Blockchain-Based Framework and Simulation Middleware for Service Level Agreement Compliance Assessment in the Context of IoT

Ali Abdulaziz K Alzubaidi

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

November 2022

In Memory of My Sister, Saliha . . .

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Ali Abdulaziz K Alzubaidi
November 2022

Acknowledgements

First and foremost, I would like to thank Allah (God) for providing me with the strength, patience, knowledge and opportunity to complete my research. Without his blessings, this would not have been possible.

My sincere admiration and acknowledgement are to my principal supervisor, Dr Ellis Solaiman and my second supervisor Dr Karan Mitra. I truly appreciate everything they have provided me with during my PhD journey. The presented work in this dissertation would not have been realisable without their excellent supervision, guidance, encouragement, exceptional support, and insightful feedback. I also extend my gratitude to Professor Aad van Moorsel for his support and involvement. I also would like to take the opportunity to thank the members of my examining committee, Professor Maciej Koutny and Dr George Theodorakopoulos.

I extend my thanks and appreciation to all those who have been part of my PhD journey. In particular, I benefited greatly from Professor Raj Ranjan's extraordinary experience, deep insight and excellent feedback. I am also thankful to Dr Pankesh Patel and Dr Carlos Molina-Jimenez for useful discussions, reviews and feedback during the early stages of my PhD. Moreover, I would like to express my gratitude to my colleague Adel Albshri for devoting his valuable time and effort to review this dissertation and providing his useful feedback and comments. Finally, I thank my dear friend, Mohammad Alzahrani, for being there every time I needed to loudly discuss and critically review the implementation of the IoT-based system in the appendices of this dissertation.

I am profoundly appreciative to Umm Al-Qura University (UQU), Saudi Arabia, for providing me with the scholarship opportunity to pursue my PhD study and for the generous financial support. I am also extremely grateful to Newcastle University for all the support I received during my study, examples of which are the generous funding of several experiments and the provision of equipment that helped me complete this study.

Finally, I would like to express my deepest appreciation to my dear and lovely wife, Tahani, for all the support she has been giving me all the way. She has helped me a lot, making sure in all possible ways that I am mentally and physically fit to focus on having this dissertation done. I extend my love and thanks to my little ones, the joy of my life, Azooz and Meshari, for their support and best wishes. I also cannot express enough my appreciation and gratitude for all the support and encouragement I received from my parents and siblings. To all of the mentioned in this acknowledgement, I thank you for your belief in me and for helping me get through difficult times. I hope to scale to your good opinion and level up with your great expectations.

Abstract

The concept of Service Level Agreement (SLA) is commonly employed to regulate the contractual relationships between service providers and consumers. While several domains relate to SLA, this thesis focuses on SLA in the context of IoT. Regardless, SLA can be fragile and susceptible to violations and trust issues without a reliable trust mechanism in place. Recently, Blockchain has presented itself as an appealing alternative for mitigating trust issues related to centralised authorities and third parties. Following the success of Bitcoin, several blockchain platforms have emerged, such as Ethereum and Hyperledger Fabric, to enable conducting distrusted processes in a non-repudiable manner. This thesis adopts Hyperledger Fabric as an underlying blockchain infrastructure and examines how Blockchain can be incorporated to serve distrusted processes related to a typical SLA life-cycle (SLA definition, monitoring, compliance assessment, penalty enforcement, and termination). First, it explores the literature of both traditional SLA practice and blockchain-based SLA studies. Accordingly, it proposes and evaluates an SLA representation and awareness approach within the Blockchain and demonstrates its benefits for SLA definition and negotiation purposes. Following, it experiments with the use of Blockchain for SLA monitoring, compliance assessment, and penalty enforcement in the context of IoT. Hyperledger Fabric employs a mechanism for preventing the double-spending problem, usually associated with monetary applications. However, this thesis demonstrates that the MVCC protocol does not align well with the high rate of transactions expected from the monitoring tool. Therefore, it proposes a set of design considerations for smart contracts to resolve these issues. Accordingly, it evaluates the performance of the proposed solution and reports a considerable improvement compared to naive approaches. Finally, this thesis contributes a middleware to close the gap between IoT simulated environments and real-world Blockchain platforms. Thus, it facilitates the usage of IoT simulators for Blockchain-based SLA purposes in terms of workload generation, metrics monitoring and benchmarking.

Table of Contents

List of Figures xvii

List of Tables xxi

1 Introduction 1

1.1 Research Motivation . . . 3
1.2 Research Aim and Questions . . . 3
1.2.1 SLA Representation and Awareness . . . 4
1.2.2 Blockchain-based SLA Compliance in the Context of IoT . . . 4
1.2.3 Utilising IoT Simulators For Experimenting Blockchain-based SLA Solutions . . . 5
1.3 Contributions . . . 6
1.3.1 First Contribution: Blockchain-based SLA Representation and Awareness Approach . . . 6
1.3.2 Second Contribution: Decentralised SLA Compliance Assessment and Penalty Enforcement . . . 6
1.4 List of Publications . . . 7
1.5 Thesis Structure . . . 9

2 Background 13

2.1 Overview on Internet of Things (IoT) . . . 13
2.1.1 Communication Protocols . . . 15
2.1.2 IoT Simulators . . . 15
2.2 Overview on Traditional SLA Practice . . . 16
2.2.1 SLA Definition and Negotiation . . . 16
2.2.2 SLA Establishment and Service Provisioning . . . 18
2.2.3 Monitoring . . . 18
2.2.4 SLA Enforcement . . . 19
2.2.5 Billing and SLA Termination . . . 21
2.2.6 SLA-guaranteed Cloud-based IoT Services . . . 21
2.2.7 The Trust Dilemma . . . 22
2.3 Overview on Blockchain . . . 25
2.3.1 Blockchain Chronology . . . 25
2.3.2 Blockchain Characteristics . . . 25

2.4 Hyperledger Fabric 29
2.4.1 Basic HLF Network 29
2.4.2 Transaction Flow 32
2.4.3 Performance Metrics 34
2.5 Considerations for Blockchain Platform Selection 35
2.6 Related Blockchain-based SLA Works 41
2.6.1 The Convergence between Blockchain and Monitoring 43
2.7 Advantages and Disadvantages of Blockchain-based SLA Solutions 44
2.8 List of Assumptions 46

3 SLA Representation and Awareness within Blockchain 49
3.1 Introduction 49
3.2 SLA Representation in Related Works 50
3.2.1 SLA-agnostic approaches 50
3.2.2 SLA-aware Approaches 51
3.2.3 SLA Negotiation in Related Works 54
3.3 IRAFUTAL: Proposed Principles for SLA Representation 55
3.4 Proposed SLA Representation Approach 57
3.4.1 Overview on the State Storage capability 57
3.4.2 SLA as Blockchain Assets 58
3.4.3 Formal SLA Specification 59
3.4.4 Designing an SLA Data Model 60
3.5 Implementation of the SLA Data Model 63
3.5.1 The Logic of SLA Asset Manager 64
3.6 Evaluation and Observation 71
3.6.1 Failure Test Units 71
3.6.2 Use Case 1: SLA Definition 72
3.6.3 Use Case 2: SLA Negotiation 73
3.6.4 Proposed Approach vs. Conventional Approaches 76
3.6.5 Threats to Validity 76
3.7 Conclusion 77

4 Blockchain-based SLA Compliance Assessment and Penalty Enforcement 79
4.1 Introduction 79
4.2 Preliminary 80
4.2.1 A Remote Healthcare Scenario 80
4.2.2 SLA Example 81
4.3 Revisiting the Current Trust Model 82
4.3.1 SLA Awareness within blockchain 82
4.3.2 SLA Monitoring 83
4.4 Compliance Assessment over Blockchain 85

4.4.1 Adjusted SLA Data Model . . . 85
4.4.2 The Compliance Assessment Logic . . . 87
4.5 Dependability Experiment . . . 91
4.5.1 Failure Test Units . . . 91
4.5.2 Decision Accuracy Validation . . . 93
4.5.3 Emulation of IoT Scenario and Monitoring . . . 97
4.6 Blockchain Benchmark and Results . . . 100
4.6.1 Experimental Setup . . . 101
4.6.2 Results and Observations . . . 102
4.6.3 Observation and Remarks . . . 103
4.7 Discussion . . . 104
4.7.1 Real-time SLA Awareness . . . 104
4.7.2 SLA Monitoring . . . 104
4.7.3 Threats to Validity . . . 105
4.8 Conclusion and Future work . . . 106

5 IoT Monitoring and Enhanced SLA Compliance Assessment Approach 107
5.1 Introduction . . . 107
5.2 Preliminaries . . . 108
5.2.1 Hypothetical IoT Scenario . . . 109
5.2.2 Architecture Overview . . . 111
5.3 Monitoring Architecture and Considerations . . . 114
5.3.1 Determining Contributing Factors to Compliance Status . . . 114
5.3.2 Monitoring Architecture . . . 117
5.3.3 Metrics Instrumentation and Exporting . . . 117
5.3.4 Metrics Collection . . . 120
5.3.5 Validating the Monitoring Approach . . . 121
5.4 Enhanced Compliance Assessment Approach . . . 126
5.4.1 Smart Contract Invocation . . . 126
5.4.2 MVCC Impact on High-Throughput Transactions . . . 126
5.4.3 Enhanced Compliance Data Model . . . 127
5.4.4 Processing Received Monitoring Metrics . . . 129
5.4.5 Compliance Assessment and Enforcement . . . 130
5.5 Experiment and Evaluation . . . 132
5.5.1 Fixed Total Transactions and Variable Rates . . . 133
5.5.2 Variable Total Transactions and Fixed Average Rate . . . 134
5.5.3 Compliance Assessment Execution Time . . . 134
5.6 Conclusion and Future Work . . . 135

6 Blockchain-based Simulation Middleware for SLA Monitoring and Benchmarking 137
6.1 Introduction 137
6.2 Simulators for Experimenting Blockchain-based Solutions 139
6.2.1 Hypothesis 139
6.2.2 Motivation: The Gap between Simulators and Blockchain 139
6.3 Proposed Architecture 140
6.3.1 The Blockchain Side 141
6.3.2 Transactions Fail-safe Mechanism 143
6.3.3 Quality Requirement Definition 144
6.3.4 Monitoring and Reporting Mechanism 144
6.3.5 The Problem of Duplicate Smart Contract Invocations 149
6.4 Validating the Concurrent Storage 150
6.5 Evaluation 153
6.5.1 Blockchain-side Deployment 154
6.5.2 Simulated IoT Model 154
6.5.3 Middleware Settings 156
6.6 Evaluation Results 156
6.6.1 Correct Behaviour 156
6.6.2 Simulator Execution Time 159
6.7 Smart Contract Benchmarking 159
6.7.1 Composing and Exporting Instruments 160
6.7.2 Instruments Gathering and Visualisation 161
6.7.3 Experimenting the Benchmarking Feature 162
6.8 Limitations and Future Work 164
6.9 Conclusion 165

7 Conclusion and Future Work 167
7.1 Discussion and Lessons Learnt 167
7.1.1 SLA Awareness 167
7.1.2 SLA Representation within Blockchain 168
7.1.3 Monitoring 169
7.1.4 Compliance Assessment in the Context of IoT 169
7.1.5 Reliability and Performance 170
7.2 Thesis Generality 172
7.3 Threats to Validity and Future Trends 173

References 177

Appendix A Miscellaneous 189
A.1 Related Achievement 189
A.2 Sample SLA-guaranteed Cloud-based IoT Services 190

A.2.1 Amazon Web Service (AWS) SLA 190
A.2.2 Google Cloud Platform (GCP) SLA 194
A.2.3 Microsoft Azure SLA 198

Appendix B Description of the IoT-based Fire Mitigation System 207

B.1 Description of the Example IoT System 207
B.1.1 Fire detection and Alert Processing 207
B.1.2 Fire Detection 208
B.1.3 Edge Computing Unit 209
B.1.4 Data Model at IoT Server 212
B.1.5 REST HTTP API 212
B.1.6 Websocket Protocol 213

Appendix C Blockchain-based Middleware for Simulated Environment 215

C.1 Asset and Wallet Management 215
C.2 Screenshots 215

List of Figures

2.1 Compact SLA Life Cycle Phases 16
2.2 SLA example between a service provider and consumers 17
2.3 Basic SLA structure and types of SLA enforcement mapped to SLA elements. . 20
2.4 An simplified example of an IoT as a cloud service. 21
2.5 Overview on current trust practice for SLA enforcement 24
2.6 Visualisation of a basic Blockchain structure (A chain of blocks). 27
2.7 Basic Hyperledger Fabric Network 30
2.8 An illustration of the state storage and data organisation 31
2.9 A basic transaction flow in Hyperledger Fabric 32
2.10 Conceptualised illustration of Ethereum approach for Smart contract upgrade cycle process. 38
2.11 Conceptualised illustration of HLF approach for Smart contract upgrade cycle process. 38

3.1 Examples of SLA-agnostic approaches 51
3.2 Types of SLA-aware approaches 53
3.3 SLA negotiation lifecycle in conventional and proposed SLA representation approaches 55
3.4 Conventional SLA representation approaches vs. the proposed approach 58
3.5 Overview on the process of SLA modelling as blockchain assets 59
3.6 Example of IRAFUTAL-compliant SLA model based on formally-specified SLA document 60
3.7 Relationship between agreement component and other SLA components. . . . 61
3.8 SLA model at the state storage and possible applications as smart contracts . . 62
3.9 A smart contract that controls and interfaces with the deployed SLA data model 63
3.10 Naive SLA manager Approach for managing SLA Assets 65
3.11 Generic JSON-formatted schema for defining various SLA assets 66
3.12 Examples of compatible JSON-formatted payloads for defining various SLA assets 67
3.13 Validating minimum instances of each component in the JSON payload against the SLA data model 68
3.14 Example graph of reusable instances SLA units persisted at state storage. . . . 73
3.15 Overview on SLA negotiation session over blockchain 74
3.16 Basic SLA Negotiation Protocol: A use case 75

4.1 Example IoT healthcare scenario employing MQTT for data exchange 81
4.2 Overview on the proposed Blockchain-based Compliance enforcement 83
4.3 The monitoring role in the Blockchain-based SLA compliance Assessment . . . 84
4.4 Adjusted SLA data model to accommodate performance reports 86
4.5 A graph of SLA assets based on the adjusted SLA data model 87
4.6 Overview on incident processing procedures at the smart contract level 88
4.7 Assessing the compliance of the service provider with regard to each quality requirement 90
4.8 IoT application and monitoring for experimenting the compliance assessment smart contract. 97

5.1 Research Methodology 109
5.2 Overview of an IoT-based Fire Mitigation System. Edges 1,2, and 3 represent the edge computing nodes. 109
5.3 Stages of a fire event from origination until being reported 110
5.4 Motivating IoT scenario where blockchain is employed for SLA monitoring and enforcement 112
5.5 Metrics collection and reporting to the blockchain-side. 113
5.6 Hyperledger Fabric's blockchain network of two organisations: IoTSP and firefighting station. 113
5.7 key stages for a fire event across different IoT layers. 115
5.8 Timeline for fire event development. 116
5.9 Employing Prometheus monitoring tool for feeding metrics to the Blockchain-side. 118
5.10 Instrumenting and exposing relevant metrics at edge level. 119
5.11 Instrumenting and exposing relevant metrics at server level. 120
5.12 A screenshot of metric collection and Incident Identification 124
5.13 Transaction Execution Journey within HLF. 125
5.14 Read-Write set conflicts caused by multiple transactions updating the same record. 127
5.15 Enhanced Data Model for Evaluation Compliance over Blockchain. 128
5.16 Algorithm 7 performance: processing received metrics at variable rates and fixed total of 1000 transactions. 133
5.17 Algorithm 7 performance: processing received metrics at fixed rate of 500 transactions and variable total transactions. 134
5.18 Latency for executing Algorithm 8 on variable collection of evaluation records stored on HLF state storage. 135

6.1 Basic workflow from simulator to smart contract 140
6.2 Blockchain-based Middleware Architecture 141
6.3 Authenticating the application client to the blockchain network and wallet generation 142
6.4 A fail-safe mechanism for transactions submission 143

6.5 Monitoring and reporting procedures . . . 145
6.6 A snap of locking mechanism acquired by agents or workers on the concurrent storage . . . 148
6.7 Demonstrating the validation process of correct operations on the concurrent storage: using 100 generated metrics . . . 153
6.8 Transactions count for all 3 test cases . . . 157
6.9 Count of compliant and breach cases per transactions on the blockchain state storage . . . 158
6.10 Actual Compliance Rate vs expected ones in all 3 cases . . . 159
6.11 Simulator's execution time before and after integration in all 3 cases . . . 160
6.12 Sequence of composing and exporting Instruments . . . 160
6.13 Design of the experiment on utilising the middleware for performance benchmarking . . . 163
6.14 Validation of the benchmarking results . . . 164

A.1 Nomination for best paper candidate IEEE SmartIoT Conference 2020 . . . 189
A.2 Nomination for best paper candidate IEEE SmartIoT Conference 2021 . . . 190

B.1 Fire detection and alerting at edge level. . . . 208
B.2 Alert handling at IoT sever level . . . 208
B.3 A set of hardware components considered for the edge layer . . . 209
B.4 Data Modelling at the IoT server side . . . 212

C.1 Quality requirement at state storage . . . 215
C.2 Reported Metrics at the state storage (CouchDB) . . . 216
C.3 Reported breach and compliant metrics for each performance report pr_i at the state storage . . . 216

List of Tables

1.1 Comparison between Chapter 4 and Chapter 5 7

2.1 Sample SLA-guaranteed Cloud-based IoT Service 22
2.2 Tradeoff Comparison: Ethereum versus Hyperledger Fabric. 35
2.3 Existing studies on Hyperledger Fabric Performance Analysis 36
2.4 key differences between the proposed blockchain-based SLA solution and related works 45

3.1 Failure Tests conducted on the SLA manager smart contract 72
3.2 Comparison table between conventional and proposed SLA representation approaches 77

4.1 Failure Tests conducted on the compliance assessment smart contract 92
4.2 Diagnostic Accuracy 2x2 Table 95
4.3 Optimum results of the Diagnostic Accuracy method 96
4.4 Hyperledger Fabric (HLF) network deployment configuration 101
4.5 Experimental settings (Hyperledger Caliper) 102
4.6 Results of Latency and Transactions success rate 102

5.1 Classification and summary of metrics covered by Algorithm 6 122
5.2 Blockchain deployment and configurations. 132

6.1 Test-bed for validating correct operations on the concurrent storage.
* delay is applied only for the first 100 generated metrics, when $x = 2$, for demonstration and visualisation purposes. 151
6.2 IoT Components Specifications: calibrating the simulated IoT model around 3 seconds for transmission time 155
6.3 Testbed for causing different compliance rate for each test case 155
6.4 Benchmarking Configurations and Expectations 163

B.1 Mapping flame states to digital outputs 209
B.2 Related analysis factors between Raspberry PI4 and ESP-WROON-32 210

List of Algorithms

1 Creation of SLA asset in accordance with the SLA data model 68
2 Agreement Composition 69
3 Validating correct update of the performance report pr_i 94
4 Conducting Diagnostic Accuracy Method 96
5 Scenario Experiment: MQTT Broker Monitoring and Compliance Assessment . 99
6 Reporting Mechanism to the Blockchain Side 121
7 Evaluation of Received Monitoring Metrics 130
8 Concluding Assessment and Enforcement Logic 131
9 Quality Requirement Establishment Protocol 144
10 Agents' Evaluation Behaviour 146
11 Workers' Reporting Behaviour 146
12 Duplication Prevention Mechanism 150
13 Controlled Experiment on Core Functionalities 152
14 Instruments Composition 161
15 Edge Layer: simple event-trigger logic 211

Chapter 1. Introduction

Due to the proliferation and advancement of service-oriented computing, service consumers can opt to outsource on-premise infrastructure to a specialised IT service provider, such as cloud providers [1][2]. Such a practice is usually justified by the need to focus on business logic rather than coping with the burden of IT management, operation and maintenance [3][4]. Resorting to specialised service providers becomes immensely appealing when consumers lack access to appropriate infrastructure, expertise, and necessary workforce [5]. Subsequently, the concept of Service Level Agreement (SLA) has gradually established its importance as a contractual method [6]. In the context of this thesis, a service consumer is an organisation that outsources IoT-related functionalities to an IoT service provider. For instance, chapter 4 assumes a telemedicine scenario where a healthcare provider adopts IoT to enable the provisioning of remote healthcare. Another example is the IoT scenario presented in chapter 5, where a firefighting station embraces IoT for prompt response to fire incidents. In both scenarios, the healthcare provider and the firefighting station outsource some or all IoT-related operations to specialised IoT service providers for many reasons, as discussed above.

SLA plays a vital role in governing the contractual relationship between service providers and consumers [1]. In particular, SLA regulates service delivery and delineates expectations, rights and obligations of each involved party [7]. According to the ISO/IEC 19086-2:2018 standard [8], the minimal form of an SLA should clearly define a set of properties as follows. First, the SLA must define SLA participants (at least service providers and consumers). Second, it includes Service Level Objectives (SLOs) that stipulate a set of obligations and responsibilities carried out by the service provider. Optimally, an SLO should represent a measurable service quality requirement such as availability, throughput, latency, jitter, and packet loss rate [3]. For instance, availability must not be less than 99.9% all the time. Finally, the SLA can state a set of violation consequences enforced on the service provider when it fails to meet the agreement. The violation consequence can be a penalty imposed on the service provider in the form of financial service credit [9].

Bakalos et al.[9] provides a set of SLA examples by prominent cloud service providers. Recently, cloud services started to accommodate the emergent requirements of the Internet of Things (IoT) paradigm [10]. Moreover, end-to-end IoT service providers have promised to alleviate the complexity burden of a typical IoT architecture, which is not only limited to cloud services but also includes physical things, edge data centres, and networks [11]. Subsequently, the concept of SLA has also extended the coverage to accommodate emergent proprieties unique to IoT. Both Girs et al. [12] and Mubeen et al.[10] extensively survey SLA in the context of

IoT. Notably, Alqahtani et al.[11] addresses the need for standardising SLA specifications in the context of IoT.

Nowadays, several cloud providers employ the SLA concept to establish their trustworthiness and assure their potential consumers about the quality of their offered services [13]. Consumers may view the SLA concept as an assessment risk instrument that may help select a proper service provider that satisfies their requirements. While SLA guarantees quality requirements (e.g. availability, latency, etc.), it is, as other contractual methods, susceptible to breaches[14]. Therefore, service providers usually guarantee their commitment and show goodwill by accepting a set of violation consequences (i.e. penalties). In the current practice, several service providers promise to process incidents in good faith, assuring their consumers to impose SLA violation consequences on themselves [9].

While intriguing, one can question which party to trust as the authority of SLA enforcement and compliance [15]? This question becomes even more delicate when dealing with critical systems that are less tolerable to failures. The current SLA practice commonly assumes cloud providers for holding responsibility for typical SLA lifecycle management, such as SLA monitoring, compliance assessment, incident management, and penalty enforcement [16][17]. It is also typically the consumers' responsibility to report a service level degradation, supported by evidence deemed irrefutable by the service provider or trusted third parties [9]. This is usually a tedious process, manually handled, time-consuming, error-prone, and requires consumers' good faith [18][19]. In some cases, service providers may not react well to poorly formed claims, regardless of their validity [20]. Both sides of a contractual relationship, service providers or consumers, may find it inviting to intentionally fabricate or manipulate evidence of violation incidents in order to maximise profit or avoid hefty penalty [13]. In some scenarios, unresolved disputes have to be escalated to jurisdiction means [21][16].

By considering the possibility of deliberate corruption, misconduct, opacity, conflict of interests, and single point of failure [22], this thesis argues that no single party should solely control SLA lifecycle management. Recently, there has been a growing interest in establishing trust mechanisms and schemes that attempt to resolve the SLA's trust dilemma; examples of which are explored in [23][13][15], such as reputation-based mechanism, usage of auditors, feedback and review systems, trust brokers, and mediators. However, this thesis questions any trust mechanism requiring consumers to depend on service providers or third parties [24].

The blockchain technology invites revisiting traditional applications wherever trust is taken for granted [25]; the SLA practice is no exception. Accordingly, this thesis leverages blockchain features to enable non-repudiable SLA compliance assessment and enforcement of violation consequences in the context of IoT. It argues that both the decentralisation of blockchain and the autonomy of smart contracts can improve the neutrality of SLA governance and mitigate associated trust issues.

In the following sections, this chapter highlights the research aim and questions that drive this thesis. Furthermore, it describes the thesis structure and highlights main contributions and a list of related publications.

1.1. Research Motivation

Bitcoin architecture primarily influences the current blockchain technology. It proposed a peer to peer (P2P) cash system in 2008, which has materialised existence months later at the beginning of 2009 [26]. Since then, it has demonstrated a decentralised ecosystem that enables distrusted monetary processes while mitigating the need for centralised authority or third parties. However, the usage of the Bitcoin blockchain network is only limited to serving its hardcoded cryptocurrency.

Therefore, several generic blockchain platforms have emerged around the year 2015, such as Ethereum [27][28], and Hyperledger Fabric [29], to enable conducting distrusted processes in a non-repudiable manner beyond cryptocurrency applications. However, they differ in the blockchain implementation philosophy of some aspects such as blockchain network, permission environment, transaction flow. Most contemporary blockchain platforms aim to enable decentralised applications to serve various problem domains. The accessibility to the blockchain technology is possible thanks to the concept of smart contracts, which enables exploiting typical blockchain features such as decentralisation, transactions immutability, transparency, traceability and resistance to the single point of failure [30]. Hence, the concept of smart contracts plays a vital role in enabling distrusted processes to operate on a blockchain-based decentralised network; beyond the influence of any party, [31].

The paradigm of blockchain-based applications is relatively new and still progressing toward maturity. Consequently, when this research effort commenced in 2017, there was little discussion on the role of blockchain for SLA purposes in general and in the context of IoT in particular. However, the literature has recently started to realise the potentiality of blockchain-based SLA solutions in several domains, including IoT. Nevertheless, most existing blockchain-based SLA studies, in section 2.6, are influenced by Ethereum's philosophy of implementing blockchain principles. The influence extends to the architecture of their proposed SLA solutions in terms of permissionless nature, underlying network, infrastructure ownership, transaction flow, execution cost, smart contract lifecycle, and limitation of the smart contract programming language (namely, Solidity).

1.2. Research Aim and Questions

This thesis argues that decentralising SLA compliance assessment can improve distrusted SLA processes in complex and distributed environments such as end-to-end IoT ecosystems. Therefore, it leverages Hyperledger Fabric [29], which is a permissioned and enterprise-grade blockchain platform, to propose, implement, and validate a decentralised SLA compliance assessment approach that operates beyond the influence of any single authority. Moreover, it revisits other stages of a typical SLA lifecycle that are related to compliance assessment and enforcement, which are SLA definition, negotiation and monitoring. Accordingly, it seeks to address the following:

1.2.1. SLA Representation and Awareness

Blockchain-based smart contracts can manifest a decentralised SLA management approach over the blockchain. For example, we can leverage smart contracts for automating distrusted SLA tasks such as compliance assessment, penalty enforcement, billing and so forth. In general, such functions needs to be aware of the SLA definition. Blockchain-wise, smart contracts must maintain necessary SLA awareness within blockchain to elegantly deliver their tasks. For instance, a compliance assessment smart contract needs to be aware of what quality requirements are agreed upon by SLA parties. A smart contract conducting a penalty enforcement also needs to know which associated penalties to apply in case of violation. Moreover, monitoring tools must align their configurations, thresholds, triggers with the SLA in order to identify abnormalities and report incidents to the blockchain side.

Most related studies, in section 3.2, tend to represent SLA properties directly in the smart contract, meaning that SLA content is hardcoded in code logic. Subsequently, SLA can inherit blockchain features such as smart contracts' immutability. This practice can also provide blockchain-based applications with a level of SLA awareness to the smart contract that encodes the SLA. However, coupling SLA tightly with the smart contract does not align with a typical SLA life cycle, where negotiation and error rectification is normally expected. That is, SLA content, being encoded in the smart contract, is then permanently immutable and cannot be updated when needed in a straightforward manner. Moreover, other automated tasks struggle to comprehend or consume an SLA represented in the code of another smart contract. For instance, assume a monitoring tool or a penalty enforcement smart contract that attempt to read an SLA content (i.g. quality requirements) defined in the logic code of another smart contract. Above all, composing an SLA in the form of a smart contract is not user-friendly in the first place, and requires blockchain experts for translation, development and deployment.

Accordingly, one of the quests of this research study is to realise a blockchain-based SLA representation and awareness that serve SLA compliance assessment and enforcement without sacrificing key properties of other SLA lifecycle stages such as SLA definition, negotiation and monitoring. Some questions that this thesis attempt to answer in this regard are as follows:

1. *How SLA is being represented in current blockchain-based SLA solutions?* and *what issues are associated with them?*

2. *How to appropriately represent SLA within the blockchain to preserve SLA immutability while maintaining resiliency to SLA amendment needed for SLA negotiation and error-rectification?*

Answering these questions is key to achieving a proper blockchain-based SLA compliance approach.

1.2.2. Blockchain-based SLA Compliance in the Context of IoT

SLA, like any contractual method, is susceptible to breaches. The current practice assumes trust in service providers to handle violation incidents, process them, and apply consequences

on themselves. Alternatively, other trust schemes take place, such as third party auditors or assessors, reputation and review-based mechanisms. However, this thesis questions the trust mechanism dependent on a single authority or a third party. It argues that they can be distorted due to the possibility of a single point of failure, whether it is because of misconduct, malicious act, forgery, lack of transparency and traceability.

Therefore, this thesis deems SLA compliance assessment and penalty enforcement as distrusted processes which should not be handled by the service provider or a third party. Blockchain holds the promise of shifting distrusted tasks to a non-repudiable environment, where they can operate autonomously beyond the influence of any entity. Subsequently, this thesis foresees that blockchain is a promising enabler for mitigating trust issues associated with the current practice of SLA monitoring and compliance assessment.

This thesis extends the proposed SLA representation and awareness approach to propose, implement and validate a blockchain-based SLA compliance assessment in the context of IoT. In particular, it mainly seeks to answer the following questions:

1. Given that SLA is properly represented within the blockchain, *How to leverage Hyperledger Fabric-based smart contract for serving a decentralised SLA compliance assessment?*
2. *To what extent the proposed solution can serve distrusted processes such as evaluation of monitoring logs and penalty enforcement in the context of IoT?*
3. Given that Hyperledger Fabric is the underlying blockchain platform, *how would the proposed compliance assessment approach perform against a massive number of consecutive monitoring logs?*

1.2.3. Utilising IoT Simulators For Experimenting Blockchain-based SLA Solutions

Realising a blockchain-based SLA solution in the context of IoT requires access to both ends, blockchain platforms and IoT infrastructure. Concerning blockchain, most existing platforms are open-source software that can be deployed using reasonable hardware requirements. Moreover, rented cloud instances can compensate for the limited resources of local machines. Thus Blockchain platforms are easily accessible for research and experimental purposes. However, on the other hand, gaining access to a large-scale IoT infrastructure can pose a real challenge for research and development.

Nevertheless, several use cases can leverage IoT simulators in order to compensate for this shortcoming [32]. For example, IoTSim-Osmosis [33] can be used to model a large-scale IoT architecture. Therefore, subjecting such a simulated IoT model enables generating a workload that can be leveraged for experimenting and benchmarking blockchain-based SLA solutions. However, while interesting, it can be a hurdle for researchers to bridge the gap between a real-world blockchain and a simulated IoT environment, which one will need to address before commencing a research effort on any blockchain-based IoT solutions in general SLA-specific projects in particular.

Therefore, this thesis seeks to answer the following questions in this regard:

1. *How to address the gap between simulated IoT models and real-world blockchain networks (e.g. Hyperledger Fabric), given their distinctive nature and execution environment?*

2. *How to realise a generic blockchain-based middleware architecture that facilitates employing any IoT simulator to experiment with any blockchain-based SLA solution?*

3. *How possible is the use of IoT simulators for experimenting and benchmarking blockchain-based SLA solutions?*

1.3. Contributions

In regards to the research questions identified above, the core contributions of this thesis are as follows:

1.3.1. First Contribution: Blockchain-based SLA Representation and Awareness Approach

This thesis examines and classifies related works based on the variation of their SLA representation approaches. Therefore, it addresses their limitations by proposing and implementing an alternative blockchain-based SLA representation and awareness approach. It also demonstrates the advantages of the proposed approach by evaluating two use cases, namely, SLA definition and negotiation. This contribution is covered in Chapter 3.

1.3.2. Second Contribution: Decentralised SLA Compliance Assessment and Penalty Enforcement

This thesis extends the first contribution to propose and experiment with a blockchain-based SLA compliance assessment and penalty enforcement in the context of IoT. As Table 1.1 illustrates, this thesis conducts two empirical studies in both Chapter 4 and Chapter 5. While they both investigate and experiment with a decentralised SLA compliance assessment in the context of IoT, they differ in terms of the hypothetical IoT scenario, SLA coverage, communication protocols and type of implementation of both the IoT scenario and the monitoring service.

These chapters altogether contribute the following:

- An insight into SLA analysis and monitoring in the context of IoT.

- An experimental study using Hyperledger Fabric on the decentralisation of compliance assessment and penalty enforcement.

- A performance benchmarking study on the performance of Hyperledger Fabric as an underlying blockchain platform, which revealed the issue of read-write set conflicts caused by the MVCC protocol (Multi-Version Concurrency Control) due to a high rate of transactions submitted from monitoring service.

Table 1.1 Comparison between Chapter 4 and Chapter 5

Facet	Chapter 4	Chapter 5
SLA Coverage	Limited to Cloud-based IoT service	Physical Edge Cloud Layers
Use case	Healthcare	Firefighting system
IoT Communication Protocol	MQTT	HTTP
IoT Implementation Type	Emulation	Real
Monitoring Implementation Type	Emulation	Real(Prometheus)
Blockchain used	Hyperledger Fabric v 1.4	Hyperledger Fabric 2.3.3
Blockchain Deployment	Local Machine	Realistic (distributed over cloud)

- A smart contract's design improvement that succeeds to eliminate transaction failures while maintaining a reliable performance under a high rate of transactions submitted from the monitoring service.
- A set of recommendations and lessons learnt regarding the incorporation of monitoring tools for blockchain-based SLA solutions.

Third Contribution: A Blockchain-based IoT Simulation Middleware

This thesis also contributes a novel middleware architecture that enables integrating IoT simulators of choice with Blockchain-based SLA solutions. In order to enable the integration between IoT simulated models and real blockchain networks, the proposed middleware architecture addresses the gap between their distinctive execution environments. The middleware equips simulators with a blockchain-based monitoring mechanism, which monitors simulated metrics and enables connectivity and communication with blockchain-based SLA solutions. Therefore, it is possible to experiment with a blockchain-based SLA solution. For example, experimenting the compliance assessment using a simulated IoT model that sufficiently represents a large IoT infrastructure. Furthermore, the proposed middleware enables utilising simulators to benchmark essential blockchain performance metrics related to deployed smart contracts such as transactions throughput, latency, transactions' success/fail rates. This contribution is covered in Chapter 6.

1.4. List of Publications

This section highlights a set of research contributions by this thesis that are either published or in review, which form the basis of this thesis.

First Publication

- **Title:**: Blockchain-Based SLA Management in the Context of IoT.
- **Status:** Published in IEEE IT Professional (Peer-reviewed) [19].

- **Cite as:** A. Alzubaidi, E. Solaiman, P. Patel and K. Mitra, "Blockchain-Based SLA Management in the Context of IoT," in IT Professional, vol. 21, no. 4, pp. 33-40, 1 July-Aug. 2019, doi: 10.1109/MITP.2019.2909216.

This article elaborates on our position in the literature of SLA in the context of IoT and highlights the trust gap found in current SLA practice. It proposes a blockchain-based SLA framework in the context of IoT and justifies why it considers Hyperledger Fabric over Ethereum as an underlying Blockchain platform for decentralising SLA management.

Second Publication

- **Title:** A Blockchain-based Approach for Assessing Compliance with SLA-guaranteed IoT Services.
- **Status:** Published in IEEE International Conference on Smart Internet of Things (Peer-reviewed) [34].
- **Notes:** Nominated for Best Paper Candidate. Refer to Figure A.1.
- **Cite as:** A. Alzubaidi, K. Mitra, P. Patel and E. Solaiman, "A Blockchain-based Approach for Assessing Compliance with SLA-guaranteed IoT Services," 2020 IEEE International Conference on Smart Internet of Things (SmartIoT), 2020, pp. 213-220, doi: 10.1109/SmartIoT49966.2020.00039.

This paper proposes a blockchain-based approach for decentralising SLA compliance assessment in the context of IoT. It assumes an SLA covering an MQTT broker, which serves an IoT-based healthcare scenario. It experiments with the approach by emulating a cloud-based IoT component offered by the Google Cloud Platform (GCP). Moreover, it accounts for the immutability of smart contracts, which hinders smooth maintenance after deployment. It also ensures the dependability of the smart contract by conducting a series of validation tests. The paper reveals that while smart contracts may pass rigorous validation in a testing environment, it may not be the case in a production environment. This is evident when the experiment deployed the smart contract to a real blockchain network and exposed to a massive number of transactions from the monitoring side, assuming a set of consecutive incidents (SLA breaches). This situation leads to some transaction failures due to conflicting Read-Write sets that are caused by the Multi-version Concurrency Control (MVCC); a protocol implemented by Hyperledger Fabric to prevent the double-spending problem. Finally, it provided the results of performance benchmarking in various settings, concluding that unless the issue of MVCC conflicts is addressed, we cannot assume the dependability of the smart contract.

Third Publication

- **Title:** Smart Contract Design Considerations for SLA Compliance Assessment in the Context of IoT

- **Status:** Published in IEEE International Conference on Smart Internet of Things (Peer-reviewed) [35].
- **Notes:** Nominated for Best Paper Candidate. Refer to Figure A.2.
- **Cite as:** A. Alzubaidi, K. Mitra and E. Solaiman, "Smart Contract Design Considerations for SLA Compliance Assessment in the Context of IoT," 2021 IEEE International Conference on Smart Internet of Things (SmartIoT), 2021, pp. 74-81, doi: 10.1109/SmartIoT52359.2021.00021.

This paper argues that while the MVCC protocol can solve the double-spending problem for monetary applications, it poses a challenge for high-throughput applications. This is evident in the second paper, section 1.4, when transactions fail due to Read-Write sets conflicts caused by a monitoring tool that transmits a massive number of transactions to the blockchain-based SLA compliance assessment. Therefore, this paper proposed a smart contract design approach that proved to mitigate MVCC conflicts while demonstrating a clear performance improvement in terms of transactions success rate, throughput and latency. Subsequently, it improved the compliance assessment approach to accommodate a hypothetical SLA covering an IoT-based Firefighting scenario, where a fire station outsources an end-to-end IoT infrastructure to a service provider.

Invited Article (In Review)

- **Title:** A Blockchain-based SLA Monitoring and Compliance Assessment for IoT Ecosystems
- **Status:** Submitted to Springer Journal of Cloud Computing Advances, Systems and Applications (Peer-reviewed).
- **Notes:** Invited by the IEEE smartIoT 2021 to extend and submit to a special issue titled *Empowering the Future Generation Cloud Computing with Internet of Things*.
- **Cite as:** In Review.

This article extends on the third paper, in section 1.4, by experimenting on a real implementation of both the monitoring tool and IoT infrastructure. It examines the IoT-based Fire station implementation and determines key co-factors that influences the compliance status of the IoT services provider. It then designs a monitoring mechanism for related metrics instrumentation, export, collection, breach identification and incident alert. It highlights a set of considerations regarding the role of monitoring tools with the blockchain-based SLA compliance assessment.

1.5. Thesis Structure

This thesis is mainly composed of the contributions listed in section 1.4. The following highlights the thesis organisation, and maps each contribution to its cosponsoring chapter.

Chapter 2: Background

Chapter 2 introduces a set of concepts that are essential for this research study. First, it provides an overview of a typical IoT architecture and sheds light on the need for IoT simulation. Moreover, it overviews the SLA concept and relevant stages of a typical SLA life cycle. Furthermore, it discusses SLA's relevance to IoT in the current SLA practice and highlights related trust issues. It also introduces blockchain technology and overviews Hyperledger Fabric components and performance. Subsequently, it considers a set of comparison elements between Hyperledger Fabric and Ethereum. Finally, it overviews related works in the literature of blockchain-based SLA studies.

Chapter 3: Blockchain-based SLA Representation and Awareness Approach

Chapter 3 covers the thesis's contribution described in section 1.3.1. It also implements an SLA data manager that satisfies a set of principles suggested by the proposed SLA representation approach. Finally, it evaluates the implementation with two use cases: SLA definition and SLA negotiation.

Chapter 4: Blockchain-based SLA Compliance Assessment and Penalty Enforcement: A Pilot Study

This chapter mainly covers and extends on the paper described in section 1.4. It conducts a pilot study on SLA compliance assessment in the context of cloud-based IoT services. The pilot study assumes an IoT-based healthcare scenario that employs MQTT protocol for communication and connectivity. It limits the scope to the cloud layer to enable exploration and observation. Subsequently, the pilot study assumes an SLA, and emulates both the cloud-based IoT component and a monitoring mechanism for experimental purposes. The pilot study also conducts dependability validation experiments in a testing environment. Then, it conducts an empirical experiment on a real blockchain network to benchmark the performance and report the outcomes in terms of transactions success rate, throughput and latency. Finally, it discusses the outcomes of the pilot study and sheds light on a set of lessons learnt and recommendations brought forward to the following chapters.

Chapter 5: IoT Monitoring and Enhanced SLA Compliance Assessment Approach

This chapter mainly covers both papers, the published paper described in section 1.4 and the in-review paper described in section 1.4. It considers the lesson learnt from chapter 4 and rethink the design of the compliance assessment with the aim to mitigate the issue of MVCC conflicts. As we gain the confidence on the proposed SLA compliance assessment, this chapter assumes an SLA that covers a real implementation of an end-to-end IoT-based firefighting scenario with the aid of enterprise-grade monitoring system; namely, Prometheus. Appendix B describes the implementation of the IoT-based firefighting system. Table 1.1 illustrates the difference between this chapter and chapter 4.

Chapter 6: Blockchain-based Simulation Middleware for SLA Monitoring and Benchmarking

This chapter mainly covers the in-preparation paper described in section 1.3.2. It mainly justifies the need for IoT simulators and proposes a middleware architecture for integrating simulators with real blockchain-based SLA solutions. It addresses the gap between them and validates the feasibility of the proposed middleware in two states. In the first stage, the experiment emulates both the blockchain side and simulator side to validate the correct functionality of the middleware without the influence of either side. Then, the experiment validates the middleware for actual integration between both sides, the blockchain and the simulator sides. Finally, it presents the experiment results, which demonstrates the viability of the middleware approach.

Chapter 7: Conclusion

This chapter concludes the thesis, and sheds light on a set of lessons learnt and observations. Finally, it also suggests some topics for further work.

Chapter 2. Background

Summary

This chapter provides a background of core concepts relevant to this research study. This chapter organises its sections as follows: section 2.1 provides an overview of a typical IoT architecture and sheds light on the need for IoT simulation. Section 2.2 overviews the SLA concept and relevant stages of a typical SLA life cycle. Furthermore, it discusses SLA's relevance to IoT in the current SLA practice and highlights related trust issues. Section 2.3 introduces blockchain technology in general and sheds light on its main characteristics. Section 2.4 draws attention the the blockchain platform employed by this thesis (Hyperledger Fabric) and overviews its key components and performance. Section 2.5 provides a comparison between Hyperledger Fabric and Ethereum. Finally, section 2.6 overviews related works in the literature of blockchain-based SLA studies.

2.1. Overview on Internet of Things (IoT)

Due to the recent advancement of connectivity and communication protocols, the web is no longer limited to human contribution or consumption but also has extended to accommodate virtually any connected object [36]. This has enabled various use cases in different domains serving several industry sectors. For example, this thesis implements IoT in two domains which are a remote healthcare application (refer to section 4.2.1) and a connected firefighting system (refer to section 5.2.1).

There have been a plethora of proposed IoT architectures such as those covered in [36] and [37]. Example of reference architectures include, but not limited to, IEEE Std 2413-2019 [38] ISO/IEC 30141:2018 [39], and AIOTI HLA [40]. Among various proposed IoT architectures, there has been understandably no mutual consensus on what constitutes a typical IoT architecture, as distinctive industry requirements and various viewpoints highly govern the final IoT architecture outlook [41]. However, this thesis conveniently adopts the basic IoT architecture presented by Alqahtani et al.[11], which is abstracted as follows:

Physical Layer

At this layer, we consider a set of physically deployed devices that can either observe its environment (sensors) or conduct physical actions (actuators). For example, a flame sensor constantly observes fire events can actuate a fire protection system. Most physical devices are

typically resources-constrained in terms of power/battery, memory, processing and storage [42]. Subsequently they usually lack essential some or all the following capabilities:

- Connectivity (e.g. WiFi, Bluetooth, Zigbee, NFC, etc.).
- Communication (e.g. HTTP, MQTT, CoAP, etc.).
- Recognised data format (e.g. XML, JSON, etc.).

Edge Layer

At this layer, we consider a capable computing unit deployable near the field of data sensing and actuation. Edge computing units manage and compensate for the limitations of physical devices such as sensors and actuators. This thesis considers edge computing units to play, at least, the following roles [42]:

- Empowers resources-constrained entities by providing extra capability that they lack.
- Remotely executes, controls and monitors the business logic and relevant processes to deployed devices and sensors.
- Forms an entry point that bridges the gap between local field deployment and the external world (e.g. cloud).

Cloud Layer

At this layer, this thesis considers a central point where geographically dispersed IoT assets (including edge computing units and their associated IoT field assets) can be authenticated, accessed, managed and supported [43]. The cloud layer also plays a crucial role in data governance, persistence, analysis, and decision-making. In this sense, The cloud layer provides services that are either classical such as infrastructure, platforms, software, etc.) or IoT-specific services such as remote accessibility, IoT assets management, visualisation, big data storage and analytics. It also facilitates typical IT overhead such as scalability, security, and maintenance.

Covered IoT Layers

This thesis proposes a blockchain-based SLA solution and experiments with it using two IoT scenarios, namely, remote healthcare (telemedicine) and firefighting systems. These two IoT scenarios are distinctive in terms of SLA-covered IoT layers. Table 1.1 highlights differences between these IoT scenarios in this regard. That is, the SLA coverage in the healthcare scenario is limited to the offered cloud-based IoT services and neglects other layers for the pilot study. On the other hand, the firefighting scenario dives into a more complex use case where the SLA extends its coverage to include all the above-mentioned IoT layers.

2.1.1. Communication Protocols

There are several communication protocols, such as HTTP and MQTT, that enable standardised data exchange and interaction between various IoT components over the internet [36]. By examining the SLA documents of prominent cloud providers (refer to Table 2.1, we find that they support these communication protocols to bridge the gap between cloud services and geographically dispersed IoT components. Figure 2.4 illustrates an example of using MQTT as a communication protocol, for that matter.

MQTT, short for *Message Queuing Telemetry Transport*, is an asynchronous data exchange protocol based on publish/subscribe mechanism [44]. The MQTT protocol is message-oriented in the sense that connected clients can use it to communicate indirectly with each other via a centralised broker [45]. The MQTT protocol enables authorised entities to publish or subscribe to topics of their interest. A published message to a topic can notify or control all entities subscribed to that particular topic. The MQTT broker manages these topics and orchestrates messages exchange conducted on them. Figure 4.1 illustrates an example of the use of MQTT in the context of an IoT-based healthcare scenario.

On the other hand, HTTP, short for *Hypertext Transfer Protocol*, is an asynchronous data exchange protocol based on request/response mechanism [46]. The HTTP protocol is resource-oriented in the sense that it provides a set of methods (i.e. GET, POST, UPDATE, DELETE, etc.), which the client can use to indicate the desired action on a resource. The server answers with a response code to inform about the request status, such as 2XX to indicate success or 4XX to indicate client error. Figure 5.2 illustrates an IoT-based firefighting system that uses HTTP as a communication protocol between edge computing units and the cloud. Appendix B provides further description of the implementation.

2.1.2. IoT Simulators

Gaining access to a large-scale IoT infrastructure can pose a real challenge for research and development. Nevertheless, several use cases can leverage IoT simulators in order to compensate for this shortcoming [32]. Furthermore, there have been several works on IoT simulation, such as those covered by [47][48][33]. Therefore, intending to select an end-to-end IoT simulator for chapter 6, this thesis requires the IoT simulator to satisfy the following criteria:

- It must cover end-to-end IoT including, but not limited to, cloud, network, edge, devices/sensors.
- It must be open source and cross-platform for extension purposes.
- It should be written in Java in order to natively support the proposed blockchain-based middleware in chapter 6.
- It must enable modelling quality requirements at various layers.

These criteria resulted in considering iFogSim[1][49], MyiFogSim[2][50], and IoTSim-Osmosis[3][33]. It excludes other IoT layer-specific simulators such as CloudSim [4] [51], GreenCloud [5] [52], and Edge-Fog [6][53]. The considered IoT simulators are based on cloudSim, a highly cited and recognised cloud simulator in the literature [48]. MyiFogSim extends iFogSim, which extends the CloudSim simulator, while IoTSim-Osmosis directly extends the cloudSim simulator. For the purposes of this study, we select IoTSim-Osmosis.

2.2. Overview on Traditional SLA Practice

Service Level Agreement (SLA) is a contractual method that regulates and governs the relationship between service providers and consumers[16][54]. In the traditional SLA practice, several well-established organisations contribute their effort in standardising the SLA practice such as ISO [8], Van der Wees Arthur et al.[55], Bakalos et al.[9], TMForum[56], OMG Cloud Working Group[21] and others. Whenever this thesis mentions either *traditional SLA practice* or *current SLA practice*, it essentially refers to the conventional practice where blockchain is not employed in any form. The current SLA practice typically subjects SLA to a life-cycle from the SLA definition until termination [7] [10]. While these SLA guidelines and standards arrange the SLA stages differently, this thesis adopts a compact lifecycle of SLA phases as depicted in Figure 2.1, and describes them as follows:

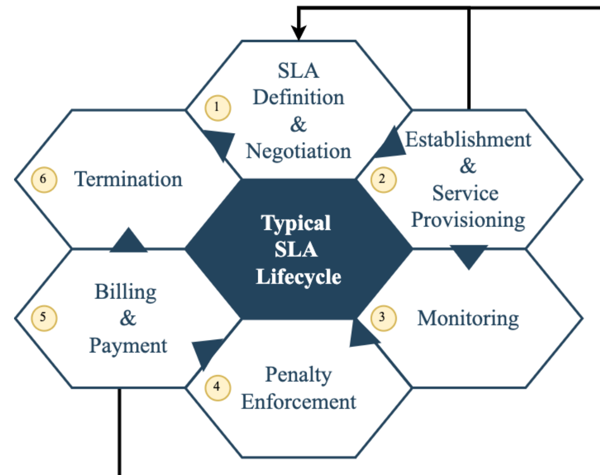


Figure 2.1 Compact SLA Life Cycle Phases

2.2.1. *SLA Definition and Negotiation*

According to most SLA standards and guidelines, highlighted in Section 2.2, the minimal form of an SLA defines the following:

[1]https://github.com/Cloudslab/iFogSim
[2]https://github.com/marciocomp/myifogsim
[3]https://github.com/kalwasel/IoTSim-Osmosis
[4]https://github.com/Cloudslab/cloudsim
[5]https://greencloud.gforge.uni.lu/install.html
[6]https://github.com/nitindermohan/EdgeFogSimulator

```
1  {
2    "SLA_Parties": [
3      {
4        "ID": "ID0001",
5        "type": "Provider",
6        "name": "IoT Service Provider"
7      },
8      {
9        "ID": "ID0002",
10       "type": "Consumer",
11       "name": "Fire Station"
12     }
13   ],
14   "Service_Level_Objectives": [
15     {
16       "Quality_Requirement_ID": "QoS0001",
17       "Quality_Requirement_Name": "Availability",
18       "Required_Level": "Graeter Than",
19       "Threshold": "99",
20       "Unit": "Percentage",
21       "Obligated_party": "0001"
22     },
23     {
24       "Quality_Requirement_ID": "QoS0002",
25       "Quality_Requirement_Name": "Transmission_Time",
26       "Required_Level": "Less Than",
27       "Threshold": "3",
28       "Unit": "Seconds",
29       "Obligated_party": "0001"
30     }
31   ],
32   "Violation_Consequences": [
33     {
34       "Quality_Requirement_ID": "QoS0001",
35       "Penalty": "0.25",
36       "Unit": "Percentage",
37       "Frequency": "1"
38     },
39     {
40       "Quality_Requirement_ID": "QoS0002",
41       "Penalty": "0.5",
42       "Unit": "Percentage",
43       "Frequency": "1000"
44     }
45   ]
46 }
```

Figure 2.2 SLA example between a service provider and consumers

- *SLA participants*: involved parties and their roles (i.e. service provider, consumers, and probably a third party such as auditors and assessors).
- *Service Level Objectives (SLOs)*: a set of obligations and responsibilities carried out by the service provider. This thesis maps an SLO to a measurable service quality requirement such as availability, throughput, latency, jitter, packet loss rate, and so-forth [57]. For instance, availability must not be less than 99.9% all the time.
- *Violation Consequences*: a set of measures that follows a failure in meeting the service level objectives such as imposing a penalty (e.g financial service credit).
- Dates of SLA establishment and termination, and what conditions that triggers them.

Girs et al.[12] conducts an extensive survey on SLA definition and modelling in the context of Cloud-based IoT services. To date, the most mature IoT-based SLA specification framework is the one proposed by Alqahtani et al. [11], which covers the requirements of an end-to-end IoT ecosystem. This study adopts their proposed SLA framework to compose and define a simplified SLA example between an IoT service provider and a consumer, an example of which is presented in Figure 2.2.

The agreement comprises three main sections: SLA parties, Service Level Objectives, and Violation Consequences. The SLA parties section (lines: 1-13) consists of the details of both the service provider and a consumer. The Service Level Objectives (SLOs) section dictates a set of quality requirements promised by the service provider. Whereas the first SLO (lines: 15-22) stipulates a quality requirement $Availability \geq 99\%$, the second SLO (lines: 23-30) states a quality requirement $Latency < 3s$. The last section lists a set of example violation consequences (lines 32-46) in the form of penalties applied on the obligated party (the service provider), shall it fail to scale to the exceptions of the consumer. For instance, a failure to meet the availability requirement *QoS0001* incurs a financial credit of 25% of the agreed cost. The latency requirement *QoS0001* incurs a financial credit of 50% for every 1000 breaches.

As with any contractual method, SLA can be negotiated either before or after SLA establishment [58]. For that, the ISO/IEC 19086-1 standard [59] recommends accounting for changes to the SLA. That is, some service providers may customise a predefined SLA, such as the one presented in Figure 2.2, to match some consumers' specific requirements [60]. For example, a

consumer may suggest an amendment to the SLA before engagement in the contractual relationship. On the other hand, a renegotiation may also occur after SLA establishment [61], which can be triggered due to a constant failure by the service provider in satisfying the promised quality requirements. Subsequently, the SLA participants may consider adapting the SLA version to match a realistic performance [9].

