UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Haddock: A Smart-Contract Command Bus for the Fishing Industry

Sivert Jakob Steinholt

INF-3981, Master thesis in Computer Science, June 2024

Supervisor

Main supervisor:	Håvard Dagenborg	UiT The Arctic University of Norway, Faculty of Science and Technology, Department of Computer Science

This thesis document was typeset using the *UiT Thesis LaTeX Template*.
© 2024 – `http://github.com/egraff/uit-thesis`

Abstract

The global fishing industry, a critical food source, faces significant challenges due to criminal activities such as illegal fishing and over-exploitation. Traditional surveillance methods can be susceptible to tampering and cannot fully ensure the integrity of recorded events. This thesis introduces Haddock, a shared, distributed logging system leveraging a two-phase dissemination protocol and the Avalanche blockchain network to provide a secure and immutable record of log data. By utilizing blockchain system's inherent properties of transparency, immutability, and decentralized nature, Haddock aims to enhance the reliability and trustworthiness of maritime surveillance systems.

Through experiments and evaluations, this thesis demonstrates the potential of using blockchain technology for logging and timely dissemination of messages within systems that tolerate a latency of a few seconds.

Acknowledgements

I extend my sincere gratitude to Professor Håvard Dagenborg, my supervisor, for valuable discussions, guidance, and support throughout this semester. I deeply appreciate you taking the time to give last-minute inputs, as they have proven to be invaluable for the success of this thesis

I would like to give a heartfelt thanks to my beloved friends and co-students, "Guttebassene". Thank you for the amazing friendship and years at the university. Additionally, my warmest thanks to my mom, dad, two brothers, and grandma for their continuous, loving support and encouragement throughout my study years.

I want to thank my friend and co-student Henning Hageli for his insightful discussions on the subject of blockchain technology and for his enthusiasm and motivation.

Throughout the thesis writing process, large language models like ChatGPT have been used for spelling, refinement, and feedback on the sections.

Contents

Abstract i

Acknowledgements iii

List of Figures ix

List of Tables xi

List of Abbreviations xiii

1 Introduction 1
- 1.1 Problem Definition . . . 2
- 1.2 Methodology . . . 3
- 1.3 Scope and Limitations . . . 4
- 1.4 Context . . . 5
- 1.5 Outline . . . 5

2 Background 7
- 2.1 Blockchain . . . 7
- 2.2 Types of Blockchains . . . 8
- 2.3 Ethereum . . . 10
- 2.4 Avalanche . . . 11
- 2.5 Decentralized Applications (DApps) . . . 12
- 2.6 REST API . . . 14
- 2.7 Cryptography . . . 15
- 2.8 Summary . . . 17

3 Requirements 19
- 3.1 Functional Requirements . . . 19
- 3.2 Non-Functional Requirements . . . 21
  - 3.2.1 Reliability . . . 21
  - 3.2.2 Responsiveness and Availability . . . 21
  - 3.2.3 Security and Resilience . . . 22
  - 3.2.4 Usability . . . 22

3.2.5 Maintainability 22
3.3 Summary 23

4 Design & Implementation 25
4.1 System Overview 26
4.2 Haddock's Workflow 27
4.3 Server-Side Components 28
4.3.1 Disseminate Message 28
4.3.2 Store Acknowledgments for Encrypted Messages . . . 29
4.3.3 Dissmeniate Key 31
4.4 Client-side Components 32
4.4.1 Contract Interaction Tools 32
4.4.2 REST API 33
4.4.3 Client Services 35
4.4.4 Contract Services 36
4.4.5 Frontend 37
4.4.6 Frontend Message Dissemination 38
4.4.7 Frontend Message Acknowledgment 40
4.5 Synchronization Options 41
4.6 Summary 42

5 Evaluation 45
5.1 Platform 45
5.2 Experiment Setup and System Specifications 46
5.3 Dissemination Experiment 47
5.4 Cost Analysis Experiment 50
5.5 Memory Growth Experiment 51
5.6 Summary 54

6 Discussion 55
6.1 Solely Key Distribution 55
6.2 Subnets and Customized Access Control 56
6.3 Single Struct Approach 56
6.4 Discussion of Non-Functional Requirements 57
6.5 Summary 59

7 Concluding Remarks 61
7.1 Contributions and Goals 61
7.2 Related Work 63
7.3 Concluding Remarks 64
7.4 Future Work 65

Bibliography 67

A Appendix A **71**

List of Figures

2.1 Blockchains stores transactions in blocks that are linked together by hashes. Inspired by Figure 1 [25] 9

4.1 An architectural overview of Haddock 26
4.2 A complete view of the Haddock user interface. 37
4.3 Haddock features a simple navigation bar. A button can be used to connect the crypto wallet to the system. 38
4.4 The publisher component includes an input text form and a button for message dissemination, a button, and a slider used to set the required acknowledgments. Lastly, a button is used to force push the decryption key to the network. 39
4.5 The transaction window from the Metamask browser extension. This box is used by the client to sign the transaction. . . 39
4.6 A display of the incomplete messages. The key in the left corner is a button to acknowledge the message. The numbers to the left of it display the required number of acknowledgments before the decryption key is disseminated. 40
4.7 A display of all complete messages. The verification mark down left in the message panel opens a modal with a list of all signers of the current message. The number to the left of the verification mark displays the number of signers. 41
4.8 The full list of singers for a given complete message. Their address, hash of the encrypted data, and a verification mark are included in the modal. 42

5.1 A bicep script deploys 30 public container instances to Azure at different regions across Northern America. Each container instance consists of a dissemination test wrapped inside a Docker container. Finally, a Python script calls the API endpoints to start and fetch test results. 47
5.2 The mean retrieval time for all 30 nodes spread across the globe. The plot is based on the table in appendix A.8 48
5.3 The time window from the first received data to the last received data and the mean retrieval time for each byte size . . 49

5.4 The increasing cost of publishing encrypted data at various byte sizes. 51
5.5 The increasing cost of disseminating the decryption key at various byte sizes. This cost will always be higher than the cost of disseminating encrypted messages. 51
5.6 The increasing cost of acknowledging data at various byte sizes. 52
5.7 The increasing gas cost of publishing encrypted data (a) and decryption key (b) at various byte sizes, as the number of data entries accumulates 53
5.8 The cost of acknowledging messages at various byte sizes as the number of data entries accumulates. The opaque plot illustrates the confidence interval for each iteration 54

List of Tables

4.1 Listing of all endpoints exposed by the client-side API service 34

5.1 System Specifications 46
5.2 System Specifications for each Azure container instance . . . 46
5.3 The cost of each function on a new empty SHIELD contract . 46

A.1 Data retrieval time in milliseconds at different sizes for each node. Each time entry is the mean value of three separate dissemination tests 73
A.2 List of Server Regions 74
A.3 Data retrieval time in milliseconds at 20 bytes with three tests 75
A.4 Data retrieval time in milliseconds at 40 bytes with three tests 76
A.5 Data retrieval time in milliseconds at 60 bytes with three test 77
A.6 Data retrieval time in milliseconds at 100 bytes with three test 78
A.7 Data retrieval time in milliseconds at 1000 bytes with three tests 79
A.8 Data retrieval time in milliseconds at 5000 bytes with three tests 80
A.9 The cost analysis of the acknowledgement function, with three tests 81
A.10 The cost analysis of the data publication function, with three tests 82
A.11 The cost analysis of the key publication function, with three tests 83

List of Abbreviations

ABI Application Binary Interface

AES Advanced Encryption Standard

AIS Automatic Identification System

API Application Interface

C-CHAIN Contract Chain

CBC Cipher-Block Chaining

CDN Content Deliver Network

CFB Cipher Feedback

CSG Cyber Security Group

DApp Decentralized Application

ETH Ether

EVM Ethereum Virtual Machine

HTTP Hypertext Transfer Protocol

IoD Internet of Drones

IV Initialization Vector

JS JavaScript

JSON JavaScript Object Notation

LCaaS Logchain-as-a-service

P2P Peer-to-Peer

PoS Proof-of-Stake

PoW Proof-of-Work

RBAC Role-Based Access Control

REST Representational State Transfer

RPC Remote Procedure Call

SHA-2 Secure Hash Algorithm 2

SHIELD Secure Handling of Information with Encrypted Logs and Decryption Key

SSE Server Sent Events

TTF Time-to-Finality

UI User Interface

UX User Experience

XOR Exclusive-Or

1 Introduction

The fishery industry provides one of the most significant sources of protein worldwide. Edible fish originates from wild fisheries and ocean-farmed species. They are contributing to 17% of the global edible meat production [9] with the number of fishing vessels more than doubled globally since 1950. The widespread adoption of motorization in most vessels has notably enhanced the efficiency of fishing operations and is primarily a consequence of advances in the industrial sector. In 2015, the total number of motorized fishing fleets was estimated to be 68% [31]. The ever-increasing demand for fishing products, alongside significant advancements in the industrial fishing sector, has elevated the rate of overfishing, threatening the global marine ecosystem. For instance, the world's Bluefin tuna and Swordfish population has decreased by 80% in just 5 five years [6]. In addition, the fishing sector is threatened by organized and transnational criminal activities. Encompassing activities like illegal fishing, money laundering, document forgery, and drug trafficking [39].

Controlling critical information in the communication channels connecting boats and land-based control centers has been suggested as fundamental for preventing criminal activities at sea and preventing overfishing [26, 3]. Modern surveillance of ship traffic, fishing, suspicious activities, etc., is done with a combination of Automatic Identification System (AIS), black-boxed surveillance tools for event recording, and satellite imagery, especially Synthetic-Aperture Radar (SAR), which utilizes radar to capture images through cloudy environment [3, 27]. On-board equipment like AIS and black-boxes is at risk of tampering and cannot be deemed fully reliable.

Reconstructing the order of events with evidence of high integrity is crucial for pursuing and suppressing criminal activities. Existing tools are susceptible to failure due to tampering. A public logging infrastructure, recording on-ship events and which information on fleet command centers act on, alongside timely dissemination of business-critical information, can, in combination with existing surveillance architecture, be part of a larger system fighting large-scale coordinated criminal activities.

Blockchains offer communication with a high degree of non-repudiation and offer mechanisms that can be used to keep an ordered log of events distributed and consistent in a network consisting of fully or partially mistrusting entities. All information can be stored publicly on a blockchain with low risk for potential tampering. Blockchains have an architectural design that naturally correlates with a logging structure in which all recordings are stored, enabling access to the complete blockchain history at any moment. Requiring data to be public to a larger degree can have positive effects when fighting crime. Making certain records public can increase the availability of the data, whereby the detection of anomalies can be more apparent. Blockchains are not fully resilient to network attacks but can be used reliably [33]. Participating in the blockchain network requires the adversary to spend money transactions and will naturally increase the threshold for engaging in network attacks.

Smart contracts operate on blockchain systems and automate the execution of code. Once deployed, they remain immutable and have a high degree of tamper resistance. The immutability combined with tamper resistance yields the property of non-repudiation, in which performed actions recorded on the platform cannot be disowned or denied by the parties involved. Public POW-based blockchains, like Bitcoin [25], are too slow and unreliable [36] and do not have a satisfactory transaction throughput to support a high workload at a reasonable cost, which is required by a distributed logging system. Newer, more contemporary blockchain networks like Ethereum [7] and Avalanche [30] are better suited for this purpose.

1.1 Problem Definition

Suppressing fishery-related criminal activities that occur out on the open sea is a complex and problematic challenge. Applying a blockchain network for its immutability, transparency, and distributed properties may prove valuable when logging on-ship events and timely disseminating fleet command center reports. A secure, distributed logging infrastructure may be helpful as a component in a larger system composed of modern surveillance tools.

Our thesis is that

A secure, shared logging infrastructure for fishery fleet command can be implemented using Blockchain smart contracts.

In this thesis, we propose Haddock: a shared, distributed logging service, utilizing the public blockchain network *Avalanche* as a backend service[1]. To investigate our thesis, we devise and implement Haddock: A decentralized logging infrastructure. Haddock is built and designed as a Decentralized Application (DApp), which combines a two-phase dissemination protocol with a barrier synchronization mechanism that records and logs the publisher of messages, as well as the initial readers. This is done with a smart contract and client-side software that interacts with and interprets the logged data. Haddock acts as a distributed synchronizer for arbitrary data types and multiple severity levels. Haddock, arguably, may not be used for the highest form of severities since, first of all, the critical need for access to the internet poses a persistent challenge, particularly in open sea environments, secondly, because of the natural latency within contemporary blockchain networks.

1.2 Methodology

In 1989 the Task Force of the Core of Computer Science tackled the debate on computer science's inclusion in the world of science and engineering. The group introduces a framework for the discipline of computing [11]. Outlining three distinctive paradigms - theory, abstraction, and design - for approaching the discipline of computing.

The first paradigm for the discipline is the theory paradigm. It is rooted in the world of mathematics. Thus, by adhering to a mathematical approach, the objects of study are explicitly stated, forming a comprehensive definition. Further, the relationship between the objects can be hypothesized, forming a theorem. Finally, an examination is conducted to determine if the relationship between the objects holds true in practice. In the theory paradigm, studies revolving around algorithms can be conducted. A problem can be defined around computer complexities, e.g., finding the quickest way to sort a list of massive numbers. Algorithms can be created and tested to solve a uniform problem. Through theoretical analysis, the results can be used to find an optimal solution and the best-suited algorithm for the given problem.

1. The system is named *Haddock* after the fish from the cod family and a somewhat known seafaring captain.

The second paradigm within the discipline is the abstraction paradigm, which finds its foundation in the experimental scientific method. Following the paradigm, a hypothesis revolving around the area of interest is formed. The hypothesis is then used to model a model that coheres to the proposition and makes predictions of the model. Experiments are conducted to collect data from the model. Through analysis, the data can be interpreted, whether it supports or disapproves the hypothesis, thereby either falsifying or strengthening the proposition made.

The third and last paradigm within the discipline of computer science is the design paradigm. This paradigm follows an engineering approach. First, the system's requirements need to be stated. Then, specify details of how the system will meet the requirements, and design and implement the system accordingly. The system is then thoroughly tested to check if it meets the initial requirements.

This master's thesis mainly builds on the discipline of design. The thesis has a problem definition that states and specifies the system's initial requirements. A demonstrator of the system will be designed and implemented to fulfill the requirements derived from the problem definition. In addition, we will conduct experiments to ensure that the system's functionality meets the initial requirements.

1.3 Scope and Limitations

Haddock has two distinct and challenging limitations.

1. Ships and vessels need access to the Internet for the system to work as intended.

2. It is not possible to record all readers of data in a fully public system, as the decentralized nature of the system makes it challenging to identify and store readers without a transaction to record.

We assume that ships and vessels have internet access when using the system. Further, we assume we cannot record all entities who read the published data in the logging infrastructure.

1.4 Context

This master thesis is written in the context of the Cyber Security Group (CSG) at The Arctic University of Norway (UiT). CSG conducts research involving distributed systems, with a focus on designing and implementing systems while investigating key properties such as scalability, fault tolerance, reliability, security, and partitioning, to mention a few. The CSG group solves distinct research problems by constructing prototype systems with an experimental approach, the primary research method for the group.

The CSG group has researched and explored subjects related to the fishing industry, particularly criminal activities, such as illegal fishing and overexploitation. The paper [27] examines a surveillance video dataset from a trawler and explores the potential applications of such data. In addition, the papers [26, 3] investigate distributed monitoring and surveillance systems for offshore environments, applying Artificial Intelligence (AI) while offering privacy guarantees. Another area of interest for the CSG group has been blockchain technology, which includes an analysis of Bitcoin's transaction fee volatility using a Machine Learning (ML) model for prediction [37], alongside a transaction inclusion model for Bitcoin [36].

This introduces only a few of the research projects the CSG group has been involved with throughout the years. This thesis will build upon this foundation by designing and implementing Haddock, a shared, distributed logging system using blockchain technology in the context of previously conducted research revolving around the maritime field and blockchain.

1.5 Outline

The thesis is structured as follows. *Chapter* 2 presents foundational knowledge on essential topics and concepts, which is key to understanding the work done in this thesis. Following, *Chapter 3* outlines the requirements, derived from the problem definition. Further on, *Chapter 4* provides an in-depth, comprehensive description of the implementation details of the shared logging infrastructure, Haddock. Next, *Chapter 5* introduces the experiments conducted to test and evaluate the system. Proceeding to *Chapter 6*, the results and architectural design of the Haddock are discussed. Finally, *Chapter 7* makes concluding remarks based on our findings, recaps the work accomplished in this thesis, and lastly, proposes future improvements for Haddock.

2
Background

This chapter outlines essential theoretical concepts and system models that are necessary to understand the presented work in this thesis. The first section introduces the general concept of blockchain technologies, followed by different types of blockchains. The next section provides an in-depth description of Ethereum, its specifications, and smart contracts, followed by a detailed explanation of Avalanche and decentralized applications, winding up the blockchain-related concepts. The background is finalized by a section on REST, a Web architectural style, and a section on essential cryptographic functionalities used by Haddock.

2.1 Blockchain

Blockchains are distributed and decentralized systems managed and sustained by multiple member nodes communicating over the wide-area Internet. These nodes, which may be individual computers or devices, run software processes and participate by sending messages over the network according to specified protocols. Each member node maintains a copy of the blockchain data structure and collaborates with others to validate transactions and uphold the distributed and decentralized network. Conceptually, a blockchain resembles a traditional linked list, comprising blocks with cryptographic references to their preceding blocks, forming a chain.

