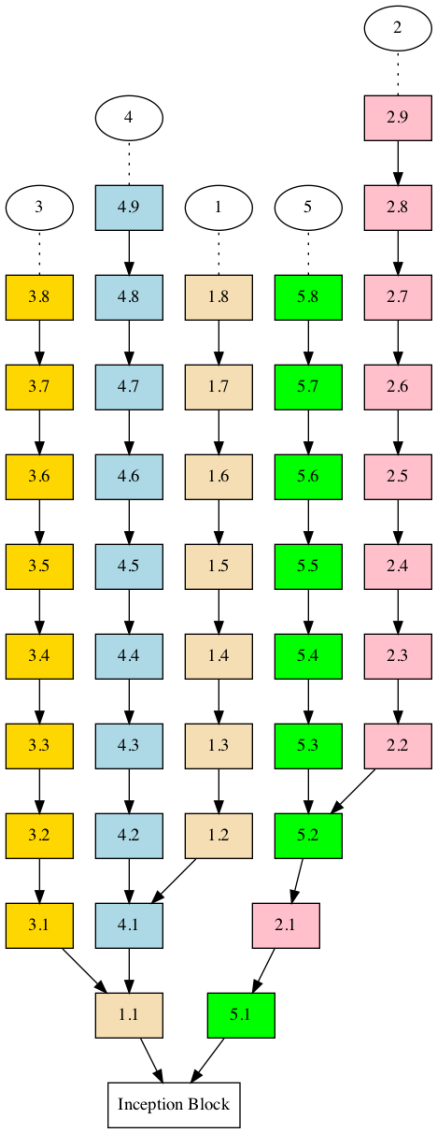

Figure 4.5. The final `block chain` from a 50 percent packet loss test in which multiple chains emerged.

Field Test Results

As discussed in Section 4.2, two UCP implementation field tests were conducted in November 2020. In the field tests the `agent`'s ID equated to the individual aircraft's tail number and does not relate to the number of vehicles flown in the experiment. IDs greater than 100 are IDs for Mosquito Hawk quadrotors, and lower IDs are for ZephyrII fixed wing vehicles. During the first experiment, `block chain`s were not saved because an implementation error caused event handler execution to end prematurely. This meant that the UAVs were

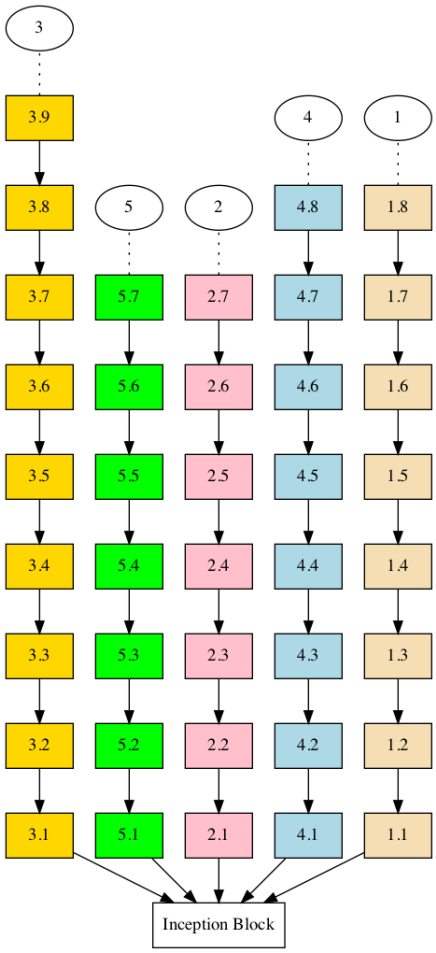

Figure 4.6. The final `block chain` from a 100 percent packet loss test in which each drone committed only the locally generated `blocks`.

not executing the UCP as designed, so the logs were largely irrelevant. The implementation error was corrected prior to the second field test.