Moreover, consider the possibility of a new owner acquiring a service provider or merging with another organisation. Therefore, the SLA may also change accordingly to reflect the new service provider and any updates on SLA terms. There are also cases where constant SLA violations lead to termination. Service providers and consumers may renegotiate the SLA instead of termination [16][7].

2.2.2. SLA Establishment and Service Provisioning

SLA establishment refers to the transition from SLA definition or negotiation stages to a final and enforceable agreement. In current practice, declaring the SLA establishment can take multiple forms. For instance, SLA participants may electronically sign a predefined or negotiated agreement to declare their engagement and commitment [7]. Moreover, SLA establishment can be declared upon actual provisioning of the service [62]. For example, we can deem the SLA established when the agreement is signed, and the service provider provisions the agreed service to the consumer.

2.2.3. Monitoring

Once the SLA is established, and the service is provisioned, the monitoring phase takes place to ensure SLA compliance [63]. Monitoring plays vital role in forming critical decision about the compliance of service providers [16]. From the perspective of a consumer, monitoring help in detecting SLA violations and supporting their claims [18]. From the perspective of service providers, monitoring help in identifying signs of service degradation in order to proactively mitigate them before violating the quality requirements in the SLA [17]. It also helps verify any consumers' violation claims [15].

Mubeen et al.[10] recognise that consumers may employ monitoring tools to compensate for the lack of access to the underlying infrastructure of the service provider. They describe SLA monitoring as the mission to confirm whether the performance of the service provider matches the promised service quality requirements. Mubeen et al. [10] survey SLA monitoring related studies and point out that it attracts considerable attention compared to other SLA stages.

Hussain et al.[15] view continuous SLA monitoring as a critical trust enabler. They explore how existing monitoring approaches serve trust between SLA participants and classify violation detection into either proactive or reactive. The former aims to detect SLA violations beforehand, while the latter detects SLA violations after their occurrence.

We find in the literature a correlation between SLA definition and SLA monitoring. For example, Labidi et al.[18] stress the importance of machine-readable SLA in order to allow monitoring tools to interpret them unambiguously. Zhang et al.[63] recommend defining quality

requirements based on historical data in order to attain a reasonable agreement. That is, service providers can judge whether they can realise the expectation of their consumers based on their historical monitoring logs. On the other hand, consumers can confirm the competency of a service provider based on its historical monitoring logs. Accordingly, historical monitoring logs from both sides can help establish a degree of trust. However, we would question the safety of these monitoring logs from forgery and misconduct.

The practical guide by OMG Cloud Working Group[21] believe that SLA monitoring is not only important for trust establishment but also for service management and dispute resolution. It highlights various monitoring models in practice as follows:

- The service provider offers their consumers an interface to its monitoring tools. In practice, most cloud providers offer their consumers a subscription option to monitoring and alerting services such as AWS, GCP and Azure. However, the guide report note that provider-dependent monitoring tools are prone to downtime. Additionally, service providers may intentionally limit the capability of their offered monitoring tool.
- Consumers may employ monitoring tools independent from the service provider, for instance, by deploying a monitoring tool under their control or by resorting to a third party. However, the service provider may not acknowledge logs produced by unrecognised monitoring tools.

2.2.4. SLA Enforcement

While most SLA guidelines, standards and related works consider enforcement as a primary phase of a typical SLA lifecycle, there is no commonly agreed-upon definition for it in the literature. For example, Kyriazis[17] considers SLA enforcement as the attainment of quality requirements by leveraging monitoring tools to identify possible breaches and employing corrective methods to recover from violations. Kyriazis[17] also review a set of related works that explore SLA enforcement in the traditional practice. On the other hand, Wu and Buyya[7] view SLA enforcement differently as a means of enforcing penalties associated with quality requirements. Rana et al. [16] also discuss SLA enforcement in the context of violation penalties.

From where we stand on the SLA literature and the current practice, we distinguish between two major categories of SLA enforcement, which are *violation prevention* and *consequences execution*. The main goal of the former category is to proactively predict and prevent SLA violation in the first place, which is of interest to both the service provider and consumers. The latter category addresses the aftermath of SLA violation which is concerned more with consumers. Figure 2.3 which maps each enforcement category with examples to their corresponding SLA part. The following subsections delve further into these categories.

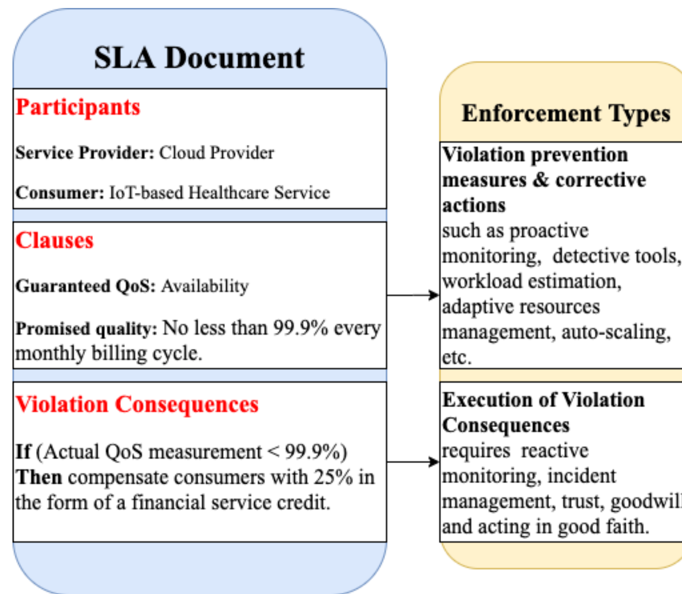


Figure 2.3 Basic SLA structure and types of SLA enforcement mapped to SLA elements.

2.2.4.1. Violation Prevention

While this thesis is limited to enforcing the execution of SLA violation consequences, it is worth briefly looking into enforcing SLA terms using preventive measures. This is to draw a clear distinction between both classes of SLA enforcement.

According to relevant guidelines and standards such as Hogan et al. [64], Kyriazis [17], and OMG Cloud Working Group [21], service providers should always mitigate the risk of SLA violation consequences. For example, by employing advanced monitoring and detective tools that can predict possible failures based on repetitive patterns, workload estimation, performance forecasting and so forth. Furthermore, providers should promptly respond to possible violation risks by applying autonomous corrective strategies (i.e. adaptive resources management, auto-scaling, replication, and so forth). For instance, the work by Wong et al. [65] presents a set of violation prevention techniques such as cloud horizontal scaling, fault tolerance and recovery. Nawaz et al.[66] call for considering proactive identification of not only internal events but also external co-factors, even if they are beyond the immediate control of cloud providers because they would impact the overall service quality.

All in all, violation preventive methods have the ultimate goal to avoid failure scenarios and strive for as quality service delivery as possible. However, the success of proactive approaches depend on their efficiency which is eventually limited by available resources [17].

2.2.4.2. Violation Consequences Execution

While prevention methods present an interesting class of problems, it is also important to discuss the aftermath of SLA violations. More specifically, when all proactive measures fail to prevent the occurrence of an incident, consumers expect service providers to apply violation consequences stated in the SLA [16][58][18]. Consider the violation consequences in example SLA presented in Figure 2.2, which imposes a set of penalties on the service provider in case of violating their

respective quality requirements. However, there remains the question of who should be trusted on enforcing agreed consequences. Here emerges the interest of this thesis in using blockchain to resolve trust issues related to this enforcement category.

2.2.5. *Billing and SLA Termination*

SLA must stipulate service fees, billing cycle, and penalties clauses [67]. During the SLA lifecycle, enforcement mechanisms should apply penalties on the service providers as stated by the SLA. On the other hand, consumers must pay the service fee on the due date. SLA also state the validity period and termination clauses [62]. For example, consumers may have the option to withdraw from the contractual relationship if the service provider constantly demonstrates incompetency in the compliance status [16]. Service providers may also have the option to deprecate the current SLA and replace it with an updated version. Depending on SLA clauses, consumers or service providers may have the option to renegotiate the SLA before termination.

2.2.6. *SLA-guaranteed Cloud-based IoT Services*

Several cloud providers respond to IoT phenomena by providing services tailored for IoT purposes. Table 2.1 lists a set of cloud providers that serve an IoT component as a service. As Figure 2.4 depicts, the IoT component enables accessing, managing, controlling and monitoring IoT assets. It also acts as a gateway to other cloud services such as computing units, time-series and historical storage, analysis, machine learning capabilities, and visualisation, which help accommodate IoT data sets and generate value out of them. Cloud-based IoT components usually support several communication protocols that can bridge the gap between cloud services and IoT assets such as HTTP, CoAP and XMPP, and MQTT.

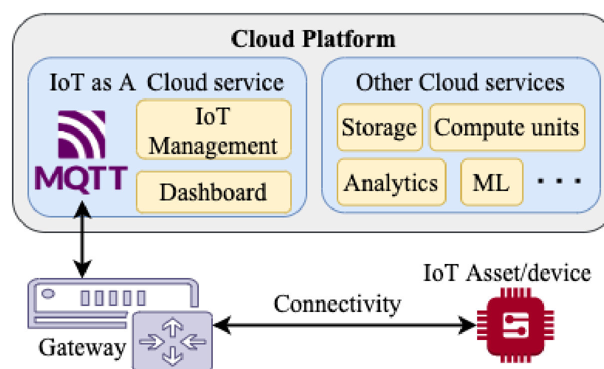


Figure 2.4 An simplified example of an IoT as a cloud service.

Due to the interest of this thesis in SLA in the context of IoT, we consider prominent cloud providers that provide IoT services, which are AWS, GCP and Microsoft Azure. All of them provide a cloud-based IoT component that serves IoT purposes over both MQTT and HTTP protocols. Table 2.1 maps each of the example cloud providers with the link to their offered SLA and snapshot of their current SLA provided in Appendix A. Figure 2.2 summaries the structure of their offered SLAs. All of these cloud providers employ the SLA to illustrate what service level that consumers should expect. That is, all of the sample SLAs promise a high quality uptime of

there service on monthly basis. For instance, an SLA primroses quality uptime of no less than 99% all the time. It is worth noting that SLA, in practice, is not limited to the uptime quality but can extend to latency, jitter, throughput, and other quality requirements as per negotiated and agreed by SLA parties.

Table 2.1 Sample SLA-guaranteed Cloud-based IoT Service

	Amazon Web Services (AWS)	Google Cloud Platform (GCP)	Microsoft Azure
SLA	Appendix A Section A.2.1	Appendix A Section A.2.2	Appendix A Section A.2.3

However, consumers must also clarify what constitute an uptime or otherwise. For that, a decent SLA must illustrate a clear definition of the uptime, how the it is calculated, what conditions that precisely deem the provider in violation and what exclusions might be. For example, the Azure's SLA measures an HTTP-based IoT server uptime as $\frac{u-d}{u} \times 100$, where u denotes the uptime of their IoT component while d denotes the downtime of their IoT component. Both u and d are measured in minutes. Both GCP and AWS use the same measurement for their provisioned HTTP and MQTT IoT services.

While these service providers use the same formula for calculating the uptime, they differ in terms of the definition of both uptime and downtime. For instance, Azure does not deem itself in violation to the service uptime quality unless the their service is down for more than one minute. We can understand from that, Azure does not deem IoT component as down even when it experiences a downtime briefly within less than 60 seconds. The AWS SLA does not consider its service down at all if the IoT client did not consume or publish HTTP or MQTT during the service downtime. The GCP does not consider its IoT component unavailable unless the IoT client experience an error rate over than 10% of its requests during the the downtime period.

As can be seen, cloud providers vigilantly craft their SLA to avoid being liable to penalties. Above that, they also ensure their committent in satisfying the quality level of IoT service. They do so by promising to impose a penalty on themselves if they fail to deliver the stipulated quality level. For instance, they oblige to compensate their consumers with 10% of the monthly paid service fees in form of a service credit. However, none of the sample SLAs obligate the service provider to proactively process violation incident as they occur. In fact, all of them consider the consumer's responsible for manually submitting a violation claim supported with irrefutable evident. Evidence can include monitoring logs that requires their confirmation and acknowledgment.

2.2.7. The Trust Dilemma

SLA is, as any contractual method, is not immune from trust issues. Huang and Nicol [23] base the trust terminology on three pillars as follows:

- **Expectancy:** consumers expect service providers to behave as promised in the SLA and will always maintain reputation qualities such as cooperation, transparency, compliance with regulations, truthfulness, attentive response to incidents.

- **Belief:** consumers are in a mental state to assume that service providers are highly likely to meet the above expectations. This can be based on a relevant evidence scheme, such as reputation, prior experience, users feedback, assessment and auditing methods, etc.

- **Willingness to take risk:** despite the previous two elements, consumers consciously tolerate risks that may occur as a consequence of a false impression, misjudgement, or unplanned events.

In many cases, a varying degree of trust is inescapable in contractual relationships [68]. There are also cases where consumers are in a position where they blindly trust the service provider [9]. However, when outsourcing a critical mission to another party, trust can be questionable for several reasons, such as the lack of transparency of internal components and procedures [69]. The literature and industry alike have been examining various trust establishment methods. For example, both studies by Hussain et al.[15] and Zhang et al.[63] consider service monitoring as an essential trust enabler.

However, several studies raise concerns about which SLA participant should be trusted for accurate and reliable monitoring logs, thus suggesting a trusted third party as an alternative [24] [17][16]. Nevertheless, Park et al.[67] point out the impact of untrusted monitoring on dispute resolution and penalty enforcement. Both [23] [23] and Habib et al.[13] explores various trust establishment mechanisms such as feedback systems, reputation-based mechanism, and resorting to trusted third parties such as brokers, auditors and assessors. Finally, Mubeen et al.[10] provide a survey on existing SLA trustworthiness mechanisms in the literature.

Assume the dependability and trustworthiness of a service provider is properly verified before engagement in the SLA contractual relationship. Moreover, assume a trusted monitoring mechanism is in place. Figure 2.5 revisits the current SLA practice and raises the question of whom should be trusted for compliance assessment and penalty enforcement. Most service providers promise to comply with the SLA and process incidents in good faith, assuring their consumers to impose violation consequences on themselves [9].

While intriguing, it is typically the consumer's responsibility to report a service level degradation, supported by evidence deemed irrefutable by the service provider or trusted third parties [9]. This is usually a tedious process, manually handled, time-consuming, error-prone, and requires consumers' goodwill [19]. In some cases, service providers may not react well to poorly formed claims, regardless of their validity [20]. Both sides of a contractual relationship, service providers or consumers, may find it inviting to intentionally fabricate or manipulate evidence of violation incidents in order to maximise profit or avoid hefty penalty [13]. In some scenarios, unresolved disputes have to be escalated to jurisdiction means [16].

Park et al. [67] recognise that monitoring data can be prone to malicious acts, and therefore it proposes shifting trust from service provider to an independent authority in terms of monitoring and billing. Wu and Buyya [7] also suggests trusting a third party for enforcing penalties on service providers in case of SLA violations. However, this thesis argues that shifting trust from service providers to a third party may improve trust establishment but does not guarantee the non-repudiation and honesty of third parties.

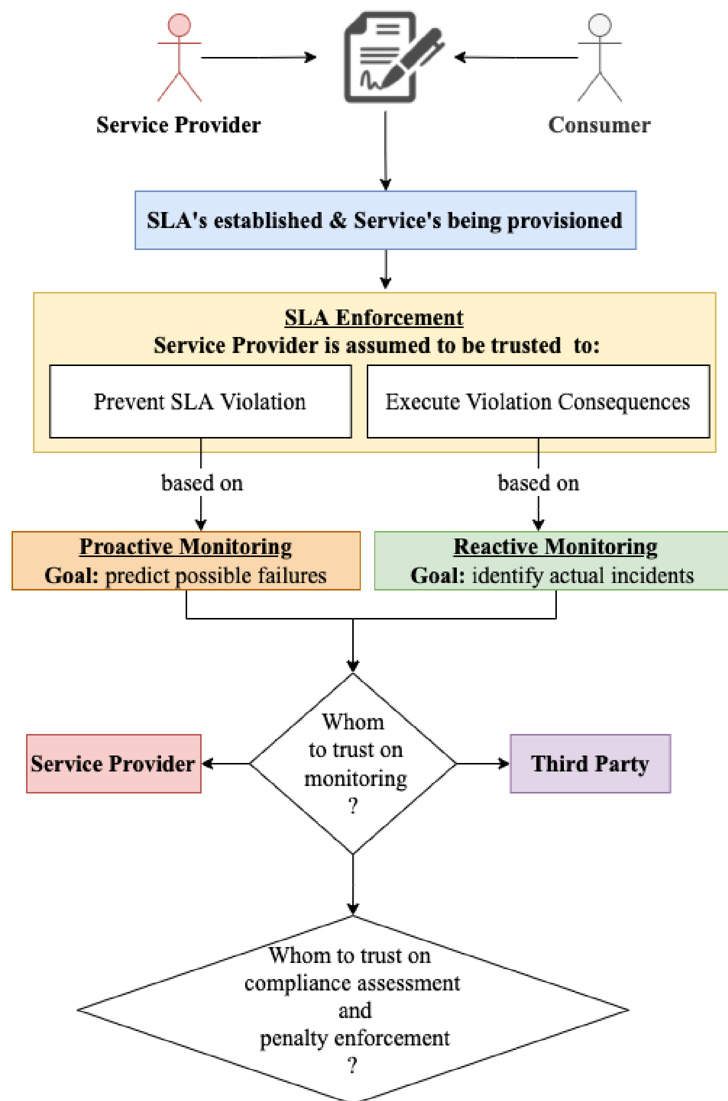


Figure 2.5 Overview on current trust practice for SLA enforcement

2.3. Overview on Blockchain

Whenever trust is a burden, blockchain presents itself as a potential solution. Blockchain holds appealing principles including, but not limited to, immutability, decentralisation, and an append-only shared ledger controlled by a consensus mechanism [70][71]. The concept of a smart contract allows applications to benefit from these features, forming what is commonly referred to as Decentralised Applications (DAPPs) [72]. Therefore, we see an opportunity in taking advantage of Blockchain to service SLA compliance assessment and penalty enforcement. Blockchain does not only address trust issues [20] but can also automate SLA-related tasks in a decentralised fashion. This section delves into Blockchain by providing a brief overview of Blockchain history. Then, it briefly sheds light on key blockchain characteristics and types of blockchain networks.

2.3.1. Blockchain Chronology

The emergence of the blockchain paradigm was motivated by the urge to mitigate the need for trust schemes that rely on centralised authorities or third party solutions [68]. We can trace the emergence of Blockchain principles back to a proposal by Haber and Stornetta [73], which addresses the trustworthiness of centralised time-stamping services. The motivation behind their proposal includes typical trust issues such as integrity, counterfeiting and collusion. While their work was not known by the name "Blockchain", it established the foundation of an anti-tampering data structure that links hashes of entries in an append-only fashion. Following, Bitcoin [26] is the first work that proposed the blockchain concept in a viable manner. In the subsequent year, this concept was known as Bitcoin, a decentralised cryptocurrency system that aims to eliminate the need for trusted financial institutions. Following, Ethereum emerged as a decentralised application platform that generalises the usage of Blockchain beyond cryptocurrency purposes, promoting the concept of smart contracts [28][27]. Such a blockchain platform enables different domains and applications to benefit from blockchain advantages such as decentralisation, immutability, transparency, data integrity, elimination of centralised authority, and robust consensus mechanism [74][75][76]. Since then, we have seen various blockchain-based applications serving different domains such as healthcare, insurance, energy, transportation, etc. [68][72]. Hyperledger Fabric [29] followed suit by proposing a modular blockchain platform that is permissioned in nature and considers enterprise needs.

2.3.2. Blockchain Characteristics

The *blockchain* terminology can be used interchangeably to refer to ledger structure or be generalised to the decentralised infrastructure, including the ledger itself, p2p network, consensus mechanism, and so forth. This section highlights common Blockchain characteristics employed by prominent networks, which are Bitcoin, Ethereum, and Hyperledger Fabric [77] [25].

Decentralisation

Most blockchain platforms are based on a decentralised peer-to-peer network that mitigates the need for single authorities [78][25]. That is, multiple geographically distributed nodes (peers) can participate in a blockchain network to maintain the validity, availability and perpetuity of the shared ledger [72]. The National Institute of Standards and Technology (NIST) [71] points out that the decentralisation of blockchain networks can mitigate trust issues and prevent a single point of failures in multiple ways, such as:

- No single entity gains ultimate control over the shared ledger, preventing misconduct and malicious behaviour.
- The ledger replication and geographically distributed locations promise high availability and continuous accessibility.
- The absence of centralised authority consolidates transparency among the network participants.
- The blockchain's distributed architecture is more resistant and resilient to security issues than centralised systems.

Blockchain networks can be either private, consortium or public [31] [77]. Both Ethereum and Bitcoin [26] are examples of public blockchain networks that allows nodes to participate anonymously on the basis of joining or leaving anytime. An example of the consortium blockchain network is Hyperledger Fabric, where participation in the network is limited to permissioned nodes with proper authentication and authorisation. Unlike Ethereum, validating nodes are committed to the Hyperledger Fabric network and are not expected to leave or join at will. Any of these blockchain networks can be adjusted to be deployed in private settings (e.g. a network belonging to one organisation), which is not truly decentralised.

Consensus Protocols

The decentralisation of blockchain network mitigates centralised authorities. That is, a properly decentralised blockchain network does not grant a single entity solo governance on transactions validation and ledger maintenance. Instead, each validating peer equally participates in processing received transactions, which are to be ordered into blocks and appended to the ledger in a decentralised fashion [31]. Due to the absence of centralised authorities, consensus mechanism has to be in place to enable coordination among validators, and ensure transactions finality [41]. Moreover, consensus mechanisms ensure integrity and consistency of transactions across the blockchain network [31].

Several studies discuss various consensus protocols employed by different blockchain network such as [79][25][77]. For instance, both of Bitcoin and Ethereum currently employ Proof-of-Work (PoW) as consensus mechanism. However, the mean Ethereum network is transitioning to a more lightweight protocol; namely Proof-of-Stake (PoS). There are other variations

of independent Ethereum-based networks that adopt other consensus protocols such Proof of Authority (PoA) and Proof of Elapsed time (PoET). On other hand, Hyperledger Fabric, being a consortium and permissioned network, pursue a modular approach in terms of consensus mechanisms such that any protocol, in theory and by design, may be adopted [29]. However, the current officially supported and recommended consensus mechanism for Hyperledger Fabric is a leader-follower protocol called Raft [80].

Blockchain (Ledger) Structure

Every validating node in the blockchain network maintains a full Blockchain ledger replica. Figure 2.6 presents a basic blockchain structure and illustrates a set of transactions data $T_1, T_2, T_3, \ldots, T_n$ ordered and organised into a collection of linked blocks [71]. Newly validated transactions are grouped into a candidate block depending on the transaction flow imposed by the blockchain network. If the candidate block meets the employed consensus protocol, then every node updates its ledger copy by appending the candidate block. According to Sanka and Cheung [77], Blockchain is said to be an append-only ledger because the candidate block's header comprises a hash digest of the following:

- The previous block's header.
- Its own content (e.g.block number, timestamp, Merkle-root of included transactions, nonce, and other attributes that vary depending on the blockchain network) .

This cryptographic linking mechanism helps preserving the ledger's integrity and immutability in two ways. First, malicious nodes encounter the challenge of traversing backsword the entire cryptographic chain of blocks. Second, even a successful attempt will break the chain of hashes, leading to invalidating the manipulated copy of the ledger, thus revealing misconduct by the malicious node. The employed consensus mechanism also protects the ledger integrity by refusing any candidate block produced based on a ledger copy inconstant the those maintained by the majority.

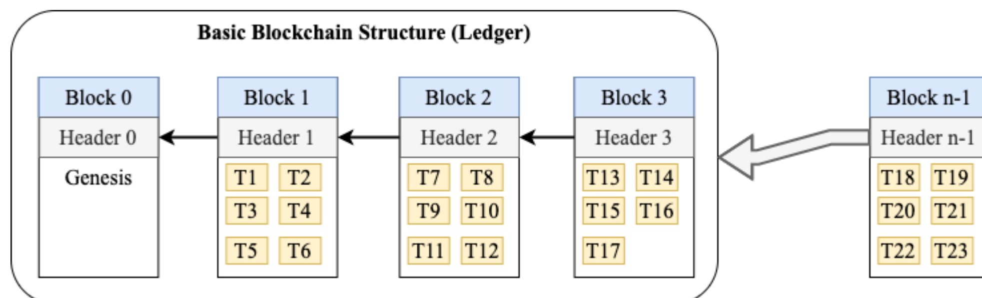


Figure 2.6 Visualisation of a basic Blockchain structure (A chain of blocks).

The Double-Spending Problem

Prior to Bitcoin, several attempt, such as eCash [81], suffered from the double-spending issue, where the same asset can be maliciously spent twice for various purposes simultaneously [25].

For that, a trusted centralised authority is required to mitigate this problem. Due to the absence of centralised authorities, Blockchain networks are designed to prevent this double-spending problem [71]. Most blockchain networks do not execute transactions individually but the entire block of transactions. For example, assume two transactions that attempt to consume the same asset in the same block. Subsequently, one can observe the importance of transactions ordering within the block, such that they are executed according to their priority. However, different blockchains differ in how they mitigate the problem. Diffident blockchain networks employ distinctive transactions validation mechanisms and ordering systems.

Vujičić et al. [82] provide an insight into how Bitcoin and Ethereum mitigate the double-spending problem. Bitcoin employs the concept of Unspent Transaction Output (UTXO) and Spent Transaction Output (STXO) to track assets status. It maps every output to an amount of its cryptocurrency. Bitcoin forbids spending an STXO twice in order to prevent the double-spending problem. On the other hand, Ethereum is an account-based blockchain network, and therefore it assigns each transaction from each account a unique nonce to guarantee that each transaction is executed once. The PoW consensus mechanism, along with the blockchain structure, further ensure that no malicious attempt by validating nodes to change the status of an asset from STXO to UTXO Bitcoin-wise or executing a transaction twice with the same nonce from the same account Ethereum-wise. This thesis focuses on Hyperledger Fabric which follows a unique transaction validation and execution flow, discussed in section 2.4.2.

Smart Contracts

Following the inception of Bitcoin, several generic blockchain platforms emerged to enable leveraging Blockchain characteristics beyond cryptocurrencies [70]. For example, both Ethereum and Hyperledger Fabric enable smart contract deployment and execution capability. Solidity is one of the programming languages for writing smart contracts for Ethereum. Hyperledger Fabric can deploy and execute smart contracts in multiple programming languages such as Java, Javascript and Golang.

The blockchain literature widely cites the smart contract definition by Szabo 1994 [83], which this thesis directly quotes as follows *"a computerised transaction protocol that executes the terms of a contract. The general objectives of smart contract design are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimise exceptions both malicious and accidental, and minimise the need for trusted intermediaries."* [71].

While Szabo envisioned the concept of smart contracts decades ago, it had not truly materialised to existence until the emergence of both Ethereum and Hyperledger Fabric [31]. In these blockchain platforms, the concept of smart contracts enables coding an event-driven and autonomous program executable within a blockchain environment beyond the immediate control of a single authority [27] [29]. Hereafter, these programs are conveniently referred to as smart contracts. They benefit from the features of the underlying blockchain platform such as decentralisation, resistance to the single point of failure, consensus mechanisms.

Wang et al. [31] note that the blockchain-based smart contract can be enforceable in a decentralised manner without intervention from a centralised authority or a third party. They also add that smart contracts act as a gateway for transactions processing and validation by participating nodes. We can infer from the study by Zou et al. [84] that a smart contract is a set of predefined conditions such that, when triggered, they are automatically executable over blockchain in a decentralised manner without human intervention. The execution of smart contracts is governed by the underlying blockchain platform's transaction flow and consensus mechanism.

From the above description of the smart contract concept, we can observe a high degree of relevance between the smart contract concept and the purposes of this thesis, particularly in terms of decentralising SLA-related tasks such as compliance assessment and enforcement. Romano and Schmid [70] also confirm the ability of smart contracts to automate the logic of traditional contracts, which does not only help save cost but also mitigates human errors and trust issues. They also the highlight blockchain's potential in enabling transparent compliance assessment and a unified view over the shared ledger. Wöhrer and Zdun [85] also believe that smart contracts can enforce the execution of predefined and negotiated contracts.

2.4. Hyperledger Fabric

Due to reasons discussed in section 2.5, this thesis selects Hyperledger Fabric as the underlying blockchain platform for decentralising SLA-related tasks. According to Androulaki et al. [29], Hyperledger Fabric, Hereafter abbreviated as HLF, is a permissioned blockchain network and supports modular consensus mechanisms and general-purpose programming languages. HLF is an enterprise-grade blockchain platform that does not depend on cryptocurrency and strives to meet industry needed while preserving key blockchain characteristics discussed above. This section sheds light on the distinctive features of a basic HLF network. It also discusses the HLF's transaction flow model and performance.

2.4.1. Basic HLF Network

This section derives its content from the official HLF's documentation[7] and the peer reviewed article [29]. Assumes an alliance of multiple known organisations that serve a common goal or share the same objective. In our case, these organisations can include any of the service providers, consumers, regulators, auditors, and so forth. Hyperledger Fabric enables these organisations to form a blockchain-based decentralised network. For simplicity, Figure 2.7 depicts an example of a basic network of two organisations (org1 and org2), such that each organisation contributes the following:

[7] https://hyperledger-fabric.readthedocs.io/en/latest/index.html

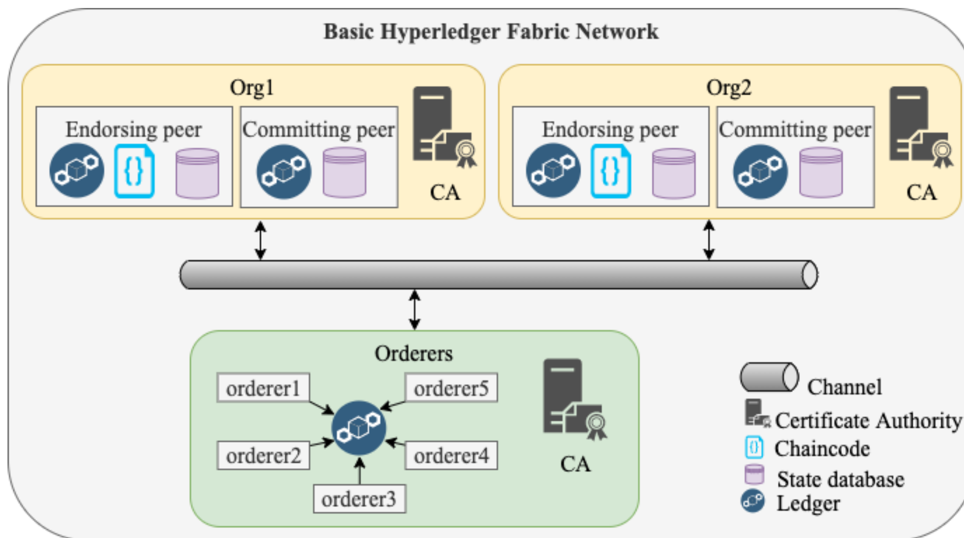


Figure 2.7 Basic Hyperledger Fabric Network

Nodes

Hyperledger Fabric, unlike Ethereum, follows an Execute-Order-Validate model, as discussed in section 2.4.2. Therefore, a node can take one of the following roles, which are:

- An endorsing peer, a node that maintains a replica of smart contracts, the state storage and the shared ledger. The endorsing peer participates in transactions execution, endorsement, validation, and ledger update.
- A committing peer, a node that maintains a replica of the state storage and the shared ledger. Unlike endorsing peers, it does not participate in transaction endorsement and validation due to the lack of smart contracts capability. However, committing peers still participate in transaction validation and ledger update.
- An orderer peer, a node that orders endorsed transactions into blocks and disseminates generated blocks to both committing and endorsing peers.

Due to the modularity of HLF, any architecture can impose the amount of each node type that an organisation must contribute to the network. For any node type, the participating organisation dedicates computational and storage resources.

Chaincode

HLF supports encoding smart contracts using different programming languages (currently supports: Java, Go, and JavaScript). A smart contract can represent the business logic as well as the storage data schema. For example, a smart contract can represent a data model for a quality requirement and related metrics. A smart contract can also define the logic of CRUD operations on such data model (Create, Read, Update,and Delete). HLF packages a collection of smart contracts into the same namespace, called Chaincode. Accordingly, all smart contracts living in the same Chaincode container directly access state storage within their shared namespace.

External smart contracts may access external state storage by invoking an internal smart contract living in the same namespace, given that both smart contracts are granted appropriate permissions. Figure 3.8 depicts an example of how we leverage the Chaincode capability for modelling the state storage and SLA-based smart contracts.

State Storage

The state storage is a document-based database that structures records in the form of a k, v, ver, where k is a unique record key, v is the value of the record, and ver is an incremental nonce used to track changes on the record. As the name indicates, the state storage reflects the last state of an asset as per the shared ledger. Figure 2.8 further illustrates a smart contract that mediates between car sellers and buyers illustrates a basic smart contract that mediates between a service provider and a consumer. The state storage reflects the latest status of an SLA in the form of k, v, ver, where k is the SLA ID, v is the current status of the SLA (Negotiation, Established, or Terminated), and ver tracks the changes on the SLA record. The Figure shows multiple records of the same asset, where $SLA_{k=1}$, for illustration purposes. In fact, the state storage only maintains one record of the asset along with its last state SLA_v, which does not change unless supported by a valid transaction immutably persisted in the ledger. Chapter 3 provides more detail on how this thesis leverages the state storage for SLA representation and awareness within the blockchain.

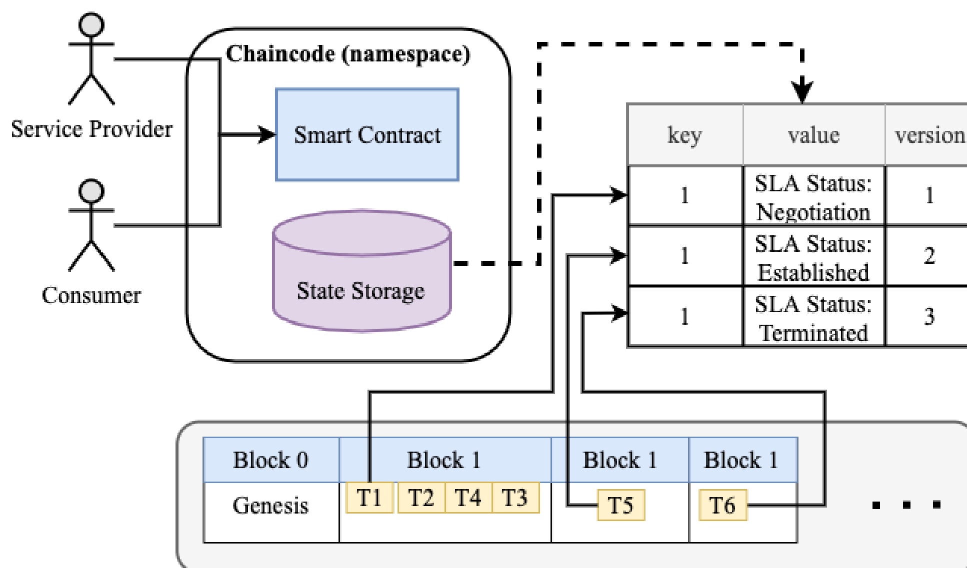


Figure 2.8 An illustration of the state storage and data organisation

Channels

HLF provides an isolation mechanism that encapsulates a set of participating organisations in a virtually separate blockchain network [29]. Therefore, the channel feature enables organisations serving a common goal to operate and communicate privately with each other. Each organisation in the channel share in common the same ledger, chaincode, and state storage.

Certificate Authority (CA)

Since HLF is a permissioned blockchain platform, it employs a Public Key Infrastructure (PKI) for authentication and authorisation purposes. Each organisation deploys a certificate authority to manage identity issuance, permissions, validation, and revocation for its members, including peers, admins, client applications, and orderers. In this thesis, clients can be service providers, consumers, a monitoring tool, a simulator, etc. A PKI typically comprises key elements: certificates authority (CA), public/private key pair, X.509 certificates, and Certificate Revocation Lists (CRL). Figure 6.3 illustrates the process of issuing a key pair for a participating entity. When communicating or transacting with the blockchain, X.509 certificates are vital for integrity verification. The private key proves authenticity and ownership of identity. For example, the application client can sign transactions using the private key, resulting in a signature verifiable by any recipient using the application client's public key.

2.4.2. Transaction Flow

Unlike conventional applications, no blockchain operation is considered to be valid unless it undergoes a set of validation mechanisms such as ESCC (Endorsement System Chaincode), VSCC (Validation System Chaincode) and MVCC (Multi-Version Concurrency Control [29] (See Figure 5.13). Androulaki et al. [29] describe the transaction flow in the HLF network, which covers the transactions journey from being submitted by an application client until being successfully validated and committed by all peers in the channel.

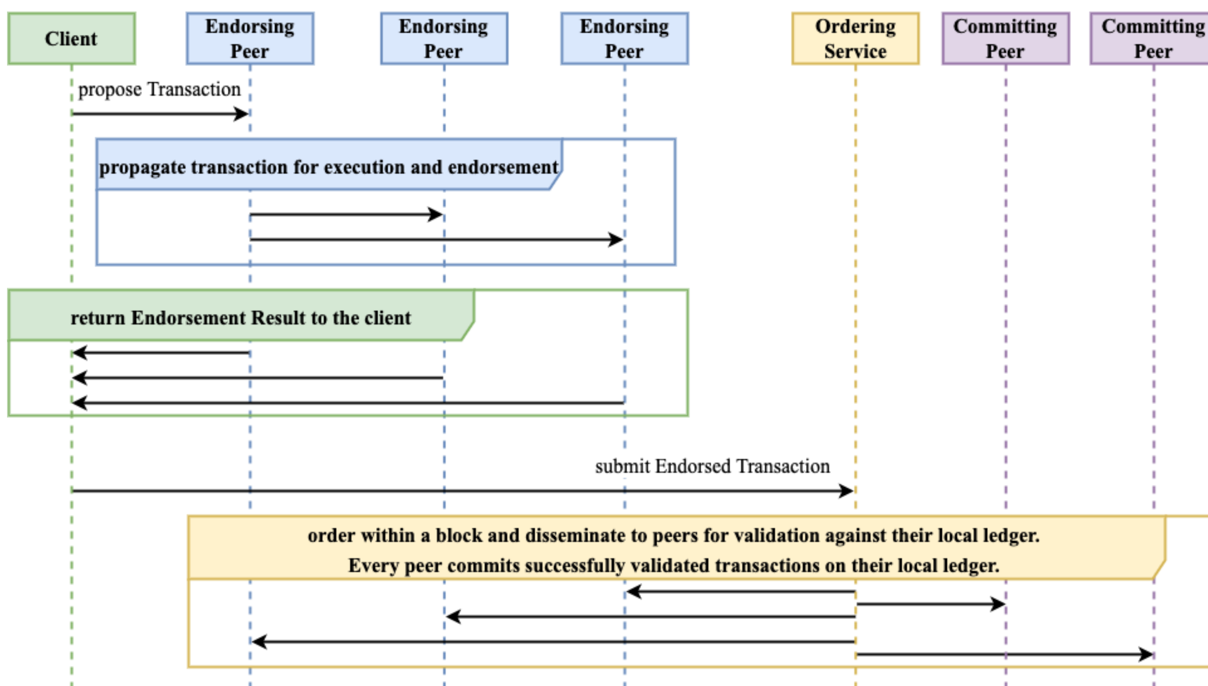


Figure 2.9 A basic transaction flow in Hyperledger Fabric

Transaction Endorsement

Assume a client to be an authenticated and authorised monitoring tool that invokes a smart contract to submit an SLA violation incident. Figure 2.9 abstracts the transaction flow and illustrates the Execute, Order, Validate model employed by HLF. Essentially, smart contract invocations undergo three main stages which are endorsement, ordering, and validation [29] [86] [87]. At the endorsement phase, a smart contract invocation is considered as a proposal to be simulated by a selection of endorsing peers against their local replicas of the state storage. Every endorsing peer execute the proposed transaction to generate a read-write set, which is in the form of $R < k, val, ver >$ while the write set is in the form of $W < k, val', ver' >$. Each endorsing peer sings its generated output and sends it back as a response to the client.

Transaction Ordering

At this stage, successfully endorsed transactions are not committed on the ledger, and the generated read-write set is not reflected on the state storage. The monitoring tool still needs to submit its proposal as a transaction for actual execution. However, it cannot do so unless its proposed transaction meets a specified endorsement policy, say two-thirds of endorsing peers sign the proposal. This will enable the client to proceed to the next phase, where it submits a transaction alongside the generated read-write set and collected endorsements. At the ordering phase, there is a collection of ordering nodes, as shown in Figure 2.7, that take the responsibility of preparing and ordering transactions into blocks. The ordering nodes adhere to a consensus mechanism that regulates how they collaborate to serve their common goal. Since Hyperledger Fabric is based on a modular architecture, we opt to select RAFT [80] as a consensus mechanism, which is a crash-fault tolerance protocol that operates on a leader-followers fashion. The ordering service frequently queues and accumulates a set of pending transactions into a block.

Transaction Validation and Committing to the ledger

A block is considered ready for validation when it satisfies a block batching configuration such as max number of transactions per block, max block size or a timeout. Subsequently, the ordering service submits the block of transactions for validation and committing on the ledger. At this stage, all peers, whether endorsing or committing, validate every transaction by conducting VSCC and MVCC mechanisms. The former is a mechanism conducted by a peer to validate whether a transaction satisfies the endorsement policy. Before committing write sets on the state storage, the latter is a critical validation mechanism. It checks whether the version of the read set is identical to the current record version on the state storage. The transaction journey ends up either being successful or aborted. If successful, peers commit write sets on the state storage; otherwise, the write set is rejected. In all cases, transactions are immutably stored on the blockchain ledger for auditing and transparency purposes.

2.4.3. Performance Metrics

We employ a blockchain benchmarking tool called Hyperledger Caliper[8], which is benchmarking initiative adopted by the Hyperledger Project. This tool follows the specification proposed by Hyperledger Performance Working Group [88], which defines throughput and latency measurements write operations as follows,

$$Avg_{tps} = \frac{\sum_i^n confirmed_{Transaction}}{\sigma} \tag{2.1}$$

$$Avg_{latency} = \frac{\sum_i^n T_d}{\sum_i^n confirmed\ Transaction} \tag{2.2}$$

Note that Avg_{tps} measures the average transactions throughput, which is calculated as the total number of successful transactions divided on σ where $\sigma = lct - fst$ given that fst is the time of sending the first successful transaction whereas lct is the time of committing the last transaction to the blockchain ledger. On the other hand, $Avg_{latency}$ measures the average transaction latency as follows:

1. First, calculate the latency of each transaction $T_d = tx_{commit} - tx_{submit}$, where tx_{submit} is the time of submitting the transaction to the blockchain side while tx_{commit} denotes the time of committing the transaction on the blockchain ledger.

2. Second, calculate the sum latency for all transactions $\sum_i^n T_d$.

3. Finally, divide $\sum_i^n T_d$ on the total number of successful transactions.

Note that these measurements only consider successful transactions and do not account for failed ones. Moreover, they are applicable on *Write* operations, which are transactions that cause a state change to a blockchain asset, such as create, update, or delete. Any write operation is immutably committed into the shared ledger.

In contrast, *read* operations are transactions that query the latest state of a blockchain asset and do not cause a change. Thus, they are not committed to the ledger. Nevertheless, the latency and throughput of a read operation can be calculated in a similar manner to write operations. However, we consider a response time to the client instead of a commit time because read operations do not cause a change to persisted assets.

For both write and read operations, we can calculate the rate of transactions success as per Equation 2.3, while the rate of transaction failure as per Equation 2.4, where T_s denotes the total number of successful transactions and T_s denotes the total number of failed transactions.

$$s_{rate} = \frac{T_s}{T_s + T_f} \times 100 \tag{2.3}$$

$$f_{rate} = \frac{T_f}{T_s + T_f} \times 100 \tag{2.4}$$

[8] https://hyperledger.github.io/caliper

Table 2.2 Tradeoff Comparison: Ethereum versus Hyperledger Fabric.

	Ethereum (Main Network)	Hyperledger Fabric
Consensus type	Mining (PoW)	Crash-Tolerant Protocols (Raft, PBFT, Kafka etc.)
Openness	Permissionless (public)	Permissioned (consortium)
Blocks generation difficulty	Difficult	Relaxed
Fees for transaction execution	Yes	N/A
Fees for storage/memory allocation	Yes	N/A
Latency	High	Acceptable
Throughput	Poor	Acceptable
Commitment	Join or leave anytime	Imposed
Smart contract maintenance	Impossible	Possible based on an endorsement policy
Access Control	Public-key cryptography	PKI (Attribute-based Access Control)
Energy consumption	Poor	Good
Smart contract expressiveness	Reasonable but limited and specific purpose (Solidity)	Rich, well-established, and general purpose (Java, node.js, Golang)
Cryptocurrency	Dependent	Independent
Architecture modularity	N/A	Pluggable components

The following steps elaborate a typical workflow for benchmarking the performance of the smart contract using Hyperledger Caliper:

1. Create an adapter to facilitate connection to and communication with the HLF blockchain network.

2. Deploy smart contract to the blockchain network.

3. Configure the workload artefact such as round scheduling, smart contract invocation, transactions send rate, timeout, and other aspects of interest.

4. Define the transaction construction and the testing logic to be implemented by a set of workers (threads).

5. Generate performance reports in terms of Latency and transaction success rate.

2.5. Considerations for Blockchain Platform Selection

Rather than building a blockchain platform from scratch, a blockchain-based solution would conventionally consider employing a well-established blockchain platform as an underlying infrastructure such as Ethereum or Hyperledger Fabric. While blockchain platforms commonly share intrinsic characteristics, they are distinctive regarding the implementation philosophy.

Accordingly, selecting a blockchain platform constrains the design choices for the overall decentralised solution built on top of it. Moreover, blockchain platforms extend influence to performance and scalability. Table 2.2 illustrates key differences between Hyperledger Fabric and Ethereum. Overall, this thesis selects Hyperledger Fabric as an underlying blockchain platform by considering the following criteria:

Table 2.3 Existing studies on Hyperledger Fabric Performance Analysis

A study by:	HLF Release	Consensus Mechanism
Pongnumkul et al. [91]	0.6	Solo
Dinh et al. [76]	0.6	PBFT
Hao et al. [79]	1.0	PBFT
Thakkar et al. [92]	1.0	Kafka/Zookeeper
Baliga et al. [93]	1.1	Solo
Sukhwani et al. [87]	1.1	Kafka/Zookeeper
Androulaki et al. [29]	1.1	Kafka/Zookeeper
Gorenflo et al. [94]	1.2	Kafka/Zookeeper
Yuan et al. [95]	1.2	Solo
Kuzlu et al. [96]	1.4	Solo
Hang and Kim [97]	1.4.7	SoloRaft
Dreyer et al. [98]	2.0	Kafka/Zookeeper

The Balance between Decentralisation and Scalability

The decentralisation nature of blockchain poses scalability and performance challenges to blockchain platforms. Altarawneh et al. [89] cites and discusses the infamous Blockchain Trilemma, a terminology raised by Ethereum's co-founder Vitalik Buterin. The Blockchain trilemma essentially revolves around the coexistence of three attributes which are decentralisation, scalability and security. However, a blockchain platform may excel at two of them at the cost of the third attribute. Sanka and Cheung[77] also discusses the blockchain trilemma, highlighting that permissioned blockchain networks tend to outperform public blockchain networks in terms of scalability but with sacrificing a level of decentralisation. Altarawneh et al.[89] confirm that public blockchain networks maintain rigour decentralisation but at the cost of scalability.

Performance

Several studies have conducted a performance and scalability analysis on Hyperledger Fabric. To the best of our knowledge, there have been no performance studies on a blockchain-based SLA solution deployed to a Hyperledger Fabric network. This section reviews a selection of studies in other domains, which draw attention to the scalability and performance analysis of Hyperledger Fabric. Table 2.3 presents related works that conduct a performance analysis on Hyperledger Fabric. These studies demonstrate that Hyperledger Fabric performs well for several scenarios under varying workloads and send rates [31].

Dinh et al. [90] benchmark the performance of Hyperledger fabric which they prove to outperform Ethereum in terms of transactions throughput and latency. However, their comparison can be questionable due to the significant difference between both blockchain platforms in terms of permission nature, consensus protocols, level of decentralisation, number of nodes, and so forth. For this reason, we find Pongnumkul et al. [91] experiment both platforms in private networks and intentionally turn off the consensus mechanism in both platforms. Nevertheless, even with this measure, Hyperledger Fabric still outperforms Ethereum.

However, due to the rapid development and improvement on Hyperledger Fabric [95], most of the presented studies in Table 2.3 are either based on obsolete Hyperledger Fabric release versions or deprecated/unapproved consensus mechanisms [98]. That is, the Hyperledger Fabric project, as of writing this thesis, recommends employing a Crash Fault Tolerant (CFT) protocol called Raft [80] as a consensus mechanism. To the best of our knowledge, there has been no performance benchmarking on a Hyperledger Fabric network where Raft is implemented as a consensus mechanism employed by multiple ordering nodes. The work by Hang and Kim [97] is the only work that we find to consider Raft, but only employs one ordering node, which raises a question of why implementing a consensus mechanism at all.

For that, this thesis contributes a performance benchmarking experiment based on Raft between multiple nodes in section 4.6 and section 5.5. Both sections consider the following:

- Raft protocol is the implemented consensus mechanism.
- A realistic deployment that considers multiple organisation nodes and ordering service nodes (refer to Figure 2.7).
- Complexity of smart contract logic.

Section 4.6 experiments Hyperledger Fabric with an earlier release version 1.4.6 deployed on a local machine, which revealed the problem of read-write sets conflict caused by the MVCC protocol. Finally, section 5.5 experiments latest Fabric release version 2.3.2, as of writing our published paper [35], which addresses the problem of read-write sets conflict and experiment on cloud infrastructure. To the best of our knowledge, this thesis contributes the first performance benchmarking in blockchain-based SLA solutions using Hyperledger Fabric.

Immutability Impact on SLA

When contemplating a typical SLA lifecycle, we find that SLA amendment is perfectly expected for SLA renegotiation or error-rectification. Therefore, to which level a blockchain platform extends immutability greatly influences architectural choices.

Ethereum-wise, the immutability feature is not only limited to transactions but also extends to smart contracts [72]. Figure 2.10 roughly depicts logic upgrade of an existing smart contract. Ethereum immutably persists deployed smart contract in the blockchain ledger. Because of the permissionless nature of Ethereum, there is no possible way of upgrading the code logic of the current smart contract. Therefore, if a smart contract needs an upgrade due to renegotiation or maintenance, the current smart contract would be completely or partially abandoned [85]. Consequently, there is no easy way to incorporate new changes unless in a new smart contract, which cannot intuitively access the state storage of the previous smart contract [84]. Subsequently, it hinders smooth maintenance needed for enhancing the logic of related operations such as compliance assessment or penalty enforcement.

Unlike Ethereum, Hyperledger Fabric does not treat smart contracts as immutable assets. Figure 2.11 illustrates the lifecycle of deploying and upgrading smart contracts in Hyperledger

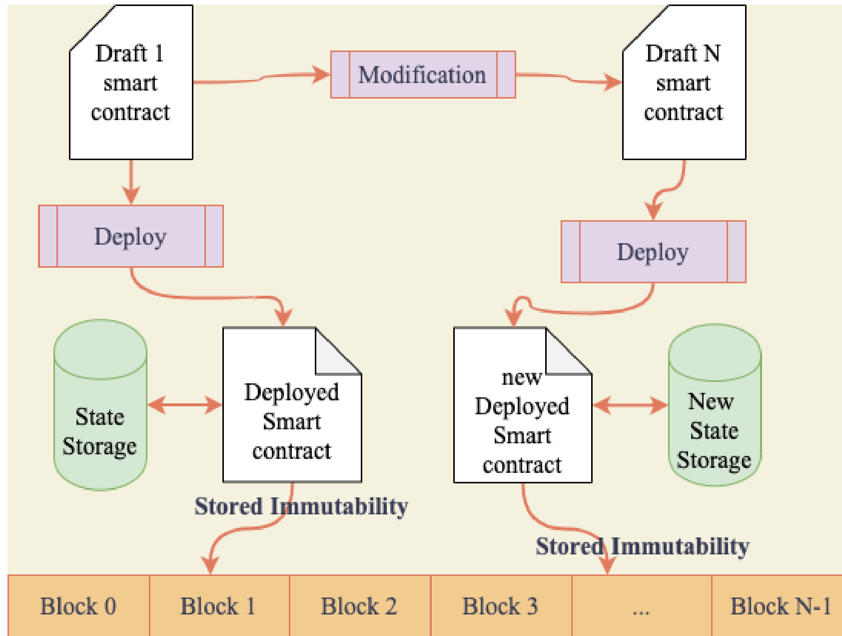


Figure 2.10 Conceptualised illustration of Ethereum approach for Smart contract upgrade cycle process.

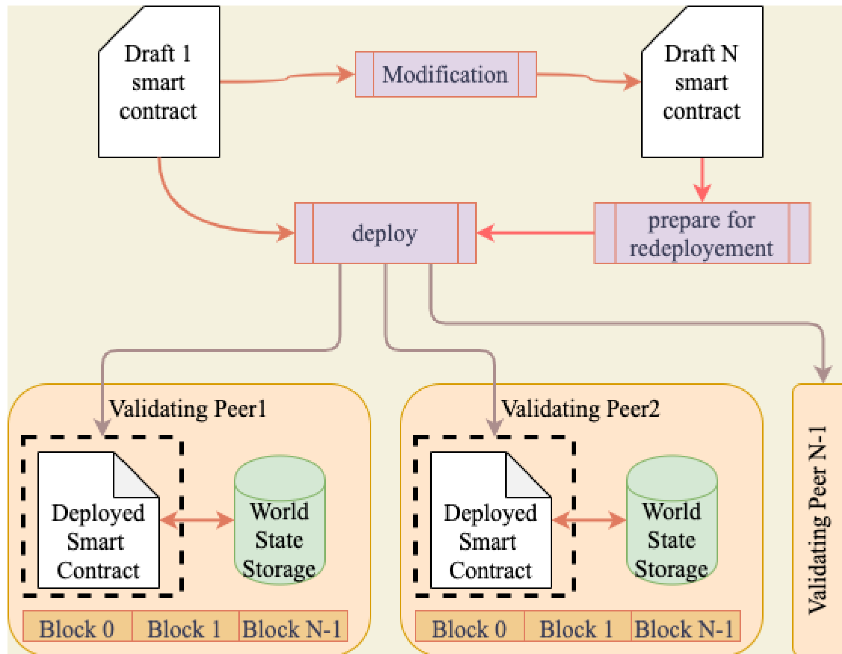


Figure 2.11 Conceptualised illustration of HLF approach for Smart contract upgrade cycle process.

Fabric, which packages them into a Chaincode. Due to the permissioned nature of Hyperledger Fabric, an endorsement policy (a voting system) is in place to govern and regulate the deployment and maintenance of Chaincode. For example, an endorsement policy may require approval by two-thirds of validating nodes to accept new changes on the code logic of deployed smart contracts.

