NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FORMAL PROOF AND PERFORMANCE TESTING OF THE UNMANNED VEHICLE SYSTEM LOGGING PROTOCOL

by

Dalton L. Duvio

September 2023

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

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC, 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2023	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE FORMAL PROOF AND PERFORMANCE TESTING OF THE UNMANNED VEHICLE SYSTEM LOGGING PROTOCOL			**5. FUNDING NUMBERS**
6. AUTHOR(S) Dalton L. Duvio			
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)

Multivehicle autonomous systems are becoming increasingly relevant, but the loss of individual vehicles and unstable communications environments make maintenance of data logs for mission reconstruction and analysis difficult. A blockchain-based distributed ledger protocol, the Unmanned Vehicle System Logging Protocol (UVSLP), has been proposed to address this problem. Formal proofs of its correctness and controlled experiments examining its performance are required to use the proposed protocol with higher confidence. This work documents the formal definition of the protocol using predicate logic with added temporal operators and provides formal proofs of the protocol's claimed properties. In addition, we use a prototype implementation in a simulated environment to characterize the protocol's performance for specific network loss rate, event generation rate, and number of vehicles values. Results are presented in the form of number of messages and volume of data produced by the protocol for each configuration. Our results show that message volume increases no worse than linearly as the number of vehicles and event generation rates increase and decreases as network loss rates increase. The culmination of this thesis is a mathematically proven protocol with known performance characteristics that improves the resiliency and availability of drone swarm data.

14. SUBJECT TERMS Unmanned Vehicle System Logging Protocol, UVSLP, swarm, formal proof, blockchain, distributed ledger			**15. NUMBER OF PAGES** 89
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** 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.

FORMAL PROOF AND PERFORMANCE TESTING OF THE UNMANNED VEHICLE SYSTEM LOGGING PROTOCOL

Dalton L. Duvio
Civilian, CyberCorps: Scholarship for Service
BA, Stanford University, 2016

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2023

Approved by: Duane T. Davis
Advisor

Cynthia E. Irvine
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Multivehicle autonomous systems are becoming increasingly relevant, but the loss of individual vehicles and unstable communications environments make maintenance of data logs for mission reconstruction and analysis difficult. A blockchain-based distributed ledger protocol, the Unmanned Vehicle System Logging Protocol (UVSLP), has been proposed to address this problem. Formal proofs of its correctness and controlled experiments examining its performance are required to use the proposed protocol with higher confidence. This work documents the formal definition of the protocol using predicate logic with added temporal operators and provides formal proofs of the protocol's claimed properties. In addition, we use a prototype implementation in a simulated environment to characterize the protocol's performance for specific network loss rate, event generation rate, and number of vehicles values. Results are presented in the form of number of messages and volume of data produced by the protocol for each configuration. Our results show that message volume increases no worse than linearly as the number of vehicles and event generation rates increase and decreases as network loss rates increase. The culmination of this thesis is a mathematically proven protocol with known performance characteristics that improves the resiliency and availability of drone swarm data.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Motivation . . . 2
1.3 Scope . . . 2
1.4 Organization . . . 3

2 Background **5**
2.1 Multivehicle Autonomous Systems . . . 5
2.2 The Unmanned Vehicle System Logging Protocol . . . 6
2.3 Proving Concurrent Processes . . . 11
2.4 Chapter Summary . . . 14

3 Formal Proofs **15**
3.1 Common Definitions . . . 17
3.2 Event Handlers . . . 20
3.3 Supplemental Definitions . . . 30
3.4 Proofs . . . 31
3.5 Chapter Summary . . . 51

4 Performance Testing **53**
4.1 Background . . . 53
4.2 Experimental Design . . . 54
4.3 Results . . . 57
4.4 Chapter Summary . . . 63

5 Conclusion **65**
5.1 Future Work . . . 65

List of References **67**

Initial Distribution List **71**

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

Figure 2.1 UVSLP block generation and commit process 8

Figure 2.2 UVSLP blockchain reconcile process 10

Figure 4.1 Correlation between number of messages and data volume 57

Figure 4.2 Average number of messages by network loss rate 58

Figure 4.3 Average data volume by network loss rate 59

Figure 4.4 Average number of messages by mean time between events . . . 60

Figure 4.5 Average data volume by mean time between events 61

Figure 4.6 Distribution of message types at 0 percent network loss rate and 60 second mean time between events 62

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 3.1 Extended logic operators used in proofs 16

Table 3.2 Assumed sets and sequences of UVSLP 16

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

API application programming interface

ARSENL Advanced Robotic Systems Engineering Laboratory

deque double-ended queue

DLP distributed ledger protocol

MP Monterey Phoenix

NPS Naval Postgraduate School

ROS Robot Operating System

SITL software-in-the-loop

UAV unmanned aerial vehicle

UDP User Datagram Protocol

UV unmanned vehicle

UVSLP Unmanned Vehicle System Logging Protocol

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 and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Experimentation with autonomous multivehicle systems, or drone swarms, for military operations has increased in recent years. Due to the potential loss of individual drones and the unstable communications environment inherent to their operation, there is a high risk of the loss of important data maintained within these systems. To address this issue, Naval Postgraduate School (NPS) researchers proposed a blockchain-based distributed ledger protocol called the Unmanned Vehicle System Logging Protocol (UVSLP) [1], [2]. This research provides formal proof of this protocol's correctness. Further, this research investigates its general performance and scalability characteristics.

1.1 Problem Statement

The UVSLP was developed to satisfy six specific properties. The initial presentation of the protocol included an informal verification of these properties using Monterey Phoenix (MP) [1], [3]. Although this work played an important step in validating the protocol's correctness, MP is inherently limited to checking models of behavior through empirical testing [4]. As a result, the completeness of this informal verification cannot be ensured. In separate initial work on UVSLP, an implementation of the protocol on the NPS Advanced Robotic Systems Engineering Laboratory (ARSENL) unmanned aerial vehicle swarm system was developed [2]. This prototype was tested in live-flight experiments and in a physically-based software-in-the-loop (SITL) simulation environment [5], [6]. However, this testing only verified the protocol's functionality; it did not include characterization of the general performance and scalability of the protocol.

This research addresses the following questions:

1. Does the protocol, as specified, satisfy each of the six asserted properties?
2. How much message traffic (i.e., number of messages and volume of data) is generated by the protocol as a function of communications loss rate?
3. How do the protocol's communications requirements scale as vehicles are added to the system?

4. How does the mean time between logged events affect message traffic?

1.2 Motivation

The motivation for this research is derived from the 2022 *National Defense Strategy*, a directive that encourages strengthening resiliency in the domain of cyberspace. In particular, the strategy states, "We will strengthen strategic stability through dialogue with competitors, unilateral measures that make command, control, and communications more robust, and by developing defenses and architectural resilience to maintain operational capabilities in cyberspace and space during conflict" [7]. This research investigates the suitability of a protocol designed to address the resilience of mission-critical data obtained during the operation of multi-vehicle autonomous systems.

1.3 Scope

This thesis uses predicate calculus to prove the correctness of each of the six formal properties of the UVSLP. The result is a formal proof of the correctness of UVSLP as originally specified. This research does not prove anything beyond the six formal properties of the protocol.

This thesis investigates the general performance and scalability characteristics of the protocol given best-case and worst-case scenarios. These scenarios are limited to factors directly related to the protocol and its prototype implementation, such as the number of vehicles and their connectivity. This research does not investigate extraneous factors generally related to unmanned vehicle operation, nor does it provide a complete mathematical characterization of the protocol's performance.

The major contributions of this research are:

- a set of formal proofs that the protocol is correct with regards to its original specification, and
- quantifiable results on how the protocol performs and scales across various numbers of vehicles, network loss rates, and event generation rates with regards to message traffic.

This work advances the suitability of drone swarms for military operations with a mathematically-proven protocol with known performance characteristics that improves the resiliency and availability of maintained data.

1.4 Organization

This thesis contains five chapters, including this introduction. Chapter 2 provides the following necessary background information: a description of multivehicle autonomous systems, a high-level overview of the UVSLP and its six formal properties, and a discussion on the methods used to prove concurrent systems. Chapter 3 presents the formal proofs of the correctness of the protocol in regards to the six properties. In addition to the proofs themselves, the chapter first examines and formally defines the various event handlers and behaviors of the protocol. Chapter 4 provides the methodology and results of the testing conducted to characterize the general performance and scalability of the protocol. Finally, Chapter 5 reviews the work completed for this thesis and discusses directions for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

Multivehicle autonomous systems are becoming increasingly popular. One subset of such systems is a swarm, which is composed of multiple autonomous vehicles that rely on simple interactions to produce useful group behaviors. However, communication between vehicles remains a major challenge for the success of swarm systems. To address this challenge, NPS researchers have proposed the UVSLP, a blockchain-based distributed logging protocol that ensures the consistency of data across the participating vehicles. The UVSLP takes a different approach from most blockchain-based distributed ledgers by allowing possibly disparate locally maintained blockchains to periodically reconcile with neighboring vehicles. This chapter discusses the challenges that multi-vehicle autonomous systems, especially swarm systems, face and how the UVSLP helps address them. Additionally, this chapter discusses the challenges of proving correctness of concurrent systems and provides background on the general approach that will be used for proving the properties of the UVSLP.

2.1 Multivehicle Autonomous Systems

The term multivehicle autonomous system refers to an unmanned vehicle (UV) system in which multiple vehicles operate cooperatively in a decentralized manner. *Decentralized* in this context means that there is no singular vehicle or centralized controller that coordinates all vehicles in the system. Instead, each vehicle operates independently and autonomously. Each vehicle interprets its sensor readings, maintains its own situational awareness, and determines its own courses of action. Data is exchanged between vehicles only as required, and mission-related events are observed and logged locally.

Swarm systems make up one subset of the much larger multivehicle system domain. Swarm systems are characterized by large numbers of vehicles whose simple interactions result in useful group behaviors. Vehicles in a swarm may share data with other swarm members to support group coordination and decision-making, or individual decisions can be based solely on locally obtained data. The collective behaviors that emerge, commonly referred to as *swarming*, allow relatively simple UVs to perform complex behaviors as a group [8].

Swarm systems operate collectively as a result of the individual vehicles' perceived local environments, any intervehicle communication, and the resulting individual actions. In this way, swarms are both decentralized and self-organizing and do not rely on external infrastructure [9].

Multivehicle autonomous systems encounter many challenges during operation. One of the most important, especially in swarms, is intervehicle communication. System dynamicity, potential loss of vehicles, and signal noise associated with the environments in which the swarms operate all affect both the reliability and resiliency of communications between vehicles [6], [10]. These limitations present a high risk of the loss of important mission data maintained within these systems.

The range between vehicles in a swarm is highly dynamic due to vehicle maneuvering, and vehicles may not always be within communication range of one another. Since each vehicle operates independently, individual vehicles may disengage from the rest of the swarm, and swarms may also divide into subswarms that transit to different locations. Thus, the network can be subject to segmentation in these situations. Further, communication signals, even within connected network segments, are subject to noise and dropped packets. As a result of the segmentation, communication channels can become so unreliable that transmissions may not be received in full or at all, and messages might not be received by all vehicles even in a completely connected swarm.

2.2 The Unmanned Vehicle System Logging Protocol

The UVSLP is a blockchain-based distributed ledger protocol (DLP) proposed by NPS researchers to address the challenges of mission log maintenance for swarms [11]. Specifically, the protocol is designed to immutably share data across multiple platforms in order to maximize the likelihood that it is retained in the event that the generating UVs fail or are lost prior to recovery.

Distributed ledgers are a form of data storage that maintains data across multiple locations [12]. A consensus process is used to synchronize and ensure the consistency of the data across the participating nodes. While distributed ledgers are often associated with cryptocurrencies, they have been shown to be suitable as a general-purpose data storage technology in a number of additional domains [13]. In the context of swarms, the distri-

bution and duplication of data can enhance the resiliency against potential data loss [11]. This will allow more thorough mission reconstruction and analysis following the recovery of surviving vehicles.

Distributed ledgers are commonly implemented in the form of immutable blockchains. Blockchains encapsulate data in blocks that are cryptographically linked using hash function digests to ensure their integrity. Each block in the chain includes the hash digest of the previous block in the chain. This provides an efficient means of comparing the consistency of two instances of a blockchain.

Most blockchain-based distributed ledger implementations are additive-only (i.e., blocks are never removed from the chain once added) and attempt to construct a single agreed-upon blockchain before appending a new block. The UVSLP, however, takes a different approach in an attempt to account for the highly dynamic communications environment for which it is intended. Instead of attempting to maintain a single "correct" blockchain, the protocol calls for possibly disparate, locally maintained blockchains that are reconciled as needed with neighboring blockchains. During the reconciliation process, blocks are temporarily removed from a local blockchain to allow the addition of missing blocks from neighboring blockchains. Removed blocks are re-added to the local blockchain and provided to neighboring nodes after the missing blocks are added.

DLPs maintain consistency across the system through a process shared among nodes known as *consensus*. The DLP's consensus algorithm accounts for temporary discontinuities and unreliable communications. The Paxos family of consensus algorithms provides the basis for many blockchain DLP approaches to consensus. These algorithms enable a distributed system eventual consensus despite unreliable communications or malfunctioning nodes [14], [15]. Rather than ensuring consensus at any specific moment, Paxos ensures that progress is made toward consensus over time.

Paxos is implemented by agents (i.e., processes) that take on three roles: *proposer*, *acceptor*, and *learner* [14]. In most implementations, every agent can assume any or all of the roles as required. A proposer with data to be added to the blockchain prepares a proposal and submits it to the acceptors. Each acceptor will either accept or reject the proposal. If a majority of acceptors accept the proposal, the data is formally added to the ledger by the learners.

While the UVSLP is based on Paxos, it cannot directly implement Paxos as specified because swarm losses and additions and network segmentation and unreliability make the size of the swarm, and, therefore, what constitutes a majority unknowable [11]. Instead, UVSLP builds upon the concepts provided by Paxos, but makes additions to the local blockchains based on a local majority of the agents that respond to broadcast messages. As the swarm consolidates, divergent blockchains are reconciled to achieve consensus across the previously disjoint system.

2.2.1 Block Generation, Block Commit, and Blockchain Reconciliation

The UVSLP is specified as a set of 20 event handlers implemented by each vehicle [1], [2]. These handlers are triggered asynchronously by both internal and external events. They are organized into two overarching processes: *block generation and commit*, and *blockchain reconcile*.

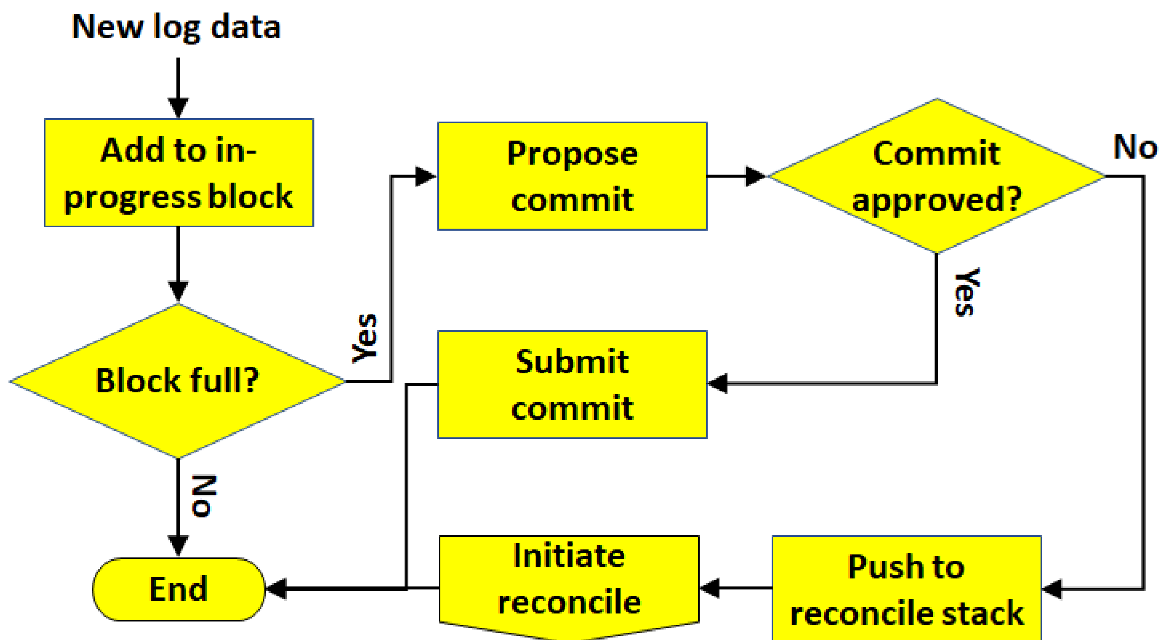

Figure 2.1. UVSLP block generation and commit process. Source: [3, figure 1]

Vehicles create blocks from log data and distribute those blocks for addition to the blockchains of other vehicles through the *block generation and commit* process, as il-

lustrated at a high-level in Figure 2.1. Block generation refers to the creation of blocks from loggable events generated by the vehicle on which the agent is executing. As log data is generated, it is added to an in-progress block. When the in-progress block is full, the agent will broadcast a request to all available vehicles to approve the addition of the new block to the blockchain. If a majority of responses approve the addition, the block is committed to the agent's local blockchain and broadcast to all available agents for commitment to their local blockchains. If a majority of responses are rejections, the block is instead pushed to a local *reconcile stack* and the *blockchain reconciliation* process is triggered.