Other implementation errors described in Section 4.2 manifested in the second experiment as discussed below, and since the UAVs were shut down before verifying that all vote and reconcile processes had concluded, it was understood that some commit and reconcile processes may have been terminated prematurely. Despite these difficulties, `block chain`s and logs from the second field test allowed for analysis. The discussion that follows is associated with results from this experiment. The `block chain`s obtained following this experiment are depicted in Figure 4.7.

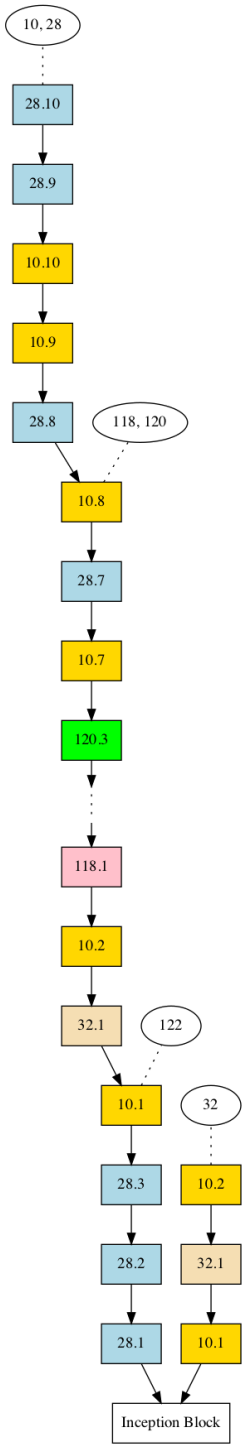

Figure 4.7. The final `block chain` from the second field test.

The most important result of this test was the successful demonstration of a UCP implementation in a live-fly environment. Satisfaction of UCP characteristics 2 and 3 were verified by noting that no block was present in any local block chain more than once and that the blocks to be committed deques and reconcile stacks were empty. In addition, log analysis indicated that all blocks that were generated by any UAV were present in at least one block chain except as affected by the bugs described below. There was no expectation that characteristics 5 or 6 would be verified, because the communications during the live-fly event were assumed to be imperfect.

Agents 10 and 28 did not encounter any errors during the experiment. These two agents exhibited correct UCP behavior for the entire mission.

During the experiment, other agents encountered the following errors: a syntax error when choosing the reconcile phase 2 majority response, and a logical error in the *PauseableTimer*.

Agent 32 and 122 exhibited a syntax error when choosing the majority response during reconcile phase 2. The error was traced to the **Reconcile Phase 2 Timer Expiration** event handler and occurred when the vote resulted in a tie. The event handler raised an exception at this point, and the reconcile process never completed. This resulted in the reconcileInProgress Boolean remaining true indefinitely and prevented the UAVs from participating in any further UCP activities, to include committing locally-generated blocks to their own block chains. Note in the figure that the block chains from these two agents are truncated significantly.

Agents 118 and 120 exhibited the *PauseableTimer* logic error early in their execution, this caused the agents to terminate reconcile phase 3 prematurely. This did not cause a violation of any required UCP characteristics, but it did lead to the mutual chain not being extended as much as it could have been. This error did not cause the agents to enter an unrecoverable state, and they were able to continue participating in UCP activities. As a result, their block chains align with those of agents 10 and 29 through block 10.8.

The final logical error, that the **Receive Response Block for Hash** event handler did not recalculate the chain hash, was found while analyzing the cause of the previous errors. This error did not manifest during our tests. However, this error would lead to unpredictable results during the commit process, as the chain hash is used to validate the addition of

`blocks` to the `block chain`.

Except for the implementation errors described above, the `block chain` and UCP implementation appeared to work correctly.

