



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

**DESIGN AND INFORMAL VERIFICATION OF A
DISTRIBUTED LEDGER PROTOCOL FOR DISTRIBUTED
AUTONOMOUS SYSTEMS USING MONTEREY PHOENIX**

by

Nickolas L. Carter

December 2020

Thesis Advisor:
Co-Advisor:

Cynthia E. Irvine
Duane T. Davis

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
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 December 2020	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE DESIGN AND INFORMAL VERIFICATION OF A DISTRIBUTED LEDGER PROTOCOL FOR DISTRIBUTED AUTONOMOUS SYSTEMS USING MONTEREY PHOENIX			5. FUNDING NUMBERS	
6. AUTHOR(S) Nickolas L. Carter				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Autonomous vehicle systems, to include multi-vehicle systems, are becoming increasingly relevant in military operations. A problem emerges, however, when logging data within these systems. In particular, loss of individual vehicles and inherently lossy and noisy communications environments can result in the loss of important mission data. This thesis presents a novel distributed ledger protocol that can be used to ensure that the data in such a system survives. We demonstrate the behavioral correctness of this protocol using informal verification methods and tools provided by the Monterey Phoenix project. We further verified the correctness of this protocol through the conduct of implementation field tests at Camp Roberts, CA.				
14. SUBJECT TERMS distributed ledger, blockchain, UAV, MP, Monterey Phoenix, Zephyr, ARSENL			15. NUMBER OF PAGES 165	
			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	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**DESIGN AND INFORMAL VERIFICATION OF A DISTRIBUTED
LEDGER PROTOCOL FOR DISTRIBUTED AUTONOMOUS SYSTEMS
USING MONTEREY PHOENIX**

Nickolas L. Carter
Civilian, CyberCorps: Scholarship for Service
BS, University of Maryland University College, 2017

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
December 2020**

Approved by: Cynthia E. Irvine
Advisor

Duane T. Davis
Co-Advisor

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Autonomous vehicle systems, to include multi-vehicle systems, are becoming increasingly relevant in military operations. A problem emerges, however, when logging data within these systems. In particular, loss of individual vehicles and inherently lossy and noisy communications environments can result in the loss of important mission data. This thesis presents a novel distributed ledger protocol that can be used to ensure that the data in such a system survives. We demonstrate the behavioral correctness of this protocol using informal verification methods and tools provided by the Monterey Phoenix project. We further verified the correctness of this protocol through the conduct of implementation field tests at Camp Roberts, CA.

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	Research Methodology	2
1.5	Organization	3
2	Background	5
2.1	Distributed Multi-Vehicle Autonomous Systems	5
2.2	Distributed Ledger Technologies	6
2.3	Block Chain	9
2.4	Consensus	11
2.5	Monterey Phoenix	13
2.6	Chapter Summary	16
3	Uniform Chain Protocol	17
3.1	Protocol Summary	18
3.2	Definitions and Terminology	18
3.3	Assumptions	20
3.4	Block Chain Data Structure	21
3.5	High-Level Overview	23
3.6	Detailed Chain Protocol.	27
3.7	Chapter Summary	64
4	Monterey Phoenix	65
4.1	Consensus	66
4.2	State	77
4.3	Chapter Summary	83

5 Design Validation	85
5.1 Monterey Phoenix	85
5.2 Field Tests	87
5.3 Chapter Summary	88
 6 Conclusion	 91
6.1 Implications	91
6.2 Future Work	92
 Appendix A Monterey Phoenix Consensus Results	 93
A.1 Analyzing the Results	93
A.2 Example.	94
A.3 Results	95
 Appendix B Monterey Phoenix State Results	 121
B.1 Analyzing the Results	121
B.2 Example.	122
B.3 Results	122
 Appendix C Field Test Results	 135
 List of References	 141
 Initial Distribution List	 143

List of Figures

Figure 2.1	Data availability within a distributed ledger system	8
Figure 2.2	Data loss within a centralized ledger system	8
Figure 2.3	Prevention of undetected data modification or removal in a block chain	10
Figure 3.1	The block chain data structure	22
Figure 3.2	An abstracted version of the Uniform Chain Protocol (UCP) . . .	24
Figure 3.3	Event handler diagram and symbol key	28
Figure 3.4	Build Data Block event handler	30
Figure 3.5	Process Next Deque Block event handler	32
Figure 3.6	Receive Request Add Block event handler	34
Figure 3.7	Receive Vote Add Block event handler	35
Figure 3.8	Vote Timer Expiration event handler	37
Figure 3.9	Receive Commit Block event handler	39
Figure 3.10	Reconcile Begin event handler	40
Figure 3.11	Reconcile Phase 1 event handler	42
Figure 3.12	Receive Request Chain Contain event handler	44
Figure 3.13	Receive Phase 1 Response event handler	45
Figure 3.14	Reconcile Phase 1 Timer Expiration event handler	47
Figure 3.15	Reconcile Phase 2 event handler	49
Figure 3.16	Receive Request Next in Sequence event handler	51
Figure 3.17	Receive Response Next in Sequence event handler	52

Figure 3.18	Reconcile Phase 2 Timer Expiration event handler	54
Figure 3.19	Reconcile Phase 3 event handler	56
Figure 3.20	Receive Request Block for Hash event handler	58
Figure 3.21	Receive Response Block For Hash event handler	60
Figure 3.22	Reconcile Phase 3 Timer Expiration event handler	61
Figure 3.23	Reconcile Finalize event handler	63
Figure 4.1	Abstracted version of the consensus process in UCP	70
Figure 4.2	Abstracted version of the state model of UCP.	80
Figure 5.1	Five Drone Bug	86
Figure 5.2	Altered Chain Bug	87
Figure 5.3	Zephyr unmanned aerial vehicle (UAV)	89
Figure 5.4	Mosquito Hawk UAV	89
Figure A.1	Consensus trace 1 of 28	95
Figure A.2	Consensus trace 2 of 28	96
Figure A.3	Consensus trace 3 of 28	96
Figure A.4	Consensus trace 4 of 28	97
Figure A.5	Consensus trace 5 of 28	98
Figure A.6	Consensus trace 6 of 28	99
Figure A.7	Consensus trace 7 of 28	100
Figure A.8	Consensus trace 8 of 28	101
Figure A.9	Consensus trace 9 of 28	101
Figure A.10	Consensus trace 10 of 28	102
Figure A.11	Consensus trace 11 of 28	103

Figure A.12	Consensus trace 12 of 28	104
Figure A.13	Consensus trace 13 of 28	105
Figure A.14	Consensus trace 14 of 28	106
Figure A.15	Consensus trace 15 of 28	107
Figure A.16	Consensus trace 16 of 28	108
Figure A.17	Consensus trace 17 of 28	109
Figure A.18	Consensus trace 18 of 28	110
Figure A.19	Consensus trace 19 of 28	111
Figure A.20	Consensus trace 20 of 28	112
Figure A.21	Consensus trace 21 of 28	113
Figure A.22	Consensus trace 22 of 28	114
Figure A.23	Consensus trace 23 of 28	115
Figure A.24	Consensus trace 24 of 28	116
Figure A.25	Consensus trace 25 of 28	117
Figure A.26	Consensus trace 26 of 28	118
Figure A.27	Consensus trace 27 of 28	119
Figure A.28	Consensus trace 28 of 28	120
Figure B.1	State trace 1 of 36	122
Figure B.2	State trace 2 of 36	123
Figure B.3	State trace 3 of 36	123
Figure B.4	State trace 4 of 36	123
Figure B.5	State trace 5 of 36	124
Figure B.6	State trace 6 of 36	124
Figure B.7	State trace 7 of 36	124

Figure B.8	State trace 8 of 36	125
Figure B.9	State trace 9 of 36	125
Figure B.10	State trace 10 of 36	125
Figure B.11	State trace 11 of 36	126
Figure B.12	State trace 12 of 36	126
Figure B.13	State trace 13 of 36	126
Figure B.14	State trace 14 of 36	127
Figure B.15	State trace 15 of 36	127
Figure B.16	State trace 16 of 36	127
Figure B.17	State trace 17 of 36	128
Figure B.18	State trace 18 of 36	128
Figure B.19	State trace 19 of 36	128
Figure B.20	State trace 20 of 36	129
Figure B.21	State trace 21 of 36	129
Figure B.22	State trace 22 of 36	129
Figure B.23	State trace 23 of 36	130
Figure B.24	State trace 24 of 36	130
Figure B.25	State trace 25 of 36	130
Figure B.26	State trace 26 of 36	131
Figure B.27	State trace 27 of 36	131
Figure B.28	State trace 28 of 36	131
Figure B.29	State trace 29 of 36	132
Figure B.30	State trace 30 of 36	132
Figure B.31	State trace 31 of 36	132

Figure B.32	State trace 32 of 36	133
Figure B.33	State trace 33 of 36	133
Figure B.34	State trace 34 of 36	133
Figure B.35	State trace 35 of 36	134
Figure B.36	State trace 36 of 36	134
Figure C.1	Block chain for UAV 10	136
Figure C.2	Block chain for UAV 28	137
Figure C.3	Block chain for UAV 32	138
Figure C.4	Block chain for UAV 120	139
Figure C.5	Block chain for UAV 122	140

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 4.1	Monterey Phoenix consensus code	71
Table 4.2	Monterey Phoenix state code	81

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ARSENL	Advanced Robotic Systems Engineering Laboratory
DLP	distributed ledger protocol
DLT	distributed ledger technology
DoD	Department of Defense
MASINT	measurement and signature intelligence
MP	Monterey Phoenix
NDS	National Defense Strategy
NPS	Naval Postgraduate School
SRWBR	short range wide band radio
TCP	Transmission Control Protocol
USN	U.S. Navy
UAS	unmanned aerial system
UAV	unmanned aerial vehicle
UCP	Uniform Chain Protocol
UDP	User Datagram Protocol
USG	United States government
UV	unmanned vehicle
UVS	unmanned vehicle system
XOR	Exclusive OR Operator

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

Distributed ledgers are viable options for businesses and governments to use as a way to verify the availability of data for post-mission analysis. In certain applications, the use of well-designed distributed ledgers has benefits that far outweigh the added infrastructure and resource costs associated with implementing them. A distributed ledger is a way of storing information that differs from traditional databases in that it is a database that is shared and synchronized across multiple sites and utilizes a consensus process [1]. Depending upon the application, the amount of data stored in a distributed ledger may be similar to, or much less than found in traditional databases [2].

1.1 Problem Statement

In distributed, multi-vehicle, autonomous systems, the most effective method of storing data is often dependent on the purpose of the system. In this project, we assume that individual vehicles observe and log notable events. Due to the nature of their operations, however, they may be unable to immediately share these logged events. In military operations, we must assume that there is a possibility that vehicles may be lost or destroyed. We also must assume that individual vehicles or groups of vehicles may experience periods of disconnected communication with other vehicles in the system. Collectively, this creates a requirement for dependable distributed log maintenance that is robust to vehicle losses and communications discontinuities. The focus of the research conducted in this project was to provide a solution for a few problems that distributed, autonomous, multi-vehicle systems may face in warfare. This research provides a solution that answers the following questions:

1. Can we create a distributed ledger protocol (DLP) that ensures data is distributed within the system, so that data generated by an individual vehicle survives its destruction by being stored on other vehicles in the system?
2. Can this DLP account for temporary discontinuity in the system's communications graph?
3. What can we prove about the correctness of this DLP?

4. Can we use a lightweight modeling system (e.g., Monterey Phoenix [3]) to facilitate the development process?

1.2 Motivation

To motivate this research and appreciate why it is significant, we examine the National Defense Strategy (NDS) for 2018. In the National Defense Strategy of 2018, the Department of Defense (DoD) prioritizes investment in the domain of cyberspace. The strategy states, “We will also invest in cyber defense, resilience, and the continued integration of cyber capabilities into the full spectrum of military operations” [4]. The DLP designed in this thesis project addresses mission event logging resilience. The DLP we created provides a solution that will allow unmanned autonomous systems operated by the DoD to maintain mission reconstruction resilience in the event that individual unmanned vehicles (UVs) are lost or destroyed.

1.3 Scope

The focus of the research presented here was the creation and verification of a DLP for UVs. The purpose of this protocol is to increase the likelihood that data in multi-vehicle, autonomous system survives the destruction or loss of a subset of the vehicles in the system. This thesis research served to complete two essential tasks. The first was to develop a DLP that can be used in distributed, multi-vehicle, autonomous systems to store logged events. The desired product for this task was a DLP that could be represented with both diagrams and pseudocode. The diagrams and pseudocode were necessary to facilitate the completion of the second task: verification that the DLP works as intended. This was accomplished by creating models with the Monterey Phoenix (MP) platform. In the second task pseudocode produced in the first task was translated into an abstracted behavioral model for use in MP. The behavioral model was used to verify the correctness of the DLP.

1.4 Research Methodology

To create the DLP for this project, an extensive literature review was conducted regarding distributed ledgers, block chains, MP, and distributed systems reaching consensus. After gaining background knowledge, Peter Pommer, my classmate at Naval Postgraduate School

(NPS), and I jointly developed a DLP that addresses the needs of a distributed, autonomous, multi-vehicles system. The protocol is documented using both diagrams and pseudocode representations. To verify the correctness of the DLP, two abstracted models were created so that they could be analyzed using Monterey Phoenix (MP). The reason that we had to use two different MP models was because we needed to examine different behaviors of the DLP. These abstracted models had two key requirements that had to be satisfied for MP to work properly. First, they had to directly map to the DLP described in Chapter 3, and, second, they had to provide the ability to prove the behavioral correctness of the features we designed into the DLP. After the abstracted models were created, they were converted to MP code. Once the MP code was completed for both models it was run on the MP platform. The results were analyzed to see if the DLPs behavior violated the intent of the DLPs design. Additionally, we conducted field tests at Camp Roberts to provide further verification of the behavioral correctness of the DLP designed for our unmanned aerial system (UAS). After verifying the DLP, we made conclusions and identified possible future work.

1.5 Organization

This thesis contains five more chapters. In Chapter 2, we will discuss the aforementioned background topics: distributed ledgers, block chains, Monterey Phoenix (MP), and distributed systems reaching consensus. In Chapter 3, we will present the detailed DLP that we created for this project. This presentation will include: requirements, a protocol summary, terminology, assumptions made, the block chain structure, a high level overview and the detailed protocol. In Chapter 4, we examine the work that was conducted with Monterey Phoenix (MP). This examination will include abstracted models, MP code, and behavior of the DLP in the MP models. In Chapter 5, we discuss how MP helped to refine the DLP we created. We also discuss results that were received from the field tests conducted at Camp Roberts. Finally, in Chapter 6 we will review the work completed for this thesis and provide direction for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:

Background

For this project, the goal is to utilize a distributed ledger to improve the resilience of unmanned vehicle systems (UVSs). To satisfy our goal, we had to create a DLP that increases availability of data in distributed multi-vehicle autonomous UVSs whose individual elements have a high risk of failure or destruction. In a DLP, integrity is typically the primary benefit of the ledger's development and implementation [5]. In contrast, availability of log data for post-mission analysis is the focus of this project. Therefore, it was imperative that the protocol be designed assuming a requirement for availability rather than integrity of log data. This chapter provides background information on topics that are critical to the protocol's design and verification. The topics to be covered include multi-vehicle UVSs, distributed ledger technology (DLT), block chain, the Paxos family of consensus protocols, and Monterey Phoenix.

2.1 Distributed Multi-Vehicle Autonomous Systems

To establish context, a clear definition for distributed, autonomous, multi-vehicle UVS is required. Once defined, we will examine the unique design challenges that accompany these types of systems.

2.1.1 Definition

Distributed, autonomous, multi-vehicle UVSs have three major design characteristics:

1. They are decentralized (distributed), which means that they do not have a central vehicle through which all data is passed. Due to the nature of their operations, the utilization of protocols that enable the distribution of data helps to ensure resiliency in the event that one or more vehicles is lost. Since the system is distributed, the vehicles in the system must communicate with each other to share log and event data.
2. They have autonomy (autonomous) and situational awareness at the individual vehicle level. Each vehicle collects and interprets its own data, determines its own courses of action, and communicates with other vehicles as required. In this context, vehicles are

- required to communicate with other vehicles to share generated log and event data.
3. Multiple vehicles operate simultaneously within the system. These vehicles can be watercraft, aircraft, spacecraft, or landcraft. The fact that numerous vehicles exist in the system can create specific challenges and benefits. This project addresses the following possible challenges: vehicle destruction, lossy or noisy communication, and disconnection from the UVS.

When designing software protocols for these systems, it is mandatory to keep the aforementioned characteristics at the core of the design process.

2.1.2 Project Background

For this project, the target system is a swarm of unmanned aerial vehicles (UAVs). These aerial vehicles are not controlled by a human operator and they are expected to be exposed to conditions that may lead to one or more UAVs being lost or destroyed. In this project, we assume that the UAS are not be able to communicate with the base while engaged in a mission. Given that these UAVs are exposed to the aforementioned conditions, the UAS needs to be resilient enough to ensure that data survives the destruction of individual members of the swarm. Furthermore, the target UAS will often be expected to engage in periods of disjoint operation. This means that some or all UAVs are called upon to detach from the swarm and operate independently or in smaller groups. Afterwards, the separated UAVs will re-assimilate back into the swarm from which they detached. Hence, this type of system is expected to experience disruptions due to packet loss and disconnection. A requirement of this project is to define the specific process by which a UAVs assimilates back into the swarm. Additionally, a swarm is expected to conduct operations that experience high vehicle loss. That means that UAVs may crash or be shot down during a mission. This type of scenario is a requirement of the research project and is accounted for by the protocol that was designed.

2.2 Distributed Ledger Technologies

2.2.1 Purpose of Distributed Ledgers

First created in the late 1970s, DLTs are a form of data storage technology that are specifically designed to facilitate the storage of pertinent information in a decentralized and duplicative

manner [6]. Decentralized and duplicative storage prevents data from being destroyed or modified easily. In fact, the system would require that the data be destroyed or modified on all the members of the distributed system for it to be successfully removed or modified in the system. In contrast, information in a centralized system can be destroyed or modified by targeting the central node. Due to their ability to decentralize information, DLT can provide great benefit in the form of increased availability of the data stored in the system, which will become apparent in Chapters 3 and 4. Generally, distributed ledgers exhibit some or all of the following characteristics: transparency, distribution, anonymity, and immutability [7]. Since this research project focuses on availability, the property of distribution is the only characteristic for which background information is necessary. Transparency, anonymity and immutability are not considered, as the benefits they may provide are out of the scope of this work.

2.2.2 Characteristics of a Distributed Ledger

A core feature of DLT is the concept of distribution of information across the system. The idea is clearly identified as a core concept by the use of the word “distributed” in the name “distributed ledger technologies.” In the context of DLT design, the idea of distribution means that all nodes in a distributed ledger store all of the information contained in that ledger. This method of storing data helps to ensure that the property of availability is maintained. By distributing stored data over multiple ledger nodes, a malicious actor would need to destroy ledger data on all of the nodes to successfully deny availability of the data. In Figure 2.1, the availability provided by a distributed system is illustrated. In this scenario, one node in the system has been destroyed and the other four nodes in the system are intact. Since all five nodes keep a copy of all pertinent information stored in the system, the information survives the destruction of location number four. In contrast, Figure 2.2 depicts a scenario where information is stored in a centralized manner. When Location 1 is destroyed, all information contained in Location 1 perishes.

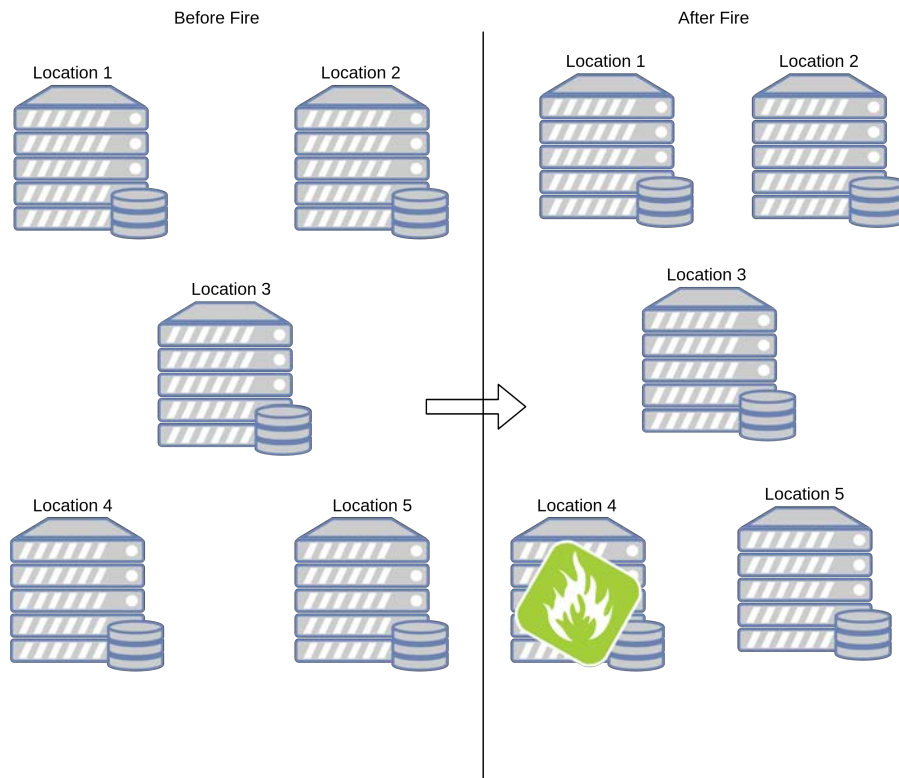


Figure 2.1. In the depicted distributed ledger scenario, one of the storage locations is destroyed, but since the data is replicated in the other 4 locations it survives.

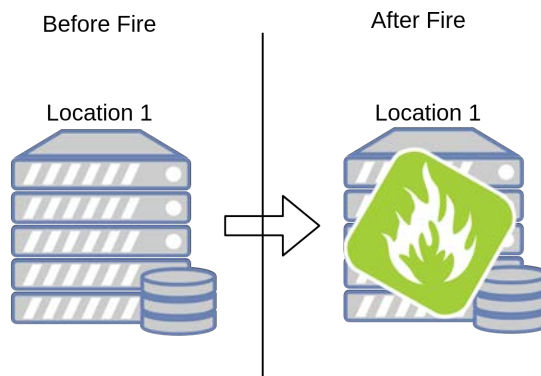


Figure 2.2. With a centralized ledger system, if a single point of storage were to be destroyed, all data would be lost.

2.3 Block Chain

Perhaps the most well-known DLT form, a block chain is a chain of blocks containing data that are linked in an agreed upon sequence [8]. Recently, block chain technologies have been associated with cryptocurrencies like Bitcoin, but practical applications extend beyond finance and currencies [9].

In a typical block chain, the chain structure is used to prevent the removal or modification of data in the chain. For this project, reordering of data is necessary in some cases, so facilitating the storage and reordering of data becomes the primary objective. In block chains, the chain structure is designed to prevent the modification of data by cryptographically linking a block to the block directly preceding it. The linking of a block to its preceding block is accomplished by hashing the combination of the preceding block's chain hash and the current block's block hash to generate the current block's chain hash. By storing blocks in a linked chain, no block in the chain can be removed or modified without altering the hashes of all subsequent blocks on the chain. We depict the concept of linking in Figure 2.3, which is explained as follows:

- **Simple Chain** - This chain is an unaltered block chain. It contains four blocks where each block is linked to the previous block by incorporation of the previous block's chain hash in the block hash. Block 3 is linked to Block 2, Block 2 is linked to Block 1, and Block 1 is linked to Block 0. This chain will be used for comparison to the other two chains in the figure.
- **Reordering of Block 1 and Block 2** - In this example, the order of Block 1 and Block 2 is changed. When comparing the resulting chain to the Simple Chain we can see that the chain hashes for Blocks 1, 2, and 3 are different while the chain hash for Block 0 is not. Since the order was changed starting at Block 1, all subsequent chain hashes for the two chains will be different.
- **Removal of Block 1** - In this scenario, Block one has been removed from the chain. When comparing this chain to the Simple Chain, we can see that the chain hashes are different for Blocks 2 and 3 but unaffected for Block 0. The removal of a block evidently alters all subsequent chain hashes in the block chain.

The examples in Figure 2.3 demonstrate two fundamental characteristics of block chains:

1. If blocks are reordered, the chain hash for the reordered blocks and all subsequent blocks is altered.
2. If a block is removed, the chain hash for all subsequent blocks is altered.

Reordering of Block 1 and Block 2	Simple Chain	Removal of Block 1
Chain_Hash_3= MD-5(Chain_Hash_1+Block_Hash_3)= 6e352e956da9bae94056c03c02555ef0 Block_Hash_3=MD-5(3)= eccbc87e4b5ce2fe28308fd9f2a7baf3 Data=3 Block 3	Chain_Hash_3= MD-5(Chain_Hash_2+Block_Hash_3)= e920a42c3b11d352fc2cc1abdc9618bc Block_Hash_3=MD-5(3)= eccbc87e4b5ce2fe28308fd9f2a7baf3 Data=3 Block 3	
Chain_Hash_1= MD-5(Chain_Hash_2+Block_Hash_1)= ea121d39d96ba26d89b8496997db373b Block_Hash_1=MD-5(1)= c4ca4238a0b923820dcc509a6f75849b Data= 1 Block 1	Chain_Hash_2= MD-5(Chain_Hash_1+Block_Hash_2)= 9f5073cd035fa6c76f0fe39b776fdc3b Block_Hash_2=MD-5(2)= 26ab0db90d72e28ad0ba1e22ee510510 Data = 2 Block 2	Chain_Hash_3= MD-5(Chain_Hash_2+Block_Hash_3)= a6b98c20fa001a4e9b96c0be15358624 Block_Hash_3=MD-5(3)= eccbc87e4b5ce2fe28308fd9f2a7baf3 Data=3 Block 3
Chain_Hash_2= MD-5(Chain_Hash_0+Block_Hash_2)= 956377f3672f87c66260093016d7c704 Block_Hash_2=MD-5(5)= e4da3b7fbce2345d7772b0674a318d5 Data = 5 Block 2	Chain_Hash_1= MD-5(Chain_Hash_0+Block_Hash_1)= c8d0f2b0726d9fa95e5269f8d2b07a22 Block_Hash_1=MD-5(1)= c4ca4238a0b923820dcc509a6f75849b Data= 1 Block 1	Chain_Hash_2= MD-5(Chain_Hash_0+Block_Hash_2)= 51817615867ea74dc80838eff713d023 Block_Hash_2=MD-5(2)= 26ab0db90d72e28ad0ba1e22ee510510 Data = 2 Block 2
Chain_Hash_0=MD-5(Block_Hash_0) MD-5(cfcd208495d565ef66e7dff9f98764da)= dcfcd07e645d245babe887e5e2daa016 Block_Hash_0=MD-5(0)= cfcd208495d565ef66e7dff9f98764da Data = 0 Block 0	Chain_Hash_0=MD-5(Block_Hash_0) MD-5(cfcd208495d565ef66e7dff9f98764da)= dcfcd07e645d245babe887e5e2daa016 Block_Hash_0=MD-5(0)= cfcd208495d565ef66e7dff9f98764da Data = 0 Block 0	Chain_Hash_0=MD-5(Block_Hash_0) MD-5(cfcd208495d565ef66e7dff9f98764da)= dcfcd07e645d245babe887e5e2daa016 Block_Hash_0=MD-5(0)= cfcd208495d565ef66e7dff9f98764da Data = 0 Block 0

Figure 2.3. This example illustrate how block chains prevent undetected modification of or deletion of data using the MD-5 hashing algorithm [10] to generate block and chain hashes

2.4 Consensus

Consensus is defined as “the judgment arrived at by most of those concerned” [11]. We define our consensus mechanism as the method by which the newest block chain is accepted by the majority of computers online. For this project, establishing consensus becomes a challenging problem, as it is impossible to guarantee that nodes in the system will always be available to participate in the consensus process. Other block chain technologies, like Bitcoin, prune data from their chains to gain consensus [9]. That type of consensus will not work in our system, however. Here, we cannot assume that the majority of nodes are connected to the system, so we based our DLP on a consensus protocol called Paxos and proof-of-trust validation.

2.4.1 Proof-of-Trust

Proof-of-trust can mean multiple things in the context of distributed systems. Therefore, to add clarity for this project, it is defined as follows: Proof-of-Trust means that for a node to participate in the consensus process, it must be a trusted node. Therefore, the node must be able to demonstrate that it is a trusted node. For this research project the demonstration is based upon authentication material being loaded onto the node prior to its deployment on a mission. This can be achieved through the use of encrypted and signed messages. For the sake of this project, there is an assumption that proof-of-trust in the system will be implemented by the underlying UVS rather than within the DLP. The UVS will use secure communications and keys for the ledger protocol will be installed on each UV prior to the mission.

2.4.2 Paxos

Paxos is a consensus protocol that enables distributed systems that experience network message losses to eventually reach consensus [12]. The protocol does not guarantee the property of unanimity, which means it cannot guarantee total consensus at any given time. Rather, it ensures that the system makes progress towards consensus as time goes by [12]. Given Paxos’ complexity, we will break it down into its three basic components, and those components will be described in more detail. These three components are choosing a value, learning a chosen value, and progress [12].

Before describing the components, we must note the three roles that nodes in a Paxos-like system perform. First, there is the role of *proposer*. This role involves sending requests to add information to the distributed system's chain of information. The second role is that of the *acceptor*. *Acceptors* as a group determine whether or not to accept the proposals broadcast by the *proposers*. Last, the *learners* are those that hear accepted proposals and commit that information to their systems. In most Paxos implementations, all nodes perform multiple roles simultaneously [12].