A majority of disapproval responses to a proposal request indicates that the proposing agent's locally maintained blockchain has diverged from those of its neighbors. The local blockchain of the agent whose proposal was rejected is brought into agreement with the rest of the connected system through the *blockchain reconcile* process. A high-level overview of this process is provided by Figure 2.2 and consists of three steps. The first identifies the point of divergence between the locally maintained blockchain and the blockchains of neighboring agents. The agent queries neighboring agents to determine if any of their blocks have a chain hash that matches the chain hash of the last block of its local blockchain. If a majority do not, it removes the last block from its blockchain and pushes it to a *reconcile stack*. This step repeats until the local blockchain is reduced to a common blockchain that a majority of neighbors share.

Once the point of divergence has been identified, missing blocks are added to the local blockchain. The agent queries neighboring agents for the next block in the sequence for the common blockchain and choose the most common block from the responses. The agent adds this block to its local blockchain either from the *reconcile stack* if it contains the block or by querying neighboring agents for the block if it does not. This step repeats until it no longer receives responses with the next block in the sequence.

Finally, the agent commits any remaining blocks in the *reconcile stack* to the local blockchain. It pops a block from the stack and progresses through the block commitment process as if it were a newly generated block. This step repeats until the *reconcile stack* is empty.

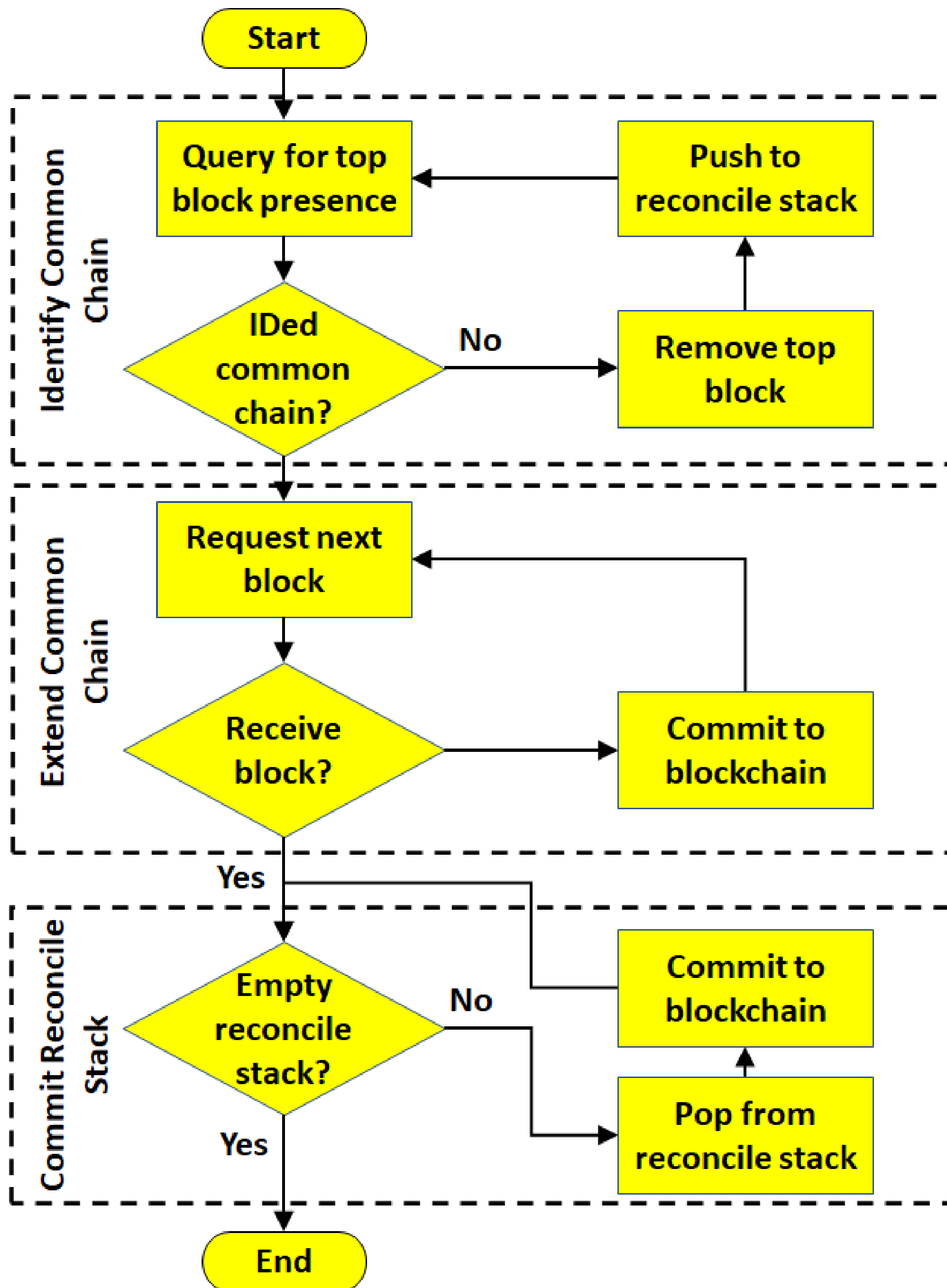

Figure 2.2. UVSLP blockchain reconcile process. Source: [3, figure 2]

2.2.2 Properties of UVSLP

UVSLP and its event handlers were developed to satisfy six formal properties to achieve its goals [3]:

1. **Block creation** – No block will exist within the system that was not proposed by a participating agent.
2. **No block duplication** – No more than one copy of a particular block will be maintained by an agent at any time. A block can be present in the local blockchain or in a temporary data structure associated with the protocol implementation.
3. **No block loss** – All blocks proposed by participating agents will be maintained by at least one agent, vehicle loss notwithstanding.
4. **Idle stop** – If an agent's protocol event handlers are in an idle state, then all blocks maintained by that agent must be present in the local blockchain.
5. **Block propagation** – In a fully connected system, all blocks will eventually be committed to all locally maintained blockchains.
6. **Uniform chain** – In a fully connected system where all agents have opportunities to reconcile blockchains with each other, one uniform blockchain will emerge.

The first four properties are associated with block generation and block maintenance. These properties assert that any particular block will be one proposed by a participating agent and it will not be erroneously duplicated or lost. Additionally, if an agent's event handlers are in an idle state, any particular block maintained by that agent will be committed to its local blockchain and not stored in a temporary data structure. The last two properties are associated with blockchain reconciliation and the distribution of blocks in a fully connected system. These properties assert that in a non-disjoint system all individual blockchains will eventually contain a copy of every proposed block and that they will eventually be reconciled to a single, uniform blockchain.

2.3 Proving Concurrent Processes

In the early days of computing, systems were developed to run sequentially in what was referred to as *batch* mode. That is, processes executed one at a time, and one statement executed after another in a completely linear fashion. As computer science progressed, concurrent systems in which multiple processes or threads executed simultaneously were

developed. Concurrent systems differ from batch systems because concurrent processes may interact with one another, may share data and resources, and may require synchronization. As a result, proving the correctness of concurrent systems with the methods commonly used for sequential systems can be difficult.

A common approach to proving the correctness of sequential processes is to describe the process as a state transformer in which the system starts in one state and transforms to another state before terminating [16]. Envisioned this way, a correctness proof takes the form of separate assertions for each program statement or control point that are proved individually using formal classical logic. Collectively, this demonstrates that the initial state leads to the final state before the program terminates.

While this state-based, assertional approach has been extended to concurrent processes, it is limited in two aspects. First, it is difficult to apply to higher level abstractions due to the reliance on details of a particular implementation. Assertions are tied to atomic operations and correctness conditions must be expressed in terms of implementation-specific objects, such as variables. Second, it has historically only been found to be useful for non-distributed concurrent processes that communicate via shared memory or that do not communicate at all [17]. This is because the order of events cannot be always assumed in distributed concurrent processes where an operation A in one process may not always precede some operation B in another process.

Although processes are typically classified as either sequential or concurrent, Lamport argues that, in terms of proving correctness, it is more appropriate to classify them as either *functional* or *reactive* [16]. A *functional system* is one that transforms an initial state to a final state, or maps an input to an output. All sequential processes fall under this category, and some concurrent processes do as well, particularly those that do not communicate with other processes. For systems that can be classified as functional, a state-based, assertional approach using classical logic is appropriate, especially if the intent is to prove the correctness of a particular implementation.

A *reactive system* is one that cannot be described strictly as a state transformer or a mapping from input to output. Rather, a reactive system interacts with its environment in more complex ways due to asynchronous communication with other processes. While only the initial and final states need to be considered for *functional systems*, one must consider

intermediate states over time for *reactive systems*. Many distributed, concurrent systems, including the UVSLP, fall into this category.

For *reactive systems*, a more appropriate approach is to extend classical logic to describe the temporal behavior of a program. That is the properties of a program are qualified in terms of time. This is referred to as temporal logic [18], [19]. Unlike the classical logic used in state-based, assertional approaches, temporal logic is suitable for both sequential and concurrent processes.

Each property of a program, or protocol in the case of this thesis, can be characterized as one of two fundamental types: a *safety property* or a *liveness property* [16], [18], [20]. A *safety property* is one that asserts that a program never does something bad. In other words, a *safety property* is a property that holds true for every sequence of states of that program, thus it will hold true for the entire execution of that program. Functional systems can be specified in terms of safety properties by asserting they hold true in reachable state. A *liveness property*, on the other hand, states that something good will eventually happen or that the property is eventually true over time. Since a liveness property is not assumed to hold true universally, there is no requirement to assert its truth over all reachable states. Since reactive systems interact with other processes asynchronously, they are often specified in terms of a combination of the two types.

Temporal logic allows one to describe and reason about both types of properties. To prove a liveness property, one must prove the eventuality of that property. The component of time (i.e., eventuality) associated with liveness properties is why temporal logic is especially well-suited for proving these types of properties. To prove a safety property, one must prove the invariance of that property, which can be expressed in terms of time as holding true forever. An additional benefit of temporal logic is that it can be used to describe a system at various levels of abstraction; the proof does not rely on specific implementation details.

When proving any property regardless of type, the property must be expressed in the form of one or more assertions that are proven individually. The mathematical proofs themselves typically use one of the following methods:

- **Direct Proof** – Proves an assertion by directly deriving the conclusion from the premise, such that the statement $p \implies q$ is true.

- **Proof by Contraposition** – Proves an assertion by proving that the negation of the conclusion implies the negation of the premise (i.e., the contrapositive) such that the statement $\neg q \implies \neg p$ is true.
- **Proof by Contradiction** – Proves an assertion by assuming that the premise does not imply the conclusion (i.e., that it is possible for the premise to be true and the conclusion to be false) and proving that the assumption leads to a contradiction, such that the statement $\neg(p \implies q)$ is false.
- **Proof by Induction** – Proves an assertion by demonstrating that it holds for the base case, asserting an induction rule for instances of size n (weak induction) or instances of up to size n (strong induction), and using one of the above methods to demonstrate that the induction rule holds for an instance of size $n + 1$.

As UVSLP is a protocol that is specified as a distributed concurrent system, its properties would be most appropriately formalized in terms of temporal logic, rather than classical logic. These formalized properties must then be decomposed as mathematical assertions to be proven correct. The proofs of these assertions will use these common methods.

2.4 Chapter Summary

In summary, this chapter discussed multivehicle autonomous systems and the communication challenges that can affect the reliability and resiliency of logging data within these systems. It also discussed how UVSLP, a blockchain-based distributed logging protocol, attempts to address these challenges and provides a high-level overview of the protocol's processes. Finally, this chapter discussed the challenges of proving correctness for concurrent systems and provided a rationale for the use of temporal logic over classical logic in this thesis. Chapter 3 formalizes the event handlers and properties in terms of temporal logic, then provides formal proofs using the above common proof methods to prove that each property is satisfied by the protocol.

CHAPTER 3: Formal Proofs

This chapter provides in-depth descriptions of the UVSLP event handlers as specified in [2] and [1]. The event handlers are formally described in the context of their behaviors and the states of various data structures maintained by each agent. In some cases, multiple event handlers are abstracted to a singular definition to represent the overall process that they collectively represent. Furthermore, supplementary definitions are provided with regards to assumptions of the protocol. The chapter culminates with formal proof of the satisfaction of each of the six fundamental properties that characterize the UVSLP protocol.

Of importance, the definitions and proofs contained in this chapter incorporate an expanded set of logic operators sourced from the Temporal Logic of Actions, as described by Lamport [21]. These operators are depicted in Table 3.1, where A is a predicate, S is a set or sequence, and F and G are actions. Additionally, the common sets and sequences assumed to exist are described in Table 3.2.

Table 3.1. Extended logic operators used in proofs. Adapted from: [21]

Operator	Description
$A \triangleq exp$	Defines predicate A to be equal to exp
$\{x \in S : p\}$	Set of elements x in set S that satisfy p
$Cardinality(S)$	Cardinality, or number of elements, of set S
$\langle e_1, \ldots, e_n \rangle$	Ordered sequence with n elements
$S[i]$	Component i of sequence S
$S[i..j]$	Components i through j of sequence S, even when $i = j$
$[i \in S \mapsto e]$	Sequence such that each component $i \in S$ is e
$Length(S)$	Length of sequence S
$Append(S, e)$	Appends element e to sequence S
$Remove(S, e)$	Removes element e from sequence S
$Max(S)$	The component of sequence S with the highest associated value
p'	Next state of variable p
$F \circ G$	Composition[1] of actions F and G, such that F then G
$\Box F$	F is always true
$\Diamond F$	F is eventually true

[1] The centered dot ($\cdot$) used by Lamport for the composition of actions is replaced with the circle ($\circ$) for readability and to remain consistent with the common notation used for the composition of functions.

Table 3.2. Assumed sets and sequences of UVSLP.

Structure	Description
$N_i = \{a_1, \ldots, a_n\}$	Set of all neighboring agents to agent i, not including i
$D_i = \langle b_1, \ldots, b_n \rangle$	Double-ended queue (deque) of agent i
$C_i = \langle b_1, \ldots, b_n \rangle$	Blockchain of agent i
$R_i = \langle b_1, \ldots, b_n \rangle$	Reconcile stack of agent i
$B_i = (D_i \cup C_i \cup R_i)$	Union of block sequences of agent i
$A = \{a_1, \ldots, a_n\}$	Set of all agents in the system
$Chains = \{C_i : i \in Agents\}$	Set of all blockchains of all agents in the system
$Blocks = \{b_1, \ldots, b_\infty\}$	Set of all possible blocks

3.1 Common Definitions

Prior to delving into the explicit definitions of the UVSLP event handlers, it is worthwhile to establish the axiomatic definitions of the common functions and predicates that are used to describe them. Each function is defined in terms of predicate calculus to describe the outcomes (i.e., postconditions) of its application.