Execution and Storage Cost

Ethereum is a permissionless blockchain platform where any node (peer) can join or leave at their connivance. Therefore, Ethereum introduces a reward scheme to motivate public contribution to computation and storage resources. For that, Ethereum charges clients fees for smart contract execution and storage allocation, which are to be paid to miners [20] [99]. For instance, assume a monitoring service as a blockchain client that alerts a smart contract of an SLA violation incident. Accordingly, the Ethereum network charges the monitoring service for the transaction execution [71]. The cost depends on the logic complexity of the smart contract and the storage needed for processing the transaction [100]. Thus, this thesis argues that the execution cost can pose a challenge to smart contract development, which may lead to sacrificing code quality for cost-saving.

On the other hand, Hyperledger Fabric does not follow the economic model introduced by Ethereum because it relies on a lightweight consensus mechanism that does not require mining by anonymous nodes. Furthermore, Hyperledger Fabric is a consortium of well-known participants who are obligated to contribute their resources to the underlying blockchain network, and infrastructure [29]. For that, Hyperledger Fabric considers the incentive mechanism redundant and unnecessary, as it neither requires fees for transaction execution nor specifies a currency. Subsequently, transactions submitted from monitoring tools to a Hyperledger Fabric-based network do not incur a cost, as would be with Ethereum.

Impact of Blockchain Platforms Selection on SLA-related Operations

The underlying blockchain platform greatly influences SLA-related operations such as monitoring, compliance assessment, and penalty enforcement. For example, consider a scenario where a blockchain-based SLA solution treats a monitoring tool as a client that actively submits transactions which trigger cosponsoring smart contracts based on predefined conditions. For instance, a monitoring tool triggers a deployed smart contract whenever a threshold is reached (i.e. throughput is degrading). On the other hand, smart contracts can be assigned operations that react to received metrics from authorised monitoring tools. For instance, a smart contract can conduct the task of compliance assessment, which examines received metrics and evaluate whether they are per the respective SLA. Based on the compliance assessment, another smart contract validates the outcomes and determines whether to apply associated penalties. Based on this scenario, the selection of an underlying blockchain platform impacts the implemented SLA solution in many ways, including:

- **Rate of Blocks Production** The rate of producing new blocks impacts the scalability of the blockchain platform in terms of queuing and processing transactions received from the monitoring side. For instance, the current block rate of Ethereum does not align well with the nature of monitoring tools, where a considerable number of metrics are generated and need to be handled by deployed smart contracts. As of writing this thesis, the public Ethereum network produces a new block every 15 seconds on average [9]. The block production rate not only degrades throughput (12-15 Transactions per second) but also latency [101]. Meanwhile, Hyperledger Fabric has proven to handle a high rate of transactions while maintaining a reasonable latency, as demonstrated in chapter 5. The modularity of Hyperledger Fabric in terms of selected components and configurations may also enable accomplishing the best possible results.

- **Economic Model:** Ethereum adopts an economic model that incentify anonymous participants to continue contributing resources to the public blockchain infrastructure. A considerable share of the incentive scheme is financed by clients (monitoring tools in our scenario) who pay transaction fees determined by many factors, some of which are the status of the network and the complexity of the logic of invoked smart contracts. The first factor leads to uncertainty because the network status fluctuates rapidly based on a combination of elements such as hash rate, the number of miners, block difficulty, etc. [102]. The second factor depends on the complexity level of the smart contract logic (compliance assessment and enforcement in our case). Thus, developing Ethereum-based smart contracts requires balancing code quality and execution cost, which clients pay for each transaction they submit. In contrast, Hyperledger Fabric alleviates the burden of transaction fees since all resources of the underlying infrastructure are provided by authorised and authenticated participants committed to the blockchain consortium.

- **State Storage:** Ethereum dedicates local storage for each smart contract that is not only financially expensive to use but also of a considerably limited capacity [100]. Therefore, it is not viable to represent and store SLA terms at the storage layer, which explains why most related works in section 2.6 represent SLA terms at the smart contract level. Consequently, legitimate modification of SLA terms is impossible due to the immutability of the smart contract. However, by considering Hyperledger Fabric, we find it fixable in terms of storage capacity and financial cost. Accordingly, it can help represent SLA at the state storage instead of the smart contract, which is more friendly to legitimate modification and maintenance.

- **Programming Languages:** Ethereum introduces a new programming language called *Solidity*, which is still in its infancy compared to other established languages such as those supported by Hyperledger Fabric (Java, JavaScript, and GoLang) in terms of reliability, expressiveness, supported features, available expertise. Therefore, Hyperledger Fabric is a

[9] https://etherscan.io/chart/blocktime

better choice for encoding quality smart contracts that represent SLA within the blockchain and conducting compliance assessment, penalty enforcement and billing logic.

2.6. Related Blockchain-based SLA Works

Recently, the literature started to observe the relevance of blockchain to the contractual processes. One can trace the exploration of blockchain in this regard back to the work by Weber et al. 2016 [103], which investigated the viability of the blockchain for contractual processes. Although it is not specifically tailored towards SLA, it mainly leverages the ledger immutability by recording the shared history of the choreography processes. Furthermore, it recognises trust issues in collaborative environments and questions which contractual party to trust on distrusted processes. In response, it devises a translator tool that converts the specification of collaborative processes into a Solidity-written smart contract deployable to the Ethereum network. However, while the author acknowledges the potential of blockchain, they observe that Ethereum is not suitable for high throughput applications.

The short paper by Di Pascale et al. 2017 [104] is one of early related works in the literature to recognise the potential of blockchain-based smart contracts for the SLA practice. This work is motivated by trust issues associated with the current SLA practice in the context of telecommunication. It briefly lists a set of advantages that smart contracts, which may revolutionise SLA immutability, enforcement, negotiation and payment. On the other hand, it also envisions some challenges associated with the decentralised nature of smart contracts. For instance, it questions how well smart contracts can align with the legal system. Additionally, the permissionless nature of some blockchain networks does not enable jurisdiction authorities to enforce legal actions. Moreover, the authors attribute the difficulty of rectifying errors to the immutability of smart contracts. While their short paper is insightful, it does not provide further elaboration or empirical experiments.

The work by Nakashima and Aoyama 2017 [105] propose a decentralised SLA approach for SLA specification in the context of web APIs. The authors find an opportunity in exploiting Ethereum for automating and orchestrating some distrusted processes in a decentralised fashion, such as SLA definition, billing and termination. The authors question the trust given to service providers in the current SLA definition and negotiation practice. They present a tool for generating a machine-readable SLA document based on the RDF (Resource Description Framework). However, they do not store the generated SLA within blockchain due to cost and storage limitations associated with Ethereum. Instead, they use the smart contract for validating the integrity of externally hosted SLA documents. This is done by persisting the digest hashes of SLA into the blockchain ledger. The authors emphasise the role of monitoring in determining SLA compliance status. Nevertheless, they do not specify how the monitoring mechanism functions and in what way they incorporate it in their proposed solution. Furthermore, being Ethereum employed as the underlying blockchain technology, the authors point out that contract execution fees hinder an effective blockchain-based SLA solution.

The short paper by Neidhardt et al. 2018 [22] discusses trust mechanisms that depend on third parties, such as external verifiers. They point out that such trust schemes may fail because trust is merely shifted from a centralised authority to a third party. Instead, the authors suggest placing trust in Ethereum for billing purposes in the context of cloud services. They highlight the challenge of trust issues related to external and anonymous monitoring services feeds. Hence, they suggest the use of a TLS-based oracle protocol [106] to verify the integrity of monitoring logs. Moreover, they point out that the oracles technology enable a distributed network of monitoring entities where their logs are averaged and reported to the blockchain side. While their presented approach is interesting, it does not provide sufficient details on the implementation of their approach in terms of monitoring and billing. Additionally, it seems to only focus on the service provider's availability. Once the service provider is down, the smart contract instantly compensates all consumers simultaneously. Therefore, it does not scale to the complexity level of SLA in terms of quality requirement specifications and calculation of violation rate. However, The authors concluded that Ethereum introduces the problem of cost estimation uncertainty of smart contract execution.

Zhou et al. 2018 [99] believe that the traditional SLA practice does not enable monitoring services to automatically enforce SLA agreements and compensate consumers for SLA violations. They agree that the current practice assumes trust in service providers in common. They criticise the authority granted to service providers in terms of violation claims verification and the decision of whether to compensate consumers for SLA violations. They also find that consumers may struggle in proving an SLA violation due to the lack of visibility on the service. Therefore, they suggest offloading the processing of SLA violation to an Ethereum-based smart contract from any involved party. They propose a distributed monitoring mechanism based on a group of selected witnesses, which observe the service provider's compliance with the SLA. However, their work is more focused on the honesty of the monitoring service, and it is not clear how they represent SLA within the blockchain. Moreover, it remains a question whether any SLA party is willing to sacrifice their data privacy to an anonymous monitoring service in a permissionless environment such as Ethereum.

Uriarte et al. 2018 [14] propose a blockchain-based SLA approach in the context of cloud services. They share in common the view that the current practice assumes trust on SLA management and mechanism which are dependent on the service provider or third party. The main motive behind their work is to exploit blockchain-based smart contracts for addressing trust issues in their previous work on SLA specification tool namely; SLAC framework [107]. Their work employs the traditional SLAC for SLA specification and negotiation. However, the authors believe that smart contracts should encode the agreed quality requirements so that SLA can benefit from blockchain features. Accordingly, they developed a utility to automate SLA translation into a Solidity smart contract. They also shift enforcement and billing to a blockchain environment using Ethereum. Their work focuses on describing the transformation from SLAC into smart contracts. Moreover, they elaborate on their perspective on the influence of blockchain-

based smart contracts on a typical SLA lifecycle. The SLA definition and negotiation in their work is conducted off-chain, understandably due to the immutability of the smart contract.

Scheid et al. 2019 [20] contemplate the current practice regarding the possibility of dishonest act by SLA parties whether it is to maximise profit or mitigate loss. They perceive blockchain-based solution as a better alternative for Trusted Third Parties (TTP) because it can proactively improve dispute settlement due to autonomy and decentralisation of smart contract. They specifically propose a Ethereum-based compensation system which depends on an trusted monitoring service. They implement a smart contract that encodes a quality requirement and a compensation condition. They observe that smart contract cannot self-execute and it must be triggered by the trusted monitoring system. They reveal that Ethereum is limited in terms of the storage size and maximum amount allowed for smart contract execution. Nevertheless, they leave SLA definition and cost analyses to a future work.

2.6.1. The Convergence between Blockchain and Monitoring

Smart contracts are supposed to be terminable and deterministic [31]. Subsequently, smart contracts are not optimal for conducting endless activities such as monitoring. Thus, an external monitoring service has to be in place to help smart contracts form a decision on the compliance level of obligated providers. The monitoring service is a client of the blockchain-based solution in our case, and it is tasked to report incidents (submit transactions) to smart contracts for compliance assessment and incident processing.

Due to the reliance of smart contracts on an external monitoring mechanism, the latter can pose a single point of failure to the entire solution [22]. For example, monitoring tools can be untrusted or faulty for various reasons. It is worth considering the concept of decentralised oracles, such as Chainlink[10] and Provable[11] for validating feeds from external entity and to ensure deterministic smart contract execution [106] [108]. Not to be confused with the oracle company, the oracle terminology here refers to a decentralised bridge between the external world and the blockchain side. Recently, related works have started to explore utilising the oracles technology for decentralising the monitoring service such as [22][14][109].

Uriarte et al. 2021 [110] criticise the level of decentralisation of existing oracle providers and claim that they are susceptible to a single point of failure and trust issues. Thus, they contribute a decentralised SLA monitoring approach. A fundamental improvement in their work is adopting a permissioned blockchain platform for monitoring purposes, namely Hyperledger Sawtooth, which is based on a consensus protocol called Proof of Elapsed Time (PoET) used to establish a common truth among involved monitoring entities. Nevertheless, they acknowledge the possibility of dishonest monitoring by some participating monitoring entities. Subsequently, they suggest a reputation-based system that they leave for future work. Zhou et al. 2018 [99] address the potentiality of misconduct and colliding between distributed monitoring agents. The authors

[10] https://chain.link/
[11] https://provable.xyz/

propose an Ethereum-based approach that employs the Nash Equilibrium of Game Theory for rewarding and punching monitoring entities depending on their conduct and behaviour.

Notwithstanding, it remains a question whether any SLA party is willing to sacrifice their data privacy to an anonymous monitoring service in a permissionless environment such as Ethereum. Moreover, Ethereum is associated with the difficulty of estimating transactions execution cost in the long run [102]. Monitoring tools trigger smart contracts by submitting a transaction; thus, it is a question of who should incur the cost associated with the decentralised monitoring service regarding transaction execution and oracle services? Furthermore, one must consider the permissionless nature of Ethereum, which enables monitoring entities to join or leave as they please, making it difficult to guarantee the stability of the monitoring service.

This thesis leverages the permissioned nature of Hyperledger Fabric, where every participant is known and that they can choose to communicate in private channels or network-wide. In addition, this thesis assumes the PKI mechanism provided by Hyperledger Fabric is sufficient for ensuring the data integrity and confidentiality among the network participants. Accordingly, it assumes an authenticated, trusted, and committed monitoring service that feeds smart contracts with data relevant to the SLA compliance assessment.

2.7. Advantages and Disadvantages of Blockchain-based SLA Solutions

As discussed in section 2.2.7, traditional SLA solutions are typically centralised. Thus all SLA-related matters (i.e. compliance assessment and penalty enforcement) rely on either the service provider or a third party. Consequently, trust is essential for establishing the contractual relationship between involved parties. That is, service consumers must have good faith in either the service provider or the third party while accepting a risk associated with the possibility of misconduct or corruption. Moreover, traditional solutions are usually linked with opacity, particularly internal processes. Therefore, consumers must assume goodwill and calculate the risk associated with the trust given to the operator of the SLA management system. On the other hand, service providers also assume trust in their clients. However, consumers may falsify evidence to maximise profit or reduce financial loss.

For such reasons, Blockchain-based SLA solutions emerged to provide a decentralised alternative which assumes no trust in any involved parties. Section 2.6 overviews related works that aim to leverage blockchain features for enabling decentralisation, transparency, traceability, and immutability, which help mitigate the need for blind trust and minimise potential disputes. The concept of smart contracts also allows the automation of several SLA-related operations in a non-repudiable fashion without total reliance on service providers or third parties. However, most related works are influenced by the philosophy of Ethereum regarding the blockchain's implementation. Section 2.5 analysis and discusses in depth various aspects of why this thesis disregards Ethereum and considers it impractical blockchain infrastructure for SLA purposes. Alternatively, this thesis selects permissioned blockchain platforms for addressing the limitation of related studies concerning scalability, terms expressiveness, maintenance upgradability, and rectifiability. This thesis also addresses the usage of blockchain for SLA

Table 2.4 key differences between the proposed blockchain-based SLA solution and related works

	Non-Blockchain Approaches	Public Blockchain Approaches	Permissioned Blockchain Approaches
Responsibility of SLA-related Operations	Provider-dependent or third-party	Smart contracts	Smart contracts
Source of Truth	Provider-dependent or third-party	Immutable ledger	Immutable ledger
Infrastructure Ownership	Provider-dependent or third-party	Public	Consortium
Centralisation	Centralised	Decentralised	Decentralised
Transaction Processing	Fast	Slow	Moderate
Maintenance	Possible	Difficult	Possible
Tamper-proof	No	Yes	Yes
Relevance to IoT	Rarely Discussed	Rarely Discussed	IoT-domain Specific

in the context of IoT. Other related works have not been thoroughly examined since most of them are invested in other domains, such as cloud and telecommunication. This thesis examined two main IoT scenarios: telemedicine and IoT-based firefighting. This thesis experiments with distinctive IoT infrastructure deployments and communication protocols in each scenario. Moreover, this thesis proposes and empirically demonstrates a set of approaches regarding SLA representation over blockchain, monitoring, violation identification, compliance assessment, and penalty enforcement.

In summary, Table 2.4 presents key differences between the proposed approach and other counterparts, whether they are blockchain-based non-blockchain studies. Regarding the responsibility of SLA-related operations (i.e. compliance assessment and penalty enforcement), non-blockchain solutions assume that either the service provider or a third-party entity holds the ultimate authority. Therefore, whoever is granted such authority would subsequently form an absolute source of truth, no matter their credibility. In contrast, blockchain-based solutions typically detach the responsibility of critical SLA-related operations from any entity. Instead, they are assigned to non-repudiable smart contracts that operate beyond the direct control of any single authority. Consequently, no involved entity will any longer form the ultimate source of truth. Instead, every involved party shares in common a replica of an immutable ledger that records all relevant activities, thus enhancing transparency and tractability while mitigating dispute possibility. In terms of infrastructure ownership, non-blockchain solutions generally assume the service provider and/or third-party entities to, partially or totally, provide and operate the necessary infrastructure (hardware, network, etc.) for SLA management systems. The ownership of underlying infrastructure reflects a degree of centralisation over the SLA management system. It is a fact that centralisation can be useful for maintenance purposes, such as the need to rectify an error, upgrade SLA terms, and software updates. Chapter 3 discusses in-depth the need for rectifiability and upgradability. Nevertheless, centralisation requires trust and willingness to accept the risk of manipulation or misconduct. Trust is usually built based on creditability checks such as reputations and feedback systems, reviews, trustworthiness assessments, etc., before engagement in the contractual relationship.

In response, blockchain-based solutions decentralise the infrastructure ownership to prevent total authority on the SLA management solution. However, the overwhelming majority of existing blockchain-based SLA solutions rely on public infrastructure that, by design, separates smart contract development and deployment from the ownership and operation of the provided infrastructure. That is, the infrastructure of public blockchain networks is mainly contributed to

and operated by anonymous participants. The aftermath is that no one can claim the authority of either the smart contract or the underlying infrastructure. Whereas such a rigour philosophy can eliminate misbehaviour and misconduct usually associated with centralised solutions, it hinders the ability to smoothly and effectively maintain the overall solution. For example, rectifying an error discovered in the smart contract or modifying its logic after deploying the smart contract to the public blockchain network is difficult. Additionally, public blockchain platforms aim to achieve a high degree of decentralisation but at the cost of scalability. Accordingly, SLA solutions that depend on a public blockchain network would be limited in terms of transaction processing. For instance, the number of violation incidents that such solutions can handle is capped by the low number of maximum transactions processed by such networks (as of writing this thesis, Ethereum can process on average only 12-15 transactions per second [101]).

Given the factors in section 2.5, this thesis selects Hyperledger Fabric as the underlying blockchain infrastructure, which has proven to strike a fair balance between decentralisation and scalability. As chapter 5 demonstrates, the deployed solution can accomplish a decent number of transactions per second, though not as scalable as would be the case in centralised solutions. Furthermore, Hyperledger Fabric can preserve a proper degree of decentralisation because the infrastructure is contributed and operated by a selection of known participants in which none can have the entire network's absolute authority. While these entities may have conflicting interests (e.g. service provider vs consumers), reaching each other and collaborating on a common goal (i.e. maintenance and upgrade) is still manageable. This thesis not only concerns the impact of the underlying blockchain platform but also approaches SLA management differently by improving the limitations of related works as per summarised in section 3.6.4. Finally, most existing blockchain or non-blockchain studies draw attention to SLA monitoring, compliance assessment and penalty enforcement in the context domains other than IoT, such as cloud and telecommunication. For that, it contributes to the literature on SLA and IoT by proposing and experimenting with a blockchain-based SLA approach using two distinctive IoT scenarios in the context of IoT (see Table 1.1).

2.8. List of Assumptions

This thesis considers Hyperledger Fabric for proposing and experimenting with the proposed blockchain-based SLA management solution in the context of IoT. Concerning the underlying blockchain network, this study assumes a consortium of validating peers that are well-known in advance, authenticated and authorised, and relevant to the SLA agreement in place. This thesis does not impose a hard requirement on who should contribute to the underlying blockchain infrastructure and participate in the validation process as long as the blockchain network is immune from being controlled by a single authority. For example, the blockchain network may include but is not limited to, the service provider, the service consumer, auditors, assessors, governmental bodies, and enforcement authorities. The wider the diversity of involved entities, the more decentralisation and less prone to manipulative actions and misconduct associated with

total control or collusion. This thesis does not assume trust in any participating entity because blockchain networks are designed to be decentralised and tolerant of failures.

In this thesis, chapter 4 and chapter 5 experiment with a blockchain network that is equally shared by the service provider and the service consumer. Optionally, other entities may join the network as agreed by SLA parties or as imposed by regulations. This study transforms SLA compliance assessment, enforcement penalty, and billing into smart contracts that operate within a blockchain environment. Thanks to the decentralisation nature of the blockchain network, these activities can operate autonomously beyond the full control of any single validating peer. However, this thesis assumes a trusted monitoring tool that acts as an honest client to the blockchain network. Section 2.6.1 further discusses why smart contracts are not optimal for conducting ceaseless monitoring tasks. Therefore, this thesis regards the monitoring activity as a trusted aid tool that feeds smart contracts with necessary data for conducting their logic. Accordingly, it leaves The decentralisation of the monitoring activity itself and the level of trust placed on it to be an open research problem for future study.

As regards the hardware requirement per each validating peer, Hyperledger Fabric does not specify a recommendation of minimum hardware capacity per each validating peer. Nevertheless, the quantity and quality of available resources influence the overall performance of the blockchain network, which includes, but is not limited to, CPU, memory, network, and storage capacity. This thesis experiments with distinctive blockchain resources and deployment models as per chapter 4 and chapter 5 and specified in table 4.4 and table 5.2; respectively. The former experiments with a blockchain network deployment over a local machine of limited hardware resources, whereas the latter experiments over a more capable cloud infrastructure. These two distinctive deployments show that Hyperledger Fabric can operate on limited resources. However, this thesis assumes that involved SLA parties and concerned regulatory entities would also consider and discuss the proper quantity and quality of hardware resources for deploying the blockchain network.

Chapter 3. SLA Representation and Awareness within Blockchain

Summary

Representing SLA within blockchain is important for providing smart contracts with the necessary SLA awareness for conducting SLA-related tasks. This chapter delves into this matter and organises its sections as follows: First, section 3.2 overviews and categorises SLA representation in related works. Then, section 3.3 suggests a set of principles, abbreviated as IRAFUTAL, that aims to address issues found in existing blockchain-based SLA solutions. Section 3.4 takes advantage of a formal IoT-based SLA specification tool to model SLA at the blockchain's state storage level. Section 3.5 demonstrates the advantage of the proposed SLA representation approach by implementing an SLA manager in the form of a smart contract that elastically operates SLA within the blockchain. Finally, section 3.6 evaluates the proposed SLA representation approach by extending the SLA manager to serve the purposes of SLA definition and negotiation. It aim to demonstrate how the proposed SLA approach can mitigate the drawbacks of the existing SLA representation approach and what potentials it can promise for the practice of blockchain-based SLA.

3.1. Introduction

Smart contracts need to maintain SLA awareness to facilitate the automation of related tasks such as SLA negotiation, compliance assessment, terms enforcement, billing and termination. To appreciate the significance of SLA representation within blockchain, consider a decentralised smart contract that assesses a service provider's compliance with an established SLA. However, it must be aware of relevant SLA content (i.e. quality requirements promised by the service provider). For the sake of argument, let the SLA be externally stored on a centralised server. We argue this practice can negatively impact the effectiveness of the decentralised compliance assessment in two main ways. First, one can question the degree of trust that should be given to the external host considering that it can be susceptible to a single point of failures such as a malicious act or unavailability [111] [112]. Second, SLA is customarily expected to evolve due to a legitimate renegotiation or error-rectification. Given the replication of smart contract execution, there is the probability that blockchain validators obtain inconsistent versions of the SLA [14]. For instance, a set of validators receive the most recent SLA version while others happen to receive a cached version of the previous SLA content. Subsequently, blockchain validators execute the same smart contract but may produce various outputs and thus hindering consensus among them.

Therefore, SLA representation within the blockchain has several advantages as follows:

- SLA content can be immune from malicious behaviour or unavailability.
- Smart contracts can maintain the necessary awareness of the SLA structure and content.
- Smart contracts are relieved from obtaining SLA content from external hosts.
- Validators can execute smart contracts in a deterministic manner [110]; meaning that the same input to the smart contract must always produce the same output.
- Validators can reach a consensus on the validity and finality of smart contract execution [75].

By examining related works that represent SLA within blockchain, in sections 3.2.2.2 and 3.2.2.3, we find that they conventionally represent it within blockchain by encoding its structure and content directly into the logic of the smart contract. Given the tight coupling of SLA content with the smart contract, the SLA, by default, inherits the immutability of the smart contract. However, this may not align well with SLA evolution purposes; a practice that is normally expected during a typical SLA lifecycle, specifically for the purpose of SLA definition, error-rectification and renegotiation [10]. Additionally, the process of encoding SLA as a smart contract and deploying it to compatible blockchain platforms is not user-friendly and only possible for subject-matter experts (i.e. developers, operators, security auditing, etc.) [72].

This chapter proposes an SLA representation and awareness approach that avoids issues found in existing approaches. It implements the proposed approach and demonstrates how it enables elasticity and ease of use needed for SLA definition, renegotiation, and error rectification.

3.2. SLA Representation in Related Works

The awareness of SLA content is critical for any automated task such as monitoring, compliance assessment, penalty enforcement, billing, and other related SLA management [113] [2]. The quest of this section is to investigate how blockchain-based SLA solutions represent SLA and to what level of SLA awareness their decentralised approaches may achieve. By investigating the related studies, we can categorise different SLA representation approaches found in related works as follows:

3.2.1. SLA-agnostic approaches

This category describes a blockchain-based solution as an SLA-agnostic when it may resolve some trust issues related to the SLA practice but omits to represent fundamental SLA properties within blockchain such as involved parties, quality requirements, violation consequences and so forth. As a result, it can be difficult or impractical to attain the SLA awareness level needed for conducting SLA-related tasks [16]. Consequently, smart contracts cannot automate and conduct tasks such as SLA establishment, monitoring, compliance assessment and penalty enforcement. This section diverges this class of approaches further into two sub-categories which are:

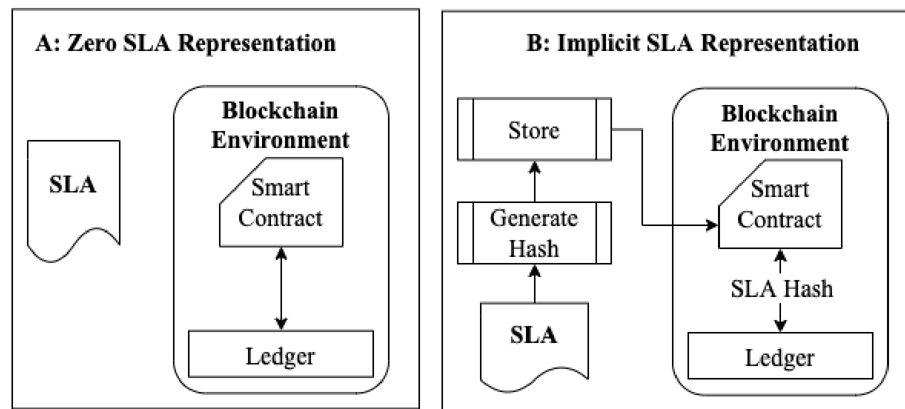


Figure 3.1 Examples of SLA-agnostic approaches

3.2.1.1. Zero Representation within Blockchain

As in Figure 3.1.A, smart contracts may serve some SLA purposes but do not represent SLA structure or content. For instance, Wonjiga et al. [114] assume an SLA where a cloud provider guarantees the integrity of consumer data. While they use blockchain to achieve this SLA objective, their work does not seem to represent SLA structure or content within the blockchain. Similarly, Singi et al. [115] use blockchain to enforce SLA compliance regarding software-related licences and security policies. However, their work only represents the subject of the SLA, which are the licences and politicises, whereas the SLA itself does not seem to take place within the blockchain.

3.2.1.2. Implicit SLA Representation

As in Figure 3.1.B, SLA can have an implicit form of presence within the blockchain environment. Albeit, smart contracts still cannot fully reason about SLA structure or content. For instance, Nakashima and Aoyama [105] use blockchain for SLA integrity verification purposes. For that, their work composes SLAs in the form of an RDF-formatted document (Resource Description Framework) and then generates a hash of it to be deposited by a smart contract into the blockchain. This approach can help reveal and invalidate any unauthorised modification, which can be realised by comparing the immutable hash of the SLA stored in the blockchain with the actual SLA residing in the external world [112]. Nevertheless, smart contracts cannot use the locally stored hash to reason the SLA content. However, smart contracts can use them to verify the integrity of the externally-stored SLA.

3.2.2. SLA-aware Approaches

This category deems blockchain-based solutions to be SLA-aware when smart contracts serve SLA purposes while maintaining sufficient awareness of SLA structure and content. As a result, smart contracts can conduct and automate SLA-related tasks in a truly decentralised manner, examples of which include, but are not limited to, SLA negotiation, compliance assessment, enforcement, billing, termination and so forth. Most of the existing Blockchain-based SLA

studies explicitly encode SLA structure and content (parties, quality requirements, penalties, etc.) in the smart contract. Following are commonly used methods for stating SLA in smart contracts:

3.2.2.1. Deployment to Compatible Decentralised Storage

As in Figure 3.2.A, some blockchain-based SLA approaches take advantage of compatible decentralised storage systems to resolve some issues related to SLA central hosting, such as unavailability of the hosting server or misconduct by a central authority. Therefore, enabling smart contracts to make use of externally-hosted SLA while maintaining immutability and consistency. For instance, Kapsoulis et al.[116] propose a tool that adopts that ISO 19086-2 SLA standard for producing a JSON-formatted SLA. Then, the generated SLA document is deployed into decentralised storage, namely, Interplanetary File System (IPFS) [117], which can be accessed by smart contracts and monitoring tools. While hosting SLA externally in decentralised storage systems is appealing in terms of immutability, consistency and availability, such scheme introduces unnecessary architecture complexity in terms of infrastructure, networking, cost, execution fees, and maintenance. Additionally, the stored SLA cannot be modified, which prevents SLA renegotiation and error-rectification. Moreover, if smart contracts sought SLA awareness by calling an externally hosted SLA, it would violate the transaction flow in most blockchain platforms [31].

3.2.2.2. Manual Smart Contract Development

SLA representation within the blockchain network can present an alternative approach for mitigating the complexity of external decentralised storage systems. As Figure 3.2.B illustrates, some existing approaches leverage smart contracts for expressing SLA structure and content (i.e. a quality requirement $latency \leq 3ms$). Consequently, smart contracts can maintain the necessary SLA awareness without needing to query the external world, either centralised or decentralised storage systems. Therefore, smart contracts can use SLA awareness to conduct and automate actionable procedures related to the expressed SLA. For instance, when a smart contract receives a breach to the stated quality requirement, it can have the necessary awareness of relevant consequences (i.e. penalities) to be enforced on the service provider. For instance, Scheid et al. [20] encode SLA terms directly into the logic of the smart contract, which enables conducting SLA management tasks related to penalty enforcement and SLA termination. This approach benefits from blockchain features such as immutability, consistency, decentralisation, high availability and so forth. However, the process of representing SLA content in the form of a smart contract requires a subject-matter expert in terms of smart contract development or deployment to the Blockchain.

3.2.2.3. Automated Smart Contract Generation

Expressing an SLA in the form of a smart contract is difficult for non-expert users. There have been some efforts to realise an automated smart contract generation mechanism. For instance,

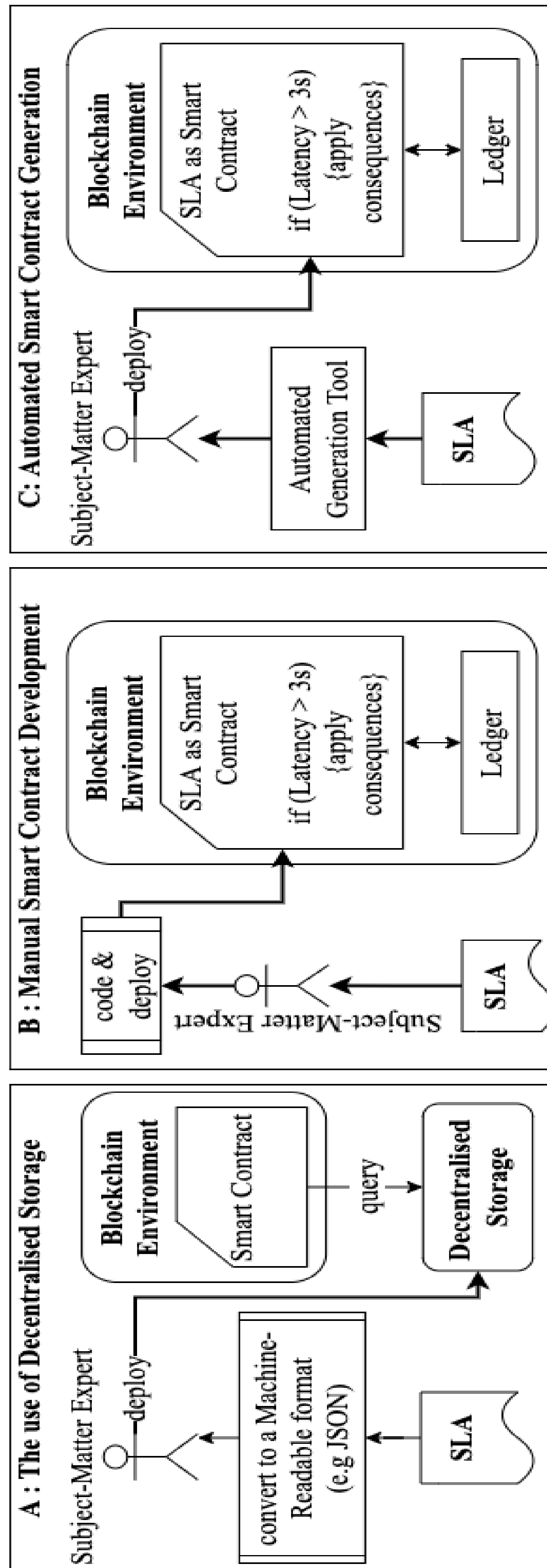


Figure 3.2 Types of SLA-aware approaches

Weber et al. [103] devise a translator tool that accepts agreements in the form of a Business Process Model and Notation (BPMN). It can recognise some critical properties of a BPMN document expressed in an Extensible Markup Language (XML) format. The translator tool transforms the file into an actionable smart contract written in Solidity programming language and deployable to the Ethereum network. Uriarte et al. [110] use an off-chain framework translator tool called SLA2C, proposed in their previous work [14], which transforms SLA into a solidity smart contract. However, the SLA2C framework can only recognise their formal language called "SLAC", previously proposed in [107]. The work by Kochovski et al. [118] provides a graphical user interface (GUI) to enable defining quality requirements. Accordingly, it generates a Solidity smart contract populated with user inputs and deploys it to Ethereum Virtual Machine (EVM)-compatible blockchain networks.

As seen in these example studies [103], and [110], their translation tools require a machine-readable format to generate Solidity smart contracts deployable to the Ethereum network. The most user-friendly translation tool is the one proposed by [118], which provide a GUI for the end-user. Nevertheless, as presented in Figure 3.2.C, generated smart contracts still need subject-expert matters for executing the deployment stage.

3.2.3. SLA Negotiation in Related Works

As discussed above, conventional SLA representation tend to encode SLA structure and content directly into the smart contract to achieve SLA awareness needed by relevant tasks such as monitoring, enforcement, billing and so forth. Consider the fact that smart contracts are immutable and cannot be amended [84], which hinders on-chain SLA negotiation either before or after SLA establishment. Figure 3.3.A illustrates a typical lifecycle for SLA negotiation in conventional approaches, where SLA content and structure are both encoded directly in the smart contract. Therefore, SLA negotiation can only be manually conducted off-chain. Thus, there is always the need for the development and deployment of a new smart contract that accommodates new changes. This is evident in many existing blockchain-based SLA approaches. For instance, Scheid et al.[20] state in their conclusion that their approach cannot address SLA negotiation because it needs human intervention, particularly in terms of smart contract development and deployment; thus, leaving it to future work. Uriarte et al.[110] also recognise the need for conducting SLA negotiation off-chain before encoding SLA in the form of a smart contract and deploying it to the blockchain side. Zhou et al.[99] also have a similar approach in terms of manual SLA negotiation and deployment in the form of a smart contract to the blockchain side. To the best of our knowledge, there has not been to date any relevant work that addresses SLA negotiation within the blockchain. A better alternative is as shown in Figure 3.3.B, where this process does not require off-chain negotiation and subject-matter experts for blockchain-related matters such as smart contract development and deployment; which is demonstrated in Section 3.6.3.

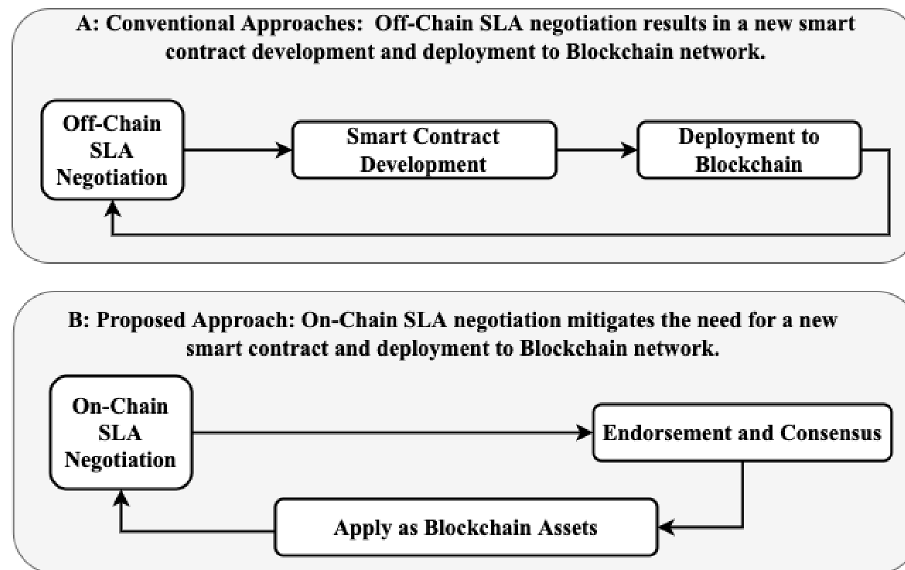


Figure 3.3 SLA negotiation lifecycle in conventional and proposed SLA representation approaches

3.3. IRAFUTAL: Proposed Principles for SLA Representation

By examining the advantages and disadvantages of existing approaches for SLA representation within blockchain, this section formulates a set of principles to be followed by the proposed SLA representation approach. This chapter collectively refers to them as *IRAFUTAL*, an abbreviation that combines the first letter of each principle. The set of IRAFUTAL principles are as follows:

Identifiable and Reusable The SLA representation should enable SLA composition based on independent, unique, and identifiable SLA components including, but not limited to, involved SLA participants, quality requirements, violation consequences, an example of which is presented in Figure 2.2. A monolithic SLA serving a specific purpose that is of no use elsewhere. On the other hand, reusable components can serve a variety of SLAs for various purposes. For instance, the same SLA participant can be reused or referred to in multiple SLAs or using the exact quality requirement in other SLAs or different smart contracts. This practice encourages linkability which facilitates fact-gathering about each SLA property. For instance, consider various agreements associated with an SLA component representing a service provider. Therefore, one can query what existing SLA this particular provider engages with or what quality requirements are under its responsibility. Such a practice also mitigates redundancy caused by restating SLA components in various places, either in the same SLA or elsewhere. This practice also mitigates the complexity of smart contract development and facilitates the maintenance of the overall blockchain-based SLA solutions. For instance, it would be less prone to human error and easier to identify the root cause of a problem if SLA components are defined once and reused multiple times. Therefore, rectifying an error or applying a remedy on one component would also apply elsewhere by default.

Accessible by smart contracts Smart contracts, which represent SLA-related tasks, must have the ability to internally access SLA properties within a blockchain environment. This is in order to provide a smart contract with necessary SLA awareness while mitigating the following:

- The need for trusting central or third party for hosting the SLA.
- Unnecessary complex architecture due to SLA hosting in external decentralised storage.

Flexible No one SLA template fits all SLA-related scenarios. Moreover, SLA templates can impose boundaries that can arbitrarily constrain blockchain-based SLA solutions, thus limiting solution creativity and viability of blockchain as an underlying solution. Therefore, the SLA representation within blockchain should refrain from imposing a specific SLA template. Alternatively, SLA representation should encourage a modular specification of SLA structure and properties. Otherwise, a set of straightforward implications could materialise as in the following:

- Null values because of some irrelevant template properties.
- The need to circumvent imposed SLA properties which could complicate smart contract development.
- Waste of computation or storage capacity.

User-friendly As can be seen in Figure 3.2, most existing solutions require subject-matter experts to represent SLA in the form of smart contracts. However, this can hinder smooth SLA definition and negotiation, which does not align well with a typical SLA lifecycle. Therefore, SLA representation within blockchain should consider mitigating the need for subject-matter experts, where end-users can define SLA content independently (i.e. via a Graphical User Interface).

Tamper-proof SLA acts as a source of truth for relevant affairs such as monitoring, compliance, penalty enforcement, dispute resolution and so forth. Therefore, SLA must be immune from unintended modification and malicious acts. Otherwise, the outcomes of related tasks are useless, even if they operate in a smart contract running on a blockchain environment. SLA can benefit from several blockchain features such as immutability, auditability, decentralisation, consensus mechanism, and resistance to the single point of failure.

Amendable The SLA representation approach should anticipate modification that normally occurs within a typical SLA lifecycle. For example, consider an established SLA which states $Latency < 3s$. For any reason, such as renegotiation or error rectification, this SLA clause may have to be changed or deprecated. Assume this clause is explicitly stated within a smart contract, as shown in Figure 3.2. Given the immutability of smart contracts, such an amendment is impossible [84][31]. Alternatively, there would be the need for creating a new smart contract or applying workarounds, examples of which are covered by Marino and Juels [119] as well

as Wöhrer and Zdun [85] which are applicable to Ethereum. Even in the case of Hyperledger Fabric, such a change to the smart contract requires a Chaincode upgrade, which is a constrained and governed process that does not only call for endorsement by validator nodes but also is conducted by developers and network operators [120]. Therefore, the challenge is to enable SLA modification during smart contract runtime in a decentralised manner.

Loosely-coupled SLA awareness is central for SLA management tasks (compliance assessment, penalty enforcement, billing, etc.). However, when modelling such tasks in the form of smart contracts, it is important to achieve a minimum degree of dependency as possible between them and the SLA itself. To clarify, consider the example smart contract snippet presented in Figure 3.2.B, and note that violating the quality requirement $Latency < 3s$ must incur the service provider's liability of consequences (i.e. penalty). If the enforcement logic is tightly coupled with a fixed SLA content, any introduced change to either of them would highly likely require revising all dependent SLA-related content or tasks. Therefore, the SLA representation approach should enable separation of concern, where SLA content is segregated from the logic of any SLA-related task such as penalty enforcement, compliance assessment and so forth.

3.4. Proposed SLA Representation Approach

By considering the IRAFUTAL principles, discussed in Section 3.3, this chapter proposes an SLA representation approach that leverages the state storage capability employed by most blockchain platforms such as Ethereum and Hyperledger Fabric.

3.4.1. Overview on the State Storage capability

Most existing blockchain platforms complement smart contracts with state storage that organise data in the form of $(key, value)$ [76]. This section focuses on Hyperledger Fabric for reasons discussed in section 2.5. In Hyperledger Fabric, every endorsing node (validator) maintains a smart contract and copy of the shared ledger, where the latter comprises state storage and a chain of blocks (see Figure 2.9). The primary benefit of the state storage capability is that it provides fast access to the state of stored assets. Hence, it mitigates the need for traversing the blockchain [29]. It is worth noting that the state storage persists the latest state of a blockchain asset in records, formatted as $(key, value, version)$, where *key* identifies an asset, *value* reflects the latest state of the asset, and *version* is a track of changes on this particular asset.

A key benefit of the state storage capability is that it enables data mutation on the latest state $(value)$ of stored assets. However, no state change is accepted unless supported with immutable transactions that are included in the shared ledger in an append-only fashion [71]. Every smart contract has direct access to its state storage and can invoke other smart contracts to access theirs as well [121]. The modifiability of the state storage enables smart contracts to execute typical CRUD operations (Create, Read, Update, and Delete) on assets stored in their state storage [122]. Nevertheless, smart contracts do not execute any functionality unless triggered with a valid transaction that undergoes a consensus mechanism and passes all validation checks

and complies with the endorsement policy in place [123][98]. That is, a transaction must be committed successfully into the ledger before being reflected on the state storage [120], as per depicted in Figure 2.9. Accordingly, one can think that state storage is about data representing the state of stored assets, while the underlying blockchain is about a chain of transaction logs that support the legitimacy of the latest state of stored assets.

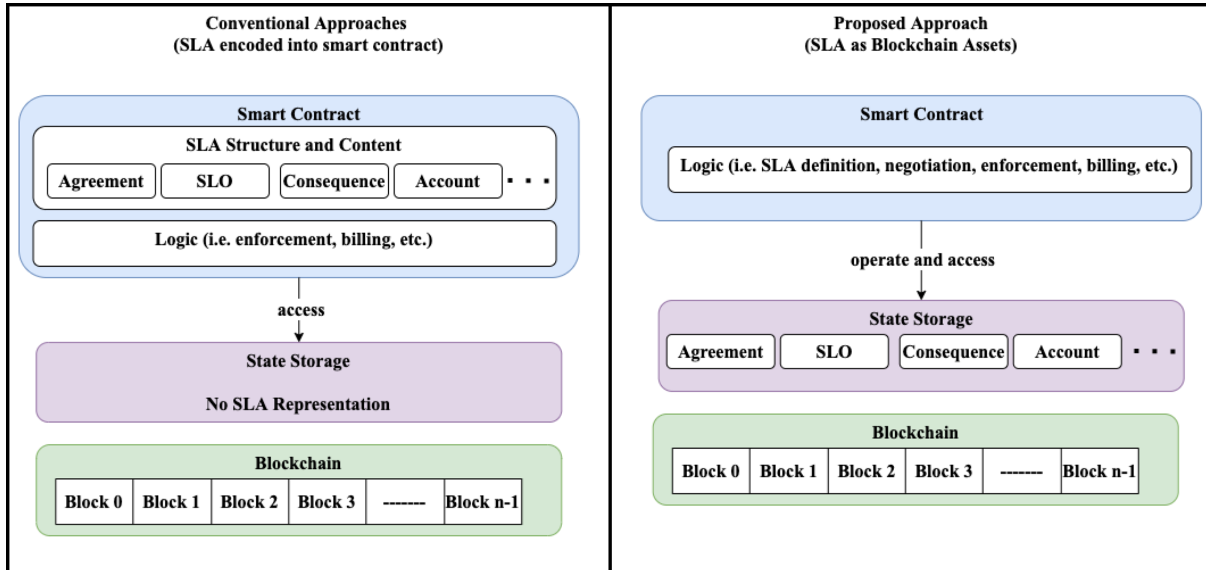


Figure 3.4 Conventional SLA representation approaches vs. the proposed approach

3.4.2. SLA as Blockchain Assets

Persisting SLA within blockchain mitigates trust issues related to central authorities and third parties and resists malicious acts such as forgery [110]. The proposed approach takes advantage of the state storage capability for realising SLA in the form of blockchain assets, which complies with IRAFUTAL principles, discussed in Section 3.3. In essence, the proposed approach discourages stipulating SLA content explicitly in the smart contract. Rather, it uses state storage for decoupling SLA content from the business logic defined in the smart contract. Figure 3.4 illustrates the primary difference between the proposed approach and conventional SLA representation approaches (see section 3.2.2). To elaborate key difference points, most surveyed blockchain-based SLA solutions tend to fully, or to a great extent, encode SLA content directly into the smart contract (i.e. see Figure 2.2). In contrast, the proposed approach refrains from directly encoding the actual SLA content in a smart contract. Instead, it designates the state storage for storing blockchain-based SLA assets. Therefore, smart contracts can maintain SLA awareness while independent of the SLA content. Moreover, smart contracts can serve various supported SLA agreements since they are no longer attached to a specific one. Vice versa, SLA content is liberated from the immutability and lifecycle of a particular smart contract. For instance, an SLA agreement stored may not concern a change to a smart contract serving an enforcement task or introducing a new smart contract. This is in contrast to conventional approaches, where an adaptation of the smart contract severely impact the lifecycle of the stored data [119][85].

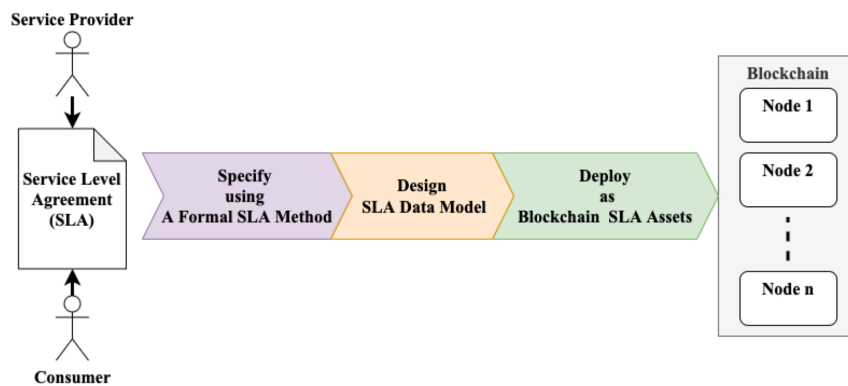


Figure 3.5 Overview on the process of SLA modelling as blockchain assets

Figure 3.5 overviews the process of modelling SLA in the form of blockchain assets. Essentially, it utilises an SLA specification tool to generate a formal SLA document. Then, it decomposes the generated SLA document into a logical collection of independent components. It also considers and constrains the association between these components. The outcome is an SLA data model deployable in blockchain assets and stored at the state storage. Following, this study elaborates this process with an example SLA model.

3.4.3. Formal SLA Specification

Formal SLA specification is a pivotal step for accomplishing a well-structured SLA document as well as for resolving issues associated with ordinary SLA documents (textual-based or semi-structured), such as ambiguity, incompleteness, and lack of interoperability [12]. Furthermore, the absence of a formally-specified SLA hinders effective maintainability and automation of SLA-related tasks such as service discovery, provisioning, incident management, monitoring, and enforcement [6][11]. Therefore, this study suggests modelling SLA data by utilising formal SLA specification methods.

There are in the literature a set of frameworks and tools dedicated for formal SLA specification; examples of which are surveyed in [9] and [12]. While any proper formal SLA specification frameworks and tools can be nominated, this study selects a specification framework contributed by Alqahtani et al. [11] for the following reasons:

- It is IoT-domain specific, which aligns well with the purposes of this study.
- It provides an open-source toolkit that helps specify an SLA that complies with their proposed formal method.
- It generates the SLA in a machine-readable format (namely, JSON).
- To the best of our knowledge, there is no other alternative dedicated to IoT-based SLA purposes.

The selected framework is used for specifying and generating an example JSON-formatted SLA, as presented in Figure 2.2 and elaborated in section 2.2.1. For demonstration purposes, the

generated example SLA is deliberately generic for the sake of demonstrating SLA modelling and representation within the blockchain.

3.4.4. Designing an SLA Data Model

This stage seeks the realisation of an IRAFUTAL-compliant SLA data model that acts as a blueprint that governs SLA assets, their properties, content, and association to each other. In essence, the modelling process utilises the formally-specified SLA document (generated in the previous step) to capture the overall SLA structure and its key properties. Figure 3.6 depicts an example SLA data model which is influenced, to a great extent, by the grammar and framework of the selected SLA specification tool. It highlights the association between independent components of the SLA model. It also adapts the formally-specified SLA document, in Figure 2.2, to accommodate additional SLA components and properties (specifically, monitoring and escrow account), which are not captured by the selected SLA framework.

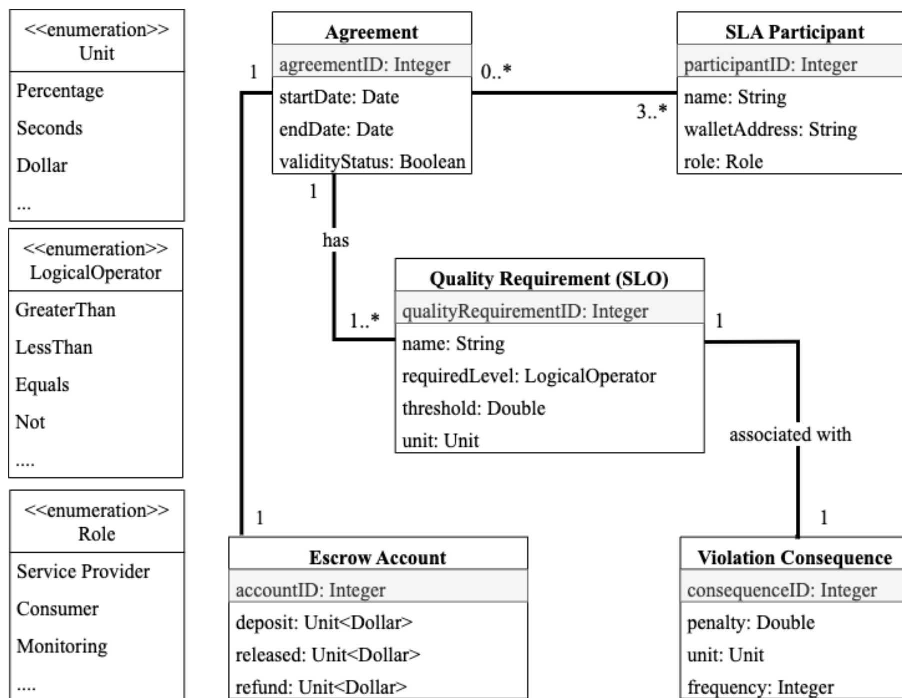


Figure 3.6 Example of IRAFUTAL-compliant SLA model based on formally-specified SLA document

3.4.4.1. Highlights on the Example SLA Model

According to the example SLA model, there are a collection of SLA components, which are SLA agreement *SA*, SLA participant *SP*, quality requirement *Q*, violation consequence *VC*, and escrow account *EA*. There can be a set of independent and uniquely identified instances created from any of these SLA components $s_i \in \{SA, SP, Q, VC, EA\}$. These instances will be eventually stored in the form of blockchain assets at the state storage level. The SLA model intentionally decouples these components from each other to enable separation of concern and reusability. Albeit, the agreement component *SA* maintains a relationship with other components by leveraging the concept of association. See Figure 3.7 which visualises the relationship

between them such that agreement act as a tree root, while others act as tree leaves. Every SLA agreement is considered enforceable within a specified duration and has a flag property that indicates whether it is active or terminated. It is important to note that, an SLA agreement $sa_i \in SA$ cannot exist without its dependencies. Furthermore, the SLA agreement component SA extends its properties to all dependent components.