The first step of the Paxos algorithm is choosing a value. For Paxos, the value is chosen in two separate phases. In the first phase a *proposer* generates a proposal request with an associated number, which we will call n . This n is a unique number that can only be used once. After generating the proposal request, the *proposer* then broadcasts its request to the *acceptors*. If the *acceptor* has not received a proposal request with a number greater than n , then it will respond to the request with an approval. If the *acceptor* has received a request with a number greater than n , then it will not approve. If a value is approved by a majority of *acceptors*, then the proposer sends an accept request to each of those *acceptors*. Once an *acceptor* receives this request, the proposal is accepted, so long as the *acceptor* has not agreed to accept a proposal with a higher n .

The second step of Paxos is learning a chosen value. To learn a chosen value, the proposer has to know that the majority of *acceptors* have accepted the proposed value. This can be implemented in numerous ways and is not strictly defined in the Paxos algorithm. The core idea behind this step is that all nodes in the distributed system must have a method of discovering what values have been chosen for commitment to the system.

The progress step of Paxos means that, as time continues, the distributed system makes progress towards consensus. This does not guarantee that consensus will be reached at any given time, but rather that progress will not stop being made.

Finally, it is important to recognize that Paxos is a family of algorithms. In this project, we base our algorithm on Paxos, but we must also make protocol-specific adjustments to assure that Paxos works on our target system. In particular, Paxos algorithms are adept at solving problems in systems that have network reliability problems, but they were not necessarily developed to solve problems with UASs. In this project, we will craft our implementation of the Paxos algorithms to address the lack of communications associated with disjoint

operations, and to address the possibility that numerous nodes (i.e., small groups of nodes) in the system may be permanently lost or destroyed.

2.5 Monterey Phoenix

A core part of this project was to ensure the behavioral correctness of our DLP. To do this, we utilized a behavioral modeling and analysis tool called MP [3]. Dating back to the early 1990s, MP is a Navy-developed formal language and approach for modeling behaviors of systems, software, hardware, people, and organizations, and their dependencies on one another and the environment [3]. This tool was instrumental in providing verification of the behavioral correctness of our DLP. Behavioral modeling tools like MP are used to prove the correctness of software algorithms. To prove correctness, an algorithm is converted to the MP's grammar. Once converted, the algorithm is run through an `exhaustive trace`. The results from the trace can then be examined to identify unintended behavior. To provide background on MP, the following information is discussed: terminology, MP as a behavior modeling tool, MP functionality, and the types of Lightweight Proofs that can be provided by MP. A basic example of MP in use is also provided.

2.5.1 Terminology

This section introduces a number of `teletyped` terms. These terms are defined as follows:

- **Correctness** - A Boolean characterization of whether or not the execution results match the intent of the designer. [13]
- **Emergent Behavior** - Patterns of behavior that occur when an algorithm is run through an `exhaustive trace`.
- **Exhaustive trace** - A “brute force” trace methodology that looks at all possible scenarios.
- **Trace(s)** - Execution of one or more instances of an algorithm.

2.5.2 MP as a Behavior Modeling Tool

The home page of MP states that “MP is used to describe operational processes, business processes, and system or software architectures for behavior, and can support behavior

descriptions from a mission level down to a detailed design level” [14]. Stated differently, MP is intended to describe and identify emergent behavior in an algorithm.

For this project, the algorithm that was verified was our DLP. By modeling the behavior of the algorithm, we can perform two critical checks: identify logical bugs and demonstrate the correctness of the algorithm. Using MP allows us to create models of an algorithm and to use those models to identify unexpected behavior. These types of models can be useful at an early stage of system or software development to ensure that a flawed algorithm is not implemented. Additionally, design details can be driven by examples of unintended behavior discovered while modeling an algorithm.

2.5.3 Functionality

MP provides a lightweight proof by performing an exhaustive trace of a process [14]. Essentially, a set of rules are written to describe the essence of an algorithm at an abstract level. These rules are utilized to better model the behaviors and interactions between abstracted modules in the algorithm used to describe the system’s behavior. Due to its exhaustive trace methodology, MP must be appropriately abstracted to prevent the possibility of a combinatoric explosion leading to a situation where the resources necessary to build and analyze a model exceed the available hardware and software resources of the system on which it is running.

2.5.4 Lightweight Proofs

When attempting to conduct proofs with MP, it must be understood what it is possible to prove with MP. The following properties are true of MP [14]:

1. MP methodologies align with the assertion made in the *small scope hypothesis*, which states that most logical bugs can be demonstrated on small counterexamples. In the context of MP, small counterexamples means that MP does not have to trace every possible scenario to be effective. In fact, the number of loop iterations must be limited in cases where the loop may run an infinite number of times. MP limits iterations of loops to a scope of five.
2. MP conducts lightweight proofs that exhaustively trace all possible scenarios. Essentially, proof relies on the notion that if one identifies and analyzes all possible

scenarios and finds no unintended behavior, then it can be concluded that an algorithm works as intended.

3. MP is a useful tool for verifying that no easily apparent bugs exist in the algorithm being examined.
4. MP does not provide formal mathematical methods of proving an algorithm, so it cannot be definitively determined that a model is mathematically correct.
5. MP is not able to prove the correctness of the algorithm's implementation. Many implementation-specific details can cause an algorithm to work incorrectly, so an assumption must be made that an algorithm's implementation will not violate the designer's intent.

2.5.5 Example : Unreliable Message Flow

To provide clarity on how MP works, we examine an unreliable message flow example. This example can be found in the MP Github repository [15]. The two-step process for an unreliable network flow is:

1. Sender sends message.
2. Receiver either receives message or does not receive message.

In this example, the unreliable network problem has been abstracted and represented as a two step process. In this process, we assume that when a message is sent, there are no issues sending the message. The only possible thing that the sender can do is send a message. On the receiving end, we assume that one of two things can happen: the receiver can either receive the message or not receive the message. We are not looking at any details pertaining to the message contents as that would be too granular for the level of abstraction in this example.

The MP code for the unreliable message flow follows:

- **MP Code** - `SCHEMA unreliable_message_flow`
- **Description** - This line of code declares the MP schema with the name `unreliable_message_flow` and is the equivalent of giving the program a title.
- **MP Code** - `ROOT Sender: (* send *)`;
- **Description** - This statement declares a root-level event called "Sender". There is only

one instance of a particular root-level event in each trace. The asterisks surrounding send are what lets us know that the event can occur from zero to an infinite number of times.

- **MP Code** - ROOT Receiver: (* (receive | does_not_receive) *);
- **Description** - This statement declares a root-level event called “Receiver”. Since it is a root-level event, there is only one instance of “Receiver”. The associated events, receive and does_not_receive, can occur from zero to an infinite number of times.
- **MP Code** -

```
COORDINATE $x: send FROM Sender,  
$y: (receive | does_not_receive) FROM Receiver
```

- **Description** - These statements constitute a rule that forces the Sender to send a message before a Receiver can receive a message. The “COORDINATE” statement is used to set \$x to the send event from the Sender. The \$y is set to the receive or does not receive conditional.
- **MP Code** - DO ADD \$x PRECEDES \$y; OD;
- **Description** - This “DO ADD” statement is used to create a relationship between the \$x and \$y variables. The word “PRECEDES” makes the relationship so that \$x must always precede \$y.

2.6 Chapter Summary

To recap this chapter, the key concepts are summarized in this section. This chapter covered distributed, autonomous, multi-vehicle systems and why they are the target systems for this project. In this project, distributed ledgers provide the platform for increasing the resilience of UASs. The type of distributed ledger utilized for this thesis is based on block chain technology, which was described. For this project, the Paxos consensus protocol was selected because of its ability to reach consensus in networks that experience significant network losses. To complete the chapter, Monterey Phoenix (MP) was covered, as it provides a methodology for determining the behavioral correctness of the Uniform Chain Protocol (UCP). In Chapter 3, we present and describe the UCP.

CHAPTER 3:

Uniform Chain Protocol

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

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

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

3.1 Protocol Summary

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

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

3.2 Definitions and Terminology

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

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

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

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

- Block data
- Block hash
- Chain hash

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Assumptions

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

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

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

3.4 Block Chain Data Structure

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

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

3.4.1 Inception Block

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

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

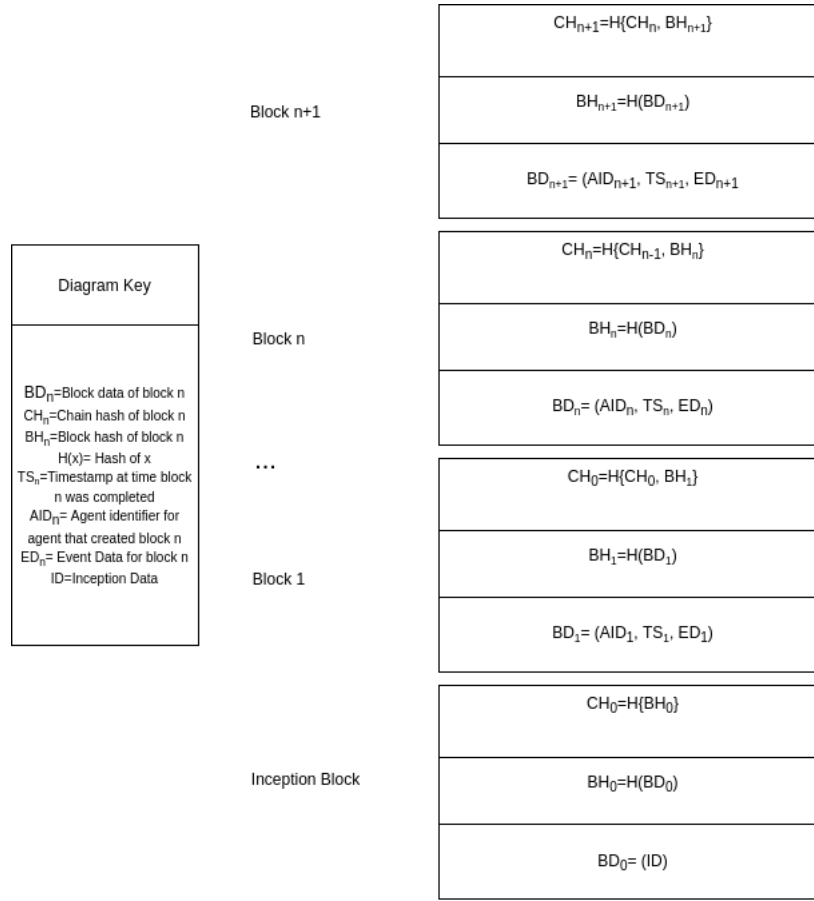


Figure 3.1. This figure is an illustration of the block chain data structure.

The inception block has the following components:

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

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

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

according to the following equation:

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

3.4.2 Subsequent Blocks

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

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

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

$$BD_n = (AID_n \parallel TS_n \parallel ED_n) \quad (3.5)$$

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

3.5 High-Level Overview

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

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

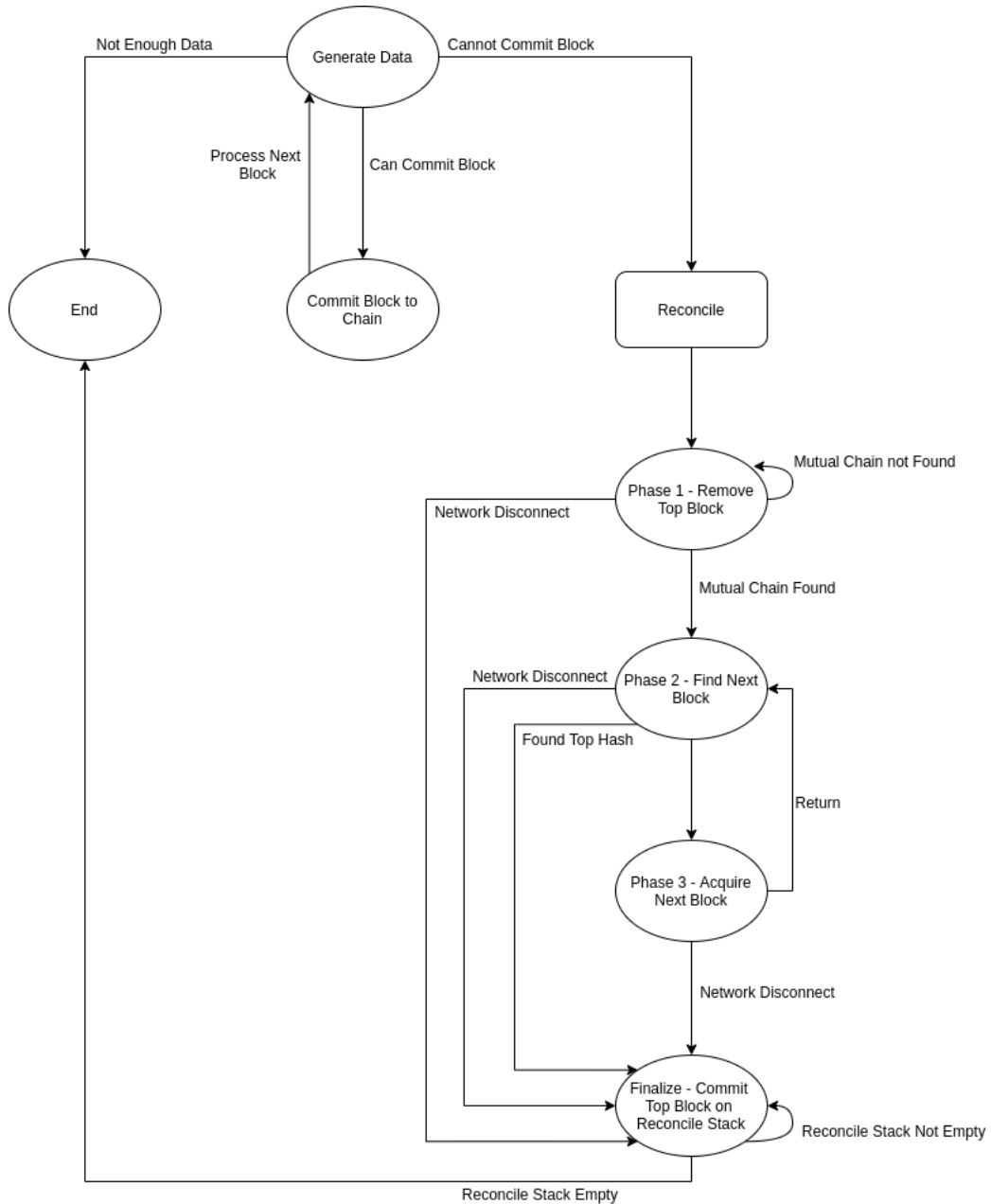


Figure 3.2. An abstracted version of UCP.

3.5.1 Generate a Block

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

3.5.2 Commit Block

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

3.5.3 Reconcile

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

Reconcile Phase 1

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

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

Reconcile Phase 2

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

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

Reconcile Phase 3

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

is conducted through the following steps:

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

Reconcile Finalize

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

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

3.6 Detailed Chain Protocol

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

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

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

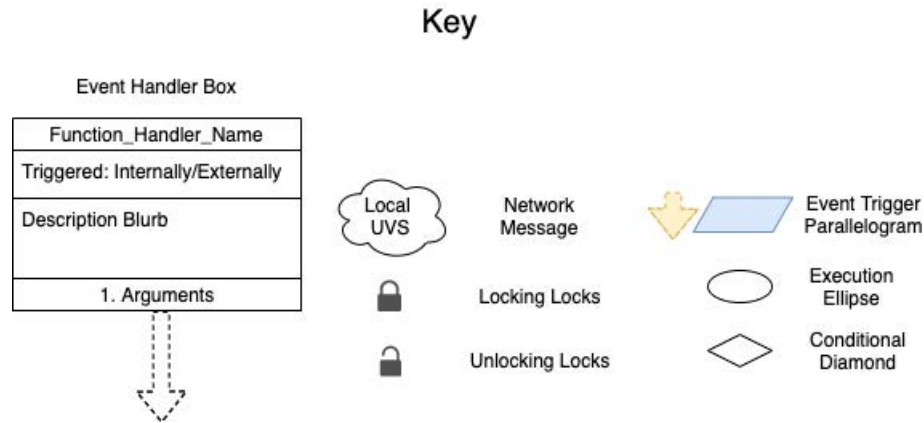


Figure 3.3. Event handler diagram and symbol key for use throughout this section.

Network Messages

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

3.6.1 Incorporation of Event Data into Blocks

Build Data Block Event

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

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

Algorithm 1 Build Data Block Event Handler

```

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

```

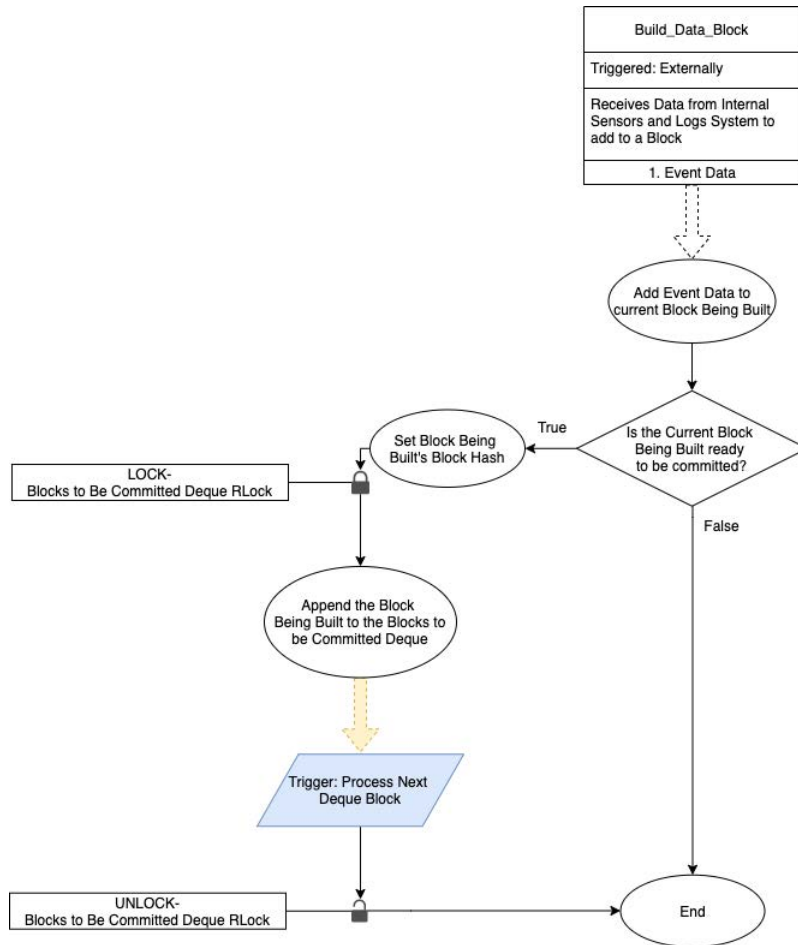


Figure 3.4. The **Build Data Block** event handler is invoked when the system generates loggable data for incorporation into the block chain.

3.6.2 The Commit Process

Process Next Deque Block Event

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

Algorithm 2 Process Next Deque Block Event Handler

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

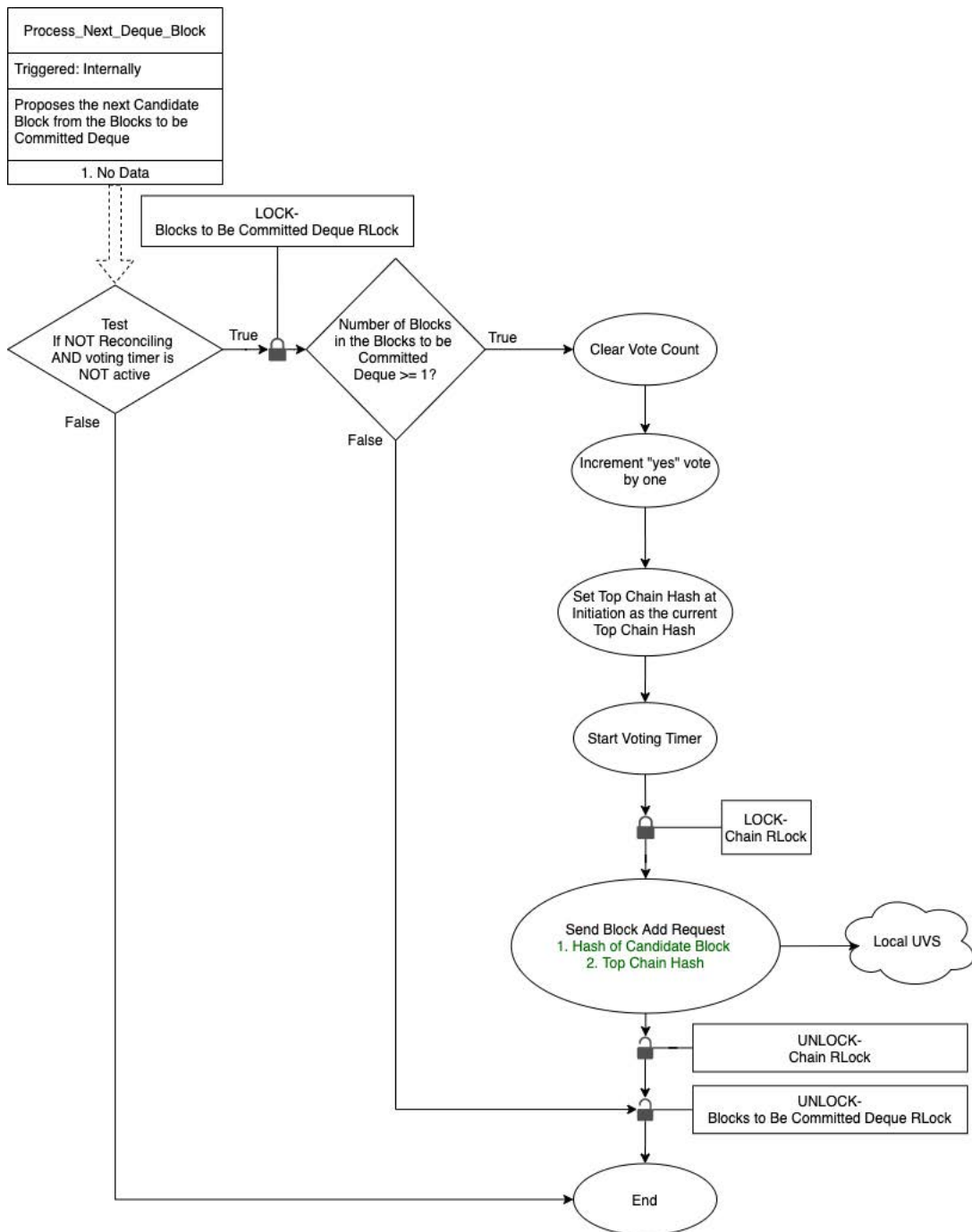


Figure 3.5. The **Process Next Deque Block** event handler is initiated internally by multiple event handlers to initiate the process by which a new block is proposed for addition to the block chain.

Receive Request Add Block Event

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

Algorithm 3 Receive Request Add Block Event Handler

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

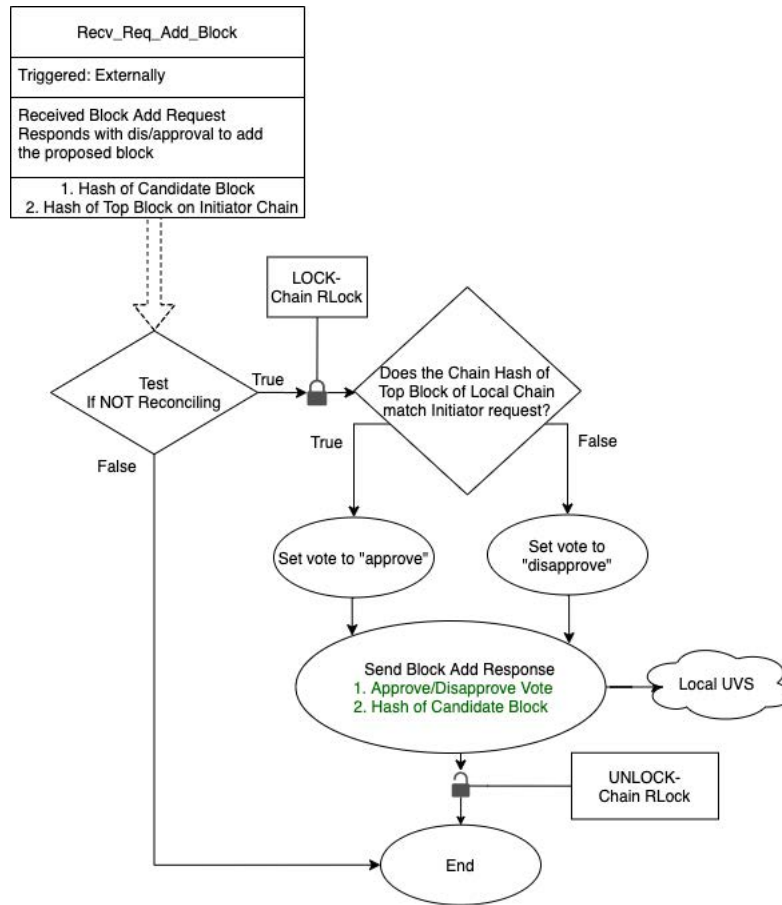


Figure 3.6. The **Receive Request Add Block** event handler is invoked upon receipt of a proposal from another agent to add a block to the block chain.

Receive Vote Add Block Event

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

Algorithm 4 Receive Vote Add Block Event Handler

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

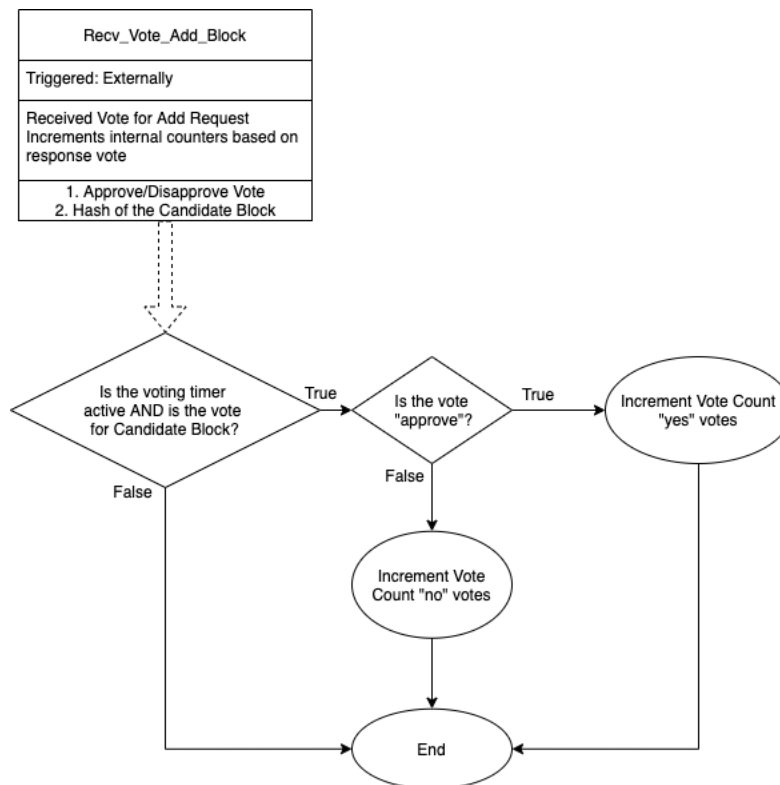


Figure 3.7. The **Receive Vote Add Block** event handler is triggered upon receipt of a vote for a previously proposed block addition and tallies “approve” and “disapprove” votes for later use.

Vote Timer Expiration Event

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

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

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

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

Algorithm 5 Vote Timer Expiration Event Handler

```

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

```

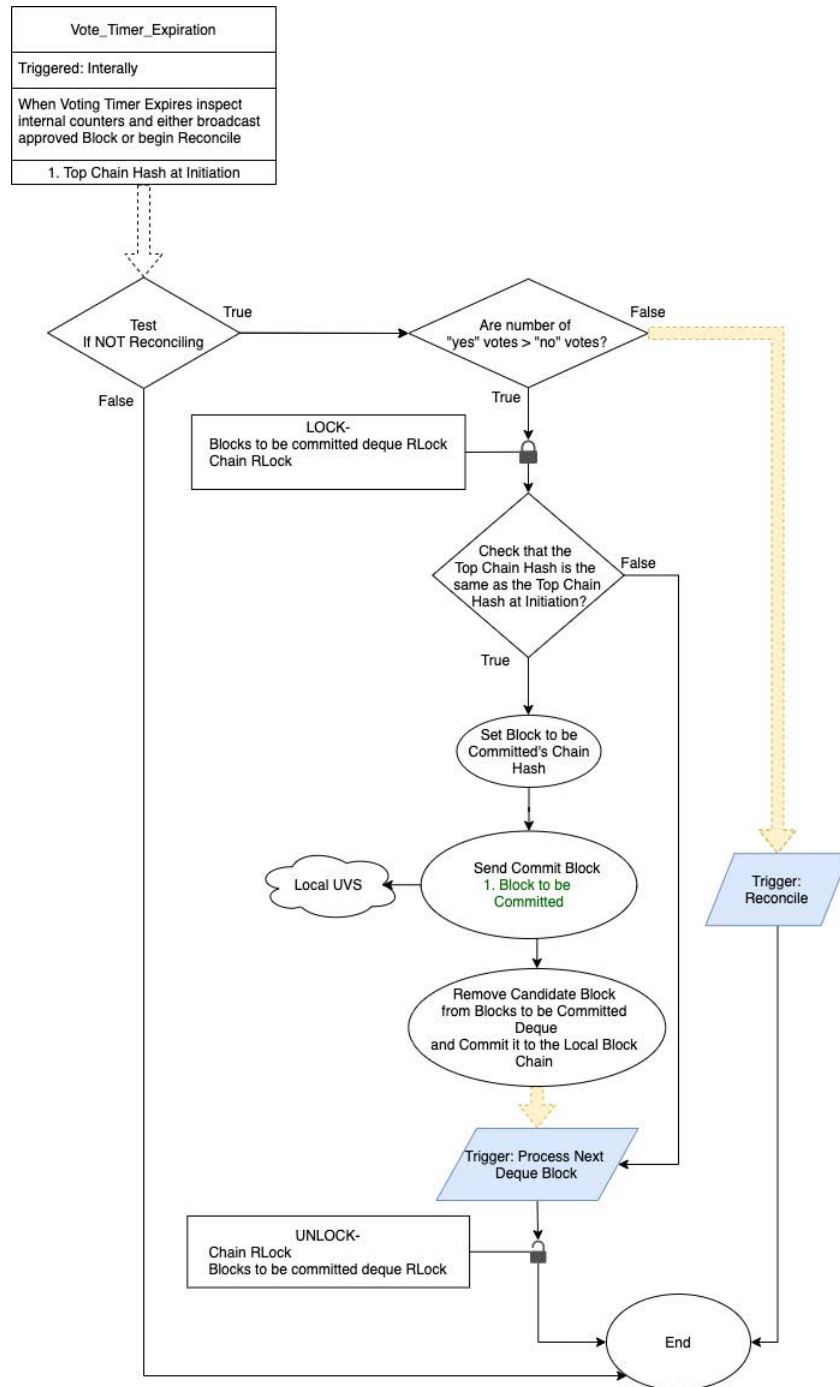


Figure 3.8. The **Vote Timer Expiration** event handler uses the results of the block add vote to determine whether to commit the proposed block to the block chain or trigger a **Reconcile Begin** event.