Understanding key concepts is crucial for grasping how blockchain technologies function. In the world of accounting, a ledger is a fundamental tool, denoting a book used to record account transactions. Primarily employed by companies, it serves to monitor the transaction history of customers, detailing their spending and outstanding balances. A ledger must ensure that every debit is matched by a corresponding credit, thus preserving balance. It is important to note that a ledger differs from a bank account, as it solely documents the financial activities within a company rather than serving as a bank account for funds.

Bitcoin [25] is perhaps one of the most renowned blockchains and functions as a distributed ledger where each participant in the network possesses its own private copy of the ledger. Whenever a participant intends to add a new entry to the ledger, it must be disseminated to all other members of the network. To address potential inconsistencies within the network, blockchain employs consensus protocols [21, 16]. Among these, the most prominent is the Proof-of-Work (POW) protocol [13, 25, 36]. POW allows participants to place trust in the ledger that has demonstrated the most computational effort. Every member of the network must complete a POW to append their transaction list to the ledger. POW involves a computationally demanding task that is easy to verify. Thus, if a participant wishes to broadcast transactions to the ledger, they must invest effort while other network members can verify and agree on the new transaction set. The introduction of a consensus protocol ensures immutability within the system. In POW, altering a single block requires the redoing of POW for all subsequent blocks, making it computationally infeasible to manipulate.

In Bitcoin, each block includes a cryptographic reference to the previous block, a collection of transactions, and a timestamp. These blocks are hashed using the Secure Hash Algorithm 2 (SHA-2) hashing algorithm. A depiction of the chain structure can be observed in Figure 2.1. The specific contents of a block will vary depending on the particular blockchain technology.

2.2 Types of Blockchains

The blockchain technology has evolved immensely in a relatively brief time period. As a consequence, blockchain systems have been tailored to suit the needs of different industries, introducing a diverse set of blockchain technologies with different applications.

A *permissionless blockchains* is open to everyone on the internet. All participants are permitted to partake in submitting transactions. Everyone has the ability to join and participate in the network's validation process [34]. The most popular

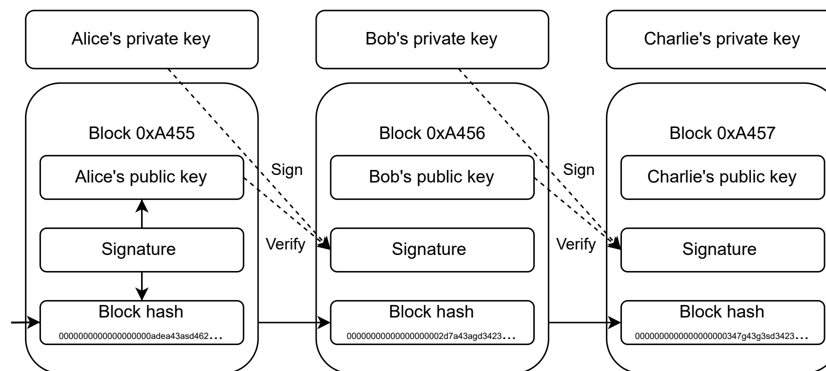

Figure 2.1: Blockchains stores transactions in blocks that are linked together by hashes. Inspired by Figure 1 [25]

blockchain technologies, like Ethereum and Bitcoin, are classified as public and permissionless.

A *permissioned blockchain* is partially or fully closed off from the open internet. Participation in submitting transactions and partaking in the validation process is not permitted for everyone [34]. This type of blockchain requires strict access control for the validation process and regular auditing and governance to mitigate anomalies or malicious activities. Permissioned blockchain can result in centralization, undermining the true nature of blockchain technology as a distributed and decentralized system. If the blockchain becomes too centralized, there is no guarantee that the blockchain remains immutable. Permissioned blockchains are, to a greater extent, more susceptible to censorship than permissionless [34].

In a *private blockchain*, a single entity governs the entire system, leading to a more centralized infrastructure [12]. Private blockchains can be publicly available for viewing but not public to be used by anyone [34]. In contrast, *Public Blockchains* are both publicly available for viewing, and anyone is permitted to use and partake in the system.

A *consortium blockchain* typically combines characteristics from both permissioned and private blockchains. Companies favor these blockchains as they offer more centralization in terms of access control and privacy of any proprietary data. Consortium blockchains share the authority of the system between the members of the network, and the infrastructure is decentralized on both homogeneous and heterogeneous hardware [12].

2.3 Ethereum

Ethereum is one of the leading blockchains today with a market cap close to 5 trillion dollars [8]. While Ethereum initially used POW as its consensus mechanism, it made a transition to *Proof-of-Stake (POS)* in 2022 [14]. In the POS consensus protocol, participants are required to commit a certain amount of funds as a stake to participate in the validation process of new blocks. The staked amount of money acts as collateral, ensuring participants follow the protocol. If any malicious behavior is detected, the staked funds risk being forfeited. Furthermore, the Ethereum network functions as a decentralized global computer anyone can use, commonly referred to as Ethereum Virtual Machine (EVM). Every network participant maintains a copy of the network's state, which is collectively agreed upon. Code execution requests sent to the blockchain network are called transactions. All transactions and previous states are recorded on the blockchain. EVM runs code execution on *smart contracts*. EVM is a virtual machine that executes and processes EVM byte code [42].

Smart contracts can be written in the language *Solidity* and is a representation of a program or code snippet compiled down to EVM byte code. Any network participant can create and deploy a smart contract to the Ethereum network. This ensures every network participant can access the functionality of an arbitrary smart contract, promoting publicity and transparency in the blockchain network. However, deploying and utilizing smart contract functionalities induces costs. Network participants must use Ether (ETH), the cryptocurrency of Ethereum, to cover these costs. Fiat currencies such as United States Dollars (USD) can be converted to ETH at crypto stack exchanges. Effectively linking EVM to monetary fees based on fiat currency. The smart contracts are stored on the blockchain, ensuring tamper-proofness and resistance to alterations.

Solidity is an object-oriented, statically written language primarily used for writing smart contracts for EVM-compatible blockchains. The language offers a diverse range of functionalities, including data types, data structures, function modifiers, and events. Solidity supports common data types such as integers, strings, and bytes, allowing developers to declare and store variables. Developers can create custom *structs*, a composite data type that can consist of different data types. Function modifiers are an important aspect of Solidity, serving as special functions designed to enforce certain conditions before code execution. Typically included as a function parameter, enabling preconditions to be set. Furthermore, Solidity provides support for events. Events are special constructors that can store and emit data to off-chain applications, incentivizing event-driven interactions with external systems. Solidity also offers visibility modifiers to variables and functions, which control the accessibility to off-chain interactions. Variables and functions prefixed with *private* will encapsulate all the data to within the smart contract, restricting access to internal contract

logic. Prefixing with *public* makes them externally available, allowing off-chain interactions via getter functions.

EVMs are capable of executing arbitrary code on request from any network participant; because of EVM's *quasi*-Turing-completeness [7, 42], it can carry out any computational sequence [38]. Ethereum uses a unit denoted *gas* to measure the computational resources required to execute operations within the network and is a mechanism to avoid misuse of Ethereum's Turing completeness [42]. This helps make the network less susceptible to either spam or never-ending programs (endless loops). Every computation requires a fee, discouraging any unnecessary operations, and participants are less likely to create programs that monopolize resources if there is an associated cost with the operations. Computations have a fixed gas cost that is universally agreed upon in the Ethereum network [42].

Gas fees are generated by combining the base fees of the computation and a priority fee. Adding the base fee per unit of gas and the priority fee per unit of gas, then multiplying the sum by the total amount of gas consumed, results in the total fee paid by the participant [42], after block creation, the base fee is burned, removing it from circulation in the network. Ethereum operates with a *gasLimit*, which indicates the maximum amount of gas a participant is willing to use for a transaction. During gas estimation, a higher gas limit allows more gas to be assigned to the base fee of the transaction. Any remaining gas can cover the priority fee. This incentivizes participants to set higher gas limits, as it increases the chance for the transaction to be included by the validator; unused gas during the gas estimation will be refunded to the participant. However, a gas limit that is set too low can run out of gas and fail the transaction while still incurring costs, with no refunding options. Gas fees are an essential part of the Ethereum network, ensuring efficient allocation of computational resources and that the network is functional and secure.

2.4 Avalanche

A new and emerging blockchain is Avalanche, an open-source blockchain employing POS consensus mechanism. Avalanche is developed by *Ava Labs* and is built to be EVM-compatible, meaning it supports the same smart contract functionalities as Ethereum, with a native token denoted *AVAX*. Avalanche offers a hybrid solution to permissioning, a hybrid model where the *Primary Network* in Avalanche is permissionless, allowing anyone to partake and build on the network while also supporting the creation of subnets with customizable restrictions. The Primary Network consists of three chains:

- Platform Chain (P-Chain)
- Contract Chain (C-Chain)
- Exchange Chain (E-Chain)

Avalanche is built and designed to support scalability on an extensive level, with the possibility of hundreds of millions of connected devices, all operating with high transaction throughput and low latency, in a decentralized manner without distinguishing between miners, developers, and users [32]. Avalanche provides an infrastructure to support multiple blockchains within the network, offering interoperability for porting existing blockchains on top of it.

Avalanche supports the creation of individual subnets, which can be either permissionless or permissioned. Permissioned subnets are ideal for organizations and businesses that rely on data privacy. The blockchain contents in the private subnet are only visible to a predetermined set of validators. Validators only validate transactions in the subnets they reside in, minimizing computational resource usage, as opposed to other blockchains where every transaction must be validated by every validator in the network. Every blockchain is validated by one subnet, and each subnet can validate multiple blockchains [32]. Subnets offer significant advantages in terms of flexibility and scalability, allowing validators to participate in arbitrarily many subnets, and arbitrarily many subnets may be created.

Avalanche features a family of consensus protocols known as *Snow*, which possess Byzantine fault-tolerant properties and are inspired by gossip algorithms [30]. The Avalanche consensus protocol ensures fast, probabilistic Time-to-Finality (TTF), the period in which a transaction is confirmed and rendered immutable. Its leaderless consensus mechanism and absence of traditional consensus protocols like Proof-of-Work allows for high transaction throughput. By deviating from PoW protocols, Avalanche drastically reduces the network's energy consumption [20, 30], making it a more energy-efficient solution.

2.5 Decentralized Applications (DApps)

A Decentralized Application (DApp) is an application or system that leverages the properties of blockchain technology to provide decentralized and immutable server environments. These applications deploy and run services on blockchain networks, benefiting from the inherent features of blockchain technology, including high availability, privacy, resistance to censorship, and full decentralization. DApps built on open permissionless blockchains are often

open-source, with open and verifiable code, ensuring properties such as full publicity and transparency. However, a larger proportion of DApps remain with closed source code [43].

While DApps offer several advantages, there are trade-offs. Running smart contract functions on blockchains can be expensive in monetary terms and requires optimized code to reduce the costs. For instance, minimizing storage writes is an optimization approach. On the positive side, applications that primarily rely on public static data only incur deployment costs of the data and then have free server-side storage as long as the blockchain network remains operational. Another trade-off can be performance and reliability; blockchain networks can suffer from high traffic and network congestion, which largely depends on the blockchain of choice. Additionally, immutability significantly affects the maintainability of DApps. Altering a deployed program cannot be done, but the functionality accessible through interactions can be changed with smart contract*upgrade* techniques [40], involving transferring the contract state to a new contract, or using a smart contract as a proxy and distribute functionality to smaller, replaceable contract. Ultimately, the maintainability of the DApp heavily relies on the design and implementation of the smart contract.

The emergence of new decentralized technologies like blockchains has sparked an idea for a web that leverages these mechanisms. A concept for a new iteration of the World Wide Web (WWW) is in progress and is referred to as *Web 3.0*, or simply *Web3*. This evolving concept is in its early stages. Currently, Web3 is often described as a buzzword or even *vaporware*, a term for announced but unreleased products. Therefore Web3 has been accused of generating hype in the world of blockchain and technologies employing cryptocurrency [2].

The rise of blockchain technologies has led to the development and creation of blockchain interaction tools tailored to leverage the decentralized and immutable properties blockchain technologies inherently offer. A prominent JavaScript (JS) library for developing DApps is Web3.js, which facilitates communication with Ethereum nodes via Hypertext Transfer Protocol (HTTP), Inter-Process Communication (IPC) or WebSocket [41]. Web3.js uses the Ethereum JSON-RPC API, a stateless and lightweight Remote Procedure Call (RPC) [19], that is exposed through an API. This makes Web3.js compatible with any EVM-compatible blockchain networks that support the JSON-RPC API. The library offers functionality to interact with, build, sign, and send transactions to smart contracts. Additionally, it supports the compilation and deployment of smart contracts, making it a valuable tool for DApp development.

2.6 REST API

Representational State Transfer (REST) Application Interface (API) is a network-based architectural style for network applications, based on a set of principles and constraints, which combined, achieves scalability, maintainability, efficiency, and security in Web services and applications [15]. REST standardizes the Web architectural model by leveraging Web protocols, especially HTTP and Web elements such as Uniform Resource Identifiers (URI) and Hypertext Markup Language (HTML). The key constraints comprising REST include Client-Server, Stateless, Cache, Uniform Interface, Layered System, and Code-on-Demand.

Client-Server is a constraint that enforces separation on client-side and server-side concerns. The User Interface (UI) and User Experience (UX) are handled by the client, while the server-side manages data storage and logic. This separation allows the components of each concern to evolve independently, improving scalability [15].

Stateless is the second constraint, requiring all communication to include all necessary information for generating and processing a request. No information or state is stored on the server, making the entire state session based on the client-side. This improves reliability during partial failure, as the client needs no reinstatement on the server-side, which allows for improved scalability since no state is stored between requests. A potential trade-off is increased network traffic by repeated requests and decreased server-side control over maintaining consistent client-side operations.

Cache is the third constraint, requiring the request to include a marking if the content is cacheable or non-cacheable. This signaling informs the client if the content may be reused as a response, which can enhance the efficiency by reducing server interactions. On the contrary, it can weaken the reliability, by potentially serving stale data, inconsistent from a non-cached response.

Uniform Interface is the fourth constraint and enforces a uniform interface for architectural components. The provided services are decoupled from the components, allowing them to develop and expand independently. Trade-offs include decreased efficiency due to standardized information transferring, as opposed to component-optimized interfacing.

Layered-System is the fifth constraint, emphasizing a layered architecture that restricts component knowledge by encapsulating component services, increasing security and scalability. This constraint allows for intermediary components to operate between layers.

Code-on-Demand is the final and optional constraint for REST, which extends client functionality by offering downloadable features that can be executed after deployment in production environments, providing extensibility. As a consequence, this can diminish application operations visibility, potentially creating a security risk.

REST API provides a robust and scalable framework for web services and applications by adhering to a set of well-defined principles and constraints. Each constraint contributes uniquely to the system's scalability, maintainability, efficiency, and security. While there are trade-offs, such as potentially increased network traffic, inconsistency, and security risks, the advantages of implementing a REST architecture style outweigh the drawbacks, making it a preferred and standardized choice for modern Web-based development.

2.7 Cryptography

Advanced Encryption Standard (AES) is a symmetric encryption algorithm that uses a single key for both encryption and decryption, operating as a block cipher on fixed-size 128-bit blocks. The key sizes are normally either 128, 196, or 256 bytes, making it infeasible for a brute-force attack, which involves trying all key combinations to decrypt a ciphertext to the corresponding plaintext [17]. Systems using this scheme are called symmetric cryptosystems and are fast and efficient in securing communication.

AES can be used with different *modes of operation*. One of the most common modes is Cipher-Block Chaining (CBC). In the following description of the modes, AES will be the encryption algorithm of choice. This encryption mode separates the plaintext data into fixed-size blocks, which must be performed sequentially. The mode is performed by using Exclusive-Or (XOR) with an Initialization Vector (IV) on the first plaintext block. After this operation, the result is encrypted with AES into a cipher block. All following plaintext blocks are XOR-ed with its subsequent cipher block before being encrypted. Forming a ciphered chain of blocks. Decrypting the ciphered chain is done by reversing the process and using the same Initialization Vector [17].

Another mode of operation is Cipher Feedback (CFB), which encrypts the fixed size plaintext blocks similarly to CBC. These modes differentiate in that CFB uses a feedback mechanism. CFB starts by encrypting the Initialization Vector with AES. Then XOR it with the plaintext. The resulting ciphertext is encrypted with AES. The decryption of the chain is done by reversing the CFB process. CFB can result in faster decryption than CBC [17].

An alternative symmetric encryption is asymmetric encryption. This approach uses a public key and a private key for data encryption and decryption. The public key can be used to encrypt data, while the private key can be for decryption. As the name suggests, the private key must be kept secret, while the public key can be accessible to anyone. The public key can be distributed to other entities, and these entities can encrypt the data with the public key. This data can only be decrypted with the corresponding private key, ensuring the data remains secure. Although private keys are typically used for decryption, they can also encrypt specific data to generate a verifiable digital signature. This signature can be verified by anyone possessing the corresponding public key, ensuring the authenticity and integrity of the signed data. Whereas the encryption and decryption scheme of asymmetric keys are strong, they are less efficient compared to symmetric cryptosystems. Therefore, asymmetric cryptosystems are often used to ensure safe symmetric key exchange amongst entities. Once the symmetric key is shared, it can be used for session-based communication, as symmetric keys are much faster in comparison [17]. A hybrid approach can leverage the robustness of both cryptosystems: secure key exchange from the asymmetric system and efficiency of data encryption and decryption from the symmetric system.