The $AddToDeque(i, b)$ and $AddToChain(i, b)$ functions, Equations 3.1 and 3.2, are used to add blocks that are received from other agents directly to an agent's locally maintained double-ended queue (deque) or blockchain. $AddToDeque$ leads to the addition of a block, b, to the back of agent i's deque, D_i, and $AddToChain$ leads to the addition of the block to the end (i.e., top) of its blockchain, C_i. These functions only affect the data structure to which the received block is added, so all other local data structures are unchanged.

$$AddToDeque(i, b) \triangleq Append(D_i, b) \wedge (C_i' = C_i) \wedge (R_i' = R_i) \tag{3.1}$$

$$AddToChain(i, b) \triangleq Append(C_i, b) \wedge (D_i' = D_i) \wedge (R_i' = R_i) \tag{3.2}$$

Equations 3.3 through 3.5 define functions that are used to remove a block b from one of agent i's locally maintained data structures and immediately add it to another. $DequeToChain(i, b)$ and $DequeToStack(i, b)$ are used respectively to move block b from the front of the deque, D_i, to the end (i.e., top) of the blockchain, C_i, or the top of the reconcile stack, R_i. Similarly, $ChainToStack(i, b)$ is used to move block b from the end of the local blockchain to the top of the reconcile stack, and $StackToChain(i, b)$ is used to move block b from the top of the reconcile stack to the end of the blockchain. As the equations indicate, none of these operations has any effect on the uninvolved local data structure.

$$DequeToChain(i, b) \triangleq Remove(D_i, b) \wedge Append(C_i, b) \wedge (R_i' = R_i) \tag{3.3}$$

$$DequeToStack(i, b) \triangleq Remove(D_i, b) \wedge Append(R_i, b) \wedge (C_i' = C_i) \tag{3.4}$$

$$ChainToStack(i, b) \triangleq Remove(C_i, b) \land Append(R_i, b) \land (D'_i = D_i) \tag{3.5}$$

$$StackToChain(i, b) \triangleq Remove(R_i, b) \land Append(C_i, b) \land (D'_i = D_i) \tag{3.6}$$

The Boolean $SubChain(C_1, C_2)$ of Equation 3.7 is specified for proof purposes. It returns true if the blockchain C_1 contains the blockchain C_2 as a subchain. Containment implies that the two blockchains are identical from their beginnings to the end of the contained blockchain (i.e., C_2). As such, differences between the two chains must consist solely of blocks that were added to chain C_1 after the last block of C_2 was added to both chains.

$$SubChain(C_1, C_2) \triangleq \exists n : C_1[1..n] = C_2 \tag{3.7}$$

The $SameChain(i)$, $NotSameChain(i)$, $Contains(i)$, and $NotContains(i)$ functions, Equations 3.8 through 3.11, provide information about the contents of the blockchains maintained by the neighbors of agent i relative to its own locally maintained blockchain. $SameChain(i)$ and $NotSameChain(i)$ respectively return the set of neighboring agents with local blockchains that are identical to or that differ from agent i's local blockchain. $Contains(i)$ and $NotContains(i)$ respectively return the set of neighboring agents whose local blockchains contain or do not contain agent i's entire blockchain as a subchain. Since the functions amount to system-wide queries, none of these functions affects any agent i data structures.

$$SameChain(i) \triangleq \{j \in N_i : C_i = C_j\} \tag{3.8}$$

$$NotSameChain(i) \triangleq \{j \in N_i : C_i \neq C_j\} \tag{3.9}$$

$$Contains(i) \triangleq \{j \in N_i : SubChain(C_j, C_i)\} \tag{3.10}$$

$$NotContains(i) \triangleq \{j \in N_i : \neg SubChain(C_j, C_i)\} \tag{3.11}$$

Boolean predicates allow for the assertion of conditions that may be demonstrated to be true. The $MajoritySameChain(i)$ predicate, if true, indicates a condition in which the number of neighboring agents to agent i that have a blockchain matching that of C_i is greater than or equal to the number of neighboring agents that do not. That is, it is true if agent i's blockchain is identical to a local majority (possibly a global plurality) of neighboring agents' blockchains. Similarly, $MajorityContains(i)$ indicates a condition in which agent i's blockchain is a subchain of a local majority. These Boolean predicates are specified in Equations 3.12 and 3.13.

$$MajoritySameChain(i) \triangleq \\ Cardinality(SameChain(i)) \geq Cardinality(NotSameChain(i)) \tag{3.12}$$

$$MajorityContains(i) \triangleq \\ Cardinality(Contains(i)) \geq Cardinality(NotContains(i)) \tag{3.13}$$

The UVSLP voting process for extending the local blockchain after a subchain has been identified is captured by the $VoteNextBlock(i, votes)$ function (Equation 3.14. For each agent j in the set of agents that contain agent i's blockchain, block b_j is the block in agent j's blockchain immediately succeeding the contained subchain. Accounting for every agent, j, results in a $votes$ object with the overall vote tallies for all candidate blocks.

$$VoteNextBlock(i, votes) \triangleq$$

$$\forall j \in (Contains(i)) : \begin{pmatrix} C_i \neq C_j & \wedge \\ n = Length(C_i) & \wedge \\ b_j = C_j[n+1] & \wedge \\ votes'[b_j] = votes[b_j] + 1 & \end{pmatrix} \quad (3.14)$$

3.2 Event Handlers

In preparation for proving the properties of UVSLP, this section examines and formally defines the protocol's event handlers. While the protocol encompasses twenty distinct event handlers, a concise approach was adopted in which states of the various data structures maintained by agents were concentrated and the associated event handlers were considered as components of higher-level processes. In this section, a description of each event handler is presented. Each description includes an explanation of underlying assumptions and a formal definition suitable for use in Section 3.4 (Proofs).

3.2.1 Block Generation and Commit Process

The *Build Data Block* event handler is responsible for generating blocks as event data is generated by the agent's logging system. It is triggered internally by the generated event data and adds this data to the current block being built. If the block is ready to be committed, it appends the block to the agent's deque and triggers the *Process Next Deque Block* event handler. If the block is not ready, the process is repeated until enough event data is generated to complete the block. As the rest of the handlers cannot act on any particular block until it is ready to be committed, a formal definition of the full *Build Data Block* is not required. Rather, only those invocations that result in the proposal of a block for addition to the blockchain need be considered.

The formal definition of Equation 3.15 assumes that the agent of any block is the agent that executed the *Build Data Block* handler that generated the block. It is assumed that all blocks are unique when generated, meaning that no block can already be maintained by the agent

when it is generated. Upon completion of the event handler, the newly generated block is added to the agent's deque and the *Process Next Deque Block* handler is invoked.

$$BuildDataBlock(i,b) \triangleq \begin{pmatrix} Agent(b) = i & \wedge \\ i \in A & \wedge \\ b \notin B_i & \wedge \\ AddToDeque(i,b) & \circ \\ \quad ProcessNextDequeBlock(i,b) & \end{pmatrix} \tag{3.15}$$

When the deque contains elements and there is no ongoing reconciliation process, the *Process Next Deque Block* handler designates the frontmost block in the deque as the candidate block. Subsequently, it initiates a voting timer and broadcasts a `Block Add Request` to all available agents. This request includes the hash of the block under consideration and the chain hash of the proposing agent's present blockchain. Expiration of the voting timer will eventually activate the *Vote Timer Expiration* handler.

Upon receiving a `Block Add Request`, the *Receive Request Add Block* handler is activated. If the chain hash of the request aligns with the hash of its own blockchain, the handler broadcasts a `Block Add Response` indicating approval of the proposed block. Conversely, if the chain hashes do not match, the vote expressed in the `Block Add Response` is one of disapproval. Upon receiving a `Block Add Response`, the *Receive Vote Add Block* handler is triggered. While the voting timer is active, receipt of a `Block Add Response` for the candidate block will cause the handler to increment the approval or disapproval vote as appropriate.

In regard to proving the properties of UVSLP, the formalization of the *Receive Request Add Block* and *Receive Vote Add Block* handlers is unnecessary. The presence of an active voting timer serves the purpose of allowing available agents sufficient time to receive and vote on a proposed block. Consequently, the functionality of these handlers can be expressed within the definition of the *Process Next Deque Block* handler by setting the approval and disapproval vote counts to the cardinalities of the set of all neighboring agents to the proposing agent that possess a matching chain and the set of all neighboring agents lacking a matching chain. Given this approach, the *Vote Timer Expiration* handler can be considered

to be directly activated by the *Process Next Deque Block* handler rather than by the actual expiration of a vote timer. Furthermore, the *Process Next Deque Block* handler can be considered to be automatically activated anytime the deque is not empty rather than being triggered by specific event handlers.

$$ProcessNextDequeBlock(i, b) \triangleq \begin{pmatrix} \neg reconcileInProgress_i & \wedge \\ D_i \neq \emptyset & \wedge \\ b = D_i[1] & \wedge \\ VoteTimerExpiration(i, b, votes) & \end{pmatrix} \tag{3.16}$$

The *Vote Timer Expiration* handler is internally triggered upon the expiration of the vote timer and is initiated by the *Process Next Deque Block* handler. In the absence of an ongoing reconciliation process, this handler takes one of two possible actions:

1. In the event that the majority of votes received were in favor of approving the proposed block, the handler proceeds to commit the block to the local blockchain. Subsequently, it broadcasts a `Commit Block` message, containing the relevant block along with its hash and the chain hash representing the agent's current local blockchain.
2. if the majority of votes received were not in favor of approving the proposed block, the handler activates the *Reconcile Begin* handler.

In formalizing the *Vote Timer Expiration* handler for the purpose of proving the properties of UVSLP, it suffices to solely consider whether the majority of available agents approved or disapproved of the proposed block. The handling of these two possibilities is segmented into separate definitions in the predicate definition of Equation 3.17 to enhance readability.

$$VoteTimerExpiration(i, b, votes) \triangleq \begin{pmatrix} BlockApproved(i, b, votes) & \vee \\ BlockRejected(i, b, votes) & \end{pmatrix} \tag{3.17}$$

When a majority of available agents express approval for the proposed block, the handler broadcasts a `Commit Block` message before moving the block from the proposing agent's

deque to its local blockchain. Conversely, when a majority of available agents do not approve of the proposed block, the handler moves the block from the deque to the reconcile stack and triggers the *Reconcile Begin* handler. These processes are captured by Equations 3.18 and 3.19 respectively.

$$BlockApproved(i, b, votes) \triangleq \begin{pmatrix} MajoritySameChain(i) & \wedge \\ SendCommitBlock(i, b) & \circ \\ DequeToChain(i, b) & \end{pmatrix} \tag{3.18}$$

$$BlockRejected(i, b, votes) \triangleq \begin{pmatrix} \neg MajoritySameChain(i) & \wedge \\ DequeToStack(i, b) \circ ReconcileBegin(i) & \end{pmatrix} \tag{3.19}$$

Upon receiving a `Commit Block` message, an agent activates the *Receive Commit Block* handler. If there is no ongoing reconciliation process and the chain hash contained within the message corresponds to the agent's own chain hash, the agent proceeds to commit the block included in the message to its local blockchain. On the other hand, if no match is found between the chain hash in the message and the agent's own chain hash, the message is disregarded.

Prior to formalizing the definition of the *Receive Commit Block* handler, it is worthwhile to formalize the concept of the `Commit Block` message being broadcast by the proposing agent. Given that the message will only be acknowledged and processed by available agents with matching chain hashes, it is reasonable to assume that the `Commit Block` message only triggers the *Receive Commit Block* handler of agents possessing an identical blockchain to the proposing agent as specified in Equation 3.20.

$$SendCommitBlock(i, b) \triangleq \\ \forall j \in SameChain(i) : ReceiveCommitBlock(j, i, b) \tag{3.20}$$

The formal definition of the *Receive Commit Block* handler (Equation 3.21) commits the received block to the local blockchain if no reconciliation process is underway. Despite the semantics described for the `Commit Block` message that already imply a matching blockchain between the proposing agent and the receiving agent, this detail is explicitly included within the definition of the *Receive Commit Block* handler, as it serves to support the proofs of UVSLP properties presented Section 3.4.

$$ReceiveCommitBlock(i, j, b) \triangleq \begin{pmatrix} \neg reconcileInProgress_i & \wedge \\ C_i = C_j & \wedge \\ AddToChain(i, b) & \end{pmatrix} \tag{3.21}$$

3.2.2 Blockchain Reconcile Process

The reconcile process is initiated when the *Reconcile Begin* handler is triggered by the *Vote Timer Expiration* handler. In the absence of an ongoing reconciliation process, this event handler initiates the *Reconcile Phase 1* handler. The formal definition of the *Reconcile Begin* handler of Equation 3.22 succinctly mirrors this functionality.

$$ReconcileBegin(i) \triangleq \begin{pmatrix} \neg reconcileInProgress_i & \wedge \\ reconcileInProgress'_i & \wedge \\ ReconcilePhase1(i) & \end{pmatrix} \tag{3.22}$$

The *Reconcile Phase 1* handler can be invoked by either the *Reconcile Begin* or *Reconcile Phase 1 Timer Expiration* handlers. Upon activation, it proceeds with the following steps: clear the reconcile votes, initiate a Phase 1 timer, and broadcast a `Chain Contain Query` containing the chain hash of the local blockchain. The *Receive Request Chain Contain* handler, in turn, is triggered upon receipt of a `Chain Contain Query`. If the chain hash included in the query is present anywhere in the local blockchain, the agent sends an affirmative `Chain Contain Response`. Otherwise, the agent sends a disapproval response. Upon receiving a `Chain Contain Response`, the agent triggers the *Receive Phase 1 Response* handler, provided that the Phase 1 timer is still active and the chain hash in the response corresponds to the chain hash of its local blockchain.

Upon expiration of the Phase 1 timer, the *Reconcile Phase 1 Timer Expiration* handler is activated. It can undertake one of three possible actions based on the prevailing conditions:

1. If both the approval and disapproval vote counts are zero, indicating a lack of available agents, the handler triggers the *Reconcile Finalize* handler.
2. In the event that the majority of votes express approval, signifying a consensus among available agents that a mutual blockchain has been identified, the *Reconcile Phase 2* handler is triggered.
3. Finally, if the majority of votes express disapproval, implying the absence of a mutual blockchain, the *Reconcile Phase 1* handler is retriggered.

Similar to the *Process Next Deque Block* handler and its voting process, the Phase 1 timer simply ensures that available agents have ample time to receive and cast their votes. As a result, the entire Phase 1 process can be succinctly captured in the single formal definition of Equation 3.23. The right side of the equation is segmented into the three potential courses of action based on the outcome of the voting process.

$$ReconcilePhase1(i) \triangleq \begin{pmatrix} NoNeighborAgents(i) & \vee \\ MutualChainFound(i) & \vee \\ MutualChainNotFound(i) & \end{pmatrix} \tag{3.23}$$

Both the Boolean conditions and the response actions are codified in the *NoNeighborAgents*, *MutualChainFound*, and *MutualChainNotFound* predicates of Equations 3.24 through 3.26. If there are no neighboring agents, the *Reconcile Finalize* handler is triggered. If a majority of neighboring agents possess the blockchain of the agent as a subchain, the *Reconcile Phase 2* process is initiated. If the majority of neighboring agents do not possess the blockchain of the agent as a subchain, the last block in the blockchain is moved to the reconcile stack, and the *Phase 1* handler is reactivated.

$$NoNeighborAgents(i) \triangleq (N_i = \emptyset) \wedge ReconcileFinalize(i) \tag{3.24}$$

$$MutualChainFound(i) \triangleq \begin{pmatrix} N_i \neq \emptyset & \wedge \\ MajorityContains(i) & \wedge \\ ReconcilePhase2Begin(i) & \end{pmatrix} \tag{3.25}$$

$$MutualChainNotFound(i, votes) \triangleq \begin{pmatrix} N_i \neq \emptyset & \wedge \\ \neg MajorityContains(i) & \wedge \\ b = C_i[Length(C_i)] & \wedge \\ ChainToStack(i, b) & \circ \\ \quad ReconcilePhase1(i) & \end{pmatrix} \tag{3.26}$$

The *Reconcile Phase 2* handler is initiated by the *Reconcile Phase 1 Timer Expiration* handler when a mutual chain is identified among a majority of available agents. Upon activation, the handler takes the following actions: clear the votes for the next block in the sequence, commence a Phase 2 timer, and broadcast a `Chain Hash Successor Query` with the hash of the current local blockchain. Upon receiving a `Chain Hash Success Query`, an agent activates the *Receive Request Next in Sequence* handler. If the chain hash of the query is internally present in the local blockchain (i.e., not the end), the agent broadcasts a `Next in Sequence Response` message with the chain hash from the query, the hash of the subsequent block in the sequence, and the associated timestamp. The reconciling agent receives the `Next In Sequence Response` messages and tallies votes for candidate blocks through the *Receive Response Next in Sequence* handler.