Receive Commit Block Event

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

Algorithm 6 Receive Commit Block Event Handler

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

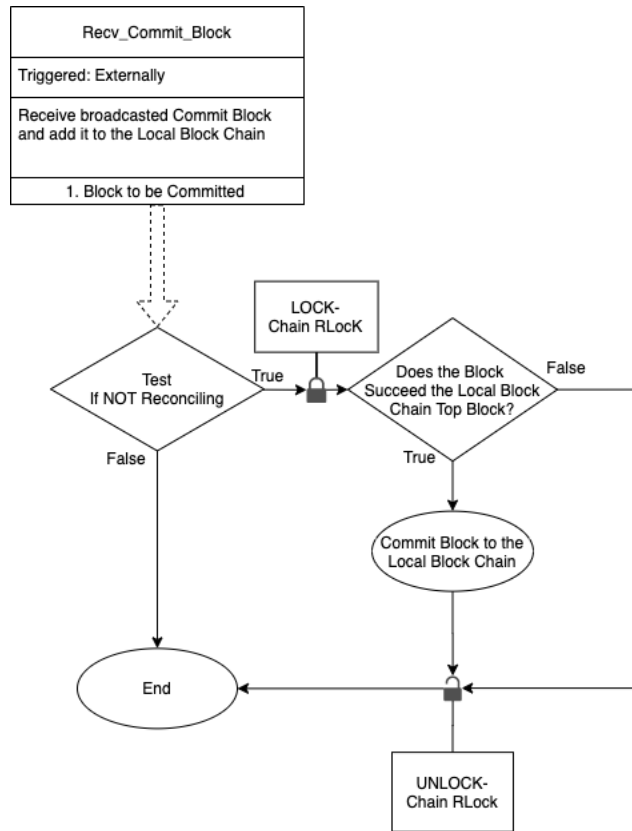


Figure 3.9. The **Receive Commit Block** event handler can be triggered by receipt of a block add message from another agent or locally by the **Vote Timer Expiration** or **Reconcile Finalize** event handlers. This event handler formally commits a block to the local block chain if appropriate.

3.6.3 Reconcile Process

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

Reconcile Begin Event

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

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

Algorithm 7 Reconcile Begin Event Handler

```

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

```

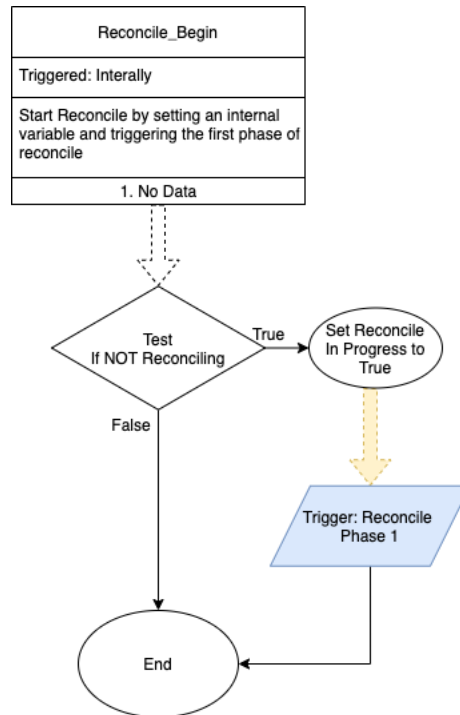


Figure 3.10. The **Reconcile Begin** event handler is triggered by the **Vote Timer Expiration** and **Receive Request Next in Sequence** events and initiates the reconcile process.

Reconcile Phase 1 Event

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

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

Algorithm 8 Reconcile Phase 1 Event Handler

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

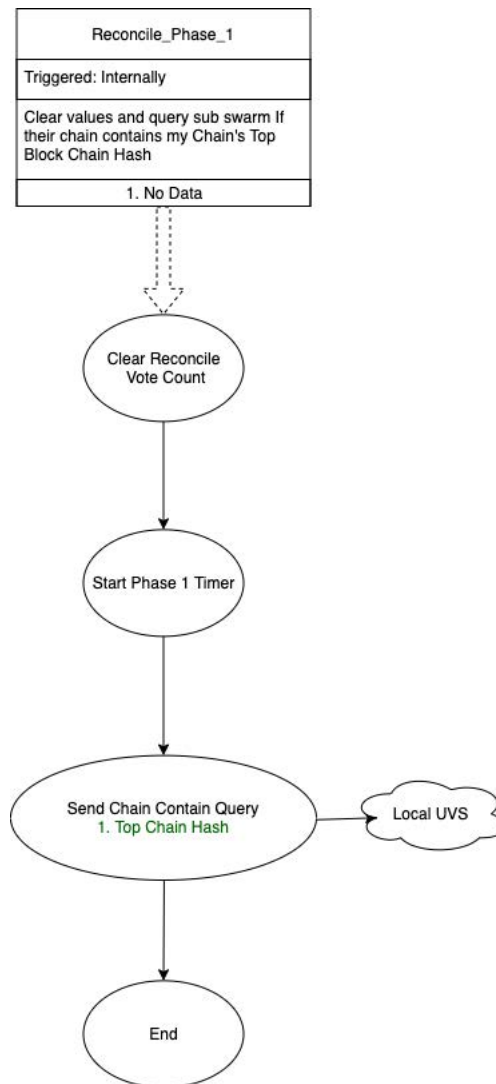


Figure 3.11. The **Reconcile Phase 1** event handler is triggered by the **Reconcile Begin** and **Reconcile Phase 1 Timer Expiration** event handlers and initiates the process of identifying a mutual chain upon which to build.

Receive Request Chain Contain Event

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

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

Algorithm 9 Receive Request Chain Contain Event Handler

```
if not reconcileInProgress then
    chainContainResponse.chainHash  $\leftarrow$  message.chainHash
    Obtain the chainRLock
    if localChain contains message.chainHash then
        chainContainResponse.vote  $\leftarrow$  “yes”
    else
        chainContainResponse.vote  $\leftarrow$  “no”
    end if
    Network send Chain Contain Response
    Release the chainRLock
end if
```

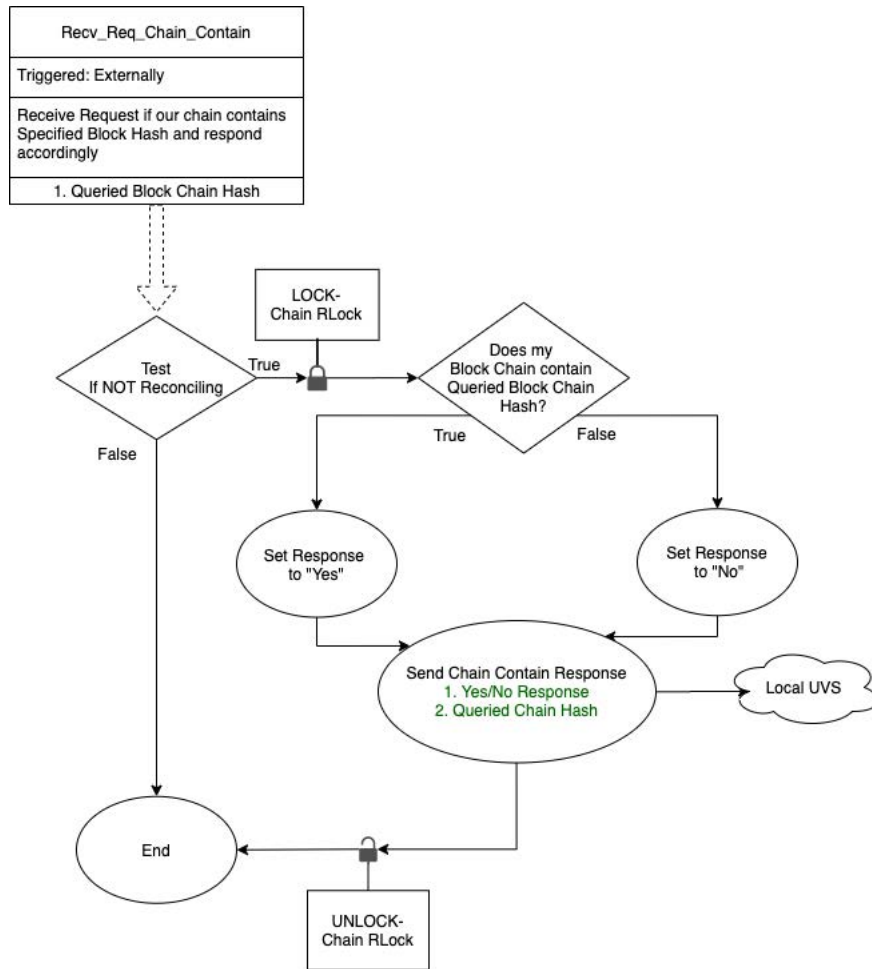


Figure 3.12. The **Receive Request Chain Contain** event handler is triggered by receipt of a message from another agent asking whether or not a particular chain hash is contained in the local block chain.

Receive Phase 1 Response Event

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

Algorithm 10 Receive Phase 1 Response Event Handler

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

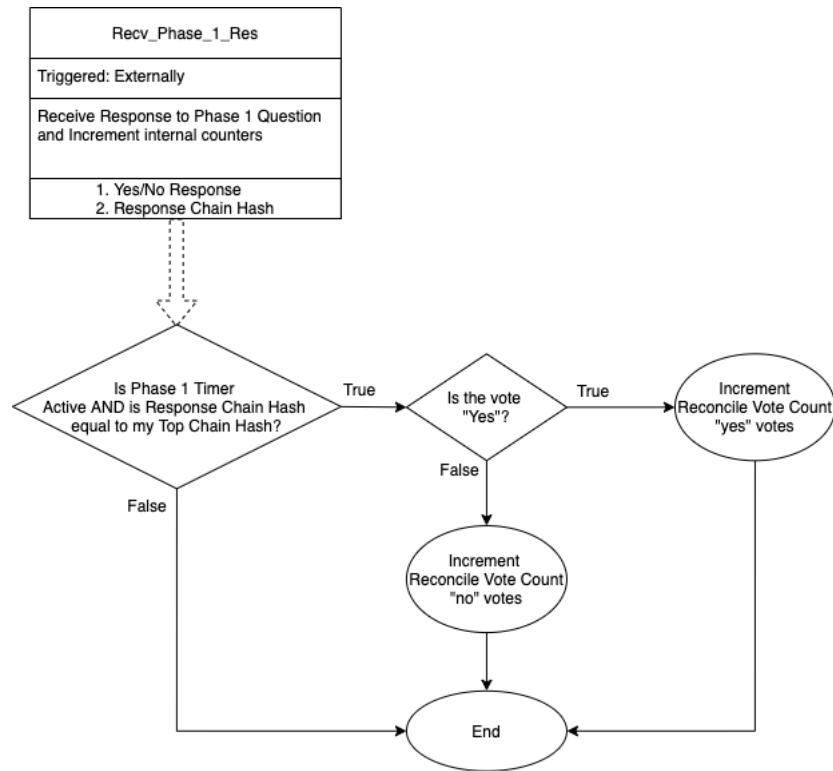


Figure 3.13. The Receive Phase 1 Response event handler is triggered by the receipt of a response to a Chain Contain Query message.

Reconcile Phase 1 Timer Expiration Event

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

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

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

Algorithm 11 Reconcile Phase 1 Timer Expiration Event Handler

```

if reconcileVoteCount["yes"] + reconcileVoteCount["no"] == 0 then
    Trigger Reconcile Finalize event
else if reconcileVoteCount["yes"] < reconcileVoteCount["no"] then
    temp ← localChain.pop()
    reconcileStack.push(temp)
    Trigger Reconcile Phase 1 event
else
    Trigger Reconcile Phase 2 event
end if

```

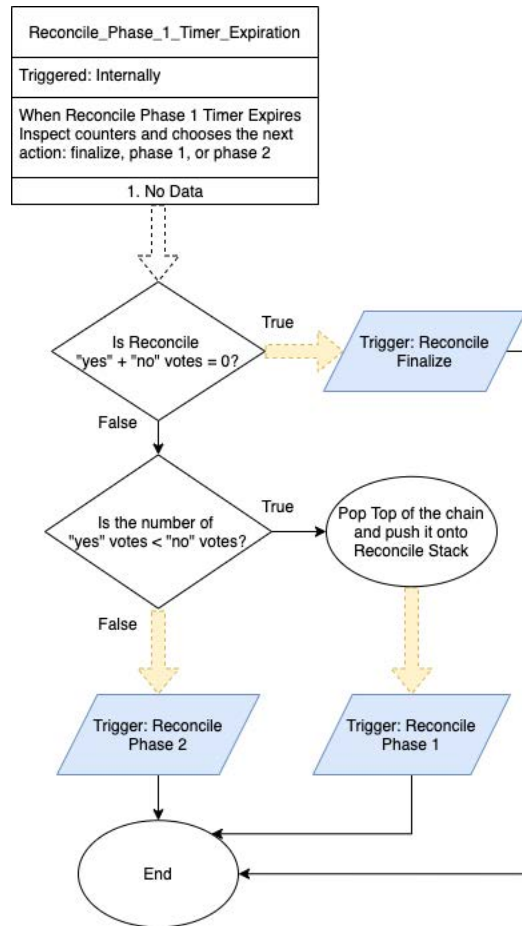


Figure 3.14. Upon expiration of the **Reconcile Phase 1 Timer**, responses are tallied to determine whether or not a **mutual chain** has been identified.

Reconcile Phase 2 Event

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

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

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

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

Algorithm 12 Reconcile Phase 2 Event Handler

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

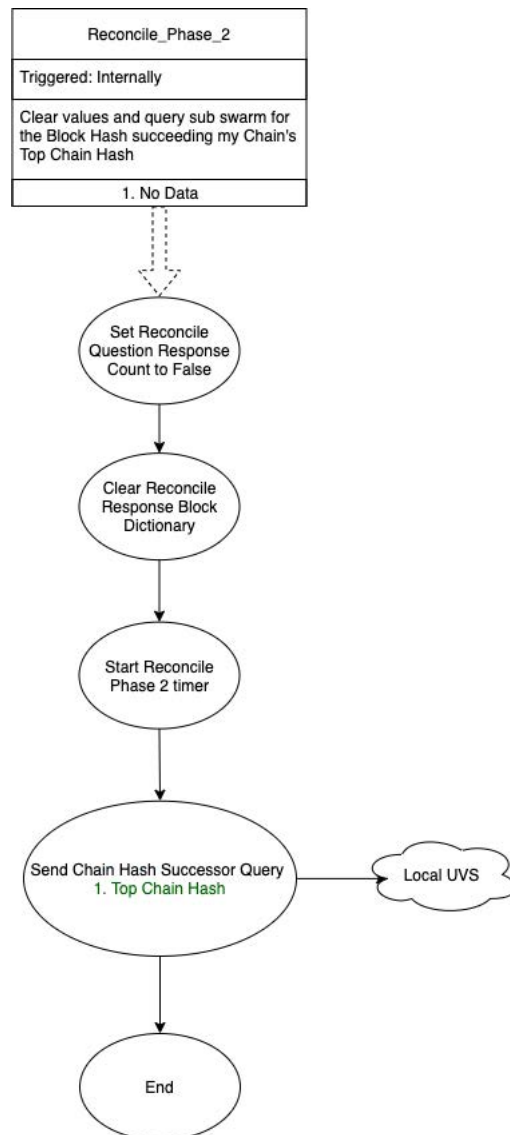


Figure 3.15. The **Reconcile Phase 2** event handler is initiated by the **Reconcile Phase 1 Timer Expiration** event handler to initiate the process of adding blocks to the mutual chain.

Receive Request Next in Sequence Event

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

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

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

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

Algorithm 13 Receive Request Next in Sequence Event Handler

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

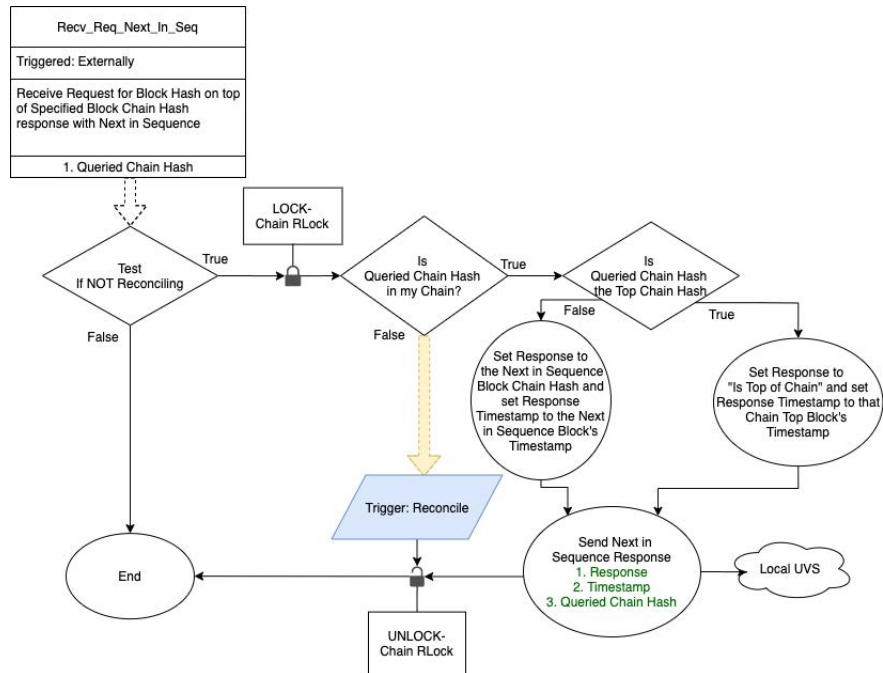


Figure 3.16. The **Receive Request Next in Sequence** event handler is triggered by receipt of a **chain hash** successor query from another agent. It responds to the query accordingly and initiates a local reconcile process as required.

Receive Response Next in Sequence Event Handler

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

Algorithm 14 Receive Response Next in Sequence Event Handler

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

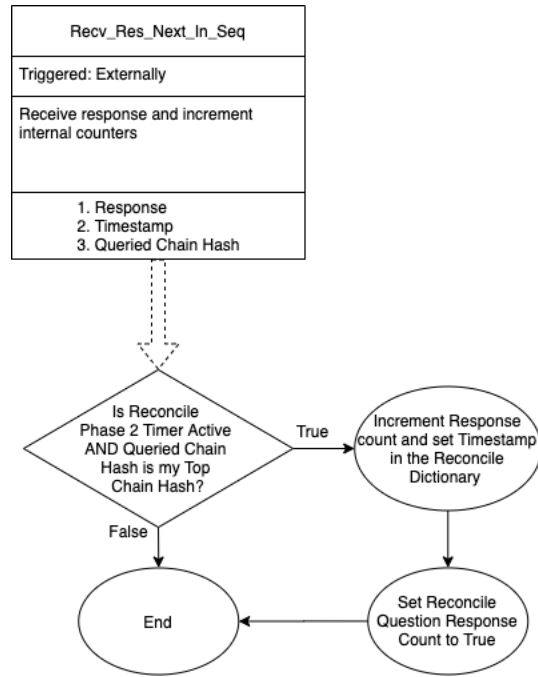


Figure 3.17. The **Receive Response Next in Sequence** event handler is triggered by receipt of a message responding to a previously transmitted chain hash successor query and maintains counters facilitating correct extension mutual chain.

Reconcile Phase 2 Timer Expiration Event

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

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

Algorithm 15 Reconcile Phase 2 Timer Expiration Event Handler

```

if reconcileResponseCount == False then
    Trigger Reconcile Finalize event
else
    if majorityResponse from reconcileResponseDictionary is a tie then
        majorityResponse ← blockHash with oldest timestamp
    else
        majorityResponse ← majorityResponse blockHash
    end if
    if majorityResponse == “Is Top of Chain” then
        Trigger Reconcile Finalize event
    else
        majorityResponseBlockHash ← majorityResponse
        Trigger Reconcile Phase 3 event
    end if
end if

```

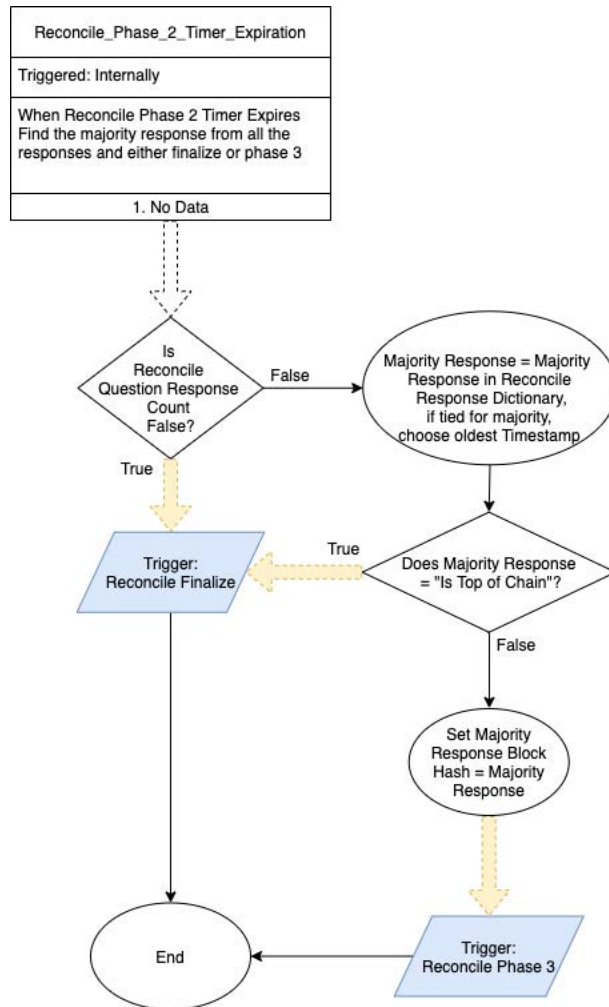


Figure 3.18. The **Reconcile Phase 2 Timer Expiration** event is triggered upon expiration of the phase 2 timer. It determines the block hash associated with the local majority mutual chain.

Reconcile Phase 3 Event

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

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

Algorithm 16 Reconcile Phase 3 Event Handler

```

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

```

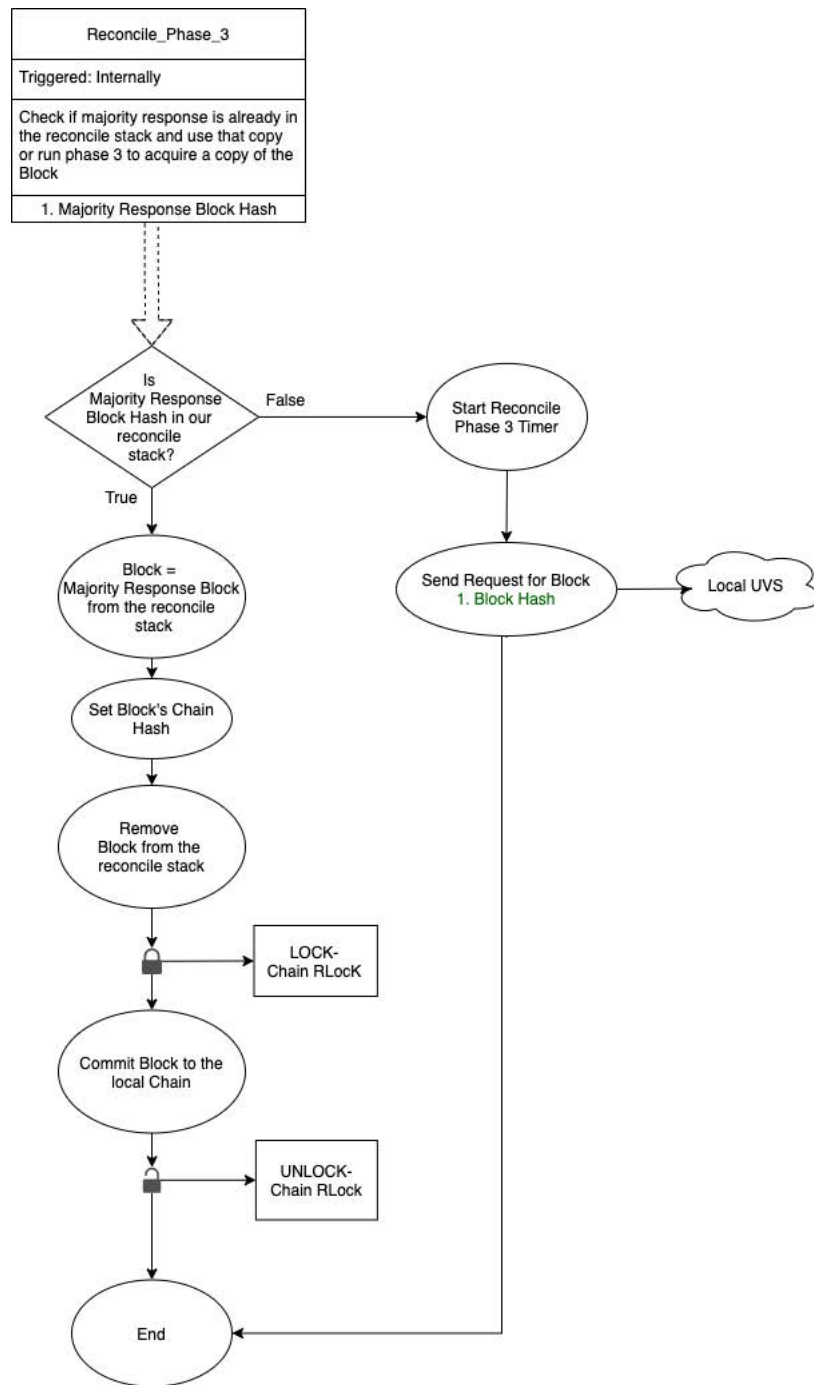


Figure 3.19. The **Reconcile Phase 3** event is triggered internally by the **Reconcile Phase 2 Timer Expiration** event handler and is responsible for initiating request for the block associated with the majority block hash.

Receive Request Block for Hash Event

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

Algorithm 17 Receive Request Block for Hash Event Handler

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

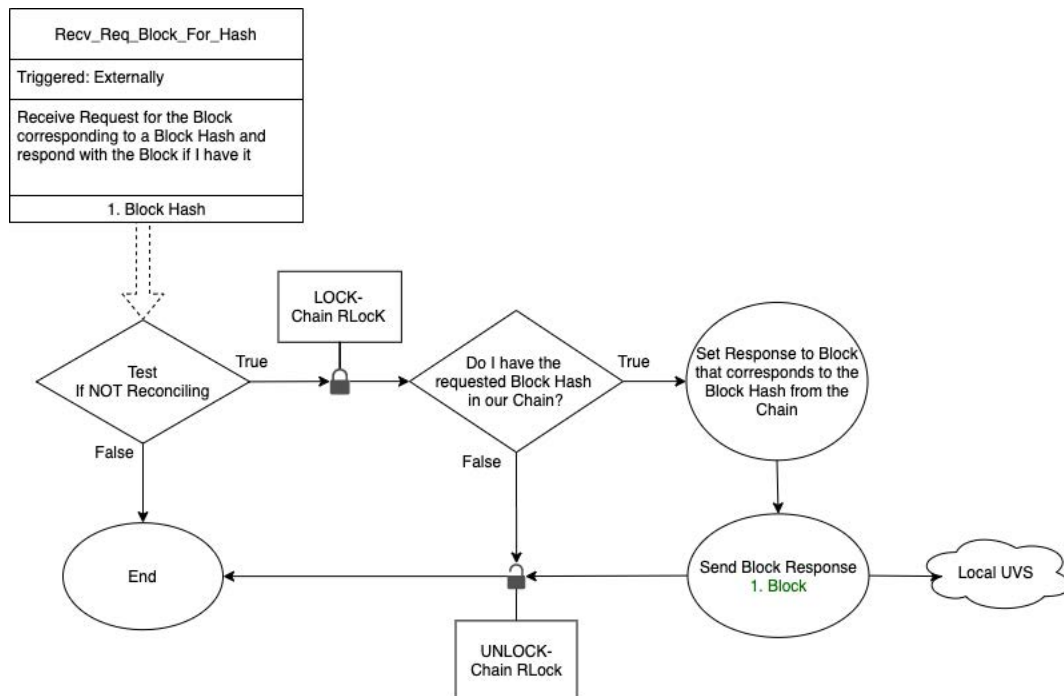


Figure 3.20. The **Receive Request Block for Hash** event handler is triggered by an external request for a block associated with a particular block hash. The agent sends block to the requesting agent if it is present in the local block chain.