A cryptographic hash function reduces the size of data by creating fixed-size checksums that are derived from the original data. This type of functionality is essential in cryptography as it generates smaller pieces of data used for data integrity and error detection after storage and transmission of data. A hash function is considered strong if it possesses the properties *collision-resistance* and is a *one-way* function. First, the hash function h should be resistant against different data inputs D, yielding equal outputs, such that $h(D) = h(D')$. Second, while the hash should be easy to generate, it should be computationally infeasible to reverse the output hash to its original state. A popular set of standardized hash functions are Secure Hash Algorithm 2 (SHA-2), which denotes four variants *SHA-224*, *SHA-256*, *SHA-384*, and *SHA-512*, each producing a digest size indicated by their suffix [17, 28].

A digital signature can be generated using asymmetric encryption. A specific message or data can be encrypted using a private key, ensuring only the possessor of the private key could have generated this digital signature. A common practical approach for digital signatures involves combining asymmetric encryption with a hash function. The hash function produces a digest of the message, which is then encrypted using the private key. This approach is more lightweight and efficient than encrypting the entire message, which can be huge. The signer's public key can be sent with the digital signature for verification, ensuring the authenticity of the signature.

2.8 Summary

In this chapter, we explored various aspects of blockchain technology, including its fundamental principles, types of blockchains, and specific blockchain networks such as Ethereum and Avalanche. We introduced the standard Web-based architectural style REST API for network applications. Further on, we discussed important cryptography concepts ensuring the security and integrity of a system, covering symmetric and asymmetric encryption, modes of operation, cryptographic hash functions, and digital signatures. Additionally, we examined the concept of Web 3.0 and its implications for DApps, highlighting the motivation for tools like Web3.js used for blockchain interaction. Overall, this chapter provides a comprehensive overview of the foundational concepts and technologies essential for blockchain ecosystems.

3

Requirements

This chapter provides the requirements for the system according to the problem definition in Section 1.1, with regard to the key concepts and knowledge presented in Chapter 2. As outlined in Section 1.2. Our thesis follows the discipline of design in regard to methodology. Therefore, we present and state functional and non-functional requirements for the system, to create a conceptual overview of the system's needs in terms of architectural design and components to align with the problem definition.

3.1 Functional Requirements

This section provides the fundamental functional requirements for the system to comply with our problem definition, defined in Section 1.1. Functional requirements specify what the system shall do and which capabilities and functionality the system possesses. These requirement attributes outline the tasks, features, or actions the system must perform to satisfy the criteria specified in our problem definition.

Requirement 1 (Multiple public communication channels)**.** The system must support multiple communication channels to qualify as a shared logging infrastructure. The number of channels, can either be static or dynamic. Each channel shall provide the end user the ability to publish messages to secure, shared, and public storage. The communication channels shall not be suscepti-

ble to hijacking or blocking. The public storage platform for the communication channel data shall build on the concepts of blockchain technology, leveraging principles such as tamper-resistance, non-repudiation, and append-only data structures. All data shall be publicly available and transparent to the end user.

Requirement 2 (Encrypt data)**.** All data stored on a public storage platform must be encrypted. While the specific type of cryptographic mechanism is not strictly defined. Symmetric or asymmetric is preferred. However, the robustness and rigidity of the encryption is not critical for this thesis. The strength of the encryption will not be tested and is considered out of scope.

Requirement 3 (Access control)**.** The system shall provide Role-Based Access Control (RBAC) for its logging infrastructure. It shall administer membership management with at least two sets of roles: an owner role and a member role. Owners shall be privileged with adding and removing users in the system. While members shall have access to the logging system's publication and acknowledgment functionality within the logging infrastructure.

Requirement 4 (Non-repudiation)**.** The system shall log the address and identity of both the message publisher and the message receivers, with a high degree of non-repudiation. These records shall be stored on a platform that builds on blockchain technology, ensuring their public, transparent, and continuous availability to end users.

Requirement 5 (Timeliness)**.** The system shall provide functionality for timely dissemination of secure data, supporting multiple severity levels. Data publication shall occur in a controlled and timely manner, ensuring the data is publicly available in an interpretable format, simultaneously across the entire system. Hence, all participants of the logging infrastructure shall have equal benefits in terms of receiving data of different severity levels. No end users shall gain a competitive advantage by receiving the data earlier than others.

Requirement 6 (User Interface)**.** The system shall implement a UI that utilizes the functionality of the system to publish and acknowledge messages in a user-friendly and understandable manner. The UI shall include a minimum of an input text box and a button for the publication functionality, providing a straightforward solution to input and submit messages. Additionally, the acknowledgment functionality shall be facilitated through a button associated with each message, allowing the end users to acknowledge their possession of the encrypted data.

Requirement 7 (Crypto Wallet)**.** The UI shall integrate a crypto wallet of choice, enabling the end user to connect their wallet to the system. Allowing the end user to monitor and control the transaction flows from the UI to the blockchain network. Signing and confirming transactions of significant importance, such as acknowledgments and key and data publication, shall go through the connected crypto wallet, ensuring end user authorization for all fund withdrawals from their wallet.

3.2 Non-Functional Requirements

Non-functional requirements represent the crucial conditions that define the effectiveness and behavior of the system. Unlike functional requirements that define what the system should do, non-functional requirements specify how effectively the system performs under various conditions, attributes, or constraints. In this section, we will discuss the importance of the non-functional requirements for the system, and how these requirements comply with our problem definition specified in Section 1.1. We will list non-functional requirements [35] and investigate how they will design and implementation of Haddock.

3.2.1 Reliability

Reliability within a system is the capability to perform system functions and services in a consistent and accurate manner. Key non-functional attributes associated with reliability are availability, security, and resilience. All of which contribute to the system's ability to deliver its services in a determinable fashion. Other aspects include stability and dependability. The User Interface (UI) shall consider reliability when handling user input and I/O interactions with the blockchain network, to ensure user interactions are processed correctly and consistently.

3.2.2 Responsiveness and Availability

Responsiveness defines the system's ability to respond to user interactions and return results on inputs and requests. It is an important attribute of how a system handles throughput, latency, and processing time, which collectively influence the system's efficiency and performance. Availability, on the other hand, is an attribute of the system's capabilities to deliver and provide services consistently.

In this thesis, we explore the use of a shared logging infrastructure with a focus on timely dissemination. Both responsiveness and availability are crucial elements for the system to comply with the functional requirements. Availability becomes particularly critical if the system is being used in large diverse geographic regions.

In a synchronization service, like the one we proposed, data availability and integrity must be present to give equal benefits to all participants. Due to the decentralized nature of blockchain technology, availability and responsiveness are inherent attributes of the blockchain network's system architecture. The logging infrastructure system architecture is dependent on blockchain networks

for these attributes. Therefore making additional measures unnecessary.

3.2.3 Security and Resilience

The system shall provide security measures to prevent any unauthorized access and mitigate security threats. Key aspects of security in a system include confidentiality, integrity, and availability, as well as authorization, authentication, and non-repudiation. Our system leverages blockchain technology, which inherently provides several robust security features. While security is a crucial element in this thesis, We primarily leverage these measures, rather than developing standalone security features. Although, the system needs to make sure cryptographic keys are kept safe and handled with care. Additionally the system shall demonstrate resilience to potential failures and external attacks, maintaining highly available services is essential in case of partially system failures. The system shall, to a certain extent, be able to adapt to unfortunate events such as attacks and arbitrary system failures. Leveraging the decentralized nature of blockchains can enhance the resilience of the system, by mitigating single points of failures. Given blockchain technology's inherent measure for resilience, implementing additional measures for resilience will not be necessary for the server-side of the system. However, the smart contract shall incorporate access control measures. Proper access control and secure key management will be considered

3.2.4 Usability

A system that relies on user input and interactions must be user-friendly to use. The design of the UI should be intuitive to ensure ease of use. Usability in the system can be improved by limiting the accessible features available at any given time [10]. Designing an intuitive UI will be regarded as an important focus.

3.2.5 Maintainability

Maintainability is important in all software and engineering systems. It determines the complexity of adding, editing, and fixing features in a system. In large-scale systems with complex components, maintainability is an essential factor to ensure that further development aligns with other non-functional requirements. In this thesis, maintainability shall be considered for the components, especially because of the complex and nascent nature of blockchain technology.

3.3 Summary

This chapter outlines the system's requirements, detailing both functional and non-functional aspects. Functional requirements include supporting multiple secure communication channels, providing a public storage platform based on blockchain technology for message logs, encrypting data, ensuring timely dissemination of secure data, implementing access control, logging data publishers and message acknowledgers, and providing user interfaces for message publishing and acknowledgment. Non-functional requirements focus on reliability, responsiveness, availability, security, resilience, usability, and maintainability, leveraging blockchain technology's inherent features to enhance these attributes while ensuring ease of use and future development.

4
Design & Implementation

This chapter presents the implementation details for our shared logging service, Haddock, using the Avalanche blockchain for its immutability and decentralized properties. A smart contract is used to disseminate and synchronize access to published data, making it feasible to track both the publisher of data and the initial message receivers. The message is published in a two-phase dissemination protocol. In the first phase, the message is published in an encrypted format. After some time has passed, the second phase disseminates the decryption key, inspired by the dissemination technique employed in *FirePatch* [18]. Haddock combines this dissemination technique with barrier synchronization to determine when the decryption key should be disseminated. A predetermined number of acknowledgments is required before the second phase can commence. Haddock is structured with a server-side and a client-side, which implements a REST API service to communicate and interact with the server side. We start by presenting a system overview and Haddock's workflow, then delve into the implementation details of Haddock's server-side and client-side.

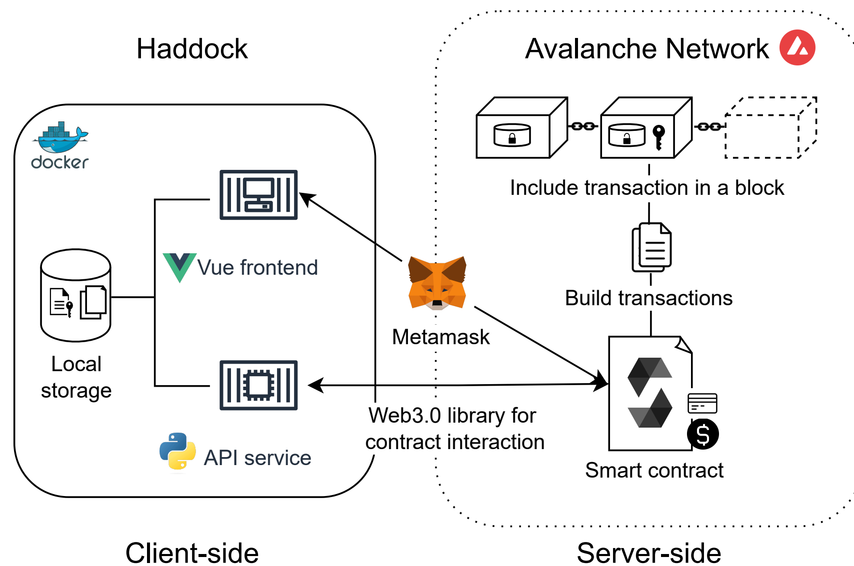

Figure 4.1: An architectural overview of Haddock

4.1 System Overview

Haddock is built as a DApp following a classic client-side server-side architecture. The Avalanche network and a deployed smart contract act as the server-side for the DApp and represent the dissemination services for the system. The client-side consists of a frontend UI, a Metamask crypto wallet integration, and a REST API service which exposes the dissemination services of the server-side to the client-side. In addition, the client-side is structured with a simple local storage, including a directory for storing smart contracts and a directory for cryptographic keys. An architectural overview is displayed in Figure 4.1.

4.2 Haddock's Workflow

This section will present a brief overview of Haddock's workflow, emphasizing how the messages are logged and disseminated with synchronization. The workflow process of message dissemination and synchronization is done in the following steps:

(1) **Encrypted Message Dissemination:** The publisher encrypts a message and sends it to Avalanche

(2) **Message Emission:** Avalanche emits the dissemination of encrypted data to readers

(3) **Message Retrieval:** Readers fetch the encrypted data

(4) **Acknowledgement:** Readers ACK the message by creating a digital signature and sending it to Avalanche

(5) **Quorum Check:** Publisher check Avalanche for quorum

(6) **Decryption Key Dissemination:** Publisher disseminate decryption key on quorum

(7) **Key emission:** Avalanche emits the publication of the decryption key

(8) **Decryption and Display:** Readers fetch the decryption key, decrypt, and read the message

(1) The publisher use the client-side frontend to input a message, which is encrypted symmetrically, the publisher use Metamask to sign the transaction, then send it to Avalanche. (2) Avalanche receives the encrypted message and stores it in struct for *incomplete messages* on the smart contract, then emits the publication of the encrypted message to readers. (3) The frontend UI for each reader automatically fetches the newly emitted encrypted message and displays it in a list. (4) Readers acknowledge the message by pressing a button in the frontend UI which create a digital signature of the encrypted data, the reader use Metamask to sign the transaction, then send it to Avalanche. (5) The publisher interacts with Avalanche and checks if the number of required acknowledgments for the message is met. (6) If a quorum is reached, the publisher disseminates the decryption key to Avalanche. (7) Avalanche receives the decryption key and stores both the existing encrypted message and decryption key in a new struct for *complete messages* on the smart contract. (8) Finally, the frontend UI for each reader automatically fetches the decryption key, decrypts the message, and displays it in a list, making it readable to readers.

```
1  event MessageDisseminated(
2      uint256 indexed id,
3      uint256 timestamp
4  );
5  event DisseminationTime(
6      uint256 indexed id,
7      uint256 time
8  );
9  event KeyDisseminated(
10     uint256 indexed id,
11     uint256 timestamp
12 );
```

Listing 4.1: The three events on the SHIELD contract

4.3 Server-Side Components

Haddock uses one contract; Secure Handling of Information with Encrypted Logs and Decryption Key (SHIELD), which is responsible for logging and the two-phase dissemination of messages, and is designed to be deployed on the Avalanche's *Primary Network*. The SHIELD contract offers three primary functions. First, the contract can disseminate and store encrypted messages in bytes on the blockchain network, making it publicly available. Second, the contract can store acknowledgments of the encrypted messages. Third, the contract can disseminate and store the decryption key for the corresponding encrypted message.

Solidity includes *events*, which are special constructors that can store and emit data to off-chain applications. The SHIELD contract includes three events: one event for signaling the completion of the decrypted message dissemination, an event to emit the dissemination time of the key, and lastly, signal the completion of the key dissemination. All events on the smart contract are illustrated in Listing 4.1.

4.3.1 Disseminate Message

During the first phase of the dissemination, encrypted messages are stored in *structs* on the SHIELD contract. The `message` struct includes necessary metadata such as an assigned message identifier, publisher details, and the number of required acknowledgments for the second phase of dissemination to commence. A full overview of the message struct is illustrated in Listing 4.2. Each message struct is then stored inside a mapping, and this mapping

```
struct Message {
    uint256 id;
    uint256 timestamp;
    address publisher;
    bytes message;
    bytes key;
    bytes hash;
    uint256 quorum;
    uint256 acks;
}
```

Listing 4.2: Message structure represents the message with all relevant metadata

```
struct Signee {
    uint256 id;
    uint256 ts;
    address signer;
    bytes signature;
    bytes public_key;
}
```

Listing 4.3: The signee struct represents an acknowledgment. All acknowledgments are stored inside an array

represents a key-value storage for all incomplete messages (encrypted messages). The identifier of the message is used as a key for the message entry in the mapping, and the identifier is stored inside a dynamic array. Finally, both the mapping and the dynamic array are stored inside the `incompleteMessage` struct, illustrated in Listing 4.4. An identical struct called `completeMessages` is implemented to store all messages that are decrypted. Storing the messages in mappings allows for multiple publishers at the same time, creating multiple public communication channels, which is an essential requirement of the system.

The SHIELD contract is implemented with function modifiers, which is a special function in Solidity designed to enforce certain conditions before code execution. Modifiers can be attached to functions to ensure specific requirements are satisfied before their execution. SHIELD uses modifiers primarily for access control, as only the owner of the contract is privileged to add new members to the contract, ensuring controlled access to the contract's functionality. Additionally, modifiers are used to verify the publisher, confirm message signatures to avoid double signing, and validate the quorum requirements. Modifiers enhance the security and resilience of the SHIELD contract's operations and are important for the functional *access control* requirement in Section 3.1. Modifiers are crucial during the message dissemination process as they can restrict the use of the SHIELD contract exclusively to members.

4.3.2 Store Acknowledgments for Encrypted Messages

In Haddock, a significant feature is the capability to record acknowledgers on-chain in a blockchain ecosystem where all data is public and transparent. A notable challenge is documenting who has accessed and read the message

```
1 struct IncompleteMessages {
2     mapping(uint256 => Message)
         incomplete;
3     uint256[] keys;
4 }
5 struct CompleteMessages {
6     mapping(uint256 => Message) complete
         ;
7     uint256[] keys;
8 }
```

Listing 4.4: The incomplete and complete structs in the SHIELD contract. The struct represents a mapping, in which the message's ID maps to an entry of the Message struct (found in Listing 4.2)

while still maintaining the decentralized properties of the system. Recording the publisher is relatively straightforward since the publisher must use transactions when disseminating. However, anyone can read data on a publicly accessible blockchain like Avalanche. Making it necessary for an alternative approach to track and log the data access without compromising decentralization.