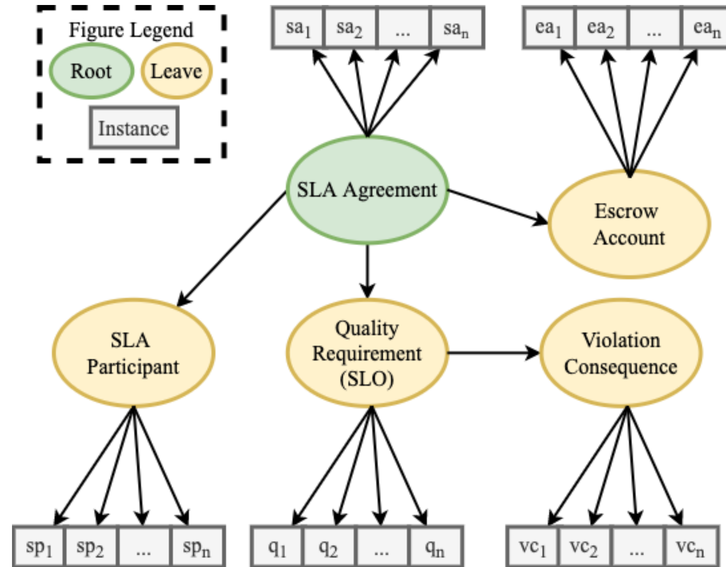


Figure 3.7 Relationship between agreement component and other SLA components.

The SLA model in Figure 3.6 declares that each agreement must be associated with at least three participants $sp_i \in SP$, at least a quality requirement $q_i \in Q$, and exactly an escrow account $ea_i \in EA$. Regarding the participants, they can take the role of service provider, consumer, or monitoring. To simplify the expression of quality requirements, the example SLA model only supports quantifiable and measured quality requirements in the form of <quality name><logical operator><value><unit> (i.e. $Availabity > 99\%$), which is sufficient for this study. The quality requirement definition can serve monitoring, compliance assessment, and proactive enforcement (i.e. corrective actions). A quality requirement shall not exist without being associated with a violation consequence $vc \in VC$ for the purposes of reactive enforcement (i.e. imposing penalty) and billing purposes. The SLA model supports providing instruction on what penalty to impose on the service provider. The SLA model supports defining the financial penalty as violation consequence in the form of $< value >< unit >$ (i.e. 25%). The violation consequence must indicate the frequency of applying the penalty per month. For example, one can define a penalty of 5% for every 1000 violation incidents to the latency requirement (See Figure 2.2). Smart contracts can apply penalties on escrow accounts associated with the agreement. The escrow account assumes a prepaid payment method. However, postpaid payment methods can be supported as well in a similar manner. For simplicity, the escrow account assumes two parties: the service provider and the consumer. The consumer deposits an agreed amount in advance, held by the smart contract and then realised by the end of the agreement.

3.4.4.2. Deployment to Blockchain

The SLA model acts as a blueprint that governs and constrains SLA representation, structure, and operation. This study uses Hyperledger Fabric (HLF) to deploy the SLA model into the blockchain network. The HLF platform employs Chaincode capability to implement two main components. The first component is the state storage, which this study utilises to implement the SLA data model in Figure 3.6. Therefore, the state storage can store instances of the SLA components $s_i \in \{SA, SP, Q, VC, EA\}$ in the form of a blockchain asset that complies with the deployed SLA data model. The second component is the smart contract that has the privilege to access and operate stored SLA assets. The smart contract can encode the logic of any SLA-related task such as SLA definition, negotiation, compliance assessment, penalty enforcement, billing, and so forth.

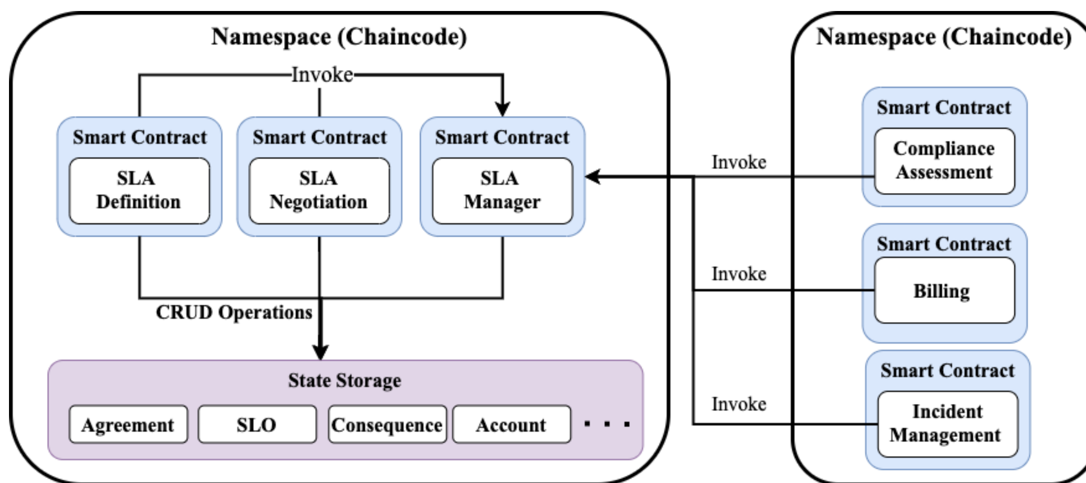


Figure 3.8 SLA model at the state storage and possible applications as smart contracts

Figure 3.8 uses an example of a possible blockchain-based architecture which can be conceivable due to the concept of decoupling the logic of SLA-related tasks from SLA content. Since SLA is being represented as blockchain assets, every SLA-related task can reuse stored SLA assets for its purposes. The architecture also suggests that every smart contract can maintain awareness of and access to SLA assets, whether it shares the same namespace of the state storage or not [1]. If a smart contract shares the same namespaces as the state storage, then it can directly access SLA assets persisted in the state storage. Otherwise, internal smart contracts can provide external counterparts with a proper access privilege to their state storage. For instance, the compliance assessment smart contract can invoke the SLA manager smart contract, as per in Figure 3.8.

Based on the example SLA data model, this chapter implements an SLA manager in the form of smart contract. Figure 3.9 depicts the SLA manager, which assumes the role of a smart contract that interfaces between the blockchain-based state storage and authorised invokers (i.e. end-users, external applications, other smart contracts), it is assigned with two primary tasks, as follows:

[1] https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/chaincodenamespace.html

- Enforcing the example SLA data model at the state storage.
- Serving basic CRUD operations (Create, Read, Update, Delete) for authorised invokers.

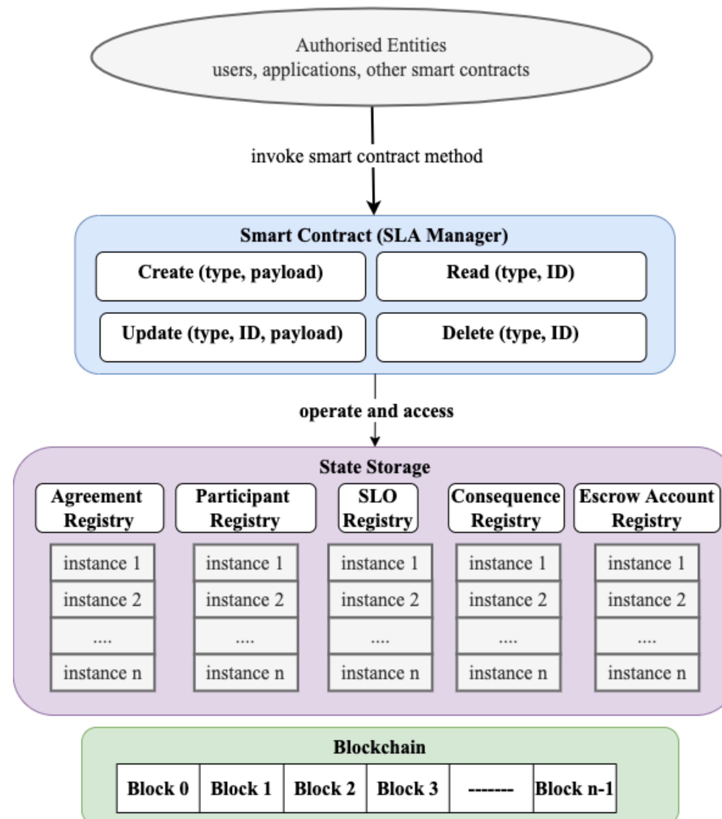


Figure 3.9 A smart contract that controls and interfaces with the deployed SLA data model

3.5. Implementation of the SLA Data Model

The main objective of an SLA data model is to enforce a set of rules and association constraints when representing SLA-related data within blockchain, particularly at the level of state storage. The example SLA data model S, in Figure 3.6, consists of a set of SLA components; namely; SLA agreement (SA), SLA participant (SP), escrow account (EA), quality requirement (Q), and violation consequences (VC). Hereafter, this chapter uses the convention $S = \{SA, SP, EA, Q, VC\}$ to denote components of the SLA data model. This study leverages the state storage to persist any instance of these SLA components $s \in S$ (see Figure 3.7) to achieve the IRAFUTAL principles (refer to section 3.3).

Hyperledger Fabric adopts a schema-less style for data representation at the state storage. It mainly organises data in the form of (k, v, ver), where k denotes asset key, v denotes asset value, and ver denotes the asset version. Every $\{k_i \in K\}$ is unique, which assist identification of, and access to stored SLA assets. Because the state storage is schema-free, every $\{v_i \in V\}$ can represent any component of the SLA data model $s \in S$; therefore, v_i can be different in size compared to another v_j. Altogether is ideal for implementing the example SLA data model due to the variation in the properties of each SLA component. Accordingly, we can state that $v_i \in S$,

where $S = \{SA, SP, EA, Q, VC\}$ which means that the value v_i can hold any SLA component in S. For instance, assume a quality requirement q_i to be a blockchain asset complying with the SLA data model. Therefore, it would exist in the state storage as (k, q, ver). This is also applicable to other components of the SLA data model where we can have (k, sp_i, ver) for SLA participants, (k, vc_i, ver) for violation consequences, (k, ea, ver) for escrow accounts, and (k, sa_i, ver) for SLA agreements. The last element ver indicates the current version of the SLA asset, such that $ver + 1$ implies a change to the state of the SLA asset.

3.5.1. The Logic of SLA Asset Manager

Having represented and implemented SLA assets at the state storage, this study proceeds to illustrate the logic of a smart contract that interfaces between authorised invokers and the SLA assets; hereafter, referred to as *SLA manager*. In essence, smart contracts act as a gateway to state storage; and thus, they play a vital role in ensuring conformance with the SLA data model. The importance of the smart contract lies in the fact that they are autonomous; hence, they are not subject to the influence of any single authority. Figure 3.9 depicts the SLA manager as a smart contract that serves CRUD operations (Create, Read, Update and Delete). Authorised entities can invoke the provided methods in order to access and operate SLA assets $S = \{SA, SP, EA, Q, VC\}$. For each smart contract invocation, and no matter which method invoked, the SLA manager ensures adherence with the example SLA data model, see Figure 3.6.

In order to invoke a smart contract method, authorised entities must submit a transaction $T(J)$, where T indicates the transaction and J denotes a payload transported by the transaction. Consider that the state storage organises and stores SLA assets in the form of records structured as (k, v, ver). Accordingly, creating a record for an SLA asset requires the payload J to contain the definition of the SLA asset. For example, assume J that defines a quality requirement as $\{q_i \in Q \mid q_i \leftarrow Latency \leq 3s\}$. Therefore, J can be used to construct an SLA asset $s \in S$ in the form of (k, v, ver). When updating an existing SLA asset $s_i \in S$, the payload J would state (k, v), where k identifies an existing SLA asset $s \in S$, and v implies the updated definition of the SLA asset. Reading existing SLA assets is important for query purposes to benefit smart contracts assigned with SLA-related tasks. For that, authorised entities can invoke read functionality with $T(J)$, where J holds the key k of an existing SLA asset. For example, but not limited to, quality requirements $q_k \in Q$ are pivotal for monitoring and compliance assessment. Another example is escrow accounts $ea_k \in EA$ and SLA participants $sp_k \in SP$ serve billing and accountability purposes, respectively. Deleting an SLA asset also requires J to the key k of the intended asset. Noteworthy that while CRUD operations are possible within blockchain at the state storage, they are subject to the logic defined at the smart contract and rigorous validation and a consensus mechanism imposed by the underlying blockchain platform. Additionally, any operation does not execute unless supported by a transaction immutably committed at the blockchain ledger.

3.5.1.1. *Naive SLA Manager Approach*

As being discussed in section 3.5, SLA assets $s \in S$ can vary in terms of their structure and properties. This variation can pose a challenge to design CRUD methods that situate all SLA assets $s_i \in S$. A naive approach would handle this variation by dedicating CRUD methods tailored specifically for every SLA asset, an example of which is in Figure 3.10. The naive approach leads to unnecessary repetition of CRUD functionalities for every SLA asset, which complicates communication between authorised entities and the smart contract and poses difficulties for the maintenance of the smart contract itself. The aftermath is particularly eminent when there is a large quantity of SLA components.

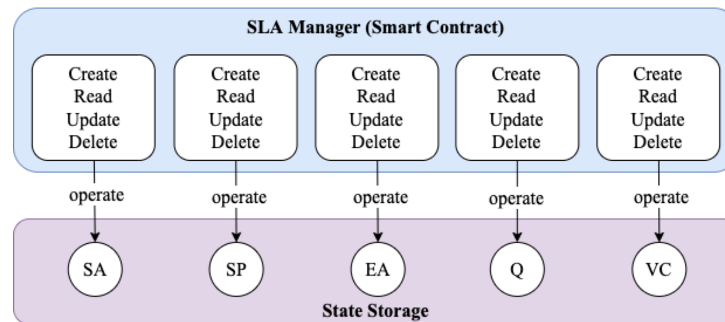


Figure 3.10 Naive SLA manager Approach for managing SLA Assets

3.5.1.2. *Enhanced SLA Manager Approach*

Figure 3.9 presents an enhanced alternative, which addresses the variation of SLA assets with a rule-based mechanism that enables generalising each of the CRUD methods for any supported SLA components. This approach enables authorised entities to communicate with generalised CRUD methods that process and operate SLA assets. To elaborate, assume a transaction $T(J)$, where J can be a JSON-formatted payload. Authorised entities must explicitly indicate in the payload J which SLA asset is concerned. Therefore, the SLA manager will have the ability to reason about the intent of the transaction, and to determine which component of the SLA data model to impose on received transactions. Ideally, any CRUD operation should occur due to mutual understanding between involved participants, for example, by providing multi-signature with the deletion transaction, which proves the authenticity of intent by all concerned parties. Hyperledger Fabric employs the concept of endorsement policy which enables specifying which entities to be involved in the transaction approval [120]. This section delves further into each CRUD operations and illustrates how they can handle various components of the SLA data model $s_i \in S$.

Creation of SLA Assets

This section demonstrates the creation of any number of SLA assets, whether they are agreements, participants, quality requirements, violation consequences, and an escrow account. The SLA manager serves a generalised creation method for any supported SLA asset. As per Algorithm 1,

the creation method consumes JSON-formatted payload J submitted as transaction $T(J)$ by authorised entities. For every received $T(J)$, the creation method runs a rule-based mechanism to examine which component of the SLA data model to impose on the received JSON-formatted payload J.

```
1 {
2   "args": [
3     "SLA component type",
4     "SLA asset definition ...",
5     [
6       "Association with existing SLA assets ..."
7     ]
8   ]
9 }
```

Figure 3.11 Generic JSON-formatted schema for defining various SLA assets

Figure 3.11 depicts a JSON-formatted template for defining various SLA assets, which are compatible with the SLA data model. The definition of any SLA asset requires the JSON payload to specify the following:

1. An SLA component type, which can be any supported element of the SLA data model $\{SA, SP, EA, Q, VC\}$.
2. The definition of the SLA component as per the SLA data model.
3. If required, identifiers of dependencies (existing SLA assets to be associated with), as per the SLA data model.

Figure 3.12 shows the compatible definition of various assets as well as an acceptable order for creating them. The association constraints imposed by the SLA data model influences the order in which the SLA component should be defined. The order means that no SLA component associates with others unless they exist and are committed to the ledger beforehand. For example, no SLA agreement $(sa_i \in SA)$ can be defined unless associated with other existing assets per the SLA data model as follows:

- Exactly one escrow account $\{ea_i \in EA\}$.
- At least one quality requirement $\{q_1, q_2, ..., q_n \in Q\}$.
  - Every quality requirement $\{q_i \in Q\}$ depends on exactly one existing violation consequence $\{vc_i \in VC\}$.
- At least three participants $\{sp_1, sp_2, ..., sp_n \in SP\}$. One of the them assumes the role of a service provider, the second participant assumes the role a consumer, while the third can assume a compliance role such as a monitoring tool or an auditor and so-forth.

Algorithm 1 illustrates the logic of processing received transactions $T(J)$, which results in the creation of SLA assets, where J is a JSON-formatted payload. The creation method deserialises

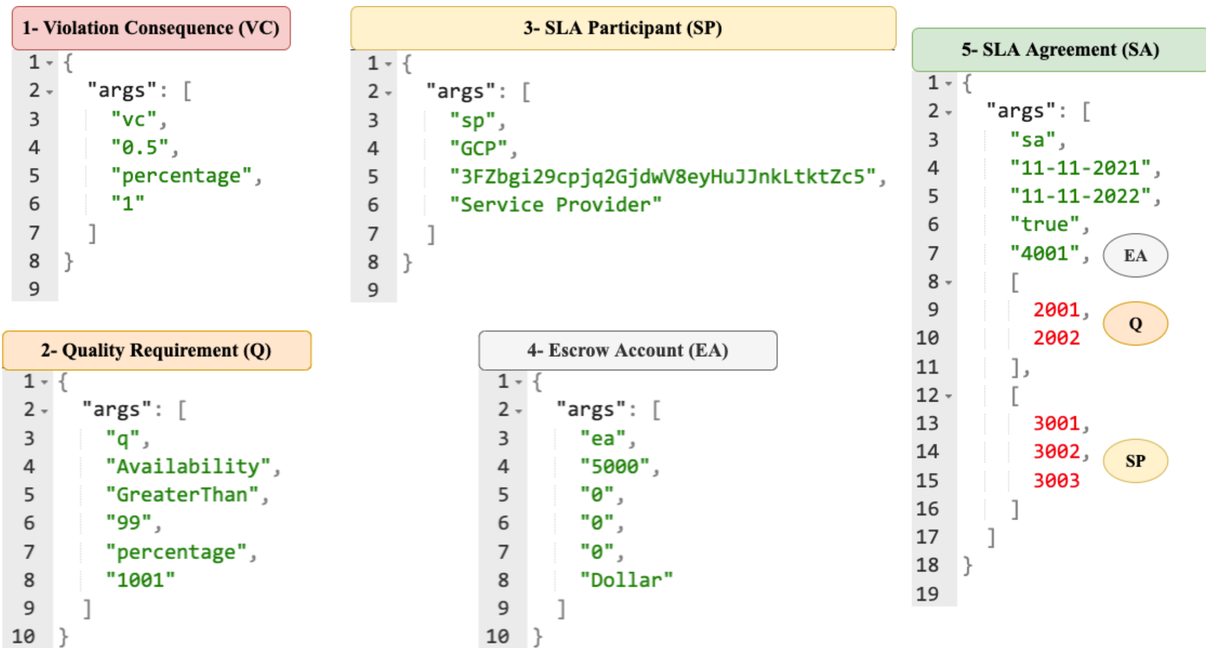


Figure 3.12 Examples of compatible JSON-formatted payloads for defining various SLA assets

the JSON payload and runs a rule-based mechanism to reason about which component of the SLA data model to impose on the received payload. If the stated SLA component is supported, the SLA manager then validates the JSON payload accordingly. It also checks whether the payload attempts to associate with non-existing assets. Afterwards, the SLA manager composes the SLA asset and persists in the state storage in the form of (k, v, ver) by extracting and using relevant properties of the JSON payload.

Creating the overall SLA agreement (SA) is done similarly to quality requirements Q in terms of association. However, the SLA agreement is more complex because it associates with multiple and various SLA components. Consider the example JSON payload for creating an SLA agreement in Figure 3.12, (the $5^t h$ JSON payload), which states main agreement properties and links to the following:

- One escrow account identified with $ea_1 = 4001$.
- Two quality requirements, identified as $q_1 = 2001$ and $q_2 = 2002$; respectively.
- Three SLA participants, identified as $sp_1 = 3001$, $sp_2 = 3002$ and $sp_3 = 3003$; respectively.

As per the SLA data model, the SLA manager validates whether the payload is associated with minimum SLA assets. Yet, the SLA agreement $sa_i \in SA$ does not explicitly state any violation consequences since it is sufficient to only specify their respective quality requirements. For the rest of the SLA components, the smart contract does not have prior knowledge of how many SLA assets are to be associated with this agreement. Therefore, the smart contact undertakes extra measures by traversing through every defined instance of escrow accounts EA, quality requirements Q and participants SP.

Algorithm 1 Creation of SLA asset in accordance with the SLA data model

```
Require: J                                   ▷ JSON-formatted SLA components from 1 to 4 in Figure 3.12
Ensure: Adherence to SLA model
 1: S = {VC, Q, SP, EA, SA}                   ▷ supported types of SLA component
 2: if J[0] ∈ S then                          ▷ is it a recognised SLA type?
 3:     if J[0] ← VC then                     ▷ Is it a violation consequence component?
 4:         if J complies with VC then
 5:             vc_penalty ← J[1]             ▷ assign penalty
 6:             vc_unit ← J[2]                ▷ assign component type
 7:             vc_frequency ← J[3]           ▷ assign violation frequency
 8:             vc_k++                        ▷ assign a unique key
 9:             vc_k ∈ VC                     ▷ add vc instance to Violation Consequence registry
10:         else
11:             reject J
12:         end if
13:     else if J[0] ← Q then                 ▷ Is it a quality requirement unit?
14:         if J complies with Q then
15:             q_name ← J[1]                 ▷ assign name for the quality requirement
16:             if J[2] ∈ Operators then      ▷ Is the logical operator recognised? e.g. Greater Than
17:                 q_level ← J[2]            ▷ assign required level
18:             else
19:                 abort
20:             end if
21:             q_threshold ← J[3]            ▷ assign threshold
22:             if J[3] ∈ VC then             ▷ query whether there exists the instance of violation consequence
23:                 q_vc ← J[4]               ▷ associate with the violation consequence
24:             else
25:                 abort
26:             end if
27:             q_k++                         ▷ assign a unique key
28:             q_k ∈ Q                       ▷ add q instance to Quality Requirements registry
29:         else
30:             reject J
31:         end if
32:     else if J[0] ← SP ∨ EA ∨ SA then      ▷ or other types of SLA components
33:         Instantiation is done in a similar manner to the above ...
34:     end if
35: else
36:     Reject
37: end if
```

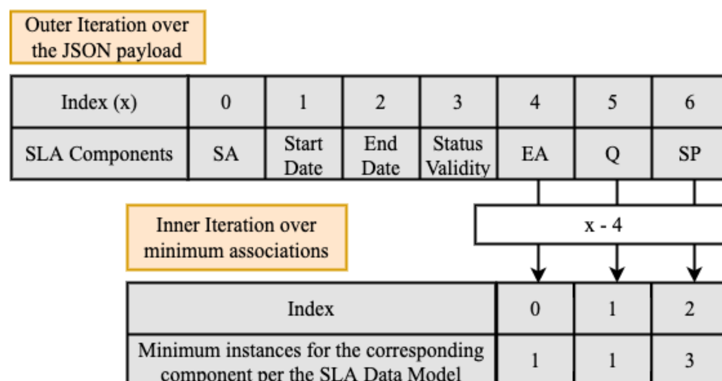


Figure 3.13 Validating minimum instances of each component in the JSON payload against the SLA data model

Firstly, line 1 in Algorithm 2 declares an array of three elements which are $|EA|$, $|Q|$, and $|SP|$. Each one of them defines the minimum instances of each cosponsoring SLA component as per the SLA data model (see Figure 3.6). For example, $|EA| = 1$, $|Q| = 1$, and $|SP| = 3$. Figure 3.13 depicts seven main elements of the JSON payload representing the complete SLA agreement. line 7 loops through JSON payload starting from $|J| - 3$, which is the fourth element, until $|J| - 1$, which is the last element. Note that $|J| - 3$ points to an existing escrow account, $|J| - 2$ points to a collection of quality requirements, and lastly $|J| - 1$ points to a collection of participants.

Secondly, the inner iteration in Figure 3.13 validates that each specified component from $|J| - 3$ to $|J| - 1$ adheres to the minimum required instanced as per enforced by the SLA data model. Finally, upon the success of SLA agreement creation, the SLA manager informs all concerned participants about this event and supplies them with the complete agreement in a JSON-formatted document, as per in Figure 3.12.

Algorithm 2 Agreement Composition

Require: J ▷ JSON-formatted SLA Agreement (the 5th component in Figure 3.12)
Ensure: Adherence to SLA Model

```
1:  sdm ← [|EA|, |Q|, |SP|]     ▷ an array of minimum instances of each SLA component as per Figure 3.6
2:  if J[0] ← SA then                                            ▷ SLA Agreement
3:      if J complies with SA then
4:          sa_startDate ← J[1]                                   ▷ assign start date
5:          sa_endDate ← J[2]                                     ▷ assign end date
6:          sa_validityStatus ← J[3]                              ▷ assign validity Status
7:          from ← |J| − 3                ▷ Loop beginning. |J| denotes the JSON (array) size
8:          to ← |J| − 1                                          ▷ Loop beginning
9:          for (x = from, x <= to, x++) do                      ▷ examine correct association
10:             size ← |J[x]|                     ▷ denotes the size of the current JSON element
11:             if size ≠ sdm[x − 4] then       ▷ validate minimum elements against the SLA model
12:                 Abort
13:             end if
14:             for (i = 0, i <= size − 1, i++) do        ▷ Loop through the current JSON element
15:                 if J[x][i] non-Exist then
16:                     Abort
17:                 end if
18:             end for
19:         end for
20:         sa_k++                                                ▷ assign a unique key
21:         sa_k ∈ SA                            ▷ add sa instance to agreements registry
22:     else
23:         reject J
24:     end if
25: end if
26: Example Output: See Figure 3.14
```

Reading SLA Assets

Any SLA-related tasks (i.e. monitoring, enforcement, billing, etc.) can maintain SLA awareness by accessing existing SLA assets at the state storage. For instance, monitoring tools residing in the external world needs to maintain awareness of quality requirements, which enables setting thresholds of when to consider the service provider is in violation [124]. Other SLA tasks can be

encoded in the form of smart contracts such as compliance assessment [112] and enforcement of violation consequences [125][118]. The SLA manager smart contract exposes a read method that enables authorised entities to query existing SLA assets. The read method requires invokers to supply both the *id* and *type* of the asset, which enables the SLA manager to query the state storage. If the asset exists, the SLA manager retrieves it and responds to the invoker with a JSON-formatted document as per in Figure 3.12.

Updating SLA Assets

Authorised entities can invoke the SLA manager to update existing SLA assets. As per discussed in section 2.2.1, SLA may need amendment for several reasons during a typical SLA lifecycle [10] such as:

- Customising a default SLA agreement during the negotiation stage.
- Adjusting an SLA due to renegotiation or recent performance report.
- Reflecting the service provider's performance on related assets such as escrow accounts.
- Rectification an error in the SLA agreement.
- SLA termination.

As per the IRAFUTAL principles discussed in section 3.3, such a process should be user-friendly and avoid the need for subject-matter experts such as smart contract developers or blockchain operators. Furthermore, the proposed approach encourages amendable, independent, reusable and identifiable SLA assets by representing them at the state storage. Accordingly, a graphical user interface (GUI) or an automated tool (external applications or other smart contracts) can interface with the SLA manager to update existing SLA assets with proper authorisation and authentication. Some uses of the update functionality include modifying properties' values of existing SLA assets as well as adding or removing the association with others.

Figure 3.9 illustrates a generic update functionality served by the SLA manager, which can adjust any existing SLA asset. The process of updating an SLA asset executes similarly to the creation method, presented in Algorithm 1 and Algorithm 2. The only difference between SLA creation and update methods is that authorised invokers must supply a key $s_k \in S$, identifying an existing SLA asset. Otherwise, the SLA manager must reject the transaction.

The updated version must comply with the example SLA data model presented in Figure 3.6. Once the updated version of the SLA asset is accepted and committed to the blockchain records, the SLA manager informs all concerned parties about the change to readjust accordingly. For example, monitoring tools need awareness of any changes in the current SLA agreement. Consequently, they can readjust thresholds and triggers accordingly. Noteworthy mentioning is that the successfulness of an update operation leads to a change of the asset version $(k, v, ver++)$. The next chapters discuss the implication of the version change in further depth; particularly

concerning the Multi-Version Concurrency Control (MVCC) [126]; a mechanism employed by Hyperledger fabric to prevent the double-spending problem [127].

Deletion of SLA Assets

While this study generally discourages assets deletion, it discusses it to refute misconceptions about asset deletion and elaborate on the difference between state storage and blockchain. First of all, the state storage maintains the last state of SLA assets, while the ledger's blockchain maintains all transactions about stored assets. While the state storage is amendable such that it accepts typical CRUD operations, no amendment is applied unless supported with a valid transaction that passes all validation checks, endorsement policies and consensus mechanism imposed by the underlying blockchain platform; namely, Hyperledger Fabric [29] [127]. Therefore, deleting an asset from the state storage does not necessarily mean deleting the log of transactions from the ledger's blockchain.

Asset deletion may be desired when there is an orphan asset that is of no use any longer. For instance, consider an agreement associated with three quality requirements. For any reason, the agreement has been updated to be associated with only two quality requirements. This is where an SLA asset can be left abandoned and not used. While it is possible to delete any existing asset from the state storage, this study discourages the deletion of any SLA asset with consideration of the following:

- When deleting an asset (i.e. SA or Q) may leave dependencies orphan and unused.
- The SLA data model encourages usability of SLA components. For example, the same quality requirement may associate with different agreements. Therefore, deleting an agreement and all its dependencies will harm other agreements that share in common these dependencies.

3.6. Evaluation and Observation

3.6.1. Failure Test Units

Unlike traditional deployment practice, it is difficult to rectify an error or conduct maintenance on smart contracts after their deployment to the blockchain network [84] [119]. For that, the smart contract must undergo careful testing coverage to ensure its compliance with the SLA data model and meeting expected behaviour. Table 3.1 shows a set of basic failure test units conducted on the smart contract before deployment to the blockchain side. While this test coverage does not claim to be exhaustive, it demonstrates how to mitigate and account for failure threats to the smart contract (SLA manager). The table presents a set of testing units on the SLA manager. For each of them, there is the following:

- **Expectation:** a description of normal behaviour.
- **Test:** a failure deliberately crafted to push issues on the surface.

Table 3.1 Failure Tests conducted on the SLA manager smart contract

Expectation	Test	Methods
Reject unrecognised asset type	SLA component with the correct structure according to the SLA data model, However, stating a type other than {VC, Q, SP, EA, SA}	Create Update
Reject unrecognised asset structure	SLA component with a recognised SLA asset type {VC, Q, SP, EA, SA}. However, the structure violates the SLA data model.	Create Update
Reject incorrect value type.	- Define unrecognised currency for an escrow account. - Define unrecognised logical operators (i.e. XOR). - Define a string value where it should be integer.	Create Update
Association with only existing SLA asset	Attempt creating SLA assets without creating their dependencies beforehand.	Create Update
Reject unrecognised ID	Use non-existing ID	Update Read Delete

- **Methods:** CRUD methods are subject to the test unit and must meet the expectations.

The Smart contract underwent multiple development iterations until it reached a sufficient level of maturity suitable for this study. This study does not approve any development iteration unless it passes at least the listed test units in Table 3.1. The testing implementation is available in the public GitHub Repository (see footnote[2]). Further experiments in this chapter and following chapters build on top of these testing units.

3.6.2. *Use Case 1: SLA Definition*

The section evaluates a basic application that demonstrates the use of the SLA manager for composing a set of reusable SLA assets at the state storage. Figure 3.14 illustrates and demonstrates a graph of SLA assets created via the SLA manager, which satisfies the IRAFUTAL principles, discussed in section 3.3. Every SLA asset of the graph is instantiated from SLA components supported by the example SLA data model $\{VC, Q, SP, EA, SA\}$. The presented graph is achievable

[2] https://github.com/aakzubaidi/slaController

thanks to the association constraints imposed by the SLA data model. It consists of a service provider sp_1 that is engaged in two SLA agreements $\{sa_1, sa_2\}$. There is also a monitoring tool sp_4, which participates in the two SLA agreements. Each SLA agreement is associated with a separate escrow accounts $\{ea_1, ea_2\}$ and two different SLA consumers $\{sp_2, sp_3\}$. There are two violation consequence assets $\{vc_1, vc_2\}$ and three quality requirements $\{q_1, q_2, q_3\}$. Both of the quality requirements $\{q_1, q_3\}$ make use of the same violation consequence vc_1. It is also possible to use a different violation consequence as in the case with q_2 and vc_2, respectively. Finally, the SLA agreement sa_1 states all existing quality requirements $\{q_1, q_2, q_3\}$. On the other hand, the other SLA agreement sa_2 shares in common the quality requirements $\{q_1, q_3\}$.

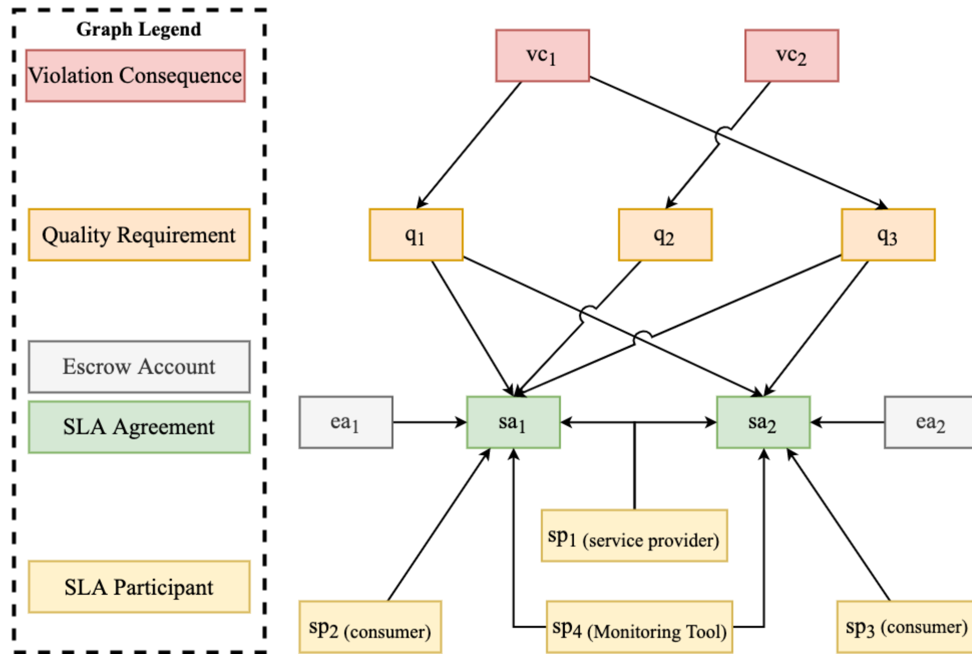


Figure 3.14 Example graph of reusable instances SLA units persisted at state storage.

3.6.3. Use Case 2: SLA Negotiation

Conventional SLA representation approaches hinder smooth and effective SLA negotiation. (section 3.2.3 provides further detail). On the other hand, the proposed approach enables instant and user-friendly SLA negotiation within a blockchain environment. Consequently, mitigating, to the minimum possible, the need for blockchain experts such as smart contract developers and operators (refer to Figure 3.3.b). Assuming a proper graphical user interface (GUI) is in place, SLA negotiation can benefit from the SLA manager, as shown in Figure 3.8. Noteworthy is that the SLA manager materialises the proposed SLA representation approach and complies with the IRAFUTAL principles. While modelling a robust SLA negotiation protocol is not the ultimate goal of this study, a basic negotiation protocol is built on top of the SLA manager to demonstrate the usefulness of the proposed approach in resolving issues related to conventional approaches. Moreover, this section highlights some considerations regarding this matter, which can be useful for future work and of interest to researchers in the domain.

Regarding the use case of SLA negotiation, consider that the SLA manager serves basic CRUD operations for SLA assets. Note that the SLA manager enforces the example SLA data model on any operation on SLA assets. Therefore, a supplementary smart contract for SLA negotiation acts as a second layer on top of the SLA manager. The smart contract mediates between service providers and consumers $\{sp \in SP\}$, with the purpose to aid them to reach an SLA agreement over the blockchain. Assuming a GUI is in place, SLA participants can partake in a contractual session to propose, approve or reject any action supported by the SLA manager on SLA components, given that they comply with the example SLA data model. Figure 3.15 overviews the concept of SLA negotiation over blockchain and illustrates primary components typically involved in the process. First, there is a smart contract dedicated to basic SLA negotiation functionalities, which enables proposing, approving or rejecting actions on SLA components. The smart negotiation contract leverages CRUD operations provided by the SLA manager for conducting the negotiation functionalities. Hyperledger Fabric provides endorsement and consensus mechanisms, exploited to facilitate consensus and endorsement on these functionalities. The GUI interfaces with the SLA negotiation smart contract and exposes the negotation functionalities in a user-friendly manner for SLA participants.

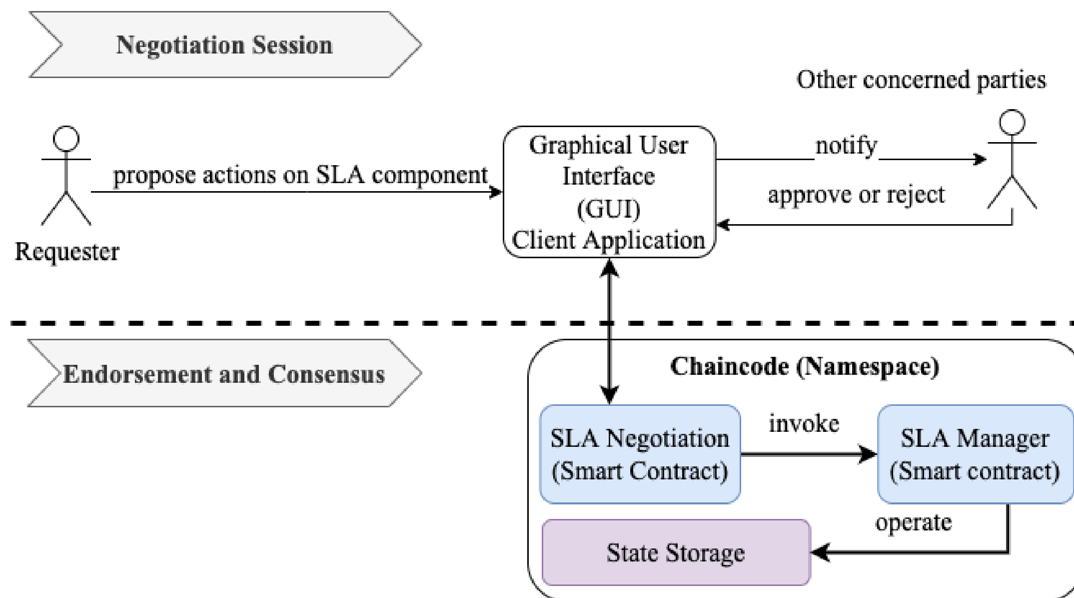


Figure 3.15 Overview on SLA negotiation session over blockchain

Figure 3.16 presents a basic SLA negotiation protocol designed and implemented to demonstrate the advantages of the proposed SLA representation. The implementation of the basic SLA negotiation protocol is publicly available as an open-source project on GitHub (refer to footnote[3]). As per the basic protocol, the SLA negotiation smart contract mediates between the involved parties and enable them to propose a new or customised SLA agreement compatible with the example SLA data model. The GUI enables composing all dependencies $\{VC, Q, SP, EA\}$ of an SLA agreement $sa_i \in SA$. The GUI produces a JSON payload for each SLA component, similar to the examples presented in Figure 3.12. The GUI interfaces with the SLA manager and

[3] https://github.com/aakzubaidi/slaController

automatically submits a series of transactions to create an SLA asset for each JSON payload. For a successful creation, the order of these transactions and the format of their attached payloads must comply with the SLA data model as discussed in section 3.5.1.2.

Following the creation of all agreement dependencies, the service provider proposes a complete SLA agreement *SA*, which associates with all desired dependencies in $\{VC, Q, SP, EA\}$, an example of which is in Figure 3.12. The GUI interfaces with the negotiation smart contract, and invokes the *propose* to start the negotiation process as illustrated in the basic negotiation protocol (see Figure 3.16). The negotiation smart contract notifies all involved participants of created SLA proposal and provides them with the key of the newly created SLA agreement key sa_k.

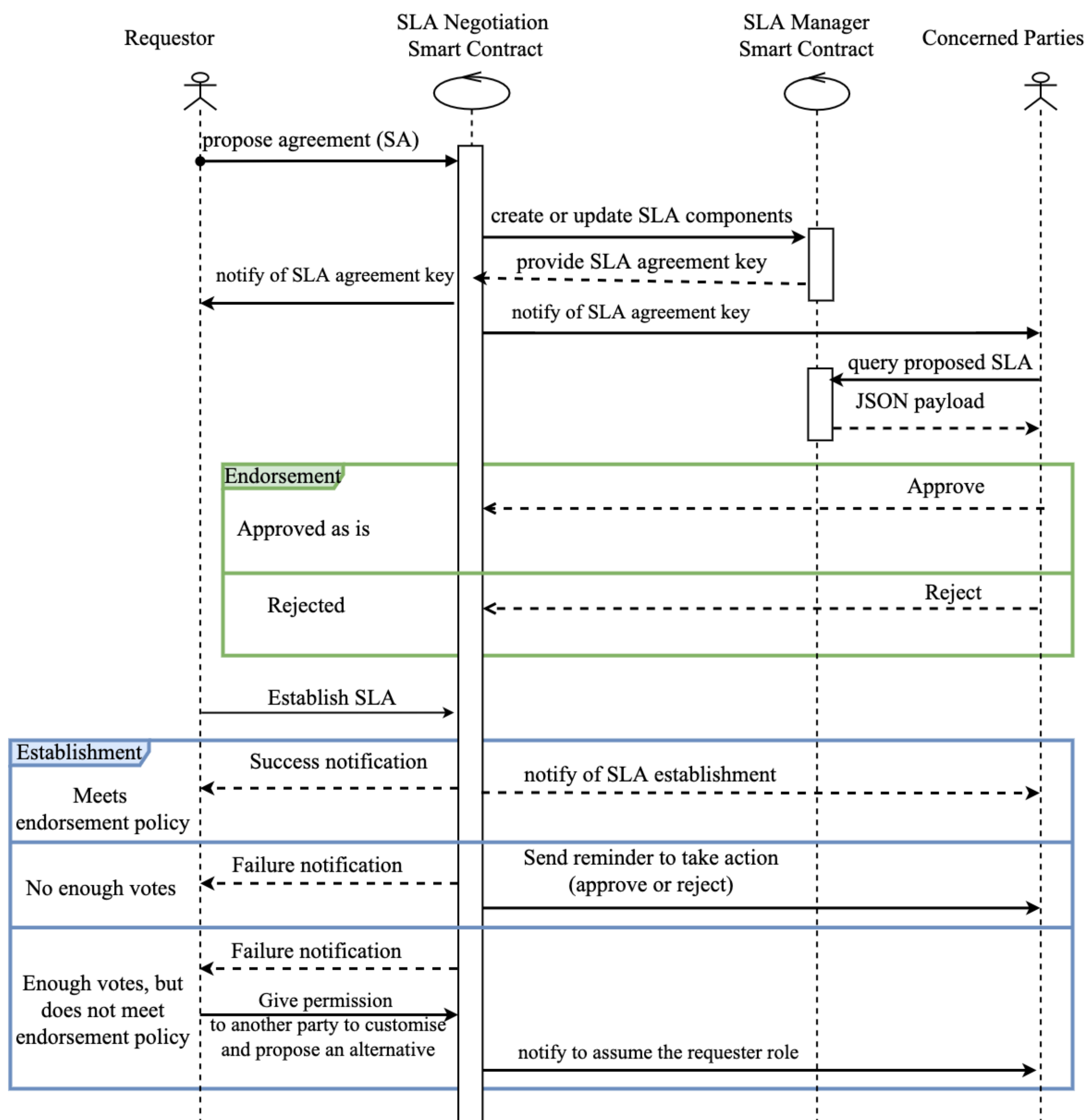


Figure 3.16 Basic SLA Negotiation Protocol: A use case

For negotiation purposes, the validity status of the SLA agreement component is by default set to *false* to indicate that this agreement is not yet established and enforced (refer to the SLA data model in Figure 3.6). The validity status of the SLA agreement does not change to *true*

unless all involved participants approve the proposed SLA agreement. In addition, the agreement component *SA* is adjusted to include two properties for each involved participant: vote (Boolean) and signature (String), which are set to null by default. These properties reflect the vote integrity of each SLA participant whether they approve or reject the proposed SLA agreement. The signature property holds the participant signature for vote integrity purposes.

The Service provider may attempt to establish the SLA at any point in time by invoking the negotiation smart contract; outcomes of which can be one of the following:

- The proposed SLA agreement meets an endorsement policy (i.e. all parties must approve). Accordingly, the validity status becomes *true*, declaring the confirmation and enforcement of the proposed SLA agreement. In practice, this event could trigger further actions as well, such as service provisioning, monitoring, billing, and payment [110][109].

- All or some of the concerned parties has not engaged in the voting process. Therefore, the negotiation smart contract notifies parties who neglected to vote. In this case, the service provider may try again to establish the SLA.

- All has voted; however, the proposed SLA agreement does not meet the endorsement policy in place. In this case, the service provider may abandon this agreement or offer a concerned party the opportunity to propose an updated version of the SLA agreement. In this case, the SLA negotiation smart contract resets all properties of SLA agreements related to the voting process (validity status = false) and (vote = false, signature = false) for every involved participant. Subsequently, the selected party (i.e. consumer) may attempt to reiterate the negotiation process, as illustrated in the basic negotiation protocol (see Figure 3.16). Unlike the service provider, involved parties can only update existing service level agreement sa_k, assigned by the service provider.

To sum up, while the example basic negotiation protocol does not claim to be exhaustive, it demonstrates how the proposed SLA representation approach can facilitate negotiation within blockchain in a user-friendly and timely manner without the need for blockchain experts or the need to migrate from existing smart contracts.

3.6.4. *Proposed Approach vs. Conventional Approaches*

This chapter presented an SLA representation approach which encounters limitations of conventional approaches, discussed in section 3.2.2. Table 3.2 presents prominent similarities and differences between both of them.

3.6.5. *Threats to Validity*

In principle, the proposed approach can be also applicable to any blockchain platform where state storage is supported. However this approach has been only evaluated on Hyperledger Fabric, a selection of it is justified in section 2.5. It would be also interesting to conduct this study on Ethereum, where the programming language for smart contract is constrained and transaction are

Table 3.2 Comparison table between conventional and proposed SLA representation approaches

Facet	Conventional Approaches	Proposed Approach
Representation within blockchain	✓	✓
Represented at:	Smart contract	State Storage
Immutable ledger records	✓	✓
Deterministic execution	✓	✓
SLA Awareness within blockchain	✓	✓
Decoupling logic from SLA	✗	✓
Suitability for SLA tasks automation	✓	✓
SLA Definition:	Blockchain Experts	User-friendly
Modifiable SLA definition	✗	✓
Resistance to malicious modification	✓	✓
Verifiable SLA integrity	✓	✓
SLA host availability	✓	✓
Reusability of SLA content	✗	✓
Negotiation before SLA establishment	External to blockchain	Within blockchain
Renegotation after SLA establishment	✗	✓
Error-rectifiability	✗	✓

executed at cost. While Hyperledger Fabric by default supports data modelling, Ethereum seems to only enable this approach by applying the data segregation patterns which separates between the business logic and data model [85]. A limitation of the proposed approach is that it only considers quantifiable quality requirements for the sake of demonstration. Notwithstanding, there remains the challenge of representing non-quantifiable requirements, a practice that while exist, but is generally discouraged by most contemporary SLA frameworks and guidelines. While the proposed approach demonstrates the advantages of SLA representation at the state storage, a future work needs to consider more complex SLA structure such as conditional statement and exception.

3.7. Conclusion

This chapter presented an SLA representation approach that addresses limitations of conventional approaches, discussed in section 3.2.2. Table 3.2 presents prominent similarities and differences between both of them. In terms of resemblances, both represent SLA within blockchain which provides smart contracts with the necessary awareness and preserves important properties such as deterministic execution, integrity, high availability, ease of access, and immunity from malicious behaviour. On the other hand, they differ mainly regarding the mechanism of SLA representation within the blockchain. Whereas conventional approaches represent SLA in the form of a smart contract, the proposed approach primarily decouples SLA from the smart contract's logic and represent it at the state storage level. While the presented approach attains key benefits of conventional approaches, it mitigates its limitations in several ways by satisfying the IRAFUTAL principles, discussed in section 3.3. Moreover, it aligns well with a typical SLA lifecycle which normally expects SLA modification due to either negotiation before SLA establishment or

renegotiation afterwards. It also enables SLA definition and error rectification in a user-friendly and timely manner while mitigating the need for blockchain experts to the minimum possible. The next chapters experiment and evaluate the proposed SLA representation approach for other key stages of a typical SLA lifecycle: SLA monitoring, compliance assessment, and penalty enforcement in the context of IoT.

Chapter 4. Blockchain-based SLA Compliance Assessment and Penalty Enforcement

Summary

This chapter considers SLAs supplied by cloud-based IoT service providers. It conducts a pilot study to explore the potentiality of a blockchain-based approach for assessing SLA compliance and enforcing violation consequences. The pilot study builds on and extends the SLA representation and awareness approach, proposed in Chapter 3 to accommodate monitoring, compliance assessment and enforcement. This chapter assumes an SLA covering a couple of quality requirements related to a cloud-based IoT component; Namely, an MQTT broker that serves a healthcare application. Given the vital role of the compliance assessment, this chapter validates the dependability of the smart contract. Additionally, it evaluates the applicability and feasibility of the proposed compliance and enforcement approach with an emulated IoT-based healthcare scenario and a monitoring tool. However, a performance benchmarking in a production environment reveals that the dependability validation in a testing environment does not necessarily guarantee reliability when deployed to a real blockchain network. It demonstrates that by deploying the proposed approach to a real blockchain network (Hyperledger Fabric) and benchmarking its throughput, latency, success, and fail transaction performance. Finally, this chapter discusses the outcomes of the pilot study and sheds light on a set of lessons learnt and recommendations brought forward to the following chapters.

4.1. Introduction

Cloud providers typically employ Service Level Agreements (SLAs) to ensure the quality of their provisioned services. Similar to any other contractual method, SLA is not immune to breaches. Ideally, an SLA stipulates violation consequences (e.g. penalties) imposed on cloud providers when they fail to conform to SLA terms [16]. The current practice assumes trust in service providers to acknowledge SLA breach incidents and execute associated consequences. That is, cloud providers promise to process incidents in good faith, assuring their consumers to impose SLA consequences on themselves. While intriguing, trust is usually taken for granted [24]. Furthermore, it is typically the consumer's responsibility to report a service level degradation, supported by evidence deemed irrefutable by the service provider [124]. This is usually a tedious process and manually handled [128].

Most traditional enforcement studies, such as those by in Faniyi and Bahsoon [62] and Mubeen et al. [10], assume trust in either service providers or trusted third parties. However,

it can be inviting for some consumers to manipulate evidence to support violation incidents. On the other hand, providers may not react well to poorly formed claims, regardless of their validity [20]. In some scenarios, unresolved disputes have to be escalated to mediators, or other jurisdiction means.

Blockchain can be relevant whenever trust is an issue. Accordingly, this pilot study seeks to explore how blockchain can serve distrusted SLA processes such as compliance assessment and penalty enforcement. It essentially aims to offload the authority privilege of such distrusted SLA tasks from centralised authorities or third parties. Rather, it assigns the authority of these distrusted tasks to smart contract that operate within blockchain environment in a non-repudiable manner and beyond the direct influence of any SLA party.

This pilot study considers using the Hyperledger Fabric platform, hereafter abbreviated as HLF, as the underlying blockchain infrastructure. HLF enables modelling distrusted processes via the concept of smart contract [29] [87]. Accordingly, this chapter conducts a pilot study to explore the potentiality of a blockchain-based approach for assessing SLA compliance and enforcing violation consequences in the context of cloud-based IoT services. For that, it assumes an SLA that covers a couple of quality requirements related to a cloud-based IoT component (MQTT-based) that serves a healthcare application. For that, it considers an simplified version of an SLA provided by a cloud-based IoT provider, namely Google Cloud Platform (GCP). Refer to Appendix A section A.2.2 for the full version of the SLA. The simplified SLA covers an SLA-guaranteed IoT component that enables IoT devices to connect and communicate with other cloud services as in Figure 2.4.

The pilot study explores the benefits of the SLA representation and awareness approach, proposed in Chapter 3, and extends on it for monitoring, compliance assessment and enforcement. Moreover, it conducts a testing framework to validate the dependability of the proposed solution. Additionally, it evaluates the applicability and feasibility of the proposed compliance and enforcement approach with an emulated IoT-based healthcare scenario and a monitoring tool. Furthermore, it deploys the proposed approach to a blockchain network (Hyperledger Fabric) and benchmarks its throughput, latency, success, and fail transaction performance. Finally, this chapter discusses the outcomes of the pilot study and sheds light on a set of lessons learnt and recommendations brought forward to the following chapters. The source code of the implementation and experiment is publicly available under GNU GPL V3.0 License on GitHub[1].

4.2. Preliminary

4.2.1. A Remote Healthcare Scenario

This pilot study considers a simplified IoT scenario where a healthcare provider hires a set of cloud-based IoT services for remote healthcare purposes. Refer to section 2.2.6, which highlights the role of cloud providers in IoT ecosystems. As depicted in Figure 4.1, three major entities communicate over MQTT protocol, which are patients, the healthcare provider, and

[1] https://github.com/aakzubaidi/MQTT-SLA-Blockchain-QoS-Enforcement

an ambulance department. The MQTT brokers manage a set of topics: two of which are vital signs and emergencies. Medical sensors/devices collect relevant vital signs about patients and publish them to a respective topic, called *vital signs*. If a patient's condition becomes severe, the MQTT broker notifies both the healthcare provider and the ambulance department through the *Emergency topic*.

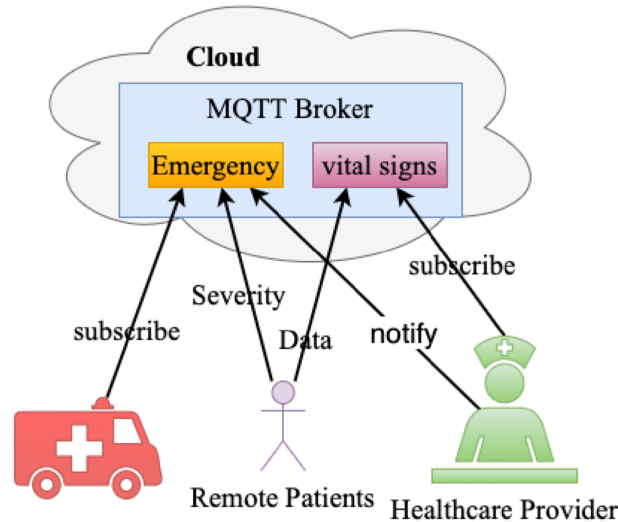


Figure 4.1 Example IoT healthcare scenario employing MQTT for data exchange

4.2.2. SLA Example

The MQTT mechanism is critical to the reliability of the presented remote healthcare scenario. A straightforward failure impact is the lack of communication among all parties: patients, ambulance, and healthcare providers. The severity intensifies further when an emergency requires urgent attention. In response, cloud providers would typically guarantee the quality of their offered services by the mean of an SLA.