Receive Response Block for Hash Event

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

Algorithm 18 Receive Response Block for Hash Event

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

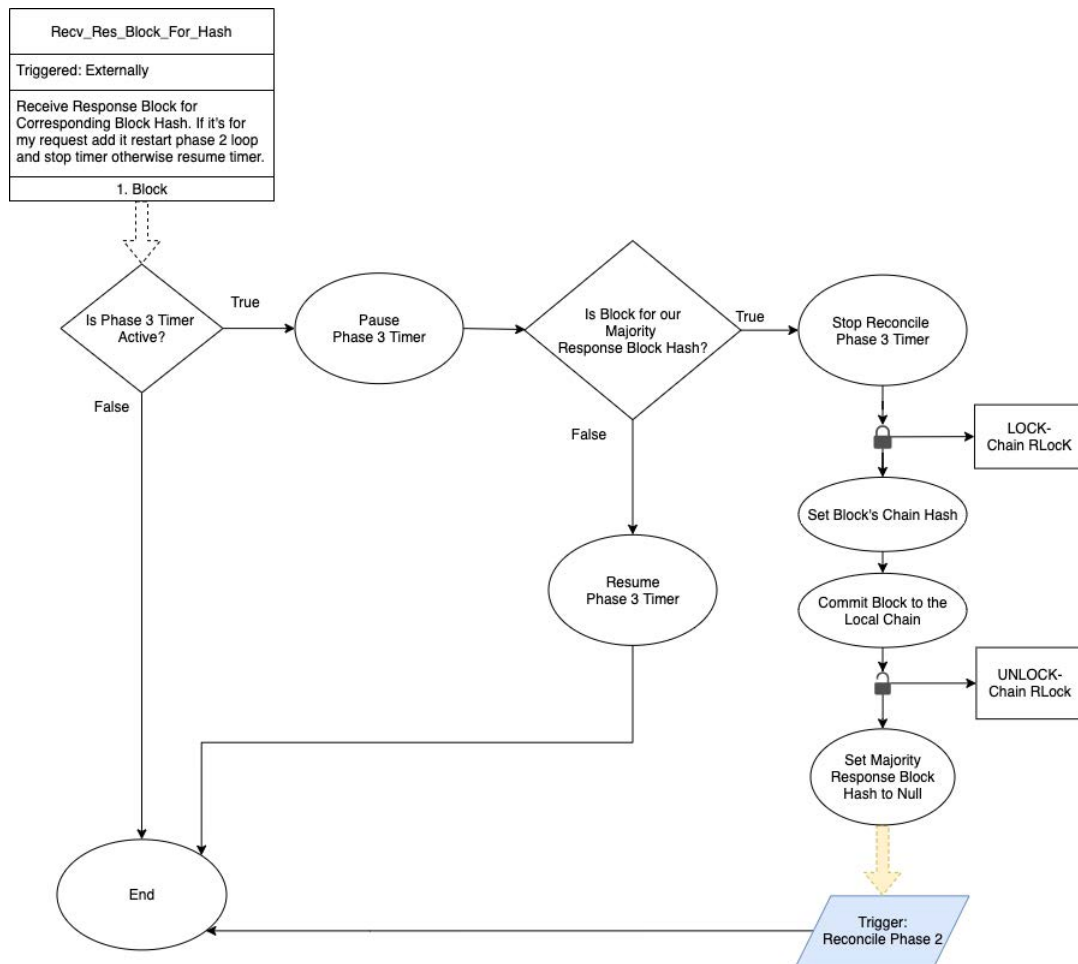


Figure 3.21. The **Receive Response Block for Hash** event handler is triggered by receipt of a response to a previously transmitted request for block. If the contained block is the one that was requested, it is added to the local block chain.

Reconcile Phase 3 Timer Expiration Event

The **Reconcile Phase 3 Timer Expiration** event is triggered upon expiration of the phase 3 timer that was started by the **Reconcile Phase 3** event handler. If a valid response is received, the phase 3 timer is terminated by the **Receive Response Block for Hash** event handler, so the phase 3 timer will only expire if no valid responses were received. This event handler, therefore, simply concludes the reconcile process by triggering a **Reconcile Finalize** event. The processing of this event is described by Algorithm 19 and is depicted graphically in Figure 3.22.

Algorithm 19 Reconcile Phase 3 Timer Expiration Event Handler

majorityResponseBlockHash $\leftarrow$ Null

Trigger **Reconcile Finalize** event

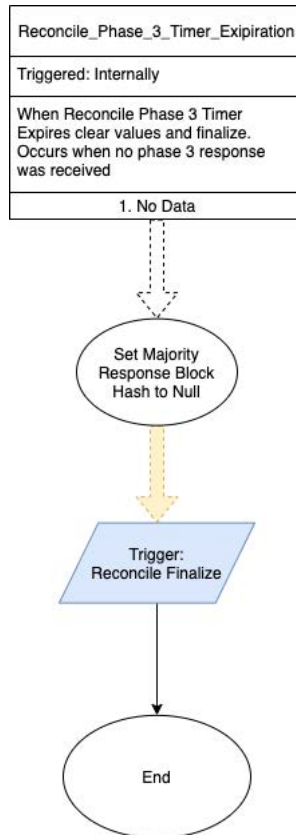


Figure 3.22. The **Reconcile Phase 3 Timer Expiration** event handler is only triggered when no valid responses are received to a request for block message. When this occurs, **Reconcile Phase 3** is terminated by triggering a **Reconcile Finalize** event.

Reconcile Finalize Event

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

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

Algorithm 20 Reconcile Finalize Event Handler

```

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

```

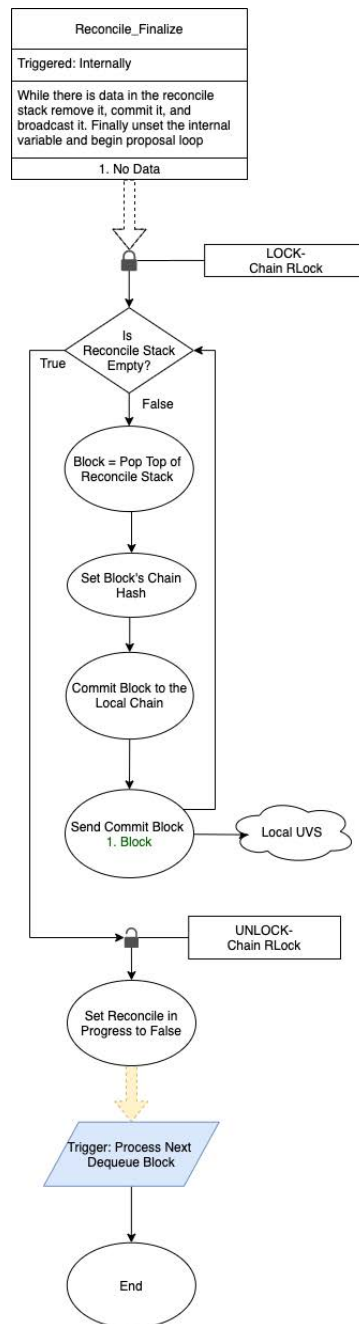


Figure 3.23. The **Reconcile Finalize** event handler can triggered by any of the reconcile phase timer expiration events or when the maximum local UVS mutual chain has been generated to recommit blocks remaining in the reconcile stack to the blockchain.

3.7 Chapter Summary

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

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

CHAPTER 4: Monterey Phoenix

In this Chapter, we describe how MP was used to verify the correctness of the UCP presented in Chapter 3. To verify that UCP is correct, we must prove the six protocol characteristics we asserted in Chapter 3 are true:

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

In this list, four of the six characteristics can be verified with MP. Characteristic 4 can be partially verified by MP, but not entirely verified. MP cannot verify that the underlying vehicle system is secure, so we cannot verify that externally generated invalid blocks do not exist in the system. However, MP can verify that blocks do not sporadically appear in the system. This is, however, a trivial verification that is demonstrated by the fact that all blocks in the system are created by the **Build Data Block** event described in Chapter 3.6.1. Correct implementation of this event handler, therefore, will preclude the generation of invalid blocks from within the system. Characteristic 1 relies on the specific implementation of the UCP, so it cannot be verified with MP. These two characteristics, however, must hold true

in the implementation for UCP to work as intended. To verify the other four characteristics, this chapter is divided into a consensus section and a state section. The consensus section will address the verification of Characteristics 3, 5, and 6. In the state section, Characteristic 2 will be verified.

For both sections, three parts are presented and discussed:

1. **Abstraction** - To scope the model and prevent exhaustion of resources, we represented the algorithm as an abstract version of UCP. Different abstractions used for the consensus and state sections. In both cases, the abstractions directly map to UCP.
2. **MP Code** - Code is used in both sections to convert the respective abstract algorithm to a model that MP can use to verify the algorithm.
3. **Discussion** - Finally, we discuss how the MP code verifies each of the characteristics identified.

4.1 Consensus

In this section, we verify the correctness of the consensus process. The goal of our MP model is to verify that the following statements, which were identified at the beginning of the chapter, are true:

- No more than one copy of a particular block will be maintained on a UV at any time. A block can exist within the locally-maintained block chain, within a “waiting to be committed” data structure, or within a data structure associated with the reconcile process.
- In a fully connected system, all blocks will eventually be committed to all locally maintained block chains, i.e., block chains maintained by each UV.
- In a fully connected system where all agents have had the chance to reconcile block chains with each other, one uniform block chain will emerge.

4.1.1 Abstraction

In this section, we break the abstraction process into components and behaviors. We first identify the component and then we identify the behaviors it exhibits that are modeled with MP.

Components

The MP model that demonstrates correctness of the protocol can be represented with five components and two data structures. The components are each abstractions of event handlers in the **Reconcile Process** (Subsection 3.6.3) of UCP:

- **Phase 1 Vote** - This component is an abstraction of the three event handlers in the first phase of reconcile. The three event handlers are **Reconcile Phase 1**, **Receive Phase 1 Response**, and **Reconcile Phase 1 Timer Expiration**. For the purpose of the model, these three event handlers have to work exactly as intended in UCP. The purpose of this component is to find a `mutual chain`.
- **Phase 2 Vote** - This component is an abstraction of the **Reconcile Phase 2**, **Receive Response Next in Sequence**, and **Reconcile Phase 2 Timer Expiration** event handlers. These three event handlers hold a vote that enables the protocol to build upon a `mutual chain`.
- **Check Reconcile Stack for Block** - This component is an abstraction of a check performed in the **Reconcile Phase 3** event handler. This check verifies that when we add a block to the chain that it is not contained in the reconcile stack.
- **Phase 3 Request** - This component requests for and adds the block that is selected upon in the **Phase 2 Vote** to the `block chain`. This component is an abstraction of **Reconcile Phase 3**, **Receive Response Block for Hash**, and the **Reconcile Phase 3 Timer Expiration** event handlers.
- **Reconcile Finalize** - This component is an abstraction of the **Reconcile Finalize** event handler.

The two data structures for this model are:

- **Reconcile Stack** - This is the `reconcile stack` data structure defined in Section 3.2.
- **Block Chain** - This is the `block chain` data structure defined in Section 3.2.

Behaviors

Now that we have identified all of the components in our MP consensus model, we can describe the behaviors of the abstract algorithm. The behaviors that the abstracted algorithm must exhibit to accurately represent UCP follow:

1. The abstracted algorithm starts with the **Phase 1 Vote**. This vote can have three behaviors that were modeled with MP: no response, no approval, or approval.
 - (a) If no response is received, the algorithm goes directly to **Reconcile Finalize**.
 - (b) If the vote is not for approval, then the following three things happen:
 - i. The top block is removed from the block chain.
 - ii. The top block is placed on the reconcile stack.
 - iii. **Phase 1 Vote** is conducted again.
 - (c) If the vote is for approval, the process moves to **Phase 2 Vote** because a mutual chain has been found.
2. The **Phase 2 Vote** is now conducted to build upon the mutual chain. This vote can result in three possible outcomes that were modeled with MP: no response, top hash found, or a block hash is received.
 - (a) If no response is received, the algorithm goes directly to **Reconcile Finalize**.
 - (b) If the majority response is that the top hash has been found, then the program goes to **Reconcile Finalize**.
 - (c) If the majority response is a block hash, then the program proceeds to **Check Reconcile Stack for Block**.
3. In the **Check Reconcile Stack for Block** component, the algorithm checks the reconcile stack for the block hash selected in the **Phase 2 Vote**. This check can result in two possible outcomes:
 - (a) If the block hash is present, the following behaviors are exhibited:
 - i. The block corresponding to the block hash is removed from the reconcile stack.
 - ii. That block corresponding to the block hash is added to the top of the block chain.
 - iii. The **Phase 2 Vote** is repeated.
 - (b) If the block hash is not present, the algorithm proceeds to the **Phase 3 Request**.
4. In the **Phase 3 Request**, the UV broadcasts a request for the UVS to send the block that corresponds to the block hash selected in the **Phase 2 Vote**. This request can result in two possible outcomes.
 - (a) If no response is received, the algorithm proceeds to **Reconcile Finalize**.
 - (b) If the block is received, the following happens:
 - i. The block is added to the top of the block chain.

- ii. The **Phase 2 Vote** is repeated.
- 5. If the **reconcile stack** is empty when the **Reconcile Finalize** component is called, then the consensus process has been completed. If it is not empty, the following behavior is exhibited:
 - (a) A block is popped off the top of the **reconcile stack** and committed to the chain. The algorithm proceeds to behavior 5.b.
 - (b) If the **reconcile stack** is empty, the algorithm proceeds to behavior 5.c. If the **reconcile stack** is not empty, repeat behavior 5.a.
 - (c) When the **reconcile stack** is empty, the consensus process is complete.
- 6. **Universal Behavior** - This is not behavior of a specific component, but instead addresses behavior that is true across all components in this abstracted model. The following behaviors are universal:
 - (a) All pathways through the algorithm will terminate after **Reconcile Finalize** and only after **Reconcile Finalize**.
 - (b) The program can only terminate if the **reconcile stack** is empty.
 - (c) The algorithm always starts at **Phase 1 Vote**.

The behavior identified in this section is depicted in Figure 4.1.

Consensus/Reconcile

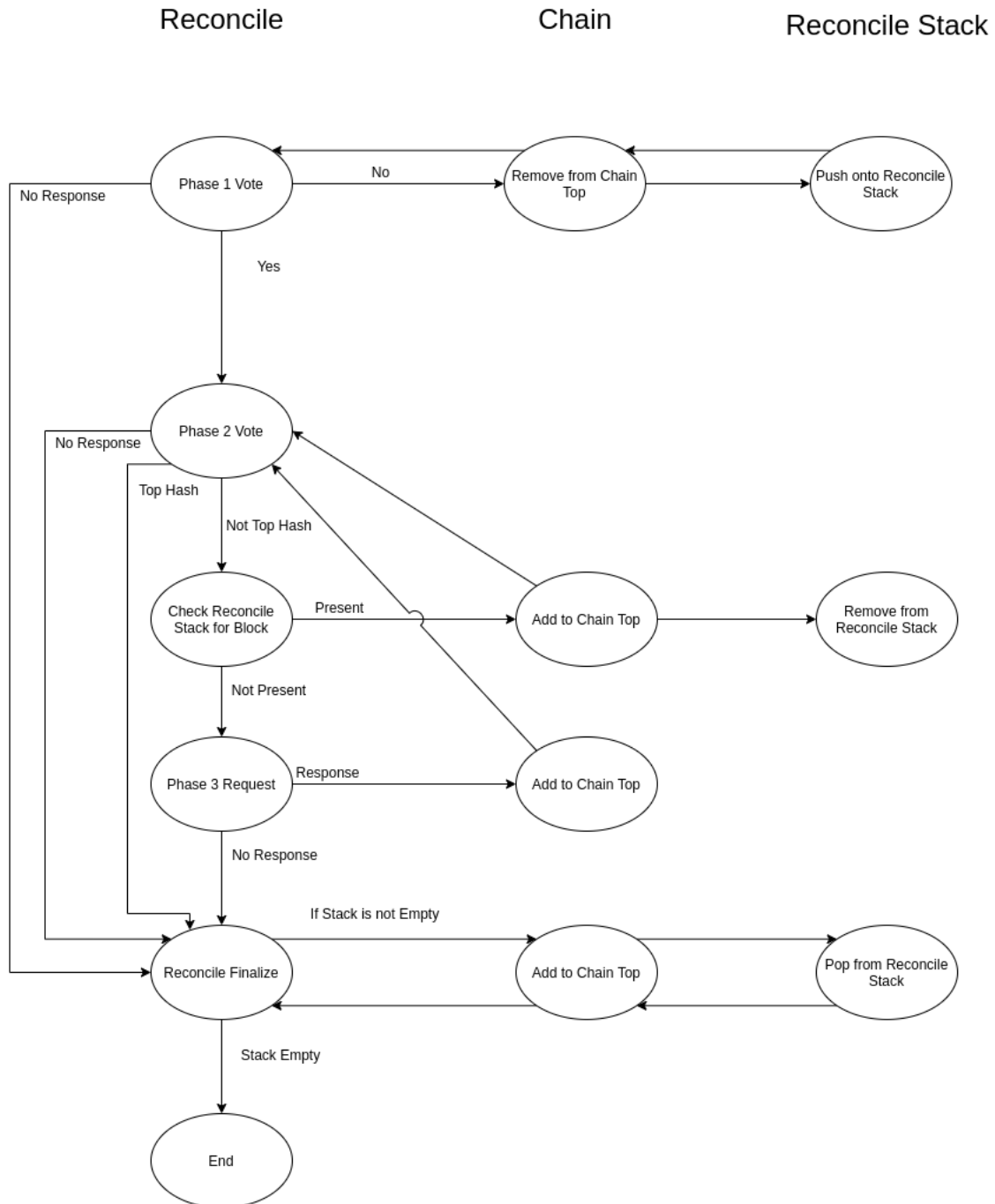


Figure 4.1. Abstracted version of the consensus process in UCP

4.1.2 Monterey Phoenix Code

Following abstraction of the UCPs consensus process, we can convert the abstracted algorithm into MP code. Below, we identify each segment of code as well as the behavior it models. In the MP code there are internal and external events with similar names. An example is `Add_to_Chain_Top` and `Add_to_Chain_Top_`. Although named similarly, they are different events. `Add_to_Chain_Top` is used for internally generated additions to the chain and `Add_to_Chain_Top_` is used for externally generated additions to the chain. This naming convention is used to distinguish other types of internal and external events. The code for the consensus process is provided in Table 4.1:

Table 4.1. Monterey Phoenix consensus model code for verification of the UCP reconciliation process.

Code Segment	MP Code	Behavior
1	SCHEMA CONSENSUS	None
2	ROOT RECONCILE:	None
3	(+ Phase_1_Vote +);	None
4	Phase_1_Vote: (No_Response Reconcile_Finalize No Yes (+ Phase_2_Vote +));	1.a - No response, proceed to Reconcile Finalize 1.b.iii - Mutual chain not reached, Repeat Phase 1 Vote 1.c - Found mutual chain, proceed to Phase 2 Vote 6.c - Always start with Phase 1 Vote

5	Phase_2_Vote: (No_Response Reconcile_Finalize Top_Hash Reconcile_Finalize Not_Top_Hash Check_Stack_for_Block);	2.a - No response, proceed to Reconcile Finalize 2.b - Top hash found, proceed to Reconcile Finalize 2.c - Received block hash, proceed to Check Reconcile Stack for Block
6	Check_Stack_for_Block: (Present Not_Present Phase_3_Request)	3.a.iii - Block hash found in reconcile stack, repeat Phase 2 Vote 3.b - Block hash not found in reconcile stack, proceed to Phase 3 Request
7	Phase_3_Request: (Response No_Response Reconcile_Finalize)	4.a - No response, proceed to Reconcile Finalize 4.b.ii - Block received, repeat Phase 2 Vote
8	Reconcile_Finalize: (* Stack_not_Empty *) Stack_Empty End;	5.c - Reconcile stack empty, consensus complete

9	<pre> ENSURE #NO_Response + #Top_Hash==1; ENSURE #Stack_Empty==1; ENSURE FOREACH \$e:End #\$\$EVENT AFTER \$e ==0; </pre>	6.a - Always terminate after Reconcile Finalize
10	<pre> ROOT BCHAIN: (* Remove_from_Chain_Top *) (* Add_to_Chain_Top *) (* Add_to_Chain_Top_ *); </pre>	None
11	<pre> COORDINATE \$a:No FROM RECONCILE, \$b:Remove_from_Chain_Top FROM BCHAIN DO ADD \$a PRECEDES \$b; OD; </pre>	1.b.i - Mutual chain not reached, remove top block from block chain
12	<pre> COORDINATE \$a:Present FROM RECONCILE, \$b:Add_to_Chain_Top FROM BCHAIN DO ADD \$a PRECEDES \$b; OD; </pre>	3.a.ii - Block hash found in reconcile stack, add block to block chain

13	COORDINATE \$a:Response FROM RECONCILE, \$b:Add_to_Chain_Top_ FROM BCHAIN DO ADD \$a PRECEDES \$b; OD;	4.b.i - Block is received, add block to block chain
14	ENSURE #Pop_from_Stack + #Remove_from_Stack== #Push_onto_Stack; ENSURE #Reconcile_Finalize==1;	3.a.i - Block hash found in reconcile stack, remove block from reconcile stack 6.a - Always terminate after Reconcile Finalize 6.b - Only terminate if reconcile stack is empty
15	ROOT STACK: (* Push_onto_Stack *) (* Remove_from_Stack *) (* Pop_from_Stack *);	None
16	COORDINATE \$a:Push_onto_Stack FROM STACK, \$b:Remove_from_Chain_Top_ FROM BCHAIN DO ADD \$b PRECEDES \$a; OD;	1.b.ii - Mutual chain not reached, placetop block onto reconcile stack

17	COORDINATE \$a:Remove_from_Stack FROM STACK, \$b:Add_to_Chain_Top_ FROM BCHAIN DO ADD \$b PRECEDES \$a; OD;	3.a.ii - Block hash found in reconcile stack, add block to block chain
18	COORDINATE \$a:Stack_not_Empty FROM RECONCILE, \$b:Pop_from_Stack FROM STACK DO ADD \$a PRECEDES \$b; OD;	5.a - Block popped from reconcile stack 5.b - If reconcile stack not empty, repeat 5.a 5.c - Reconcile stack is empty, consensus complete

4.1.3 Discussion

In this section we analyze the results that were received when we compiled the model. The full results from the compiled MP model are provided in Appendix A.3. Here we describe how the results from the MP model satisfy the three statements we made at the beginning of Section 4.1:

1. **Statement** - No more than one copy of a particular block will be maintained on a UV at any time. A block can exist within the locally-maintained block chain, within a “waiting to be committed” data structure, or within a data structure associated with the reconcile process.

Analysis - To satisfy this statement, we must establish that the program does not store a block in the chain more than once. In the analysis of the results provided by MP, the model always checked to see the block being added to the chain was in the reconcile stack. This check provides assurance that a block does not exist in the block chain more than once. To further prove this statement, we will examine the

three structures that a block can be committed to the block chain from:

- (a) **Reconcile Stack** - The only way that a block can get on the reconcile stack is from the block chain. If two copies of a block do not exist in the block chain, then two copies cannot be put on the reconcile stack.
- (b) **Local UVS** - If a second copy of a block comes from the Local UVS, then the check we identified prevents it from being saved to the block chain twice.
- (c) **Blocks to be committed deque** - This structure is reserved only for new blocks. By its nature, we cannot have a block committed from this structure twice as it only performs the initial commit of a block.

By conducting this check, we have shown the algorithm prohibits all three vectors in which a block may be added twice. Therefore, the statement is satisfied.

2. **Statement** - In a fully connected system, all blocks will eventually be committed to all locally maintained block chains (i.e., the block chains maintained by each UV).

Analysis - To satisfy this statement, we must establish that the program does not stop running until all blocks are in the chain. In the analysis of the results provided by MP, the program only terminated with an empty reconcile stack. The fact that the program only terminated with an empty reconcile stack satisfies this statement.

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

Analysis - For this statement, fully connected cases exclude results where the model traced the path of No_Response. By definition, No_Response means that the program is not fully connected. To satisfy this statement, we must establish that in fully connected cases the following two-part sequence of events occurred:

- (a) The program iterated through **Phase 1 Vote** until it found a mutual chain.
- (b) The program iterated through a combination of **Phase 2 Vote, Check Reconcile Stack for Block**, and **Phase 3 Request** until it found the top hash.

In the analysis of the results provided by MP, the fully connected cases always completed the first part and then moved onto and completed the second part. The completion of both parts, therefore, satisfies this statement.

4.2 State

In this section, we verify the correctness of UCP's state. The goal of our MP model is to verify that the following statement is true:

1. If a UV's protocol event handlers are in an idle state, then all of its completed blocks must be committed to its local block chain.

4.2.1 Abstraction

For this section, we divide the abstraction process into components and behaviors. We first identify the component and then we identify the behaviors they exhibit that are modeled with MP.

Components

The MP model that demonstrates correctness of the protocol can be represented with five internal components and three external components. These components are abstractions of event handlers in the UCP. The internal components are:

1. **Block Ready** - This component is an abstraction of the **Build Data Block** event handler in Section 3.6.1. This abstraction only represents the case where a full block has been built and is ready to be committed to the block chain.
2. **Hold Votes and Determine Approval** - This component is an abstraction of the **Process Next Deque Block**, and **Receive Vote Add Block** event handlers in Section 3.6.2. This component also represents a partial abstraction of the **Vote Timer Expiration** event handler. The rest of the abstraction is in the next two components.
3. **Verify Chain** - This component represents another part of the abstraction for the **Vote Timer Expiration** event handler. This component is the abstraction of the part of the handler where the chain is verified to be unaltered before a block is committed.
4. **Commit Block to Chain** - This component is the last part of the abstraction for the **Vote Timer Expiration** event handler. This component represents the part of the event handler where a block that has been approved is committed to the block chain.
5. **Reconcile** - This component is an abstraction of the **Reconcile Process** in Section 3.6.3. This component represents the entire **Reconcile Process** and all event

handlers contained within it.

The external events are as follows:

1. **Sufficient Data Generated** - This component is an abstraction of the input provided by the system running UCP. This component is utilized to represent scenarios where enough data has been generated to build a full **block**.
2. **Receive Message to Commit Block** - This component is an abstraction of the **Receive Commit Block** event handler in Section 3.6.2.
3. **Hear Hash on Top of Hash not in Chain** - This component is an abstraction of the **Receive Request Next in Sequence** event handler in Section 3.6.2. This component is used to represent behavior that occurs when a vehicle hears a request for a **block** on top of a **block** that it does not have.

Behaviors

Now that we have identified all of the components required in our MP model, we can describe the behaviors of the abstract algorithm. The behaviors that the abstracted algorithm must exhibit to represent UCP are:

1. When **Sufficient Data Generated** is generated, the algorithm proceeds to call the **Block Ready** component.
2. When **Receive Message to Commit Block** is generated, the algorithm will behave in one of two ways:
 - (a) If the vehicle can commit the block, then the algorithm proceeds to the **Commit Block to Chain** component.
 - (b) If the vehicle cannot commit the block, then the algorithm ignores the request.
3. When the **Hear Hash on Top of Hash not in Chain** event is generated, the algorithm will behave in one of two ways:
 - (a) If the algorithm is in a reconciling state, then the algorithm ignores the event.
 - (b) If the algorithm is not in a reconciling state, then the algorithm proceeds to the **Reconcile** component.
4. The **Block Ready** component leads to one of two possible behaviors:
 - (a) If the algorithm is in a reconciling state, then the algorithm terminates.
 - (b) If the algorithm is not in a reconciling state, then the algorithm proceeds to **Hold**

Votes and Determine Approval.

5. The **Hold Votes and Determine Approval** component determines whether or not a block can be added to the block chain. The behaviors corresponding to this event are:
 - (a) If the vote does not lead to approval, then the algorithm proceeds to the **Reconcile** component.
 - (b) If the vote does lead to approval, then the algorithm proceeds to the **Verify Chain** component.
6. The **Verify Chain** component determines whether or not the chain has changed since the vote was initiated. This can lead to the following outcomes:
 - (a) If the chain is not verified (i.e., it has changed), then the algorithm proceeds to the **Hold Votes and Determine Approval** component.
 - (b) If the chain is verified (i.e., it is unchanged), then the algorithm proceeds to the **Receive Message to Commit Block** component.
7. When the **Commit Block to Chain** component commits a block, the process enters an idle state while it waits for the next externally generated event.
8. When the **Reconcile** component is called, the algorithm proceeds to complete the **Reconcile Process** and then waits for another externally generated event.

The behavior identified in this section is depicted in figure 4.2.