To create controlled access for the primary data readers. Haddock implements a barrier synchronization to record the recipients of the encrypted messages. This ensures tamper-proof evidence that the readers possess all the necessary information to access the encrypted message, increasing non-repudiation, an essential requirement for the system. However, once the key is disseminated, there is no way, from this point and onward, to determine new readers who have accessed the messages. As a consequence of the barrier synchronization, the logging functionality of Haddock naturally shifts towards serving as a synchronization service that can be used for logging purposes, rather than a comprehensive system with complete access control over message publishers and readers.

Each acknowledgements are stored in a struct. The `Signee` struct contains the identifier of the message, a timestamp, the address of the signer, and a digital signature of the received encrypted message, along with the public key of the signer, which is illustrated in Listing 4.3. All `Signee` structs are then stored inside an array. In addition, the acknowledgment function includes a modifier, ensuring no double signing can take place.

A reason for maintaining a separate signer and message struct is to avoid dynamic nested structs and arrays, which Solidity currently does not support. Instead, a single signer is appended to an independent struct. Including the

signer struct within the message, struct can lead to memory issues due to the unknown size of the signer struct. There are several benefits to this design choice. First, reading data requires less computation since there is less data to iterate through. Additionally, storing acknowledgments within the `message` struct is complicated because of the dynamism of the `signee` struct. Keeping a separate struct simplifies the process and improves the structure and organization of on-chain data. Overall, a separate signer struct is an optimal choice for Haddock.

4.3.3 Dissmeniate Key

When a quorum of acknowledgment is met, the second phase of dissemination will commence. The server-side dissemination of the decryption key involves three major steps:

(1) Move the `Message` from the `incompleteMessages` to `completeMessages` mapping

(2) Disseminate the decryption key

(3) Emit an event, signaling the dissemination of the decryption key is concluded.

(1) The `Message` is moved by copying the `Message` stored at the given identifier in the `incompleteData` mapping, and store it with the same identifier in the`completeData`. The `Message` is then deleted from the `incompleteData` map. (2) The decryption key is stored in the `Message` struct, which is now located in the `completeData` map, with a hash of the encrypted message from the publisher, which can be used for verification of the digital signatures. (3)Lastly emit the `KeyPublished` event, specified in Listing 4.1. This emitted event is a notification, signaling listening off-chain application that the decryption key has been published to the blockchain.

Synchronization options are important for the logging infrastructure to achieve timely dissemination. So far, we have presented the events, data structs, and the three primary functions of the server-side of Haddock. The server-side is responsible for storing the messages and working as a secure, shared, distributed system. The synchronization mechanism is part of the client-side, which is responsible for the dissemination of the decryption key in a timely manner. Different synchronization options for timely dissemination will be discussed further in Section 4.5.

4.4 Client-side Components

The client side of Haddock consists of a frontend User Interface implemented in Vue.js, a JavaScript (JS) framework for Web interfaces, and a client-side REST API service implemented in Python with Flask. The API service relies on the Web3 libraries for interactions with the blockchain network, including sending transactions, smart-contract operations, and reading blocks, which is used to create a toolchain of smart contract interaction tools. The REST API service's primary function is to interact with the SHIELD contract. Close to all contract interactions happen through the REST API service except for two transaction types, which happen directly in the frontend.

4.4.1 Contract Interaction Tools

The contract interaction tools are a toolchain that offers wrapper functions for common Web3 functions. The Web3 library for Python offers a lot of functionality for blockchain interaction. Most of the functions are primitive and can cause hard coding and redundancy in code. The inspiration for the toolchain is to reduce redundancy and make contract interaction more dynamic and accessible for developers. These tools are designed for building, signing, and sending arbitrary transactions to arbitrary contracts with Python. All endpoints exposed by the client-side API service in Section 4.4.2 leverage the contract interaction tools for blockchain interaction.

Communicating with a smart contract is not as straightforward as using a connection string, URL, or other connection methods. A common method for interacting with blockchain ecosystems, such as Ethereum and Avalanche, is with an Application Binary Interface (ABI). Web3 can generate the ABI for the SHIELD contract and store it within the client-side, as Haddock has a local storage for storing Solidity contracts. Ideally, the contract could be fetched directly from the blockchain as long as the contract address is known. The contract must be open-source, uploaded, and verified to the blockchain's block explorer. Haddock is designed to support both retrieval options. Since the contract has been continuously developed, Haddock has preferred the former options.

With a smart contract's ABI and the contract's address, a Web3 object that represents the contract can be instantiated. This Web3 object supports methods for building, signing, and sending transactions, as well as contract interaction. Two unique wrapper functions are introduced to streamline smart contract interaction. Both provide a convenient interface for transaction and View functions.

The first wrapper function is designed to send transactions. It takes a smart contract function name and a list of arbitrary parameters as arguments, making it versatile and useful for any transactional function. It constructs the transaction and signs it using the user's private key before dispatching it to the network. Upon successful execution, it returns a receipt confirming the transaction.

Similarly, the second wrapper function is tailored for View function interaction. View functions in Solidity are functions that allow off-chain applications to read and query data from blockchains without modifying it or altering the state of the smart contract, by providing the wrapper function with the smart contract view function name, and a list of arbitrary parameters as arguments, access to any View function can be done seamlessly. Facilitating smart contract interaction. Together, these wrapper functions offer an interface, ensuring efficiency and ease of use, simplifying both the process of altering and querying smart contract data.

4.4.2 REST API

The client-side offers a REST API service for most of the blockchain interactions. One main reason for having a separate API service is for more compute-heavy processes. The shared logging service relies heavily on encrypted data. Encryption can be resource-intensive, particularly for larger datasets. Establishing a separate service offers the flexibility to host it on a device with scalable computing resources. Additionally, using a frontend-backend architecture for the client-side allows for a more structured and organized implementation of the application. We use the term REST API service to refer to the backend of the client-side to distinguish it from Avalanche, which acts as the backend for Haddock. All API endpoints represent unique services and are listed in Table 4.1. Endpoints prefixed with *client* are operations on the client-side, while those prefixed with *contract* interact with and execute computations on the SHIELD contract on the server-side. The most essential services will be described in more detail.

The most important mechanisms of the API service are the encryption of the message, creating a digital signature, key dissemination, and fetching incomplete and complete logs. These are facilitated through the endpoints `encrypt-msg`, `digital-signature` and `disseminate-key`, which are client-side services, and `incomplete-log` and `complete-log`, which are calls to the SHIELD contract.

Endpoint	Method	Description
`/api/contract/disseminate-key`	POST	Disseminate the decryption key to the network
`/api/contract/force-push-key`	POST	Force push the decryption key to the network. *Requires* quorum
`/api/contract/incomplete-log`	GET	Get all incomplete messages on the contract
`/api/contract/incomplete-new`	GET	Get the latest incomplete message
`/api/contract/complete-log`	GET	Get all complete messages on the contract
`/api/contract/complete-new`	GET	Get the latest complete message
`/api/client/encrypt-msg`	GET	Encrypt the data
`/api/client/digital-signature`	GET	Create a digital signature with the encrypted message
`/api/client/abi`	GET	Get the ABI of the contract
`/api/client/contract_address`	GET	Get the address of the contract

Table 4.1: Listing of all endpoints exposed by the client-side API service

4.4.3 Client Services

The *encryption service* (`encrypt-msg`) takes the plaintext message sent from the frontend as an argument. The cryptographic implementation utilizes the python library *Cryptography*. The process involves three steps: First, a 256-bit encryption key is generated and stored in the local storage on the client-side. Second, the plaintext message is symmetrically encrypted using AES with CFB mode, with a randomized 16-bit Initialization Vector. AES is used because it is standardized and easy to implement with libraries. Third, the resulting ciphertext is encoded with Base64, a binary-to-text encoding scheme that converts the ciphertext to American Standard Code for Information Interchange (ASCII) encoding. A standard format supported by Web-based communication channels. Allowing the API service to effectively send the encrypted data to recipients while aligning with the *encrypt data* requirement from Section 3.1.

The *signing service* (`digital-signature`) produces a digital signature which is done in primary steps. Initially, the encrypted message is concatenated with a timestamp, then hashed using SHA-2 algorithm with a 256-bit output, producing a fixed-size hash. Further on, the hashed message is encrypted(signed) with the signer's private key, effectively creating a digital signature. Including a timestamp in the digital signature can increase the non-repudiation of the signatures, even in scenarios of a compromised private key [44]. To verify the digital signature, the system, or any other parties, can decrypt it with the signer's public key and compare it against the corresponding data hash stored on-chain. Lastly, the digital signature is returned to the recipient.

The *key dissemination service* (`disseminate-key`) handles a POST request and runs as a background process that spawns a thread, idling until a quorum of acknowledgments is met by interacting with the SHIELD contract. Once the quorum is met, a transaction is sent to the smart contract, emitting an event with the decryption key's dissemination time in Portable Operating System Interface (POSIX) time to avoid potential time zone complexities. By default, the final publication time is set to 10 minutes after the initial quorum is received. The decryption key (in byte format), a hash of the encrypted message for verification, and the corresponding identifier are sent with a transaction to the SHIELD contract. Concluding the key dissemination functionality on the client side. Section 4.3.3 explained the server-side handling of the key dissemination.

4.4.4 Contract Services

The *logging service* (`complete-log` and `incomplete-log`) includes fetching incomplete and complete messages from the SHIELD contract. The endpoints are routes defined to handle GET requests, retrieving log data stored on the SHIELD contract. The process begins by fetching the incomplete and complete message keys, stored in respective dynamic arrays, on the SHIELD contract. These keys are the identifiers that map to incomplete and complete messages, as described in Section 4.3.1. The service fetches the associated incomplete and complete messages from the mappings for each key retrieved. The key also fetches the list of signers corresponding to each complete message. These endpoints vary slightly for incomplete and complete data retrieval, particularly for dictionary formatting. For incomplete message retrieval, the message is encrypted, while the complete message is decrypted and stored in plaintext in the dictionary. The dictionary contains details such as message (encrypted or plaintext), identifier, timestamp, publisher, signers, and acknowledgment number, depicted in Figure A.1. These dictionaries are collected into a list. Finally, the function serializes the lists to JavaScript Object Notation (JSON) format and returns them to the recipient.

Similarly, the `incomplete-new` and `complete-new` endpoints provide the same functionality as the logging service but are used to fetch the newest incomplete or complete messages. These endpoints are implemented as event listeners using Server Sent Events (SSE) to stream new messages to the client in real time.

Smaller services include the forceful dissemination of the decryption key to the SHIELD contract. This feature is available if the key dissemination service should fail. If Haddock encounters any errors or crashes resulting in a restart, there is no mechanism to restart the background publication process. This feature enables the publisher possessing the decryption key to disseminate the key by force to the blockchain, given that the incomplete message has received a quorum. However, this is not an optimal solution; an automatic mechanism should handle this upon restart to maintain the integrity of the system's operations. The current force push solution undermines the requirements for timely dissemination, as the client can disseminate the key at any time as long as a quorum is met. thereby disrupting the synchronization process.

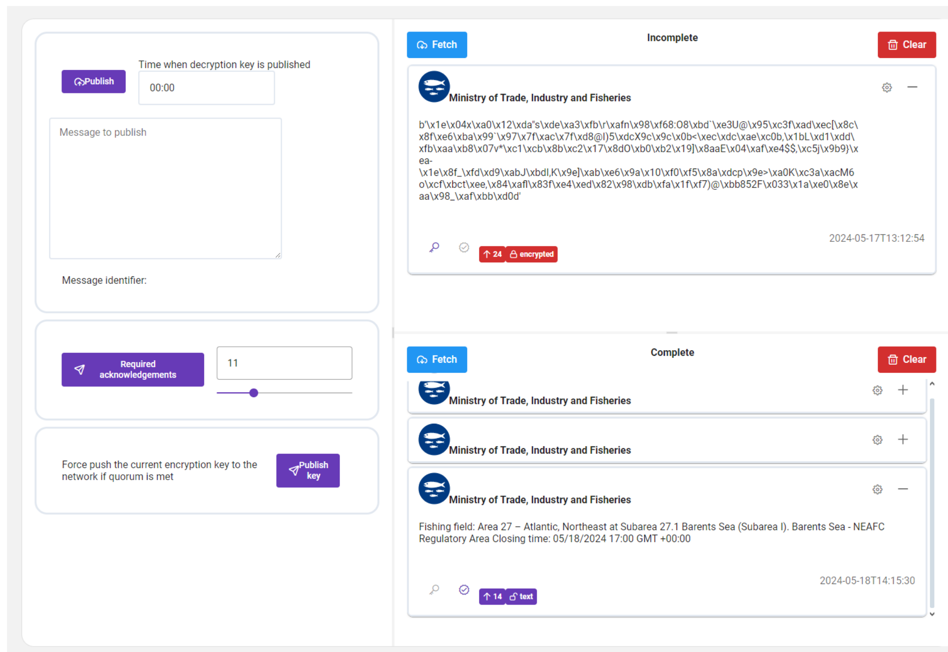

Figure 4.2: A complete view of the Haddock user interface.

4.4.5 Frontend

The client-side's frontend UI is developed using Vue.js. The frontend UI is built and serves as a demonstrator of Haddock's capability as a logging and dissemination service, integrating services from the REST API service to seamlessly interact with Avalanche. The UI offers three main functions. Disseminate messages to Avalanche, browse message logs, and acknowledge incomplete messages. A full overview of Haddock's UI is illustrated in Figure 4.2.

Haddock incorporates the cryptocurrency wallet Metamask, a Web browser extension that enhances user control over cryptocurrency funds. Metamask allows users to sign transactions directly in the browser, removing the need for hardcoded signatures and gas limits in the code. If the total amount of gas exceeds a hardcoded gas limit, the transaction is invalidated, and there is no way for the user to change this except by modifying the source code. Making the Metamask integration a crucial requirement for Haddock's functionality. The front end provides a user-friendly interface for connecting a crypto wallet to Haddock, accessible via a button on the navigation bar, the connection is facilitated with Web3 tools, depicted in Figure 4.3.

Initially, the architecture was designed to handle all transaction-related computing within the API service. However, due to the integration of Metamask,

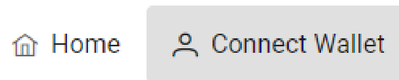

Figure 4.3: Haddock features a simple navigation bar. A button can be used to connect the crypto wallet to the system.

the frontend needs to build, sign, and send the transaction to the blockchain network. As a result, the architectural model has become more intricate and less structured. Consequently, the message dissemination and the acknowledgment of incomplete messages are done directly in the frontend code and not through the API service because Python cannot access the functionality of browser extensions. Despite the added complexity, integrating a crypto wallet, like Metamask, is essential as it aligns with the functional requirements specified in Section 3.1.

4.4.6 Frontend Message Dissemination

The message dissemination is designed through a simple UI component comprising an input text form and a button for executing the process. Additionally, the component includes a slider for setting the number of required acknowledgments and a button to force key dissemination, as illustrated in Figure 4.4. The encryption service, as described in Section 4.4.3, is used to encrypt the message. Subsequently, the frontend builds and sends the transaction to the SHIELD contract, in which the publisher is prompted with a request from Metamask to sign and confirm the transaction, as illustrated in Figure 4.5. Upon transaction confirmation, the frontend receives a receipt indicating whether the message was successfully disseminated to the SHIELD contract. Finally, the key dissemination service is invoked, starting a background process and idling until it reaches a quorum.

The layout of the encrypted messages presented in Figure 4.6 features two buttons and a list composed of message panels. When Haddock is initiated, two separate event listeners are initialized on the `incomplete-new` and `complete-new` endpoints described in 4.4.4. These event listeners continually check for new entries in the incomplete and complete message structures on the SHIELD contract and update the lists. If an event listener fails, the end user can manually fetch messages from Avalanche by clicking the button titled "Fetch" and invoke the logging service from the presented in Section 4.4.4. The other button clears all messages from the message list. Each message panel displays essential information, including the publisher of the message, the encrypted message, a timestamp, and the required number of acknowledgments. In addition, every

Figure 4.4: The publisher component includes an input text form and a button for message dissemination, a button, and a slider used to set the required acknowledgments. Lastly, a button is used to force push the decryption key to the network.

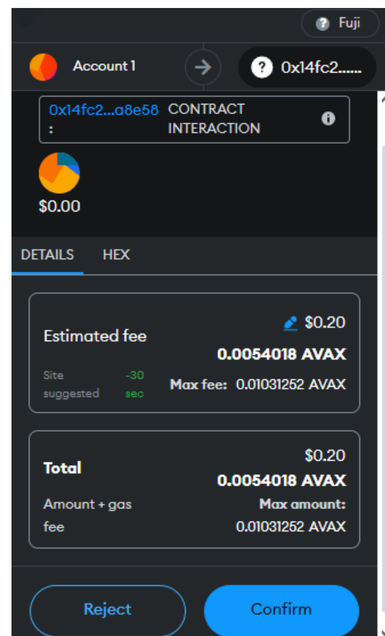

Figure 4.5: The transaction window from the Metamask browser extension. This box is used by the client to sign the transaction.