4.4 Chapter Summary

This chapter covered the UCP development process, and tests and experiments conducted with the implementation. Analysis was performed to identify and correct implementation bugs and to ensure the desired UCP characteristics were satisfied. UCP development included implementation of all event handlers associated with the voting, commit block, and reconcile processes described in Chapter 3. Experiments conducted in both the SITL simulation environment and in live field experiments were described, and the analysis of the results was discussed. Implementation bugs were identified and corrected at various stages of the development and experimentation process. Importantly, no flaws in the overall design of the UCP were identified, and the final implementation appears to perform as desired. The next chapter summarizes our research and discusses possible future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion

This thesis described development and implementation of the Uniform Chain Protocol (UCP), a protocol supporting data availability in a multi-vehicle, distributed, autonomous system. The UCP uses a block chain to provide for integrity and availability of a distributed database and employs a reconcile process to ensure the consistency of the data structure among the system's UVs. Distribution of the data across the UVS supports the survivability of data in scenarios where the UV that generated the data is lost or destroyed. Blockchain features are used to achieve consensus among the local UVS agents and to ensure database consistency in lossy communications environments. The UCP was implemented in Python as a ROS node for use on the ARSENL multi-UAVs system. Testing was performed with bench simulations using the SITL testing framework and in live-fly field tests at Camp Roberts in southern Monterey County, CA. These tests demonstrated the functionality and usability of the UCP.

Although the UCP was implemented on a UAV system, it is not specifically limited to this type of system. The UCP is suitable for deployment on any UVS that has inter-UV network communication. The UCP was designed for UVSs expected to experience lossy and disjoint network communications and vehicle loss. The Department of Defense (DOD) operates many such systems where communications are limited and the risk of UV loss is assumed [33] [34] [35] [36] [37]. With the addition of the UCP to such a UVS, generated or recorded data can be available for post-mission analysis without requiring the survival of every originating UV. The code developed for this thesis is archived on NPS's GitLab website in the ARSENL swarm-autonomy repository.

5.1 Future Work

This thesis covered creation and prototype implementation of the UCP, yet there are many areas for follow-up work. These include:

Additional Testing - The testing performed in this thesis was not comprehensive. Tests were performed with a small set of UVs to allow manual analysis of the generated

log files. While we hypothesize that the UCP will perform satisfactorily with larger UVSs, this has not been demonstrated or formally proven. Additional testing with more vehicles could demonstrate and characterize the protocol's scalability. This work would include the development of tools to augment manual analysis.

Testing Framework Improvements - Testing performed in this thesis did not simulate disjoint networks (i.e., conditions under which subsets of the UVS separate and recombine). Performing such tests would demonstrate a key aspect of UCP's reconciliation process: the generation of a system-wide mutual chain by deconstructing and rebuilding competing subsystem mutual chains.

Encryption and Authentication - Currently the UCP relies on the underlying system to ensure the data recorded in a block is encrypted and signed. Adding encryption and signing keys to the UCP would enable the protocol to function on systems that do not satisfy that assumption, allowing it to run on a wider variety of systems to possibly include UVSs run by coalition forces.

Support Tools - The work documented in this thesis did not attempt to address implementation details of the protocol's use with real-world systems. Among other systems and software engineering topics, future work should include the development of tools to consolidate and visualize stored data for post mission analysis, to create and deploy mission-specific configurations (e.g., the inception block) to swarm participants, to download and perform a final reconcile process to combine all of the block chains when UVs return to the depot, and to provision keys when encryption and authentication are incorporated to the UCP.

Identification of Complexity Characteristics - Space and time complexity were not taken into account when developing the implementation. Formal characterization of the protocol's complexity characteristics could identify bottlenecks and lead to significant improvements.

List of References

[1] M. Hancock and E. Vaizey, *Distributed Ledger Technology: Beyond Block Chain*. The National Archives, London, 2016. Available: https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf

[2] J. Mattis. (2018). Summary of the 2018 National Defense Strategy. Department of Defense. [Online]. Available: https://dod.defense.gov/Portals/1/Documents/pubs/2018-National-Defense-Strategy-Summary.pdf. Accessed Dec. 2020.