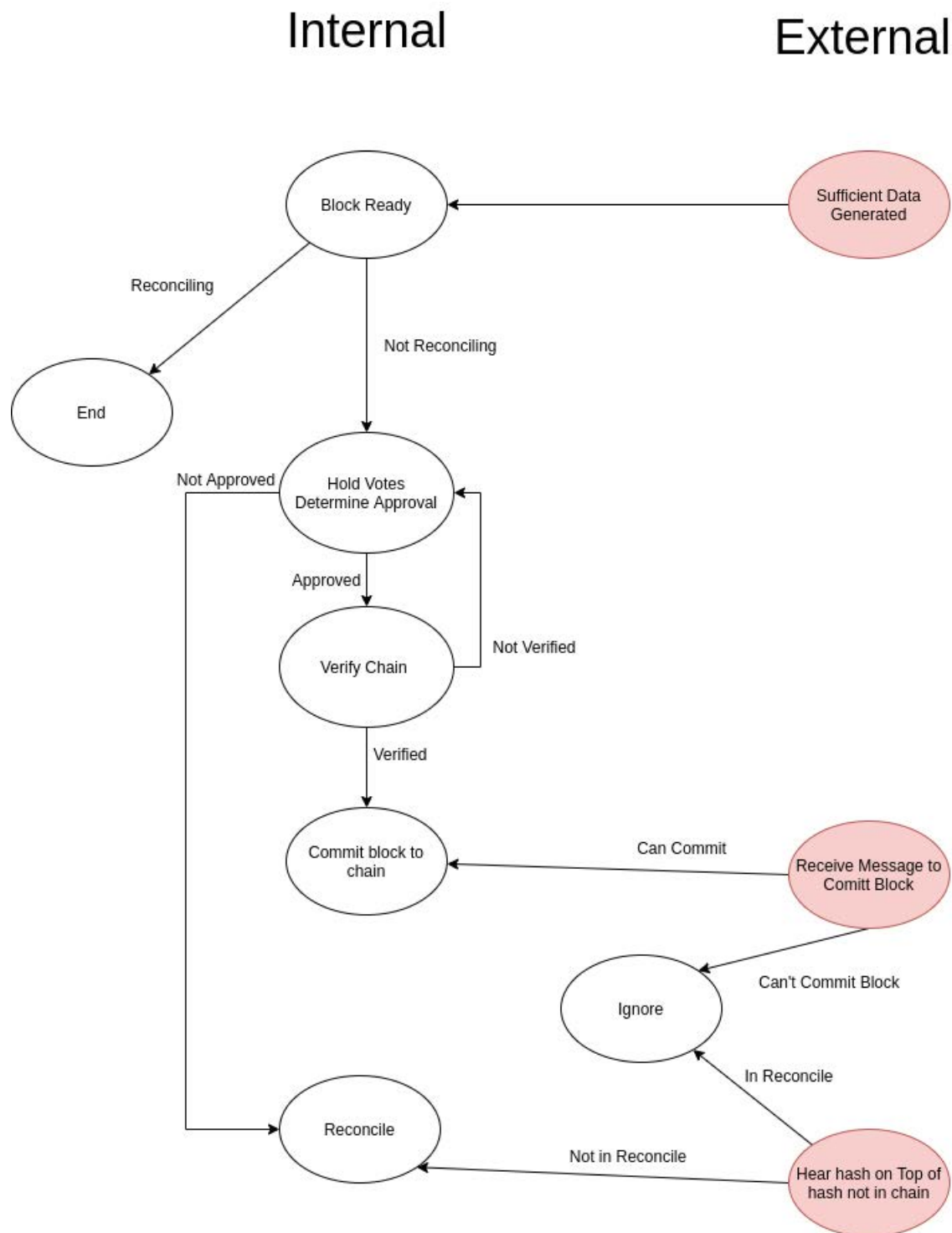


Figure 4.2. Abstracted version of the state model of UCP.

4.2.2 Monterey Phoenix Code

Following abstraction of the UCPs consensus process, we can convert the abstracted algorithm into MP code. In Table 4.2, we identify each segment of code as well as the behavior it models.

Table 4.2. Monterey Phoenix state model code for verification of the UCP commit process.

Code Segment	MP Code	Behavior
1	SCHEMA DLP_STATE	None
2	ROOT INTERNAL: (* Block_Ready *), (* Commit_Message *), (* Hear_Hash_on_top_of_Hash_Not_Present *);	None
3	Block_Ready: (Reconciling End (Not_Reconciling_Conduct_Vote_Not_Aproved Reconcile (Not_Reconciling_Conduct_Vote_Aproved_Not_Verified Commit_Block Not_Reconciling_Conduct_Vote_Aproved_Verified Commit_Block)));	4.a - Block ready, reconciling, algorithm terminates 4.b - Block ready, not reconciling, go to Hold Votes and Determine approval 5.a - Vote disapproval, go to Reconcile 5.b - Vote approval, go to Chain Verify 6.a - Chain changed, go to Hold Votes and Determine Approval 6.b - Chain unchanged, go to Receive Message to Commit Block

4	Commit_Message: (Can_Commit Commit_Block Cant_Commit Ignore) ;	2.a - Can commit, Proceed to Commit Block to Chain 2.b - Can't commit, ignore request 7 - Wait for another external event
5	Hear_Hash_on_top_of_Hash_Not_Present: (Not_Reconciling Reconcile Reconciling Ignore);	3.a - Hear hash, currently reconciling, ignore 3.b - Hear hash, not reconciling, proceed to Reconcile
6	Reconcile: Enter_Reconcile;	8 - Conduct Reconcile and then wait
7	ROOT EXTERNAL: (*Generate_Block_Ready*), (*Generate_Commit_Message*), (*Generate_Hear_Hash_on_top_of_Hash_Not_Present*);	None
8	ENSURE #Reconcile<=\$scope; ENSURE #Commit_Block<=\$scope;	None
9	COORDINATE \$a:Generate_Block_Ready FROM EXTERNAL, \$b:Block_Ready FROM INTERNAL DO ADD \$a PRECEDES \$b; OD;	1 - Sufficient data, go to Block Ready
10	COORDINATE \$a:Generate_Commit_Message FROM EXTERNAL, \$b:Commit_Message FROM INTERNAL DO ADD \$a PRECEDES \$b; OD;	2 - Receive Commit Message

11	COORDINATE \$a:Generate_Hear_Hash_on_top_of_Hash_Not_Present FROM EXTERNAL, \$b:Hear_Hash_on_top_of_Hash_Not_Present FROM INTERNAL DO ADD \$a PRECEDES \$b; OD;	3 - Hear Hash on Top of Hash not in Chain
12	INTERNAL SHARE ALL Reconcile;	None

4.2.3 Discussion

In this section we analyze the results that we received when we compiled the MP model. The full results from the compiled MP model are contained in Appendix B.3. Here we describe how the results from the MP model satisfy the statement made at the beginning of Section 4.2:

1. **Statement** - If a UV's protocol event handlers are in an idle state, then all of its completed blocks must be committed to its local `block chain`.

Analysis - To satisfy this statement, we must establish that the program does not stop running until all of its completed blocks have been committed to the chain. For this to hold true, the algorithm can only terminate with the **Commit Block to Chain** component or the **Reconcile** component. The reason this is the case is because the **Reconcile** component commits all blocks to the `block chain` before it terminates. Likewise, the **Commit Block to Chain** component will continue to commit blocks to the `block chain` until its run out of blocks. In the MP model, every trace ended with the **Commit Block to Chain** or the **Reconcile** component. These results, therefore, satisfy this statement.

4.3 Chapter Summary

To recap, in this chapter we examined two models of the Uniform Chain Protocol (UCP) that we represented with Monterey Phoenix (MP). By examining these models, we were able to verify that the UCP behaved as intended. We also discussed how these two models served to verify the correctness of the four statements we identified at the beginning of the

chapter. In the next chapter, we will discuss how early results provided by MP influenced the design of UCP. We will also discuss the results of field tests conducted at Camp Roberts.

CHAPTER 5:

Design Validation

In this chapter, we discuss design changes we made to Uniform Chain Protocol (UCP) that were influenced by MP and the results of our field tests. Both MP and our field tests demonstrated the behavioral correctness of UCP. For details of the trace evaluation process and the full trace results, refer to Appendix A and Appendix B. For the results of the field tests, refer to Appendix C.

5.1 Monterey Phoenix

In this section, we discuss two design flaws we discovered in the development of UCP as a direct result of the behavioral modeling we performed using MP. By discovering these flaws early, we were able to tweak the design of UCP to eliminate the bugs before developing an actual implementation. For this section, we examine the following two bugs:

1. **Five Drone Bug** - The first bug that was discovered in our original algorithm permitted an `agent` to loop indefinitely in the reconcile process. This problem presented itself when we first ran the consensus model in MP and was in the following steps:
 - (a) The UV attempts to commit a block and cannot because its block chain has diverged from the majority chain of `local UVS`.
 - (b) The UV reconciles with the `local UVS` but does not receive the majority of the votes needed to commit a block.
 - (c) The UV attempts to commit the block and is again unable to do so.
 - (d) Steps b and c are repeated indefinitely.

Figure 5.1 depicts the MP trace in which the five drone bug occurs. In this model, a block was removed from the `block chain` and then immediately re-added. This occurs when the block the UV is trying to commit has a plurality of the votes in the `local UVS`, but lacks the majority required to finalize the commit. Therefore, the attempt to commit the block fails and the `agent` has to immediately reconcile again. This behavior violates Characteristic 6 from Chapter 4, because it prevents a uniform chain from emerging in a fully connected system. To resolve this bug, we added the following:

- (a) A statement into the protocol that requires the UV to commit all of its blocks at the end of reconcile.
- (b) A requirement that UVs must reconcile if they receive a message that shows another UV reconciling towards a different block chain.

With those two adjustments to the protocol, we conducted further UCP testing that verified the elimination of the Five Drones Bug.

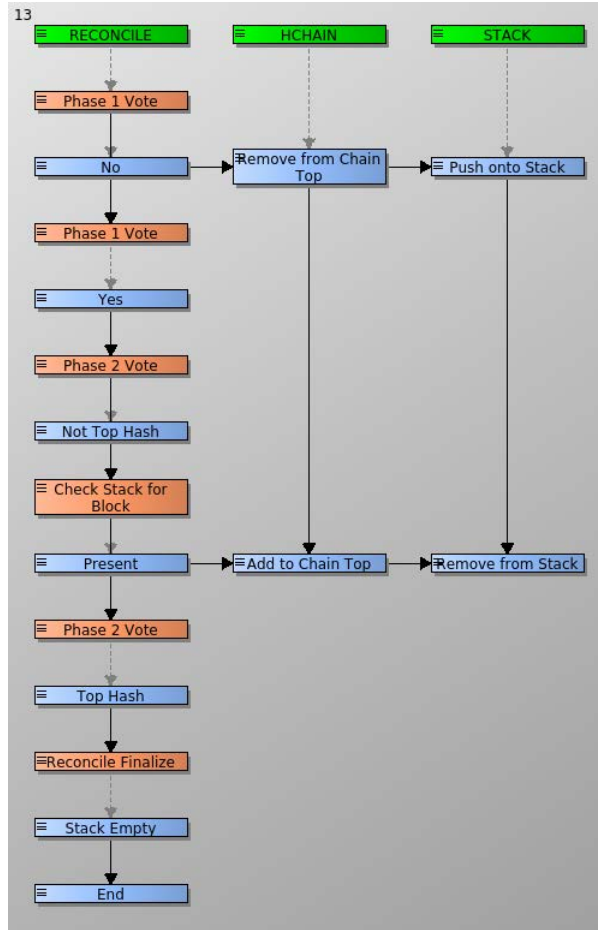


Figure 5.1. Five Drone Bug displayed in the model.

2. **Altered Chain Bug** - While running the state model, we encountered a race condition that could cause corruption of the block chain. In this scenario, the UV commits a block while it is in the process of conducting a vote to commit a previously generated block. In Figure 5.2, we can see a block being committed while a vote is in progress. If a different block is committed to the chain while the vote is in progress, it will

experiment, the swarm consisted of a total of five UAVs that were launched consecutively. After launch, the UCP ran for approximately thirty minutes while the system executed various swarm behaviors. By their very nature, field tests involve a UVS that is not fully connected, the exact sort of system for which this protocol was developed. As a result, we did not expect a uniform block chain to emerge (Characteristic 6). At the conclusion of the field tests, we downloaded the block chains for each of the five UAVs used in the test. Highlights from the field tests include the following:

- We discovered three implementation bugs:
 1. A syntax error in the reconciliation response dictionary from Section 3.6.3. This error was promptly fixed by correcting the syntax.
 2. A syntax error in the implementation of the timers. The syntax was corrected.
 3. A logical error in the implementation of the **Phase 3 Response** in Section 3.6.3. This was fixed in the implementation by altering the implementation to match UCP.
- Importantly, we discovered no protocol bugs. This further validates that UCP works as intended.
- Three of the five UAVs successfully ran UCP and achieved consistent block chains. The other two encountered the aforementioned implementation bugs which prevented them from running UCP successfully. The block chains files downloaded from these UAVs are located in Appendix C.

Overall, the field tests were successful as we found no bugs in the UCP. This serves to further validate the correctness of UCP.

5.3 Chapter Summary

In this chapter, we examined two design bugs in the UCP that were identified using MP. We also presented the design changes we made to UCP to eliminate these bugs. Furthermore, we discussed the results of the successful field tests we conducted at Camp Roberts. In the next chapter, we provide a conclusion and identify future work that may stem from this thesis.



Figure 5.3. A Zephyr fixed-wing UAV being launched at Camp Roberts, CA.

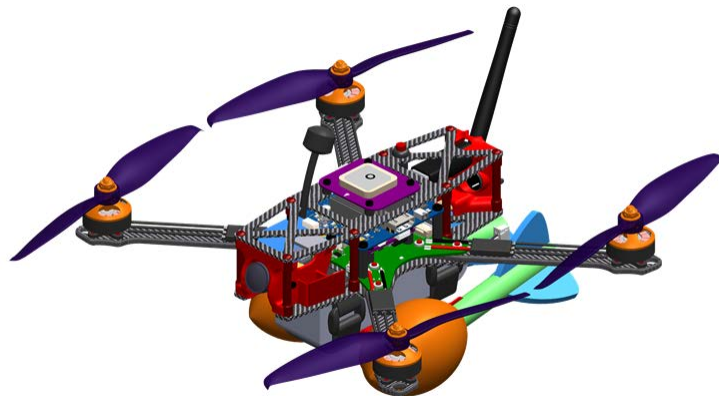


Figure 5.4. A Mosquito Hawk quadrotor UAV.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:

Conclusion

This thesis contributes a novel Distributed Ledger Protocol (DLP) that assures the availability of data generated by distributed, autonomous, multi-vehicle systems for post-mission analysis. The Uniform Chain Protocol (UCP), which was designed and verified in this thesis project, provides Unmanned Vehicle Systems (UVS) the ability to distribute data across all Unmanned Vehicles (UV) in the system. The intent of distributing data in this project was that it allows the data to be saved for post-mission analysis. UCP prevents data generated during a mission from being lost, even if the UV that generated the data does not survive the mission. Monterey Phoenix (MP) was used to verify the behavioral correctness of this program in small examples. In the MP models, we showed that no apparent issues exist within UCP. Finally, field tests were conducted to validate the functionality of UCP. While conducting these field tests, we found no errors in the UCP, which further serves to validate that UCP works as it was designed to. For the rest of this chapter, we will discuss the implications of UCP for the Department of Defense (DoD) as well as future work that may originate from this thesis.

6.1 Implications

The UCP is especially useful for an organization like the DoD. In the design of UCP, we made the assumption that the protocol would be run on vehicles that experience destruction or network disconnection. The DoD operates within an environment where UVs are assumed to be at a risk of being damaged or destroyed. In that aspect, the DoD can benefit from resilience provided by the UCP. Although UCP was designed for UVs, there is no requirement that UCP be run only on UVs. Manned vehicles, or any other distributed vehicle system, has the potential to run UCP. Likewise, although the target domain for UCP is airspace, UCP has no design requirement that prevents it from being implemented on land, space, or water based vehicles. The culmination of these characteristics makes UCP a great asset for use in distributed, multi-vehicle, autonomous systems.

6.2 Future Work

The work conducted in this thesis has uncovered numerous possibilities for future work. These possibilities include the following work:

Monterey Phoenix Upgrade - The RIGAL compiler that MP runs was developed in 1992. Due to its age, the MP platform is a legacy 32-bit application. Since 32-bit applications are limited to 4GB of RAM, MP's ability to perform exhaustive traces is greatly diminished by its lack of resources. If MP were upgraded to a 64-bit application, the memory constraint would be alleviated. Upgrading MP from a 32-bit application to a 64-bit application would be a great use of resources.

Further Development of Monterey Phoenix (MP) Models - We created two models using MP for this project, but we have the ability to create more models and prove more behaviors. In this project, we had to limit our scope to creating a protocol that satisfied all of the design requirements. There may be more behaviors that we want to verify with MP and those behaviors could be addressed in future work.

Proofs in COQ - COQ is a mathematical theorem proving tool that was developed in the 1980's [16]. COQ proofs can be created and compared to the models that we created in this project with MP. This would enable further validation of the correctness of UCP. It could also lead to the discovery of other behaviors that were not identified by our MP model.

Time Complexity of UCP - In this thesis project, we completed design, validation and implementation. In the future, it would be an excellent contribution to evaluate the time complexity in the best case, worst case and average case of UCP. This work could be used to determine how well UCP performs in comparison to other types of distributed ledger protocol (DLP).

Formal Mathematical Methods - In this thesis, we conducted lightweight proofs with Monterey Phoenix (MP). In future work, it would be beneficial to determine what kinds of things can be formally proven about UCP. The exploration of this type of work could lead to a rigorous mathematical proof that provides more depth than MP.

APPENDIX A:

Monterey Phoenix Consensus Results

This appendix contains the results for the Monterey Phoenix consensus traces with commentary.

A.1 Analyzing the Results

To understand these results, we must examine them within the context of the design intent. To analyze these results, its recommended to follow these steps:

1. Examine with the first event following RECONCILE. Proceed to step 2.
2. In this step, we make a determination of whether or not the behavior exhibited up to this event in the trace is intended. We desire to answer the following general questions:
 - Are the events that led to this event intended in the design of the Uniform Chain Protocol (UCP)?
 - Are there any scenarios where this behavior is not intended?

To determine the answer to the general questions, we bear in mind the characteristics we identified in Section 4.1. We ask the following specific questions:

- (a) Does the behavior up to this point in the trace demonstrate more than one copy of a `block` being stored on the `block chain`?
- (b) Is this a fully connected system (Does not contain a `No_Response` event)? If so, are any `blocks` not committed to the `block chain`?
- (c) If this is a fully connected system, does it demonstrate that one uniform `block chain` does not emerge?

If any of these questions result in an answer of “yes”, then the behavior exhibited by this trace is unintended. This shows us that the our protocol has a design flaw that must be resolved. If the answer to all of these questions is “no”, then we can conclude that this trace does not demonstrate a design flaw in the UCP. We now proceed to Step 3.

Note : When asking these questions, we must be mindful that we may have abstracted away details in our model. For example, in this model we abstracted away locks. Therefore, we are not really particularly concerned with race conditions in the context

- of this model.
3. If this is the last event in the model, the behavior is correct if we discovered no unintended behavior. If it is not the last event, we examine the next event. We now proceed back to Step 2.

A.2 Example

In this section, we use Figure A.1 as an example of how to analyze the MP results. The steps are completed as follows:

Step 1. Examine Phase_1_Vote.

Step 2. Answer questions a, b, and c. Proceed to Step 3.

- a. No. Does not demonstrate a block being stored more than once.
- b. No. Does not demonstrate that blocks are left to commit.
- c. No. Does not demonstrate a full block chain does not emerge.

Step 3. Examine next event. This is No_Response. Proceed to step 2.

Step 2. Answer questions a, b, and c. Proceed to Step 3.

- a. No. Does not demonstrate a block being stored more than once.
- b. No. Not in a fully connected system because No_Response was encountered.
- c. No. Not in a fully connected system because No_Response was encountered.

Step 3. Examine next event. This is Reconcile_Finalize. Proceed to step 2.

Step 2. Answer questions a, b, and c. Proceed to Step 3.

- a. No. Does not demonstrate a block being stored more than once.
- b. No. Not in a fully connected system because No_Response was previously encountered.
- c. No. Not in a fully connected system because No_Response was previously encountered.

Step 3. Examine next event. This is Stack_Empty. Proceed to step 2.

Step 2. Answer questions a, b, and c. Proceed to Step 3.

- a. No. Does not demonstrate a block being stored more than once.
- b. No. Not in a fully connected system because No_Response was previously encountered.
- c. No. Not in a fully connected system because No_Response was previously encountered.

Step 3. Examine next event. This is End. Proceed to Step 2.

Step 2. Answer questions a, b, and c. Proceed to Step 3.

a. No. Does not demonstrate a block being stored more than once.

b. No. Not in a fully connected system because No_Response was previously encountered.

c. No. Not in a fully connected system because No_Response was previously encountered.

Step 3. End is the last event. All of the Step 2 questions received a “no” answer, so the trace completed with no unintended behavior.

A.3 Results

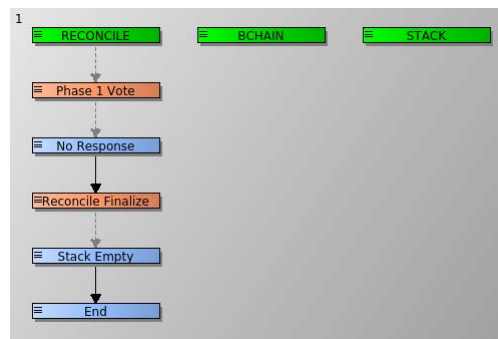


Figure A.1. This traces a scenario where the UAV is disconnected from the UVS. This behavior is intended.

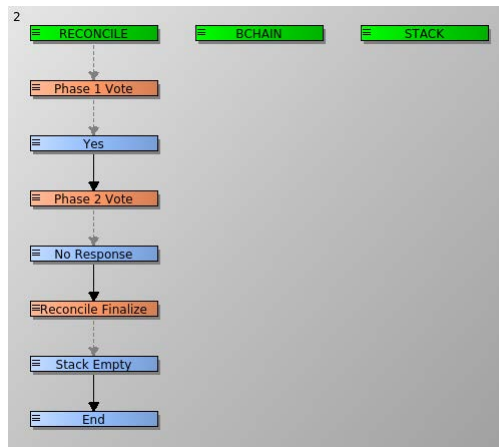


Figure A.2. This traces a scenario where the UAV is disconnected from the UVS. This behavior is intended.

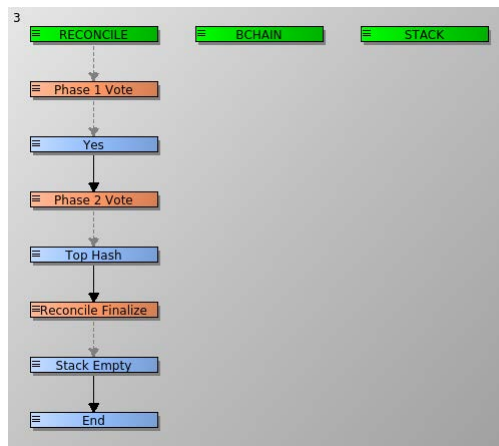


Figure A.3. This traces a scenario where the UAV finds the top hash immediately. This behavior is intended.

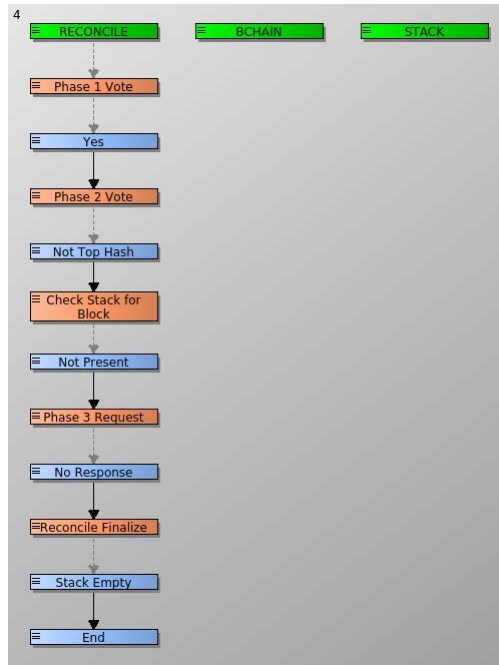


Figure A.4. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

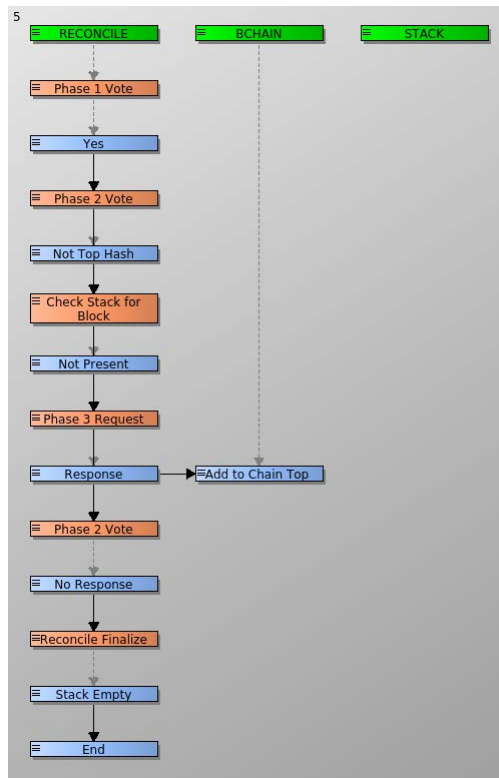


Figure A.5. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

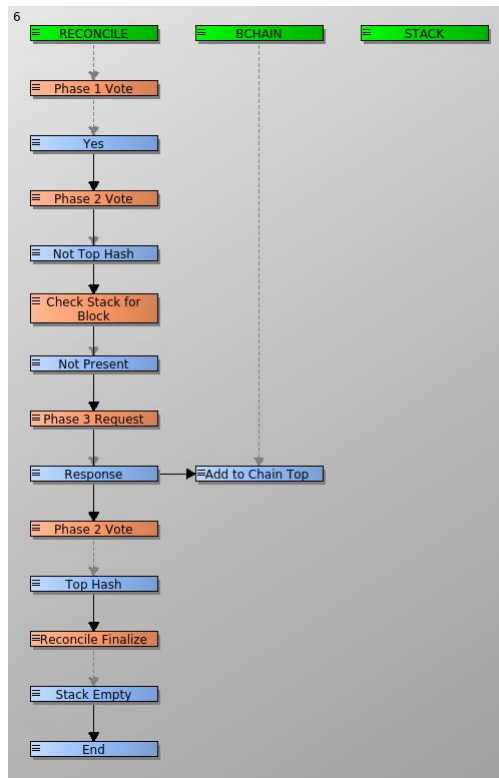


Figure A.6. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

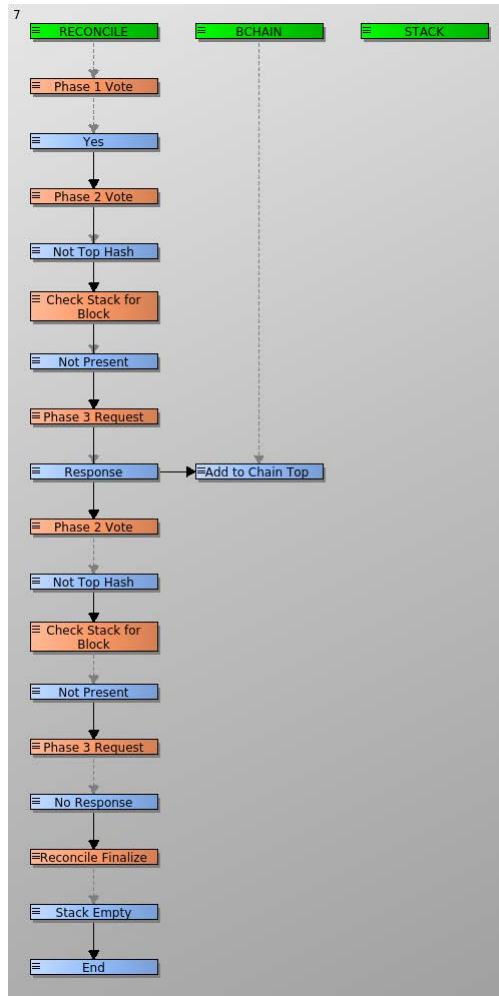


Figure A.7. This traces a scenario where the UAV starts to reconcile with the UVS and then becomes disconnected. This behavior is intended.