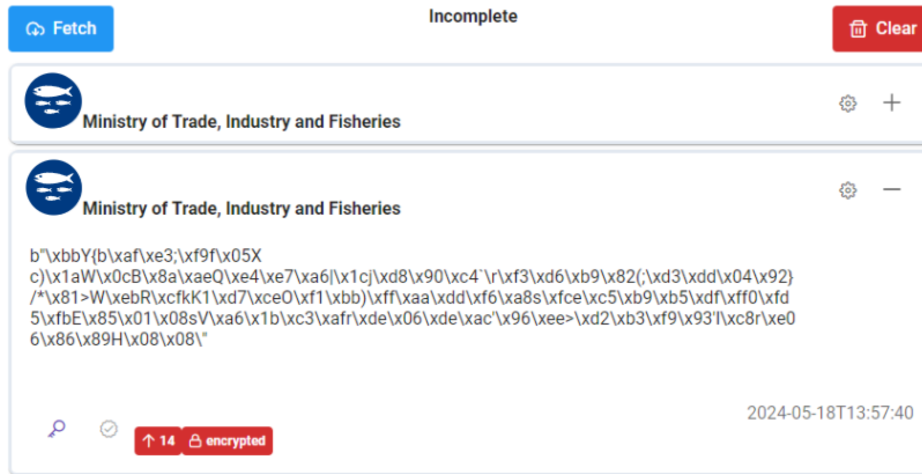

Figure 4.6: A display of the incomplete messages. The key in the left corner is a button to acknowledge the message. The numbers to the left of it display the required number of acknowledgments before the decryption key is disseminated.

panel includes a *key* button at the bottom left to acknowledge the message. The message lists are cached in the browser to increase efficiency.

4.4.7 Frontend Message Acknowledgment

To acknowledge an incomplete message, a user must click the key button in the message panel. The *signing service* is invoked to create a digital signature, then sends a transaction to the SHIELD contract, passing the digital signature of an encrypted message, identifier, and the signer's public key. The identifier for the acknowledgment message is stored in the browser's cache, disabling the user from acknowledging a message multiple times. Mechanisms for ensuring no double signing on the server side were reflected in Section 4.3.2.

When a quorum is met, the background process from the key dissemination service will disseminate the decryption key. Completed messages are stored in a separate layout, like the incomplete message list. Displayed in Figure 4.7, but the content is decrypted and displayed in plaintext, making it readable and understandable to the user. The ability to acknowledge messages is removed from the log view. Each message panel includes a verification mark button in the bottom left of the message panel, which expands a modal displaying all signers of the respective message. Next to this button is a number indicating

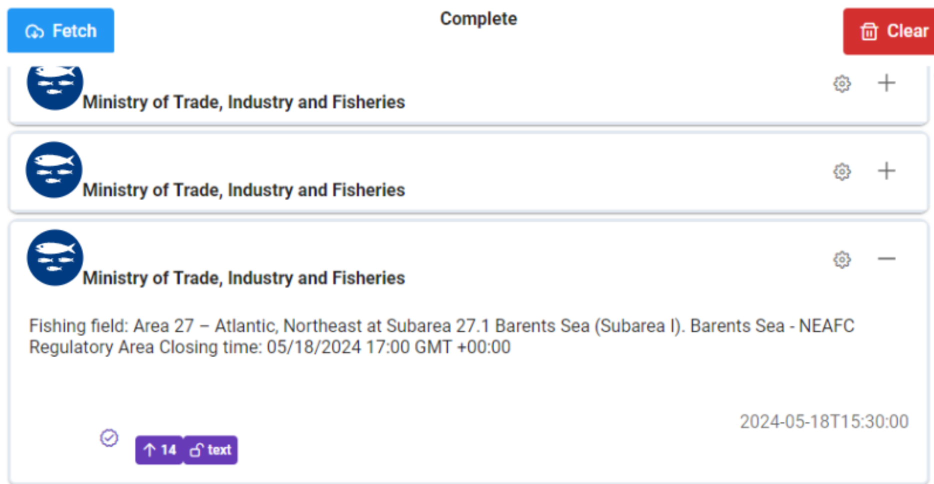

Figure 4.7: A display of all complete messages. The verification mark down left in the message panel opens a modal with a list of all signers of the current message. The number to the left of the verification mark displays the number of signers.

the quantity of acknowledgers.

The list of signers comprises the address of the signee, the digital signature, and, lastly, a verification mark signifying the legitimacy of the digital signature. The verification mark is awarded when the digital signature is successfully verified against the hash of the encrypted, which is disseminated concurrently with the decryption key. The signer list is depicted in Figure 4.8

4.5 Synchronization Options

Haddock has multiple options when it comes to barrier synchronization preferences concerning the decryption key. While the encrypted message has no initial publication requirements, in terms of synchronization, encrypted messages can be disseminated anytime. Synchronization is strictly applied to key dissemination, with synchronization mainly occurring off-chain on the client-side, complemented by restrictions on-chain, enforced by SHIELD contract. The decryption key can only be disseminated once a predefined quorum of acknowledgment is met. These system restrictions set constraints on the available synchronization options.

The most elemental synchronization primitive involves disseminating the decryption key on a regular time interval after the initial quorum is met. The

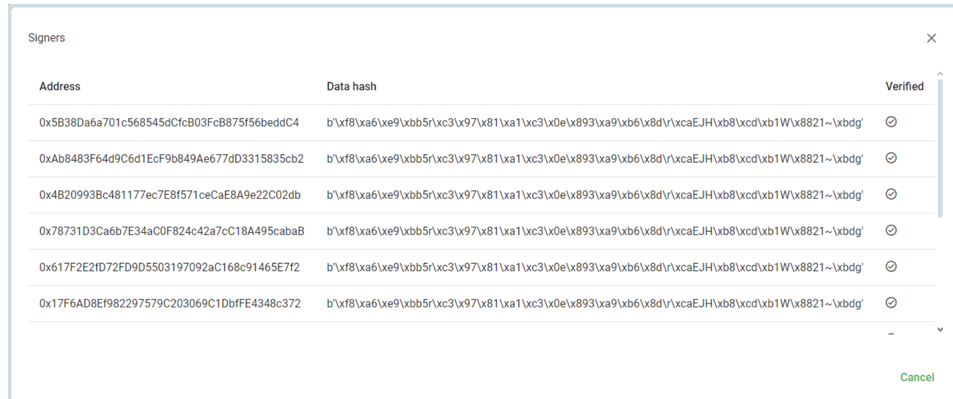

Figure 4.8: The full list of singers for a given complete message. Their address, hash of the encrypted data, and a verification mark are included in the modal.

dissemination time could be seconds, minutes or hours after the quorum. This options would suffice in most cases. By default, Haddock has a final dissemination time of 10 minutes after the initial quorum is reached. Although, the publisher can override the default time interval and set a desired time interval in the frontend.

Another approach is to schedule the dissemination of the key for a specific time and date, with the constraint that the quorum must be met before the key is disseminated. As illustrated in Figure 4.4, next to the publish button is an input field for final dissemination time, and will only activate this option if the input field is changed, otherwise it will remain on the default 10 minute time interval. However, this approach runs the risk of no key dissemination if the quorum is not satisfied within the designated timeframe. To mitigate the risk, a mechanism could be implemented to reschedule the dissemination for the next day or delay the final key dissemination time by an hour until the quorum is met and the key is successfully published.

4.6 Summary

This chapter presents comprehensive design and implementation details of Haddock, a shared logging service leveraging the Avalanche blockchain for its immutability and decentralized properties. It details how a smart contract is used to synchronize data access and tracking, using a two-phase dissemination protocol for encrypted messages and their corresponding decryption keys. The workflow involves steps such as message dissemination, acknowledgments, and key dissemination. The system architecture includes both client-side and

server-side components. The server-side handles smart contract interactions, and the client-side provides a REST API and a frontend UI with a Metamask integration for transactions.

5 Evaluation

This chapter presents experiments conducted on Haddock to evaluate the system's properties and assess if they are suitable for a logging and synchronization system. The first section outlines the testing platform, followed by an explanation of the experimental setup and system specifications. Next, the chapter discusses the dissemination experiment, which explores the latency of Avalanches in different regions. Subsequently, a cost analysis investigates the feasibility of the system in terms of fiat currency. Concluded by a memory growth experiment that explores gas cost associated with increased memory usage.

5.1 Platform

We evaluate Avalanche's Fuji test network and investigate how efficiently it can work as a backend server for Haddock. Fuji is designed for smart contract testing, and offers AVAX free of charge through drops, and is intended to mirror the actual price of AVAX at on the C-CHAIN. It is important to notice that AVAX tokens on Fuji hold no real monetary value and are merely used for experimentation purposes. The number of AVAX tokens set some limits for the tests that were conducted. Running extensive tests on Fuji requires a high number of AVAX tokens, while AVA Labs drops 0.5 AVAX a day. Therefore, the smart contract has not been tested to its fullest extent. It is important to mention that the system is designed for public blockchain networks. Hence,

Component	Specification
Processor	13th Gen Intel(R) Core(TM) i7-13700 2.10 GHz
RAM	128GB 3600MHz DDR4
Graphics Card	NVIDIA GeForce GTX 3070
Operating System	Windows 11 Enterprise
Virtual subsystem	Windows Subsystem for Linux 2.0.9.0

Table 5.1: System Specifications

Component	Specification
Processor	Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz
Number of CPU cores	1
RAM	1GB
Operating System	Linux

Table 5.2: System Specifications for each Azure container instance

the Fuji network is chosen as a test platform rather than a locally managed test network because this more closely reflects real-world scenarios.

5.2 Experiment Setup and System Specifications

The dissemination test has been conducted on Azure, with Fuji acting as an API server. The Azure setup consists of nodes deployed using Azure's Container Instance service. The system setup in Azure is presented in Table 5.2. Additionally, the cost analysis has been conducted and run on a desktop with the system specifications presented in Table 5.1.

Functions	AVAX	USD	gas
Contract deployment	0.085156	$3.28	2852801
appendMember	0.00204449	$0.08	68492
removeMember	0.00107388	$0.04	35976
setData (1 byte)	0.00540081	$0.020	196393
acknowledgeData (1 byte)	0.00450263	$0.17	163732
setKey (16 bytes)	0.00796649	$0.30	266884

Table 5.3: The cost of each function on a new empty SHIELD contract

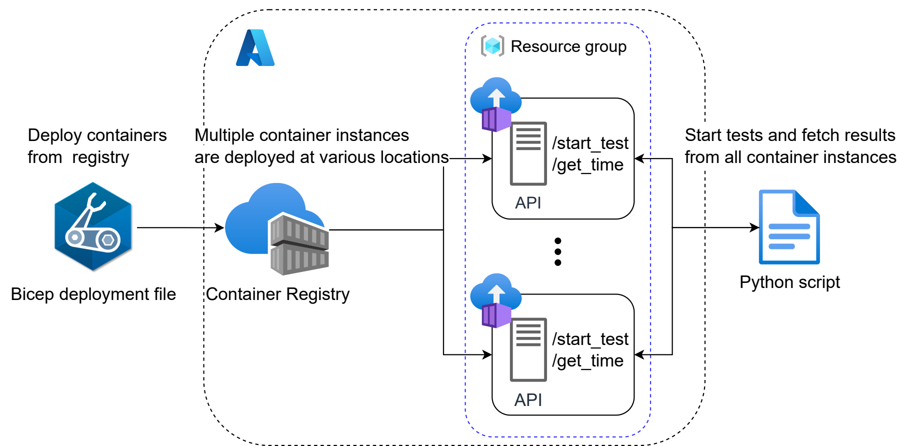

Figure 5.1: A bicep script deploys 30 public container instances to Azure at different regions across Northern America. Each container instance consists of a dissemination test wrapped inside a Docker container. Finally, a Python script calls the API endpoints to start and fetch test results.

5.3 Dissemination Experiment

The dissemination experiment measures the latency of Haddock's blockchain network. It is designed to evaluate the performance of retrieving complete data, which is defined as information that includes the encrypted data and the decryption key for said data. The purpose is to check the mean latency when retrieving data from the blockchain and validate the feasibility of the barrier synchronization that Haddock uses as part of its shared command bus system.

The dissemination experiment is conducted and tested in Azure. Each test is performed three times. Every iteration is tested on an empty, newly deployed contract. This ensures that the gas remains consistent on a given size for each test. Whenever a decryption key is published to the contract, it is emitted as an event. An event listener is set up to listen for this event. The event listener is wrapped in a docker container, which allows deployment to Azure to be done more easily and effectively. A bicep deployment script is used for deployment. The deployment script includes a series of server regions in Northern America, Europe, and Asia, efficiently spreading the container deployments across the globe. The full list can be found in Appendix A.2. This way, 30 containers, each one at one of 20 predefined regions in the world, can be tested. All containers are set up to have API endpoints for starting the event listening and fetching the stored data retrieval time from the concluded listening. The experiment is conducted at multiple byte sizes for the encrypted data to see how this affects the latency when fetching from the contract. The operation of storing and

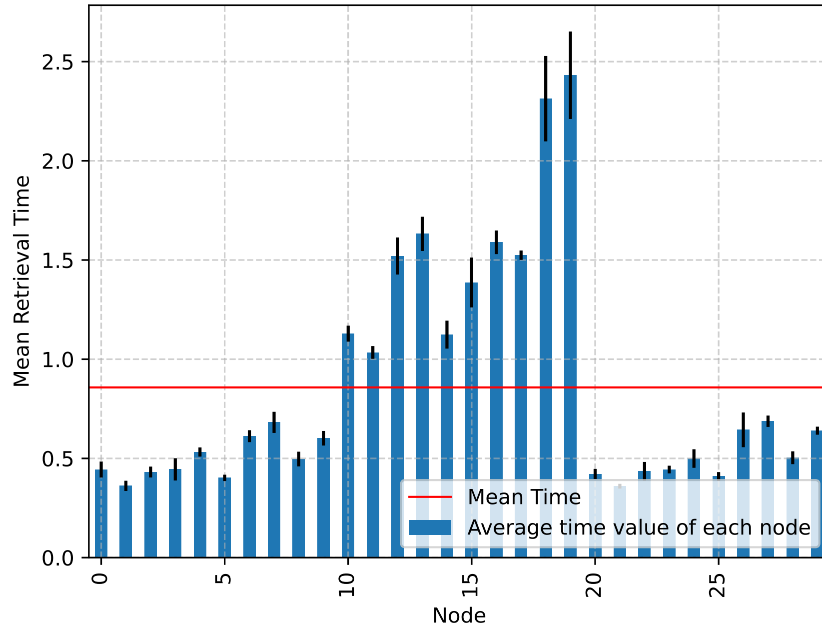

Figure 5.2: The mean retrieval time for all 30 nodes spread across the globe. The plot is based on the table in appendix A.8

disseminating the decryption key involves moving a storage reference from the incomplete data structure to the complete data structure on the contract, which, in theory, should have a relatively constant time complexity.

The dissemination experiment indicates a mean retrieval time of 0.86s on message sizes ranging from 20 to 5000 bytes, as illustrated in Figure 5.2. There is no substantial increase in delay observed within the tested byte interval, which suggests the retrieval time remains relatively consistent, regardless of the size of the data being read from the Avalanche network within the given byte interval. The total overhead of the data packet for a message entry is 343 bytes when the encrypted data and the decryption key are excluded, meaning the real packet size ranges from 396 to 5375 bytes when both the message and 256-bit decryption key are included. Therefore, by minimizing the overhead, the system could potentially lower the latency for smaller messages, as highly critical information should have minimal latency.

Parts of the latency can be due to client-side operations. Each time a key event is emitted. The client fetches a list of completed IDs to cross-check if the ID in the emitted event exists, which acts as an extra measure of security and avoids potential errors on the smart contract. The latency of data retrieval will

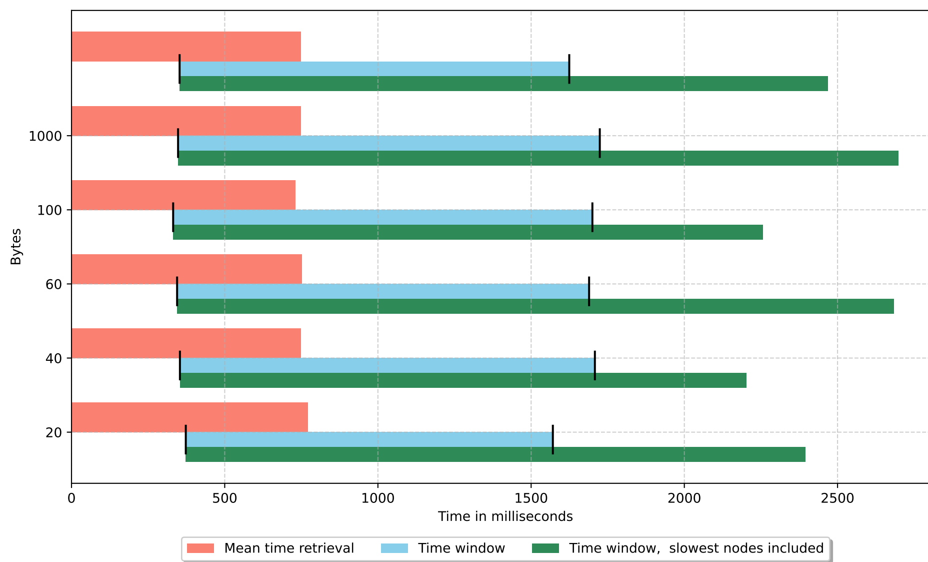

Figure 5.3: The time window from the first received data to the last received data and the mean retrieval time for each byte size

heavily rely on the bandwidth of the recipient and network congestion on the blockchain network.