Upon expiration of the Phase 2 timer, the *Reconcile Phase 2 Timer Expiration* handler is activated and performs one of the following actions based on the vote counts:

1. If no votes have been received, indicating that no available agent has a successor block to the current blockchain, the handler triggers the *Reconcile Finalize* handler.
2. In the presence of a block with a larger vote count than others, the handler designates the block with the majority of responses as the response block and activates the *Reconcile Phase 3* handler.
3. In the event of a tie in the vote counts, the handler selects the tied block with the oldest timestamp as the majority response block and triggers the *Reconcile Phase 3* handler.

As with other voting processes, the Phase 2 timer simply allows sufficient time for available agents to receive the query and cast their votes on the subsequent block in the sequence. As a result, the entire Phase 2 process can be represented by the two formal definitions of Equations 3.27 and Equations 3.28.

$$ReconcilePhase2Begin(i) \triangleq \begin{pmatrix} votes = [b \in Block \mapsto 0] & \wedge \\ VoteNextBlock(i, votes) \circ ReconcilePhase2End(i, votes) & \end{pmatrix} \quad (3.27)$$

$$ReconcilePhase2End(i, votes) \triangleq \begin{pmatrix} (votes[Max(votes)] = 0 \wedge ReconcileFinalize(i)) & \vee \\ (votes[Max(votes)] \neq 0 \wedge ReconcilePhase3(i, Max(votes))) & \end{pmatrix} \quad (3.28)$$

Equation 3.27 captures the start of the Phase 2 process. It initializes the votes for all potential blocks to zero and collects the votes for the next block in the sequence from all neighboring agents to the reconciling agent and possessing the current blockchain as a subchain. It then activates the conclusion of the Phase 2 process.

Equation 3.28 represents the conclusion of the Phase 2 process. If the block with the highest vote count has a count of zero (i.e., no votes were received), the *Reconcile Finalize* handler is triggered. If, on the other hand, the block with the highest vote count is not zero, the Phase 3 process is initiated. Since the tiebreaking heuristics have no effect on the proofs, the predicate is written such that $Max(votes)$ implicitly chooses the correct block in the event of a tie.

The *Reconcile Phase 3* handler is triggered by the *Reconcile Phase 2 Timer Expiration* handler when a successor block to the current blockchain has been identified. If the agent already has the successor block in its reconcile stack, it is removed from the stack and committed to the agent's local blockchain. Otherwise, the handler initiates a Phase 3 timer and broadcasts a `Request for Block` message containing the hash of the successor block.

An agent receiving a `Request for Block` message activates the *Receive Request Block for Hash* handler. If the agent has the requested block in its blockchain, the handler broadcasts a `Block Response` message containing the requested block. Upon receiving the first `Block Response` (assuming the Phase 3 timer is active), the reconciling agent activates the *Receive Response Block for Hash* handler to commit the block to its local blockchain, clear the Phase 3 timer, and retrigger the *Reconcile Phase 2* handler. If the Phase 3 timer ends without receiving a response, the *Reconcile Phase 3 Timer Expiration* handler is triggered to initiate the *Reconcile Finalize* handler.

Once again, the timer affords available agents sufficient time to receive and respond to messages and can be incorporated directly into the *Phase 3* predicate. The Phase 3 process is formally defined based on the availability of the successor block in Equation 3.29.

$$ReconcilePhase3(i,b) \triangleq \begin{pmatrix} BlockInStack(i,b) & \wedge \\ BlockInNetwork(i,b) & \wedge \\ BlockUnavailable(i,b) & \end{pmatrix} \tag{3.29}$$

As with the *Phase 1* and *Phase 2* definitions, the right side of Equation 3.29 is broken down by possible outcomes that capture both the Boolean condition and the response. If the successor block is located in the reconcile stack, it is moved to the blockchain and the Phase 2 process is reactivated. If a neighboring agent possesses the successor block in its blockchain and is able to pass it to the reconciling agent, the block is committed to the local blockchain and the Phase 2 process is reactivated. Finally, if the successor block is not present in the reconcile stack and cannot be obtained from a neighboring agent, the *Reconcile Finalize* handler is triggered. This functionality is depicted in Equations 3.30, 3.31, and 3.32.

$$BlockInStack(i,b) \triangleq \begin{pmatrix} b \in R_i & \wedge \\ StackToChain(i,b) & \circ \\ \quad ReconcilePhase2Begin(i) & \end{pmatrix} \tag{3.30}$$

$$BlockInNetwork(i,b) \triangleq \begin{pmatrix} b \notin R_i & \wedge \\ \exists j \in N_i : b \in C_j & \wedge \\ AddToChain(i,b) & \circ \\ \quad ReconcilePhase2Begin(i) & \end{pmatrix} \tag{3.31}$$

$$BlockUnavailable(i,b) \triangleq \begin{pmatrix} b \notin R_i & \wedge \\ \neg(\exists j \in N_i : b \in C_j) & \wedge \\ ReconcileFinalize(i) & \end{pmatrix} \tag{3.32}$$

The *Reconcile Finalize* handler is activated by any of the following event handlers: *Reconcile Phase 1 Timer Expiration*, *Reconcile Phase 2 Timer Expiration*, or *Reconcile Phase 3 Timer Expiration* when the portion of the reconcile process dependent on external agents is complete or cannot proceed further. While the reconcile stack remains nonempty, the handler performs the following actions: it removes a block from the top of the stack, broadcasts a `Commit Block` message containing the block, and commits the block to the agent's local blockchain.

The formal definition of the *Reconcile Finalize* handler can be partitioned into two distinct actions based on the status of the reconcile stack:

$$ReconcileFinalize(i) \triangleq StackNotEmpty(i) \vee StackEmpty(i) \tag{3.33}$$

The right-side functionality of Equation 3.33 is specified by Equations 3.34 and 3.35. If the reconcile stack is not empty, the handler sends the top block in the reconcile stack to available agents before moving the block from the reconcile stack to the agent's local blockchain. Subsequently, the *Reconcile Finalize* handler is reactivated. The reconcile process is complete when the reconcile stack is empty.

$$StackNotEmpty(i) \triangleq \begin{pmatrix} R_i \neq \emptyset & \wedge \\ b = R_i[Length(R_i)] & \wedge \\ (SendCommitBlock(i, b) & \circ \\ \quad StackToChain(i, b) & \circ \\ \quad ReconcileFinalize(i)) & \end{pmatrix} \tag{3.34}$$

$$StackEmpty(i) \triangleq \begin{pmatrix} R_i = \emptyset & \wedge \\ reconcileInProgress_i & \wedge \\ \neg reconcileInProgress'_i & \end{pmatrix} \tag{3.35}$$

3.3 Supplemental Definitions

In addition to the formal definitions of event handlers, a few supplemental definitions are useful when proving the six properties of the protocol. In particular, these definitions formalize the notions of the idle state of all event handlers and full connection of the system, and asserts a sequential (i.e., iterative) reconcile mechanism that can be invoked when all agents are fully connected (possibly at the end of a mission).

For proving the *Idle Stop* property, it is necessary to establish a definition of all event handlers associated with an agent being idle. For the purpose of the ensuing proof, all event handlers can be deemed idle solely under the condition that the deque is empty and no ongoing reconcile process is in progress. The presence of an empty deque indicates that the *Build Data Block* event handler has not appended a new block to the deque, thereby preventing the commencement of the commit process by the *Process Next Deque Block* event handler. Additionally, the reconcile process is deemed to be underway from initiation of the *Reconcile Begin* event handler until the *Reconcile Finalize* event handler concludes.

$$AllIdle(i) \triangleq (D_i = \emptyset) \wedge \neg reconcileInProgress_i \tag{3.36}$$

The next definition defines a state in which all agents in the system are connected. This holds particular significance in proving the *Uniform Chain* and *Block Propagation* properties. With the premise that each agent has a set encompassing the agents neighboring it, the following

characterizes a state of the system being fully connected as, for every agent, this set of neighboring agents is equivalent to the set of all agents.

$$FullyConnected(A) \triangleq \forall i \in A : N_i = A \quad (3.37)$$

The final supplementary definition concerns a sequential reconcile mechanism. It is assumed that a UVSLP implementation will possess a mechanism to systematically initiate a reconciliation process in each agent, one after the other, at the conclusion of a given operation or mission. Consequently, the predicate formalizing this mechanism is defined as Equation 3.38 to describe a situation in which there exists exactly one agent within the set of agents that is currently engaged in an ongoing reconciliation process.

$$SequentialReconcile(A) \triangleq \Box(\exists_{=1} i \in A : reconcileInProgress_i) \quad (3.38)$$

3.4 Proofs

With formal definitions established, the ensuing section involves presenting proofs for each of the six properties of the UVSLP. The properties are precisely formalized, drawing upon the formal definitions introduced in Section 3.2 to substantiate their validity and correctness.

3.4.1 Proof of the Block Creation Property

The *Block Creation* property is defined as the implicit condition where each block maintained by an agent originated from an agent belonging to the set of agents within the system. To directly prove this property, the proof must demonstrate that all means by which a block enters the local system imply that the block was generated by a valid agent.

Theorem 3.1. *Block Creation: No block will exist within the system that was not proposed by a participating agent.*

$$\forall i, b : (i \in A) \wedge (b \in B_i) \Rightarrow \Box\, Agent(b) \in A \quad (3.39)$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a set of blocks, B_i.

1. $(b \notin B_i) \wedge (b \in B'_i) \Rightarrow AddToDeque(i, b) \vee AddToChain(i, b)$
2. $AddToDeque(i, b) \Rightarrow BuildDataBlock(i, b)$
3. $AddToChain(i, b) \Rightarrow ReceiveCommitBlock(i, j, b) \vee BlockInNetwork(i, b)$
4. Claim: $BuildDataBlock(i, b) \Rightarrow Agent(b) \in A$
   (a) $BuildDataBlock(i, b) \Rightarrow (Agent(b) = i) \wedge (i \in A)$
   (b) $(Agent(b) = i) \wedge (i \in A) \Rightarrow Agent(b) \in A$
5. Claim: $ReceiveCommitBlock(i, j, b) \Rightarrow Agent(b) \in A$
   (a) $ReceiveCommitBlock(i, j, b) \Rightarrow SendCommitBlock(j, b) \wedge (j \neq i)$
   (b) $SendCommitBlock(j, b) \Rightarrow VoteTimerExpiration(j, b)$
   (c) $VoteTimerExpiration(j, b) \Rightarrow ProcessNextDequeBlock(j, b)$
   (d) $ProcessNextDequeBlock(j, b) \Rightarrow b \in D_j$
   (e) $b \in D'_j \Rightarrow AddToDeque(j, b)$
   (f) $AddToDeque(j, b) \Rightarrow BuildDataBlock(j, b)$
   (g) $BuildDataBlock(j, b) \Rightarrow (Agent(b) = j) \wedge (j \in A)$
   (h) $(Agent(b) = j) \wedge (j \in A) \Rightarrow Agent(b) \in A$
6. Claim: $BlockInNetwork(i, b) \Rightarrow Agent(b) \in A$
   (a) $BlockInNetwork(i, b) \Rightarrow \exists j \in N_i : b \in C_j$
   (b) $b \in C_j \Rightarrow b \in B_j$
   (c) $b \in B'_j \Rightarrow \begin{pmatrix} BuildDataBlock(j, b) & \wedge \\ ReceiveCommitBlock(j, k, b) & \wedge \\ BlockInNetwork(j, b) & \end{pmatrix}$

■

Step 1 establishes that if a block is present in the subsequent state of the set of all blocks maintained by an agent but not in the current state, it implies that the block was either appended to the agent's deque or to its blockchain.

Step 2 asserts that a block being added to the agent's deque implies that the *Build Data Block* handler was triggered (Equation 3.15). Similarly, Step 3 states that a block being added to the agent's blockchain without being transferred from the deque or reconcile stack implies that the block was received from another agent. That is, it was either received through receipt of a `Commit Block` message (Equation 3.21) or by receipt of a `Block Response` message during the *Reconcile Phase 3* process (Equation 3.31).

Step 4 demonstrates that execution of the *Build Data Block* handler implies that the agent of a block generated by the handler is a member of the set of agents. This is supported by the formal definition of the handler in Equation 3.15, which establishes that the agent of the generated block is the one that executed the handler and that agent belongs to the set of agents.

Step 5 states that receipt of a `Commit Block` message implies that the agent of the message block is a member of the set of agents. When an agent receives a `Commit Block` message, Equation 3.20 implies that another agent sent the message. Consequently, the other agent's sending of the `Commit Block` message indicates that the agent's *Vote Timer Expiration* handler (Equation 3.17) was triggered, which, in turn, implies that its *Process Next Deque Block* handler (Equation 3.16) was triggered. As demonstrated in Step 2, the *Process Next Deque Block* handler is activated when a block is added to the deque, indicating that the block was generated by the agent's *Build Data Block* handler (Equation 3.15). Thus, it can be concluded that the *Build Data Block* handler implies that the agent of the block is a part of the set of agents.

Step 6 posits that a block received by another agent during Phase 3 of the reconcile process recursively implies that the agent of that block is a member of the set of agents. Receipt of a block from the network during the third phase of the reconcile process (Equation 3.31) signifies the existence of another agent with the block present that agent's blockchain. Consequently, the block's existence in the other agent's blockchain implies its inclusion in the set of blocks maintained by that agent. This, in turn, implies that the agent either generated the block through its *Build Data Block* handler or received the block from some other agent.

3.4.2 Proof of the No Block Duplication Property

The *No Block Duplication* property is the implicit condition for each block in an agent's set of maintained blocks to exclusively exist in one of its three structures: the local blockchain, the deque, or the reconcile stack. To directly prove this property, the proof must demonstrate that for all methods by which a block is introduced into a system that it is only added to one of its three structures. Additionally, the proof must demonstrate that for all methods by which a block is moved from one structure to another that the block exclusively existed in the first structure prior to the move and exclusively exists in the second structure following the move.

Theorem 3.2. *No Block Duplication: No more than one copy of a particular block will be maintained by an agent at any time.*

$$\forall i, b : (i \in A) \wedge (b \in B_i) \Rightarrow \Box \begin{pmatrix} ((b \in D_i) \wedge (b \notin (C_i \cup R_i))) & \vee \\ ((b \in C_i) \wedge (b \notin (D_i \cup R_i))) & \vee \\ ((b \in R_i) \wedge (b \notin (D_i \cup C_i))) & \end{pmatrix} \tag{3.40}$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a deque, D_i, local blockchain, C_i, and reconcile stack, R_i.