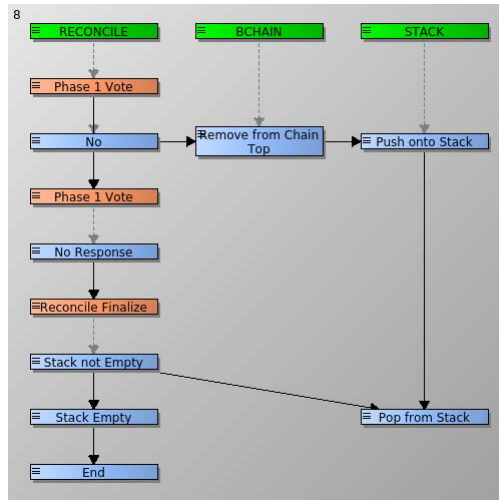


Figure A.8. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

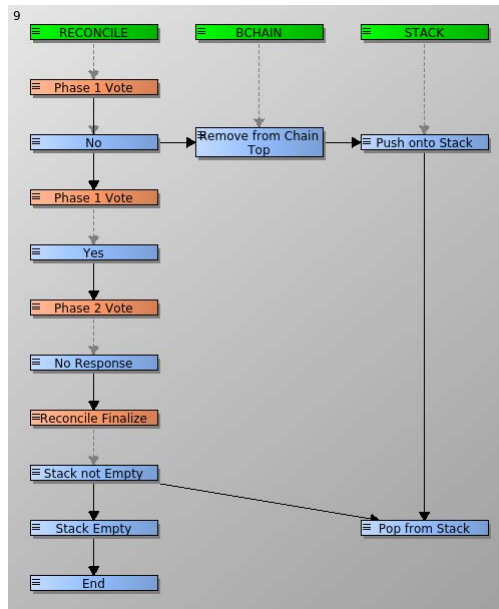


Figure A.9. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

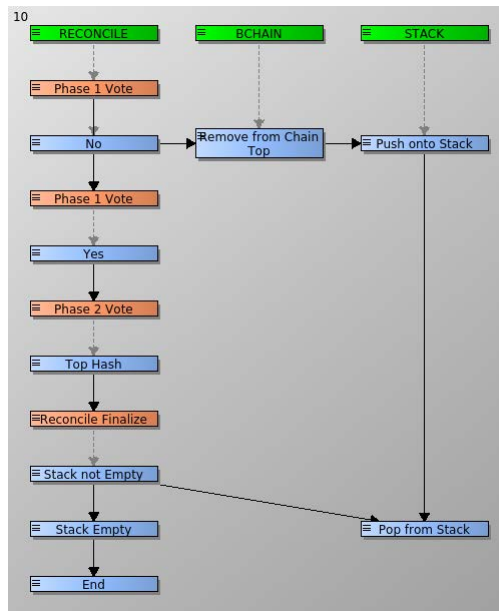


Figure A.10. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

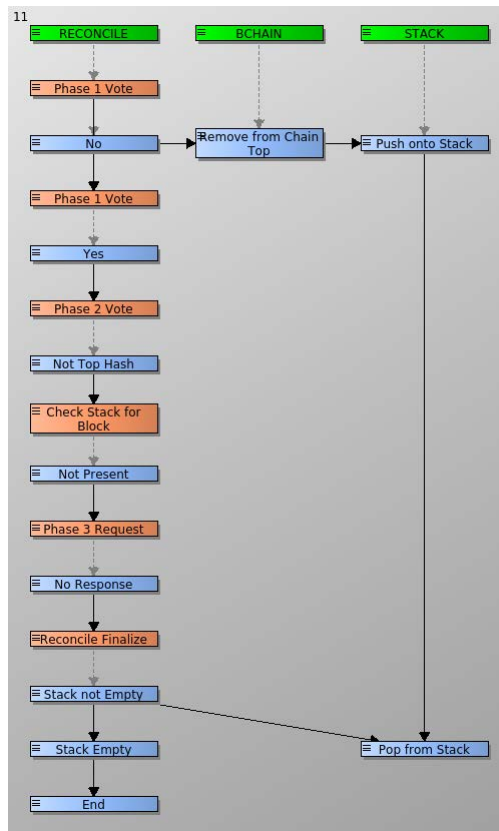


Figure A.11. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

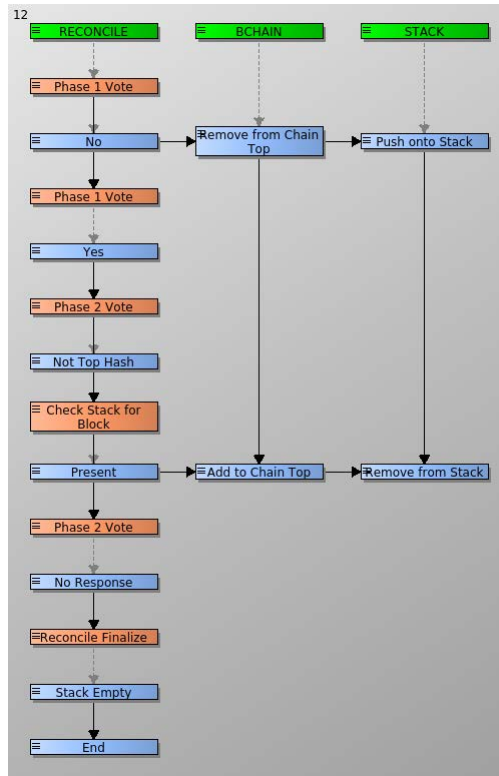


Figure A.12. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

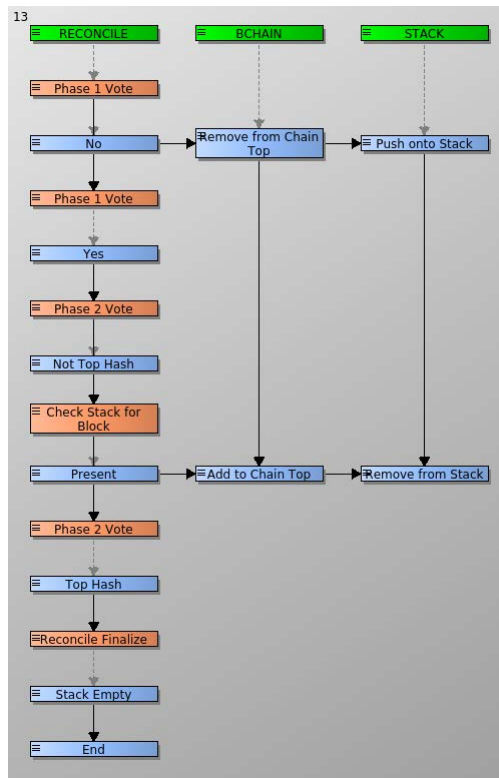


Figure A.13. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

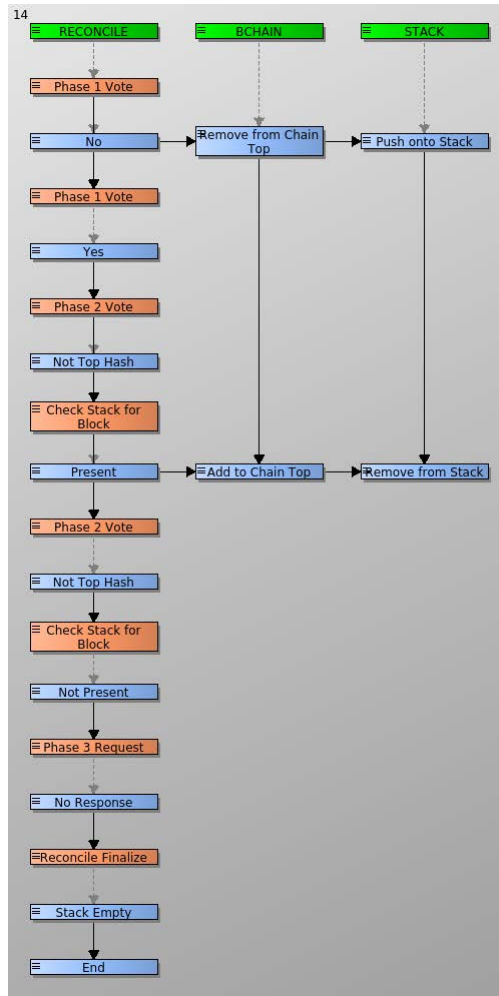


Figure A.14. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

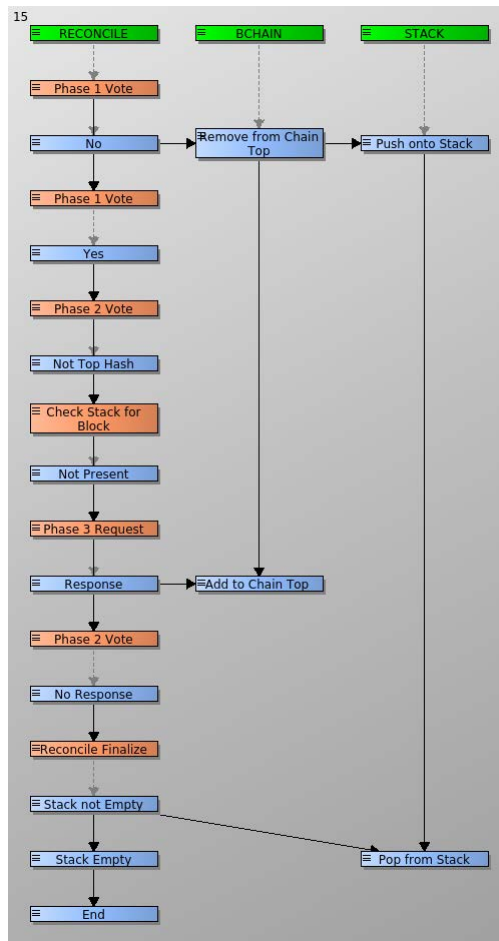


Figure A.15. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

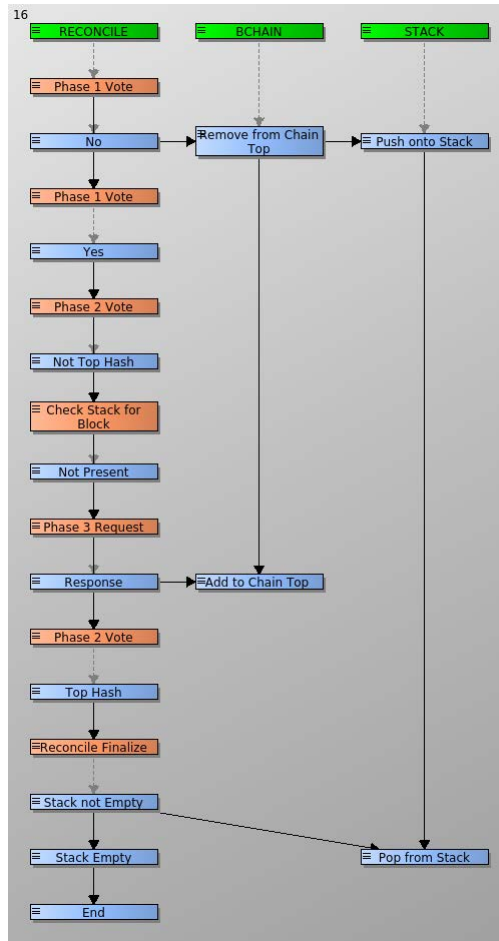


Figure A.16. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

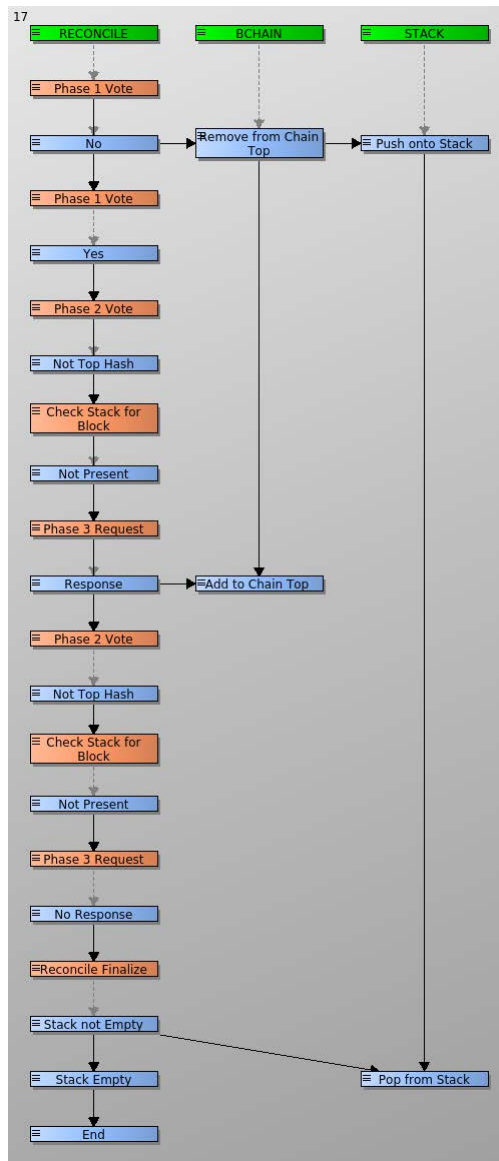


Figure A.17. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

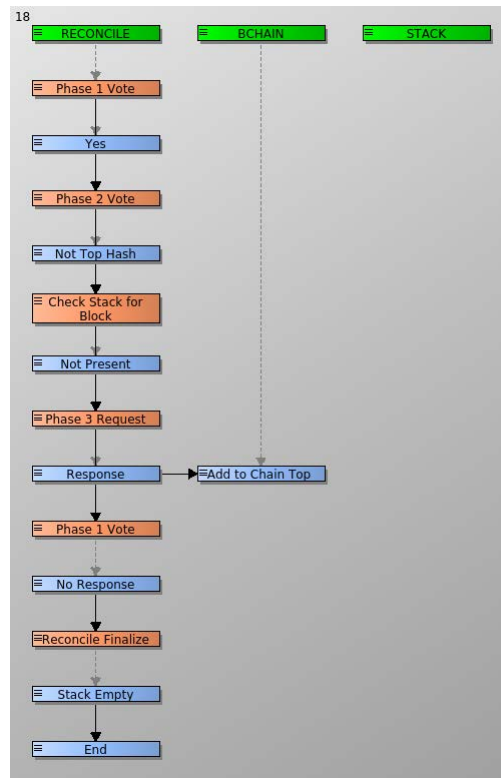


Figure A.18. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

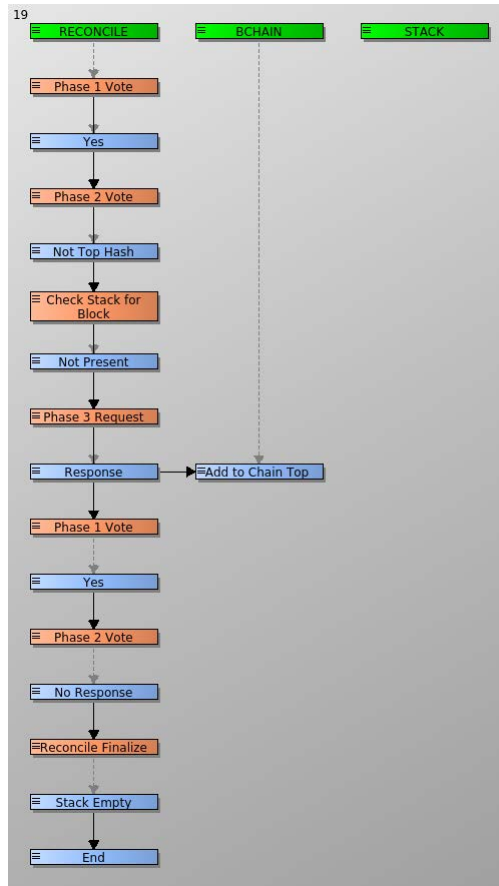


Figure A.19. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

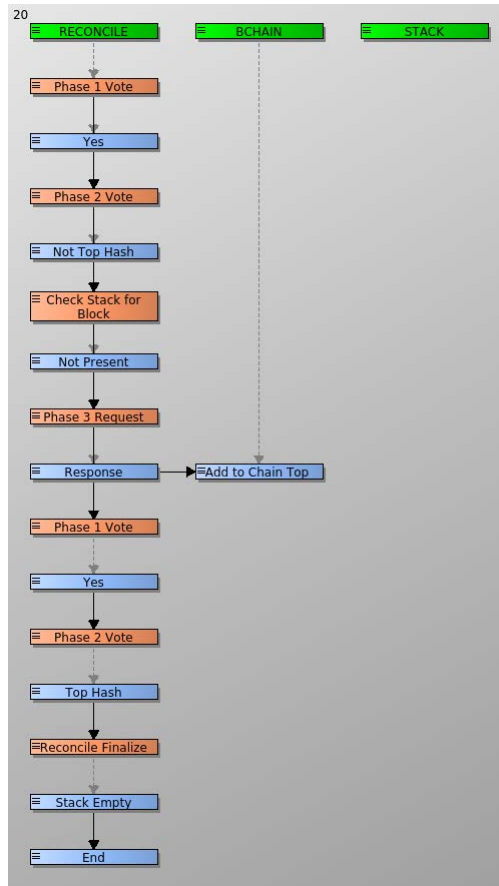


Figure A.20. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

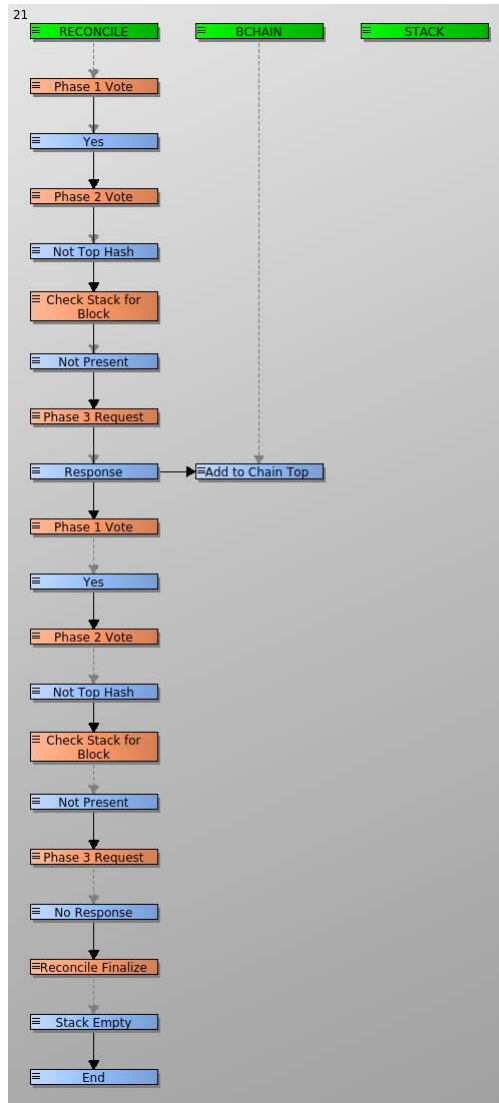


Figure A.21. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

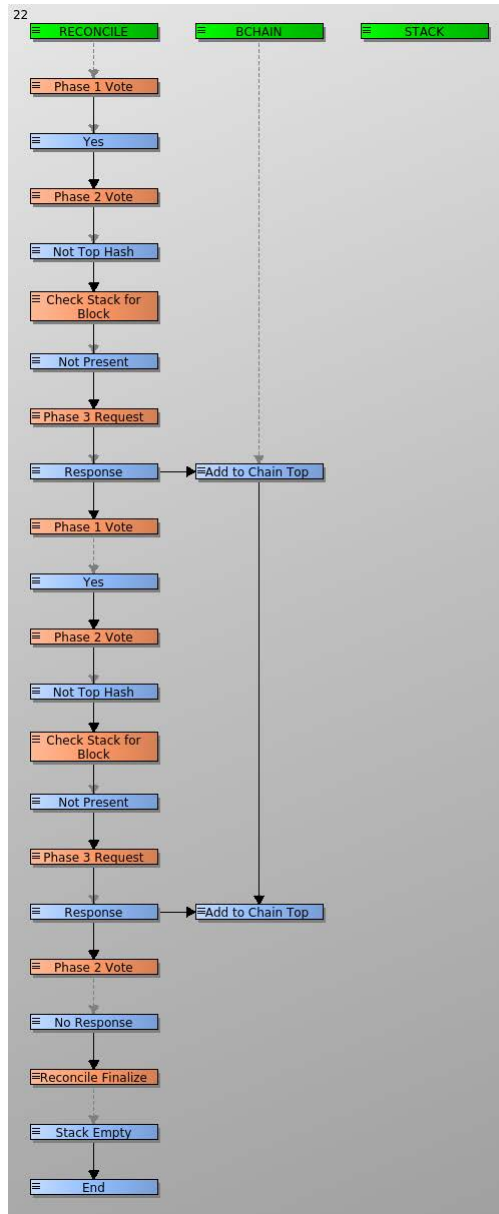


Figure A.22. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

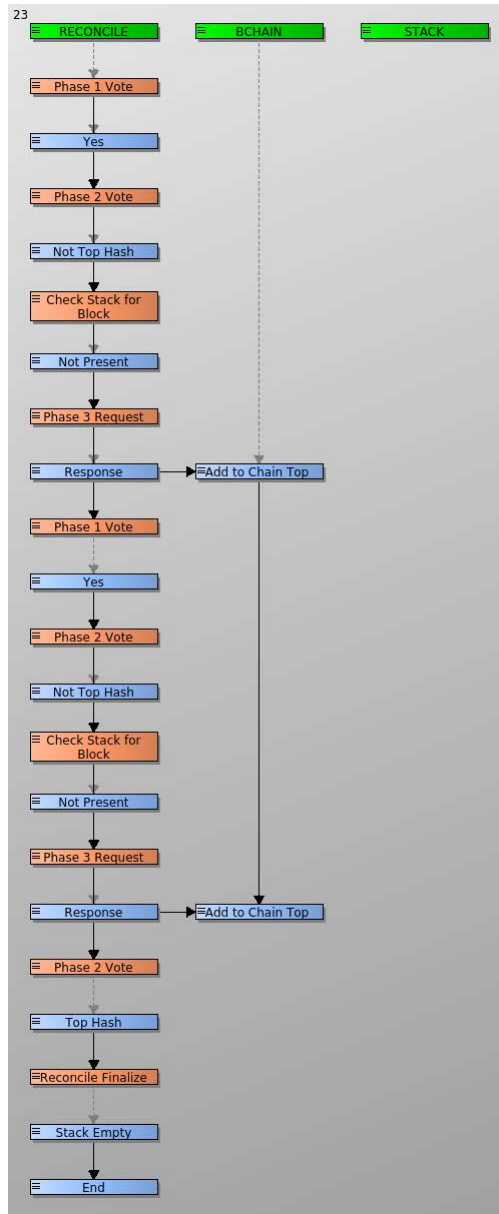


Figure A.23. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

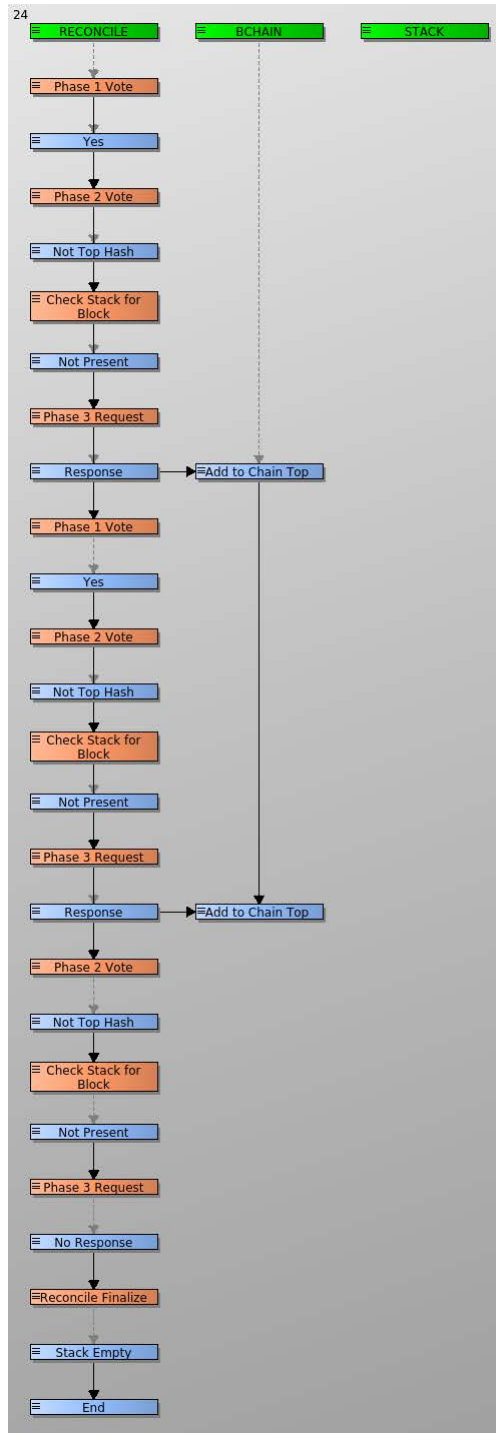


Figure A.24. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

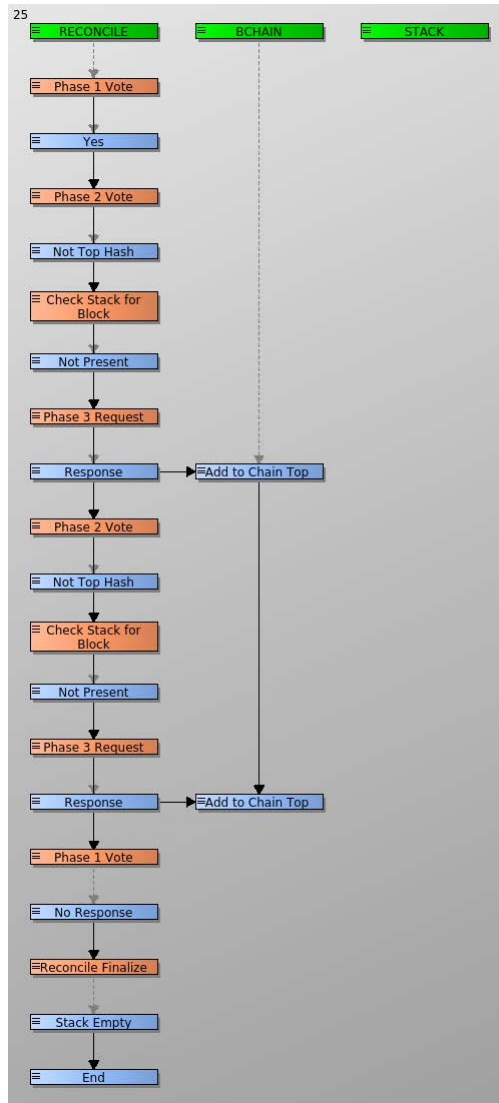


Figure A.25. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

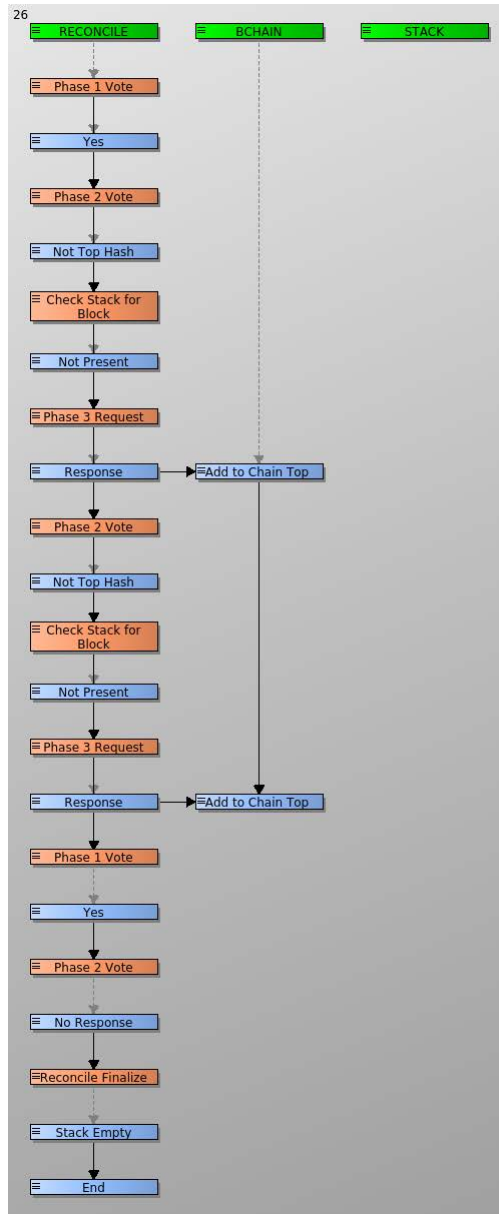


Figure A.26. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

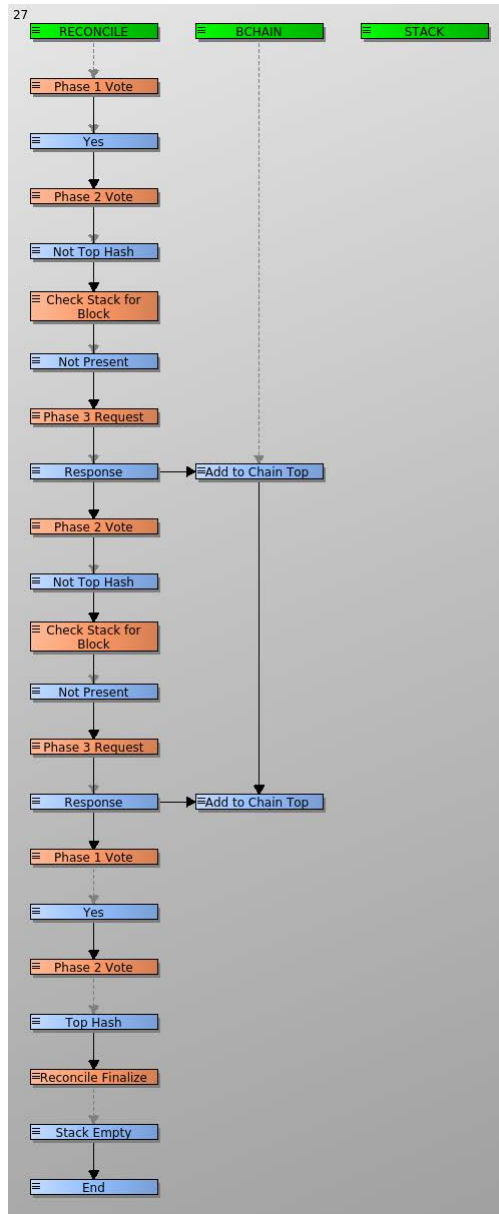


Figure A.27. This traces a scenario where the UAV successfully reconciles with the UVS. This behavior is intended.