[3] M. Hati, "Swarm robotics: A technological advancement for human-swarm interaction in recent era from swarm-intelligence concept," 2016.

[4] G. Francesca and M. Birattari, "Automatic design of robot swarms: Achievements and challenges," *Frontiers in Robotics and AI*, vol. 3, 2016.

[5] S. B. Heppe, "Problem of UAV communications," in *Handbook of Unmanned Aerial Vehicles*, K. P. Valavanis and G. J. Vachtsevanos, Eds. Dordrecht: Springer Netherlands, 2015, pp. 715–748. Available: https://doi.org/10.1007/978-90-481-9707-1_30

[6] S. S. Ponda, L. B. Johnson, A. Geramifard, and J. P. How, "Cooperative mission planning for multi-UAV teams," in *Handbook of Unmanned Aerial Vehicles*, K. P. Valavanis and G. J. Vachtsevanos, Eds. Dordrecht: Springer Netherlands, 2015, pp. 1447–1490. Available: https://doi.org/10.1007/978-90-481-9707-1_16

[7] Z. Lau, Dylan, "Investigation of coordination algorithms for swarm robotics conducting area search," M.S. thesis, Naval Postgraduate School, 2015. Available: https://calhoun.nps.edu/handle/10945/47293

[8] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis, and M. Jones, "Live-fly, large-scale field experimentation for large numbers of fixed-wing UAVs," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, A. Okamura, Ed. Stockholm, Sweden: IEEE, 2016, pp. 1255–1262.

[9] J. Elston, M. Stachura, C. Dixon, B. Argrow, and E. W. Frew, "Layered approach to networked command and control of complex UAS," in *Handbook of Unmanned Aerial Vehicles*, K. P. Valavanis and G. J. Vachtsevanos, Eds. Dordrecht: Springer Netherlands, 2015, pp. 781–811. Available: https://doi.org/10.1007/978-90-481-9707-1_33

[10] W. Zhao, *Building Dependable Distributed Systems*, 1st ed. (Performability Engineering Series.). Somerset: Wiley, 2014.

[11] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, no. 1, p. 77–97, Jan. 1987. Available: https://doi.org/10.1145/7531.7533

[12] E. Semsar-Kazerooni and K. Khorasani, *Team Cooperation in a Network of Multi-Vehicle Unmanned Systems: Synthesis of Consensus Algorithms*, 1st ed. New York, NY: Springer-Verlag, 2013.

[13] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[14] Industrial Internet Consortium. (2020). Distributed Ledgers in IIoT. [Online]. Available: https://www.iiconsortium.org/pdf/Distributed_Ledgers_in_IIoT_White_Paper_2020-07-22.pdf. Accessed Dec. 2020.

[15] S. Kadam, "Review of distributed ledgers: The technological advances behind cryptocurrency," in *International Conference Advances in Computer Technology and Management (ICACTM)*, March 2018. Available: https://www.researchgate.net/publication/323628539_Review_of_Distributed_Ledgers_The_technological_Advances_behind_cryptocurrency

[16] NARA Blockchain White Paper. (2019, Feb.). National Archives and Records Administration. [Online]. Available: https://www.archives.gov/files/records-mgmt/policy/nara-blockchain-whitepaper.pdf. Accessed Dec. 2020.

[17] J. E. Silva, "An overview of cryptographic hash functions and their uses," *GIAC*, vol. 6, Jan 2003. Available: https://www.sans.org/reading-room/whitepapers/vpns/overview-cryptographic-hash-functions-879

[18] RSA Laboratories. What is a hash function? [Online]. Available: https://web.archive.org/web/20051230111857/http://www.rsasecurity.com/rsalabs/node.asp?id=2176. Accessed Dec. 2020.