High latency in a synchronization service is not always problematic, as the most important factor is that end users receive the information in the smallest possible time window, which is the window spanning from the first receiver of data to the last, as depicted in Figure 5.3. This figure presents the average retrieval time for every byte size, the time window during the experiment, and an extended time window, including nodes with the highest latency. However, the results indicate both a high latency, averaging at 0.86 s and a large time window averaging around 1 s reducing the potential applications for Haddock, for instance, some business disciplines like accounting, finance and economics heavily depends on extremely low latency. However, the system can be acceptable for its designed environment, *the sea*, for message dissemination with a tolerable latency of a few seconds.

Given these results, it is crucial to consider the non-functional requirements of availability and responsiveness from Section 3.2.2. The dissemination results demonstrate significant latency variations across different regions, where North America has significantly lower latency compared to nodes in Europe and Asia, which can imply a higher distribution of Avalanche nodes in North America [4]. With variations to responsiveness this substantial between continents, applying such technology at sea may prove challenging, as the variation could become even greater, especially since Internet availability is highly limited at open

sea.

Thorough transaction throughput testing of Avalanche indicates that the average transaction confirmation time is around 0.3 s, with a maximum of 0.4 s [30]. However, the dissemination experiment does not test transactions per second and the latency of transaction confirmation. This needs to be taken into consideration. According to Avalanche's developers, the time to finality is around 1 s [5, 30]. Although this is a short time, the legitimacy of the decryption key cannot be verified, before 1 s has surpassed from the first indication of the decryption key on-chain. The key can still be verified by the recipient off-chain by creating a hash of the encrypted data and comparing it against the verification hash created by the original publisher of the encrypted data.

The dissemination experiment conducted on Haddock provides insight into its performance and feasibility as a synchronization service. While the experiment suggests relatively consistent retrieval times regardless of data size, latency variations across different regions and the challenge of potential high latency at sea present notable considerations for responsiveness. Client-side operations and network congestion further influence data retrieval latency, which again can affect Haddock's responsiveness. Overall, these findings prove that Haddock can be used in systems that accept and tolerate a latency and time window of a few seconds, although it may face considerable availability challenges at sea.

5.4 Cost Analysis Experiment

The cost analysis experiments explore the increasing cost of smart contract functionality at various byte sizes. The purpose of the cost analysis is to assess the growth of expenses as the data's size varies and explore whether public blockchains can be viable and feasible in monetary terms. This experiment mainly focuses on the cost fluctuations across several days.

The cost analysis is done by checking the cost of the core functions on the smart contract at different byte sizes, on an interval between 10 to 3200 bytes, effectively 8 messages per experiment. The core functions include publishing the encrypted data, acknowledging said data, and publishing the decryption key. This analysis has been conducted three times on different dates to include standard deviation. All experiments are done at a new and empty smart contract, in which all functions will have the same gas estimation across the iterations.

The cost analysis experiment signifies a linear growth both for data and key

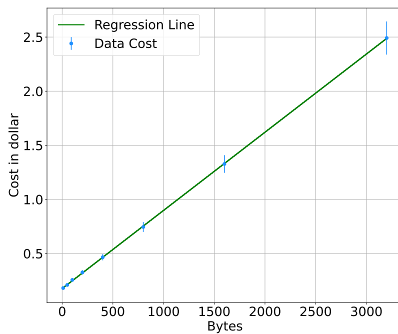

Figure 5.4: The increasing cost of publishing encrypted data at various byte sizes.

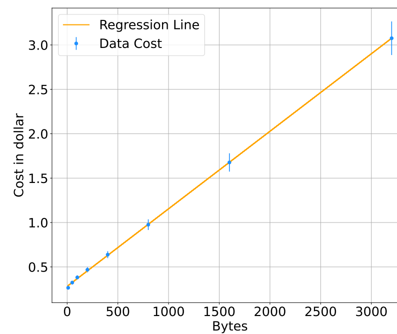

Figure 5.5: The increasing cost of disseminating the decryption key at various byte sizes. This cost will always be higher than the cost of disseminating encrypted messages.

publication for increasing message sizes, depicted in Figure 5.4 and 5.5, and a consistent cost for acknowledgments in Figure 5.6. The increase in gas cost indicates that larger data needs more computation when storing the data, which is expected, and highlights the predictable nature of gas consumption, which strengthens the reliability of the system. In the span of multiple days, the cost for the computation in dollars does not fluctuate much, which can indicate a stable and consistent economic environment within the Avalanche ecosystem, further strengthening the reliability of the system. Publishing complete messages of size 1600 bytes (1943 bytes including overhead) averages at around 2.96 dollars from the demonstrated results. As the gas price grows predictably, large datasets can be expensive to deploy on the Avalanche network in terms of monetary value, suggesting that smaller messages are more feasible for Haddock. Arguably, for crucial data with moderate deployment frequency, the associated cost may be regarded as acceptable, particularly for large businesses and government agencies and institutions.

5.5 Memory Growth Experiment

The smart contract stores all messages in structures on the smart contract. Perhaps one of the most expensive operations a smart contract can execute is writing to storage. As the number of data entries accumulates in the contract's memory, the gas cost of appending the messages can increase. The purpose of this experiment is to measure the gas cost as the data structures grow. The

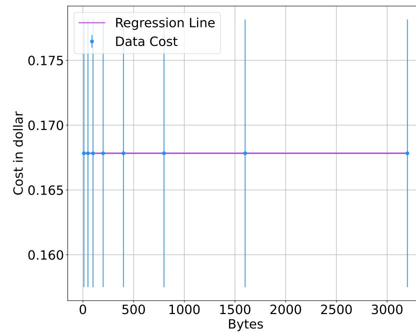

Figure 5.6: The increasing cost of acknowledging data at various byte sizes.

experiment is conducted similarly to the cost analysis in Section 5.4. Messages ranging from 10 to 3200 bytes are deployed, effectively publishing 8 messages to the smart contract. The experiment script has been run 13 times on the same contract to achieve around 100 messages stored on the SHIELD contract. Since gas computation is predictable, error bars are excluded, as they would have been consistent across multiple experiments (Only the cost in dollars would fluctuate).

According to the findings illustrated in Figure 5.7a and 5.8, the gas cost remains relatively consistent at fixed byte sizes across multiple iterations regardless of the data storage and accumulated data struct. This can be a consequence of optimized mappings in Solidity; theoretically, a mapping can support an entry of 32 bytes with 2^{256} keys using Keccak-256 hash [22], with the maximum storage capacity of $(2^{256}) \times 32$ bytes per contract. The overhead of each message is 343 bytes, which theoretically indicates the mapping can store around 2^{253} empty messages. Despite an extremely vast number of storage slots, the available storage on various nodes will significantly vary, which is influenced by hardware and storage capacities, individual node configurations, and the increasing state size as more data is appended to the blockchain. However, theoretically, mappings suggest a favorable storage alternative on Avalanche, as the gas cost remains relatively consistent and predictable for fixed byte sizes when appending to a growing struct, thereby enhancing the reliability of the system.

The results in Figure 5.7b indicate that the accumulated data entries in the incomplete message struct drastically change the gas cost when moving a data entry from the incomplete message struct to the complete message struct. The increased gas cost can be a result of several factors, including storage

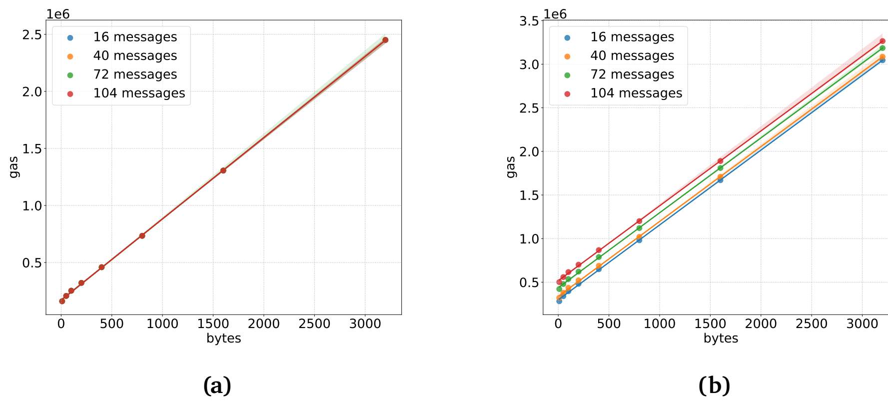

Figure 5.7: The increasing gas cost of publishing encrypted data (a) and decryption key (b) at various byte sizes, as the number of data entries accumulates

writes, deletion, and event emission. During the message transfer, each data entry in the struct must be allocated and copied to the complete struct. The identifier of the message must be appended in the dynamic array in the struct. Finally, the old entry in the incomplete struct must be deleted, involving the removal of the mapping entry and the identifier from the dynamic array, which requires iteration, concluded by emitting an event signaling the publication of the key. This process involves redundant computations similar to those performed during data publication, leading to a consistently higher gas cost for key publication. Interestingly, in the later iterations, the gas cost of storing even smaller messages surpasses that of storing larger messages in earlier iterations, which emphasizes the impact of accumulated data entries on gas costs for key publication, highlighting the importance of efficient storage. To optimize gas cost efficiency, it can be advantageous to use a single struct for both the incomplete and complete messages.

The memory growth experiment highlights the critical impact of data storage and struct design on smart contract efficiency. While mappings in Solidity offer optimized storage capabilities, the gas cost of moving a data entry from one struct to another can significantly increase gas cost due to accumulated data entries. Efficient storage management, such as combining the message structs, can mitigate the rising gas costs. These findings suggest that a struct with mappings is favorable for Haddock, as it has the potential to store significant numbers of fixed size byte entries with a consistent and predictable gas cost.

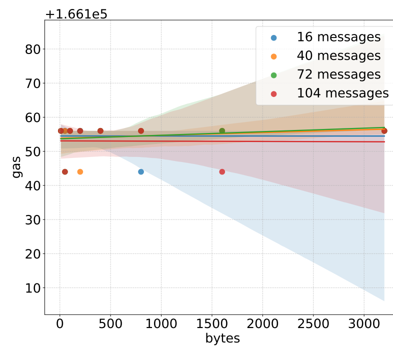

Figure 5.8: The cost of acknowledging messages at various byte sizes as the number of data entries accumulates. The opaque plot illustrates the confidence interval for each iteration

5.6 Summary

The chapter provides an evaluation of Haddock's performance and availability on the Avalanche network, as well as cost implications and smart contract efficiency in terms of storage. The dissemination experiments imply a relatively consistent retrieval time across different byte sizes, although with some notable latency variations across geographic regions, which may present even greater challenges at sea. Meanwhile, the cost analysis experiments indicate consistent and predictable gas costs, with low fluctuation in monetary value in the span of several days. Lastly, the memory growth experiment suggests that mappings support the storage of a vast number of fixed-size byte entries with consistent and predictable gas costs.

6 Discussion

This chapter discusses the design choices, problems, and challenges with the current implementation of Haddock, along with improvements. The first section discusses the use of Haddock solely for key dissemination, followed by optimization with custom subnets. Lastly, an assessment of the non-functional requirements from Section 3.2.

6.1 Solely Key Distribution

In considering alternatives to Haddock's current barrier synchronization mechanism, one approach could involve leveraging Avalanche solely for key distribution, rather than storing data. This adaptation addresses the potential cost concerns when deploying large-sized data to Avalanche and enhances the efficiency of the SHIELD contract. By shifting the storage of encrypted data to external systems like Peer-to-Peer (P2P) and Content Deliver Network (CDN), the overall volume of data stored on the blockchain is significantly reduced, potentially lowering the overall cost of the service. However, this shift of the system's storage means, would remove the logging mechanism, as the system would only disseminate a decryption key. While this approach could optimize gas efficiency, it will compromise the logging feature of the system, which is an important feature of Haddock's current design.

```
struct Messages {
    mapping(uint256 => Message) incomplete;
    mapping(uint256 => Message) complete;
    uint256[] keysIncomplete;
    uint256[] keysComplete;
}
```

Listing 6.1: A single struct including both incomplete and complete messages

6.2 Subnets and Customized Access Control

Haddock is designed to be deployed on Avalanche Primary network's C-CHAIN. A more sophisticated approach would involve using custom subnets deployed on Avalanche, potentially improving performance and latency since fewer transactions need to be validated by partitioning validators in the subnet. Only the transaction within the subnet would require validation.

While leveraging Avalanche's subnet properties for stricter access control, could benefit Haddock. The current design uses function modifiers in Solidity to achieve RBAC. Modifiers are chosen as they minimize complexity, have no extra step when deploying the solution, and increase the interoperability of the smart contract itself, allowing it, theoretically, to be applied to other blockchain networks using EVM or EVM compatible virtual machines.

6.3 Single Struct Approach

The initial design of Haddock separates the incomplete and complete messages on the SHIELD contract. The results from the memory growth experiment in Section 5.5, indicates that mappings have substantial storage capabilities, with consistent and predictable gas costs. Therefore, consolidating the incomplete and complete struct can reduce the storage cost, by eliminating the need to move a message from one mapping to the other upon its completion. The initial concept of separating the structs in Haddock was intended to encapsulate each message type and mitigate the high gas costs associated with increased storage. If the gas cost of appending became large, messages in the struct could be emitted as events, effectively storing them in the transaction receipt, and reducing the storage and gas cost on the struct. With the results from the experiments, there is no reason to keep the structs separated. A single struct solution would include the mappings for both the incomplete and complete messages, in addition to their corresponding keys, stored in dynamic arrays, as depicted in Listing 6.1

The decision to use two structs in Haddock is also a consequence of altered design choices during implementation. Ideally, all of the current features could be achieved with a single struct, which would be a preferable option as it could decrease the overall costs and complexity of Haddock, leading to a more straightforward solution.

6.4 Discussion of Non-Functional Requirements

This section presents a brief summation of the non-functional requirements from Section 3.2, followed by a discussion on how the system conforms to these requirements.

Non-functional:

1. **Reliability**: Perform system functions and services in a consistent and accurate manner

2. **Responsiveness and availability**: The system's ability to respond to user interactions and return results on inputs and requests, and the system's capabilities to deliver and provide services consistently

3. **Security and resilience**: Measures to prevent unauthorized access and address security threats, leveraging blockchain technology, while maintaining highly available services and being resilient to potential failures and attacks

4. **Usability**: The design of the UI should be intuitive to ensure ease of use.

5. **Maintainability**: The complexity of adding, editing, and fixing features in a system

The experiments from Section 5.4 and 5.5 contribute to strengthening the *Reliability* of the system. These findings demonstrate a stable economic environment within the Avalanche network, along with consistent and predictable gas cost and gas consumption across EVM's data structures. This reliability ensures that Avalanche delivers its services in a predictable manner. The frontend components in the UI are designed to accurately transmit all information to Avalanche via an API service, thereby ensuring correct processing of all data and mitigating the risk of conflicting data types. However, the frontend's key dissemination mechanism lacks a backup, posing a risk of failure and potentially preventing key dissemination, which is a significant consideration regarding the reliability of Haddock. Despite the key dissemination mechanism,

these aspects collectively contribute to the reliability requirement.

Responsiveness and availability is crucial for Haddock to operate accordingly. The results from Section 5.3 give insight into the responsiveness and availability of Avalanche, which provides various latencies across large geographical regions, indicating that the distribution of nodes affects these properties. Overall, the responsiveness appears adequate for a system that tolerates a few seconds of latency. However, the substantial latency variation across the continents may suggest even greater variations in offshore environments, potentially posing a notable challenge for the availability of the Haddock.

Security and resilience measures in Haddock are provided by the inherent security features of blockchain technology. Blockchains use cryptography to ensure confidentiality and integrity of data. Each block in a blockchain is cryptographically hashed and linked to the previous block, making it more resistant to tampering and increasing non-repudiation. Additionally, consensus mechanisms, an essential mechanism for blockchains, validate transactions and detect fraudulent records, ensuring only transactions redeemed valid, are appended to the ledger. Haddock implements RBAC to control the access to the SHIELD contracts functionality. Membership management within the SHIELD contract is restricted to the contract owner, who exclusively holds the authority to add and modify members, thereby minimizing the potential for adversaries to infiltrate the logging system, enhancing overall security, resilience, and integrity. Members are granted full access to all functions within SHIELD contract. However, the functions are designed to preserve the contract's state, thereby preventing any unauthorized alterations, except for specific operations such as disseminating data, decryption keys, and acknowledgments. The user's private key is stored in plaintext inside the local storage of the client, which is not sufficient and fails to provide adequate protection, leaving it susceptible to unauthorized access and potential compromise. As Haddock is primarily a demonstrator for a logging infrastructure, this practice is sufficient for a demonstrator system

Haddock's *Useability* is of utmost importance, as it should be easy to use and navigate in offshore environments. The publication window presents and displays the input in an understandable and intuitive manner as a consequence of limiting the available features to the end user at any time[10]. Each button includes a tooltip with informative text when hovered over, and the message list for incomplete and complete messages, which consist of a unique message panel, are clearly separated, increasing the overall user experience and ensuring the end users can distinguish between different message states. Acknowledging a message may not be the most user-friendly solution, as the button in the message panel is quite small and represented by an icon of a key. This icon can be easily mistaken for a message tag or a symbol indicating that

the message is encrypted, potentially causing confusion. In summary, these usability considerations contribute to better user integration and are sufficient for Haddock's useability requirement.