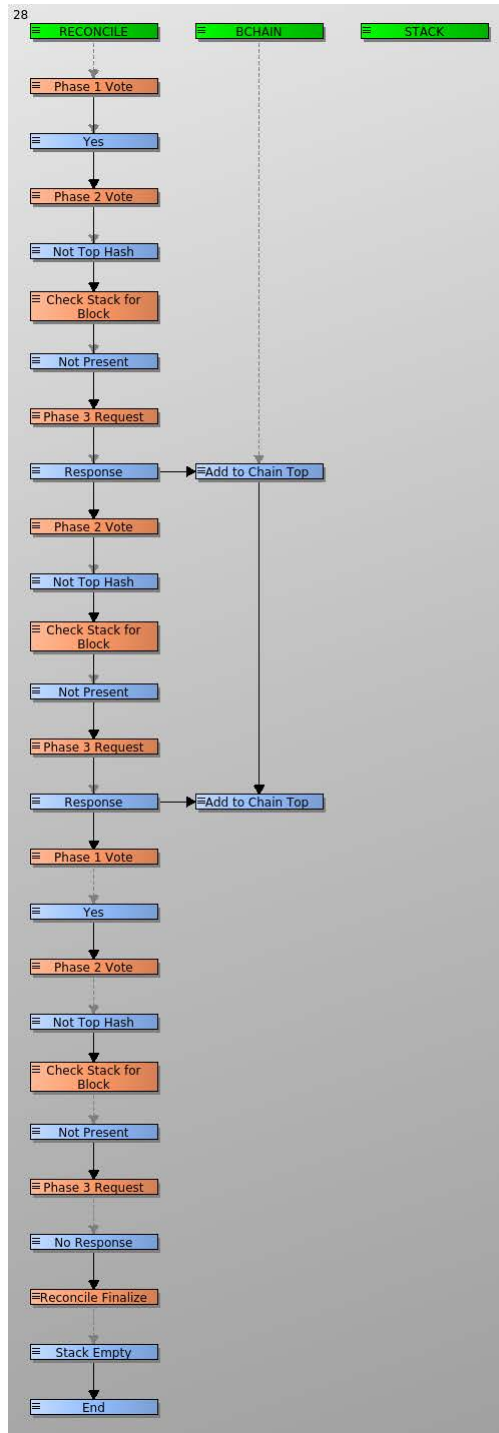


Figure A.28. This traces a scenario where the UAV becomes disconnected from the UVS while its reconciling. This behavior is intended.

APPENDIX B:

Monterey Phoenix State Results

This appendix contains the results for the Monterey Phoenix state traces with commentary.

B.1 Analyzing the Results

To understand these results, we must examine them within the context of the design intent.

To analyze these results, its recommended to follow these steps:

1. Examine the first event following INTERNAL.
2. Next, we ask the following general questions:
 - Are the events that led to this event intended in the design of UCP?
 - Are there any scenarios where this behavior is not intended?

To determine the answer to these questions, we reference the characteristic we are trying to verify in Section 4.2. We ask the following specific questions:

- (a) Is the UV in an idle state?
- (b) Does this behavior demonstrate that any of the completed blocks are not committed to the block chain?

If both of these questions result in an answer of “yes”, then the behavior exhibited by this trace is unintended. This shows us that the our protocol has a design flaw that must be resolved. If the answer to either of these questions is “no”, then we can conclude that this trace does not demonstrate a design flaw in the UCP. We now proceed to Step 3.

Note : When asking these questions, we must be mindful that we may have abstracted away details in our model. For example, in this model we abstracted away locks. Therefore, we aren’t really particularly concerned with race conditions in the context of this model.

3. If this is the last event in the model, the behavior has been verified if no unintended behavior was discovered. If it is not the last event, we examine the next event. We now proceed back to Step 2.

B.2 Example

In this example, We use Figure B.1 as an example of how to analyze the MP results. The steps are completed as follows:

Step 1. Examine Hear_Hash_on_Top_of_Hash_Not_Present.

Step 2. Answer questions a and b. Proceed to Step 3.

a. No. Not in an idle state.

b. No. We have not demonstrated that any completed blocks are not in the block chain.

Step 3. Examine next event. This is Not_Reconciling. Proceed to step 2.

Step 2. Answer questions a and b.

a. No. Not in an idle state.

b. No. We have not demonstrated that any completed blocks are not in the block chain.

Step 3. Examine next event. This is Reconcile. Proceed to step 2.

Step 2. Answer questions a and b.

a. No. Not in an idle state.

b. No. We have not demonstrated that any completed blocks are not in the block chain.

Step 3. Examine next event. This is Enter_Reconcile. Proceed to step 2.

Step 2. Answer questions a and b.

a. No. Not in an idle state. Enter_Reconcile is not considered an idle state.

b. No. We have not demonstrated that completed blocks are not in the block chain.

Step 3. Enter_Reconcile is the last event. All of the Step 2 questions received a “no” answer, so the trace completed with no unintended behavior.

B.3 Results



Figure B.1. This figure traces a scenario where nothing occurs. This behavior is intended.

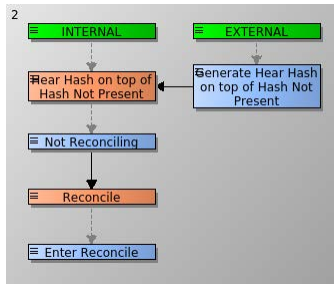


Figure B.2. This figure traces a scenario where a hash on top of a hash is heard and the UAV reconciles. This behavior is intended.

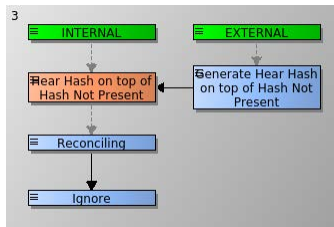


Figure B.3. This figure traces a scenario where a Hash on top of a Hash is heard while the UAV is reconciling. This behavior is intended.

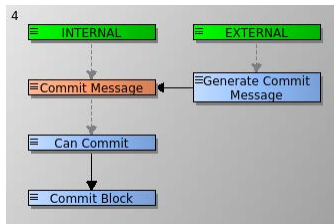


Figure B.4. This figure traces a scenario where a block is committed. This behavior is intended.

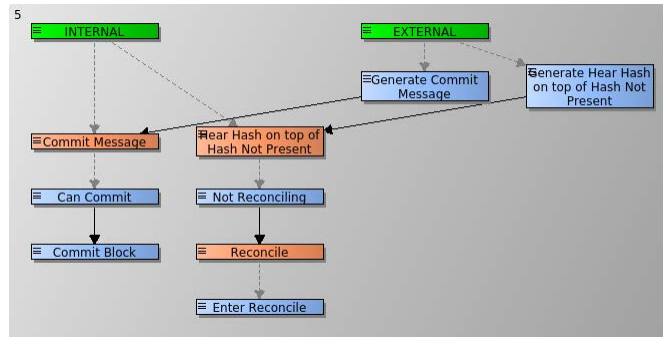


Figure B.5. This figure traces a scenario where a block is committed and a hash on top of a hash is heard and the UAV reconciles. This behavior is intended.

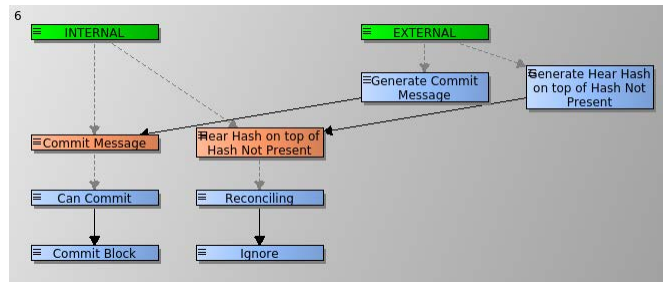


Figure B.6. This figure traces a scenario where a block is committed and a hash on top of a hash is heard while the UAV reconciling. This behavior is intended.

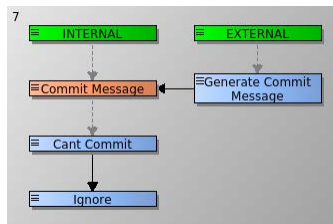


Figure B.7. This figure traces a scenario where a block is can't be committed. This behavior is intended.

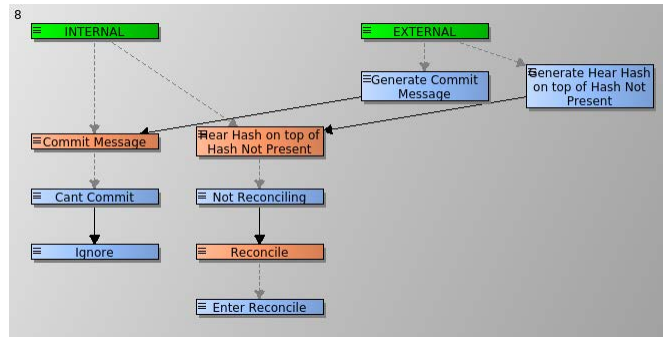


Figure B.8. This figure traces a scenario where a block is can't be committed and a hash on top of a hash is heard and the UAV reconciles. This behavior is intended.

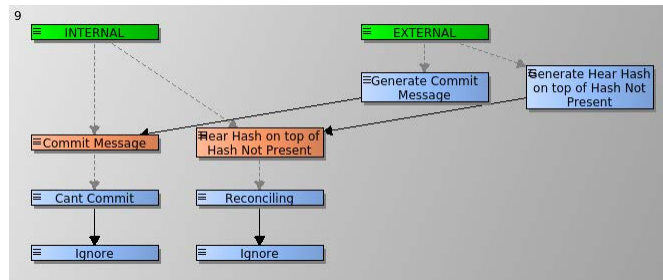


Figure B.9. This figure traces a scenario where a block is can't be committed and a hash on top of a hash is heard while the UAV reconciles. This behavior is intended.

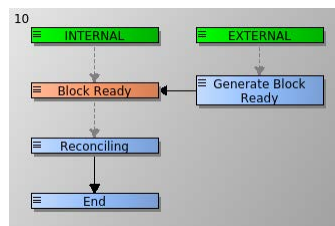


Figure B.10. This figure traces a scenario where a block is determined to be ready while the UAV reconciles. This behavior is intended.

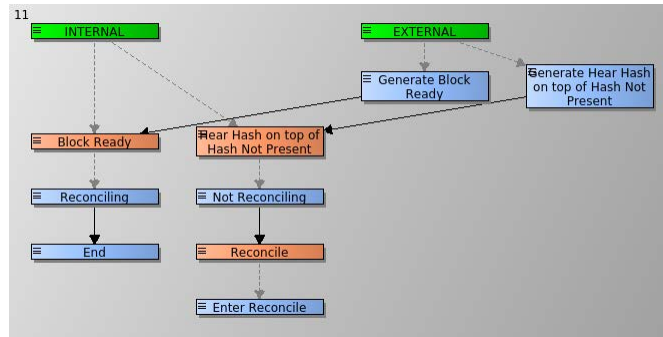


Figure B.11. This figure traces a scenario where a hash on top of a hash is heard causing the UAV to reconcile. Thereafter, a block is determined to be ready while the UAV reconciles. This behavior is intended.

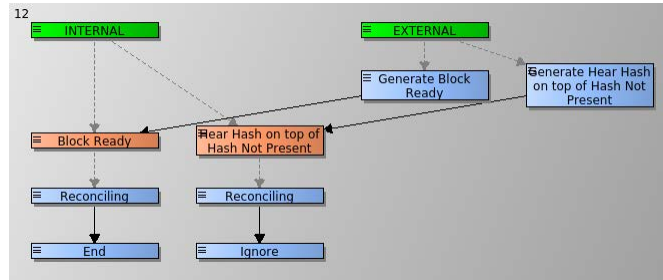


Figure B.12. This figure traces a scenario where a block is determined to be ready while the UAV reconciles and a hash on top of a hash is heard at the same time. This behavior is intended.

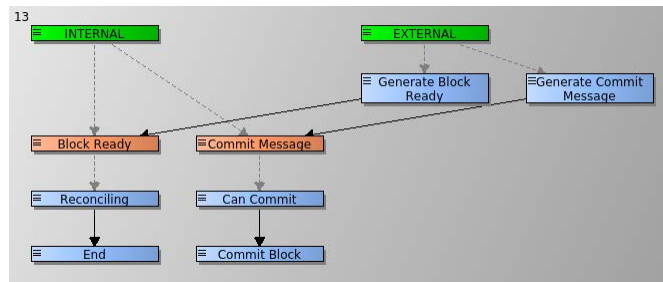


Figure B.13. This figure traces a scenario where a block is determined to be ready while the UAV reconciles. Immediately after reconcile finishes, the UAV gets a commit message. This behavior is intended.

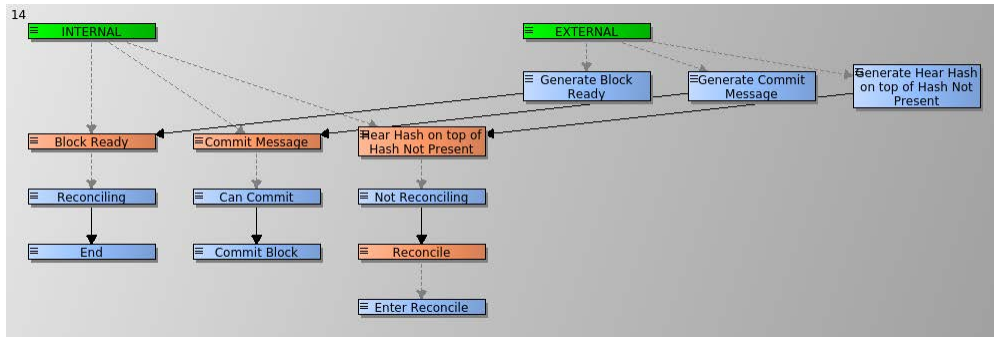


Figure B.14. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

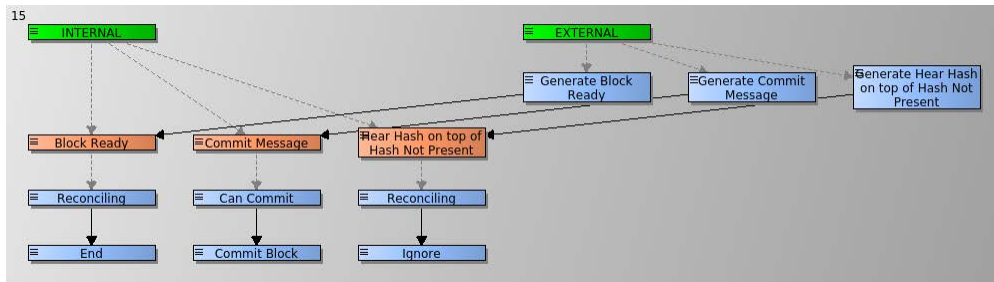


Figure B.15. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

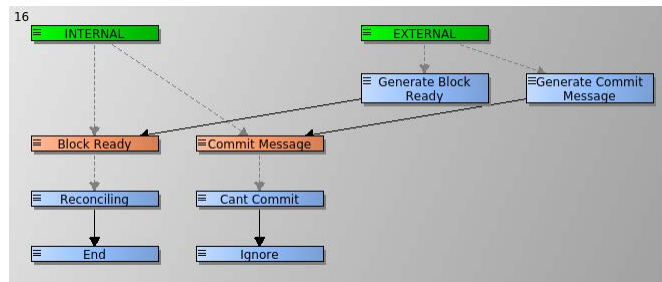


Figure B.16. This figure traces a scenario block is determined to be ready, and a commit message is received. This behavior is intended.

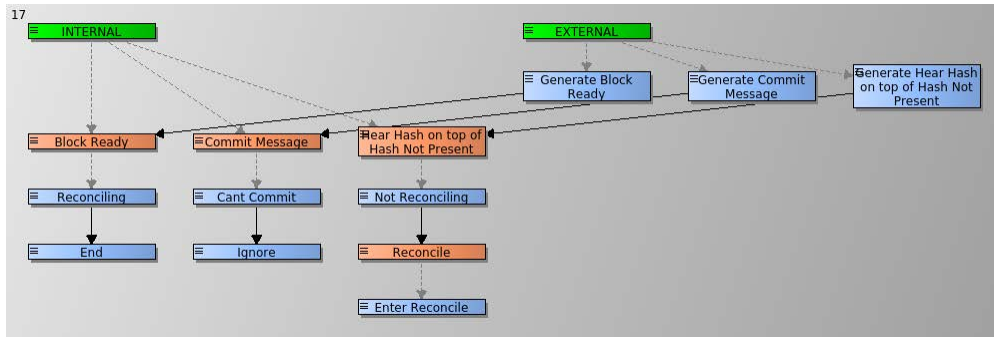


Figure B.17. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

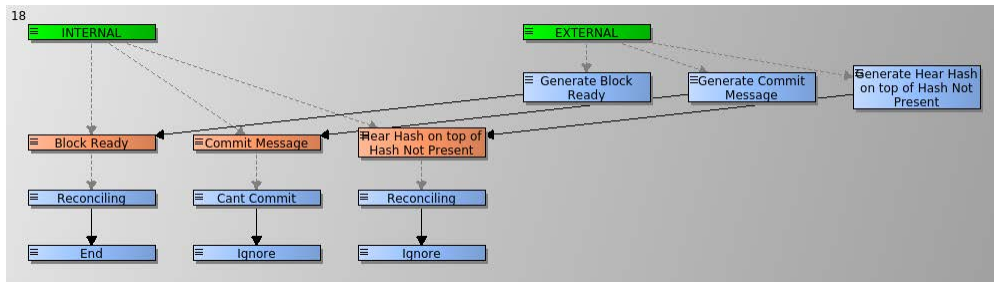


Figure B.18. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

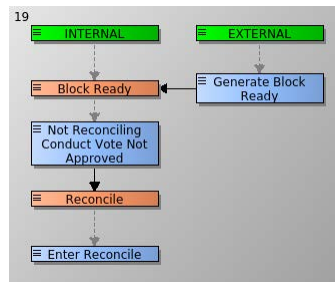


Figure B.19. This figure traces a scenario where a block is determined to be ready. This behavior is intended.

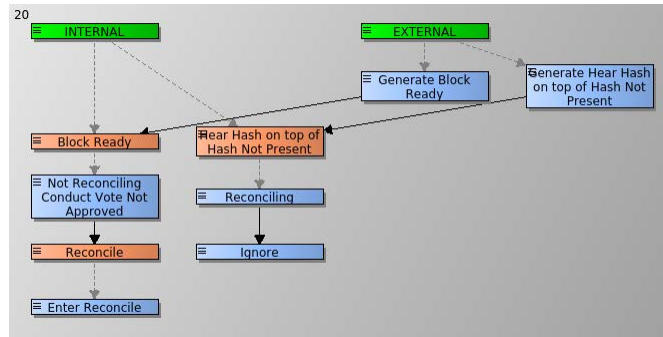


Figure B.20. This figure traces a scenario where a hash on top of a hash is heard, and a block is determined to be ready. This behavior is intended.

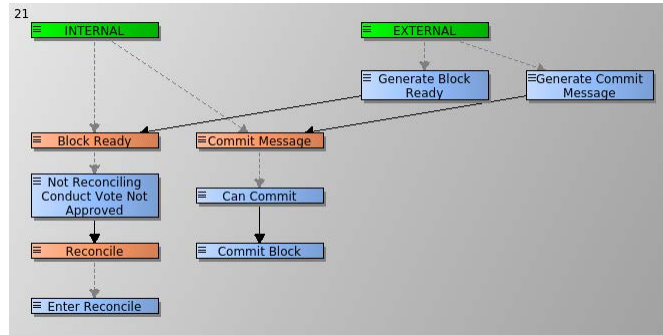


Figure B.21. This figure traces a scenario where a block is determined to be ready, and a commit message is received. This behavior is intended.

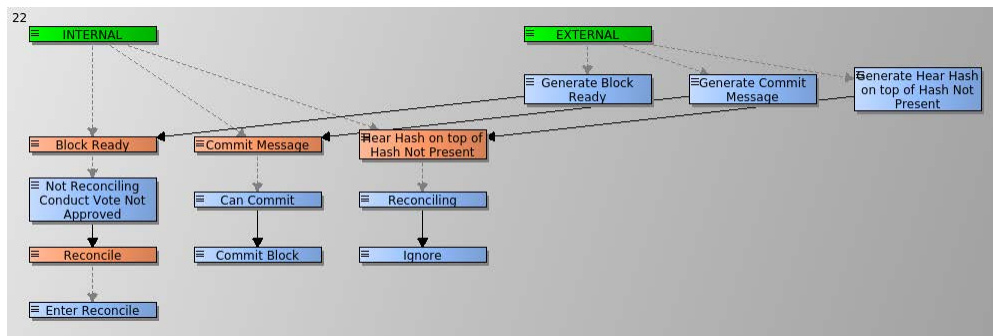


Figure B.22. This figure traces a scenario where a hash on top of a hash is heard, block is determined to be ready, and a commit message is received. This behavior is intended.

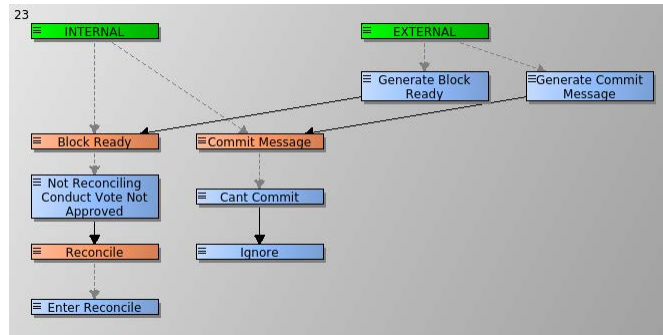


Figure B.23. This figure traces a scenario where a block is determined to be ready, and a commit message is received. This behavior is intended.

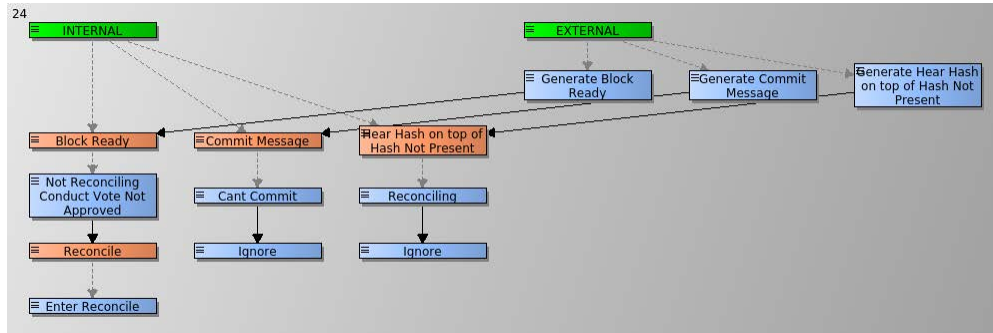


Figure B.24. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

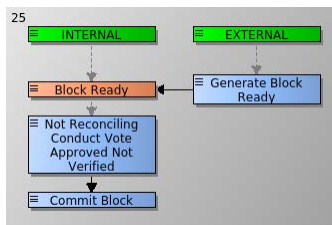


Figure B.25. This figure traces a scenario where a block is determined to be ready. This behavior is intended.

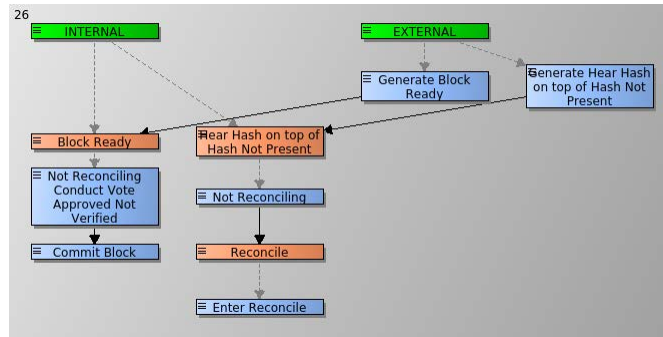


Figure B.26. This figure traces a scenario where a hash on top of a hash is heard, block is determined to be ready, and a commit message is received. This behavior is intended.

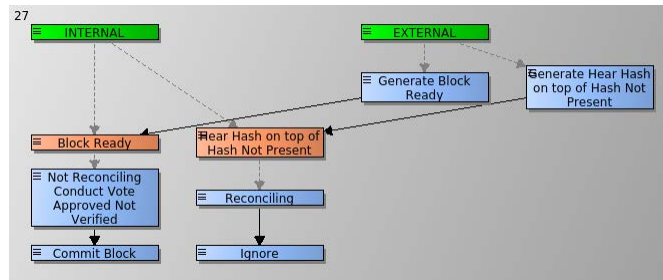


Figure B.27. This figure traces a scenario where a hash on top of a hash is heard, and a block is determined to be ready. This behavior is intended.

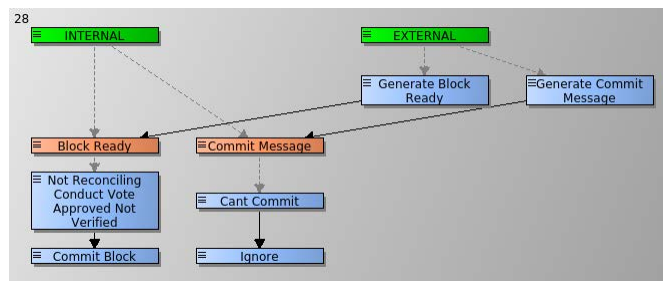


Figure B.28. This figure traces a scenario where a block is determined to be ready, and a commit message is received. This behavior is intended.

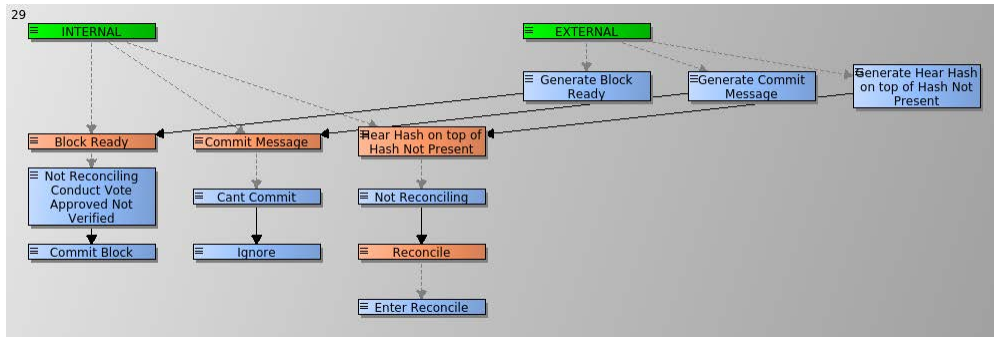


Figure B.29. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

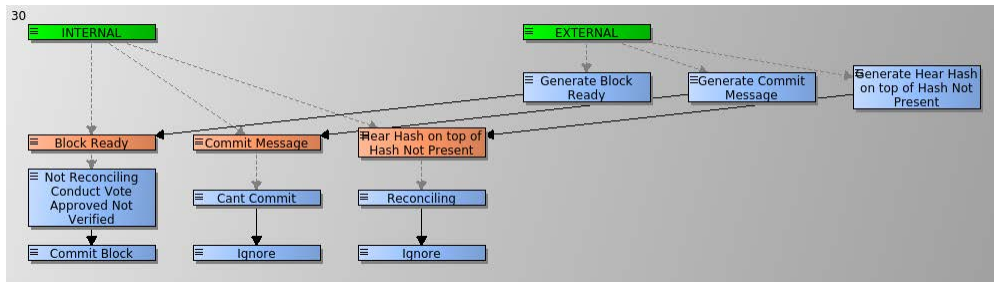


Figure B.30. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

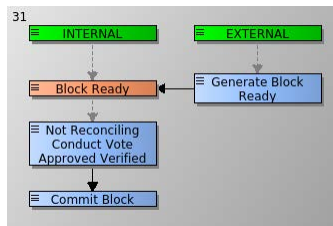


Figure B.31. This figure traces a scenario where a block is determined to be ready. This behavior is intended.

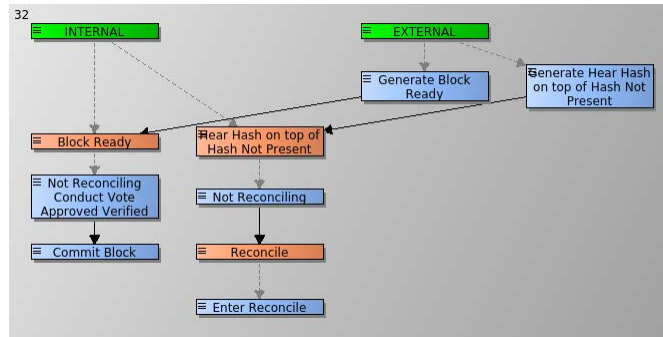


Figure B.32. This figure traces a scenario where a hash on top of a hash is heard, and block is determined to be ready. This behavior is intended.

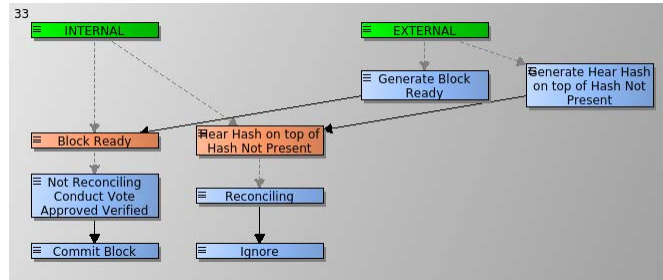


Figure B.33. This figure traces a scenario where a hash on top of a hash is heard, and block is determined to be ready. This behavior is intended.

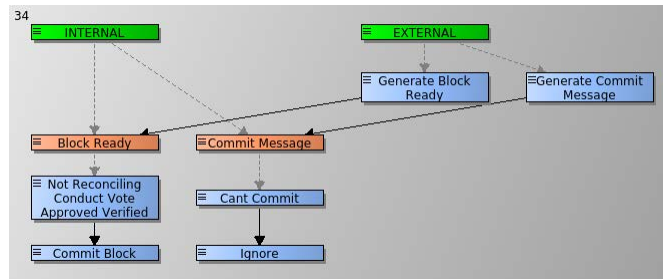


Figure B.34. This figure traces a scenario where a block is determined to be ready, and a commit message is received. This behavior is intended.