1. $b \in B_i \Rightarrow b \in (D_i \cup C_i \cup R_i)$
2. $b \in D_i \Rightarrow AddToDeque(i, b)$

3. $b \in C_i \Rightarrow \begin{pmatrix} AddToChain(i,b) & \vee \\ DequeToChain(i,b) & \vee \\ StackToChain(i,b) & \end{pmatrix}$

4. $b \in R_i \Rightarrow DequeToStack(i,b) \vee ChainToStack(i,b)$

5. Claim: $AddToDeque(i,b) \Rightarrow (b \in D_i') \wedge (b \notin (C_i' \cup R_i'))$

   (a) $AddToDeque(i,b) \Rightarrow BuildDataBlock(i,b)$

   (b) $BuildDataBlock(i,b) \Rightarrow b \notin (D_i \cup C_i \cup R_i)$

   (c) $AddToDeque(i,b) \wedge ((b \notin (D_i \cup C_i \cup R_i)) \Rightarrow (b \in D_i') \wedge (b \notin (C_i' \cup R_i'))$

6. Claim: $AddToChain(i,b) \Rightarrow (b \in C_i') \wedge (b \notin (D_i' \cup R_i'))$

   (a) $AddToChain(i,b) \Rightarrow \begin{pmatrix} ReceiveCommitBlock(i,j,b) & \vee \\ BlockInNetwork(i,b) & \end{pmatrix}$

   (b) $ReceiveCommitBlock(i,j,b) \Rightarrow b \notin (D_i \cup C_i \cup R_i)$

   (c) $BlockInNetwork(i,b) \Rightarrow b \notin (D_i \cup C_i \cup R_i)$

   (d) $AddToChain(i,b) \wedge (b \notin (D_i \cup C_i \cup R_i)) \Rightarrow (b \in C_i') \wedge (b \notin (D_i' \cup R_i'))$

7. Claim: $DequeToChain(i,b) \Rightarrow (b \in C_i') \wedge (b \notin (D_i' \cup R_i'))$

   (a) $DequeToChain(i,b) \Rightarrow BlockApproved(i,b,votes)$

   (b) $BlockApproved(i,b,votes) \Rightarrow VoteTimerExpiration(i,b)$

   (c) $VoteTimerExpiration(i,b) \Rightarrow (b \in D_i) \wedge (b \notin (C_i \cup R_i))$

   (d) $\begin{pmatrix} DequeToChain(i,b) & \wedge \\ b \in D_i & \wedge \\ b \notin (C_i \cup R_i) & \end{pmatrix} \Rightarrow \begin{pmatrix} b \in C_i' & \wedge \\ b \notin (D_i' \cup R_i') & \end{pmatrix}$

8. Claim: $DequeToStack(i,b) \Rightarrow (b \in R_i') \wedge (b \notin (D_i' \cup C_i'))$

   (a) $DequeToChain(i,b) \Rightarrow BlockRejected(i,b,votes)$

(b) $BlockRejected(i, b, votes) \Rightarrow VoteTimerExpiration(i, b)$

(c) $VoteTimerExpiration(i, b) \Rightarrow (b \in D_i) \wedge (b \notin (C_i \cup R_i))$

(d) $\begin{pmatrix} DequeToStack(i, b) & \wedge \\ b \in D_i & \wedge \\ b \notin (C_i \cup R_i) & \end{pmatrix} \Rightarrow \begin{pmatrix} b \in R'_i & \wedge \\ b \notin (D'_i \cup C'_i) & \end{pmatrix}$

9. Claim: $ChainToStack(i, b) \Rightarrow (b \in R'_i) \wedge (b \notin (D'_i \cup C'_i))$

(a) $ChainToStack(i, b) \Rightarrow MutualChainNotFound(i, votes) \wedge (b \in C_i)$

(b) $MutualChainNotFound(i, votes) \Rightarrow ReconcilePhase1End(i)$

(c) $ReconcilePhase1End(i) \wedge (b \in C_i) \Rightarrow (b \in C_i) \wedge (b \notin (D_i \cup R_i))$

(d) $\begin{pmatrix} ChainToStack(i, b) & \wedge \\ b \in C_i & \wedge \\ b \notin (D_i \cup R_i) & \end{pmatrix} \Rightarrow \begin{pmatrix} b \in R'_i & \wedge \\ b \notin (D'_i \cup C'_i) & \end{pmatrix}$

10. Claim: $StackToChain(i, b) \Rightarrow (b \in C'_i) \wedge (b \notin (D'_i \cup R'_i))$

(a) $StackToChain(i, b) \Rightarrow b \in R_i \wedge \begin{pmatrix} BlockInStack(i, b) & \vee \\ StackNotEmpty(i) & \end{pmatrix}$

(b) $BlockInStack(i, b) \Rightarrow ReconcilePhase3(i, b)$

(c) $ReconcilePhase3(i, b) \wedge (b \in R_i) \Rightarrow (b \in R_i) \wedge (b \notin (D_i \cup C_i))$

(d) $StackNotEmpty(i) \Rightarrow ReconcileFinalize(i)$

(e) $ReconcileFinalize(i) \wedge (b \in R_i) \Rightarrow (b \in R_i) \wedge (b \notin (D_i \cup C_i))$

(f) $\begin{pmatrix} StackToChain(i, b) & \wedge \\ b \in R_i & \wedge \\ b \notin (D_i \cup C_i) & \end{pmatrix} \Rightarrow \begin{pmatrix} b \in C'_i & \wedge \\ b \notin (D'_i \cup R'_i) & \end{pmatrix}$

■

Step 1 of the Theorem 3.2 proof establishes that a block's existence in the agent's set of maintained blocks implies that it is present in at least one of the three structures.

Steps 2, 3, and 4 establish the methods by which blocks must be added to the blockchain, the deque, and the reconcile stack. Step 2 establishes that if a block exists in the agent's deque, it implies that the block was appended to the deque (Equation 3.1). Step 3 asserts that if a block exists in the agent's local blockchain, it implies that the block was either appended to the blockchain (Equation 3.2), transferred from the deque to the blockchain (Equation 3.3, or moved from the reconcile stack to the blockchain (Equation 3.6). Step 4 then demonstrates that if a block exists in the agent's reconcile stack, it implies that the block was either moved from the deque to the reconcile stack (Equation 3.4) or from the blockchain to the reconcile stack (Equation 3.5).

Subsequently, Step 5 proves that adding a block to the deque through Equation 3.1 implies that the block solely exists in the subsequent state of the deque and not in the next states of the blockchain or reconcile stack. When the agent adds a block to the deque, it does so through the *Build Data Block* handler (Equation 3.15). As a result, the block was not present in any of the agent's structures before being added to the deque. Introduction of a block that was not previously an element of any of the agent's data structures to the deque implies that the block exclusively exists in the next state of the deque and not in the subsequent states of the blockchain or reconcile stack.

Similarly, Step 6 proves that adding a block to the blockchain through Equation 3.2 implies that the block solely exists in the subsequent state of the blockchain and not in the next state of the deque or reconcile stack. When a block is added to the blockchain, it is either received through a `Commit Block` message (Equation 3.21) or during Phase 3 of the reconcile process (Equation 3.31). In both instances, it is implied that the block was not previously present in any of the agent's structures. The addition to the blockchain of a block that did not previously exist in any of the structures implies that the block exclusively exists in the next state of the blockchain and not in the subsequent state of the deque or reconcile stack.

Steps 7, 8, 9, and 10 further prove that the movement of blocks between the blockchain, the deque, and the reconcile stack (Equations 3.3 through 3.6) ensures their exclusive existence in the next states of the respective receiving structures. Each step addresses

specific scenarios and the associated implications when moving blocks between these structures during different phases of the reconcile process or through the execution of particular event handlers. In each case, it is demonstrated that the movement preserves the block's unique existence in the subsequent state of the structure to which it is moved, while excluding its presence in the subsequent states of the other structures.

3.4.3 No Block Loss

The *No Block Loss* property is defined as the implicit condition where, for all blocks whose agents are part of the set of agents, there exists at least one agent that includes the block in its set of maintained blocks. To directly prove this property, the proof must demonstrate that for all methods by which a block is introduced into the system that the block is always maintained in one of its three structures. That is, it must demonstrate that any block accepted into the system is added to one of the three data structures and that any block that is removed from one of the structures is immediately added to another.

Theorem 3.3. *No Block Loss: All blocks proposed by participating agents will be maintained by at least one agent.*

$$\forall b : Agent(b) \in A \Rightarrow \Box(\exists i \in A : b \in B_i) \tag{3.41}$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a deque, D_i, local blockchain, C_i, and reconcile stack, R_i

1. $Agent(b) \in A \Rightarrow \exists i \in A : BuildDataBlock(i, b)$
2. $BuildDataBlock(i, b) \Rightarrow b \in D_i$
3. $b \in D_i \Rightarrow b \in B_i$
4. $ReceiveCommitBlock(i, j, b) \vee BlockInNetwork(i, b) \Rightarrow b \in C_i$
5. $b \in C_i \Rightarrow b \in B_i$

6. $b \in B_i \Rightarrow \Box \begin{pmatrix} ((b \in D_i) \wedge (b \notin (C_i \cup R_i))) & \vee \\ ((b \in C_i) \wedge (b \notin (D_i \cup R_i))) & \vee \\ ((b \in R_i) \wedge (b \notin (D_i \cup C_i))) & \end{pmatrix}$

■

Step 1 establishes that if the agent of a block is in the set of agents, it implies that there exists an agent that triggered the *Build Data Block* handler through Equation 3.15, leading to the creation of the block. Step 2 further asserts that the *Build Data Block* handler implies that the block is added to the agent's deque. Step 3 builds on this to establish that the block being present in the deque implies its inclusion in the set of blocks maintained by that specific agent.

Step 4 establishes that any external block introduced into the agent's local system, either through receipt of a `Commit Block` message (Equation 3.21) or during Phase 3 of the reconcile process (Equation 3.31), implies that the block is added to the agent's blockchain. Step 5 builds on this to establish that the block being present in the blockchain also implies its inclusion in the set of blocks maintained by that specific agent, similar to the conclusion asserted in Step 3.

Step 6 relies on the proof of the *No Block Duplication* property to demonstrate that the block's presence in the set of blocks maintained by the agent signifies that the block always exists in one of the three structures: the blockchain, the deque, or the reconcile stack. In particular, steps 6 through 10 of the proof of Theorem 3.2 demonstrate that any time a block is removed from one local data structure, it is immediately added to another.

3.4.4 Proof of the Idle Stop Property

The *Idle Stop* property states that if an agent is in a state in which all event handlers are idle, the difference between the set of all blocks maintained by that agent and the agent's blockchain is the empty set. To prove this property, we adopt the contrapositive approach to demonstrate that a difference between the set of blocks maintained by an agent and the set of blocks in that agent's blockchain implies that all event handlers cannot be idle.

Theorem 3.4. *Idle Stop: If an agent's protocol event handlers are in an idle state, then all blocks maintained by that agent must be present in the local blockchain.*

$$\forall i \in A : AllIdle(i) \Rightarrow \Box(B_i - C_i = \emptyset) \tag{3.42}$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a deque, D_i, local blockchain, C_i, and reconcile stack, R_i.

1. $(AllIdle(i) \Rightarrow B_i - C_i = \emptyset) \equiv (B_i - C_i \neq \emptyset \Rightarrow \neg AllIdle(i))$
2. $B_i - C_i \neq \emptyset \Rightarrow \exists b \in B_i : (b \in D_i) \vee (b \in R_i)$
3. Claim: $b \in D_i \Rightarrow \neg AllIdle(i)$
   (a) $b \in D_i \Rightarrow D_i \neq \emptyset$
   (b) $D_i \neq \emptyset \Rightarrow \neg AllIdle(i)$
4. Claim: $b \in R_i \Rightarrow \neg AllIdle(i)$
   (a) $b \in R_i \Rightarrow DequeToStack(i, b) \vee ChainToStack(i, b)$
   (b) $DequeToStack(i, b) \Rightarrow BlockRejected(i, b, votes)$
   (c) $BlockRejected(i, b, votes) \Rightarrow DequeToStack(i, b) \circ ReconcileBegin(i)$
   (d) $ChainToStack(i, b) \Rightarrow MutualChainNotFound(i, votes)$
   (e) $MutualChainNotFound(i, votes) \Rightarrow ReconcilePhase1(i)$
   (f) $ReconcilePhase1(i) \Rightarrow ReconcileBegin(i)$
   (g) $ReconcileBegin(i) \Rightarrow reconcileInProgress'_i$
   (h) $reconcileInProgress_i \wedge \neg reconcileInProgress'_i \Rightarrow ReconcileFinalize(i)$
   (i) $ReconcileFinalize(i) \wedge \neg reconcileInProgress'_i \Rightarrow R_i = \emptyset$
   (j) $reconcileInProgress_i \Rightarrow \neg AllIdle(i)$

Step 1 of the proof establishes that the implication of all event handlers of an agent being idle and the difference between the set of blocks maintained by that agent and the agent's blockchain being an empty set is equivalent to its contrapositive. That is, the difference between the set of blocks maintained by that agent and the agent's blockchain not being the empty set implies that not all event handlers are idle.

Step 2 establishes that if the difference between the set of blocks maintained by that agent and the agent's blockchain is not an empty set, it implies the existence of a block in either the deque or reconcile stack. This assertion is supported by Theorem 3.3 (i.e., a block that has been accepted into the system must be present in the blockchain, the deque, or the reconcile stack).

Step 3 proves that a block's presence in the deque implies that not all event handlers are idle. The existence of a block in the deque leads to the conclusion that the deque is not empty, indicating that not all event handlers are in an idle state as the commit process for that block is not complete. This follows from the Section 3.2 description of Equation 3.16 in which the handler is automatically triggered by a nonempty deque.

Step 4 demonstrates that a block's presence in the reconcile stack also implies that not all event handlers are idle. A block's existence in the reconcile stack indicates that the block was either moved from the deque to the reconcile stack following rejection of a proposed commit (Equation 3.19) or from the blockchain to the reconcile stack as part of the reconcile process (Equation 3.26). When a block is moved from the deque to the reconcile stack, Equation 3.4 initiates the reconciliation process by triggering the *Reconcile Begin* handler (Equation 3.22) which in turn triggers the *Reconcile Phase 1* handler (Equation 3.23). Similarly, if a block is moved from the blockchain to the reconcile stack, it implies that the movement occurred during Phase 1 of the reconcile process per Equation 3.26 which also results in retriggering the *Phase 1* process. In either case, the movement results in initiation of *Phase 1*. Further, substeps (c) through (j) demonstrate that the reconcile process continues until the *Reconcile Finalize* handler empties the reconcile stack. Thus, it is implied that not all event handlers are in an idle state until that is the case.

3.4.5 Proof of the Uniform Chain Property

The *Uniform Chain* property states that a uniform blockchain will eventually emerge in a fully connected system where all agents have opportunities to reconcile with each other. This property is proven by first establishing two lemmas (1) that a reconciling agent will identify a common chain with a majority of the agents and (2) that the total number of unique blockchains in the set of all blockchains is reduced when an agent completes the reconcile process in a fully connected system. The proof is finalized by using these lemmas to demonstrate that the number of unique blockchains among the agents is equal to one following iterative reconciliation by all agents.

Lemma: Reconciliation Phase 1 Majority Subchain

Lemma 3.1 posits that within a completely connected system, wherein an agent is executing the *Reconcile Phase 1* process, the agent's blockchain will reduce to a subchain of a majority of agents' blockchains, at which point the agent will proceed to the *Reconcile Phase 2* handler. The lemma is proven directly by assuming all blockchains possess the same initial block as described in [1] and [2].

Lemma 3.1. *In a fully connected system, an agent in the Reconcile Phase 1 process will have a local blockchain that is a subchain of a majority of agents before proceeding to Reconcile Phase 2.*

$$\forall i \in A : \begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ FullyConnected(A) & \wedge \\ ReconcilePhase1(i) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} MajorityContains(i) & \wedge \\ ReconcilePhase2Begin(i) & \end{pmatrix} \tag{3.43}$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a local blockchain, C_i, and a reconcile stack, R_i.

1. $\forall i, j \in A : (C_i[1] = C_j[1]) \Rightarrow \exists n : (C_i[1..n] = C_j[1..n])$
2. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ FullyConnected(A) & \wedge \\ ReconcilePhase1(i) & \end{pmatrix} \Rightarrow N_i \neq \emptyset \wedge \begin{pmatrix} NotContains(i) \neq \emptyset & \vee \\ Contains(i) \neq \emptyset & \end{pmatrix}$

3. $\begin{pmatrix} NotContains(i) \neq \emptyset & \vee \\ Contains(i) \neq \emptyset & \end{pmatrix} \Rightarrow \begin{pmatrix} \neg MajorityContains(i) & \vee \\ MajorityContains(i) & \end{pmatrix}$

4. $\begin{pmatrix} ReconcilePhase1(i) & \wedge \\ N_i \neq \emptyset & \wedge \\ \neg MajorityContains(i) & \end{pmatrix} \Rightarrow \begin{pmatrix} b = C_i[Length(C_i)] & \wedge \\ ChainToStack(i, b) & \circ \\ \quad ReconcilePhase1(i) & \end{pmatrix}$

5. $\begin{pmatrix} ReconcilePhase1(i) & \wedge \\ N_i \neq \emptyset & \wedge \\ MajorityContains(i) & \end{pmatrix} \Rightarrow ReconcilePhase2Begin(i)$

■

Step 1 of the proof asserts that all agents in the same system must share a common subchain by virtue of having the same initial block.