Maintainability is a crucial factor for blockchain-related systems. Due to its nascent nature, the technology evolves rapidly, continuously exploring new applications. Modern blockchain systems are highly complex, which means they require a structured approach to ensure they remain maintainable and adaptable over time. Haddock addresses maintainability by implementing a single smart contract, complemented by documentation and adhering to best practices. The frontend components are designed to be independent, making them easy to update or replace. While Haddock currently features some temporary solutions and altered design choices, it is sufficient for a demonstrator system.

6.5 Summary

The chapter discusses alternative approaches and optimizations for Haddock. An alternative approach would leverage Avalanche solely for decryption key distribution, suggesting the use of external systems like P2P and CDN for encrypted data storage, as it could reduce the overall cost. Furthermore, the chapter discusses the use of custom subnets in Avalanche for better performance and access control and the possibility of consolidating the incomplete and complete message struct into a single struct to reduce extra gas costs and complexity. Lastly, we assess how Haddock meets the non-functional requirements stated in Section 3.2, encompassing reliability, responsiveness and availability, security and resilience, usability, and maintainability while acknowledging some compromises and temporary solutions that are suitable for a demonstrator system.

7 Concluding Remarks

This chapter outlines Haddock's contributions and objectives in line with the problem definition for this thesis, along with the findings. It also includes a concise overview of related work. The chapter concludes with final remarks and suggestions for future research on the system. Revisiting the problem definition outlined in Section 1.1:

Reconstructing the order of events with high-integrity evidence is crucial for suppressing fishery-related criminal activities on the open sea. Blockchain's immutability, transparency, and distributed properties can be valuable for logging on-ship events and disseminating critical fleet command center reports. A secure, distributed logging infrastructure could serve as a valuable component within a broader system of modern surveillance tools.

Our thesis is that

A secure, shared logging infrastructure for fishery fleet command can be implemented using Blockchain smart contracts.

7.1 Contributions and Goals

In this thesis, Haddock explores the use of a logging infrastructure in the fishery and maritime domain and contributes by implementing a logging service, using

a smart contract-based system, on a public blockchain network, which can be extended to other EVM-compatible systems. The system offers non-repudiation for messages and timely dissemination of log entries of different severity levels. Haddock is unique compared to the reviewed papers, in which it provides a lightweight solution in terms of a single, smart contract for logging and disseminating messages rather than extensive frameworks or architectures. In addition, Haddock includes a UI, which can make the dissemination and acknowledgment of messages efficient and understandable by end users, along with displaying and accessing the logged data.

To ensure the system aligns with the problem definition, the requirements specified in Chapter 3 must be fulfilled. Haddock allows each end user to disseminate messages concurrently to a public, available blockchain, Avalanche (requirement 1), in which the system symmetrically encrypts the messages with AES(requirement 2) before message dissemination. The SHIELD contract's function modifiers allow for RBAC to Haddock's functionalities, with two roles, owner and members (requirement 3). This controlled access, in combination with recording the publishers and acknowledgers of the messages on an immutable blockchain, enhances the non-repudiation of the system (requirement 4). The two-phase dissemination protocol, combined with barrier synchronization, allows for timely dissemination of the messages(requirement 5). Although with some challenges related to the synchronization options. Haddock's services, including disseminating encrypted messages and decryption keys, acknowledging messages, and reading the messages, are available through a frontend User Interface (requirement 6). In addition, the frontend integrates the crypto wallet Metamask to let the end users sign and confirm transactions through their connected wallet (requirement 7).

In Section 6.4, we assessed the non-functional requirements for Haddock. In summary, *reliability* is achieved by consistent and predictable gas cost for the functionalities provided by the SHIELD contract and by ensuring correct processing of data between the components in the system. However, reliability is compromised due to the absence of backup measures in the key dissemination mechanism in the frontend. The evaluated results in Section 5.3 indicate various message retrieval latencies across large geographical regions, suggesting that the variations in offshore environments may be even greater, posing potential *availability* problems for Haddock. However, the *responsiveness* proves adequate for systems that tolerate a few seconds of delay. Haddock leverages the inherent properties in blockchain systems to provide *security and resilience*, along with function modifiers for access control but fails to store the private keys securely within the system, but the applied practice is considered sufficient in a demonstrator system. Haddock achieves *useability* by limiting the available features to the end user at any time and clearly distinguishing the frontend components. Lastly, Haddock addresses *maintainability* by providing

all blockchain-related functions in a single, documented smart contract, adhering to best practices. However, Haddock features some temporary solutions due to altered design choices, which is sufficient for a demonstrator system.

7.2 Related Work

Similar to Haddock, *Logchain: Blockchain-Assisted Log Storage* [29] is a blockchain-assisted logging system providing an API service for storing logs. The paper presents *Logchain*, a prototype logging service referred to as Logchain-as-a-service (LCaaS). An API service for storing logs or log hashes in a hierarchical immutable ledger. The API service allows the client to send, verify, and retrieve data from the distributed storage. Pourmajidi and Miranskyy argue that log tamper detection software exists but is not adequate for Cloud-based solutions [29]. Therefore, the primary use for the LCaaS is to store technical logs from Cloud solutions in a tamper-proof environment, as Cloud-based solutions can be vulnerable to tampering. LCaaS is a prototype framework designed to sit on top of blockchains, transforming it into a two-level hierarchical ledger that can be expanded if necessary. The motivation for a hierarchical ledger is to decrease the validation process of consensus protocols that require every node to process every block in the network. The higher-level blocks on the ledger consist of a locked portion of the blockchain. All lower-level blocks are validated in the system by validating a high-level block, minimizing the number of blocks to process, and increasing scalability [29], which is especially important for systems where the log data can be extensive. For future studies, the authors plan to test the implemented LCaaS with existing blockchain systems and find possible solutions to integrate their blockchain framework onto these systems.

ProvChain: A Blockchain-based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability [23] presents *ProvChain*, a blockchain architecture for collecting and recording provenance data. ProvChain explores embedding the provenance data into Merkle trees [24]. A series of provenance data represents the leaf nodes and is the fundamental hashes that construct the Merkle tree. The Merkle root is the top hash, representing the list of provenance hashes included in the blockchain transaction. By using this approach, ProvChain leverages the same validation principles as transactions in blockchain systems but with provenance data. Consequently, altering a piece of provenance data in a block requires modifying all preceding blocks to ensure the hashes correlate, which can be infeasible in many blockchain systems. Their evaluation of the system demonstrates an acceptable distribution time for transactions, and the retrieval time for provenance data is averaged at 221ms at 1.004 KB [23]. For future work, the paper addresses increased interoperability

with different cloud providers and implements receipt validation with an open-source architecture to improve performance, flexibility, and security.

A New Secure Data Dissemination Model in Internet of Drones [1], Aggar *et al.* proposes a system model utilizing blockchain technology in Internet of Drones (IOD) systems to enhance data integrity, accountability, authorization, authentication (AAA), data confidentiality and non-repudiation. Similar to Haddock, the authors investigate Ethereum-based blockchain technology for data dissemination. They explore how this technology, combined with POS consensus mechanisms, can be used for selecting a forger node, creating and validating blocks, and disseminating data in an IOD environment. The forger nodes are selected with an algorithm using Game theory; further, POS is responsible for both the creation and validation process of a new block, with the forger node generating the block hash for POS. Lastly, the forger node is responsible for the dissemination of data among drones, which is done in multiple steps:

- Send encrypted data to the blockchain network
- Blockchain accepts data and updates the ledger
- The forger generates and sends a digital signature to the blockchain
- The blockchain verifies the digital signature
- Blockchain forwards encrypted data to participants
- Participants decrypt data with the forger's public key and send an acknowledgment

The authors argue that the system model is superior in computation cost and time when evaluating and comparing to other systems [1]. They plan to explore using different platforms, particularly private permissioned blockchains, for future research.

7.3 Concluding Remarks

In this thesis, we have designed, implemented, and evaluated Haddock, a shared, distributed logging service using Avalanche, according to the problem definition. The intention of this thesis was to explore a command bus system for logging critical information from command fleets in the maritime field by using blockchain and smart contract technology for its transparency, immutability,

and decentralized properties.

Haddock consists of a server-side using the SHIELD contract on Avalanche and a client-side exposing a frontend and a REST API to interact with the server-side, serving as a logging infrastructure with a two-phase dissemination protocol combined with a barrier synchronization mechanism to achieve timely message dissemination. In the first phase the message is disseminated in an encrypted format, waiting for a predetermined number of acknowledgments, the second phase commences when a quorum is met and disseminate the decryption key. Haddock presents a UI to interact seamlessly with the dissemination services provided by the SHIELD contract, offering techniques for message dissemination to Avalanche, browsing message logs, and acknowledging incomplete messages.

Through extensive experiments and evaluations, this thesis, despite the limitations, demonstrates the feasibility and potential of using blockchain technology for logging and timely dissemination of messages within systems that tolerate a latency of a few seconds. The findings provide a robust foundation for future research on the application of blockchain technology within the fishing sector, particularly in enhancing the security, transparency, and efficiency of logging and disseminating critical information.

7.4 Future Work

While Haddock demonstrates potential by fulfilling the specified functional and non-functional requirements, several areas require further investigation and development to enhance the system and the findings in this study. Future work for Haddock should consider consolidating the `incompleteMessages` and `completeMessages` structs, as this can significantly reduce the overall gas costs associated with the system. Additional research should focus on integrating more sophisticated access control mechanisms, potentially through custom subnets on the Avalanche network. Furthermore, a more robust mechanism for key dissemination in the frontend should be explored, ensuring a higher guarantee of key dissemination. Lastly, exploring Haddock's compatibility with other blockchain systems would improve its interoperability.

Bibliography

[1] Shubhani Aggarwal et al. “A New Secure Data Dissemination Model in Internet of Drones.” In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8761372.

[2] Bobby Allyn. *People are talking about Web3. Is it the Internet of the future or just a buzzword?* Accessed: 2024-05-22. 2021. URL: https://www.npr.org/2021/11/21/1056988346/web3-internet-jargon-or-future-vision.

[3] Joakim Aalstad Alslie et al. “Áika: A Distributed Edge System for AI Inference.” In: *Big Data and Cognitive Computing* 6.2 (2022). ISSN: 2504-2289. DOI: 10.3390/bdcc6020068. URL: https://www.mdpi.com/2504-2289/6/2/68.

[4] “Analyzing the Distribution and Growth of AVAX Stake in the Avalanche Ecosystem.” In: (). Accessed: 2024-05-28. URL: https://medium.com/@avascan/analyzing-the-distribution-and-growth-of-avax-stake-in-the-avalanche-ecosystem-873a93a28b29.

[5] Usman Asim. *Time to Finality (TTF): The Ultimate Metric for Blockchain Speed*. 05.13.24. No date. URL: https://www.avax.network/blog/time-to-finality-ttf-the-ultimate-metric-for-blockchain-speed.

[6] Daniel Bardey. “Overfishing: pressure on our oceans.” In: *Research in Agriculture Livestock and Fisheries* 6 (Jan. 2020), pp. 397–404. DOI: 10.3329/ralf.v6i3.44805.

[7] Vitalik Buterin. “Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform.” In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Coinbase. *Ethereum (ETH) price, charts, and market cap*. Accessed: 2024-05-21. 2024. URL: https://www.coinbase.com/price/ethereum.

[9] Christopher Costello et al. “The future of food from the sea.” In: *Nature* 588.7836 (2020), pp. 95–100.

[10] Ali Darejeh and Dalbir Singh. “A review on user interface design principles to increase software usability for users with less computer literacy.” In: *Journal of Computer Science* 9 (Nov. 2013), pp. 1443–1450. DOI: 10.3844/jcssp.2013.1443.1450.

[11] P.J. Denning et al. “Computing as a discipline.” In: *Computer* 22.2 (1989), pp. 63–70. DOI: 10.1109/2.19833.

[12] Omar Dib et al. "Consortium Blockchains: Overview, Applications and Challenges." In: (Sept. 2018).

[13] Cynthia Dwork and Moni Naor. "Pricing via processing or combatting junk mail." In: *Annual international cryptology conference*. Springer. 1992, pp. 139–147.

[14] Ethereum. *The Merge*. Accessed: 2024-05-21. 2024. URL: https://ethereum.org/en/roadmap/merge/.

[15] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process." In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411. DOI: 10.1145/3149.214121. URL: https://doi.org/10.1145/3149.214121.

[17] M. Goodrich and R. Tamassia. *Introduction to Computer Security*. Aways learning. Pearson Education Limited, 2013, pp. 27, 35–36, 399–404. ISBN: 9781292025407. URL: https://books.google.no/books?id=NPYsngEACAAJ.

[18] Håvard Johansen, Dag Johansen, and Robbert van Renesse. "FirePatch: Secure and Time-Critical Dissemination of Software Patches." In: *New Approaches for Security, Privacy and Trust in Complex Environments*. Ed. by Hein Venter et al. Boston, MA: Springer US, 2007, pp. 373–384. ISBN: 978-0-387-72367-9.

[19] "JSON-RPC." In: (). Accessed: 2024-05-26. URL: https://www.jsonrpc.org.

[20] Varun Kohli et al. "An analysis of energy consumption and carbon footprints of cryptocurrencies and possible solutions." In: *Digital Communications and Networks* 9.1 (2023), pp. 79–89. ISSN: 2352-8648. DOI: https://doi.org/10.1016/j.dcan.2022.06.017. URL: https://www.sciencedirect.com/science/article/pii/S2352864822001390.

[21] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[22] "Layout of State Variables in Storage." In: (). Accessed: 2024-05-28. URL: https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html.

[23] Xueping Liang et al. "ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability." In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2017, pp. 468–477. DOI: 10.1109/CCGRID.2017.8.

[24] Ralph C. Merkle. "Protocols for Public Key Cryptosystems." In: *1980 IEEE Symposium on Security and Privacy*. 1980, pp. 122–122. DOI: 10.1109/SP.1980.10006.

[25] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008.

[26] Tor-Arne S. Nordmo et al. "Dutkat: A Multimedia System for Catching Illegal Catchers in a Privacy-Preserving Manner." In: *Proceedings of the 2021 ACM Workshop on Intelligent Cross-Data Analysis and Retrieval*. ICDAR '21. Taipei, Taiwan: Association for Computing Machinery, 2021, pp. 57–61. ISBN: 9781450385299. DOI: 10.1145/3463944.3469102. URL: https://doi.org/10.1145/3463944.3469102.

[27] Tor-Arne Schmidt Nordmo et al. "Njord: A Fishing Trawler Dataset." In: *Proceedings of the 13th ACM Multimedia Systems Conference*. MMSys '22. Athlone, Ireland: Association for Computing Machinery, 2022, pp. 197–202. ISBN: 9781450392839. DOI: 10.1145/3524273.3532886. URL: https://doi.org/10.1145/3524273.3532886.

[28] Wouter Penard and Tim van Werkhoven. "On the secure hash algorithm family." In: *Cryptography in context* (2008), pp. 1–18.

[29] William Pourmajidi and Andriy Miranskyy. "Logchain: Blockchain-assisted log storage." In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 978–982.

[30] Team Rocket et al. *Scalable and Probabilistic Leaderless BFT Consensus through Metastability*. 2020. arXiv: 1906.08936 [cs.DC].

[31] Yannick Rousseau et al. "Evolution of global marine fishing fleets and the response of fished resources." In: *Proceedings of the National Academy of Sciences* 116 (May 2019), p. 201820344. DOI: 10.1073/pnas.1820344116.

[32] Kevin Sekniqi et al. "Avalanche Platform." In: (2020). Accessed: 2024-05-24. URL: https://www.avalabs.org/whitepapers.

[33] Saurabh Singh, A. S. M. Hosen, and Byungun Yoon. "Blockchain Security Attacks, Challenges, and Solutions for the Future Distributed IoT Network." In: *IEEE Access* PP (Jan. 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3051602.

[34] Siamak Solat, Philippe Calvez, and Farid Naıt-Abdesselam. "Permissioned vs. Permissionless Blockchain: How and Why There Is Only One Right Choice." In: *J. Softw.* 16.3 (2021), pp. 95–106.

[35] I. Sommerville. *Engineering Software Products: An Introduction to Modern Software Engineering*. Pearson, 2020, pp. 95–98. ISBN: 9780135210642. URL: https://books.google.no/books?id=M2kNuwEACAAJ.

[36] Enrico Tedeschi et al. "On Optimizing Transaction Fees in Bitcoin Using AI: Investigation on Miners Inclusion Pattern." In: *ACM Trans. Internet Technol.* 22.3 (July 2022). ISSN: 1533-5399. DOI: 10.1145/3528669. URL: https://doi.org/10.1145/3528669.

[37] Enrico Tedeschi et al. "Predicting Transaction Latency with Deep Learning in Proof-of-Work Blockchains." In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 4223–4231. DOI: 10.1109/BigData47090.2019.9006228.

[38] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem." In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: https://doi.org/10.1112/

plms/s2-42.1.230. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[39] UNODC. "Fisheries Crime: transnational organized criminal activities in the context of the fisheries sector." In: (2016).