This study selects Google Cloud Platform (GCP) as an example cloud provider that offers an IoT component as a service and supports MQTT protocol. By examining the GCP SLA (see Appendix A), it primally promises the following two quality requirements (SLOs):

- q_1: a monthly uptime percentage of cloud-based IoT component as $Availability \nless \%99.9$. The Availability percentage can be measured in seconds and calculated as per Equation 4.1.
- q_2 a maximum error rate due to MQTT broker malfunction $ErrorRate \ngtr 10\%$. The SLA measures the error rate according to equation 4.2, which reads as total *failed MQTT messages* due to MQTT Broker's malfunction divided by the total number of *valid MQTT messages*. In the equation, f refers to a fail message due to a failure of the MQTT broker. The SLA does not recognise an MQTT message as f unless it is a valid MQTT message but fails due to the MQTT broker's malfunction. This implies that the SLA does not recognise any failures due to other factors such internet connection instability or a fault at

the client side. Therefore, valid MQTT messages can be the sum of both f and successful messages s.

Therefore, the violation rate can be generalised as per Equation 4.3, where c refers to compliance cases while b refers to violation cases. The example SLA holds GCP accountable for violations of these quality requirements. As a remedy, it promises to compensate consumers by the mean of financial service credits if they happen to violate the promised quality of their IoT component. For instance, assume vc_i to be 25% to be applied the deposited amount in case of violation of the any quality requirement either q_1 or q_2.

$$Availability = (\frac{uptime - downtime}{uptime}) \times 100 \tag{4.1}$$

$$ErrorRate = (\frac{\sum_{i=1}^{n} f}{\sum_{i=1}^{n} s + f}) \times 100 \tag{4.2}$$

$$violationRate = \frac{\sum_i^n b}{\sum_i^n b + \sum_i^n c} \times 100 \tag{4.3}$$

4.3. Revisiting the Current Trust Model

As with any contractual method, SLA is susceptible to breaches. In the current practice, obligated providers must acknowledge SLA violations in order to execute SLA consequences. Consumers must trust that cloud providers will meet this expectation. Figure 4.2 suggests revisiting the current trust model by shifting distrusted tasks from obligated providers to executable contracts operating in a non-repudiable fashion. This pilot study takes into account a set of considerations for realising the proposed approach, as follows:

4.3.1. SLA Awareness within blockchain

SLA awareness is necessary for automated tasks such as smart contracts and monitoring tools. As chapter 3 argues, SLA must be represented within the blockchain to attain the IRAFUTAL principles (see section 3.3). These principles are relevant to this study in several ways. First, it enables smart contracts to maintain direct access to SLA content while guaranteeing deterministic behaviour. Second, SLA can achieve immunity from malicious acts and the risk of a single point of failure (i.e. unavailability), which is an important feature for both the monitoring tool and smart contracts. Chapter 3 extensively highlights the significance of these principles and others. Accordingly, this pilot study represent the example GCP SLA in section 4.2.2 in accordance with SLA data model in Figure 3.6. It considers two quality requirements which are:

- $q_1 \leftarrow Availability \nleq \%99.9$.
- $q_2 \leftarrow ErrorRate \ngtr 10\%$.

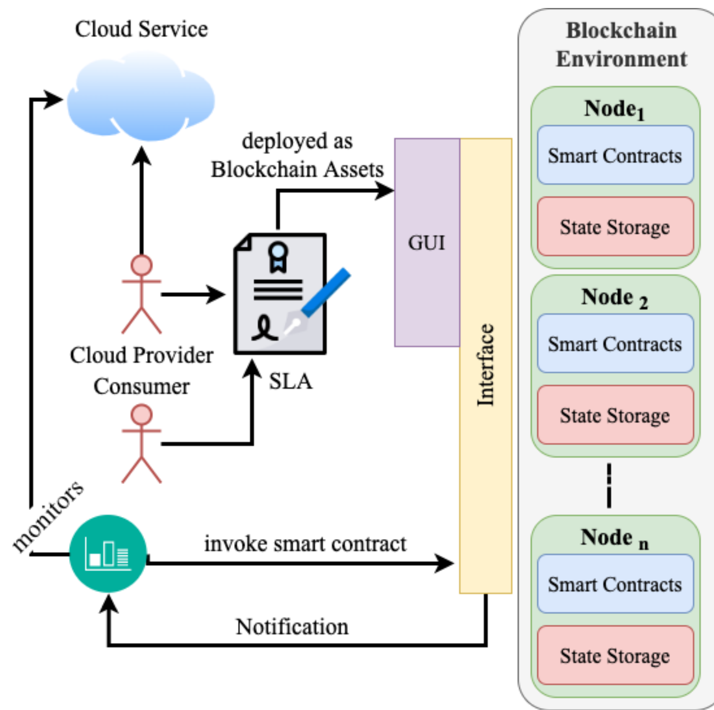


Figure 4.2 Overview on the proposed Blockchain-based Compliance enforcement

With regard to penalties, the pilot study assumes one violation consequences vc_1 reusable for both q_1 and q_2. The vc_1 enforces a penalty of 25% of the deposited amount on the cloud provider.

4.3.2. SLA Monitoring

As being discussed earlier in section 2.6.1, smart contracts are supposed to be terminable and deterministic [31]. Therefore, smart contracts are not optimal for conducting endless activities such as monitoring. Thus, external monitoring/alerting tools must be in place to help smart contracts form a decision on the compliance level of obligated providers. Section 2.6.1 also highlights that monitoring tools can form a single point of failure in terms of availability, efficiency and trust. Subsequently, this study assumes proper measures are in place and thus focuses on compliance assessment over blockchain via smart contracts.

4.3.2.1. SLA Awareness for Monitoring Tools

For monitoring tools to support smart contracts, they must maintain necessary awareness of SLA in place with regard to relevant quality requirements and what constitutes a violation at which thresholds and intervals. Figure 4.3 illustrates that authorised monitoring tools can leverage the SLA manager, discussed in section 3.5.1, to query the SLA agreement sa_i persisted at the blockchain state storage. Subsequently, monitoring tools can adjust its threshold, triggers and interval accordingly. In case of an SLA renegotiation or termination, the SLA manager takes advantage of the event emitter, supported by Hyperledger fabric, to notify the monitoring tool about the latest state of the SLA. As a result of SLA negotiation, the monitoring tool

reconfigures its parameters to accommodate the latest version of the SLA. For SLA termination, the monitoring tool halt the process of submitting further transactions to the blockchain side.

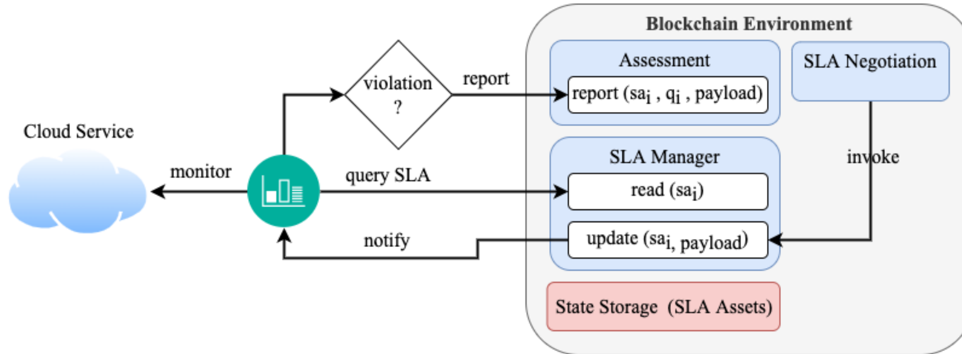


Figure 4.3 The monitoring role in the Blockchain-based SLA compliance Assessment

4.3.2.2. Metric Collection

The primary task of the monitoring tool is to ceaselessly observe the service associated with the SLA (e.g. Cloud-based IoT Component) and report its performance to the blockchain side. For example, consider the promised quality of the MQTT broker in the GCP SLA, namely availability q_1 and error rate q_2. Accordingly, the monitoring tool must collect metrics related to the performance of the MQTT broker, such as up-time and the number of received/sent messages. The MQTT specifications [2] describe the MQTT brokers, how to reason about various defects, what other data can be collected. This pilot study considers the MQTT specification for designing and implementing the monitoring approach. The MQTT specification describes when the MQTT broker ungracefully disconnects and what error codes indicate this event. It is important to distinguish this broker downtime from other causes of inability to access the MQTT broker, possibly due to instability of internet connection or a failure at the MQTT client-side and not at the MQTT broker side.

4.3.2.3. Alert of Incidents

In most existing blockchain platforms, such as Ethereum and Hyperledger Fabric, smart contracts neither self-execute themselves nor should take the initiative to query the external world [31]. To elaborate, assume a smart contract that assesses a service provider's compliance with the SLA. The smart contract cannot take the initiative to query metrics collected by the monitoring tool. Alternatively, an alerting mechanism can trigger the smart contract and provide it with data necessary for formulating a decision on the compliance status of the obligated provider. As Figure 4.3, the monitoring tool must alert the smart contract by submitting a transaction about any identified incidents to the compliance assessment smart contract. This action will trigger the smart contract to conduct the compliance assessment. Given the rapid frequency of logged events, this pilot study intentionally applies a throttling mechanism to control and limit the

[2]https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html

sending rate from the monitoring tool to the blockchain side. In essence, the proposed approach considers the service provider to be compliant unless proven to be otherwise. Therefore, the monitoring tool does not send a transaction to the blockchain for every logged event. Instead, it aggregates collected logs and only transacts with the blockchain upon detecting an SLA violation. The application of such measure intends to prevent overwhelming the blockchain side, which helps avoid performance bottlenecks and achieve an efficient storage usage of each validating peer. Given the assumed frequency of incidents in this thesis, smart contracts are triggered based on the occurrence of SLA violations. However, such a measure would not be sufficient in the case of rare incidents due to the large payload of events, which would cause a transaction failure. Subsequently, it might be more viable to introduce scheduling and queuing strategies that periodically enable submitting transactions at regular intervals. Regardless of the sending control mechanism type, the proposed smart contract design can process received data (logs) about compliance and violation events, as discussed in the following sections.

4.4. Compliance Assessment over Blockchain

The concept of smart contracts is the key enabling feature for realising blockchain-based decentralised applications. Accordingly, this pilot study employs the concept of smart contracts for realising decentralised compliance assessment and penalty enforcement over the blockchain. For that, this section adjusts the example SLA data model in Figure 3.6 for the pilot study. Furthermore, it also describes a proposed logic of compliance assessment and penalty enforcement.

4.4.1. Adjusted SLA Data Model

We use the concept of smart contracts to automate decision-making on the compliance level of obligated providers (Cloud providers) beyond the control of centralised authorities. While the example SLA data model in Figure 3.6 considers basic SLA components, it must be adjusted to accommodate data necessary for compliance assessment. Figure 4.4 illustrates an enhanced SLA data model that considers an extra component, performance report *PR*, for persisting data needed for conducting the compliance assessment. However, the enhanced SLA data model is simplified for illustration and demonstration purposes.

The compliance assessment is responsible for operating the performance report *PR* data records and periodically instantiates $pr_i \in PR$ for each quality requirement. For instance, if the cloud provider follows a monthly billing cycle, we normally expect $pr_1, pr_2, pr_3, pr_4..., pr_{24} \in PR$ evenly distributed over the two quality requirements q_1 and q_2. In other words, each quality requirement will be associated with 12 unique performance reports by the end of a typical financial year.

The smart contract persists each $pr_i \in PR$ at the state storage in the form of (k, v, ver), k is a unique key of the performance report, v contains the properties of the performance report as in Figure 4.4, and ver is an incremental value that reflects current version of the record at each update operation. The value of each record v consists of three properties as follows:

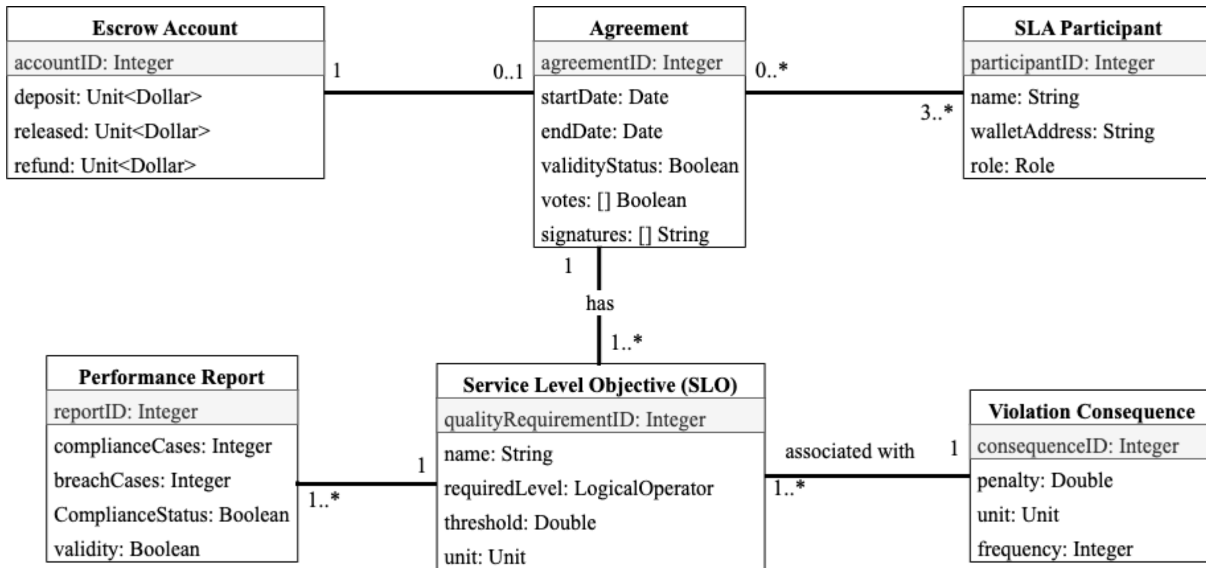


Figure 4.4 Adjusted SLA data model to accommodate performance reports

- **Compliance cases** pr_c: total count of events in which the service provider demonstrates to be compliant with a quality requirement.
  - For the Availability quality requirements q_1, it expresses total uptime in seconds.
  - For the error rate quality requirements q_2, it expresses the count of successful MQTT messages.
- **Breach cases** pr_b: total count of events in which the service provider fails to satisfy a quality requirement.
  - For the Availability quality requirements q_1, it expresses downtime in seconds.
  - For the error rate quality requirements q_2, it expresses the count of failed MQTT messages.
- **Compliance status** pr_{cs}: a flag used by the smart contract to indicate whether the service provider meets the associated quality requirement. The flag is a boolean value such that *true* implies compliant while false *implies* violation.
- **Validity** pr_v: a flag to indicate whether this performance report is usable by the smart contract. If the flag holds *true*, it implies that this performance is usable for the current billing cycle. On the other hand, it means the performance report is not current. Rather, it is related to a previous billing cycle (e.g. one of the lapsed months). Therefore, the smart contract must create a new performance report pr_{i++} and associates it with the respective quality requirement q_i.

Figure 4.5 adopts the scenario presented in section 4.2.1 and the example SLA presented in section 4.2.2 to produce a graph of SLA assets persisted at the state storage, where there is one SLA agreement sa_i that associate with three participants which are:

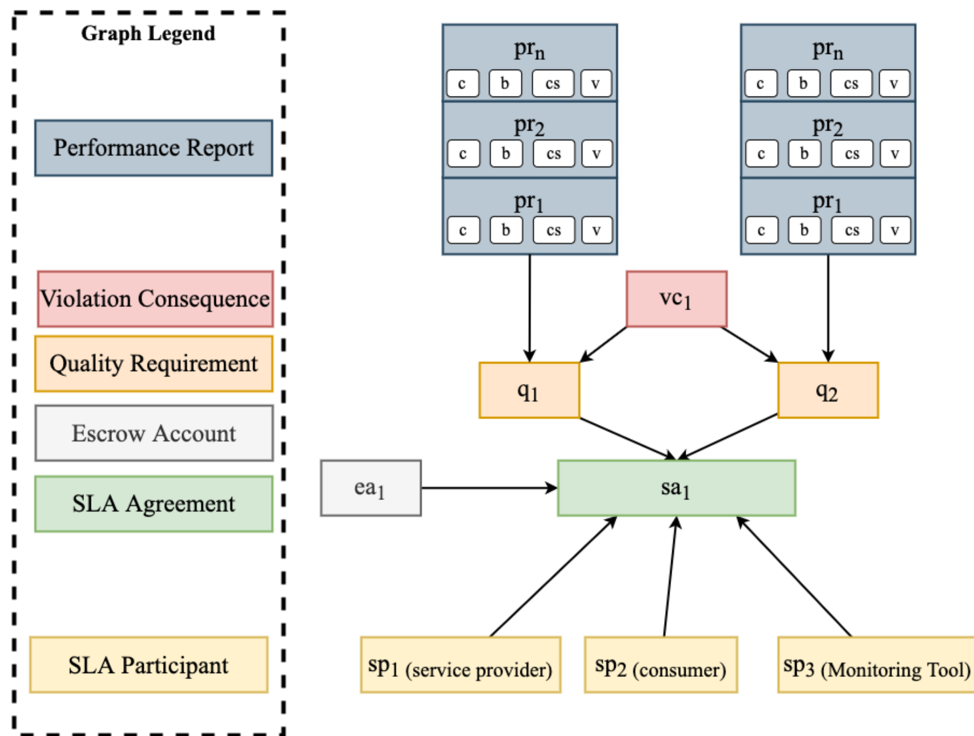


Figure 4.5 A graph of SLA assets based on the adjusted SLA data model

- sp_1: service provider (presumably, the Google Cloud Platform).
- sp_2: consumers (presumably, a healthcare provider).
- sp_3: a monitoring tool.

Moreover, Figure 4.5 depicts an escrow account ea_1 that is associated with agreement sa_1, where the consumer deposit the agreed amount for the provisioned service (Cloud-based IoT component). There are two quality requirements q_1 and q_2 associated with agreement. Both of them use the same definition of the violation consequence vc_1. For each of the quality requirements, there is a set of performance reports $rp_1, rp_2, ..., rp_n$ managed by the compliance assessment smart contract.

4.4.2. The Compliance Assessment Logic

The pilot study leverages the smart concept to design and implement a compliance assessment logic. Two stages take place in every billing cycle (every month, for example), which are incidents processing and penalty enforcement; detailed as follows:

4.4.2.1. First Stage: Incident Processing

For each billing cycle (e.g. monthly), the smart contract waits for violation incidents submitted by authorised monitoring tools. As Figure 4.6 depicts, authorised monitoring tools triggers the compliance assessment whenever there is a violation incident. This is done by submitting a transaction to the assessment smart contract, which invokes a dedicated method called *report*.

This method accepts three parameters: the SLA agreement sa_i, the concerned quality requirement q_i, and a payload.

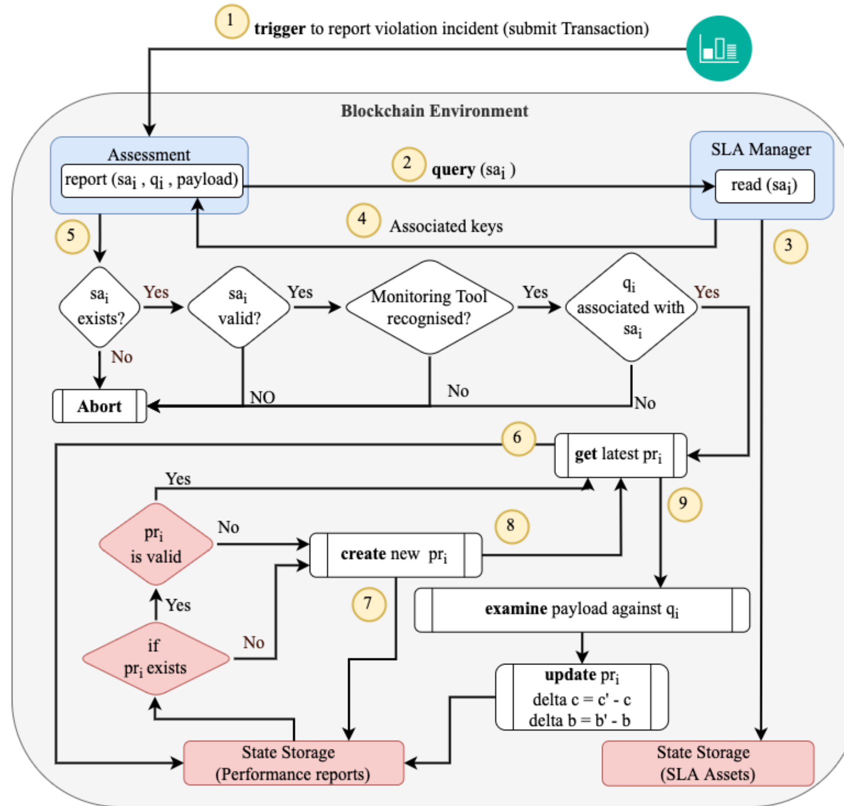


Figure 4.6 Overview on incident processing procedures at the smart contract level

As Figure 4.6 illustrates, the assessment smart contract does not process any reported incident unless the transaction passes a basic validation. First, the assessment smart contract invokes the SLA manager to query associated assets related to the agreement sa_i. As section 3.5.1.2 explains, the SLA manager retrieves all assets associated to the the agreement sa_i and responds with a JSON-formatted document similar to Figure 3.12. Subsequently, the assessment smart contract uses the retrieved JSON document and proceeds with a set of checks as follows:

1. asserts whether the supplied SLA agreement sa_i is recognised.

2. asserts whether the supplied SLA agreement sa_i is valid, meaning that it is established and not terminated or expired.

3. asserts whether the monitoring tool is authorised to report incidents of relation to this sa_i by checking whether it is listed as a participant.

4. asserts whether the supplied quality requirement q_i is in fact associated with the sa_i.

Although these validation checks do not claim to be exhaustive, they satisfy the purposes of this study. For each q_i, there is a set of performance reports $pr_i \in PR$ (see section 4.4.1). The smart assessment contract operates these performance reports to track the compliance records of the service provider. It uses the supplied key of the quality requirement q_i to query the last

performance report pr_i associated with it. If it happens to be the first reported incident about this particular quality requirement, the smart contract instantiates a new record (k, v, ver) for the performance report, where v comprises of the following:

- **Compliance cases:** $pr_c \leftarrow 0$.
- **Breach cases:** $pr_b \leftarrow 0$.
- **Compliance status:** $pr_{cs} \leftarrow true$ which express compliance of the service provider towards the associated quality requirement q_i.The proposed approach considers the service provider to compliant unless proven to be otherwise at the end of the billing cycle.
- **Validity of the performance report:** $pr_v \leftarrow true$, which express that the performance report is currently usable for compliance assessment purposes.

As section 4.4.1 discusses, each performance report $pr_i \in PR$ accommodates incidents by updating the total count of compliance and breach cases. Hence, the monitoring tool uses the payload parameter to provide the latest about both compliant c' and breach cases b'. The smart contract uses the payload to update pr_c and pr_b properties of the persisted performance report pr_i by applying the difference in accordance to Equation 4.4 and Equation 4.5; respectively. That is, the smart contract updates both pr_c and pr_b by applying the difference between the current value and the received value. Note that, the smart contract imposes a set of conditions in order accepted the received payload. First, both pr_c and pr_b must not hold negative values. Second, both c' and b' must be at least equal or greater than their counterparts, pr_c and pr_b; respectively. These measures are in place to prevent negative values and misbehaviour. For example, if the received value is negative, then there will be a miscalculation that will render the assessment useless.

$$\Delta c = c' - pr_c \mid \{c', pr_c\} \in \mathbb{N} \land c' \geq pr_c \tag{4.4}$$

$$\Delta b = b' - pr_b \mid \{b', pr_b\} \in \mathbb{N} \land b' \geq pr_b \tag{4.5}$$

For instance, consider the availability quality requirement q_1 and the acceptable error rate quality requirement q_2, discussed in sections 4.2.2 and 4.3.1. The q_1 requires the total time of both uptime and downtime in seconds. Subsequently, the smart contract maps the uptime to c' while the downtime to b'. Similarly, q_2 requires the total count of *successful* and *failed* MQTT requests. Therefore, the smart contract maps the successful requests to c' and the failed requests to b'.

The monitoring tool supplies the latest metrics for each quality requirement within the payload of the submitted transaction. The smart contract uses these metrics to update the current performance report in accordance to Equations 4.4 and 4.5. It is worth mentioning that update operations are executed at the state storage provided by HLF. However, the blockchain does not commit any update operation unless backed up with an immutably stored transaction that meets all relevant checks such as consensus mechanism and endorsement policies.

4.4.2.2. Second Stage: Penalty Enforcement

The second stage takes place at the end of the billing cycle. At this stage, any involved participant (i.e. service provider, consumer, monitoring tool) can trigger the smart contract to conclude the billing cycle. As Figure 4.7 illustrates, the smart accounts for any remaining logs by the monitoring tools in terms of compliant cases c' or violation cases b' for each quality requirement q_i. Similar to the procedure presented in Figure 4.6, it updates the associated performance report pr_i prior to the conduct of the compliance assessment. Before proceeding further, the smart contract ensures whether it is the end of the billing cycle. If so, it decides on the compliance status of the obligated provider concerning each quality requirement. Moreover, it applies the associated violation consequences vc_i on the associated escrow account ea_i if the smart contract finds the service provider in violation of the quality requirement.

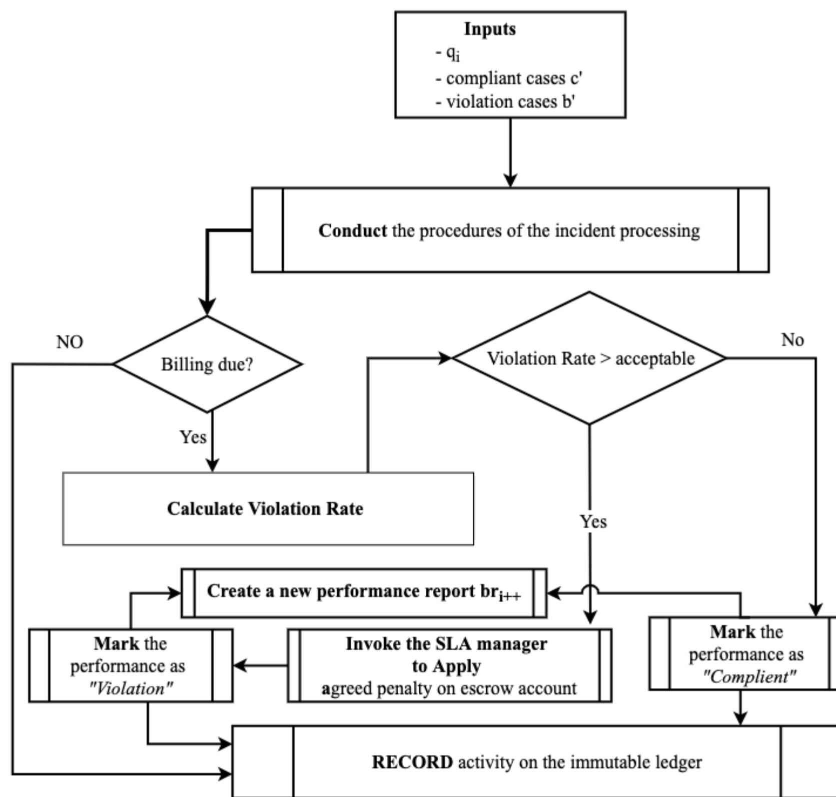


Figure 4.7 Assessing the compliance of the service provider with regard to each quality requirement

The smart contract bases its decision on the compliance status mainly on examining the *violation rate*, calculated as per Equation 4.3, against the associated quality requirement. For instance, it examines the service provider's performance against the minimum threshold set for the uptime; say $q_1 \nless 99.9\%$. Another example is by examining the service provider's performance in terms of error rate against the maximum threshold stipulated in the SLA in place; say $q_2 \ngtr 10\%$. If the smart contract concludes that the obligated provider has fulfilled its promise by not exceeding the stipulated threshold, then the *compliance status* property of the performance report pr_i remains intact; expressed as *true*. Otherwise, the smart contract marks *compliance status* property of the performance report pr_i, expressed as *false*, to indicate violation.

Following the compliance assessment of each quality requirement, the smart contract invokes the SLA manager to apply the agreed penalty stated on the escrow account ea_i, as per the associated violation consequence vc_i (see chapter 3 for more on the SLA data model and the SLA manager). Therefore, the smart contract falsifies the current performance report pr_i by marking its validity property as *false*. Subsequently, it creates a new performance report pr_{i++} associated with the respective quality requirement in order to utilise it for the next billing cycle.

4.5. Dependability Experiment

To ensure the reliability of our approach, we consider three facts. First, the smart contract conducts assessment tasks, which decides on the compliance level of obligated providers. Therefore, the dependability of the decision logic has to be validated [129]. Second, smart contracts operate beyond the control of a single authority. Therefore, rectifying a logical error in decentralised applications is not as straightforward as it would be in traditional applications [75]. Third, smart contracts are executed in a deterministic fashion. Thus, we expect the same cloud provider performance to always produce the same assessment and decision on the compliance status (either violation or compliant).

Accordingly, this section presents how we validate the dependability of the decision making for every development iteration. The smart contract is written in Java depending on Java chaincode library[3] provided by Hyperledger Fabric. For every development iteration on the smart contract, we implement and execute a set of testing and validation units using Junit 5.0 test framework [4]. The smart contract is deployed to the IBM blockchain platform [5]. The platform is also used to facilitate the configuration and management of all necessary elements (such as blockchain network, identities, access level, crypto materials, etc.).

4.5.1. Failure Test Units

Table 4.1 presents a test coverage on the assessment smart contract in terms of both stages; the incident processing and the penalty enforcement. The coverage test is conducted following each development iteration on the smart contract to ensure that it behaves as expected. Both the presented approach and the testing coverage result from several improvement iterations on the assessment smart contract. Each one corrects and accommodates failures that we identify in its precedent.

For instance, the presented approach (see Figure 4.6) has evolved to account for some edge cases where the monitoring presumably provides invalid parameters. For example, assume a scenario where the current SLA is updated due to renegotiation or error-rectification. As Figure 4.3 illustrates, the SLA manager must notify the participating monitoring tool of such an update on the SLA, which invites the monitoring tool to adjust its configuration accordingly. Examples of SLA amendment may include but are not limited to SLA termination, expiry,

[3] https://hyperledger.github.io/fabric-chaincode-java/
[4] https://junit.org/junit5/docs/current/user-guide/
[5] https://marketplace.visualstudio.com/items?itemName=IBMBlockchain.ibm-blockchain-platform

Table 4.1 Failure Tests conducted on the compliance assessment smart contract

Incident Processing	Penalty Enforcement	Test unit	Expected Behaviour
✓	✓	sa_i does not exist	abort
✓	✓	sa_i exists but terminated or expired	
✓	✓	q_i does not exist	
✓	✓	sa_i and q_i exist but not associated	
✓	✓	monitoring tool $\notin SP$	
N/A	✓	Escrow account ea_i does not exists	
✓	✓	no pr_i for q_i	create new br_i
✓	✓	pr_i exists but invalidated	ignore and create br_{i++}
N/A	✓	Billing not due	enforcement disabled
✓	✓	$c' < pr_c$ or $b' < pr_b$	reject
✓	✓	$c' \notin \mathbb{N}$ or $b' \notin \mathbb{N}$	reject
✓	✓	$pr_c \notin \mathbb{N}$ or $pr_b \notin \mathbb{N}$	reject
N/A	✓	$violationRate = 0\%$	even though, pr_i must be created

deletion, or disassociation from quality requirements. If the monitoring tool neglects, for any reason, to accommodate most recent SLA updates, it may provide invalid parameters or, otherwise, valid but disassociated. Consequently, the presented approach and the testing coverage are upgraded to accommodate this edge case.

Additionally, the pilot study designs Equations 4.4 and 4.5 to constrain and validate inputs (c' and b') from the monitoring tools. This is because the experiment revealed a miscalculation that renders the assessment useless when testing the smart contract using negative values. Another example is experimenting with a use case of 100% compliance rate, meaning that no incident was reported from the monitoring side. Previously, the creation of a performance report pr_i was not triggered unless the smart contract received an incident from the monitoring tool. Therefore, the case of 100% compliance rate would not create any performance report pr_i at all. Therefore, the enforcement logic in Figure 4.7 requires submitting the last metrics for each quality requirement and going through the incident processing for the last time. This measure takes place for two primary reasons. First, this measure accounts for any remaining metrics, either compliant c' or breach b', before finalising the current billing cycle. This measure forces the creation of a performance report pr_i, which is key for deciding on compliance status and any actions to be taken on the escrow account (e.g. penalty enforcement).

Moreover, the experiment reveals that transactions submitted from monitoring tools may fail (e.g. due to a timeout). The logic used to require the monitoring tool to only submit new cases. For instance, consider 3 transactions T_1, T_2, and T_3 that are submitted from the monitoring tool. Assume that T_2 fails for some reason, such as a transaction timeout. Therefore, this missed transaction leads to an unintentional drop in reported metrics. This case contributed to the requirement that c' or b' must always be equal or greater than their counterparts. This condition helps the smart contract reason about the integrity of reported metrics. It also leads to the recommendation of having a transaction retry mechanism at the monitoring side, as being illustrated in Algorithm 5.

All in all, the test coverage and the presented approach have been updated to accommodate these edge cases. While the coverage test is not exhaustive, it raises basic edge cases regarding the proposed approach. Following, the pilot study proceeds to conduct further validation experiments. The experiment does not approve a smart contract implementation unless it passes all testing units for each stage of the compliance assessment as presented in Table 4.1.

4.5.2. Decision Accuracy Validation

The assessment smart contract conducts a critical decision-making task on the service provider's compliance. Therefore, this section examines the the two stages of compliance assessment discussed in section 4.4.2, which are the *incident processing* as in Algorithm 3 and *Penalty Enforcement* as in Algorithm 4.

4.5.2.1. Validating Stage 1: Incidents Reporting

The proposed compliance assessment approach is based on Equation 4.3, which is composed of two elements: compliant cases c and breach cases b. This proposed approach applies this equation for calculating the violation rate for all supported quality requirements. The elements c and b of the violation rate equation are mapped to their equivalent properties of the performance report pr_c and pr_b. The proposed approach depend on metrics from the monitoring tool to update these properties. More specifically, the Incident processing functionality conducts its calculation based on c' to update pr_c (see Equation 4.4) and b' to update pr_b (see Equation 4.5).

Given the criticality of the violation rate calculation, Algorithm 3 investigates and validates the ability of the smart contract to update properties of the performance report correctly pr_c and pr_b. The experiment simulates a monitoring tool that repeatedly reports incidents to the smart contract. The simulated monitoring tool submits compliant and breaches cases for each incident, c' and b', respectively. The experiment expects the smart contract to behave per the logic presented in Figure 4.6. After each incident processing, the experiment queries the current values of pr_c and pr_b and asserts whether they are appropriately updated.

The validation experiment adopts the minimum reporting interval of 60 seconds (1 minute), as specified by Google metrics instrumentation page [6]. The experiment assumes the worst-case scenario, where the MQTT broker exhibits malfunction behaviour lasting for 30 days. Thus, the monitoring tool keeps triggering the smart contract every minute. This means the smart contract should receive 43200 incidents by the end of the month. Therefore, the experiment iterates 43200 times, such that there are new compliant c' and a new breach b' for every iteration. The experiment assumes a fixed value for each c' and b'. Accordingly, by the end of algorithm execution, pr_c should equals $c' \times 43200$ and pr_b should equals $b' \times 43200$.

The validation experiment also assumes the best-case scenario, where the MQTT broker maintains a perfect uptime percentage $uptime \leftarrow 100\%$ and a perfect error rate $errorRate \leftarrow 0\%$. Consider that the presented approach requires monitoring tools to refrain from submitting transactions to the smart contract unless there is an incident to report. This measure is in place to

[6] https://cloud.google.com/monitoring/api/metrics

Algorithm 3 Validating correct update of the performance report pr_i

Require: sa_i, q_i, c', b'
Ensure: $pr_c \wedge pr_b$ are correctly updated
1: pr_c should equals $c' \times \sum_0^n T$ ▷ T: transaction from monitoring tool
2: pr_b should equals $b' \times \sum_0^n T$
3: **for** $i \leftarrow 1$ to $Transactions_count$ **do**
4: **Invoke** assessment smart contract (sa_i, q_i, c', b')
5: **Wait** for transaction resolution
6: **Query** (pr_c , pr_b)
7: **if** $(pr_c \vee pr_b)$ NOT correctly updated **then**
8: **Terminate** with error
9: **Abort**
10: **end if**
11: **end for**
12: **Return** (pr_c, pr_b)

prevent monitoring tools from overwhelming the smart contract with unnecessary transactions because it assumes the service provider to be compliant unless proven otherwise. However, this experiment assumes a scenario where the monitoring tool submits a transaction at a regular interval of 1 minute where $b' \leftarrow 0$ and any fixed value for the metric c' where $c' > 0 \mid c' \in \mathbb{N}$. This indicates constant compliance performance with no incident at all.

For either the worst-case or best-case scenarios, the validation experiment in Algorithm 3 is conducted on both quality metrics q_1 and q_2. in all cases, the validation experiment proved the ability of the smart contract to pass the checks in Table 4.1 and accommodate all received metrics submitted from the simulated monitoring tool and reflect them as per expected on the performance report.

4.5.2.2. Validating Stage 2: Penalty Enforcement

At the end of every billing cycle (assume 30 days), the smart contract assesses the compliance level of obligated providers by examining the violation rate (see Equation 4.3) against the threshold of the associated quality requirement. For instance, by running the violation rate calculation of pr_2 against the agreed *Error Rate Threshold* of q_2, For example 10%. Accordingly, it decides whether to enforce a penalty on the escrow account as per the associated violation consequence vc_1.

To examine the decision accuracy, we select quality requirement q_2, which states that the error rate should not exceed 10%. Then, we prepared a set of cases that are already known to us to be either *violation* or *compliant*. In other words, each case is a test round where we target a fresh performance report pr_i and deliberately feed the smart contract with metrics that eventually causes the pr_i to be classified as compliant or violation. The violation group has 50% of these cases, in which each pr_i exceeds the 10% *error rate* threshold. The compliant group has the other half of the cases, which does not exceed the 10% error rate threshold.

Table 4.2 Diagnostic Accuracy 2x2 Table

	Violation	Compliant
Positive	True Positive (TP)	False Positive (FP)
Negative	False Negative (FN)	True Negative (TN)

Given the criticality of the enforcement task, the experiment relies on a testing method called *Diagnostic Accuracy* [130] for validating the decision made by the smart contact. This testing method employs a set of measures and calculations based on 2×2 table, as shown in Table 4.2. The table is composed of 4 elements which are True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). According to the error rate quality requirement, we can consider *TP* to be the count of the cases that is correctly classified to be breaching the 10% threshold. Therefore, *TN* is the count of the cases that are correctly classified to be compliant and does not exceed the 10% threshold. *FP* and *FN* are the cases where the smart contract incorrectly identify cases as *breach* or *compliant*; respectively. Based on the Diagnostic Accuracy table, we select and define the following measurements:

- **Sensitivity:** The proportion of violation cases correctly classified by the smart contract; calculated as follows:

$$Sensitivity = \frac{TP}{TP+FN} \tag{4.6}$$

- **Specificity:** The proportion of compliant cases correctly classified by the smart contract; calculated as follows:

$$Specificity = \frac{TN}{TN+FP} \tag{4.7}$$

- **Positive Predictive Value (PPV):** The probability of accurate decision on violation cases; calculated as follows:

$$PPV = (\frac{TP}{TP+FP}) \times 100 \tag{4.8}$$

- **Negative Predictive Value (NPV):** The probability of accurate decision on compliant cases; calculated as follows:

$$NPV = (\frac{TN}{TN+FN}) \times 100 \tag{4.9}$$

The goal is to examine the smart contract ability to correctly classify the prepared cases into their corresponding groups; either *Violation* or *Compliance*. Algorithm 4 illustrates the conduct of the Diagnostic Accuracy method. First, We feed all prepared sets of cases into

Algorithm 4 Conducting Diagnostic Accuracy Method

Require: $TEST_CASES$
Ensure: $Sensitivity$, $Specificity$, PPV , NPV
1: $C = \{c : c \leq threshold\}$ ▷ Compliant cases
2: $V = \{v : v > threshold\}$ ▷ Violation cases
3: $TEST_CASES = \{C \cup V\}$
4: **for each** $tc \in TEST_CASES$ **do**
5: **assessCompliance** (tc) ▷ invoke the smart contract
6: **determine** whether decision is TP, TN, FP, or FN
7: **if** TP **then**
8: $TP_count++$
9: **else if** TN **then**
10: $TN_count++$
11: **else if** FP **then**
12: $FP_count++$
13: **else if** FN **then**
14: $FN_count++$
15: **end if**
16: **end for**
17: **Calculate** $Sensitivity$, $Specificity$, PPV , NPV

the smart contract. We fill in table 4.2 according to the outcome of the smart contract. That is, for each case we classify the result of the smart contract decision on every case to be one of the listed categories (TP, TN, FP, or FN). We conduct the calculation of *Sensitivity*, *Specificity*, *PPV* and *NPV* to find out about accuracy of the compliance assessment. We implemented this validation method (Diagnostic Accuracy) using Junit testing framework, and executed it for every development iteration on the smart contract with consideration to the failure test coverage in table 4.1.

After several improvement rounds on the smart contract, experimenting in the testing environment has been able to achieve optimum results as in table 4.3. However, moving the solution the the production stage (realistic blockchain deployment) revealed the fact that, transactions may experience a failure due to various reasons, such as due to a timeout. Thus. monitoring agents must have the ability to observe internal blockchain events relevant to the submitted transaction. For any failure, it must be able to retry until successful. This has been accommodated to achieve the optimum result as shown in Algorithm 5 in the following section. Additionally, as

Table 4.3 Optimum results of the Diagnostic Accuracy method

Sensitivity	Specificity	PPV	NPV
100%	100%	100%	100%

4.5.3. Emulation of IoT Scenario and Monitoring

This section moves from a testing environment to an emulated environment to affirm the viability of the proposed approach in context of IoT applications. Ideally, this evaluation would be conducted on an actual cloud provider to experiment with malfunctions and downtime of cloud-based IoT services. However, it is unfeasible due to ethical issues and the difficulty to cause a cloud service to fail at will. So instead, this experiment emulates both the monitoring tool and the cloud-based IoT component to mimic a production environment. The emulation implements the IoT-based healthcare scenario presented in section 4.2.1 and covers all main components of the proposed architecture in Figure 4.2. All of these components are developed in Java programming language, which is available as an open-source project on GitHub[7].

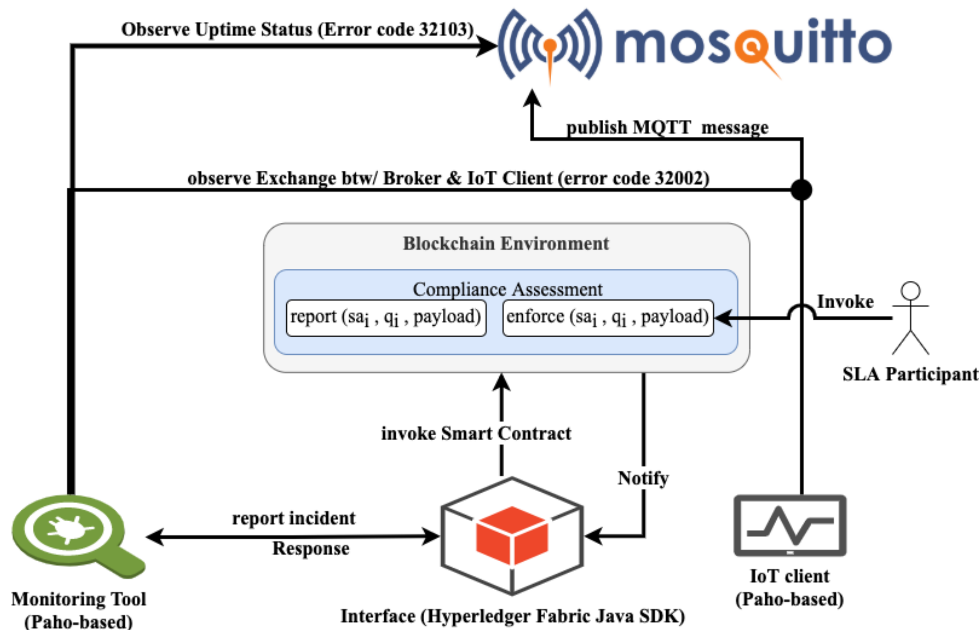


Figure 4.8 IoT application and monitoring for experimenting the compliance assessment smart contract.

Figure 4.8 depicts the implementation of the emulation environment, which consists of the following:

- **The assessment smart contract**: Implemented as proposed in this chapter and discussed earlier in this section 4.5.
- **An SLA-guaranteed cloud-based IoT component**: We employ an MQTT broker using *Eclipse Mosquitto*[8], to mimic the behaviour of the cloud-based IoT component. It maintains a set of topics and orchestrates communication between connected IoT clients.
- **IoT client**: We implemented an IoT client using *Eclipse Paho*[9], which is an MQTT client SDK. This client publishes MQTT messages to a topic called "emergency", notifying about critical patient health situations.

[7]https://github.com/aakzubaidi/MQTT-SLA-Blockchain-QoS-Enforcement
[8]https://mosquitto.org/
[9]https://www.eclipse.org/paho/

- **Monitoring tool**: It is also implemented based on *Eclipse Paho* because it complies with the MQTT specification and enables exploiting defined error codes and exception handlers provided. In particular, it uses the error code 32103, defined in the MQTT specification, to intercept an event where the SLA holds the service provider accountable for downtime, as per the quality requirement q_1. Moreover, it uses the error code 32002 to intercept events where the SLA holds the service provider accountable for a failed message, as per the quality requirement q_2. See footnote[10] for further detail on the error reason codes. The task of the monitoring is to ceaselessly observe connections between the IoT client and the MQTT broker. Whenever it intercepts either of these errors, it reports the incident to the smart contract and provides it with the respective SLA agreement key sa_1 and quality requirement q_i.

- **Blockchain Interface:** We implemented an API based on Fabric SDK[11]. This Interface facilitates authentication and communication between the monitoring tool and blockchain. In addition, the monitoring tool uses the Interface to submit incident transactions to the blockchain-backed smart contract and listen to events such as notification of enforcement tasks or SLA updates.

Algorithm 5 presents the experiment design conducted on the emulated environment. It emulates a billing cycle (e.g. every 30 days), in which IoT clients publish a random number of messages per day to the MQTT broker. The latter should be operative as quality as promised by the SLA. The evaluation assumes no fault by IoT clients since this study mainly focuses on the compliance of the emulated cloud provider towards the promised quality requirements. Thus, we deliberately ignore any other failure's causes such as network issues or incorrect configuration and so forth.

The monitoring tool observes both the status of the MQTT broker and messages exchanged between it and IoT clients. In order to deliberately cause incidents that violate the quality requirements q_1 and q_2, we design the following:

- **Uptime quality requirement** q_1: We simply take the broker down at random in order to cause a downtime event. This leads to emitting an error code 32103.

- **Error rate quality requirement** q_2: We manipulate the configurations of the MQTT broker (Mosquitto), by introducing a low timeout limit, which causes irregularities to the rate of successful and failed messages. A failed message of this sort returns with an error code 32002 .

The monitoring tool must accumulate the count of compliant and breach events. Once the monitoring identifies an incident, it automatically submits a transaction to the smart contract. The implementation considers featuring the monitoring tool with the ability to listen to blockchain

[10]https://github.com/eclipse/paho.mqtt.java
[11]https://github.com/hyperledger/fabric-sdk-java

Algorithm 5 Scenario Experiment: MQTT Broker Monitoring and Compliance Assessment

Require: *MQTT Messages*
Ensure: *Compliance Status*

1: $\Omega = \{compliant, breach\}$ ▷ MQTT Broker Status
2: *Query* SLA agreement sa_i
3: $\Psi = \{q_1, q_2\}$ ▷ QoS metric
4: **while** SLA is valid **do**
5: *Listen* to any sa_i amendment event and accommodate.
6: **for** *Specified Duration* **do** ▷ days
7: **for** *Random Iterations* **do**
8: $r \xleftarrow{R} \Omega$ ▷ Random Broker Status
9: **Publish** MQTT message
10: **if** $r = compliant$ **then**
11: c'++
12: **else**
13: b'++
14: **repeat**
15: **Report_Incident** (sa_1, Ψ_i, c', b')
16: **until** Sucessful
17: **end if**
18: **end for**
19: **end for**
20: **end while**
21: $result =$ **enforce penalty** (sa_1, Ψ_i, c', b')
22: $ViolationRate = (\frac{b'}{c' + b'}) \times 100$
23: **if** $result = ViolationRate$ **then**
24: Pass
25: **else**
26: Fail
27: **end if**

events regarding its submitted transaction. If a transaction failure occurs, it will retry until successful.

The experiment ensures the monitoring tool preserves a set of properties as follows:

- The monitoring tool is authorised.
- It maintains SLA awareness by querying SLA from the blockchain-side and no where else.
- It correctly identifies incidents.
- It reports if and only if incidents occur.
- It confirms the status of submitted transactions and retry in case of failures.
- Follow the specified reporting procedures, as per Algorithm 5.

For observatory purposes, we track the total count of both fail and successful MQTT messages labelled as c' and b', respectively. As Algorithm 5 illustrates, the experiment uses them to locally

calculate the violation rate and produce a known compliance assessment for comparison with the one produced by the smart contract. Therefore, we say that the experiment outcome is satisfactory and as intended if the smart contract assessment matches the locally produced one. The experiment is conducted several times, each time with a different use case as follows:

1. 100% Violation rate.
2. 0% Violation rate.
3. start with a normal operation and then cause violations until the end of the experiment.
4. start with abnormal operation and then then cause compliance until the end of the experiment.
5. exhibit irregular behaviour (compliant and violation) throughout the experiment.
6. terminate the SLA agreement during the experiment. The monitoring tool must halt as a result.

In each of these uses cases, the compliance assessment smart contract proves to behave as expected. However, the experiment observes and confirms that a failure in the monitoring mechanism can threats the validity of the proposed compliance assessment approach. Therefore, this pilot concludes that a robust monitoring mechanism is compulsory for the reliability of this proposed approach, which is a challenge left for future study. The experiment implementation is available on a public GitHub Repository[12].

4.6. Blockchain Benchmark and Results

The previous section looked into the viability of a decentralised incident management approach. However, we still need to confirm the feasibility of HLF as an underlying blockchain technology. In the literature, HLF has proven to perform well for several scenarios. However, to the best of our knowledge, there has been a performance experiment on a Hyperledger Fabric that implements the latest recommended consensus protocol; namely Raft. Refer to section 2.4.3 for further details and how this performance study is distinctive from others.

As we are interested in how well HLF is able to handle concurrent transactions, we simulate multiple monitoring agents that resemble the behaviour in Algorithm 3. They simultaneously report distinct incidents to the blockchain side with the aim to benchmark the performance. Therefore, this section investigates how HLF network copes with concurrent incident reports (assuming the worst scenario of a cloud provider). In particular, we look into HLF's latency and transaction success/fail rate. The previous experiment assumes only single monitoring agent. In this performance benchmarking, we employ concurrent monitoring agents to observe how the HLF would perform and how that would impact the overall solution.

[12] https://github.com/aakzubaidi/MQTT-SLA-Blockchain-QoS-Enforcement

Table 4.4 Hyperledger Fabric (HLF) network deployment configuration

Factor	Settings
Host Machine	Local running MacOS Catalina OS, 2.9GHz Dual-core intel Core i5 CPU, 2GB LPDDDR3 Memory Ram.
Containerization	Docker version 2.2, Engine version 19.3.5. Allocated Resources: CPUs: 3, Memory: 7GB, Swap: 3GB
Network Topology	HLF Version 1.4.6, 2 Organisations, 4 committing peers, 5 orderers, each one in a separate container. One dedicated channel, LevelDB is used as a ledger database.
Consensus Protocol	Raft Protocol
Endorsement Policy	One peer of each organisation
Chaincode Settings	
Execution Timeout	60 seconds
Programming Language	Java
Logging	Enabled

4.6.1. Experimental Setup

We focus on the most demanding functionality in our modelled smart contract, which is processing the received incidents, as described in Algorithm 3. At the infrastructure level, Table 4.4 illustrates both, the testing environment and HLF configuration.

We employ a blockchain benchmarking tool called Hyperledger Caliper for deploying the smart contract, benchmarking the performance, and simulating the behaviour of monitoring agents. Hyperledger Caliper is based on a set of performance measurements discussed in section 2.4.3. Table 4.5 summarises the experimental settings of 4 different test cases. They are all similar in terms of benchmarking settings, where we denote $\sim$ to indicate similarity. However, these test cases are different in terms block batching configurations. For this experiment, we assume consecutive transactions. Therefore, we aim to figure out a suitable block batching configuration that avoids concurrency issues while maintaining a reasonable throughput and latency. We employed two factors for defining the readiness of a block for validation, which are *timeout*, and maximum allowed number of *transactions per block*; whichever occurs first. These two factors vary in every testing, seeking the best performance possible.

Our testing plan is that, we set all parameters in Table 4.5 to the minimal values possible, and then we scale gradually until we reach a bottleneck. For example, we started from 1 worker sending only one transaction per second. However, we encountered conflicting transactions

Table 4.5 Experimental settings (Hyperledger Caliper)

Facet	Test 1	Test 2	Test 3	Test 4
Function under test	Report Violation (Asset update)	∼	∼	∼
Total workers (client thread)	1 worker	∼	∼	∼
Control rate: Fixed Rate	300 transactions	∼	∼	∼
Sent Transactions (per second)	1 Tps	∼	∼	∼
Execution Timeout	30 seconds for: - Chaincode engine - Worker - Caliper runtime	∼	∼	∼
Fabric Block Batching Configuration				
Block Batching: Timeout (milliseconds)	1000	500	1000	500
Transactions per block	10	10	1	1

Table 4.6 Results of Latency and Transactions success rate

	T1	T2	T3	T4
Success	298	297	299	295
Fail	2	3	1	5
Max Latency (s)	1.22	1.54	1.43	1.68
Avg Latency (s)	0.77	0.77	0.69	0.78
Min Latency (s)	0.68	0.68	0.77	0.34

for all different test cases as reported and discussed below. Therefore, we exclude the mission of testing concurrent transactions, as we already know that the issue of conflict transactions would only intensify. Thus, we limit the number of workers to 1 for all 4 test cases and be more focused on block batching configuration. We started with 1 second timeout and 10 transactions per seconds for test case 1. Then, we went to see how HLF would perform with lower values in the rest of test cases.

4.6.2. Results and Observations

4.6.2.1. Latency

All four test cases in Table 4.3 reveal that HLF latency, measured in seconds, is generally acceptable given the limited resources of the host machine. We observe no major difference in average Latency, which exhibits a relatively similar behaviour. We can see from case 4 that, these is a clear fluctuation between minimum and maximum latency. We can attribute this to the high rate of commits on the ledger across the network [87], resulted by generating a block every half second milliseconds. This leads many transactions to miss the chance of being included in the current block, and thus wait for the next one.