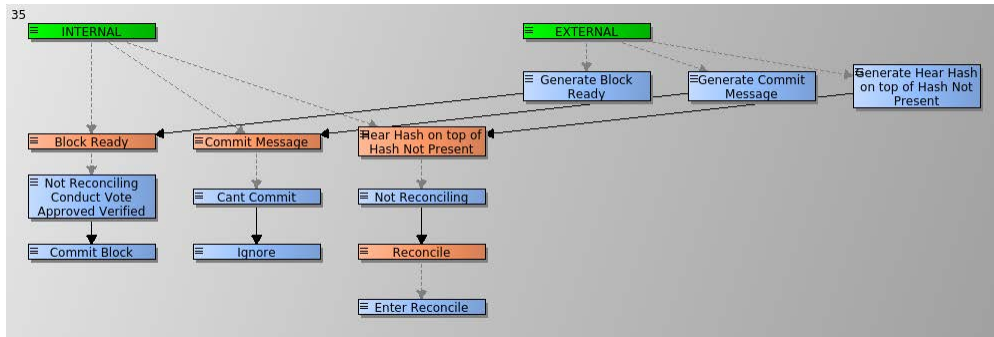


Figure B.35. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

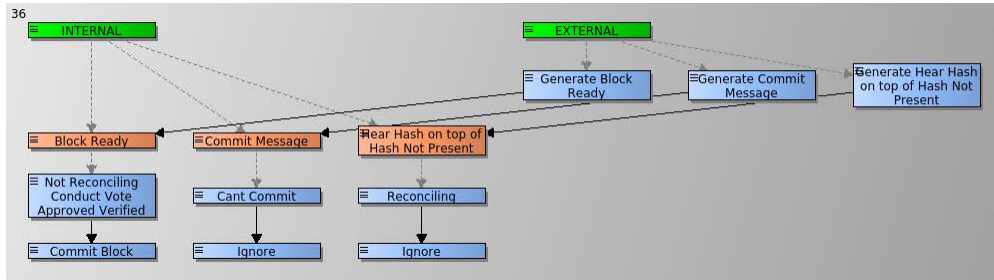


Figure B.36. This figure traces a scenario where a hash on top of a hash is heard, a block is determined to be ready, and a commit message is received. This behavior is intended.

APPENDIX C: Field Test Results

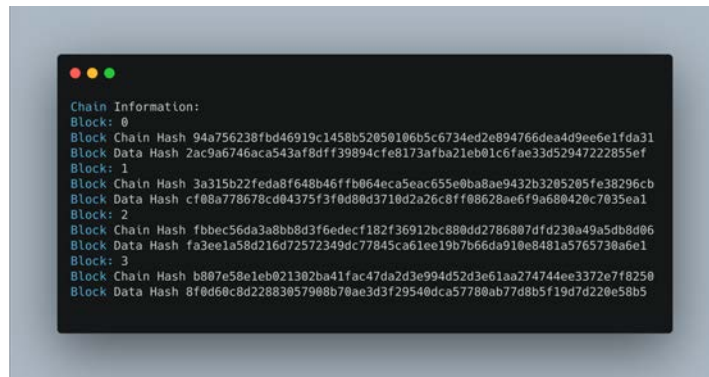
This appendix contains the results for the field tests conducted at Camp Roberts. The purpose of this appendix is to provide a concrete example of the results for the field test.

```
Chain Information:
Block: 0
Block Chain Hash 94a756238fbd46919c1458b52050106b5c6734ed2e094766dea4d9ee61fda31
Block Data Hash 2ac9a6746aca543af8df139894cfe8173afba21eb01c6fae33d5294722855ef
Block: 1
Block Chain Hash c2ec617dcca4505d31bcbf09f7845059289bb525e5d41e9b5ee6dfb313843d9
Block Data Hash 2c7bbfd1f59b513276c04bce5f350186630112271b19c1ea23c7ffde4ae7ed
Block: 2
Block Chain Hash 51b2c06e8095bc8c41d1b9fe9f8c3357f6d444009f48edac23c72d1d9081a0f4
Block Data Hash faaeac58f441d6b7eccfd95094140d122f57863b4ec84c17324198de377e549
Block: 3
Block Chain Hash 81a899ff559086a172011d199fbb79b9deeb3ab4feab0ea0638aa4eade26af3
Block Data Hash f0586c184282386190084832f91241e4d8fdd9844b9492e72a546b6b42805a84
Block: 4
Block Chain Hash d0680a9aa3f5946e81f44043019b0332225ebca9f7be6a9efff80e8773889281f
Block Data Hash cf08a778678cd04375f3f0d80d3710d2a26c8ff08628ae6f9a680420c7035ea1
Block: 5
Block Chain Hash 7741b158e9434333cc99811022f0a3901a2dcd01b1f217f2cb89368230c84e8
Block Data Hash fa3ee1a58d216d72572349dc77845ca61ee19b7b66da910e8481a5765730a6e1
Block: 6
Block Chain Hash 8cb2aa13d1e7657282be935da1dc1a0a83ef0f61ace631de777665cb71b51ef8
Block Data Hash 8f0d60c8d22883057908b70ae3d3f29540dca57780ab77d8b5f19d7d220e58b5
Block: 7
Block Chain Hash 67fd50ace5388967eb53e83883e00246f37a2491ddcd590625d6a00fcb9d95f1
Block Data Hash ff1ad74ef6e7aa828cc820ad3dc445515fc4fc41a7aedacf379610eb1caaaaf
Block: 8
Block Chain Hash 791df14b95a12e66faf6e096089ba52b3cb08d1fd002ad84d5f38fd8a31ee130
Block Data Hash 9eda3eb3a9b732c8e91e476431d09000dbb16bc4818656293075f5f75c98ee54
Block: 9
Block Chain Hash 22a870eaa9d74c8af7822718a1db7dd241a944726413819292bd90dc4240bcf
Block Data Hash bb23258f28a0583d5332b24fa2f6b793005b694103bf58c62e22926bc23b3f3
Block: 10
Block Chain Hash 3df45dc07969e75754ac18a739b9c93b1bd61ab6a1e00c934fae80db9f60a892
Block Data Hash a410af8eedbe1585e3ae0e98ec32b675ceae700831ab0b61784200c079411838
Block: 11
Block Chain Hash 2bd1eaab40965dad8ca3733ad329a8b48e78faf224d1ac823e8387cc96b8e68b
Block Data Hash cf38486d09521721c7d79f7a04d2b6d88e91d8c2376d5d361edd3351831c2
Block: 12
Block Chain Hash aebd28eccdc46c24c0ed7df53c6bda19b4c3bc70c5b245afd83b41a3b8ce0b9
Block Data Hash 40bca33dbfafb80b3cd5d90da7511b986a7f1269db315aab3dfdbbc22dc79bfc
Block: 13
Block Chain Hash 8bc5cf099fa18246a1218ed20638e695e1d4ef8ae89210b815e933cae9fc7b075
Block Data Hash 758dfc191fc223ddb50cccd9f8254afc2bd873cfe0a22b67157077a59cc257d
Block: 14
Block Chain Hash 95c793977e62ae4736936318d41a2f0aa643d1aba3361ada310dbdf82e7c70f1
Block Data Hash f639abf1634d3d45668a2db98289ec545e26fd0acaf9f62d47515888ace748
Block: 15
Block Chain Hash ced58de12d2978b349da10b14d1d775e3f26945209e7b5190997c5794acb4125
Block Data Hash ea7ec4208ef077a46b6284350ef6c7e54ca54681f232a75b3980c525b87021f7
Block: 16
Block Chain Hash 973197c063919ef07861f800fb659064833c36bdea3e9ef89b5fdd35fd9a6c0e
Block Data Hash 7022b0da37a2c527aee429f3c32626b3b401d55c8abb4060af3d9b2e45923f3
Block: 17
Block Chain Hash 5cf2eab7147e6f4fcb502bba468155e7b4bc51a8abafcb3e89e4e96cc13e356
Block Data Hash eb820883d6d345fc855aa58c48415005957e19d730c9ae36d935a47e7a97c8e9
Block: 18
Block Chain Hash 3c05c96448b76a4d279d87fe27fe00551c5f381382e0c80af0517755a521105
Block Data Hash db60c50fac14d9e30efbbc151366a5213cd8d3582ccce02452bbdf6364a795
Block: 19
Block Chain Hash 32cda31690682d47f5a8baef40aebc7bf7e5854968bf5c965de8354692872a5
Block Data Hash 5600d814f8d3998f68504eadae43c5934c164dea36c3a413affea391799a5ca4
Block: 20
Block Chain Hash 97716d959bf9e6b758c5646b2f8563c0ce475ec99f716fbaf79fcc840e1f14a2
Block Data Hash 4b13a8db911dab64d8e4b5fb89dfe6858d05bf927384e430ac9eb6d494ee2cc1
Block: 21
Block Chain Hash 128e12850559507fd8b5dd14a3fe9fb6a8ac03e346ec5274a426583624ff0d5
Block Data Hash 59fb19e6859bb59c551865163e61b0247049a9f254980beaae65edbd5bf37519
Block: 22
Block Chain Hash d1d2e69de5f8aea8fc78168e4bedf855efa8b9aaac0c9d47485506a48bc40c67
Block Data Hash 797b11dc362c0de937c79b20fd6d78a9d3ec5a4104cd946475612b893012c121
Block: 23
Block Chain Hash 61ca7dcd1c8291d8f709201bf8c890590b30a88b8494e4a006df1d2e747f2c91
Block Data Hash 56b274f02a0f2f853650137bffd7db3b45feea6dd5d1eb0d6f63175a1fd62a6
Block: 24
Block Chain Hash 4181fd5c97d16d02a1e2f7463a9601b2160635931af1268d65997d0d4fab6f4
Block Data Hash c9d07e2da165fd9cc71b33c75db0cb9975e3d8f4dbd198f67d1173c0590f26b
Block: 25
Block Chain Hash 72326ea1bf305f3baae03c1a6279ff9f40d92e20723157bd3e5c8e29b665b78
Block Data Hash 25daea5493f6f935b37e6821ef8e3e40f3f54335f77242be10fc74829d935ac
Block: 26
Block Chain Hash 86246a34192342c0d14ef11fcfb8928c539a05fbd1fe121f51442122727ce96
Block Data Hash bf7972180b4644bb9a9610e807ada83ddf4583ecbd02cf03da8cbaa5855bf6
Block: 27
Block Chain Hash c85bb75ac69b6f904a6fa3d14e1725c231825ddbc2a4c54ee292b1be9b0652f3
Block Data Hash c641017d33daf2e40c907067cf5252dfdf77bd4c264cc31d90cbe36dbc9fec3
```

Figure C.1. The block chain from the UAV with ID number 10.

```
Chain Information:
Block: 0
Block Chain Hash 94a756238fbd46919c1458b52050106b5c6734ed2e894766dea4d9ee61fda31
Block Data Hash 2ac9a6746aca543af8df139894cfe8173afba21eb01c6fae33d5294722855ef
Block: 1
Block Chain Hash c2ec617dcca4595d31bcf09f78450559289bb525e5d41e9b5ee6dfb313843d9
Block Data Hash 2c7bbfd1f59b513276c84bce5f350186630112271b19c1ea23c7ffde4ae7ed
Block: 2
Block Chain Hash 51b2c06e8095bc8c41d1b9fe9f8c3357f6d444099f48edac23c72d1d9081a0f4
Block Data Hash faaeac58f441d6b7eccfd95894140d122f57863b4ec84c17324198de377e549
Block: 3
Block Chain Hash 81a899ff559086a172011d199fbb79b9deeb3ab4feab0ea0638aa4eade26af3
Block Data Hash f0586c1847282386190084832f91241e4d8fdd9844b9492e72a546b6b42805a84
Block: 4
Block Chain Hash d0680a9aa3f5946e81f44043019b0332225ebca9f7be6a9efff80e8773889281f
Block Data Hash cf08a778678cd84375f3f0d80d3710d2a26c8ff08628ae6f9a680420c7035ea1
Block: 5
Block Chain Hash 7741b158e9434333cc99811022f0a3901a2dcd01b1f217f2cb89368230c84e8
Block Data Hash fa3ee1a58d216d72572349dc77845ca61ee19b7b66da910e8481a5765730a6e1
Block: 6
Block Chain Hash 8cb2aad131e7657282be935da1dc1a0a83ef0f61ace631de777665cb71b51ef8
Block Data Hash 8f0d06c8d22883057908b70ae3d3f29540dca57780ab77d8b5f19d7d220e58b5
Block: 7
Block Chain Hash 67fd50ace5388967eb53e83883e00246f37a2491ddcd590625d6a00fcb95f1
Block Data Hash ff1ad74ef6e7aa828cc820ad3dc445515fc4fc41a7aedacf379610eb1caaf
Block: 8
Block Chain Hash 791df14b95a12e66faf6e096089ba52b3cb08d1f002ad84d5f38fd8a31ee130
Block Data Hash 9eda3eb3a9b632c8e91e476431d09000dbb16bc4818656293075f5f75c98ee54
Block: 9
Block Chain Hash 22a070eaa9d74c8af7822718a1db7dd241a944726413819292b9d0dc240bcf
Block Data Hash bb23258f28a0583d5332b24fa2f6b793005b694103bf58c62e22926bc23b3f3
Block: 10
Block Chain Hash 3df45dc07969e75754ac18a739b9c93b1bd61ab6a1e00c934fae80db9f60a892
Block Data Hash a410af8eedbe1585e3ae0e98ec32b675ceae700831ab0b61784200c079411838
Block: 11
Block Chain Hash 2bd1eaa40965dad8ca3733ad329a8b48e78faf224d1ac823e8387cc96b8e68b
Block Data Hash cf38486d09521721c7d797a04d2b6d88e91d8c2376d5d361edd3351831c2
Block: 12
Block Chain Hash aebd28eccdc46c24c0ed7df53c6bda19b4c3bc70c5b245afd83b41a3b8ce0b9
Block Data Hash 40bca33dbfafb80b3cd5d90da7511b986a7f1269db315aab3dfdbbc22dc79bfcc
Block: 13
Block Chain Hash 8b5cf909fa18246a1218ed20638e695e1d4ef8ae89210b815e933cae9fc7b075
Block Data Hash 758dfc191fc223ddb50cccd9f8254afc2bd873cfe0a22b67157077a59cc257d
Block: 14
Block Chain Hash 95c793977e62ae4736936318d41a2f0aa643d1aba3361ada310dbdf82e7c70f1
Block Data Hash f639abf1634d3d45668a2db98289ec545e26fd0acaf9f62d47515888ace748
Block: 15
Block Chain Hash ced58de12d2978b349da10b14d1d775e3f26945209e7b5190997c5794acb4125
Block Data Hash ea7ec4208ef077a46b6284350ef6c7e54ca54681f232a75b3980c525b87021f7
Block: 16
Block Chain Hash 973197c063919ef07861f800fb659064833c36bdea3e9ef89b5fdd35fd9a6c0e
Block Data Hash 7022b0da37a2c327aee429f3c32626b3b401d55c8abb406af3d9b2e45923f93
Block: 17
Block Chain Hash 5cf2eab7147e6f4fcb502bba468155e7b4bc51a8abafcb3e89e4e96cc13e356
Block Data Hash eb820883d6d345f855aa58c48415005957e19d730c9ae36d935a47a7e97c8e9
Block: 18
Block Chain Hash 3c05c96448b76a4d279d87fe27fe00551c5f381382e0c80af051775a5521105
Block Data Hash db60c50fac14d9e30efbbc151366a5213dc8d3582ccce02452bbdf6364a795
Block: 19
Block Chain Hash 32cda31690682d47f5a8baef40aebc7bf7e5854968bf5c965de8354692872a5
Block Data Hash 5600d814f8d3998f68504eadae43c5934c164dea36c3a413affea391799a5ca4
Block: 20
Block Chain Hash 97716d959bf9e6b758c5646b2f8563c0ce475ec99f716fbaf79fcc840e1f14a2
Block Data Hash 4b13a8db911dab64d8e4b5fb89dfe6858d05bf927384e430ac9eb6d494ee2cc1
Block: 21
Block Chain Hash 128e12850559507fd8b5dd14a3fe9fb6a8ac03e346ec5274a426583624ff0d5
Block Data Hash 59fb19e6859bb59c551865163e61b0247049a9f254980beaae65edbd5bf37519
Block: 22
Block Chain Hash d1d2e69de5f8aea8fc78168e4bedf855efa8b9aaaa0c9d47485506a48bc40c67
Block Data Hash 797b11dc362c0de937c79b20fd6d78a9d3ec5a4104cd946475612b893012c121
Block: 23
Block Chain Hash 61ca7dcd1c8291d8f709201bf8c890590b30a88b8494e4a006df1d2e747f2c91
Block Data Hash 56b274f02a0f2f853650137bffd7db3b45feea6d5d1eb0d6f63175a1fd62a6
Block: 24
Block Chain Hash 4181fd5c97d16d02a1e2f7463a9601b2160635931af1268d65997d0d46fab6f4
Block Data Hash c9d07e2da165fd9cc71b33c75db0cb9975e3d8f4dbd198f67d1173c0590f26b
Block: 25
Block Chain Hash 72326ea1bf305ffb3aae03c1a6279ff9f40d92e20723157bd3e5c8e29b665b78
Block Data Hash 25daea5493f6f935b37e6821ef08e3e40f3f54335f77242be10fc74829d935ac
Block: 26
Block Chain Hash 86246a34192342c0d14ef11fcfb8928c539a05fbd1fe121f51442122727ce96
Block Data Hash bf7972180b4644bb9a9610e8007ada83ddf4583ecbd02cf03da8cbaa5855bf6
Block: 27
Block Chain Hash c85bb75ac69b6f904a6fa3d14e1725c231825ddbc2a4c54ee292b1be9b0652f3
Block Data Hash c641017d33daf2e40c907067cf5252d1df77bd4c264cc31d90cbe36dbc9fec3
```

Figure C.2. The block chain from the UAV with ID number 28.

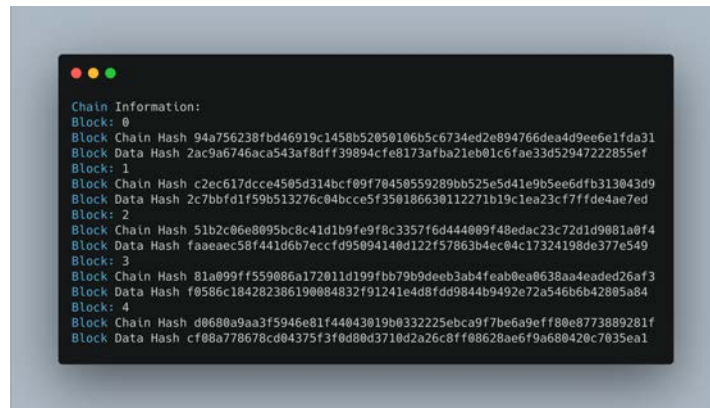


```
Chain Information:
Block: 0
Block Chain Hash 94a756238fbd46919c1458b52050106b5c6734ed2e894766dea4d9ee6e1fda31
Block Data Hash 2ac9a6746aca543af8dff39894cfe8173afba21eb01c6fae33d5294722855ef
Block: 1
Block Chain Hash 3a315b22feda8f648b46ffbb64eca5eac655e0ba8ae9432b3205205fe38296cb
Block Data Hash cf08a778678cd04375f3f0d80d3710d2a26c8ff08628ae6f9a680420c7035ea1
Block: 2
Block Chain Hash fbbec56da3a8bb8d3f6edecf182f36912bc880dd2786807dfd230a49a5db8d06
Block Data Hash fa3ee1a58d216d72572349dc77845ca61ee19b7b66da910e8481a5765730a6e1
Block: 3
Block Chain Hash b807e58e1eb021302ba41fac47da2d3e994d52d3e61aa274744ee3372e7f0250
Block Data Hash 8f0d60c8d22883057908b70ae3d3f29540dca57780ab77d8b5f19d7d220e58b5
```

Figure C.3. The block chain from the UAV with ID number 32. This UAV did not function properly.

```
Chain Information:
Block: 0
Block Chain Hash 94a756238fbd46919c1458b52050106b5c6734ed2e094766dea4d9ee61fda31
Block Data Hash 2ac9a6746aca543af8df139894cfe8173afba21eb01c6fae33d5294722855ef
Block: 1
Block Chain Hash c2ec617dcca4505d31bcf09f78450559289bb525e5d41e9b5ee6dfb313843d9
Block Data Hash 2c7bbfd1f59b513276c04bce5f350186630112271b19c1ea23c7ffde4ae7ed
Block: 2
Block Chain Hash 51b2c06e8095bc8c41d1b9fe9f8c3357f6d444009f48edac23c72d1d9081a0f4
Block Data Hash faaeac58f441d6b7eccfd95094140d122f57863b4ec84c17324198de377e549
Block: 3
Block Chain Hash 81a099ff559086a172011d199fbb79b9deeb3ab4feab0ea0638aa4eade26af3
Block Data Hash f0586c184282386190084832f91241e4d8fdd9844b9492e72a546b6b42805a84
Block: 4
Block Chain Hash d0680a9aa3f5946e81f44043019b0332225ebca9f7b6a9efff80e8773889281f
Block Data Hash cf08a778678cd04375f3f0d80d3710d2a26c8ff08628ae6f9a680420c7035ea1
Block: 5
Block Chain Hash 7741b158e9434333cc99811022f0a3901a2dcd0b1b1f217f2cb89368230c84e8
Block Data Hash fa3ee1a58d216d72572349dc77845ca61ee19b7b66da910e8481a5765730a6e1
Block: 6
Block Chain Hash 8cb2aa13d1e7657282be935da1dc1a0a83ef0f61ace631de777665cb71b51ef8
Block Data Hash 8f0d60c8d22883057908b70ae3d3f29540dca57780ab77d8b5f19d7d220e58b5
Block: 7
Block Chain Hash 67fd50ace5388967eb53e83883e00246f37a2491ddcd590625d6a00fcb9d95f1
Block Data Hash ff1ad74ef6e7aa828cc820adc3dc445515fc4fc41a7aedac379610eb1caaf
Block: 8
Block Chain Hash 791df14b95a12e66faf6e096089ba52b3cb08d1fd002ad84d5f38fd8a31ee130
Block Data Hash 9eda3eb3a9bf32c8e91e476431d09000dbb16bc4818656293075f5f75c98ee54
Block: 9
Block Chain Hash 22a070eaa9d74c8af7822718a1db7dd241a944726413019292bd90d4c240bcf
Block Data Hash bb23258f28a0583d5332b24fa2f6b793005b694103bf58c62e22926bc23b3f3
Block: 10
Block Chain Hash 3df45dc07969e75754ac18a739b9c93b1bd61ab6a1e00c934fae80db9f60a892
Block Data Hash a410af8eedbe1585e3ae0e98ec32b675ceae700831ab0b61784200c079411838
Block: 11
Block Chain Hash 2bd1eaab40965dad8ca3733ad329a8b48e78faf224d1ac823e8387cc96b8e68b
Block Data Hash cf38486d09521721c7d79f04d2b6d88e91d8c2376d5d361edd3351831c2
Block: 12
Block Chain Hash aebd28eccdc46c24c0ed7df53c6bda1a9b4c3bc70c5b245afd83b41a3b8ce0b9
Block Data Hash 40bca33dbfafb80b3cd5d90da7511b986a7f1269db315aab3dfdbbc22dc79bfc
Block: 13
Block Chain Hash 8b5cf909fa18246a1218ed20638e695e1d4ef8ae89210b815e933cae9fc7b075
Block Data Hash 758dfc191fc223ddb50cccd9f8254afc2bd873cfe0a22b671575077a59cc257d
Block: 14
Block Chain Hash 95c793977e62ae4736936318d41a2f0aa643d1aba3361ada310dbdf82e7c70f1
Block Data Hash f639abf1634d3d45668a2d2b98289ec545e26fd0acaf9f62d47515888ace748
Block: 15
Block Chain Hash ced58de12d2978b349da10b14d1d775e3f26945209e7b5190997c5794acb4125
Block Data Hash ea7ec4208ef077a46b6284350ef6c7e54ca54681f232a75b3980c525b87021f7
Block: 16
Block Chain Hash 973197c063919ef07861f800fb659064833c36bdea3e9ef89b5fdd35fd9a6c0e
Block Data Hash 7022b0da37a2c527aee429f3c32626b3b401d55c8abb4060af3d9b2e45923f93
Block: 17
Block Chain Hash 5cf2eab7147e6f4fb502bba468155e7b4bc51a8abafcb3e89e4e96cc13e356
Block Data Hash eb820883d6d345fc855aa58c48415005957e19d730c9ae36d935a47e7a97c8e9
Block: 18
Block Chain Hash 3c05c96448b76a4d279d87fe27fe00551c5f381382e0c80af0517755a5521105
Block Data Hash db06c50fac14d9e30efbbc151366a5213dc8d3582ccce02452bbdf6364a795
Block: 19
Block Chain Hash 32cda31690682d47f5a8baef40aebc7bf7e5854968bf5c965de8354692872a5
Block Data Hash 5600d814f8d3998f68504eadae43c5934c164dea36c3a413affea391799a5ca4
Block: 20
Block Chain Hash 97716d959bf9e6b758c5646b2f8563c0ce475ec99f716fbaf79fcc840e1f14a2
Block Data Hash 4b13a8db911dab64d8e4b5fb9dfe6858d05bf927384e430ac9eb6d494ee2cc1
Block: 21
Block Chain Hash 128e12850559507fd8b5dd14a3fe9fb6a8acbb3e346ec5274a426583624ff0d5
Block Data Hash 59fb19e6859bb59c551865163e61b0247049a9f254980beaae65edbd5bf37519
Block: 22
Block Chain Hash d1d2e69de5f8aea8fc78168e4bedf855efa8b9aaac0c9d47485506a48bc40c67
Block Data Hash 797b11dc362c0de937c79b20fd6d78a9d3ec5a4104cd946475612b893012c121
```

Figure C.4. The block chain from the UAV with ID number 120.



```
Chain Information:
Block: 0
Block Chain Hash 94a756238fbd46919c1458b52050106b5c6734ed2e894766dea4d9ee6e1fda31
Block Data Hash 2ac9a6746aca543af8dff39894cfe8173afba21eb01c6fae33d5294722855ef
Block: 1
Block Chain Hash c2ec617dce4505d314bcf09f70450559209bb525a5d41e9b5ee6dfb313043d9
Block Data Hash 2c7bbfd1f59b513276c04bce5f350186630112271b19clea23cf7ffde4ae7ed
Block: 2
Block Chain Hash 51b2c06e0095bc8c41d1b9fe9f8c3357f6d444009f48edac23c72d1d9001a0f4
Block Data Hash faaeac58f441d6b7eccfd95094140d122f57863b4ec04c17324198de377e549
Block: 3
Block Chain Hash 01a099ff559086a172011d199fbb79b9deeb3ab4feab0ea0638aa4eade26af3
Block Data Hash f0586c184282386190084832f91241e4d8fdd9844b9492e72a546b6b42805a84
Block: 4
Block Chain Hash d0680a9aa3f5946e81f44043019b033225ebca9f7be6a9eff00e8773889281f
Block Data Hash cf08a778678cd04375f3f0d80d3710d2a26c8ff08628ae6f9a680420c7035ea1
```

Figure C.5. The block chain from the UAV with ID number 122. This UAV did not function properly.

List of References

- [1] C. Majaski. Distributed Ledgers Definition. Accessed May 10, 2020. [Online]. Available: <https://www.investopedia.com/terms/d/distributed-ledgers.asp>
- [2] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev, “Trade-offs between distributed ledger technology characteristics,” *ACM Computer Survey*, vol. 53, no. 2, May 2020.
- [3] M. Auguston. “About,” Monterey Phoenix, NPS Wiki. Naval Postgraduate School. Accessed on Aug. 26, 2020. [Online]. Available: <https://wiki.nps.edu/display/MP/About>
- [4] J. Mattis. Summary of the 2018 National Defense Strategy. Department of Defense. 2018. [Online]. Available: <https://dod.defense.gov/Portals/1/Documents/pubs/2018-National-Defense-Strategy-Summary.pdf>
- [5] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain technology overview,” National Institute of Standards and Technology, Oct. 2018. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf>
- [6] A. T. Sherman, F. Javani, H. Zhang, and E. Golaszewski, “On the origins and variations of blockchain technologies,” *IEEE Security Privacy*, vol. 17, no. 1, pp. 72–77, 2019.
- [7] D. Burkhardt, M. Werling, and H. Lasi, “Distributed ledger,” in *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, 2018, pp. 1–9.
- [8] “Nara blockchain white paper,” National Archives and Records Administration, Feb. 2019. [Online]. Available: <https://www.archives.gov/files/records-mgmt/policy/nara-blockchain-whitepaper.pdf>
- [9] S. Nakamoto, “bitcoin.pdf,” 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>
- [10] M. Salad. md5 Hash Generator. Accessed on Aug. 18, 2020. [Online]. Available: <https://www.miraclesalad.com/webtools/md5.php>
- [11] “Consensus”. Merriam-Webster. Accessed on Aug. 19, 2020. [Online]. Available: <https://www.merriam-webster.com/dictionary/consensus>
- [12] L. Lamport, “Paxos made simple,” Nov. 2001. [Online]. Available: <https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>

- [13] O. Etzion, “Correctness and consistency of event-based systems,” in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems* (DEBS ’19). New York, NY, USA: Association for Computing Machinery, 2019, p. 197–198. [Online]. Available: <https://doi-org.libproxy.nps.edu/10.1145/3328905.3332463>
- [14] M. Auguston. “Documentation,” Monterey Phoenix, NPS Wiki. Naval Postgraduate School. Accessed on Aug. 26, 2020. [Online]. Available: <https://wiki.nps.edu/display/MP/Documentation>
- [15] B. Allen. Monterey Phoenix / User Interfaces / Gryphon · GitLab. Accessed Nov. 1, 2020. [Online]. Available: <https://gitlab.nps.edu/monterey-phoenix/user-interfaces/gryphon/>
- [16] S. Böhne and C. Kreitz, “Learning how to prove: From the coq proof assistant to textbook style,” in *Proceedings 6th International Workshop on Theorem Proving Components for Educational Software* (EPTCS), P. Quaresma and W. Neuper, Eds., 2017, vol. 267, pp. 1–18.

Initial Distribution List

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California