Step 2 establishes that in a fully connected system with an unspecified number of agents, the set of neighboring agents for the reconciling agent cannot be empty. Further, it observes that there must exist another agent in either the set of neighboring agents that possess the reconciling agent's blockchain as a subchain or in the set of agents that do not. Following this, steps 3 asserts that either of these sets being nonempty implies that either the majority do not contain the subchain or that a majority do.

Step 4 establishes that if a majority of available agents' blockchains do not contain the reconciling agent's blockchain, a block will be moved from the blockchain to the reconcile stack and *Reconcile Phase 1* will be reinitiated (Equation 3.26). Step 5, on the other hand, asserts if a majority of available agents' blockchains do contain the reconciling agent's blockchain, *Reconcile Phase 2* will be initiated. Thus, in a completely connected system of an unspecified number of agents, the agent will not progress from *Reconcile Phase 1* to *Reconcile Phase 2* until a majority of agents contain its blockchain as a subchain.

Lemma: Reconciliation Phase 2 Matching Chains

Lemma 3.2 posits that, in a completely connected system, an agent executing the *Reconcile Phase 2* process will eventually attain a blockchain identical to that of another agent prior to

concluding the reconciliation process. This lemma is proven directly by building upon the conclusion of Lemma 3.1 and demonstrating that the reconciling agent's blockchain will be extended by adding blocks from another agent's blockchain.

Lemma 3.2. *In a fully connected system, an agent in Phase 2 of the reconcile process will eventually have a local blockchain that equals the local blockchain of some other agent before ending the reconcile process.*

$$\forall i \in A : \begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ FullyConnected(A) & \wedge \\ ReconcilePhase2Begin(i) & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} \exists j \in A : C_i' = C_j & \wedge \\ \neg reconcileInProgress_i' & \end{pmatrix} \quad (3.44)$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a local blockchain, C_i, and a reconcile stack, R_i.

1. $Phase2Begin(i) \Rightarrow MajorityContains(i)$
2. $FullyConnected(A) \wedge MajorityContains(i) \Rightarrow (Contains(i) \neq \emptyset)$
3. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Contains(i) \neq \emptyset & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix} \Rightarrow \exists j \in Contains(i) : C_i \neq C_j$
4. $\begin{pmatrix} ReconcilePhase2Begin(i) & \wedge \\ \exists j \in Contains(i) : C_i \neq C_j & \end{pmatrix} \Rightarrow \begin{pmatrix} n = Length(C_i) & \wedge \\ \exists j \in Contains(i) : b = C_j[n+1] & \wedge \\ Max(votes) = b & \wedge \\ votes[Max(votes)] \neq 0 & \wedge \\ ReconcilePhase2End(i, votes) & \end{pmatrix}$

5. $$\begin{pmatrix} n = Length(C_i) & \wedge \\ \exists j \in Contains(i) : b = C_j[n+1] & \wedge \\ Max(votes) = b & \wedge \\ votes[Max(votes)] \neq 0 & \wedge \\ ReconcilePhase2End(i, votes) & \end{pmatrix} \Rightarrow ReconcilePhase3(i, b)$$

6. $$\begin{pmatrix} FullyConnected(A) & \wedge \\ ReconcilePhase3(i, b) & \end{pmatrix} \Rightarrow \begin{pmatrix} \begin{pmatrix} StackToChain(i, b) & \vee \\ AddToChain(i, b) & \end{pmatrix} & \circ \\ ReconcilePhase2Begin(i) & \end{pmatrix}$$

7. $$\begin{pmatrix} ReconcilePhase2Begin(i) & \wedge \\ \forall j \in Contains(i) : C_i = C_j & \end{pmatrix} \Rightarrow \begin{pmatrix} votes[Max(votes)] = 0 & \wedge \\ ReconcilePhase2End(i, votes) & \end{pmatrix}$$

8. $$\begin{pmatrix} votes[Max(votes)] = 0 & \wedge \\ ReconcilePhase2End(i, votes) & \end{pmatrix} \Rightarrow ReconcileFinalize(i)$$

9. $$\begin{pmatrix} ReconcileFinalize(i) & \wedge \\ R_i = \emptyset & \end{pmatrix} \Rightarrow \neg reconcileInProgress'_i$$

10. $$\begin{pmatrix} ReconcileFinalize(i) & \wedge \\ R_i \neq \emptyset & \end{pmatrix} \Rightarrow \begin{pmatrix} b = R_i[Length(R_i)] & \wedge \\ (SendCommitBlock(i, b) & \circ \\ \ StackToChain(i, b) & \circ \\ \ ReconcileFinalize(i)) & \end{pmatrix}$$

11. $$\begin{pmatrix} FullyConnected(A) & \wedge \\ C_i = C_j & \wedge \\ SendCommitBlock(i, b) \circ StackToChain(i, b) & \end{pmatrix} \Rightarrow C'_i = C'_j$$

■

Step 1 establishes that the commencement of *Reconcile Phase 2* within a fully connected system implies that a majority of agents possess the reconciling agent's blockchain as a subchain per Lemma 3.1. Step 2 uses this assertion to establish that the set of agents possessing the reconciling agent's blockchain as a subchain is nonempty (Equations 3.13 and 3.37).

Subsequently, steps 3, 4, and 5 assert that with an undefined number of agents in the system, there must exist some agent in the set of neighboring agents that contains the reconciling agent's blockchain as a subchain that possesses the successor block chosen by the reconciling agent if one exists (Equation 3.28) and that the reconciling agent will proceed to *Reconcile Phase 3*. Step 6 establishes that, when the agent progresses to the *Reconcile Phase 3* process (Equation 3.29), it will commit the successor block to its blockchain, where that successor block will be either from its reconcile stack (Equation 3.30) or the blockchain of some other agent (Equation 3.31). In both cases, the agent will recommence the *Reconcile Phase 2* after adding the block.

Steps 7 and 8 establish that transition from *Reconcile Phase 2* to the *Reconcile Finalize* handler in a fully connected system will only occur when there exists at least one other agent with an identical blockchain (Equation 3.28). Finally, steps 9, 10, and 11 establish that, in a fully connected system where the reconciling agent commits any remaining blocks from its reconcile stack and broadcasts `Commit Block` messages for each, any agent with the same blockchain will continue to have a matching blockchain (Equations 3.33 through 3.35).

Lemma: Reconciliation Reduced Number of Differing Blockchains

Lemma 3.3 asserts that the completion of the reconcile process by an agent in a fully connected system reduces the total number of unique blockchains in the set of all blockchains by one. This is proven directly by combining Lemmas 3.1 and 3.2.

Lemma 3.3. *In a fully connected system where an agent has a unique local blockchain, completion of the reconcile process by an agent reduces the total number of differing blockchains in the system by one.*

$$\forall i \in A : \begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Cardinality(Chains) = k & \wedge \\ FullyConnected(A) & \wedge \\ reconcileInProgress_i & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix} \Rightarrow \diamond \begin{pmatrix} Cardinality(Chains') = k - 1 & \wedge \\ \neg reconcileInProgress'_i & \end{pmatrix} \tag{3.45}$$

Proof. Given a set of agents, A, where each agent $i \in A$ maintains a local blockchain, $C_i \in Chains$, and a reconcile stack, R_i.

1. $reconcileInProgress_i \Rightarrow ReconcileBegin(i)$
2. $ReconcileBegin(i) \Rightarrow ReconcilePhase1(i)$
3. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ FullyConnected(A) & \wedge \\ ReconcilePhase1(i) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} MajorityContains(i) & \wedge \\ ReconcilePhase2Begin(i) & \end{pmatrix}$
4. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ FullyConnected(A) & \wedge \\ ReconcilePhase2Begin(i) & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} \exists j \in A : C_i = C_j & \wedge \\ \neg reconcileInProgress_i & \end{pmatrix}$
5. $\begin{pmatrix} Cardinality(Chains) = k & \wedge \\ C_i \neq C_j & \wedge \\ C'_i = C'_j & \end{pmatrix} \Rightarrow Cardinality(Chains) = k - 1$

■

Steps 1 and 2 establish that a reconcile in progress implies that the *Reconcile Begin* handler was triggered, which subsequently triggers the *Reconcile Phase 1* process. Using Lemma 3.1, step 3 asserts that an agent in *Reconcile Phase 1* in a fully connected system implies that a majority of agents will eventually contain the blockchain of the reconciling agent as a subchain and that the agent will transition to *Reconcile Phase 2*. Step 4 asserts that in a fully connected system, an agent in *Reconcile Phase 2* will eventually attain a blockchain that matches that of another agent prior to concluding the reconcile process (Lemma 3.2). The final step asserts that given the set of all blockchains, if two blockchains do not match in one state but match in the next, then the number of unique blockchains in the set of all blockchains will be reduced by one following the state transition.

Theorem: Uniform Chain Property

Final proof of the *Uniform Chain* property demonstrates that the number of unique blockchains will eventually be one (i.e., all agents will have the same blockchain). This property is proven directly by building on Lemma 3.3 and the definition of the sequential reconcile mechanism described in Section 3.3 (i.e., Equation 3.38).

Theorem 3.5. *Uniform Chain: In a fully connected system where all agents have opportunities to reconcile blockchains with each other, one uniform blockchain will emerge.*

$$\forall i \in A : \begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Cardinality(Chains) > 1 & \wedge \\ FullyConnected(A) & \wedge \\ SequentialReconcile(A) & \end{pmatrix} \Rightarrow \Diamond(Cardinality(Chains') = 1) \quad (3.46)$$

Proof. Given a set of n agents, A, where each agent $i \in A$ maintains a local blockchain, $C_i \in AllChains$, and a reconcile stack, R_i.

1. $SequentialReconcile(A) \Rightarrow \Box(\exists_{=1} i \in A : reconcileInProgress_i)$
2. $A = \{a_1, \ldots, a_n\} \Rightarrow Cardinality(Chains) \leq n$
3. $\begin{pmatrix} Cardinality(Chains) > 1 & \wedge \\ SequentialReconcile(A) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} Cardinality(Chains) = k & \wedge \\ \exists_{=1} i \in A : reconcileInProgress_i & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix}$
4. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Cardinality(Chains) = k & \wedge \\ FullyConnected(A) & \wedge \\ reconcileInProgress_i & \wedge \\ \neg(\exists j \in A : C_i = C_j) & \end{pmatrix} \Rightarrow \Diamond \begin{pmatrix} Cardinality(Chains') = k - 1 & \wedge \\ \neg reconcileInProgress_i & \end{pmatrix}$
5. $\begin{pmatrix} Cardinality(Chains) = k - 1 & \wedge \\ SequentialReconcile(A) & \wedge \\ FullyConnected(A) & \end{pmatrix} \Rightarrow \begin{pmatrix} Cardinality(Chains') > 1 & \vee \\ Cardinality(Chains') = 1 & \end{pmatrix}$

6. $\begin{pmatrix} SequentialReconcile(A) & \wedge \\ FullyConnected(A) & \wedge \\ (Cardinality(Chains) > 1 & \vee \\ Cardinality(Chains) = 1) & \end{pmatrix} \Rightarrow \Diamond(Cardinality(Chains) = 1)$

■

Step 1 restates the definition of the assumed reconcile mechanism in that there will always only be a single agent that is in the reconcile process. Step 2 establishes that if there are n agents in the set of all agents, then the cardinality of the set of all blockchains must be less than or equal to n. Step 3 asserts that the number of blockchains being greater than one and the sequential reconcile mechanism active implies that there will eventually exist an agent reconciling that does not share a blockchain with any other agent. Using Lemma 3.3, step 4 asserts that in such a case, the number of unique blockchains will be reduced by one after the reconciling agent concludes the reconcile progress. Finally, steps 5 and 6 indicate that upon completion of a reconciliation, the reduced number of unique blockchains will be greater than one or equal to one. Subsequent iterations of the *SequentialReconcile* process will further reduce the set's cardinality. Thus, so long as there is an agent reconciling a blockchain that does not match any other agent's, the number of unique blockchains will eventually be reduced to a single, uniform blockchain. That is, reducing the cardinality of the set of unique chains by one every time an agent reconciles will lead to a set with a single element following a sequential reconcile by all agents.

3.4.6 Proof of the Block Propagation Property

The final property, *Block Propagation*, asserts that in a fully connected system where there is more than one unique blockchain and the sequential reconcile mechanism is active, all blocks that exist in the system will eventually exist in the blockchains of every agent.

Theorem 3.6. *Block Propagation: In a fully connected system, all blocks will eventually be committed to all locally maintained blockchains.*

$$\forall i \in A : \begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Cardinality(Chains) > 1 & \wedge \\ FullyConnected(A) & \wedge \\ SequentialReconcile(A) & \end{pmatrix} \Rightarrow \Diamond(\forall b \in \{B_1 \cup \ldots \cup B_n\}, i \in A : b \in C_i) \tag{3.47}$$

Proof. Given a set of n agents, A, where each agent $i \in A$ maintains a local blockchain, $C_i \in AllChains$, and a reconcile stack, R_i.

1. $\begin{pmatrix} A = \{a_1, \ldots, a_n\} & \wedge \\ Cardinality(Chains) > 1 & \wedge \\ FullyConnected(A) & \wedge \\ SequentialReconcile(A) & \end{pmatrix} \Rightarrow \Diamond(Cardinality(Chains') = 1)$
2. $(Cardinality(Chains) = 1) \Rightarrow \forall i, j \in A : (C_i = C_j)$
3. $\forall b : Agent(b) \in A \Rightarrow \exists i \in A : b \in B_i$
4. $\forall i \in A : AllIdle(i) \Rightarrow \forall b \in B_i : (b \in C_i)$
5. $\begin{pmatrix} \forall i, j \in A : (C_i = C_j) & \wedge \\ \forall b \in B_i : (b \in C_i) & \end{pmatrix} \Rightarrow \forall b \in \{B_1 \cup \ldots \cup B_n\}, i \in A : b \in C_i$

■

Steps 1 and 2 restate the *Uniform Chain* property, proven in Theorem 3.5, asserting that there will eventually be a single blockchain in the set of all blockchains. Step 3 restates the *No Block Loss* property proven in Theorem 3.3, establishing that there exists at least one agent in the system that maintains any proposed block. Step 4 restates the *Idle Stop* property proven in Theorem 3.4, establishing that when all event handlers are idle on a particular agent, all blocks maintained by that agent will be present in its blockchain. The final step asserts that these three properties combined imply that all blocks in the system will be present in the uniform blockchain shared by all agents.

3.5 Chapter Summary

This chapter examined the various event handlers specified by the UVSLP and formally defined them in the form of predicates. Supplemental definitions were provided to formalize additional aspects of the protocol properties, such as the requirements for all event handlers to be idle and the existence of a sequential reconcile mechanism. An extended set of logical operators describing temporal behaviors were then applied to the formally defined behaviors of the protocol to prove its satisfaction of the six properties asserted in its original specification. Chapter 4 describes the methodology and results of the protocol in a simulation environment to provide a characterization of its general performance and scalability.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Performance Testing

This chapter describes experimentation conducted to characterize the performance and scalability of the UVSLP. Background information is provided on the prototype implementation and physically-based simulation environment in which testing was conducted. A factorial design was adopted to investigate how the number of vehicles, network loss rates, and event generation rates affect the number of messages and volume of data produced by the protocol. The chapter culminates with a discussion of the results of this experimentation.

4.1 Background

Early work on the UVSLP included an prototype implementation of the protocol incorporated into the NPS ARSENL multi-unmanned aerial vehicle (UAV) system [2]. The ARSENL system supports experimentation on drone swarms, encompassing large numbers of autonomous UAVs [6]. In addition to live environments, the ARSENL system includes a software-in-the-loop (SITL) simulation capability providing a physically-based simulated environment [5]. The SITL environment supports experimentation with on-vehicle software and firmware that closely aligns with observed real-world results. SITL environment functionality includes the ability to simulate lossy networks such that arbitrary network packets are subject to probabilistic omission rather than being forwarded to individual vehicles. This simulated environment enables comprehensive, close to realistic testing of the UVSLP while circumventing the resource-intensive nature of real-world experimentation.