4.6.2.2. Success/Fail Rates

We have to recognise the fact that HLF employs an optimistic locking mechanism, namely: Multi-Version Concurrency Control (MVCC) to prevent a known blockchain problem called double-spending problem [122]. This mechanism caused all test cases to experience transactions failures due to conflict in MVCC Read-Write sets. The MVCC conflict is triggered because HLF experienced consecutive transaction reporting incidents to the smart contract. The evaluation results reveal that, test case 3, (1 Tps and 1s timeout) experiences the least transactions failures. The worst performance is observed in test case 4, where block batching is set to the minimum possible (1 Tps and 500ms timeout).

4.6.3. Observation and Remarks

While MVCC has proven to work well for some scenarios such as money transfer, it is not the case for demanding applications. For example, it is abnormal for a money transfer application to receive frequent updates on the same account within a few seconds. However, we ideally expect consecutive transactions from monitoring agents. Nevertheless, the lock mechanism causes some monitoring transactions to experience MVCC conflicting Read-Write sets, represented as (k, v, ver). When a transaction tries to update a key, it acquires the latest version of that key. Since our scenario assumes consecutive monitoring transactions, failure can accrue when transactions carry out update operations based on an obsolete key version. This situation can happen when an unsettled update transaction (not committed yet to the ledger) finally manages to be committed, causing a change of the current value and version. Therefore, any transaction based on an obsolete version is to be invalidated regardless of how correct they are [131].

There have been several workaround solutions to address this matter. For example, a composition of multiple keys can bypass MVCC validation because there is a new key generated for every transaction. However, this contradicts with the purpose of MVCC and introduces cumbersome key management. A retry mechanism is another workaround, such that client applications are notified of such conflicts and then retry submitting again. While applicable for some scenarios, this may not be suitable for others. There are also other techniques such as queuing transactions before submitting them to the blockchain. However, this does not benefit from the high throughput promised by HLF.

To sum up, in all test cases presented in table 4.5, the MVCC conflict was present, which is an unpleasant issue that has to be addressed. We do not expect such an issue to appear when incidents occur much less frequently. However, in certain extreme scenarios, we expect HLF to exhibit malfunctions due to MVCC conflicts. For example, when a very poorly performing cloud provider causes monitoring agents to report incidents every 1 second. Even when we attempt to increase the frequency of block batching, there is value in addressing and mitigating concurrency issues [87][95]. Nevertheless, Chapter 5 address this issue by proposing an enhanced compliance approach which demonstrates its effectiveness in mitigating MVCC conflicts.

4.7. Discussion

This section discusses the outcomes of the pilot study and highlights a set of observations, recommendations and threads to validity; as follows:

4.7.1. Real-time SLA Awareness

The SLA representation and awareness approach, proposed in Chapter 3, demonstrates to be effective for this pilot study in several ways. First, a monitoring tool can maintain SLA awareness while relying on trustable SLA assets immutably stored at the blockchain side. Furthermore, a monitoring tool can be engaged in real-time with any events on the SLA, such as amendment or termination. In conventional approaches, such events would require redesigning the entire solution due to deploying a new smart contract with a new address and functionalities. The SLA representation approach mitigates this challenge by separating SLA from data logic. As shown in Figure 4.3 monitoring tools only need to be aware of which SLA agreement to consider for monitoring of which cloud-based service. In terms of the SLA, the SLA manager notifies the monitoring tool to readjust its configurations (thresholds and triggers) accordingly to accommodate the latest update on the SLA. Concerning changes of the provisioned service, Kochovski et al.[118] show that it is possible to employ an orchestration system to help monitoring tools accommodate events at the infrastructure level. However, this point is beyond the scope of this study.

4.7.2. SLA Monitoring

Smart contracts are inefficient for conducting ceaseless operations such as SLA monitoring due to their unique aspects, such as terminability and determinism. For that, monitoring and altering tools are the key enablers for the blockchain-based compliance assessment approach. Therefore, monitoring tools must have the ability to communicate back and forth with the blockchain side. That is, selected monitoring must enable sending transactions to the blockchain side, for instance, by supporting Software Development Kits (SDKs) that facilitate wallet-based authorisation and authentication to access API endpoints exposed by smart contracts. It must also enable real-time configurability by exposing APIs consumable by the SLA manager's smart contract. Therefore, smart contracts can notify the monitoring tool about relevant blockchain events such as SLA termination or amendment.

We consider that smart contracts should exhibit an event-driven behaviour, and execute transactions in a terminable and deterministic manner [75]. Since monitoring metrics are volatile and changeable over the course of milliseconds, endorsing peers should be limited to receiving metrics from the monitoring-side rather than self-obtaining them. Otherwise, there is a high chance that, each smart contract replica will obtain distinctive data leading to an undesired malfunction. Therefore, endorsing peers must be limited to executing the smart contract only upon received transactions from the monitoring side.

The compliance assessment smart contract depend on metrics submitted as transactions from the monitoring side. We observe that submitting transactions to a blockchain platform is not a fire-and-forget operation. Transactions could face a temporary failure due to various factors, either internally due to the complexity of such systems or externally during transaction transit from the monitoring side to the blockchain side. Examples of temporary failures include, but are not limited to, network instability, execution timeout, authentication failure, absence of a corresponding blockchain peer (node), or a failure to satisfy a check mechanism such as MVCC (Multi-Version Concurrency Control) [127] and so forth. Accordingly, a transaction failure will not only causes a defect in the smart contract's functionality but can also negatively impact the reliability and performance of the overfall solution. For instance, we would expect a sort of deviation in smart contract calculations that conduct a compliance assessment operation. Unless there is a fail-safe mechanism in place, we would question the reliability output decisions made by the decentralised application. Section 6.3.2 accounts for these issues by applying a fail-safe mechanism.

Moreover, monitoring tools are expected to form a considerable burden to the performance of the blockchain. That is, smart contracts depend on metrics reported from the monitoring side. Thus, this study recommends employing customisable monitoring tools for the alerting and reporting mechanism as to when to trigger the smart contract. This is to control the transaction rate as much as reasonably practicable. Two critical factors to consider for setting the invocation trigger of the compliance assessment smart contract. First, monitoring tools do not impact the storage of every node participating in the blockchain network. They do not cause a bottleneck performance to the blockchain performance in terms of throughput or latency. Scheduling and queuing strategies can promise a smooth operation as intended.

4.7.3. Threats to Validity

While the proposed approach proved the possibility of mitigating the risk of authority abuse, there is still the need to address a set of issues. First of all, unlike SLA definition and negotiation, compliance assessment may face the challenge of handling a high rate of transactions submitted by monitoring tools, which causes transactions conflicts or pending status. However, the proposed approach considers the service provider to be compliant unless proven otherwise. This is limits interactions between monitoring tools and the blockchain side to the occurrence of incident events, which prevents overwhelming the blockchain network with unnecessary transactions. Therefore, the proposed compliance approach is designed accordingly.

Another threat to the validity is that the violation rate equation (see Equation 4.3) may not apply to every quality requirement. However, other equations can be employed to situate various quality requirements in future work. Moreover, the compliance assessment approach only considers the proposed SLA data model. However, a future study will examine its generality on distinctive SLA schemas.

The pilot study results reveal edge cases and failures that could not be discovered in testing environments, as in section 4.5.2. For example, Table 4.3 shows optimal results after several

iterations of development and testing. However, Table 4.6 demonstrates that stressing the underlying blockchain network reveals transactions failures which lead the proposed approach to miscalculate.

Finally, this pilot study concludes that while monitoring tools are necessary for the proposed compliance approach, they can pose a significant threat to its validity. This threat is because they can introduce a single source of failure. For example, if a monitoring tool neglects accommodating amendments on the SLA or may feed the smart contract with malicious data. Another example is when a monitoring tool fails to maintain reliable availability. Therefore, there is the need for a none-repudiable and highly-available monitoring mechanism. However, related works such as by Uriarte et al.[110] and Scheid et al.[20] all seem to accept this level of threat and, to some extent, accept trust in monitoring tools agreed by SLA parties. Furthermore, other studies propose mechanisms to minimise the risk of manipulating monitoring tools. For example, the work by Taghavi et al. [109] and the work by Zhou et al.[99] propose a game-based mechanism to prevent tampering with monitoring tools. Wang et al.[31] suggests the possibility feeding data to smart contracts via distributed oracles.

4.8. Conclusion and Future work

This pilot study explores a blockchain-based approach that aims to automate SLA compliance assessment and penalty enforcement with the aid of monitoring tools. It benefits and extends on Chapter 3 to achieve the proposed approach in the context of IoT. Given unique aspects of smart contract, such as immutability and difficulty to upgrade, it employs a validation framework to ensure the dependability of the proposed approach before deployment to the blockchain network. While testing methods proved so, blockchain benchmarking concluded that the MVCC conflicts can threaten its validity. This is particularly eminent when the smart contract handles a high rate of incidents per second from monitoring tools. Therefore, the issue of the MVCC conflicts has to be addressed. Otherwise, we cannot safely assume dependability. The following chapter enhances the proposed approach to work around MVCC conflicts and experiments. It also enhances the proposed approach to work around the MVCC issue and experiment at a larger IoT scale.

Chapter 5. IoT Monitoring and Enhanced SLA Compliance Assessment Approach

Summary

This chapter proposes a set of smart contract design considerations to improve the compliance assessment approach, previously presented in Chapter 4. It mainly addresses the issue of MVCC (MultiVersion Concurrency Control), which poses a reliability challenge when dealing with a high rate of incident transactions from the monitoring side. We experimentally evaluate the new enhancement and demonstrate to mitigate MVCC issues while maintaining clear performance improvements in transaction success rate, throughput and latency. Therefore, this chapter enforces the enhanced compliance approach on a more complex SLA covering a hypothetical IoT-based firefighting system with the aid of an enterprise-grade SLA monitoring mechanism, namely: (Prometheus[1]). This chapter utilises this monitoring tool to architect metrics instrumentation, collection, incident identification, and reporting the blockchain side.

5.1. Introduction

The previous chapter, Chapter 4, explored the potentiality of Blockchain for compliance assessment and penalty enforcement in the context of IoT. However, it narrowed the SLA coverage to IoT-based cloud services and thus limiting the scope of failure diagnostic and monitoring. In addition, it concluded that the MVCC protocol employed by the underlying blockchain platform (Hyperledger Fabric) could cause severe failure to the proposed compliance assessment approach.

Limiting interaction occasions between the monitoring and blockchain sides to incident events can potentially mitigate this threat. However, there is still the threat of an event where the monitored service experiences constant and long-lasting failures, which causes the monitoring tool to submit a high rate of transactions on the blockchain side. Such an event forces the issue of MVCC conflicts (read-write sets conflicts) to arise on the surface.

Therefore, the primary goal of this chapter is to extend Chapter 4, and encounter the challenge of MVCC conflicts by considering the following alternatives; as follows:

1. *at smart contract level*: improve the design of the proposed SLA compliance assessment approach.
2. *at middleware level*: develop a queuing and scheduling mechanism that resides between monitoring solutions and the blockchain side.

[1] https://prometheus.io/

3. *at Blockchain infrastructure level*: suggest an improvement that enhances or replaces the MVCC protocol.

This study resolves the MVCC issue by proposing a design improvement for the compliance assessment at the smart contract level. Subsequently, it discards the exploration of other alternatives either at the middleware or blockchain levels. However, further improvement at these levels could be necessary for other reasons, which is beyond the scope of this study. Accordingly, this chapter proposes an enhanced SLA data model and improves the design of the smart contract with the aim to mitigate MVCC issues encountered by Chapter 4. It also experiments on whether the enhanced compliance approach can maintain sound performance against a massive number of incident alerts submitted by monitoring tools. This chapter also enforces the enhanced compliance approach on a hypothetical End-to-End IoT system. For that, it designs and implements a simplified end-to-end IoT system and employs an enterprise-grade monitoring solution (Prometheus[2]). The IoT system is a fire mitigation system that automates detecting and alerting a firefighting station of fire events. This chapter assumes an SLA between the firefighting station (consumer) and an IoT provider who deploys and operates the end-to-end IoT system. Accordingly, this study examines possible failure cofactors that determines the compliance level with the assumed SLA. This study contributes the following:

- Mitigates MVCC conflicts with an enhanced Blockchain-based compliance assessment approach.
- Enforces the proposed approach on a more complex SLA that covers a hypothetical end-to-end IoT firefighting system.
- An evaluation and benchmarking of the enhanced compliance assessment approach.

Appendix B describes the implementation of the end-to-end IoT system and configurations of the monitoring solution. We also provide an open-source project in a GitHub repository[3], which includes the implementation of the employed IoT scenario, monitoring-side, blockchain-side configuration, smart contract logic and benchmarking as well as relevant data sets. Figure 5.1 summarises this study methodology, and the rest of this chapter describes the conduct of each step.

5.2. Preliminaries

This section provides a context for this chapter by describing a simplified end-to-end IoT-based firefighting system. Appendix B describes the implementation of the IoT system, which observes fire events and report them to the firefighting station. According to the implemented IoT system, this section describes the fire event journey from its origination until reported to the firefighting station. It also presumes a hypothetical SLA between a firefighting station and an IoT Service

[2] https://prometheus.io/
[3] https://github.com/aakzubaidi/BlockchainQoT

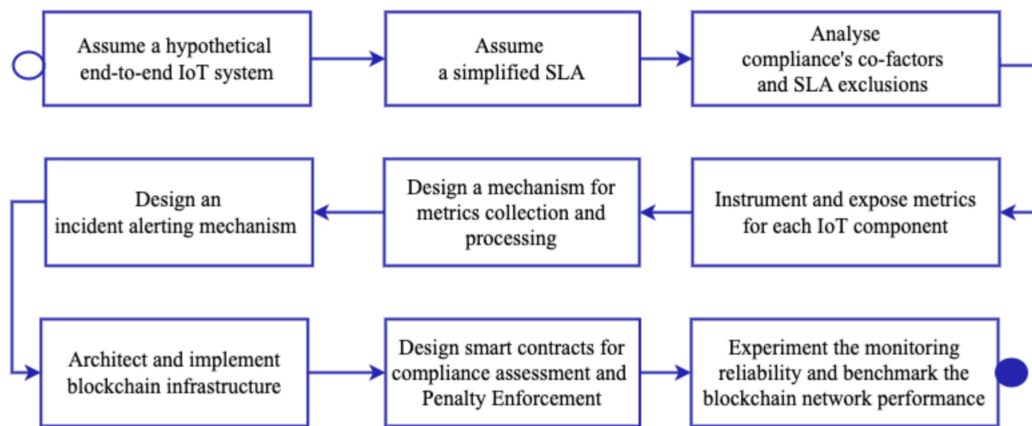


Figure 5.1 Research Methodology

Provider (IoTSP), which regulates their contractual relationship and governs quality requirements and violation consequences. Then, this section sheds light on co-factors that influence the rate of the IoTSP's compliance with the SLA.

5.2.1. Hypothetical IoT Scenario

Assume a contractual relationship between a firefighting station and an IoT solution provider, hereafter abbreviated as *IoTSP*. The firefighting station decides to embrace IoT for quicker response to fire events and severity mitigation. In order to alleviate the burden of dealing with IoT complexity, the firefighting station outsources IoT-related tasks such as deployment, operations and management to the IoTSP. In this scenario, outsourcing such tasks leaves the firefighting station only responsible for responding to fire events emitted by the IoTSP. Figure 5.2 depicts the responsibility of the IoTSP, which covers geographically dispersed sensors controlled by edge computing units that locally observe their environment in a real-time manner. The IoTSP also covers a centralised cloud-based IoT server that governs these field assets.

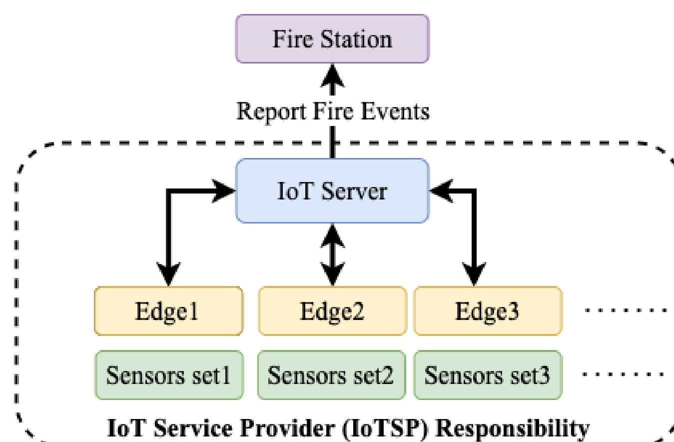


Figure 5.2 Overview of an IoT-based Fire Mitigation System. Edges 1,2, and 3 represent the edge computing nodes.

5.2.1.1. *Fire Event Journey*

Figure 5.3 conceptualises a simple sequence of stages for a fire event. Simply put, specialised fire detection sensors are deployed to observe flames within their range. These sensors periodically send collected data to their respective edge computing units. The latter analyses received data to identify whether it indicates a fire event. The edge computing unit must immediately notify the central IoT server of any identified fire events. When the IoT server receives an incident, it must not act instantly. Rather, it must allow a specified duration (e.g. 5 seconds) for a follow-up message from the edge unit about the same location. Meanwhile, one of the following cases may occur:

- The IoT server receives a *Discard* message from the edge computing unit, and thus no further action is taken.
- The IoT server receives a *Confirm* message, which immediately triggers a report of a confirmed fire event to the firefighting station.
- The IoT server receives neither a *Confirm* nor a *Discard* message within the specified wait time (timeout). Therefore, the IoT server must take precautionary action by self-confirming the initial fire event and reporting it to the firefighting station.

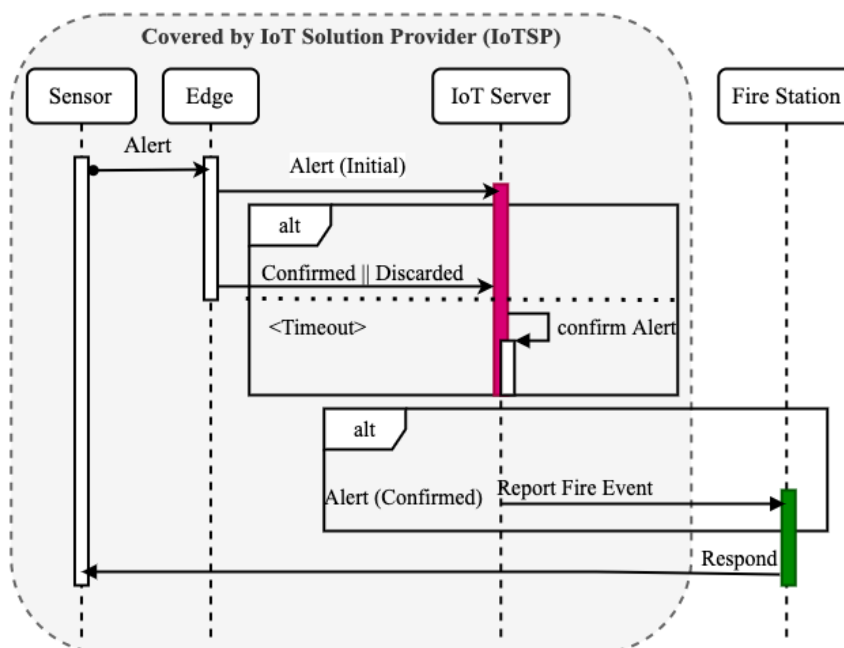


Figure 5.3 Stages of a fire event from origination until being reported

5.2.1.2. *SLA between IoTSP and Firefighter Station*

In light of the hypothetical IoT-based firefighting application, assume an SLA that governs the relationship between the IoTSP and the firefighting station, which obligates the former to comply with a set of quality requirements. For instance, the IoTSP must observe for fire

events $f \in F$ where F represents the set of fire events and report them to the firefighting station within a specified duration $(t_s \leq d)$. The firefighting station expects quality availability all the time, especially during a confirmed fire event. In this study, availability is not only limited to the IoT server but also extended to edge units. This quality requirement can be denoted as $A_{edge} \wedge A_{server} \neq down$. The SLA also specifies a set of breach categories BC, where it holds the IoTSP accountable for their consequences in case of violation. For example, consider three breach categories $bc_1, bc_2, bc_3 \subseteq BC$ as follows:

- bc_1: consider a situation where the IoTSP fails to report a confirmed fire event f due to a downtime of any covered component $(\neg A_{edge} \vee \neg A_{server})$, and therefore a failure to calculate the duration needed for processing and reporting the fire event, which can be denoted as $t_s = 0$, where it should be, instead, $0 < t_s \leq d$. Accordingly, a monitoring metric tuple M is classified as a breach of type bc_1 when it holds $(f,\ t_s = 0,\ (\neg A_{edge} \vee \neg A_{server}))$,
- bc_2: consider a situation where the IoTSP fails to maintain availability of the server or any edge computing unit. However, this case does not occur during a confirmed fire event f and thus it is less critical since $t_s = 0$ is perfectly normal and expected. Accordingly, M is classified as bc_2 when it holds $(\neg f,\ t_s = 0,\ (\neg A_{edge} \vee \neg A_{server}))$
- bc_3: consider a situation where the IoTSP maintains available components, and manages to process and report confirmed fire events f, but fails to do so within the specified duration where it should be $t_s \nleq d$. Accordingly, M is classified as breach of type bc_3 when it holds $(f,\ t_s \nleq d,\ (A_{edge} \wedge A_{server}))$

Other breach categories can be defined in a similar manner. Depending on the severity of each breach category $bc_j \subseteq BC \mid i \in \mathbb{N}$, the SLA defines a maximum tolerance mt to the violation frequency. Moreover, the SLA stipulates what penalty should be applied on the IoTSP if mt is reached. This done by tracking the violation rate vr for each bc_i, which is calculated as per Equation 5.1,

$$vr = \frac{\sum_i^n b}{\sum_i^n b + \sum_i^n c} \times 100 \tag{5.1}$$

where b is the count of breach cases and c is the count of compliant cases. As long as the violation rate (vr) does not exceed the assigned max tolerance $vr \ngtr mt$, the SLA validity remains intact $\lambda \leftarrow true$; however, penalties are enforced whenever applicable. Otherwise, the SLA is terminated $\lambda \leftarrow false$, and a full refund is issued to the consumer. Once the SLA is established, it declares the commitment of the IoTSP towards these promised quality requirements and violation consequences.

5.2.2. Architecture Overview

Assuming untrusted relationship between the firefighting station and IoTSP, we consider automating and operating distrusted processes within a blockchain environment such as compliance assessment and penalty enforcement. Figure 5.4 envisions the overall architecture where the

IoTSP's compliance level is under a continuous monitoring and examination against a set of promised Quality of Service (QoS) requirements. The architecture employs blockchain-based solution for SLA-related distrusted processes such as definition, compliance assessment, billing and penalty enforcement. Both chapter 3 and chapter 4 provide a detailed description of the SLA-related tasks. Additionally, refer to Appendix B for a description of the implemented IoT system.

To materialise the blockchain-based monitoring and compliance architecture, we consider two primary components, which are the monitoring-side and blockchain-side; discussed as follows:

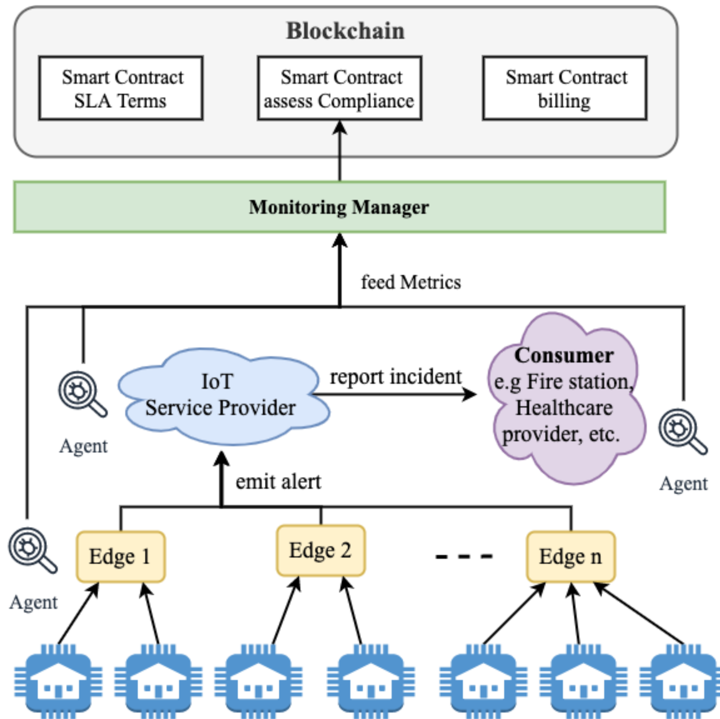


Figure 5.4 Motivating IoT scenario where blockchain is employed for SLA monitoring and enforcement

5.2.2.1. Monitoring-side

A monitoring mechanism is necessary for providing the awareness and visibility needed for executing SLA distrusted processes [10][55]. As illustrated in Figure 5.5, the monitoring side is responsible for metrics collection related to quality requirements stated in the SLA. For example, it ceaselessly observes fire events f and tracks their journey from the initiation stage at the edge level until the processing stage at the server level and then the reporting stage to the firefighting station. It also continuously observes the availability of both the edge and the server computing units. Whenever the monitoring manager encounters an incident that requires attention, it alerts the smart contract by submitting a transaction consisting of a set of collected metrics $M = (f, t_s, A_{edge}, A_{server})$, such that

- f indicates whether there was a confirmed fire event.
- t_s the duration it takes the IoTSP to process and report a confirmed fire event if any.

- A_{edge} and A_{server} are availability indicators of both edge computing units and the server.

In order to avoid overwhelming the smart contract with unnecessary interactions, the monitoring manager controls the alerting mechanism such that transacting with the smart contract occurs only in the event of a confirmed fire event f or a breach B. Identifying either of them will cause the monitoring manager to submit a transaction to the blockchain.

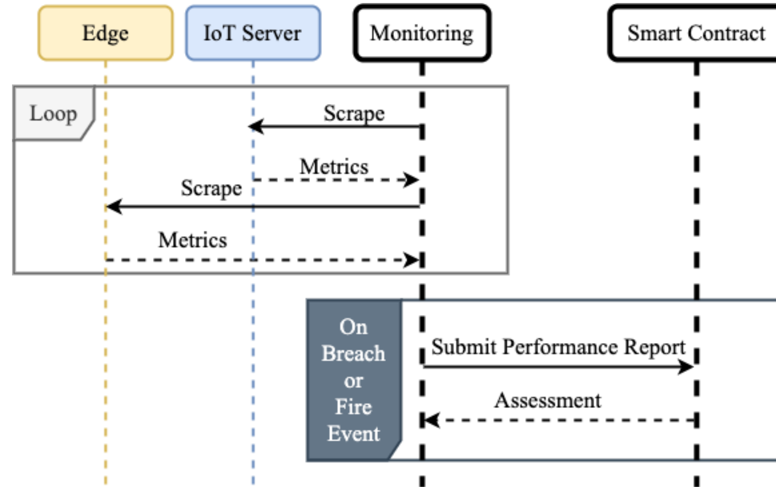


Figure 5.5 Metrics collection and reporting to the blockchain-side.

5.2.2.2. Blockchain-side

In this study, we employ Hyperledger Fabric (HLF) platform, which enables benefiting from several blockchain principles such as decentralisation, transactions immutability, consensus mechanism, and other blockchain features. Influenced by HLF philosophy, we consider a distributed system where involved parties construct a blockchain network and contribute to the infrastructure and computing resources.

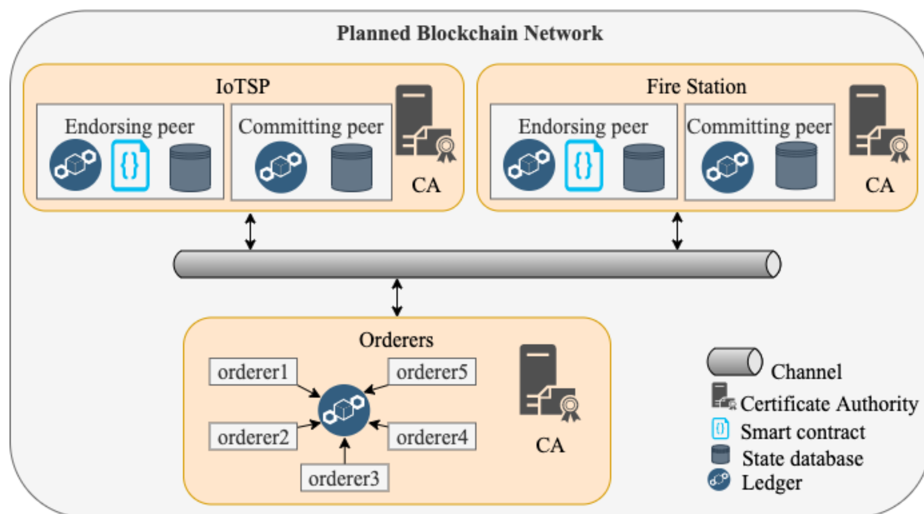


Figure 5.6 Hyperledger Fabric's blockchain network of two organisations: IoTSP and firefighting station.

As depicted in Figure 5.6, we consider at least two organisations, which are the firefighting station and the IoTSP. Every organisation hosts a set of peers for high availability. In this

distrusted environment, each participating organisation holds replicas of three essential elements, that are:

- A replica of the ledger; needed for committing and appending blocks of transactions.
- A replica of the state storage: needed for reflecting the latest state of persisted records.
- A replica of a set of smart contracts (Chaincode), which executes distrusted processes and acts as a gateway to the local state storage.

Other components of the blockchain architecture are highlighted in Section 2.4.

5.3. Monitoring Architecture and Considerations

Most distrusted SLA-related processes are of a decision-making nature, such as compliance assessment and penalty enforcement [15]. Transforming such processes into an autonomous decentralised application requires feeding them with relevant metrics from monitoring means [132]. By considering the presented SLA in section 5.2.1.2, this section examines which relevant co-factors that could impact the compliance rate of the IoTSP towards its obligations. It then describes the overall monitoring architecture, metrics collection, as well as reporting mechanism to the blockchain side. The ultimate goal of this section is to engineer a mechanism for metrics collection and reporting to the blockchain with the aim to account for failed transactions and avoid overwhelming the blockchain side with unnecessary transactions.

5.3.1. Determining Contributing Factors to Compliance Status

In order to determine the adherence level towards a quality requirement, we need to determine co-factors that influence the compliance status. For demonstration purposes, we consider the following quality requirements:

- $QoS(Availability_{e,c})$, where $e \leftarrow edge$ and $c \leftarrow IoTserver$. We assume a centralised server, and several geographically dispersed edge computing units.
- $QoS(t_s) \leq d$, which mandates the IoTSP to process and report a fire alert to the firefighting station within a specified duration.

Determining the IoTSP compliance level with the availability requirement is a relatively straightforward process. That is, a binary decision tree of $\{true, false\}$ can help determine the compliance level towards the availability of any covered IoT component (the server and edge computing units). However, this is not the case in terms of the second quality requirement, which relates to the transmission time of a fire alert from its origination until being reported to the firefighting station.

Consider a dispute that arises of whether the IoTSP fulfilled its duty in reporting a fire event within $t_s \leq d$. Following are some cases which can lead to a dispute regarding this quality requirement which are:

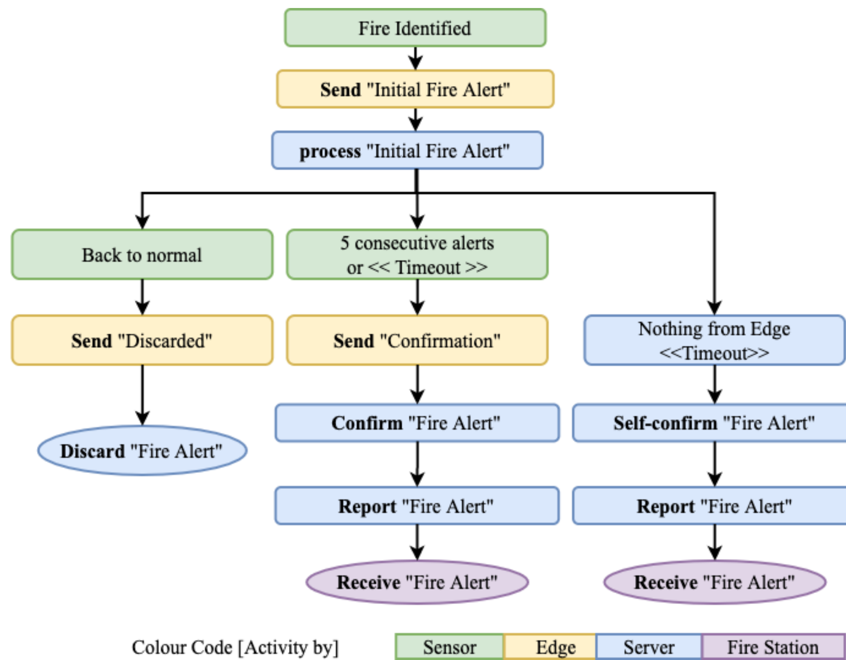


Figure 5.7 key stages for a fire event across different IoT layers.

- The firefighting station's system fails to log the fire alert once received.
- The IoTSP fails to satisfy $t_s \leq d$, but it claims otherwise.
- The IoTSP satisfies $t_s \leq d$; however, the firefighting station claims otherwise.

Therefore, we analyse the journey of a fire event from its initiation until being delivered to the firefighting station. This is to unambiguously determine what co-factors precisely determines the IoTSP's compliance level towards $t_s \leq d$. Based on Figure 5.3 and Appendix B, we identify three possible scenarios where fire events develop from the state of identified until being either discarded or reported to the firefighting station; as per depicted in Figure 5.7. These three scenarios are as follows:

1. *False positive fire alert*: It occurs when an edge computing unit issues an initial fire alert to the server and then follows up with a discard message during the waiting period. Accordingly, the server must discard and refrain from reporting it to the firefighting station.

2. *True positive fire alert*: It occurs when an edge computing unit issues an initial fire alert to the server and then follows up with a confirmation within the time limit (i.e. within five seconds). Accordingly, the server must immediately report the fire event to the firefighting station.

3. *Dangling fire alert*: It occurs when an edge computing unit sends an initial fire alert to the server; however, it fails to follow up with either confirm or discard messages within the time limit. Accordingly, the server assumes criticality at the edge side (e.g. fire damage); and thus report the fire event to the firefighting station.

Since this section focuses on co-factors contributing to the IoTSP compliance level, we can optionally omit the first scenario where fire events are classified as false positive and thus discarded. That is, the SLA obligates IoTSP to report fire events, which leaves the other two scenarios where fire events end up confirmed and reported to the firefighting station either because the edge computing unit issues a confirmation or because the IoT server self-confirms it for a precautionary reason.

For both of these scenarios, a fire alert undergoes a total transmission time as in Equation 5.2, where T measures the actual transmission time of a fire alert from its origination (an edge computing unit) until being delivered to the firefighting station.

$$T \leftarrow t_s + t_r \tag{5.2}$$

$$t_s \leftarrow t_{reported} - t_{identified} \tag{5.3}$$

$$t_r \leftarrow t_{ack} - t_{reported} \tag{5.4}$$

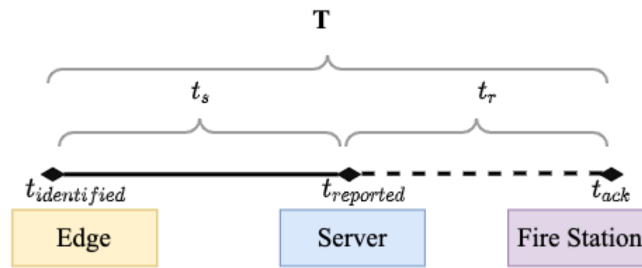


Figure 5.8 Timeline for fire event development.

Figure 5.8 illustrates the total transmission time T forms the total of two main elements, which are as follows:

- t_s refers to the duration that takes the fire alert from being issued at an edge computing unit $t_{identified}$ until being reported by the server $t_{reported}$ (calculated as per Equation 5.3).
- t_r refers to the rest of the fire alert journey, which is the duration that takes it from being reported by the server $t_{reported}$ until being finally delivered to the firefighter station (calculated as per Equation 5.4).

Figure 5.3 assigns the IoTSP with the responsibility of both the server and the edge computing unit. Subsequently, we can draw attention to t_s which determines the compliance level of the IoT towards the quality requirement $t_s \leq d$. On the other hand, the SLA understandably does not cover t_r because it can be subject to several factors beyond the immediate control of the IoTSP (e.g. Internet routing delay) or issues at the firefighting station system. For that, monitoring must not only aligns with quality requirements but also with SLA executions [9]. However, the blockchain-based solution can be designed to keep records of both t_r for auditing and dispute resolution purposes.

5.3.2. Monitoring Architecture

Figure 5.9 illustrate a monitoring and alerting architecture based on a well-established open-source project, namely Prometheus[4]. Reasons for this selection are summarised in [133], which include, but are not limited to,

- It is hosted by the Cloud Native Computing Foundation (CNCF) [5] and enjoys wide adoption and community support in terms of documentation, maintenance, integration tools and libraries.
- It adopts a pull approach for metric collection, in which target entities (edge, IoT server, firefighting station system) can export relevant metrics via REST APIs to be scraped by the monitoring manager.
- The Prometheus's overall architecture considers high availability, replication, and fault-tolerance.
- Supports flexible query language, namely PromQL, for defining rules and querying thresholds and alerts. it also provides a rich set of libraries and instrumentation tools for exporting relevant metrics from the targeted instances (application, containers, infrastructure, services, etc.)
- Employs an alerting system that can be automatically triggered based on predefined conditions.

Using Prometheus, this study instruments a set of relevant metrics for each component (edge, cloud, application). It exposes these metrics via REST APIs. The monitoring manager collects and aggregates exposed metrics and stores them in a time-series database based on a set of rules. The *Alert Manager* regulates the alerting mechanism and uses a query language (PromQL) to define what thresholds to trigger associated smart contracts. There is a component called Fabric REST Server that facilitates communication, authentication, and interaction between Prometheus (monitoring/alerting system) and smart contracts on the blockchain side. Prometheus manager also enables outsourcing metrics from different components to a visualisation tool such as Grafana [6] for analytics and insights that we need for experimental purposes. A full implementation and configuration set are provided in our GitHub Repository [7]. The following sections delve further into the design and implementation of these components.

5.3.3. Metrics Instrumentation and Exporting

As shown in Figure 5.9, both the monitoring manager and alerting mechanism depend on metrics exposed from each component of the IoT ecosystem. On the one hand, Prometheus's exporters

[4]https://prometheus.io/
[5]https://www.cncf.io/cncf-prometheus-project-journey
[6]https://grafana.com/
[7]https://github.com/aakzubaidi/BlockchainQoT/tree/main/monitoring

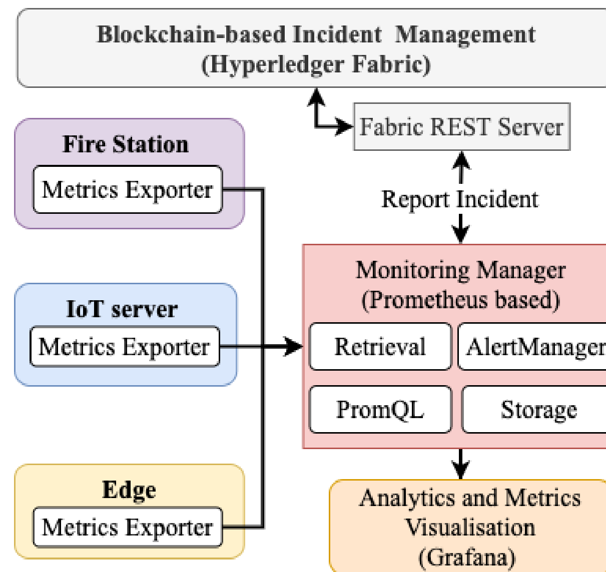


Figure 5.9 Employing Prometheus monitoring tool for feeding metrics to the Blockchain-side.

enable instrumenting and exposing relevant metrics from each component via REST API. On the other hand, the monitoring manager regularly collects metrics from components covered by the IoTSP (edge and sever) and the firefighting station system.

It is noteworthy that various IoT components can be deployed to different locations of distinctive timezones. For example, the firefighting station possibly deploys its system to a data centre that differs from the IoTSP server or edge computing units. Hence, there arises the possibility of different timezones. As Figure 5.9 depicts, there is a metric exporter which resides at the location of each component and thus is subject to the employed timezone settings of the respective component. Consider the fact that the calculation of t_s or t_r depend on timestamps from different timezones. To prevent unintended miscalculation, we employ a Unix timestamps system, which is a standardised time representation and timezone independent. Therefore, each exporter instruments and composes metrics using this Unix timestamp system.

The Prometheus exporters are provided in the GitHub repository[8]. Following, we illustrate the instrumentation and exposure of relevant metrics at every component in the example IoT ecosystem.

5.3.3.1. Edge-side Exporter

As per discussed in Section 5.3.1 and presented in figure 5.8 edge computing units are responsible for identifying fire events. Therefore, the Prometheus exporter composes and exports the metric $t_{identified}$ at the edge side. Note that, Equation 5.3 deems $t_{identified}$ as an essential element for evaluating the IoTSP's adherence towards the quality metric t_s. Moreover, edge computing units are responsible for confirming fire alerts. Therefore, we use $t_{confirmed}$ to assert whether and when the edge computing unit was able to confirm a fire event.

[8]https://github.com/aakzubaidi/BlockchainQoT/tree/main/IoT

Figure 5.10 illustrates the logic of instrumenting and exposing both $t_{identified}$, which indicates when a fire event was first identified, and $t_{confirmed}$ which indicates the time of confirming the fire event. It uses the following conventions:

- f' an initial fire alert.
- f a confirmation of a fire event

Provided that there is a capable device at the edge-side such as Raspberry PI4, a Prometheus exporter can be deployed to compose and expose relevant metrics via a REST API for collection by the monitoring manager. For example, once the edge computing unit identifies a fire event and sends an initial alert f', the Prometheus exporter assigns a timestamp to $t_{identified}$ and then exposes it for collection. Afterwards, the Prometheus exporter allows a delay to observe whether the edge unit confirms the fire event. When the fire event is confirmed f, it assigns $t_{confirmed}$ a timestamp to be exposed for the monitoring manager. Otherwise, it rests $t_{identified}$ to zero, which can indicate when the edge unit declares the fire event as a false positive.

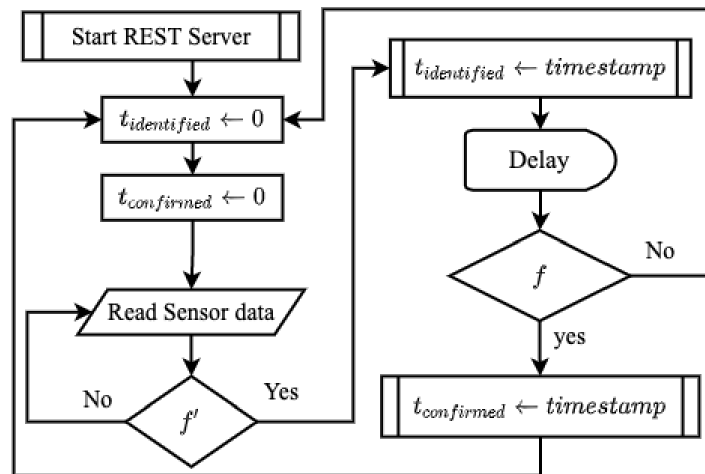


Figure 5.10 Instrumenting and exposing relevant metrics at edge level.

5.3.3.2. Server-side Exporter

Recall that the IoT Server reports fire events to the firefighting station only when they are confirmed either by the edge or self-confirmed by the server for precautionary reasons (refer to section 5.3.1). That is why we do not only expose $t_{confirmed}$ from the edge side, but also the IoT server-side as well. Figure 5.11 illustrates the logic of exposing relevant metrics from the server-side, which captures when the fire event f is confirmed $t_{confirmed}$ and reported $t_{reported}$. Note that, Equation 5.3 deems the latter as the second essential element for evaluating the IoTSP's adherence towards the quality metric t_s.

Moreover, note that $t_{confirmed}$ metric can be assigned a timestamp by either the exporter at the edge side or the one at the IoT server. This measure is in place to account for natural disaster at the edge side which can cause a downtime to the edge computing unit and its Prometheus exporter as

well. In this case, the IoT server self-confirms the fire event. Therefore, its Prometheus exporter assigns a timestamp for $t_{confirmed}$ metric to indicate when the fire event is deemed true positive.

The Prometheus exporter rests all metrics to zero in two cases:

- *False positive*: the edge unit sends a "Discard" message.
- *True positive*: the fire alert is confirmed by either the edge unit or the IoT server itself. However, it fails to report it within the specified period.

Moreover, we track whether and when the firefighting station receives the reported fire event t_{ack}. While the latter does not contribute to evaluating IoTSP's compliance, it is exposed and collected for assertion and auditing purposes.

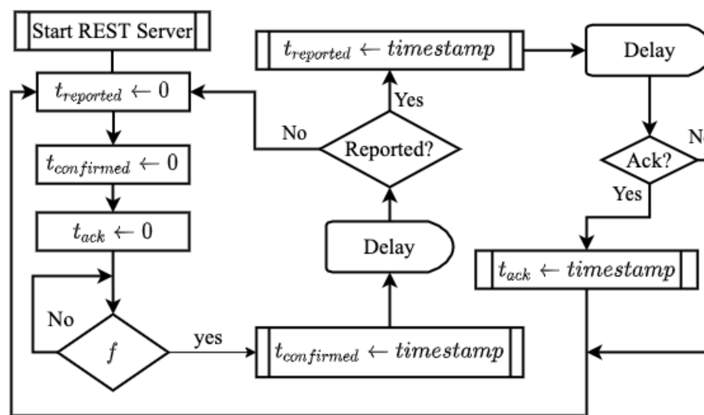


Figure 5.11 Instrumenting and exposing relevant metrics at server level.

5.3.4. Metrics Collection

Since Prometheus adopts a metric pull mechanism, the monitoring manager regularly collects and analyses exposed metrics and decides where there is an incident to report to the blockchain side (See Figure 5.5). As per the SLA, Figure 5.12 visualises relevant collected metrics such as availability/downtime of covered IoT components, which are Edge-side and Server-side, as per Figure 5.12a. It also shows different fire states (confirmed fire f or no fire $\neg f$), as well as when a confirmed fire was identified $t_{identified}$ and reported to the firefighting station $t_{reported}$. For the sake of an example, we assume a quality requirement $QoS(t_s \ngtr 3)$ in order to cause deliberate breaches for experimental purposes.

The monitoring manager does not only collect metrics but also regulates when to report the IoTSP's performance to the blockchain side. As shown in Figure 5.5, the IoTSP's performance is reported either on the occasion of a confirmed fire event f or a breach to a quality metric B, which can be due to unavailability of an edge computing unit $A_{edge} \leftarrow down$, unavailability of the server $A_{server} \leftarrow down$ or a breach to $t_s \ngtr d$.

Algorithm 6 illustrates the procedure of metrics analysis, providing the following:

- The unavailability of any component, edge or server, implies a breach case that triggers the compliance evaluation. The algorithm exempts the edge unavailability as in line 8

which does consider it in a breach of the availability requirement unless there is no fire event $f = false$. In other words, it exempts edge computing units from the availability requirement in case of natural disaster caused by a confirmed fire event $f = ture$.

- $t_{confirmed} \in \mathbb{N} \mid t_{confirmed} > 0$ indicates a confirmed fire event f, which triggers the compliance evaluation. This metric is provided by both sides edge and server.
- In case of a confirmed fire event f, the monitoring manager examines whether the IoTSP reports the fire alert to the firefighting station. If so, it then examines the IoTSP's compliance towards $t_s \not> d$ in accordance with Equation 5.3, which is the duration consumed by the IoTSP for processing and reporting the confirmed fire event, as shown in Figure 5.8.

Algorithm 6 Reporting Mechanism to the Blockchain Side

Require: $B \vee f$ ▷ B for Breach whereas f for fire event
Output: $Performance\ Report$ ▷ sent to Blockchain
1: **repeat**
2:   **Monitoring Manager:** scrape Metrics
3:   $A_{edge} \xleftarrow{R} \{up, down\}$ ▷ Edge Availability
4:   $A_{server} \xleftarrow{R} \{up, down\}$ ▷ Server Availability
5:   **if** $A_{server} \leftarrow down$ **then**
6:     $B \leftarrow True$
7:   **end if**
8:   **if** $(A_{edge} \leftarrow down \wedge f = false)$ **then**
9:     $B \leftarrow True$
10:   **end if**
11:   **if** $t_{confirmed} \neq 0$ **then** ▷ fire event? positive value other than 0 indicates a fire event
12:     $f \leftarrow True$
13:     **if** $t_{reported} \neq 0$ **then** ▷ reported? positive value other than 0 indicates successful
14:       $t_s \leftarrow t_{reported} - t_{identified}$
15:       **if** $t_s \not< d$ **then** ▷ Breach of specified duration
16:         $B \leftarrow True$
17:       **end if**
18:     **else**
19:       $B \leftarrow True$
20:     **end if**
21:   **end if**
22:   **if** f $\leftarrow True \vee B \leftarrow True$ **then**
23:     $M = (f, t_s, A_{edge}, A_{server})$
24:     **Alert Manager:** report M to Blockchain-side
25:   **end if**
26: **until** End of SLA

5.3.5. Validating the Monitoring Approach

This section particularly focuses on validating the monitoring part of this architecture which includes the following:

Table 5.1 Classification and summary of metrics covered by Algorithm 6

f	Transmission Time Metrics			Availability Metrics		Incident?
true	$t_{identified}$	$t_{reported}$	t_s	Edge	Server	
true	>0	>0	$\leq d$	up	up	No
true	>0	>0	$\leq d$	down	up	No
true	>0	0	>0	up	down	Yes
true	>0	>0	$>d$	up	up	Yes
false	0	0	0	up	up	No
false	0	0	0	down	up	Yes
false	0	0	0	up	down	Yes
false	0	0	0	down	down	Yes

- Metrics instrumentation and exposure from each covered IoT component (edge and cloud).
- metrics collection from by the monitoring manager.
- Alerting system, which is used for triggering the compliance assessment smart contract.

Timestamping is the essence of each instrumented metric discussed above. Consider Figure 5.4, which depicts the overall architecture of a blockchain-based IoT monitoring and compliance assessment. Refer to Appendix B for further detail on the IoT system implementation. For simplicity, we use Digital Ocean [9] to deploy both the IoT server application and the firefighting systems to different virtual machines located at different regions of distinctive timezones. The edge computing unit is deployed using a Raspberry PI4 at a distinctive region and timezone as well.

The specification of each virtual machine is irrelevant because the aim of this section is to validate whether the monitoring approach functions as it should. This is particularly important given the geographically despaired deployment of each IoT component where the Internet is the only possible way for these IoT components to connect and communicate over HTTP protocol. A Prometheus exporter is attached for each IoT component to expose relevant metrics. As a result, metrics exporters are influenced by the disparity of their associated IoT components regarding the timezone difference and the need for an internet connection. The monitoring manager is also deployed to another cloud instance of different regions and timezone and thus needs to reach each Prometheus exporter in order to collect exposed metrics.

Table 5.1 summaries metrics used for validating the monitoring approach (see Algorithm 6). Note that each metric combination has a different degree of criticality violation severity. Therefore, each one of them should be assigned a different maximum tolerance rate mt as discussed in section 5.2.1.2. Regardless, Table 5.1 maps each combination of these metrics to a decision of whether to consider it as an incident that leads to triggering the compliance assessment smart contract.

For experimental purposes, we deliberately cause incidents to observe whether the monitoring can correctly classify them; and thus trigger the alerting mechanism. The designed incidents are as follows:

[9] https://www.digitalocean.com

- To cause an edge unit downtime, we intentionally disconnect from the internet to halt its operation.
- To cause downtime to an IoT server or firefighting system, we simply halt the execution of the deployed application or shutdown or suspend the cloud virtual machine.
- To cause a fire alert, we expose the flame sensing module to extreme light or fire (See figure B.3a).
- To cause a breach to t_s, we calculate the average time of processing and transmitting a confirmed fire event f from its origination until being reported to the firefighter station, which resulted in 3 seconds. Therefore, we assign the quality requirement $t_s \leq 3$ *second*, which causes any reading below to be considered a breach.

Figure 5.12b depicts a visualised sample of metrics collected by the monitoring manager. To visualise the collected metrics, we design a dashboard using Grafana and PromQL language. The dashboard shows the ability of the monitoring manager to constantly collect metrics from various exporters and identify incidents as well. For instance, it shows that the server maintained a constant uptime until it experienced a brief downtime, roughly between 5:40 am and 5:50 am. Regarding the edge unit, it shows a constant uptime expect three occasions as follows:

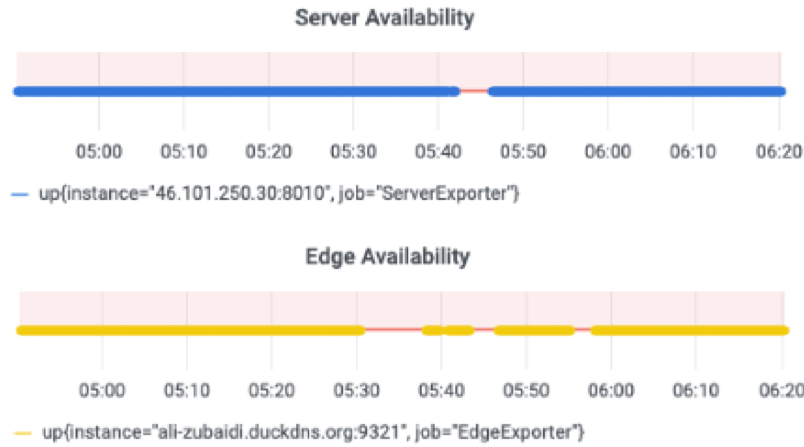
- Before the server expedience a downtime ($A_{edge} \leftarrow down$ while $A_{server} \leftarrow up$).
- During the server downtime ($A_{edge} \leftarrow down$ while $A_{server} \leftarrow down$)
- After the server resumed an uptime status ($A_{edge} \leftarrow down$ while $A_{server} \leftarrow up$)

The dashboard also depicts the journey of various fire alerts, as follows:

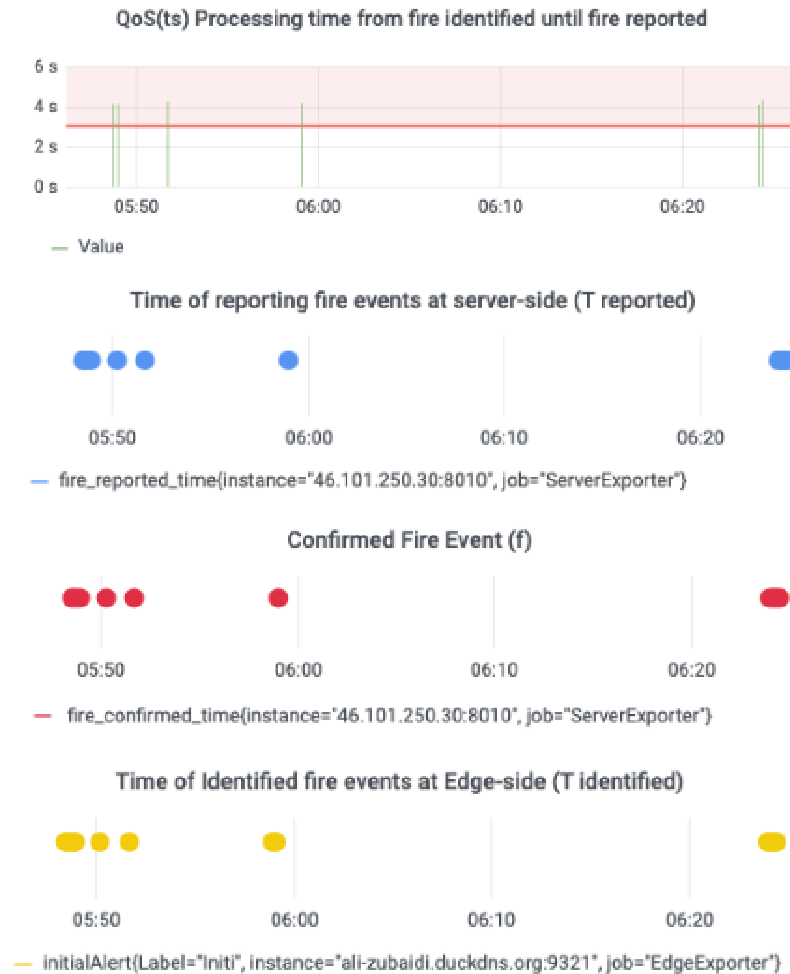
1. **First stage**: timestamps of when fire events are being identified at the edge side (Yellow colour).
2. **Second stage**: timestamps of when the fire alert is being confirmed (Red colour).
3. **Third stage**: timestamps of when the fire alert is being reported (Blue colour).