[19] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *International Workshop on Fast Software Encryption*. Springer, 2004, pp. 371–388. Available: https://web.archive.org/web/20081230145507/http://www.inf.unisi.ch/faculty/shrimpton/relates-full.pdf

[20] S. Nakamoto, "A peer-to-peer electronic cash system," *Bitcoin*, vol. 4, 2008. Available: https://bitcoin.org/bitcoin.pdf

[21] ROS Developers. (n.d.). ROS Documentation: Introduction. [Online]. Available: http://wiki.ros.org/ROS/Introduction. Accessed Dec. 2020.

[22] ROS Developers. (n.d.). Why ROS? [Online]. Available: https://www.ros.org/is-ros-for-me/. Accessed Dec. 2020.

[23] ROS Developers. (n.d.). ROS Documentation. [Online]. Available: http://wiki.ros.org/rospy. Accessed Dec. 2020.

[24] S. Tarkoma, *Publish/Subscribe Systems: Design and Principles* (Wiley series on communications networking & distributed systems). London: John Wiley & Sons, 2012.

[25] ROS Developers. (n.d.). ROS Documentation: Topics. [Online]. Available: http://wiki.ros.org/Topics. Accessed Dec. 2020.

[26] C. Chen, Y. Tock, and S. Girdzijauskas, "BeaConvey: Co-design of overlay and routing for topic-based publish/subscribe on small-world networks," in *Proceedings of the 12th ACM International Conference on distributed and event-based systems* (DEBS '18). ACM, 2018, pp. 64–75.

[27] ROS Developers. (n.d.). ROS Documentation: Publishers and Subscribers. [Online]. Available: http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers. Accessed Dec. 2020.

[28] M. A. Day, M. R. Clement, J. D. Russo, D. Davis, and T. H. Chung, "Multi-UAV software systems and simulation architecture," in *2015 International Conference on Unmanned Aerial Systems*. Denver, CO: IEEE, 2015, pp. 426–435.

[29] D. Davis, K. Jones, M. Jones, and K. Giles, "Advanced swarm UAV capabilities through collaborative field experimentation," Naval Postgraduate School, Monterey, CA, Tech. Rep., Apr 2018.

[30] ArduPilot Documentation. (n.d.). ArduPilot Dev Team. [Online]. Available: https://ardupilot.org/ardupilot/index.html. Accessed Dec. 2020.

[31] MAVLink Developer Guide. (n.d.). Dronecode Project. [Online]. Available: https://mavlink.io/en/index.html. Accessed Dec. 2020.

[32] L. Carter, Nickolas, "Design and informal verification of a distributed ledger protocol for distributed autonomous systems using Monterey Phoenix," M.S. thesis, Naval Postgraduate School, 2020.

[33] A. Jarocki, "US Navy funds underwater drone swarms," *Defense News*, 2020. Available: https://www.defensenews.com/unmanned/2018/06/26/us-navy-funds-underwater-drone-swarms/

[34] T. Chung. (2020). OFFensive Swarm-Enabled Tactics (OFFSET). DARPA. [Online]. Available: https://www.darpa.mil/program/offensive-swarm-enabled-tactics. Accessed Dec. 2020.

[35] J. Keller, "The US Navy could soon operate drone swarms—underwater," *The National Interest*, 2020. Available: https://nationalinterest.org/blog/reboot/us-navy-could-soon-operate-drone-swarms\T1\textemdashunderwater-165308

[36] D. Lafontaine. (2020). Army Looks to Enhance Mission Command with Robotic Swarms. US Army. [Online]. Available: https://www.army.mil/article/226268/army_looks_to_enhance_mission_command_with_robotic_swarms. Accessed Dec. 2020.

[37] E. Bone and C. Bolkcom, *Unmanned Aerial Vehicles: Background and Issues for Congress*. Congressional Research Service, 2003. Available: https://apps.dtic.mil/sti/pdfs/ADA467807.pdf

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California