The ARSENL system is built upon Robot Operating System (ROS) [6], a middleware architecture for robotic systems that runs in Linux and Unix environments [22]. ROS provides Python and C++ application programming interfaces (APIs) to support the development of independent modules referred to as ROS nodes. Within the ARSENL system, the autonomous functionalities governing individual vehicles are implemented as ROS nodes using the Python API. Communication between ROS nodes uses a publisher-subscriber model in which nodes publish messages to named topics to which other nodes can subscribe.

The UVSLP prototype was implemented as two supplementary ROS nodes within the existing ARSENL onboard architecture [2]. The *blockchain_bridge* node implements a vehicle's protocol event handlers. These event handlers effect the transmission of messages to other UAVs on the network by publishing ROS messages to topics to which the ARSENL *network_bridge* node that manages network communications subscribes. The *network_bridge* node repackages the ROS message's contents into an ARSENL message that is transmitted as a User Datagram Protocol (UDP) broadcast [6]. The *network_bridge* node on a receiving vehicle recreates the original ROS message and publishes it to a topic to which the receiving vehicle's *blockchain_bridge* node subscribes. Additionally, the *blockchain_bridge* node implements the vote timers of relevant event handlers and allows ten seconds for commit request votes and five seconds for each reconcile phase vote.

A *blockchain_event_generator* node generates events to be logged by the vehicle. Each event is composed of a timestamp, an event identifier, and the event data. As the actual data has no impact on the protocol itself, the event identifiers and event data are randomly generated. Event identifiers are random integers between one and 254, and event data are sequences of 16, 32, 64, 128, or 256 bytes of random binary data. The *blockchain_event_generator* node publishes events to a ROS topic to initiate the *blockchain_bridge* node's *block generation and commit* event handlers. Due to the ARSENL system's use of UDP for intervehicle communications, the size of UVSLP blocks was limited to no more than 1024 bytes to ensure that a block could be encoded into a single UDP packet. To more accurately reflect real-world conditions in which the rate of logged events is non-uniform, the time between generated events is determined by sampling from an exponential distribution. Setting the scale parameter of the exponential distribution probability density function is used to vary the mean time between events as described in Section 4.2.

4.2 Experimental Design

Experiments were designed to test several hypotheses related to the general performance and scalability of the UVSLP. These hypotheses are as follows:

1. The average number of messages per minute and the average volume (i.e., bytes) of data per minute are positively correlated. This hypothesis is premised on the assumption that, as the number of messages increases, more data will be transmitted over the network.

2. An increase in the number of vehicles will increase the average number of messages per minute and the average volume of data per minute at a predictable rate. As more vehicles are added to the system, all forms of message traffic will increase accordingly. More events will be generated, more blocks will be proposed for addition, more responses will be transmitted, and, ultimately, more reconcile processes will be required. This hypothesis supposes that the required message traffic will prove to be a function of the system size (i.e., the number of vehicles).
3. An increase in network loss rate will lead to a decrease in the average number of messages per minute and the average volume of data per minute. Higher network loss rates will lead to fewer vehicles receiving requests. As a result, fewer responses will be required resulting in a lower overall traffic volume.
4. Increasing the mean time between events will decrease the average number of messages per minute and the average volume of data per minute. By observation, it is evident that, as the mean time between events increases, blocks will be built and proposed less frequently. We hypothesize that aggregate network traffic will decrease.
5. No agent will trigger the reconciliation process when the network loss rate is 0 percent, which equates to a fully connected network environment. In a fully connected system, all vehicles will receive all transmitted messages. Since all vehicles will receive all proposed blocks, there should be no opportunity for individual blockchains to diverge.

The prototype implementation was used in the SITL simulation environment to investigate how the number of vehicles, network loss rate, and event generation rate affect the number of messages and volume of data transmitted per vehicle. A factorial design was adopted to test combinations of select discrete values of each independent variable. Functionality was added to the prototype implementation to record information related to each message transmitted by each vehicle. This information included a timestamp of when the message was sent, the identifier of the sending vehicle, the type of message sent, and the size of the message in bytes.

Experiments were conducted with system sizes of 5, 10, 20, 30, 40, and 50 vehicles. The individual laptop computers used for these experiments were not capable of simulating more than 10 vehicles at a time without the computational requirements impacting performance. To circumvent this limitation, the ARSENL system provides functionality to distribute simulated vehicles across multiple hosts using the local network. As such, trials involved between one and five identical laptops, depending on the number of vehicles tested.

Network loss rates were simulated using the functionality provided by the SITL environment, as described in Section 4.1. Experiments were conducted with packet loss probabilities of 0.00, 0.25, 0.50, and 0.75. A packet loss probability of 0.00 equates to a fully connected network environment where every packet is delivered. On the other hand, a packet loss probability of 0.75 equates to a network environment where any particular packet has a 75 percent likelihood of being dropped.

The mean time between generated events for each trial was one of the following: 5, 15, 30, and 60 seconds. Notably, the event generation rate is not constant. Rather, the actual time between events in the prototype implementation is not constant, so the event generation rate reflects the value of the exponential distribution probability density function scale parameter, β, of Equation 4.1. The vehicle-specific wait times between the generation of individual events are sampled from this distribution.

$$f(x) = \frac{1}{\beta} e^{-x/\beta} \tag{4.1}$$

Each combination of number of vehicles, network loss rates, and event generation rates were tested for a total of 96 trials. Each simulation trial was allowed to run for a minimum of 30 minutes. Following each trial, the recorded message data from each vehicle was aggregated and normalized to reflect exactly 30 minutes from the earliest timestamp included in that trial's data to maintain consistency across all of the trials. The aggregated information was then used to determine the average number of messages per minute and data volume per minute transmitted per vehicle for each trial and to characterize the types of messages that were transmitted.

4.3 Results

The results of the trials support the hypothesis that there is a positive correlation between the average number of messages per minute and the volume of data per minute. A scatter plot of both measures across all 96 trials is provided in Figure 4.1. These results show a strong correlation between the two dependent variables. Further, the results indicate that the relationship between number of messages and data volume is linear. This suggests that an independent variable affecting one will similarly affect the other.

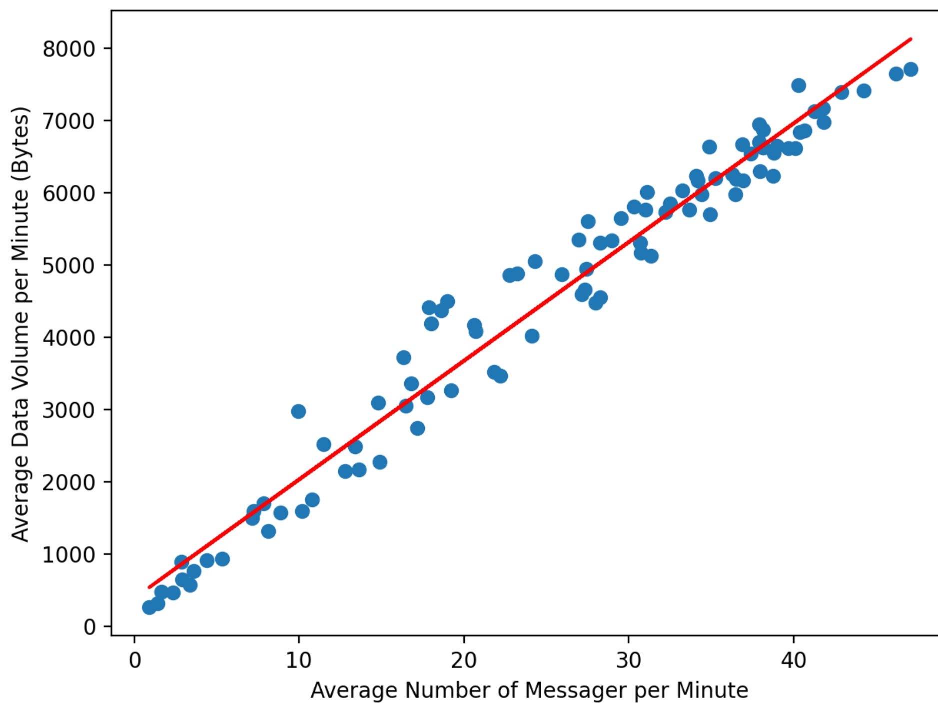

Figure 4.1. Correlation between number of messages and data volume.

Figure 4.2 shows the average number of messages transmitted by each vehicle versus the number of vehicles in the system. A separate graph is provided for each of the experimental network loss rates. Each line in the respective graphs represents a specific mean time between logged events, as indicated. Figure 4.3 shows the same graphs for the average volume of data per minute rather than the average number of messages.

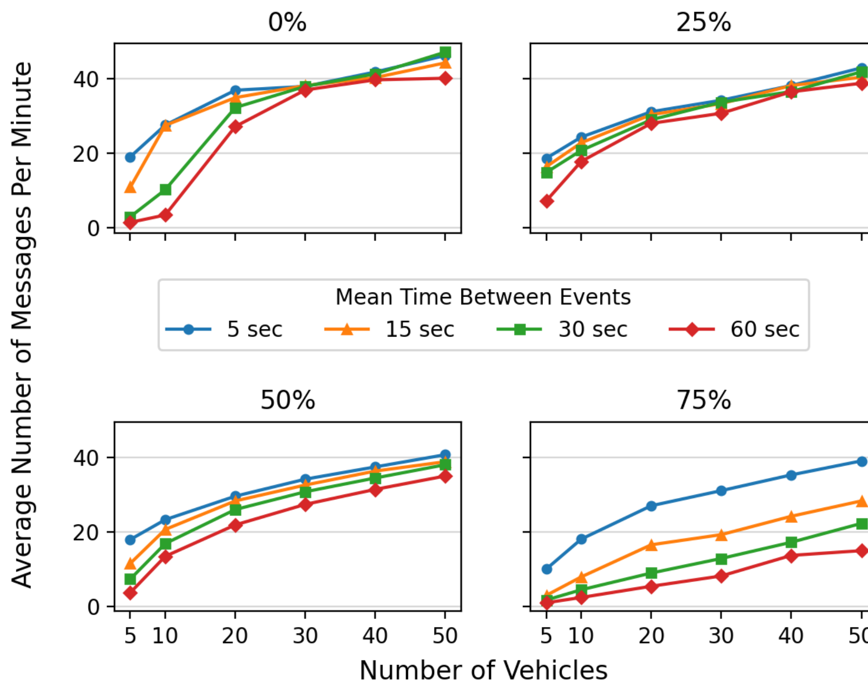

Figure 4.2. Average number of messages by network loss rate.

Since number of messages and volume of data are positively correlated, the conclusions to be drawn from Figures 4.2 and 4.3 are similar. Lower mean times between generated events results in a higher numbers of messages and higher volumes of data for all system sizes and network loss rates. These results support the hypothesis that an increase in event generation rate increases both the average number of messages and average data volume. Additionally, the average number of messages and average data volume increase as the number of vehicles increases, regardless of network loss rate or event generation rate. This increase grows predictably in most cases, and the growth appears to be either linear or logarithmic (additional experimentation with increasing system sizes will be required to more fully quantify the growth characteristics). Interestingly, the rate of increase appears much less predictable when the network loss rate is 0 percent. In particular, lower-than-expected message and data volumes were observed in systems of five and 10 UAVs and mean times between events of 30 and 60 seconds. This anomaly is discussed in further detail below.

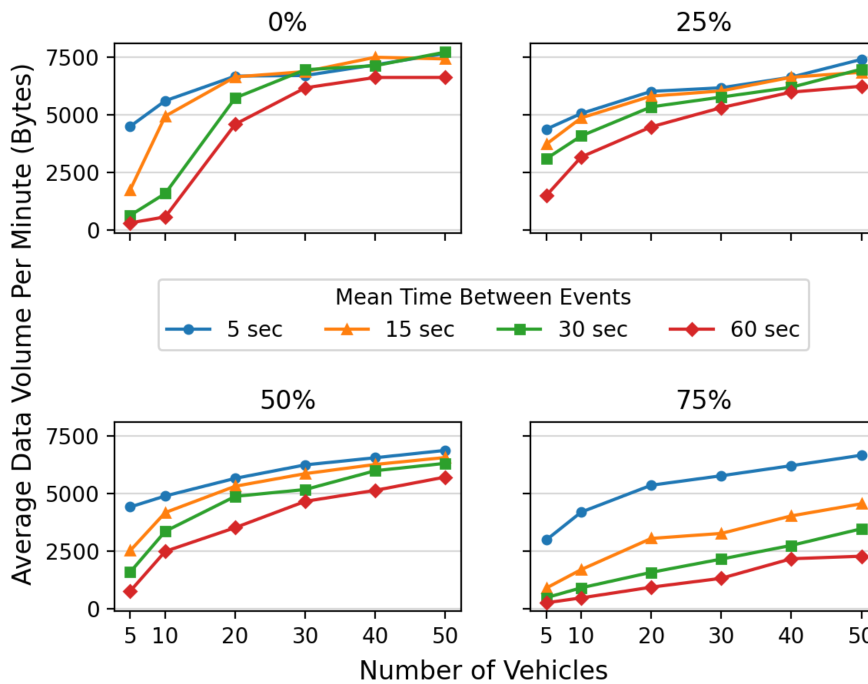

Figure 4.3. Average data volume by network loss rate.

Figure 4.4 shows the average number of messages versus the number of vehicles for each of the test event generation rates. Each line represents a specific network loss rate. Figure 4.5 shows the same set of graphs for the average volume of data per vehicle per minute. Once again, the average number of messages and average data volume are shown to increase as the number of vehicles increases. Similar to Figures 4.2 and 4.3, the increase in number of messages and data volume appears to grow predictably in most cases with either a linear or logarithmic trajectory. However, these graphs provide additional insight into the previously noted anomalous performance with small systems and longer mean times between events. One possible hypothesis might be that an increase in network loss rate decreases the average number of messages per minute. That is, higher loss rates equate to fewer messages received and therefore fewer responses. While this hypothesis is supported by the results when there was packet loss, two interesting phenomena occurred with a network loss rate of 0 percent.

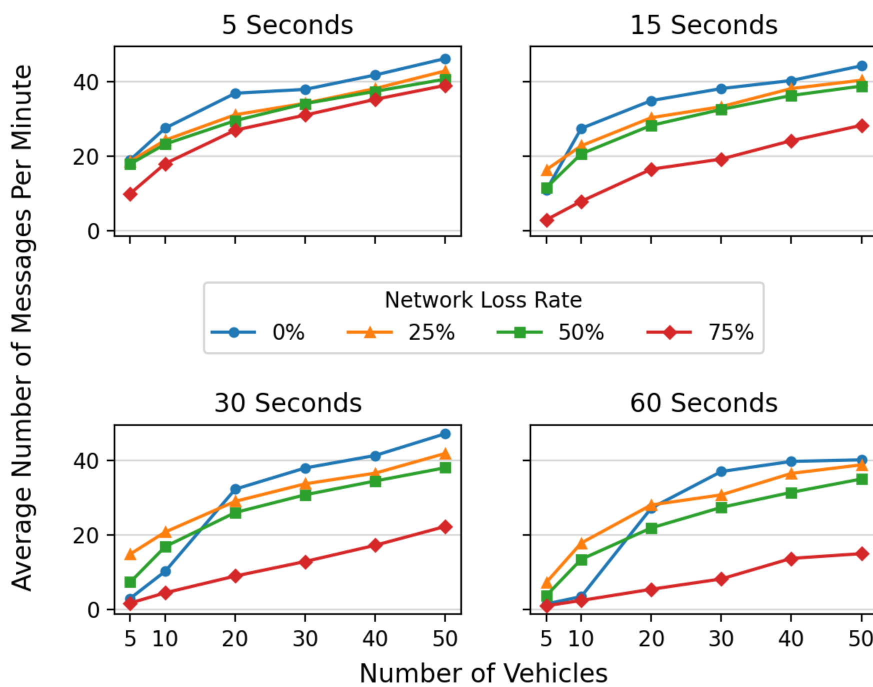

Figure 4.4. Average number of messages by mean time between events.

The first is that as the mean time between events increased, more vehicles were required for the average number of messages to surpass that of the other tested network loss rates. In general, lower network loss rates equate to higher message volumes. With longer mean times between events, however, this relationship does not manifest until the system size exceeds some number of vehicles (e.g., 20 for a mean time between events of 60 seconds).