[40] "Upgrading smart contracts." In: (2023). Accessed: 2024-05-26. URL: https://ethereum.org/en/developers/docs/smart-contracts/upgrading/.

[41] "web3.js - Ethereum JavaScript API." In: (2024). Accessed: 2024-05-26. URL: https://web3js.readthedocs.io.

[42] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger." In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[43] Kaidong Wu et al. "A first look at blockchain-based decentralized applications." In: *Software: Practice and Experience* 51.10 (2021), pp. 2033–2050. DOI: https://doi.org/10.1002/spe.2751. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2751. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2751.

[44] J. Zhou and K.Y. Lam. "Securing digital signatures for non-repudiation." In: *Computer Communications* 22.8 (1999), pp. 710–716. ISSN: 0140-3664. DOI: https://doi.org/10.1016/S0140-3664(99)00031-6. URL: https://www.sciencedirect.com/science/article/pii/S0140366499000316.

Appendix A

```
1  [
2      {
3          "severity": "info",
4          "content": "b\"6\\x1a\\x15\\x0ef\\x98'o...",
5          "id": "4",
6          "timestamp": "2024-05-21T14:58:26",
7          "setter": "0x0319423b8163a4eb...",
8          "signers": [],
9          "ack_num": "32"
10     },
11     {
12         "severity": "info",
13         "content": "Fishing field: Area 27.
              ↪ Atlantic, Northeast at Subarea
              ↪ 27.1\nBarents Sea (Subarea I).
              ↪ Barents Sea. NEAFC Regulatory
              ↪ Area\nClosing time: 05/18/2024 17:00
              ↪ GMT +00:00",
14         "id": "3",
15         "timestamp": "2024-05-18T14:15:30",
16         "setter": "0x0319423b8163a4eb08F950F0...",
17         "signers": [
18             {
19                 "setter":
                      ↪ "0x0319423b8163a4eb08F95...",
20                 "data_hash":
                      ↪ "b'\\xf8\\xa6\\xe9\\xb..."
21             }
22         ],
23         "ack_num": "14"
24     }
25 ]
```

Listing A.1: An incomplete and complete JSON entry. They have different data stored on the "content" key. The incomplete entry includes encrypted data, while th compelete includes plaintext.

Table A.1: Data retrieval time in milliseconds at different sizes for each node. Each time entry is the mean value of three separate dissemination tests

Azure Nodes	Time retrieval in milliseconds at different sizes					
by region	20 bytes	40 bytes	60 bytes	100 bytes	1000 bytes	5000 bytes
westus	512	430	432	388	449	450
westus2	406	353	355	331	347	374
eastus	431	427	415	447	473	393
eastus2	516	386	513	404	425	421
centralus	555	532	510	500	539	556
northcentralus	409	415	405	375	390	415
southcentralus	653	566	594	633	611	613
westcentralus	745	695	657	615	638	736
canadacentral	480	472	470	500	569	487
canadaeast	568	582	614	561	654	628
northeurope	1158	1074	1164	1089	1165	1119
westeurope	1069	1020	1016	987	1041	1068
eastasia	1570	1469	1689	1448	1478	1464
southeastasia	1548	1682	1511	1699	1724	1624
japaneast	1136	1115	1254	1106	1071	1057
japanwest	1348	1570	1389	1307	1220	1485
australiaeast	1549	1708	1559	1577	1558	1581
australiasoutheast	1539	1513	1544	1483	1523	1542
centralindia	2395	2165	2205	2106	2699	2306
southindia	2279	2201	2683	2256	2698	2468
westus	456	394	444	424	396	412
westus2	372	367	344	370	350	352
eastus	516	435	435	396	444	383
eastus2	449	458	436	467	439	412
centralus	527	542	538	436	503	447
northcentralus	438	417	410	419	391	377
southcentralus	720	519	554	648	717	703
westcentralus	700	710	673	661	653	724
canadacentral	539	482	474	515	540	469
canadaeast	660	606	631	656	631	651

Server Regions
westus
westus2
eastus
eastus2
centralus
northcentralus
southcentralus
westcentralus
canadacentral
canadaeast
northeurope
westeurope
eastasia
southeastasia
japaneast
japanwest
australiaeast
australiasoutheast
centralindia
southindia

Table A.2: List of Server Regions

Table A.3: Data retrieval time in milliseconds at 20 bytes with three tests

Azure Nodes	Time retrieval in milliseconds at 20 bytes		
by region	1	2	3
westus	0.653816	0.430926	0.452351
westus2	0.423772	0.378011	0.417188
eastus	0.498257	0.421682	0.374478
eastus2	0.666531	0.425452	0.456287
centralus	0.586758	0.475382	0.604237
northcentralus	0.464274	0.414568	0.348681
southcentralus	0.656229	0.671235	0.632442
westcentralus	0.836115	0.689630	0.710989
canadacentral	0.548706	0.476738	0.416790
canadaeast	0.611386	0.521861	0.573258
northeurope	1.273008	1.124370	1.079420
westeurope	1.114033	1.056695	1.039057
eastasia	1.728379	1.486474	1.497919
southeastasia	1.594974	1.521046	1.530628
japaneast	1.241478	1.054329	1.113982
japanwest	1.182526	1.570595	1.292110
australiaeast	1.649756	1.505727	1.493820
australiasoutheast	1.602887	1.499218	1.515307
centralindia	2.173141	2.321677	2.692048
southindia	2.133899	2.428116	2.276651
westus	0.500123	0.400780	0.469696
westus2	0.447044	0.306554	0.365387
eastus	0.640451	0.409777	0.499081
eastus2	0.476721	0.481055	0.391415
centralus	0.587013	0.457803	0.538498
northcentralus	0.583534	0.404588	0.328656
southcentralus	0.784209	0.722701	0.655506
westcentralus	0.818615	0.628835	0.655344
canadacentral	0.575677	0.486782	0.555888
canadaeast	0.752503	0.669158	0.559652

Table A.4: Data retrieval time in milliseconds at 40 bytes with three tests

Azure Nodes by region	Time retrieval in milliseconds at 40 bytes		
	1	2	3
westus	0.418668	0.428980	0.443872
westus2	0.352107	0.364407	0.345018
eastus	0.441919	0.407476	0.431913
eastus2	0.483842	0.341803	0.334252
centralus	0.538278	0.556167	0.502906
northcentralus	0.401896	0.427336	0.416046
southcentralus	0.756222	0.483352	0.460240
westcentralus	0.604506	0.830090	0.653194
canadacentral	0.510904	0.409403	0.497857
canadaeast	0.535795	0.578228	0.633763
northeurope	1.104910	1.098014	1.021526
westeurope	1.069781	1.000278	0.991171
eastasia	1.420596	1.511996	1.475608
southeastasia	1.629826	1.913385	1.503067
japaneast	1.137339	1.123535	1.085362
japanwest	1.547464	1.581256	1.581719
australiaeast	1.575343	1.995000	1.553969
australiasoutheast	1.514370	1.514196	1.510783
centralindia	2.060990	2.383210	2.053410
southindia	2.321744	2.319707	1.963906
westus	0.341747	0.413098	0.428774
westus2	0.299900	0.458428	0.344571
eastus	0.431943	0.493350	0.382186
eastus2	0.495390	0.406987	0.473117
centralus	0.457493	0.705269	0.465786
northcentralus	0.388494	0.438460	0.425373
southcentralus	0.458426	0.440051	0.659829
westcentralus	0.625050	0.874566	0.632403
canadacentral	0.435650	0.528176	0.482669
canadaeast	0.594836	0.589558	0.636042

Table A.5: Data retrieval time in milliseconds at 60 bytes with three test

Azure Nodes	Time retrieval in milliseconds at 60 bytes		
by region	1	2	3
westus	0.428640	0.408962	0.459565
westus2	0.328417	0.372782	0.366750
eastus	0.381552	0.426900	0.437227
eastus2	0.530631	0.535942	0.475092
centralus	0.526180	0.441233	0.562677
northcentralus	0.366296	0.421471	0.430093
southcentralus	0.607237	0.574453	0.601507
westcentralus	0.698268	0.621045	0.652848
canadacentral	0.510035	0.452081	0.450766
canadaeast	0.673053	0.581960	0.587749
northeurope	1.080499	1.335136	1.077986
westeurope	1.024705	1.014826	1.009891
eastasia	1.569959	1.478433	2.019201
southeastasia	1.508278	1.477208	1.548272
japaneast	1.351041	1.319591	1.094337
japanwest	1.060351	1.796032	1.310678
australiaeast	1.520545	1.578897	1.578886
australiasoutheast	1.518950	1.580656	1.534425
centralindia	2.046000	2.433708	2.137912
southindia	2.757177	2.349610	2.944310
westus	0.443069	0.444532	0.446465
westus2	0.393060	0.301571	0.339234
eastus	0.449527	0.403620	0.454320
eastus2	0.443161	0.408092	0.458383
centralus	0.475159	0.544895	0.596195
northcentralus	0.368533	0.430186	0.433984
southcentralus	0.474372	0.596523	0.593709
westcentralus	0.754914	0.643340	0.622686
canadacentral	0.445986	0.521627	0.455960
canadaeast	0.629190	0.593609	0.672806

Table A.6: Data retrieval time in milliseconds at 100 bytes with three test

Azure Nodes by region	Time retrieval in milliseconds at 100 bytes		
	1	2	3
westus	0.448518	0.391438	0.325994
westus2	0.358077	0.304269	0.332882
eastus	0.533483	0.323418	0.486486
eastus2	0.356407	0.507841	0.350291
centralus	0.535391	0.521807	0.444002
northcentralus	0.405090	0.317422	0.402495
southcentralus	0.593603	0.652861	0.652828
westcentralus	0.585555	0.628617	0.632261
canadacentral	0.551343	0.430357	0.518705
canadaeast	0.542289	0.580865	0.561099
northeurope	1.123371	1.033598	1.111124
westeurope	0.964760	0.998955	1.000034
eastasia	1.437026	1.452760	1.456938
southeastasia	1.549204	1.645556	1.905099
japaneast	1.100692	1.098126	1.122061
japanwest	1.307368	1.310585	1.303541
australiaeast	1.558487	1.599541	1.575508
australiasoutheast	1.507329	1.477768	1.464238
centralindia	2.075695	2.065580	2.178575
southindia	2.263776	2.243897	2.260729
westus	0.425564	0.399106	0.448143
westus2	0.335454	0.375269	0.401160
eastus	0.397998	0.439445	0.352498
eastus2	0.453716	0.442980	0.505250
centralus	0.433625	0.483928	0.391244
northcentralus	0.424192	0.365465	0.468739
southcentralus	0.635488	0.648676	0.660951
westcentralus	0.678537	0.648553	0.655922
canadacentral	0.408553	0.595722	0.540981
canadaeast	0.714490	0.614714	0.639587

Table A.7: Data retrieval time in milliseconds at 1000 bytes with three tests

Azure Nodes by region	Time retrieval in milliseconds at 1000 bytes		
	1	2	3
westus	0.484753	0.351765	0.512577
westus2	0.390529	0.299417	0.353433
eastus	0.576793	0.370673	0.472646
eastus2	0.517505	0.380188	0.379106
centralus	0.624613	0.492993	0.499935
northcentralus	0.391212	0.380035	0.399892
southcentralus	0.699130	0.574972	0.561759
westcentralus	0.671207	0.585750	0.657457
canadacentral	0.505907	0.702439	0.499160
canadaeast	0.637924	0.678384	0.647450
northeurope	1.153718	1.191575	1.152282
westeurope	1.053235	1.012551	1.058360
eastasia	1.461322	1.540848	1.432600
southeastasia	1.571697	1.557352	2.043291
japaneast	1.119560	1.038545	1.056766
japanwest	1.064967	1.572503	1.023435
australiaeast	1.555775	1.581705	1.538988
australiasoutheast	1.469888	1.596856	1.503390
centralindia	2.095376	3.318131	2.683805
southindia	2.265394	3.473813	2.355969
westus	0.450713	0.405325	0.333813
westus2	0.399777	0.328000	0.324284
eastus	0.522205	0.454617	0.357964
eastus2	0.418316	0.420650	0.478577
centralus	0.585116	0.510616	0.414191
northcentralus	0.468719	0.368115	0.337767
southcentralus	0.709453	0.728653	0.714851
westcentralus	0.741359	0.603754	0.615028
canadacentral	0.600617	0.569735	0.449889
canadaeast	0.645989	0.647335	0.599901

Table A.8: Data retrieval time in milliseconds at 5000 bytes with three tests

Azure Nodes by region	Time retrieval in milliseconds at 5000 bytes		
	1	2	3
westus	0.494938	0.423876	0.432344
westus2	0.363119	0.411012	0.347964
eastus	0.438061	0.395766	0.346503
eastus2	0.378506	0.427613	0.459582
centralus	0.526835	0.500740	0.641973
northcentralus	0.461719	0.413147	0.371725
southcentralus	0.653625	0.599630	0.586337
westcentralus	0.719317	0.672185	0.818940
canadacentral	0.509619	0.525360	0.427245
canadaeast	0.530501	0.703984	0.649985
northeurope	1.157565	1.074704	1.125833
westeurope	1.160166	1.055225	0.990543
eastasia	1.472810	1.464829	1.456184
southeastasia	1.489551	1.480372	1.903391
japaneast	1.090606	1.042110	1.040818
japanwest	1.601225	1.282079	1.573684
australiaeast	1.585145	1.586277	1.573854
australiasoutheast	1.559159	1.520343	1.548271
centralindia	2.808739	2.049928	2.061371
southindia	2.243023	2.895147	2.267336
westus	0.352493	0.436058	0.450315
westus2	0.351158	0.372502	0.334689
eastus	0.386379	0.376254	0.387381
eastus2	0.437065	0.381855	0.418019
centralus	0.437747	0.431546	0.472273
northcentralus	0.407172	0.376833	0.349627
southcentralus	0.704029	0.633358	0.772523
westcentralus	0.692972	0.706382	0.773655
canadacentral	0.453071	0.464417	0.490271
canadaeast	0.566612	0.686097	0.702407

Table A.9: The cost analysis of the acknowledgement function, with three tests

Dollar	Avax	Bytes
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.17973047537219500	0.00004786430768	163732
0.1612852921431600	0.000046346348317	163732
0.1612852921431600	0.000046346348317	163732
0.1612734714636000	0.000046342951570	163732
0.1612852921431600	0.000046346348317	163732
0.1612852921431600	0.000046346348317	163732
0.1612852921431600	0.000046346348317	163732
0.1612852921431600	0.000046346348317	163732
0.1612852921431600	0.000046346348317	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732
0.16247097780579400	0.000046235338021	163732

Table A.10: The cost analysis of the data publication function, with three tests

Dollar	Avax	Bytes
0.19384375024735875	0.00005162283628425	176589
0.22447209243969125	0.00005977951862575	204491
0.27415011746500125	0.00007300935218775	249747
0.34898329587284625	0.00009293829450675	317919
0.49869026800862500	0.00013280699547500	454300
0.79794614184524250	0.00021250230142350	726918
1.42119371716388000	0.00037848035077600	1294688
2.66773387396665875	0.00071044843514425	2430269
0.1739501652350700	0.00004998567966525	176589
0.2014352153253300	0.00005788368256475	204491
0.2460149381726100	0.00007069394775075	249747
0.3131682187529700	0.00008999086745775	317919
0.4474994063294400	0.00012859178342800	454288
0.7160790367254600	0.00020576983813950	726942
1.2753409981814400	0.00036647729832800	1294688
2.3939407704539100	0.00068791401449825	2430257
0.17522895646390050	0.00004986595232325	176589
0.20290415478190550	0.00005774164905575	204479
0.24782351216661150	0.00007052461928475	249747
0.31548237635143950	0.00008977870698675	317931
0.45081300459840400	0.00012829055338600	454312
0.72134327772273900	0.00020527697146350	726942
1.28472855010115000	0.00036560288847500	1294700
2.41146833361208250	0.00068624596858625	2430185

Table A.11: The cost analysis of the key publication function, with three tests

Dollar	Avax	Bytes
0.28358604426183625	0.00007552224880475	258343
0.34625109231336250	0.00009221067704750	315430
0.40879319669326875	0.00010886636396625	372405
0.50124574004015500	0.00013348754728100	456628
0.68342960028798125	0.00018200521978375	622595
1.04501132936781625	0.00027829862300075	951991
1.79549352998826250	0.00047816072702750	1635670
3.29373670478320875	0.00087716024095425	3000549
0.2544824849640900	0.00007312715085175	258343
0.3107164128009000	0.00008928632551750	315430
0.3668400142951500	0.00010541379721125	372405
0.4498044388436400	0.00012925414909300	456628
0.6132913325548500	0.00017623314153875	622595
0.9377650462503300	0.00026947271443975	951991
1.6112275779921000	0.00046299643045750	1635670
2.9557106860898700	0.00084934215117525	3000549
0.25635330796229350	0.00007295199429775	258343
0.31298873083788100	0.00008906907536650	315418
0.36953683146707250	0.00010516130662125	372405
0.45311116735582600	0.00012894455530900	456628
0.61778802418137350	0.00017580763351775	622583
0.94465900759970950	0.00026882726454175	951991
1.62307248593801500	0.00046188744619750	1635670
2.97743953524172050	0.00084730777895325	3000549