The dashboard also maps the IoTSP's performance in terms of t_s, the duration it takes for processing and reporting each fire alert. As the dashboard shows, we calibrated t_s to 3, which the IoTSP fails to challenge at the fire events. Therefore, the red area of the ts in the dashboard indicates a breach by the IoTSP regarding t_s.

To sum up, the monitoring approach has proven to work properly and reliably for this study. Furthermore, it also demonstrates the correct operation of the implemented IoT because the monitoring precisely reacted as per actions conducted on the IoT system. Accordingly, we can consider the monitoring approach reliable for triggering the smart contract compliance assessment, as discussed in the following sections.



(a) Availability indicators of covered IoT components



(b) collecting metrics of $t_{identified}$, $t_{reported}$ and f, as well as calculation of t_s

Figure 5.12 A screenshot of metric collection and Incident Identification

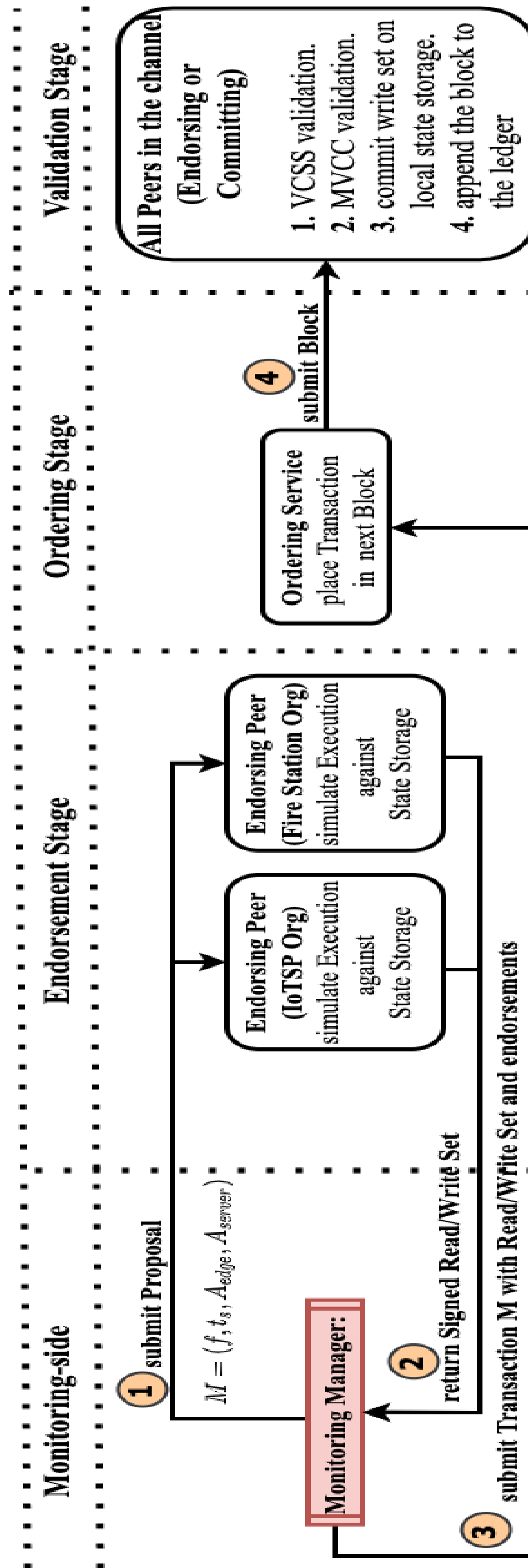


Figure 5.13 Transaction Execution Journey within HLF.

5.4. Enhanced Compliance Assessment Approach

This section proposes an enhanced smart contract design for metrics evaluation and SLA compliance assessment. It does not only promise to resolve MVCC conflicts but also maintains reasonably higher throughput and less latency. First, it highlights how the MVCC protocol can impact a high rate of transactions from the monitoring side. Then, it proposes an enhanced SLA data model and an improved smart contract design to encounter the challenge of MVCC conflicts. The enhanced compliance assessment is enforced on the hypothetical IoT-based firefighting scenario approach with the aid of the above-discussed monitoring mechanism.

5.4.1. Smart Contract Invocation

In this chapter, we consider a monitoring manager that interacts with the blockchain-side. Whenever it encounters an incident that requires attention, it invokes a respective smart contract's method and passes a set of metrics M, where $M = (f, t_s, A_{edge}, A_{server})$, as described in section 5.2.2.1. Unlike conventional applications, no blockchain operation is considered to be valid, unless it undergoes a set of validation mechanisms such as ESCC (Endorsement System Chaincode), VSCC (Validation System Chaincode) and MVCC (Multi-Version Concurrency Control [29] (see Figure 5.13). Section 2.4.2 provides further details on a typical transaction execution journey within Hyperledger Fabric.

5.4.2. MVCC Impact on High-Throughput Transactions

Different blockchain platforms apply distinctive schemes to mitigate the double-spending problem. For example, HLF employs the MVCC mechanism to control records consistency by tracking version changes of a record in the form of $(key : value, version)$. As depicted in Figure 5.14, whenever there is a transaction T that causes an update operation to a record, there is a read set $(k : val, ver)$. Based on this read set, a write set $(k : val', ver')$ attempts to update the state storage. However, before applying and committing the write set, the MVCC mechanism checks whether version ver of the read set is applicable. Otherwise, the version could have been changed due to another transaction T_1 that managed to be committed. In this case, the version of the read set will be classified as obsolete. Therefore, T_2 fails when it tries to commit the write set [86].

The MVCC mechanism can pose a challenge to high-throughput applications where multiple read-write sets are the norm, and double-spending is of no issue. For example, in our case, a high rate of transactions expected from the monitoring side would typically cause multiple read-write operations on blockchain records. However, such transactions may highly likely face Read-Write sets conflicts due to the MVCC mechanism [127].

Chapter 4 attributed this issue to multiple update transactions that happen to update the same asset while landing on the same block. By investigating the issue of MVCC conflicts, it appears that one transaction would succeed the MVCC validation, while the rest eventually fail due to a version change caused by that successful transaction. Table 4.6 shows the results of

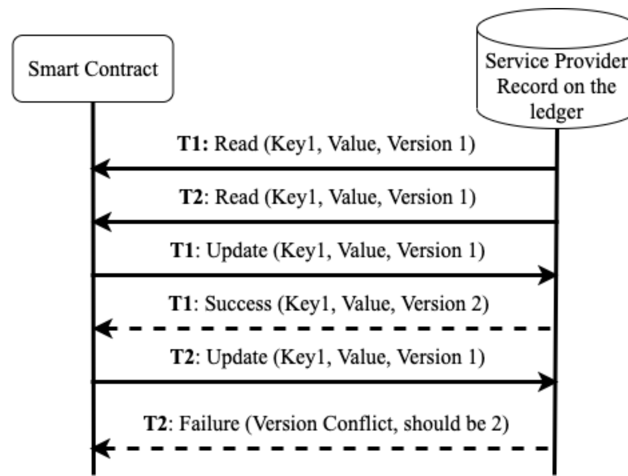


Figure 5.14 Read-Write set conflicts caused by multiple transactions updating the same record.

an experiment that attempted to experiment adjusting the blockchain batching configuration to prevent or reduce the chance of read-write conflicts. However, the experiments reveal that this practice was at the cost of performance and solution viability. Moreover, it does not completely mitigate MVCC issues. Therefore, this chapter enhances both the SLA data model and the design of the smart contract to address conflicting read-write sets.

5.4.3. Enhanced Compliance Data Model

By studying the impact of the MVCC protocol on the compliance assessment, we found that the design of both the smart contract and the data model plays a vital role in mitigating Read-Write set conflicts. Therefore, this section proposes an enhanced compliance assessment approach based on a simple but effective, which essentially prohibits update operations on performance records $pr_i \in PR$ such that $(k : val, ver) \mid \Delta ver = 0$. This is to eliminate changes on records versions, which theoretically mitigates the possibility of MVCC conflicts. In addition, instead of updating an existing performance record at the occurrence of each incident, the enhanced approach redesigns the solution to create a new pr_i for each incident and then aggregate them at the end of every billing cycle. The following sections delve further into implementing the enhanced compliance assessment approach.

The compliance assessment is enforced on the IoT ecosystem with the aid of the monitoring mechanism discussed in the above sections. For that, this chapter adjusts the SLA data model presented previously in Figure 3.6 for this study. Figure 5.15 presents an enhanced SLA data model, which accommodates the following:

5.4.3.1. Breach Categories

The previous SLA data model considers one violation consequence vc_i for each quality requirement q_i, which situated simple SLA agreements that may cover a single IoT component such as the example presented in section 4.2.2. However, it does not fit the purposes of a more complex SLA agreement that covers an end-to-end IoT system such as the one covered by section 5.2.1.2.

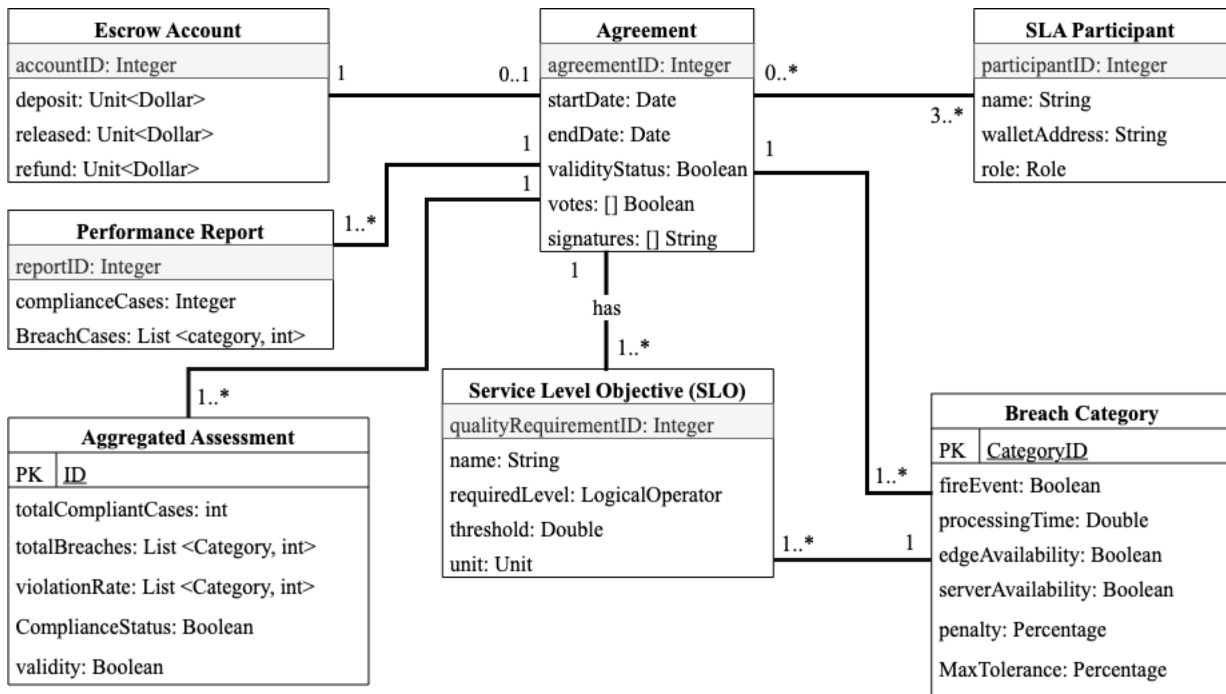


Figure 5.15 Enhanced Data Model for Evaluation Compliance over Blockchain.

The complexity is drawn from the fact that the SLA, presented in this chapter, covers various breach categories $bc_i \in BC$, such that each of them is based on a combination of quality requirements as shown in Table 5.1. Therefore, the enhanced SLA data model replaces the component of violation consequences with another one called *Breach Category*, which enables defining various types of breaches $bc_i \in BC$ based on multiple quality requirements. Observe Figure 5.15 which associates every $bc_i \in BC$ to multiple quality requirements. Moreover, the SLA data model enables assigning distinctive max tolerance and penalty to every bc_i. Each bc_i has a different criticality level and, therefore, distinctive consequences.

5.4.3.2. *Performance Reports*

The enhanced SLA data model adjust the performance data model to accommodate various breach categories in the form of $[bc_j : b]$, where bc_j is a unique identifier of a breach category, and b is the count of its occurrence. The performance report no longer serves periodic assessment. Rather, the smart contract can create as many performance reports as long as there are incidents reported by the monitoring tool. Section 5.4.4 further elaborates the logic of smart contract with performance reports. Therefore, the performance report does not consist of the properties *validity* or *compliance status*. Refer to Figure 4.4 to observe the difference. The enhanced SLA data model moves these properties to another component called *aggregated assessment*.

5.4.3.3. *Aggregated Assessment*

The enhanced SLA data model introduces a new component called *aggregated assessment*. For each billing cycle (monthly), the smart contract can create a new aggregated assessment instance to aggregate all existing performance reports. As can be seen in the enhanced SLA data model, it

accumulates the total count of compliant cases c and count of breaches b of each breach category bc_j. It also holds the violation rate vr for breach category against the total compliant cases c. The smart contract also uses the *aggregated assessment* to determine the overall compliance of the service provider. The validity property reflects the relevancy of this assessment to the current billing cycle.

5.4.4. Processing Received Monitoring Metrics

Recall that the monitoring manager submits transactions to the blockchain-side upon the occurrence of an incident of either B or f. Refer to Algorithm 6:Line 24, which reports a payload of collected metrics in the form of $M = (f,\ t_s,\ A_{edge},\ A_{server})$. The integration between both sides, the blockchain side and monitoring side, considers and benefits from the lessons learnt from integrating monitoring solutions with the blockchain (see section 4.7.2).

Based on the enhanced SLA data model, in Figure 5.15, this section addresses the limitations of the previous compliance assessment approach (see chapter 4). Algorithm 7 overviews a smart contract method for processing and evaluating received metrics M. As long as the SLA is valid $\lambda = true$, it accepts transactions from the monitoring tool and evaluates received metrics M against the respective quality requirements. As a result, the evaluation process classifies the performance of the IoTSP to be either compliant c or one of the predefined breach categories bc_i. Examples of breach categories are provided in section 5.2.1.2 and illustrated in Table 5.1.

For every metrics evaluation, the smart contract creates a new performance record in the form of a tuple $(k+1, pr, ver)$, where

- $k+1$ is a unique identifier of the performance report.
- pr is performance report that holds the result of a metric evaluation against breaches categories $bc_i \in BC$.
- ver is version that tracks modifications on the performance record.

We consider pr to consist of $(\ c\ ,\ [bc_j:\ b])$ where c indicates the count of compliant cases, and $[bc_j:\ b]$ indicates a the frequency of an incident belonging to a breach category, where bc_j is a unique identifier of a breach category, and b is the count of its occurrence. Example of evaluation outcomes are as follows:

- *Compliance case*: $pr \leftarrow (\ 1\ ,\ \emptyset)$.
- *Breach case*: $pr \leftarrow (\ \emptyset\ ,\ [0002:1])$.

As Algorithm 7 demonstrates, we opt to avoid the practice of updating an existing performance report (k, e', ver'). Instead, the proposed design dictates that there must be a newly created record $(k+1, e, ver)$ for every subsequent metric evaluation process. Once an evaluation record is created, it shall never be updated but may only be used for query purposes. In this way, we ensure there will always be one write operation, and therefore the version ver would not change at all. This perpetually mitigates the issue of conflicting read-write sets associated with the high rate of monitoring transactions per block.

Algorithm 7 Evaluation of Received Monitoring Metrics

```
Require: M = (f, t_s, A_edge, A_server)
Output: (k : pr)                                   ▷ Evaluation Performance Report
 1: bc_j ⊆ BC | i ∈ ℕ_>0                           ▷ Breach Category
 2: λ ← true                                       ▷ SLA Active
 3: k ← 0                                          ▷ key of Performance Report
 4: repeat
 5:     if M ≠ ∅ then
 6:         k++
 7:         if M ∈ BC then
 8:             pr ← (sp_i , ∅ , (bc_j : b))        ▷ breach case
 9:         else
10:             pr ← [sp_i , c , ∅]                 ▷ compliant case
11:         end if
12:         create (k : pr)                        ▷ create Performance Report
13:     end if
14: until λ = false                                ▷ SLA Termination
```

5.4.5. Compliance Assessment and Enforcement

The smart contract can be instructed to periodically conduct an overall compliance assessment. We consider that *read operations* do not cause version modification, and thus we do not expect MVCC conflicts. Therefore, the compliance assessment process should theoretically have the ability to aggregate all existing performance records at once $(k:\ pr) \mid i \leq k \leq n \mid k \in \mathbb{N}$.

Algorithm 8 illustrates considering a set of performance records for the compliance assessment; for example from k_i to k_n. The overall aim is to calculate the violation rate *vr* of each breach category $bc_j \in BC \mid j \in \mathbb{N}$ by examining its ratio against the total compliance cases. For that, the smart contract examines every $(k_i:\ pr)$ and query its properties $(\ c\ ,\ bc_j:\ b))$. It aggregates the total count of compliance cases, denoted as tc. Additionally, for every breach category $bc_j \in BC$, it aggregates the total count of breach cases, denoted as tb. Afterwards, the violation rate *vr* of every bc_j is calculated as per Equation 5.1 and examined against the respective max tolerance *mt*. Note that the total compliance cases cannot be $tc \nless 1$ to prevent division on zero in case of no breach cases.

The smart contract can take actions based on the outcomes of the compliance assessment. For example, the smart contract can determines to terminate the SLA if the violation rate *vr* exceeds the max tolerance *mt* of any breach category bc_j (refer to Algorithm 8 Line 19). This leads the smart contract to issue a full refund and halt further metrics evaluations because Algorithm 7 does not process any incidents if the SLA is terminated. Otherwise, the smart contract can the aggregated assessment to make an informed decision on whether to enforce a penalty on the escrow account. Finally, the smart contract removes all processed performance records for the state storage to avoid reusing them for the next aggregated assessment. However, they still remain permanently stored on the blockchain for future auditing purposes.

Algorithm 8 Concluding Assessment and Enforcement Logic

Require: a set of $(k: pr) \mid i \leq k \leq n \mid k \in \mathbb{N}$ ▷ Existing Performance Reports
Output: Decision
1: $\lambda \leftarrow true$ ▷ SLA Active
2: $tc \nleftarrow 1$ ▷ Total Compliance Cases
3: $\forall bc_j \exists tb \leftarrow 0$ ▷ Total Breach Cases
4: $\forall bc_j \exists vr \leftarrow 0$ ▷ violation Rate
5: $\forall bc_j \exists mt$ ▷ Max tolerance to violation rate
6: $\forall bc_j \exists Penalty$ ▷ penalty
7: **for each** $(k : pr)$ **do**
8:     **if** $true$ **then** ▷ compliant record?
9:         $ts += c$ ▷ add count of compliant cases
10:     **else**
11:         **for each** bc_j **do** ▷ every identified breach type
12:             $tb += b$ ▷ add count of violation cases
13:         **end for**
14:     **end if**
15: **end for**
16: **for each** $bc_j \in BC$ **do** ▷ Each Identified Breach Category
17:     $vr = \frac{tb}{tb+tc} \times 100$ ▷ Calculate Breach Rate
18:     **if** $vr > mt$ **then** ▷ Max Tolerance reached?
19:         $\lambda \leftarrow false$ ▷ Terminate SLA
20:     **else**
21:         Apply *Penalty* on the escrow account.
22:     **end if**
23: **end for**
24: **if** $\lambda = false$ **then**
25:     Issue full refund to consumer.
26: **else**
27:     Release remaining amount to service provider, if any.
28:     Refund to consumer, if any.
29:     Remove all processed performance reports from the state storage.
30: **end if**

Table 5.2 Blockchain deployment and configurations.

Element	Description
Hyperledger Fabric	Fabric version (2.3.2)
Blockchain Network	See Figure 5.6
Blocks Frequency Configuration	- Transactions per Block: 10. - Timeout: 1s. - No size restrictions.
Smart Contract Language	Java
Chaincode Timeout	30 seconds
Benchmark Tool	- Hyperledger Caliper V0.4.2 - 5 workers
Consensus Protocol	Raft
State Storage	CouchDB
Resources Allocation	- 32 x vCPU Intel(R) Xeon(R) Gold 6140 @2.30GHz - 64GB RAM.
Operating System and Docker	- Ubuntu Linux 20.04.2 (64-bit). - Docker Version 20.10.6 (No restrictions on resources usage).

5.5. Experiment and Evaluation

The purposes of the experiment is to evaluate whether the proposed smart contract design proves to mitigate MVCC conflict issues while maintaining sound performance. Therefore, we experiment and stress the proposed approach to investigate in terms of throughput and latency. More specifically, this experiment is concerned with two tasks assigned to the smart contract, which are metrics evaluation and SLA compliance assessment as in Algorithm 7 and Algorithm 8; respectively.

Table 5.2 illustrates the deployment of the blockchain network as well as relevant configurations and specifications. We choose to deploy the blockchain network, as in figure 5.6, on cloud infrastructure, as specified in Table 5.2. We experiment on the latest HLF version, as of writing this thesis, and adopted the recommended consensus protocol; namely RAFT [80]. All default parameters of the test network provided by HLF remains intact except the block batching configurations. We employ a blockchain benchmarking tool called Hyperledger Caliper for experimenting the performance of the enhanced compliance assessment approach. Hyperledger Caliper is based on a set of performance measurements discussed in section 2.4.3. The experiment considers the following:

- For each transaction execution, Algorithm 7 conducts a limited number of read operations (e.g. query quality requirements) and a single write operation (creating evaluation record). However, this algorithm is expected to be invoked very frequently; whenever the monitoring-side encounters an incident that requires attention. According to [97], using 5 worker for experimenting high rates of transactions seems to produce realistic results. We

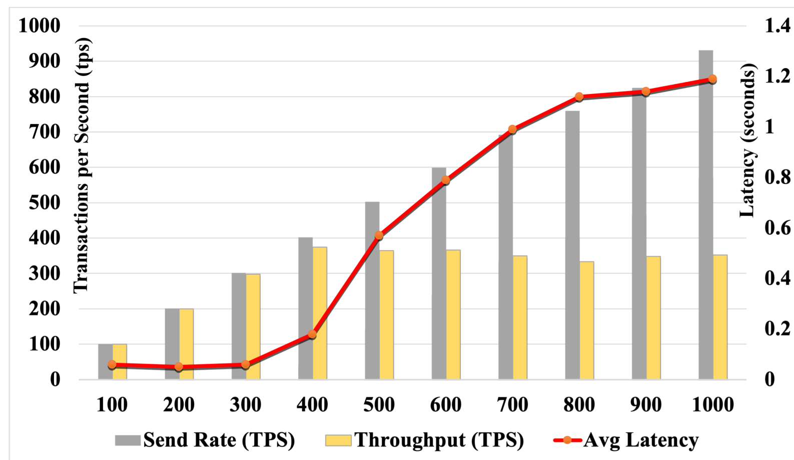


Figure 5.16 Algorithm 7 performance: processing received metrics at variable rates and fixed total of 1000 transactions.

validated that from our side, and thus we use 5 workers for submitting transaction from the monitoring-side.

- For each transaction execution, Algorithm 8, conducts a massive number of read operations on existing evaluation records, as explained in section 5.4.5. It then results in a limited number of write operations which can include persisting the compliance assessment on the ledger, and handling the escrow account for enforcement purposes. Noteworthy mentioning that, this algorithm is only executed on limited occasions (i.e. monthly billing).

5.5.1. Fixed Total Transactions and Variable Rates

For Algorithm 7, we examine how it performs under various rates of transactions per second. We set 10 rounds, as in Figure 5.16, where we fix total transactions to 1000. However, we increase the transaction submission rates by 100 Tps (Transactions per second) for each subsequent round. The aim is to investigate for each test round: (i) the send rate can be generated from the monitoring-side; (ii) the average throughput that can be archived at the blockchain-side; and (iii) the average round-trip transactions latency. We aim also to find out whether any of the transaction would encounter unforeseen failure such as MVCC conflicts.

As shown in Figure 5.16, for all test rounds, there was no transaction failure at all. The send rate tends to be identical to the intended transactions rate until the seventh round, after which the send rate gradually exhibits a modest degradation. With regard to throughput,the benchmark shows an identical throughput to the send rate for the three first rounds. Thereafter, we observe an increasing transaction processing time and thus less throughput compared to send rate. Both studies in this[96] and [92] find that the increasing queue of transactions waiting for VSCC validation may explain this phenomena. In a study by [87], the long queue of transaction can be also attributed to a delay within the blockchain network. Nevertheless, the throughput flattened out for the rest of test round at approximately 380 Tps, with unremarkable changes. In a similar manner, the latency remains very low without major difference for the first 4 test round.

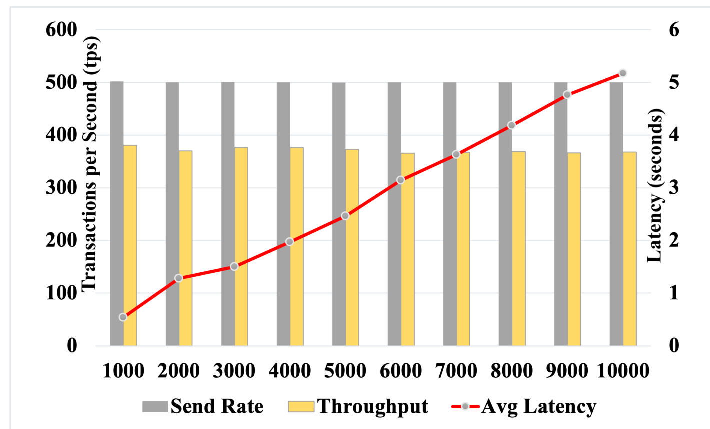


Figure 5.17 Algorithm 7 performance: processing received metrics at fixed rate of 500 transactions and variable total transactions.

Thereafter, it exhibits the possibility to break beyond 1 second, which is the max timeout set for block batching.

5.5.2. Variable Total Transactions and Fixed Average Rate

To verify our outcomes, we fixed the send rate to 500 Tps, which is more than the best achieved throughput from the above benchmark. We relaxed the total transactions with a minimum of 1,000 and maximum of 10,000, where we increase 1000 transactions for each test round. The aim to see whether this relaxation would achieve better throughput or does it have a negative impact on it. We also investigate how to this relaxation can be correlated to latency. As shown in Figure 5.17, there is no significant disruption in the throughput rate for all test rounds as long as the send rate is the same. Nevertheless, we observe an overall linear latency increase influenced by the linear increase workload of transactions. Consider the last round of Figure 5.16 with first round of Figure 5.17 , were all try to submit 1000 Tps but at different send rates. we observe that the latter achieve better latency than the former, which confirms a positive correlation between the send rate and expected latency [92] [96] [93].

All in all, the experiment reveals the ability of Algorithm 7 to accommodate a high rate of transactions without encountering MVCC conflicts. It also promises a sound performance, given the complexity of the smart contract logic and the blockchain configurations. We also report that, the experiments altogether did not consume more than 15% of the allocated resources, as illustrated in Table 5.2.

5.5.3. Compliance Assessment Execution Time

Periodically, the smart contract aggregates and consumes a set of evaluation records for compliance assessment as illustrated in Algorithm 8. As shown in Figure 5.18, we examine average latency of conducting the compliance on linearly variable number of stored records; increased by 1000 record for each round. For each round, we only need one transaction to trigger the compliance assessment, and thus we only focus on average latency and omit throughput. We observe that it exhibits a linear increase as a response to the amount of stored evaluation records.

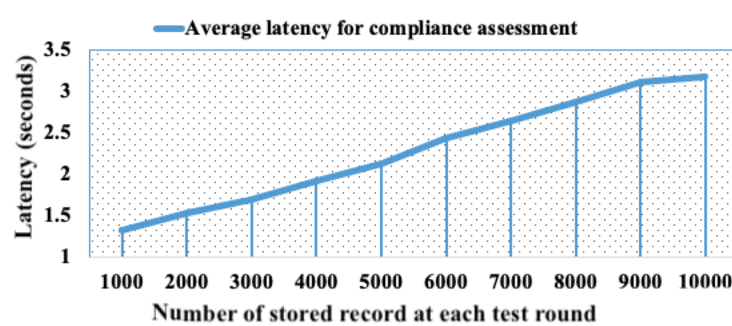


Figure 5.18 Latency for executing Algorithm 8 on variable collection of evaluation records stored on HLF state storage.

Overall, the smart contract proved to execute 10,000 evaluation records within no more 3.5 seconds. We deem this to be satisfactory, given that it is a non frequent task conducted occasionally, for billing and concluding purposes, over a massive number of records. We also do not normally expect such number of records unless there is a breach B or a fire event f, as specified per section 5.2.2.1.

5.6. Conclusion and Future Work

This chapter presented an enhanced blockchain-based enhanced compliance assessment approach in the context of IoT. It designs a monitoring mechanism that collects metrics related to an SLA that covers a hypothetical IoT-based firefighting scenario. It examines and discusses critical aspects and considerations at both sides of the argument, monitoring-side and blockchain-side. While conventional software design strategies have proven to work well for centralised applications, we cannot safely assume the same in the context of distributed blockchain applications. As demonstrated in this work, it is vital to consider unique blockchain characteristics when designing a blockchain-based solution, such as transaction processing, execution behaviour, configurations and implemented protocols.

This chapter draws attention to high rates of transactions emitted from the monitoring-side, and aims to resolve MVCC conflicts while maintaining sound performance. From the monitoring-side, it demonstrates how we can determine the most critical co-factors of relation to SLA compliance assessment. It designs a monitoring architecture and the reporting mechanism which which does not only account for possible failed transactions, but also engineers a mechanise for metrics collection and incident reporting with aim to avoid overwhelming the blockchain-side with unnecessary transactions.

From the blockchain-side, the enhanced compliance assessment approach revolves around a simple, yet effective principle, which segregates between read and write operations at both levels; the smart contract design and data representation at the state storage. This has paved the way for future work to investigate improving the performance at HLF infrastructure level. For example, finding optimal block configurations, which plays a vital role on throughput and latency. HLF modularity makes it also interesting to study the impact of different HLF's aspects on the overall performance, such as network size in terms of organisations, endorsing and committing peers.

There is also the ordering service and employed consensus mechanism, chaincode configurations, smart contract programming languages and others.

Chapter 6. Blockchain-based Simulation Middleware for SLA Monitoring and Benchmarking

Summary

This chapter is motivated by the need for IoT simulators for experimenting with blockchain-based SLA solutions. For instance, the previous chapter 5 conducts its experiments on a real IoT scenario implemented in Appendix B. However, the IoT system is limited and does not represent the large scale and complexity of IoT systems in the industry. This limitation poses a challenge for research and development purposes. While existing IoT simulators, see section 2.1.2, may compensate for this shortcoming, they are not SLA-aware and do not provide a monitoring emulation. Above all, they are not readily capable of transacting with real blockchain platforms such as Hyperledger Fabric, given the distinctive nature of their execution environment. Subsequently, this chapter investigates the potentiality of bridging this gap by proposing a generic blockchain-based middleware that enables utilising existing IoT simulators for experimenting and benchmarking blockchain-based SLA solutions. This chapter assumes that the middleware can interface with any IoT simulators, under the assumption that the IoT simulators are written in Java. This chapter validates the middleware approach by re-implementing the IoT scenario in the previous chapter using an IoT simulator called *IoTsim-Osmosis*. Then, the proposed middleware is in place to utilise the simulated IoT model for experimenting and benchmarking the enhanced compliance assessment.

The rest of the chapter is organised as follows: Section 6.2 illustrates the context of the research, by setting the IoT scenario as a stimulating example and making use of SLA monitoring. It also gives a brief overview of Blockchain-based monitoring for Service Level Agreement (SLA) purposes. Section 6.2.2 provides the problem and challenges of adopting a simulated model to experiment and benchmark. Section 6.3 provides an overview of the proposed approach and delve into the formulation of the proposed logic for mentoring SLA. Section 6.4 presents the verifies way of the core functionalities of the middleware scales to our predefined expectations. Section 6.5 provides evaluating the middleware with regard to bridging the gap between an IoT simulator and a real-world blockchain platform. Section 6.7 provides instrumentation of essential measurements such as throughput, latency, success and failure rates.

6.1. Introduction

Realising a blockchain-based SLA solution for IoT purposes through empirical research and development requires access to both ends, blockchain platforms and IoT infrastructure. Con-

cerning blockchain, most existing platforms tend to be open-source software deployed with reasonable hardware requirements. Moreover, rented cloud instances can compensate for the limited resources of local machines. Thus blockchain platforms are easily accessible for research and experimental purposes. However, gaining access to a large-scale IoT infrastructure can pose a real challenge for research and development.

For instance, this study is interested in experimenting with blockchain-based SLA solutions in the context of IoT. Chapter 5 implemented a blockchain-based compliance assessment and deployed it to a real-world blockchain network, namely, Hyperledger Fabric. However, it did not have access to large scale IoT infrastructure, which limited the experiment to a few sensors and edge computing units (see Appendix B). This limitation hinders reliable and effective research conduct. For example, an IoT-based firefighting scenario would properly assume a large number of connected homes, sensors, complex networks and communication systems.

Nevertheless, several use cases can leverage IoT simulators in order to compensate for this shortcoming [32]. For example, IoTSim-Osmosis [33] can be used to model a large-scale IoT architecture that does not only include physical things but also edge computing units, complex networks, data centres, and cloud services. Therefore, generated workload from the IoT simulator can be leveraged for experimenting and benchmarking blockchain-based SLA solutions.

While interesting, it can be a hurdle for researchers to bridge the gap between a real-world blockchain and a simulated IoT environment, which one will need to address before commencing a research effort on any blockchain-based IoT solutions in general, and SLA-specific projects in particular. For instance, consider addressing differences in the execution nature between real-world blockchain platforms and simulated IoT tools and handling communication and connectivity between both sides. Due to our prior experience with the use of Hyperledger Fabric for SLA purposes [34] [35], we select it to represent the blockchain-side. While there are several IoT simulators such as IoTSim-Osmosis [33] and IFogSim[49], none is well-suited for transacting with Blockchain, nor are provided with the logic of monitoring and alerting mechanism.

Hence, the contribution of this chapter is a generic middleware architecture that enables integrating Java-based IoT simulators of choice with Blockchain-based SLA solutions. In order to enable the integration between IoT simulated models and real blockchain networks, the proposed middleware architecture addresses the gap between their distinctive execution environments. The middleware equips simulators with a blockchain-based monitoring mechanism, which monitors simulated metrics and enables connectivity and communication with blockchain-based SLA solutions. Therefore, it is possible to experiment with a blockchain-based SLA solution, such as the compline assessment, using a simulated IoT model that sufficiently represents a large IoT infrastructure. Furthermore, the proposed middleware enables utilising simulators to benchmark essential blockchain performance metrics related to deployed smart contracts such as throughput, latency, transactions' success/fail rates.

This chapter validates the correct behaviour of the middleware and demonstrates its usage by integrating IoTSim-Osmosis simulator [33] with Hyperledger Fabric [29] for SLA monitoring

and enforcement purposes in the context of IoT. We also provide a reference implementation of the proposed middleware as an open-source project in a GitHub Repository[1].

6.2. Simulators for Experimenting Blockchain-based Solutions

6.2.1. Hypothesis

The implementation and deployment of a decentralised application can be feasible due to the open nature of most blockchain platforms. For instance, this thesis has demonstrated the use of Hyperledger Fabric for realising various blockchain-based SLA tasks such as the enhanced SLA compliance assessment approach (refer to section 5.4). Nevertheless, it can be difficult to access a large-scale IoT infrastructure to implement an IoT scenario such as the firefighting scenario presented in Figure 5.4. To appreciate the access difficulty, assume a scenario where an experiment aims to observe and evaluate the compliance assessment approach with a massive number of connected homes (i.e. 30,000) that simultaneously emit fire events.

In such a case, it is commonly acceptable to employ simulators whenever it is impractical to access real-world systems [32]. Consequently, experimental studies can leverage IoT simulators for modelling IoT systems. Several existing simulators enable modelling simulated IoT systems and produce out-of-the-box generated metrics (i.e. throughput, latency, CPU usage, memory consumption, etc.) or enable modelling metrics of choice. Examples of IoT simulators are covered in section 2.1.2.

For this study, IoTsim-Osmosis[33] can model a hypothetical IoT-based firefighting scenario and produce a set of metrics (i.e. $transmission_{time}$). Therefore, we see an opportunity in utilising the simulator's generated metrics for examining the performance of the service provider against agreed quality requirements (i.e. $transmission_{time} \leq d$), where d refers to a time in seconds. Moreover, it is possible to exploit generated output of the simulator for experimenting and benchmarking the deployed blockchain-based SLA solution.

6.2.2. Motivation: The Gap between Simulators and Blockchain

Figure 6.1 roughly illustrates a basic workflow for utilising existing IoT simulators to experiment and benchmark blockchain-based SLA solutions (i.e. SLA compliance assessment). The workflow assumes the following:

1. The ability of the simulator to communicate and transact with the blockchain side.
2. The simulator's awareness of defined quality requirement at the blockchain side (e.g. $transmission_{time} \leq d$).
3. A monitoring mechanism that gathers generated metrics from the simulator and evaluates them against defined quality requirements.

[1] https://github.com/aakzubaidi/BMBmid-Middleware

4. An alerting mechanism that triggers smart contract based on predefined conditions (e.g. incident alert).

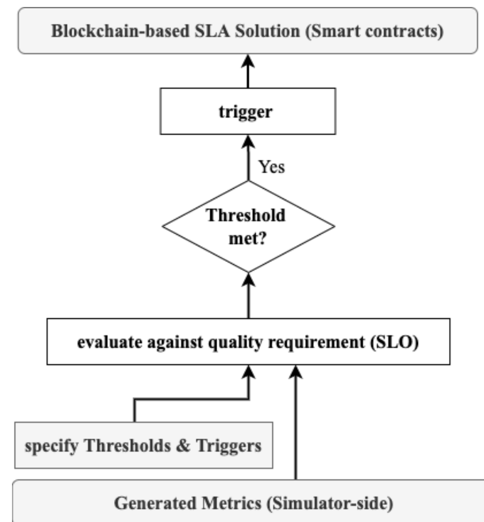


Figure 6.1 Basic workflow from simulator to smart contract

To the best of our knowledge, none of the existing IoT simulators satisfies these assumptions. This chapter proposes a middleware that enables featuring existing IoT simulators such capabilities. Therefore, the proposed middleware can enable existing IoT simulators for experimenting and benchmarking blockchain-based SLA solutions.

Moreover, there is still the need to address the gap between the real execution environment of blockchain platforms (i.e. Hyperledger Fabric) and simulated environments of IoT simulators (i.e. IoTsim-Osmosis). However, the middleware must consider two aspects for bridging the gap between these two distinctive environments. First, Discrete Event Simulators (DES) mostly represents events' progression throughout a virtually short-lived time. Second, the output of such simulators can be dependent on preset calculations and execution duration.

For instance, a simple integration between these two distinctive execution environments reveals that simulators do not align well with the real blockchain platform. That is, the integration severely influenced the execution runtime of IoTsim-Osmosis due to a tight coupling with the execution runtime of Hyperledger Fabric. This is because the naive experiment caused the IoTsim-Osmosis simulator to halt execution whenever it transacts with the blockchain. Therefore, the simulator suffers from miscalculation, which leads to unrealistic outputs. For that, the middleware must consider the variation between simulators and real platforms such that while both blockchain platforms and simulators are integrated, they execute their assigned tasks independently from each other.

6.3. Proposed Architecture

This section proposes a generic middleware architecture that facilitates the usage of Java-based IoT simulators for experimenting and benchmarking real-world blockchain-based SLA solutions. Figure 6.2 depicts an architectural overview of the proposed middleware approach, which

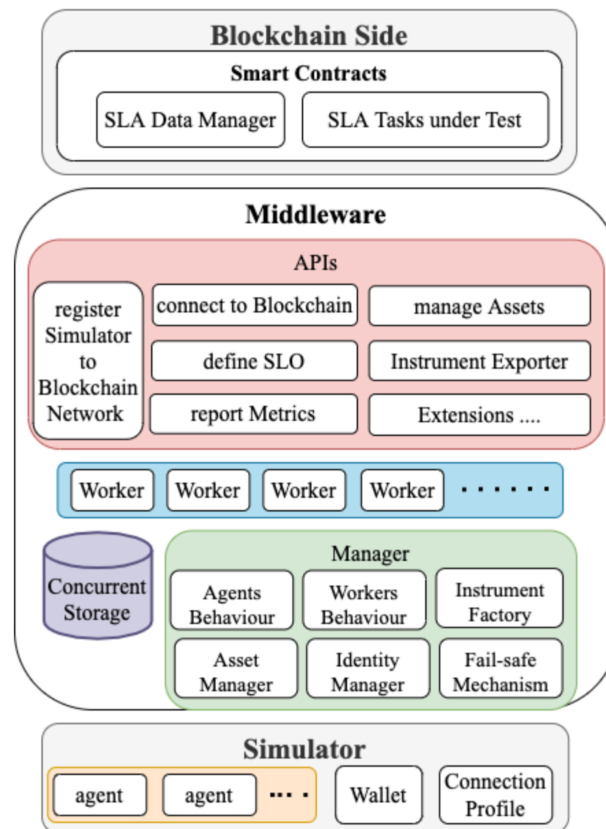


Figure 6.2 Blockchain-based Middleware Architecture

integrates between the blockchain side and the simulator side. This section highlights the main components of the proposed architecture and justifies their existence with relevance to the research problems discussed in section 6.2.2. Other auxiliary components are covered in Appendix C.

6.3.1. The Blockchain Side

Regarding the blockchain side, the middleware is influenced by the work done in the previous chapters, as in the following:

6.3.1.1. Blockchain Network

This study adopts Hyperledger Fabric as an underlying blockchain platform. Figure 2.7 depicts a basic blockchain network based on Hyperledger Fabric. Section 2.4.1 provides further detail on the infrastructure and components of the blockchain network. Accordingly, the middleware follows Hyperledger Fabric's philosophy in terms of the permissioned environment, authentication and transaction flow.

6.3.1.2. Identity Management

Hyperledger Fabric is a permissioned blockchain platform and therefore authentication and authorisation must be facilitated. Figure 6.3 roughly illustrates the process of registering

and enrolling a simulator to the blockchain network, which is automated by the middleware. Eventually, the output of this process is a generated wallet, which consists of two identities with different roles; an admin and client application. In this paper, we consider identity to be a collection of public/private key pairs as well as a signed digital identity encapsulated in the form of an X.509 certificate. The admin identity is for the usage of the middleware while the client application identity is for the simulator usage. The wallet of identity is usable for the middleware and the simulator for connectivity and communication with the blockchain side. For instance, the simulator uses its identity to sign its transactions when invoking a smart contract.

In detail, the process of generating a wallet starts with registering an admin identity for the middleware, which yields an id and a secret. The middleware then supplies these credentials for self-enrolment to a certificate authority (CA) host by the blockchain side. The CA validates the supplied credentials and consequently generates an admin identity for the middleware. This identity is provided with administrative attributes that enable the middleware to register and enrol an application client's identity for the simulator. Noteworthy that, rebuilding the blockchain network makes the existing wallet irrelevant and thus unusable. Whenever this situation occurs, the middleware, optionally, replaces the existing wallet with a newly generated one for usability and convenience purposes.

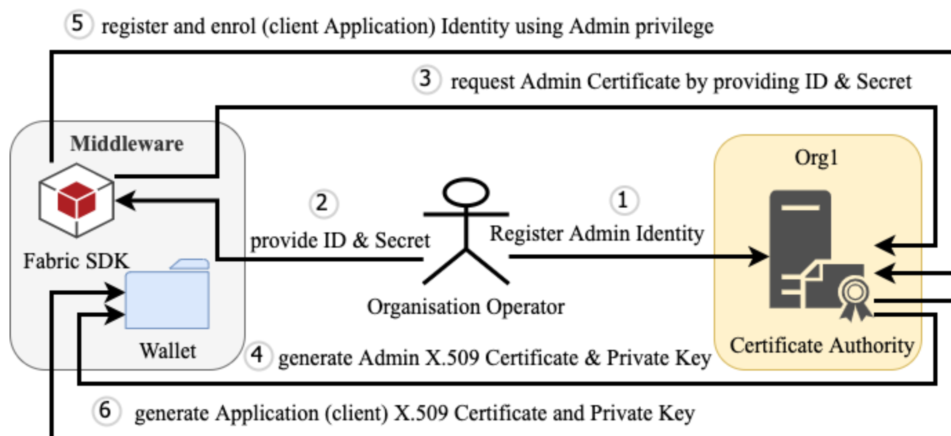


Figure 6.3 Authenticating the application client to the blockchain network and wallet generation

6.3.1.3. SLA Data Manager

The middleware also assumes blockchain-based SLA solution to be following a similar approach to the one presented in Figure 3.8. For example, it assumes a set of smart contracts for managing and defining SLA assets (SLA participants, quality requirements, violation consequences, etc.). Refer to Chapter 3 for further detail on SLA representation and definition with Blockchain. For that, the middleware interfaces with the SLA data manager to leverage CRUD methods for creating, reading, updating or deleting SLA assets such as quality requirements (refer to Figure 3.9 for illustration). The middleware also utilises the SLA manager to manage data at the state storage. For instance, the middleware supports the following actions:

- Adds or updates a quality requirement $q_i \in Q$ for experimental purposes.

- Removes existing records to prevent key conflicts with new ones. (i.e. $pr_i \in PR$).
- Read existing records such as reading a quality requirement or the outcome of a compliance assessment.

6.3.1.4. Smart Contract Under Experiment

The middleware assumes the deployment of a blockchain-based SLA solution such as the compliance assessment approach discussed in section 5.4. For example, Figure 4.3 depicts a compliance assessment smart contract that enables monitoring solutions to report the performance of service providers. In the same principle, the middleware can invoke the smart contract to feed it with the data originating from the underlying simulator. For experimental and observation purposes, one can model an IoT simulated model and adjust the middleware settings to control the data flow to the smart contract.

6.3.2. Transactions Fail-safe Mechanism

As discussed in section 4.7.2, there must be a mechanism that accounts for failed transactions in order to attain reliable outcomes. For that, as Figure 6.4, the middleware subjects all submitted transactions to a fail-safe mechanism. Assume a maximum number of trials in place (e.g. $trials_{max} \leftarrow 5$). The middleware assigns a listener to transactions events on the blockchain side for each smart contract invocation. This is in order to observe whether a transaction is successfully processed and eventually manages to get committed on the ledger. Otherwise, the middleware will retry submitting the transaction as long as the assigned number of trials does not exceed the assigned threshold of max trials. Finally, the middleware prepares a response describing the status of the submitted transaction, which enable the simulator users to form an informed decision about the validity of their experiment. For example, consider a case where a transaction does not succeed to be properly submitted no matter how many times the middleware tries to resubmit due to some issues on the blockchain side. Therefore, the simulator users would decide to abort the entire model simulation due to a deviation from the intended behaviour.

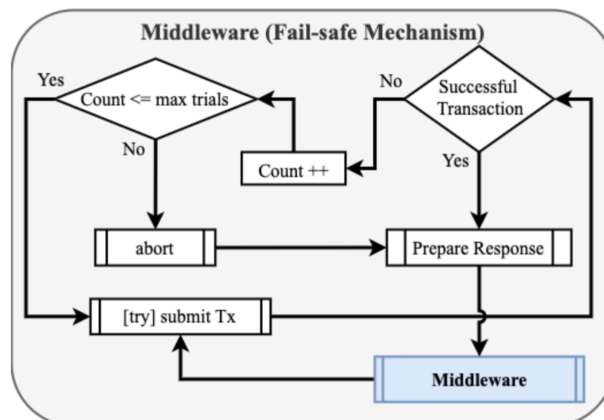


Figure 6.4 A fail-safe mechanism for transactions submission

6.3.3. *Quality Requirement Definition*

There can be any number of quality requirements defined against generated metrics. For example, assume a quality requirement q_i to be $Latency \leq 3s$. As illustrated in Algorithm 9, the middleware takes q_i as an input, and persists it at both sides, the simulator and the blockchain. The creation of q_i requires triggering SLA data manager smart contract. Only when the creation transaction succeeds, the middleware proceeds further to create a replica of the quality requirement q_i' at the local storage; Otherwise, it aborts the entire process. Section 3.5.1.2 provides further details on the creation of SLA components such as $qi \in Q$ at the blockchain-side.

Algorithm 9 Quality Requirement Establishment Protocol

Require: q_i ▷ Quality requirement
Ensure: whether q_i and q_i' are successfully created.
1: $\alpha \leftarrow false$ ▷ success flag
2: **repeat**
3: create a replica q_i at blockchain side
4: **if** q_i **then** ▷ successfully created
5: $\alpha \leftarrow true$
6: **end if**
7: **until** $(\alpha \leftarrow true) \vee maxRetry$
8: **if** α **then** ▷ successful creation of q_i at blockchain-side
9: create q_i' at the simulator-side
10: **end if**

6.3.4. *Monitoring and Reporting Mechanism*

Several blockchain-based SLA tasks, such as compliance assessment, require a monitoring mechanism that observes the service provider's performance. Simulators can model IoT systems and generate metrics about the performance of the simulated model. The proposed approach uses simulators to observe how IoT service providers would perform in various simulation settings. The simulator's generated metrics can represent the performance of an IoT service provider.

However, there is still the need for an SLA-aware monitoring mechanism to evaluate generated metrics against the quality requirements persisted at the blockchain side. For that, the middleware extends the underlying simulator to evaluate generated metrics M against a pre-defined quality requirement q_i and report the outcomes to smart contract under test (e.g. compliance assessment).

Figure 6.5 illustrates the monitoring and reporting mechanism, which is designed to preserve the performance and reliability of the underlying simulator. Recall section 6.2.2 which discusses the impact of coupling the execution time of simulators with a real blockchain platform.

Therefore, the middleware considers three components: agents, workers, and concurrent storage. These components are in place to enable independence of these distinctive execution environments such that agents interface with the simulator side, whereas workers interface with the blockchain side. Both of them share in common a concurrent storage for conducting their operations. The following sections delve further into these three components.

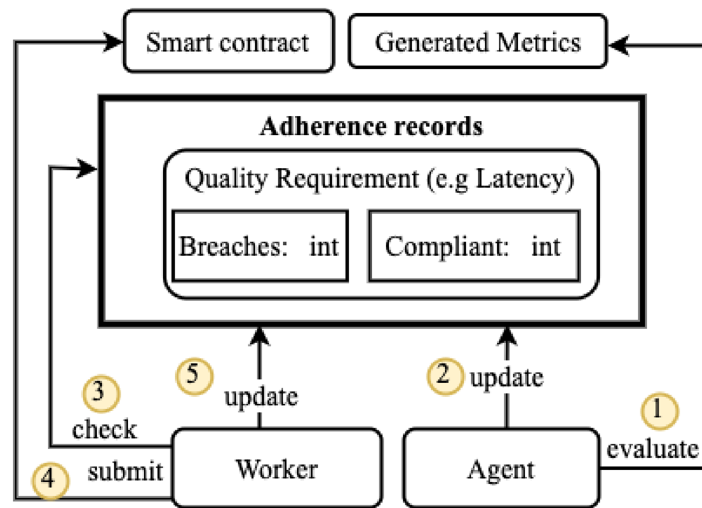


Figure 6.5 Monitoring and reporting procedures

6.3.4.1. Agents

The middleware enables deploying agents at the simulator level for observation and evaluation purposes. In particular, one can deploy agents wherever metrics $\{m_1, ..., m_n \in M\}$ are generated from the underlying simulator. The main task of an agent is to capture and instantly evaluate each $\{m_i \in M\}$ against a $q_i \in Q$ (e.g. $latency \leq d$). Accordingly, the agent classifies whether a generated metric is a breach b or compliant c based on the defined quality level l and threshold t. Algorithm 10 illustrates how agents can intuitively adapt to a set of supported quality levels l (namely: GraterThan, LessThan, Equal, Not).

Based on generated metric's evaluation, agents must report the outcomes of whether a generated metric is classified as a breach or compliant. Note that agents do not communicate directly with blockchain in order to prevent the impact of the blockchain environment on the underlying simulator. For instance, to prevent introduced delay on the underlying simulator's execution runtime. Alternatively, they interface with the local concurrent storage that holds a set of defined quality requirements and their indicators (see Figure 6.5).

The local storage dedicates a slot for each defined quality requirement $q_i \in Q$. Each slot has two properties which are the total compliant cases c and the total count of breach cases b. Accordingly, agents reflect the evaluation outcomes by updating the shared concurrent storage with the total count of the breach b or compliant c cases of the respective quality requirement $q_i \in Q$. Section 6.3.4.3 explains more about the concurrent storage's concept.

6.3.4.2. Workers

Workers are concurrent entities scheduled to interface and transact with the blockchain side. The main worker's role is to consume c and b of every quality requirement $q_i \in Q$ stored at the concurrent shared storage. Workers examine the shared storage to decide whether there is a need to report the latest update to the respective smart contract. Once a worker successfully does so, it updates both c and b by applying the difference between their current values and reported ones (see Figure 6.5).