The second phenomenon is that a mean time of 60 seconds between generated events shows a distinct plateau of the positive trend between 40 and 50 vehicles at a network loss rate of 0 percent. The same phenomenon occurred when the mean time between events was 15 seconds and the network loss rate was 0 percent. In that case, however, there was a slight decrease in the average message volume when the number of vehicles increases from 40 to 50. More specifically, the average message volume decreases from 7493.38 bytes to 7420.63 bytes per minute. As such, the hypothesis that the average volume of data increases as the number of vehicles increases is not fully supported by the results. Additional

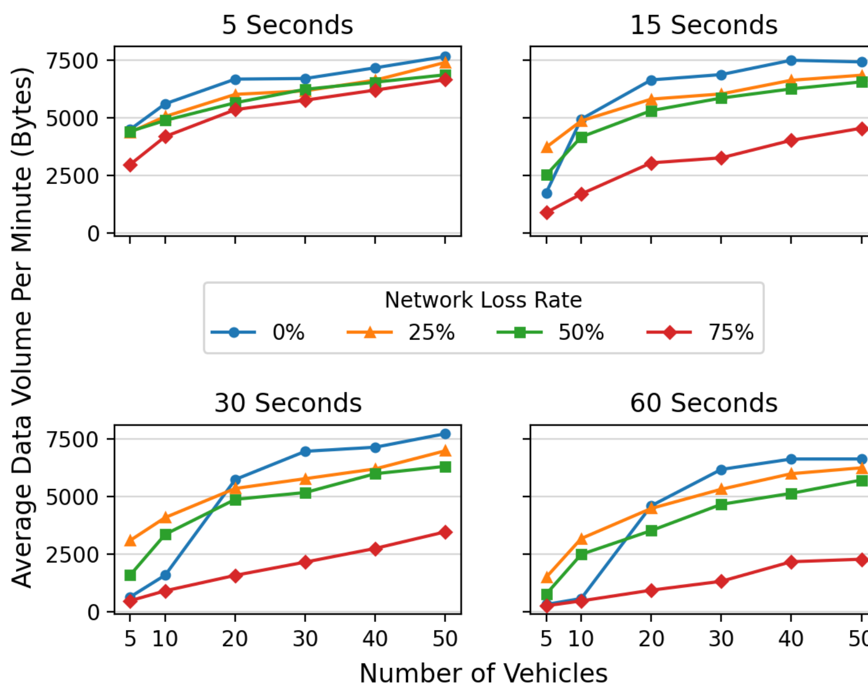

Figure 4.5. Average data volume by mean time between events.

experimentation will be required, however, to determine whether this result is indicative of an actual protocol characteristic or merely a single anomalous result.

The most surprising phenomenon also occurred with a network loss rate of 0 percent. In most cases, message volume is inversely correlated with network loss rate. For fully connected systems, however, this correlation also appears to be dependent on number of vehicles and event generation rate. With a mean time between events of five seconds, for instance, the 0 percent loss-rate experiment indicated message volume similar to the higher loss-rate experiment for systems of five vehicles. With a mean time between events of 15 seconds, however, the 0 percent loss-rate volume was less than all but the 75 percent loss-rate volume with fewer than 10 vehicles. For a mean times between events of 30 and 60 seconds, the 0 percent volume did not exceed all of the other loss rate results until the 20 and 30 vehicle experiments respectively. This phenomenon merits further investigation.

Figure 4.6 shows the distribution of message types transmitted across trials when the network loss rate was 0 percent, mean time between events was 60 seconds, and the number of vehicles was between 5 and 30. When there were either 5 or 10 vehicles, there was no reconciliation process as hypothesized when the network loss rate is 0 percent. The only messages sent were the `Block Add Request`, `Block Add Response`, and `Commit Block` messages. However, this hypothesis was unsupported when the number of vehicles reached 20, where messages related to the reconciliation process were sent (i.e., `Chain Contain Query`, `Chain Contain Response`, `Chain hash Successor Query`, `Next in Sequence Response`, `Request for Block`, and `Block Response` messages). `Chain Contain Response` messages were the most commonly observed reconcile process messages.

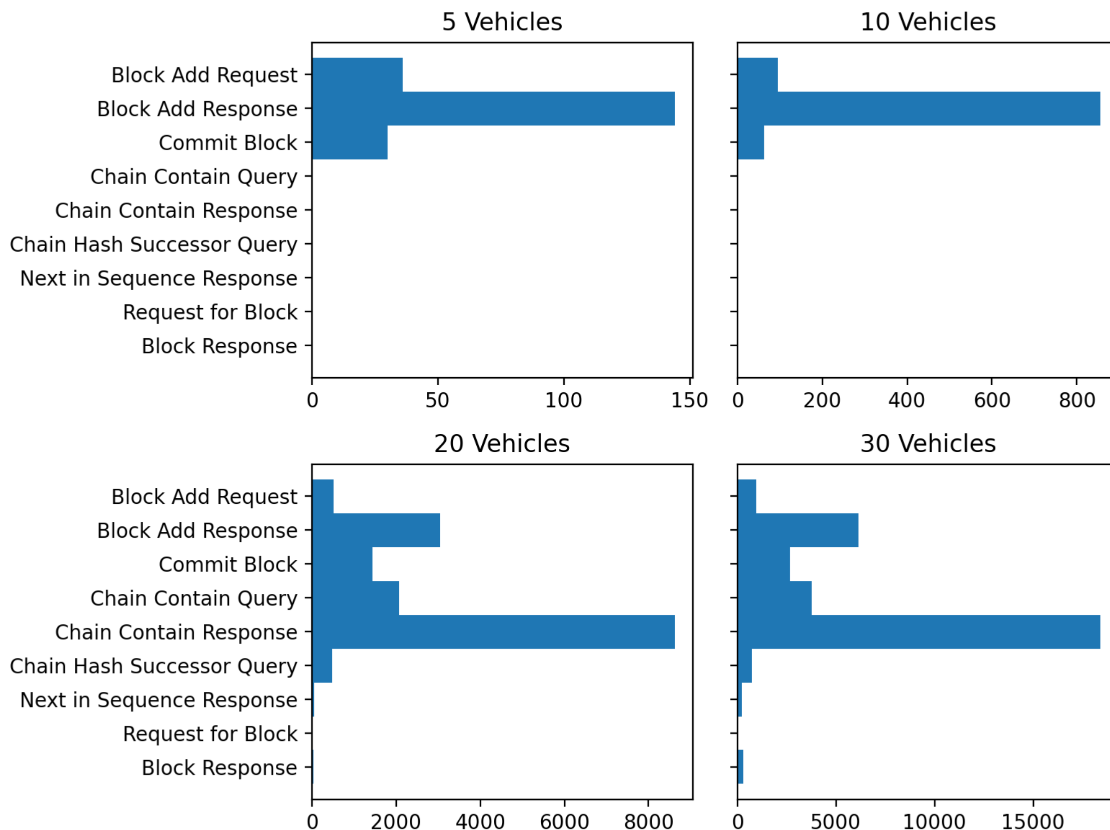

Figure 4.6. Distribution of message types at 0 percent network loss rate and 60 second mean time between events.

Upon reflection, this is actually an expected result of the voting process. As the number of vehicles increases, the probability of multiple vehicles proposing a block for addition to the blockchain nearly simultaneously increases. That is, it becomes increasingly likely that situations will arise in which a vehicle is waiting for a vote timer to expire when another vehicle sends a `Commit Block` message. Per the protocol definition, when this occurs the vehicle will ignore the commit despite receiving the message, thus causing a divergent blockchain and a subsequent reconcile process. Specifically, the testbed UVSLP implementation utilized a 10-second voting timer for proposed blocks. Once a vehicle initiates a block proposal, any vehicle that initiates its own block proposal before the expiration of the first vehicle's timer will ignore the eventual `Commit Block` message while its own timer is active. Further, the second vehicle's `Commit Block` will be rejected by any vehicle that added the first vehicle's proposed block to its blockchain.

This is evidently a race condition that arises any time a vehicle proposes a block for addition. As the mean time between events decreases or the number of vehicles increases, the probability of the race condition manifesting increases leading to external blocks that cannot be committed. It is also not surprising that the highest number of messages would be `Chain Contain Response` messages in scenarios where reconciliations occur due to a reconciling agent potentially reducing and comparing its blockchain to those of available agents multiple times before finding a mutual chain that it shares with the majority. Interestingly, this race condition exists in all loss experiments. It does not lead to hypothesis contradictions for those systems, however, and was thus not noticed. A true understanding of this phenomenon requires a formal characterization of the relationship between the race condition, the number of vehicles, the network loss rate, the event generation rate, and the voting timer duration. This, however, is left to future work for the time being.

4.4 Chapter Summary

This chapter investigated how the number of vehicles, network loss rates, and event generation rates influence the performance and scalability of the UVSLP. Across 96 trials, an increase in the number of vehicles increased the average number of messages and average volume of data transmitted within the system, except for an anomaly when the network loss probability was 0 percent and the mean time between events was 15 seconds where 50 vehicles produced less data volume per minute than 40 vehicles. In most trials, an increase

in network loss probability and mean time between events decreased these measures. However, findings suggest that reconciliation may still occur in a fully connected system with no lost packets due to a race condition in which a vehicle is waiting for a vote timer to expire when receiving an external block. As a result, the number of messages and volume of data may be significantly reduced in a fully connected system so long as the number of vehicles is low enough and the mean time between events is high enough such that the probability of such a race condition is also reduced.

CHAPTER 5: Conclusion

This thesis formally described event handler behaviors and data structure states of the UVSLP using predicate logic. Supplementary definitions were provided to represent assumptions of the protocol that were not specified in its original presentation but that are useful for the proof of its claimed properties. By incorporating an expanded set of logic operators based upon the Temporal Logic of Actions, the six fundamental properties of the protocol were formally defined in the context of temporal behavior. Each of its six properties were then formally proven to be satisfied by the protocol. This work results in a mathematically-proven protocol that improves the availability and resiliency of data in multivehicle autonomous systems.

Additionally, this thesis described experimentation conducted to characterize the performance and scalability of the UVSLP. This experimentation investigated how the number of messages and volume of data produced by the protocol were affected by the number of vehicles, network loss rates, and event generation rates of the system. A prototype implementation from the protocol's original presentation was used in a simulated environment to test combinations of selected discrete values of each independent variable. Results showed that an increase in the number of vehicles and event generation rates increases the number of messages and volume of data. On the other hand, increases in network loss rates decrease the number of messages and volume of data. The results of this experiment also revealed a race condition where a request to commit a block may be broadcast to other vehicles that are waiting for timers to expire. Even in situations where there is no loss of packets on the network, vehicles may be unable to commit an external block to their local blockchain, thus imposing a requirement for future reconciliation.

5.1 Future Work

This thesis provided formal proofs of the protocol's correctness and characterization of its performance. However, there are additional areas for future work. These include:

Further Development of UVSLP – Proving the correctness of the protocol required the assumption of mechanisms not specified in the protocol's original presentation. In particular, the proofs assumed that a UVSLP implementation would possess a mechanism to systematically initiate a reconciliation process sequentially across agents at the conclusion of a given operation. A race condition was also revealed to occur even in situations where no packets are lost. Future work on the protocol may include specifying additional mechanisms and functionality to satisfy additional properties and improve performance.

Proving Additional Properties – The formal proofs provided in this thesis were limited to the six fundamental properties specified in its original presentation. There may exist additional properties that could be satisfied by either current or future iterations of the protocol.

Additional Testing – This thesis investigated how the number of vehicles, network loss rates, and event generation rates affect the number of messages and volume of data produced by the protocol. However, these independent variables were limited to a set of discrete values for each. Additional values may be worth investigating including higher numbers of vehicles to determine how the protocol further scales and increased granularity in event generation rates to provide more insight into the anomalies discovered. Additional independent variables may be worth investigating. For example, it is currently unknown how maximum block size and timer durations may affect these measures. There may also be other dependent variables worth measuring, such as the time to achieve a uniform blockchain at the conclusion of a mission.

Live-fly Experimentation – The experimentation performed in this thesis was limited to a simulation environment in which network loss rates and event generation rates were emulated using probability. Testing in live environments may provide a richer, more accurate characterization of the protocol's performance in real-world scenarios. For example, live-fly experimentation would enable testing disjoint networks in which subsets of vehicles separate and recombine rather than relying on the probabilistic ignoring of network packets.

List of References

[1] N. L. Carter, "Design and verification of a distributed ledger protocol for Distributed Autonomous Systems Using Monterey Phoenix," M.S. thesis, Naval Postgraduate School, Dec. 2020.

[2] P. J. Pommer, "Design and implementation of a distributed ledger to support data survivability in an unmanned multi-vehicle systems," M.S. thesis, Naval Postgraduate School, Mar. 2021.

[3] N. L. Carter, D. T. Davis, C. E. Irvine, and P. J. Pommer, "Verification of a distributed ledger protocol for distributed autonomous systems using Monterey Phoenix," in *Hawaii Internation Conference on System Sciences*, 2023.

[4] M. Auguston, "Monterey Phoenix, or how to make software architecture executable," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 1031–1040.

[5] 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*. IEEE, 2015, pp. 426–435.

[6] 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 *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1255–1262 [Online]. Available: https://doi.org/10.1109/ICRA.2016.7487257

[7] L. J. Austin III, "2022 national defense strategy of the United States including the 2022 nuclear posture review and the 2022 missile defense review," Department of Defense Washington United States, Rep., 2022 [Online]. Available: https://apps.dtic.mil/sti/citations/trecms/AD1183514

[8] M. Hati, "Swarm robotics: A technological advancement for human-swarm interaction in recent era from swarm-intelligence concept," *International Journal of Science and Research (IJ // IJSR)*, vol. 5, no. 5, pp. 1165–1168, 2016.

[9] G. Francesca and M. Birattari, "Automatic design of robot swarms: Achievements and challenges," *Frontiers in Robotics and AI*, vol. 3, 2016 [Online]. Available: https://doi.org/10.3389/frobt.2016.00029

[10] M. Coppola, K. N. McGuire, C. De Wagter, and G. C. H. E. de Croon, "A survey on swarming with micro air vehicles: Fundamental challenges and constraints," *Frontiers in Robotics and AI*, vol. 7, p. 18, 2020 [Online]. Available: https://doi.org/10.3389/frobt.2020.00018

[11] N. Carter, P. Pommer, D. T. Davis, and C. E. Irvine, "Increasing log availability in unmanned vehicle systems," in *National Cyber Summit (NCS) Research Track 2021*, K.-K. R. Choo, T. Morris, G. Peterson, and E. Imsand, Eds. Cham: Springer International Publishing, 2022, pp. 93–109.

[12] A. Sunyaev, *Distributed Ledger Technology*. Springer International Publishing, 2020, pp. 265–299 [Online]. Available: https://doi.org/10.1007/978-3-030-34957-8_9

[13] T. Alladi, V. Chamola, N. Sahu, and M. Guizani, "Applications of blockchain in unmanned aerial vehicles: A review," *Vehicular Communications*, vol. 23, p. 100249, 2020.

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

[15] L. Lamport, "The part-time parliament," in *Concurrency: The Works of Leslie Lamport*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 277–317.

[16] L. Lamport, "Verification and specification of concurrent programs," in *A Decade of Concurrency Reflections and Perspectives*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 347–374.

[17] L. Lamport, "A new approach to proving the correctness of multiprocess programs," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 84–97, 1979 [Online]. Available: https://doi.org/10.1145/357062.357068

[18] L. Lamport, "What good is temporal logic?" in *IFIP Congress*, 1983, vol. 83, pp. 657–668.

[19] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (Sfcs 1977)*. IEEE, 1977, pp. 46–57 [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[20] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 455–495, 1982 [Online]. Available: https://doi.org/10.1145/357172.357178

[21] L. Lamport, "Specifying concurrent systems with TLA+," *Calculational System Design*, pp. 183–247, 1999.

[22] A. Koubâa *et al.*, *Robot Operating System (ROS).* Springer, 2017, vol. 1.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

DUDLEY KNOX LIBRARY

NAVAL POSTGRADUATE SCHOOL

WWW.NPS.EDU

WHERE SCIENCE MEETS THE ART OF WARFARE