Algorithm 10 Agents' Evaluation Behaviour

Require: q_i and m_i ▷ Quality requirement and Metric
Ensure: c or b. ▷ Compliant or Breach

```
1:  l ← q_i(level)        ▷ quality level (eg. GraterThan, LessThan, Equals, Not)
2:  t ← q_i(threshold)    ▷ get quality threshold
3:  if l = GraterThan then
4:      if m_i < t then
5:          b++
6:      else
7:          c++
8:      end if
9:  else if l = LessThan then
10:     if m_i > t then
11:         b++
12:     else
13:         c++
14:     end if
15: else if l = Equals then
16:     if m_i ≠ t then
17:         b++
18:     else
19:         c++
20:     end if
21: else if l = Not then
22:     if m_i = t then
23:         b++
24:     else
25:         c++
26:     end if
27: else
28:     undefined
29: end if
```

Algorithm 11 Workers' Reporting Behaviour

Require: $\{contract, method, Q\}$
Ensure: updated beach and compliant counters for each $q_i \in Q$

```
1:  for each q_i ∈ Q do
2:      b_t1 ← b                       ▷ current state of breach counter
3:      c_t1 ← c                       ▷ current state of compliant counter
4:      if b_t1 > 0 or c_t1 > 0 then
5:          send Transaction (contract, method, q_i, b_t1, c_t1)   ▷ report metrics
6:          if successful transaction then
7:              b_t2 ← b               ▷ latest state of breach counter
8:              c_t2 ← c               ▷ latest state of compliant counter
9:              Δb ← b_t2 − b_t1       ▷ Difference between latest and current
10:             Δc ← c_t2 − c_t1       ▷ Difference between latest and current
11:             if Δb ≮ 0 then
12:                 b ← Δb             ▷ update breach counter
13:             else
14:                 b ← 0
15:             end if
16:             if Δc ≮ 0 then
17:                 c ← Δc             ▷ update compliant counter
18:             else
19:                 c ← 0
20:             end if
21:         end if
22:     end if
23: end for
```

In detail, the middleware can schedule a pool of workers to repeatedly carry out the task highlighted in Algorithm 11. The middleware constructs a worker by assigning it with the following:

- The name of a smart contract to transact with (e.g. compliance assessment).
- A particular method of that smart contract (e.g. report incident).
- A set of quality requirements $\{q_1, q_2, q_3, ...q_n\} \in Q$.

For each q_i, workers query the concurrent storage seeking to identify whether there is a status change to the count of c and b of every assigned quality requirement. In order to prevent overwhelming the blockchain side with unnecessary overhead, workers are restricted from transacting with the respective smart contract unless there is a change to the stored counters, as per-defined in Algorithm 11 Line 4. That is, the worker consider them b or c reportable to the blockchain side only when they are $b > 0$ or $c > 0$. Subsequently, it submits their values to the assigned smart contract. It then updates the counters of b and c by applying the difference between their current values and submitted values.

However, applying the difference to b or c is not a straightforward process because of the fact that agents and workers share the same storage, see section 6.3.4.3. For that, it is important to address possible race conditions between agents and workers that operate on the same quality requirement at the shared storage. In particular, when a worker transacts with the blockchain side, there is a considerable elapsed time between when a worker examines the quality requirement at time t_1 and when it finishes transacting with blockchain at time t_2. Meanwhile, agents or other workers can highly likely manage to update the state of b and c. For that, the middleware has in place a concurrency safety mechanism that accounts for elapsed time and possible race condition, as follows:

- Only one entity (either agent or a worker) acquire a write access (update) on the c or b of a quality requirement q_i stored in the concurrent storage, as per detailed in section 6.3.4.3.
- Agents may acquire a write access during elapsed time $\Delta t = t_2 - t_1$, to increment the values of b and c. When the worker finishes transacting with the blockchain side, it cannot acquire write access (update) on the respective q_i unless no other entity holds the lock. Therefore, the worker waits until the lock is released and holds it to apply the difference Δb and Δc between the latest state of both b and c previous their states at t_1. In this way, only successfully transacted metrics are wiped from the concurrent storage while preserving new metrics to be reported in the upcoming transacting round.
- For the sake of safety and correctness, workers take extra measure by refraining from applying Δb and Δc on the concurrent storage if they are less than zero.

6.3.4.3. *Concurrent Storage*

Recall that agents interface with the concurrent storage from the simulator side, whereas workers do so from the blockchain side. For that, the middleware employs concurrent storage that accounts for concurrent operations conducted by agents and workers on locally stored quality requirements. Figure 6.5 depicts how the both agents and worker interacts with the concurrent shared storage.

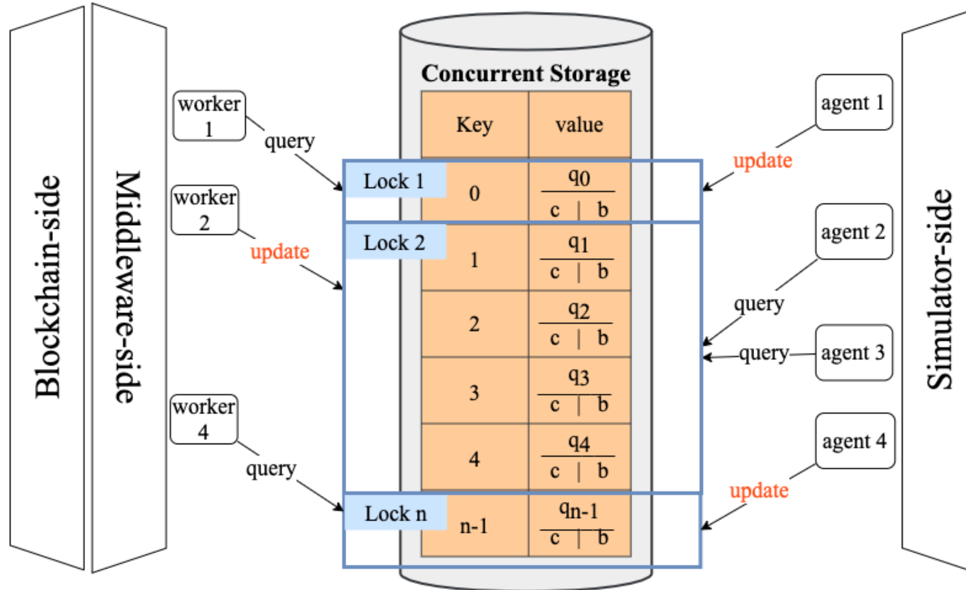


Figure 6.6 A snap of locking mechanism acquired by agents or workers on the concurrent storage

Contrary to agents, workers do not share the execution runtime of the underlying simulator but instead operate in isolation from it. The separation between their execution environments is in place to prevent the impact of the blockchain's transaction flow on the underlying simulator. Refer to section 2.4.2 for more on blockchain transaction flow. That is, a direct integration can introduce undesired delays or halts to simulated models, which eventually cause miscalculation and unpredictable simulation outcomes. To preserve the simulator reliability and prevent race conditions between agents and workers, we design the concurrent storage in Figure 6.6. It leverages a Java ConcurrentHashMap feature structured as $(key, value)$, where *key* refers to an Identifier of a quality requirement, while *value* holds the current status of breach *b* compliant *c* metrics of the quality requirement.

The concurrent storage inherits a locking mechanism that temporally grants write access (insert, delete, and update) to only one entity (worker or agent) while simultaneously enabling multiple read access to any entity. While Concurrent-Hash-Map applies the same principle, it is fixable in that it does not lock the entire concurrent storage to only one writer, which would hinder the overall performance. Instead, it divides the concurrent storage into a set of segments and thus, write locks are applied segments-wise instead of locking the entire storage.

This enables multiple writers to operate on concurrent storage while preventing shared write access on any segment. As a consequence, it increases throughput and enhances overall performance. Figure 6.6 provides a snapshot example illustrating how the concurrent storage

is segmented due to locks acquired a set of stored quality requirements by either an agent or a worker. We shall observe the following:

1. Segment 1 is locked by agent 1, which wants to report breach b and compliant c cases as a result of evaluating metrics against q_0. Meanwhile, worker 1 is able to query the latest status of q_0 for examination purposes, which is possible because there is no restriction on reading operations.

2. Segment 2 is locked by worker 2 as a result of a successful transaction about metrics related to a set of quality requirements $\{q_1, q_2, q_3, q_4\}$, and thus aims to reflect Δb and Δc on each quality requirement contained by segment 2. Meanwhile, agents 2 and 3 will be able to query some quality requirements that happen to fall into this locked segment without collisions.

3. Whenever a worker or an agent requires a write access privilege to a particular segment, it waits until the lock is realised then the requester can acquire the lock for its own benefit.

6.3.5. The Problem of Duplicate Smart Contract Invocations

The middleware assigns each worker various quality requirements to examine them on the concurrent storage and report them to the blockchain side. It also permits parallel execution by enabling the creation of multiple workers. However, the middleware must account for multiple invocations of the smart contract; a situation that leads to unintentional records duplication at the blockchain side.

To elaborate, assume a case when workers share the assignment of the same quality requirements in common. Then, consider that the locking mechanism does not restrict multiple read access. Therefore, workers are triggered to transact with the blockchain to invoke the smart contract. However, according to the mechanism of the compliance assessment approach, this invocation leads to creating a new performance report $pr_i \in PR$ (see section 5.4). Subsequently, multiple workers will unnecessarily create duplicate performance reports simultaneously, which will deviate the logic of the smart contract from the intended behaviour. For instance, the compliance assessment smart contract could end up wrongly considering an obligated party as compliant or in violation of a quality requirement.

Revisit Figure 6.6 and note that the middleware prevents multiple workers from sharing the same quality requirements at the same time. This measure is in place to mitigate the possibility of duplicate smart contract invocation. Furthermore, As Algorithm 12 illustrates, when the middleware assigns a set of quality requirements Q a worker w_i, the middleware holds a list L, which comprises a set of already assigned quality requirements to some existing workers. This list is in place to ensure that no contained quality requirement $q_i \in L$ is assigned twice to multiple workers. Therefore, if $q_i \notin L$, the worker may assume responsibility for it. When a worker terminates for any reason, the middleware removes their assigned quality requirements from L, declaring their availability for other workers.

Algorithm 12 Duplication Prevention Mechanism

Require: $\{w_i, Q, L\}$
 ▷ w_i stand for a worker
 ▷ Q A list of assigned quality requirements
 ▷ L a List of already assigned quality requirements
Ensure: no $q_i \in Q$ assigned previously to existing workers
1: **for each** $q_i \in Q$ **do**
2: **if** $q_i \notin L$ **then**
3:   assign q_i to w_i
4:   add q_i to L
5: **end if**
6: **end for**
7: **if** w_i is terminated **then**
8: remove q_i from w_i
9: **end if**

6.4. Validating the Concurrent Storage

This section verifies whether the core functionalities of the middleware scales to our predefined expectations. For that, we implemented the middleware architecture as presented in Figure 6.2. We provide it as an open-source project on GitHub [2] as well as the source code of all testing and experiments conducted and illustrated in the following sections.

The focus of this section is to ensure the correct operation of the concurrent storage because it acts as the backbone for middleware which crucially reflects on the middleware's overall performance and reliability. To appreciate the criticality of the concurrent storage, consider the fact that it is shared between two distinctive execution environments; the simulator side and the blockchain-based middleware side (see Figure 6.6). Additionally, recall that agents are designated to operate on the simulator side, while workers at the other side as per-defined Algorithms 10 and 11; respectively. Therefore, it is pivotal that the middleware must account for the race condition between agents and workers that commonly update the same b and c counters of a quality metric q_i. Accordingly, the state of the concurrent storage can represent a major indicator of the correctness of the middleware's behaviour.

That is being said, we devised an experiment, as shown in Algorithm 13, which examines the state validity of the concurrent storage. The experiment does not employ either the simulator or a blockchain platform at this stage. Rather, we focus on the middleware and conduct a mock test that aims to validate its correctness beyond the influence of both ends. This section leaves the actual evaluation of the middleware in a real setting to the next sections.

For the mock test, we assume a quality requirement q_i to be $Latency \leq 1$ second which to be imposed on a set of random metrics $m_i \in M \mid m_i \subset \mathbb{Q}$. The experiment runs several iterations, such that for each iteration the mock test generates random metrics of size $|M| = 10^x \mid 2 \leq x \leq 6$, where x is initialised by 2 for the first iteration and then incremented by 1 for each subsequent iteration until $x = 6$; inclusive. Each iteration evenly distributes M into two subsets, where the first set consists of $\{m_i \in M \mid m_i > 1\}$ that we already know to be a breach to q_i. On the other hand, the second set comprises $\{m_i \in M \mid m_i \leq 1\}$, which we already know to be compliant with

[2] https://github.com/aakzubaidi/BMBmid-Middleware

Table 6.1 Test-bed for validating correct operations on the concurrent storage.
* delay is applied only for the first 100 generated metrics, when $x = 2$, for demonstration and visualisation purposes.

Quantity		Initial Delay	Periodic Delay
Agents	1	1s*	0
Workers	3	2s	3s
Generated Metrics		$10^x \mid x \in \{2,3,4,5,6\}$	
		Compliant	50%
		Breach	50%

q_i. Therefore, the experiment initiates an agent a_i and a set of workers w_i, where $\{i \in \mathbb{N}\}$ to carry out their tasks. Table 6.1 presents a summary of the experiment settings. We presume the following expectations:

- a_i is to evaluate every $\{m_i \in M \mid |M| = 10^x\}$ against q_i and accordingly updates the concurrent storage in terms of b and c; as per-defined in Algorithm 10.
- w_i is to process existing b and c associated with q_i and to correctly apply, whenever relevant, Δb and Δc on the concurrent storage; as per-defined Algorithm 11.

Figure 6.7 shows a sample of how we observe the state validity of the concurrent storage. For visualisation and demonstration purposes, this sample depicts a small number of generated metrics $M = 10^x \mid x = 2$ (100 generated metrics) used in the first experiment iteration. The small size is to visualise changes on the state of the concurrent storage over a reasonable time frame. For that, Table 6.1 introduces a delay of 1 second to the agent to clearly show the state change only for this sample, where $x = 2$.

To justify the agents delay at the first experiment iteration where $x = 2$, consider that whenever a worker visits the concurrent storage, it consumes all existing b and c counters at once. Therefore, small size of generated metrics with no delay would enable agents to process all of them before the worker visit, which prevents us from properly observing and visualising changes on the concurrent storage over a proper time frame. Therefore, we controlled the metric rate generation $m_i \in M$ by 1 second at the first iteration. Therefore, agents are also influenced by this intentional delay, which effectively causes workers w_i to undergo several rounds to consume changes on the concurrent storage. Subsequently, this delay helps demonstration and visualisation when $x > 2$. Subsequent iterations, where $x \geq 3$, do not apply a delay since they are not visualised in Figure 6.7.

On the other hand, we arbitrarily establish three workers in a pool targeting any positive values of either $b > 0$ or $c > 0$ related to the quality metric q_i. Recall that the middleware does not enable more than a worker to operate on the same q_i at a time. Therefore we are interested in observing whether an unintended behaviour materialises, such as a duplication or a miscalculation. The experiment introduces an initial delay of 2 seconds to workers to allow room for a state change to occur on either b or c due to agent evaluation. In order to emulate the behaviour of transacting with blockchain, the middleware introduces a periodic delay of 3 seconds to the selected worker.

Algorithm 13 Controlled Experiment on Core Functionalities

Require: $\{M, a_i, w_i, q_i\}$
 ▷ M a set of metrics, such that $\{m_j \in M\} \subset \mathbb{Q}$
 ▷ a_i stands for an agent
 ▷ w_i stands for a worker
 ▷ q_i stands for a quality requirement
 ▷ $tma \leftarrow 0$ Total metrics evaluated by a_i
 ▷ $tmw \leftarrow 0$ Total metrics processed by w_i

Ensure: valid state of concurrent storage and correct operations
1: Let q_i be $Latency \leq 1$
2: $b \leftarrow 0$ & $c \leftarrow 0$ ▷ counts for breach and compliant cases
3: Assign q_i to w_i
4: $x \leftarrow 100$
5: **while** $x < 1000000$ **do**
6: $j \leftarrow 0$
7: **while** $j < x$ **do**
8: **if** $x \leq 100$ **then** ▷ Intentional delay for visualisation purposes
9: Delay 1 second
10: **end if**
11: **if** $(j \bmod 2) \leftarrow 0$ **then**
12: $m_j \overset{R}{\leftarrow} \mathbb{Q} \mid m_j \leq Latency$ ▷ should be a compliant metric
13: **else**
14: $m_j \overset{R}{\leftarrow} \mathbb{Q} \mid m_j > Latency$ ▷ should be a violation metric
15: **end if**
16: Assign m_j to a_i
17: j++
18: **end while**
19: $\alpha \leftarrow tma - tmw$
20: $\beta \leftarrow tma + tmw$
21: **if** $\alpha = 0$ & $\beta = 2x$ **then**
22: pass
23: **else**
24: fail
25: **end if**
26: **if** $b \neq \frac{tma}{2}$ **or** $c \neq \frac{tma}{2}$ **then**
27: incorrect classification (fail)
28: **else**
29: correct classification (pass)
30: **end if**
31: $x = x \times 10$
32: **end while**

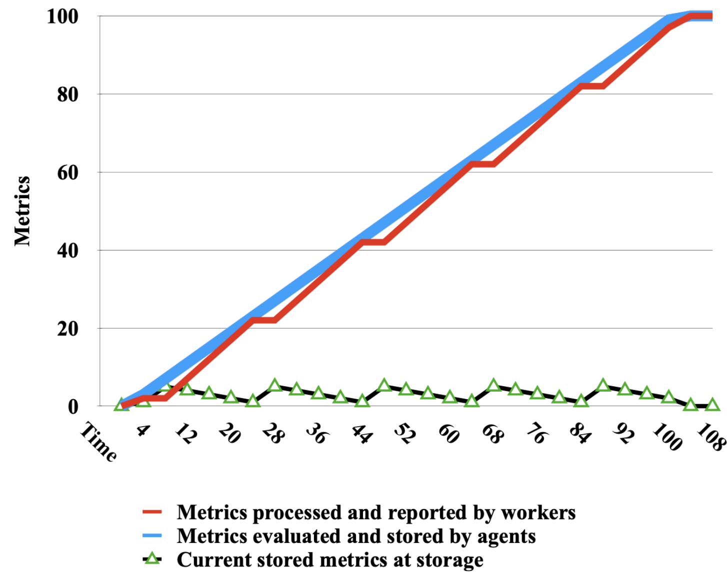


Figure 6.7 Demonstrating the validation process of correct operations on the concurrent storage: using 100 generated metrics

At the end of each experiment iteration, we apply a set of checks as follows:

- tma which tracks the total metrics evaluated by an agent a_i of an mi, which must be exactly $tma = 10^x$.
- b tracks the count of breach cases, which must be exactly $\frac{tma}{2}$, indicating correct classification by the agent.
- c tracks the count of compliant cases, which must be exactly $\frac{tma}{2}$, indicating correct classification by the agent.
- tmw tracks the total metrics processed by all workers, which must be exactly $tmw = 10^x$.
- $\alpha \leftarrow tma - tmw$, which tracks the difference between the count of metrics logged by agents and those processed workers. Accordingly, α must always reflect a correct difference between $tma - tmw$ and be exactly $\alpha \leftarrow 0$ by the end of each experiment iteration.

As visualised in Figure 6.7 and being examined by algorithm 13, the middleware performs as intended for the first iteration where $x = 2$. The experiment also demonstrated correct behaviour with other iterations where the size of generated metrics is $\{10^x \mid 2 < x \leq 6 \in \mathbb{N}\}$. While it is impractical to visualise them in this chapter, we provide the experiment as open source in the above-mentioned GitHub repository for replication and reproduction purposes.

6.5. Evaluation

This section evaluates the utilisation of the middleware to bridge the gap between IoT simulators and real-world blockchain platforms. For that, we select IoTSim-Osmosis simulator [33] to model the firefighting ecosystem as per Figure 5.4. On the other side, we employ Hyperledger Fabric[29] to compose a basic blockchain network as per Figure 2.7.

This section demonstrates the usage of the middleware in integrating the simulated IoT model with the smart contract and the ability to facilitate connectivity and communication between the two-side of the argument. We use three test cases with different predefined compliance rates. Accordingly, we judge the integration successfulness and correct behaviour of the middleware by confirming the smart contract's ability to produce the exact compliance rate as expected. The following subsections detail the experiment and present the outcomes.

6.5.1. Blockchain-side Deployment

On the blockchain side, we selected Hyperledger Fabric as an underlying blockchain platform. Table 5.2 illustrates the configuration of the blockchain network. Section 2.4.1 provides further detail on the components of the blockchain network. Moreover, we deploy the SLA data manager to the blockchain network to facilitate CRUD operations for the middleware (refer to section 6.3.1.3). Chapter 3 describes in length the detail of SLA components such as quality requirements. For this experiment, we assume the quality requirement q_i to be $Transmission_{time} \leq 3s$. Therefore, the Middleware invokes the SLA data manager to create this quality requirement. As regarding the smart contract under test, we deploy the enhanced compliance assessment approach to the blockchain network. Refer to section 5.4 and Figure 4.3 for further description of the compliance assessment approach. The ultimate aim is to illustrate how the middleware facilitates experimenting with the compliance assessment using a simulated IoT model.

6.5.2. Simulated IoT Model

At the simulator side, we employ IoTSim-Osmosis simulator [33] to model the IoT-based fire fighting architecture as illustrated in Figure 5.4. In terms of IoT architecture, the simulated model consists of three main layers:

- A multi-cloud layer that serves both the fire station and IoTSP, such that each one of them is hosted in a separate data centre.
- three distributed edge data centres such that each edge data centre serves a geographically separate areas.
- 30,000 smart houses such that each set of 10,000 houses connect to an edge data centre.

As the SLA in section 5.2.1.2 specifies, the simulated model assigns the IoTSP provider responsible for handling fire alerts emitted from every connected house through their respective edge data centres up to its hosted cloud-based system. Therefore, the IoTSP transmit confirmed fire alerts to another cloud-based system dedicated to the fire station.

The experiment focuses on the transmission time that takes fire alerts to travel from every connected house, through the IoTSP, to the fire station. Therefore, we calibrate the simulated model's specifications and workload altogether around a quality requirement q_i presumed to be

Table 6.2 IoT Components Specifications: calibrating the simulated IoT model around 3 seconds for transmission time

IoT Provider (IoTSP)						Fire Station			
Edge Data Center		Cloud				Cloud			
		Host		VM		Host		VM	
CPUs	4	CPUs	4	CPUs	4	CPUs	4	CPUs	4
Bandwith	100 Mbps	Bandwith	1000 Mbps	Bandwith	1000 Mbps	Bandwith	1000 Mbps	Bandwith	1000 Mbps
RAM	4 GB	RAM	8 GB	RAM	4 GB	RAM	8 GB	RAM	4 GB
MIPS/CPU	250	MIPS/CPU	1250	MIPS/CPU	500	MIPS/CPU	1250	MIPS/CPU	500
Storage	200 GB	Storage	500 GB	Storage	200 GB	Storage	500 GB	Storage	200 GB

Table 6.3 Testbed for causing different compliance rate for each test case

	Test1	Test2	Test3
Expected Compliance Rate	0%	66.67%	100%
Fire Station Cloud VM Allocation	3 VMs	3 VMs	3 VMs
IoTSP Cloud VM Allocation	1 VM	2 VMs	3 VMs
Distributed Edge Data Centres	3		
Connected Houses	30,000		
Allocated houses to each Edge Data Centre	10,000		
Max Fire Alerts per House	1		

$Tansmission_{Time} \leq 3s$. Table 6.2 illustrates how we calibrate the simulated model around q_i and highlights primary specifications that we used for modelling each layer. Then, the experiment conducts three disincentive test cases where the IoTSP performs differently against this quality requirement.

Table 6.3 presents the similarities and differences in terms of allocated resources and specified workload for each of the test cases. Regarding the similarities, we allocate the IoTSP responsible for handling and managing geographically despaired 30,000 connected houses, evenly distributed to three different edge data centres. In all test cases, we assume the worst-case scenario where all connected houses simultaneously emit fire alerts, in which the IoTSP transmit them through their associated edge data centres up to the IoTSP cloud service. Each connected house emits only one fire alert, making the total of simultaneous 30,000 fire alerts handled by the IoTSP. The fire station dedicates three virtual machines for its cloud hosting in all test cases.

Regarding the differences, the test cases differ in how the IoTSP (the obligated service provider) allocates virtual machines to its cloud hosting. While the IoTSP conforms to the virtual machine specification as per Table 6.2 for all of the test cases, it allocates a different number of virtual machines in each test case. Therefore, the variation of allocated virtual machines impacts the IoTSP's compliance rate against q_i in each test case as follows:

1. Test 1: the IoTSP exhibits the best compliance rate as high as 100% by providing at least three virtual machines.
2. Test 2: the IoTSP exhibits the worst compliance rate as low as 0% by providing only one virtual machine.
3. Test 3: the IoTSP exhibits a moderate compliance rate of 66.67% by providing two virtual machines.

6.5.3. Middleware Settings

We imported the middleware as a library to the IoT simulator to integrate it with the blockchain side. Then, we instantiate a middleware manager with the following configuration:

- URI to the certificate authority and associated TLS certificate.
- An admin id and secrete, manually issued by the certificate authority.
- A client identity name for simulator; to be issued by the middleware.
- wallet path indicating where the middleware must place generated identities.
- A connection profile describing the blockchain network: used for peers discovery and connection purposes.
- Blockchain channel name.
- the name of the SLA data manager smart contract.
- The name of the smart contract under test (compliance assessment).

Afterwards, we defined a quality requirement to be $Transmission_{time} \leq 3s$ at the local storage, which is to be persisted at the blockchain side. Then, we assigned this quality requirement to an agent and a pool of workers as per Algorithms 10 and 11; respectively. Both of them are configured as per in Table 6.1, except that there is no delay is imposed on the agent. Then, we injected the agent into the source code of the simulator to capture generated metrics related to the transmission time. Workers are assigned the task to transact with the compliance smart contract by invoking a specific endpoint dedicated to processing the updated count of the breach and compliant metrics. By running all three simulated IoT models (test cases), the middleware succeeds in bridging between the simulator-side and blockchain-side. Investigating both the state storage and logged transactions validates that the middleware managed to invoke the SLA data manager in order to persist the defined quality requirement.(see Appendix C, Figure C.1).

6.6. Evaluation Results

This section presents the evaluation's results for all three test cases about utilising the middleware to integrate a Java-based IoT simulator with the blockchain environment.

6.6.1. Correct Behaviour

Given the involvement of both the blockchain side and simulator side, we further validate whether the count of compliant and breach cases meet the predefined expectation. Consider M to be a set of the simulator's generated metrics about *transmission time*. Workers periodically examine the concurrent storage looking for new breach or compliant cases, and then report them to the blockchain side. Whenever the smart contract receives an update from workers, it processes the

transaction payload and creates a new record pr_i on the blockchain's state storage, containing the count of new breach cases b and compliant cases c. Therefore, we expect the total count of both breach and compliant cases to be identical to the size of $|M|$, as per equation 6.1:

$$\left(\sum_{pr_0}^{pr_{n-1}} b + \sum_{pr_0}^{pr_{n-1}} c \right) \stackrel{?}{=} |M| \tag{6.1}$$

As Figure 6.8 illustrates, the middleware submits a variable number of transactions for each test case to the blockchain side. The variability of transaction count is influenced by the distribution of both breach and compliance metrics in each test case. For instance, the first two test cases exhibit either 100% or 0% compliance rate, meaning that all generated metrics are compliant or in violation. Therefore, workers needed to submit three transactions in both test cases. As for the third test case, the generated metrics contain a mix of breach and compliance elements. This introduces further operations to workers and thus is reflected on the count of transactions needed for submitting all metrics to the blockchain side.

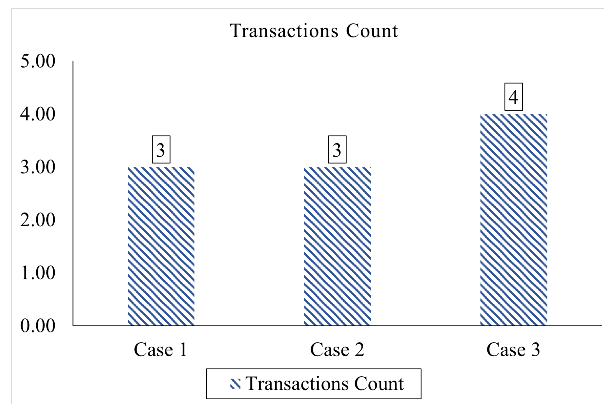
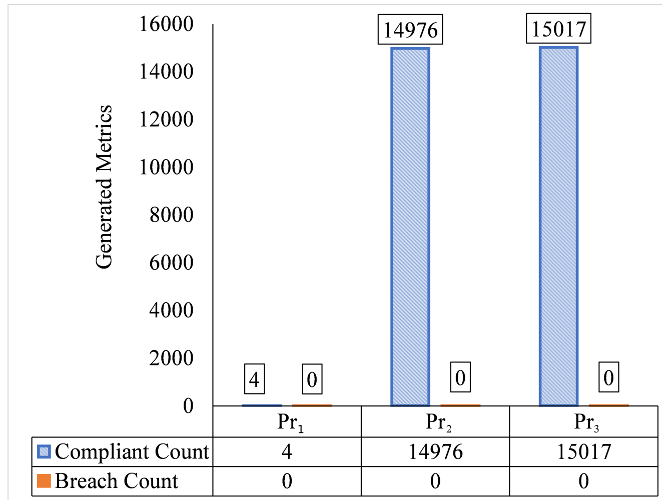


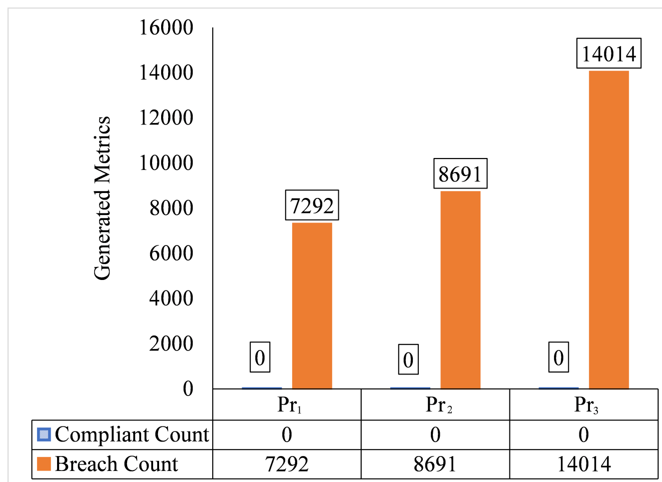
Figure 6.8 Transactions count for all 3 test cases

Moreover, we examine the ledger state on the blockchain side, intending to investigate the count of compliant and breach metrics per transaction. Figure 6.9 illustrates for each of test cases the count of compliant and breach metrics per transaction $c, b \in pr_i$. For instance, Figure 6.9a shows that it takes 3 transactions to report all metrics to the blockchain side. Therefore, the compliance assessment smart contract created three performance reports pr_1, pr_2, and pr_3. Examine the performance record pr_2 in Test 3 (see Figure 6.9c), which comprises a mix of compliant c and breach b metrics by the count of 9 and 9784, respectively. One can read other charts in Figure 6.9 in the same manner. Figure C.2 in Appendix C depicts a sample of a performance report at the state storage. Furthermore, we investigate the ledger to confirm whether the total of submitted metrics is identical to the size of generated metrics $|M| = 30,000$. Figure 6.9 demonstrates that middleware accounts for all generated metrics and successfully in all three test cases and accordingly submits them to the blockchain side.

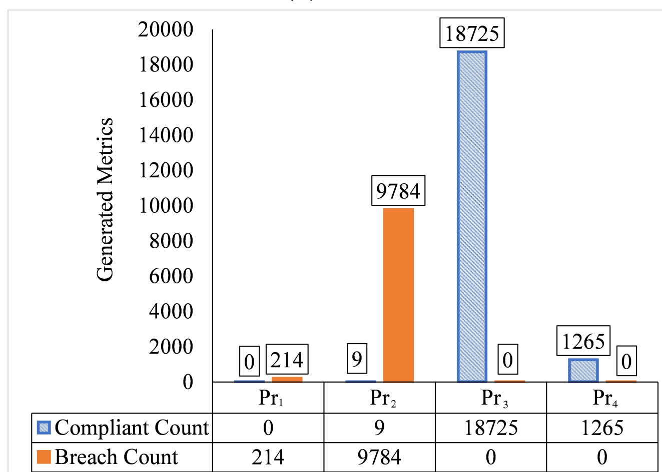
Subsequently, we expect the compliance rate in each test case to match the pre-defined expectation for it, where we assume 100% compliance rate for the first case, 0% compliance rate for the second case, and 66.67% for the last case, which we calculate and validate as in Equation 6.2:



(a) Test 1



(b) Test 2



(c) Test 3

Figure 6.9 Count of compliant and breach cases per transactions on the blockchain state storage

$$\left(\frac{\sum_{pr_0}^{pr_{n-1}} c}{\sum_{pr_0}^{pr_{n-1}} b + \sum_{pr_0}^{pr_{n-1}} c} \times 100\right) \stackrel{?}{=} \left(\frac{\sum_{m_0}^{m_{n-1}} c}{|M|} \times 100\right) \tag{6.2}$$

The right side of Equation 6.2 represents the expected compliance rate for each test case as per Table 6.3. We use it to calculate the ratio of compliant metrics relative to the size of all generated metrics. The other side of the equation represents the compliance rate as processed by the compliance assessment smart contract (refer to Algorithm 8). Figure 6.10 shows that, all test cases managed to match the predefined expectation as per Table 6.3. The evaluation outcome does not only demonstrate the correct behaviour of the middleware, but also the accuracy of compliance smart contract.

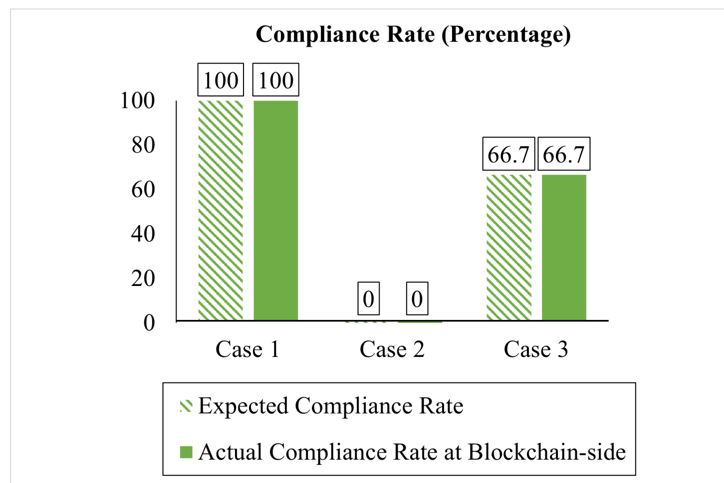


Figure 6.10 Actual Compliance Rate vs expected ones in all 3 cases

6.6.2. Simulator Execution Time

One of the main goals of the middleware is to maintain the performance of the underlying simulator to the best possible. As discussed in section 6.2.2, using the simulator's execution runtime for transacting with the blockchain-side can influence the execution time and thus negatively impacts the accuracy of generated metrics. The middleware architecture addresses this concern by decoupling the simulator core logic from introduced blockchain-related tasks, as being illustrated in Figure 6.6. This forms a positive impact on the simulator execution time, as is evident in Figure 6.11, which shows an identical execution time of the simulator before and after integration with the blockchain platform. Subsequently, it eliminates concerns related to the simulator execution time and the reliability of generated data.

6.7. Smart Contract Benchmarking

The middleware provides out-of-the-box instrumentation of essential measurements such as throughput, latency, success and failure rates. We base the calculation of these measurements

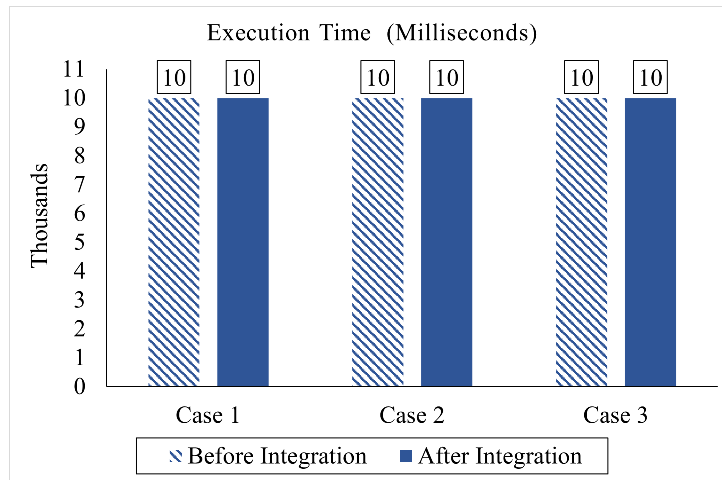


Figure 6.11 Simulator's execution time before and after integration in all 3 cases

on Hyperledger Performance Working Group [88]. Section 2.4.3 provides further detail on blockchain benchmarking and measurements. This section details the instrumentation process, export, analysis, and visualisation. It also describes the methodology used to validate the accuracy of the middleware in terms of measurement instrumentation.

6.7.1. Composing and Exporting Instruments

Figure 6.12 illustrates the sequences of composing and exporting instruments for every invoked method of the smart contract. The middleware actively observes interactions between workers and the smart contract under test (e.g. compliance assessment). In particular, it observes all method invocations whenever a worker attempts to transact with the smart contract under test. For instance, the compliance assessment smart contract provides a method called *report incident* for receiving and processing transactions emitted by invoking entities (see Figure 4.3).

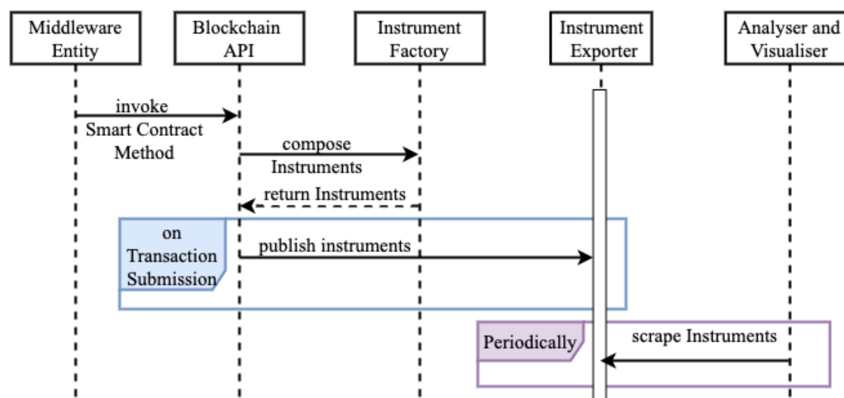


Figure 6.12 Sequence of composing and exporting Instruments

There is a component called *Instrument Exporter*, which assigns a set of metrics for the interaction between the worker and the smart contract. For the instrumentation, we use Micrometer[3] because it is open-source, vendor-free, and supports a wide range of monitoring tools. The instrumented metrics are essential elements for the performance benchmarking Equations

[3] https://micrometer.io

in section 2.4.3, which we adopt for measuring the performance of the smart contract. These instruments include successful transactions T_s, failed transactions T_f, and transaction latency T_d.

Algorithm 14 demonstrates how the middleware self-composes instruments for every smart contract. Whenever a worker submits a transaction, the middleware confirms whether the smart contract needs instrumentation. It does so by checking a list that maintains all smart contracts that have been instrumented beforehand. If the smart contract does not exist in the list, the *Instrument Exporter* creates a new set of instruments for it, which includes successful transactions T_s, failed transactions T_f, and transaction latency T_d. Subsequently, it includes the smart contract into the list to prevent duplication.

After submitting the transaction, the middleware increment either T_s or T_f according to their status. Regarding the transaction latency T_d, the middleware calculates the duration from submitting a transaction until receiving a response from the blockchain side. It is calculated as $T_d = timer_{end} - timer_{start}$, where tx_{start} is the time of submitting the transaction to the blockchain side while tx_{end} denotes the time of committing the transaction on the blockchain ledger. Note that T_d effectively does not consider latency of failed transactions as per described in section 2.4.3.

Algorithm 14 Instruments Composition

Require: $\{contract\}$
Output: $\{T_s, T_f, T_d\}$
▷ T_s successful transactions
▷ T_f failed transactions
▷ T_d transaction latency

```
1: let I = {method_i | i ∈ ℕ}                  ▷ List of all method of the smart contract
2: for each smart contract invocation do
3:     if method ∉ I then
4:         compose Instruments {T_s, T_f, T_d} ∈ method
5:         include contract into I
6:     end if
7:     timer_start
8:     submit Transaction to method
9:     if Transaction is successful then
10:        timer_End
11:        T_d = timer_end − timer_start
12:        T_s++
13:    else
14:        T_f++
15:    end if
16:    publish {T_s, T_f, T_d} to HTTP server
17: end for
```

6.7.2. *Instruments Gathering and Visualisation*

The previous section discussed a component called *Instrument Exporter*, which instruments $\{T_s, T_f, T_d\}$ for each transaction submitted from the middleware to a smart contract under test (e.g. compliance assessment). The middleware enables launching three components for the purposes of gathering and visualising these instruments as follows:

1. An HTTP server for exporting the instruments of each invoked method of the smart contract.

2. Prometheus[4] for periodically gathering exported instruments from the HTTP server.
3. Grafana[5] for formulating the benchmarking measurements and visualisation purposes.

Based on the gathered instruments, and in accordance to section 2.4.3, we can formulate a set of measurements for benchmarking the performance of the smart contract, as follows:

$$Avg_{tps} = \frac{\sum_i^n T_s}{\sigma} \tag{6.3}$$

$$Avg_{latency} = \frac{\sum_i^n T_d}{\sum_i^n T_s} \tag{6.4}$$

$$s_{rate} = \frac{T_s}{T_s + T_f} \times 100 \tag{6.5}$$

$$f_{rate} = \frac{T_f}{T_s + T_f} \times 100 \tag{6.6}$$

All benchmarking measurements depend on instruments composed by the *Instrument Exporter*. Note that Avg_{tps} measures the transactions throughput, which is calculated as the total of successful transactions divided on σ where $\sigma = lct - fst$ given that fst is the time of sending the first successful transaction whereas lct is the time of committing the last transaction to the blockchain ledger. Both s_{rate} and f_{rate} calculate the ratio of successful transaction and failed transaction to the total number of transactions, respectively. Regarding the average latency $Avg_{latency}$, it accumulates the total spent time for processing ever transaction and calculates its ratio relative to only successful transactions.

6.7.3. Experimenting the Benchmarking Feature

In order to demonstrate the benchmarking feature, we designed an experiment as per table 6.4. we configured the simulator to generate a total of 30,000 metrics. In order to enable outcomes prediction, the experiment limits metrics generation by one transaction every 100 milliseconds. Consequently, we expect each agent to feed concurrent storage normally with 10 additional metrics per second. For illustration, figure 6.13 depicts three monitoring agents that observe three edge data centres. Therefore, we expect the concurrent storage to persist 3 metrics every 100 milliseconds.

Meanwhile, the experiment instructs the worker to examine the concurrent storage every 1 second. Therefore, we expect the worker to normally find approximately 30 metrics every 1 second and submit a transaction (incident report) to the blockchain side. Accordingly, the expected total number of transactions is approximately 1000 transactions needed for reporting 30,000 metrics.

The experiment uses Grafana to apply equations 6.3, 6.4, 6.5 and 6.6. By running the simulator, the experiment resulted in a total of 1030 transactions which are over the expected

[4] https://prometheus.io
[5] https://grafana.com

Table 6.4 Benchmarking Configurations and Expectations

Total generated incident alerts	30,000
edge data centres	3
agents per edge data centre	1
Incident generation frequency	1 per 100ms
Agent frequency	100ms
Worker frequency (no initial delay)	1s
Expected transactions per second	1
Expected incident per transaction	≈ 30
Expected total transaction count	≈ 1000

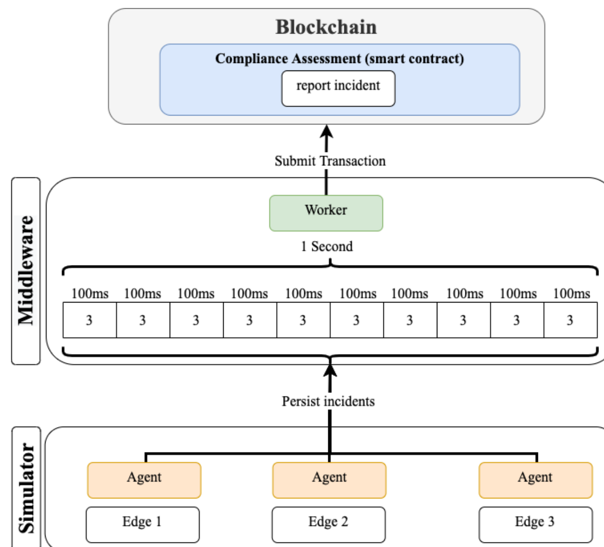


Figure 6.13 Design of the experiment on utilising the middleware for performance benchmarking

one in Table 6.4. Figure 6.14 presents a visualised dashboard of obtained results for average throughput, latency, rate of successful and failed transactions. By investigating the blockchain ledger, we find out that every transaction reports nearly ±30 metrics, thus a marginal deviation from the expected total number of transactions. That is, we originally expected the total number of transactions to be 1000. However, the middleware succeeded to correctly and reliably execute the actual 1030 transactions, which resulted in reporting all the 30,000 metrics generated by the simulator to the blockchain side.

We attribute the marginal deviation of total transactions to two possibilities. First, the influence of Hyperledger Fabric's transaction flow, which holds the worker waiting until its current transaction is resolved [87] before it can commence the second transacting round. Second, the locking mechanism implemented by the concurrent storage which can hold the worker waiting until the agent releases the lock. Nevertheless, we confirm whether the worker managed to submit all 30,000 metrics by running Equation 6.1 on the state storage, which holds valid. Since the marginal error of total transaction does not impact the middleware core functionalities, we leave investigating it further to future works.

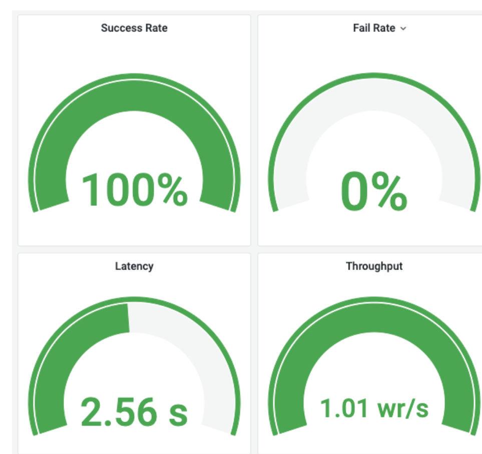


Figure 6.14 Validation of the benchmarking results

6.8. Limitations and Future Work

This chapter aims to architect a generic and a modular middleware that enables the integration of any Java-based IoT simulator with a real-world blockchain environment. In principle, the middleware is implemented in Java and thus should be easily imported as a library to any Java-based simulator, such as those in section 2.1.2, and regardless of the simulation domain, whether it serves IoT or others. While it is possible, in theory, to use the middleware for any Java-based simulator, this chapter carries out experiment only on IoTsim-Osmosis. However, future work will consider demonstrating the middleware for other simulators as well. For simulators implemented in programming languages other than Java, the middleware can possibly act as an intermediate server between the IoT simulator and the blockchain network. However, there is still the need to investigate how the middleware would perform in such a case and what limitations may emerge.

The middleware demonstrates to work with any smart contract deployed on the blockchain network. This is evident because we demonstrated the usage of the middleware with both smart contracts, the SLA data manager and the compliance assessment. The middleware invokes the SLA data manager to persist and query the SLA, while the compliance assessment was under test and benchmarking experiment. The middleware provides an API that is loosely coupled with smart contracts and agnostic to their logic. Therefore, the middleware user must provide information about the blockchain network, channel name, smart contract name, and method name. In this way, the middleware is resilient such that it does not impose the logic of a smart contract or a transaction payload, which is essential for the sake of testing any smart contract. In practice, this feature means that the middleware can be generic and used beyond SLA, perhaps useful for other domains.

Nevertheless, the current architecture only supports Hyperledger Fabric as the leading blockchain platform. Therefore, the current form of the middleware is primarily influenced by Hyperledger Fabric philosophy regarding blockchain implementation, networking, transaction execution flow, supported consensus mechanism, smart contract life-cycle, and so forth. Therefore, future work will consider integrating other blockchain platforms such as Ethereum.

6.9. Conclusion

This chapter discussed the possibility of employing IoT simulators for experimenting and measuring the performance of blockchain-based SLA solutions. On the one hand, several existing blockchain platforms are open source and accessible such as Hyperledger Fabric. On the other hand, IoT simulators can reasonably compensate for the lack of access to large scale IoT infrastructure. Therefore, we see an opportunity to glue the gap between the read execution environment of the blockchain platform and the simulated execution environment of IoT simulators.

This chapter proposed a middleware architecture to bridge the gap between IoT simulators and blockchain platforms. It described the implementation philosophy of the middleware and how it resolves issues related to the distinctive nature of both execution environments. Then, it conducts a mock test to validate the correctness and reliability of the middleware beyond the influence of either end, simulators or blockchain network. Then, this chapter evaluates utilising the middleware for integrating an example simulator IoTsim-Osmosis and Hyperledger Fabric. The evaluation employed a compliance assessment smart contract and a monitoring strategy imposed on an IoT service provider that promises to deliver a quality IoT-based firefighting system. The middleware demonstrates to work as expected in terms of both (I) the integration between the IoT simulator and blockchain and (II) benchmarking the smart contract under test.

Future work will expand the middleware coverage to support other blockchain platforms such as Ethereum and other simulators written in programming languages other than Java. In theory, the middleware can be used for purposes other than IoT or SLA because of its generic and modular nature. However, it is also interesting to experiment in practice how the middleware can be generalised and extended to other domains of interest.

Chapter 7. Conclusion and Future Work

Summary

The blockchain concept provides a set of appealing features that help mitigate trust issues associated with centralised authorities and third parties. Examples of these features include but are not limited to decentralisation, shared ledger, immutable records, consensus mechanisms, traceability and transparency. Generic blockchain platforms, such as Hyperledger Fabric and Ethereum, introduce the concept of smart contracts, which enable autonomous code execution while leveraging the blockchain paradigm's features. Therefore, this thesis finds it appealing to transform SLA distrusted processed into decentralised applications that resist the influence of service providers and third parties.

Section 1.4 provides a summary of the thesis's contributions in this regard. This chapter provides a discussion and insight into the thesis outcomes. Moreover, it highlights threats to the validity of the compliance assessment. Finally, it suggests a set of topics for a future study.

7.1. Discussion and Lessons Learnt

This thesis focuses on the role of blockchain in overcoming trust issues related to SLA compliance assessment in the context of IoT. This section highlights a set of recommendations and lessons learnt:

7.1.1. SLA Awareness

Shifting SLA operation to the blockchain requires sufficient awareness of stipulated content in the SLA. This thesis realises that smart contracts must abide by the transaction flow of the underlying blockchain platform. Most prominent blockchain platforms, such as Hyperledger Fabric and Ethereum, design smart contracts to be event-driven such that they do not initiate calls to the external world. Instead, they should stand still until invoked by an authorised entity, whether it is an external entity or another smart contract. Most blockchain platforms impose this transaction flow to preserve deterministic behaviour and achieve consensus on transaction validity and finality [134].

As Chapter 3 discusses, despite the compliance assessment's decentralisation, there is the risk and complexity of relying on externally hosted-SLA, as follows:

- If the compliance assessment smart contract enquires externally-hosted SLA, it violates the rules of the transaction flow. Thus, there is a high chance of causing the difficulty of reaching a consensus on the transaction validity.

- Hosting the SLA on a centralised server poses the risk of a single point of failure. For example, a downtime, truing the SLA unreachable by the smart contract. Another example is the risk of misconduct or manipulation of a single authority hosting the SLA, which can maliciously influence the behaviour of the compliance assessment smart contract.

- Hosting the SLA in decentralised storage brings unnecessary architectural complexity. Albite, smart contracts must not violate the transaction flow by initiating communication with the external sources, even decentralised ones.

For that, this thesis recommends hosting the SLA internally within the blockchain. This does not only provide necessary SLA awareness for the compliance assessment protocol but also help in mitigating the above-listed issues.

7.1.2. SLA Representation within Blockchain

This thesis recommends that the decentralised SLA compliance assessment protocol only depend on an SLA represented and hosted within the blockchain. Chapter 3 suggests a set of principles grouped into a terminology called IRAFUTAL. These principles enable the SLA to benefit from blockchain features while mitigating issues found in related blockchain-based SLA studies. Table 3.2 summarises the similarities and differences between the proposed approach and counterpart studies.

The key message is that SLA must not be tightly coupled with the lifecycle of any smart contract representing a distrusted SLA task such as compliance. Instead, SLA must reside independently at the state storage independent from any smart contact. However, this thesis recommends the SLA must follow a formal SLA data model enforceable by an SLA data manager to achieve interoperability between different components (e.g. monitoring tools, smart contracts, and users). That is, SLA assets are not under the direct control of any entity. In fact, persisted SLA data assets are under the ownership of the SLA data manager, which is a smart contract that operates beyond the influence of any party.

Different entities need to agree on a standard format comprehensible by machines and human beings alike. The SLA data manager enables authorised entities to consume and manipulate SLA assets. For example, chapter 3 extends the SLA data manager to implement an SLA definition within the blockchain. The same chapter also extends the SLA data manager to demonstrate a more complex example, where SLA parties negotiate and amend SLA assets (content). These cases demonstrate utilising the SLA data manager for creating and updating SLA assets. In terms of SLA consumption, Both chapter 4 and chapter 5 demonstrate the case where monitoring tools can utilise the SLA data manager to read SLA assets.

Ultimately, both of the latter chapters demonstrate the case where the compliance assessment smart contract leverages SLA data manager to consume SLA assets for assessing the performance

of the obligated service provider. Moreover, it consumes the SLA assets to enforce stipulated violation consequences. The compliance assessment smart contract can also uses the SLA manager to terminate the SLA, which causes pushing a notification to the cosponsoring monitoring tool. The notification instructs the monitoring tool to cease transacting with the blockchain about the terminated SLA. Finally, Chapter 6 also uses the SLA data manager as a gateway to the blockchain to control and manipulate existing records. For instance, the middleware can leverage the SLA data manager to manipulate or delete existing records for every simulation round. Otherwise, it would be inevitable to bring down the blockchain network to clear the existing data.

7.1.3. Monitoring

Smart contracts must be terminable, meaning that they cannot execute their operations indefinitely. Therefore, a smart contract cannot actively perform continuous and ceaseless monitoring task. Therefore, the compliance assessment smart contract must rely on trusted monitoring mechanism that observes performance of the service provider in accordance with the SLA-stipulated quality requirements. However, the smart contract must abide by the transaction flow; meaning that it cannot enquire the monitoring tool. Instead, it must waits for invocation by the monitoring tool. Moreover, the monitoring tool should apply a fail-safe mechanism that accounts for transactions failure. For instance, chapter 6 applies a listener to blockchain events to confirm whether the transaction succeeds or resubmission is a must. On the other hand, the smart contract must apply an integrity mechanism to check whether there is discrepancy in the received metrics. Because monitoring tools play an integral part of the compliance assessment, they must maintain high availability, security and performance. Otherwise, they can pose the threat of a single point of failure to the decentralised compliance assessment.

Furthermore, a proper reporting mechanism should avoid overwhelming the blockchain-based compliance assessment. Scheid et al.[20] believe that that their work is limited because the smart contract processes submitted violation metrics only when triggered by the monitoring tool. However, this thesis argues that this is actually a good practice and recommends reporting collected either at long enough intervals or upon the occurrence of an incident. Two critical factors to consider for setting the invocation trigger of the compliance assessment smart contract. First, the monitoring service should not impact the storage of every node participating in the blockchain network. It also should not cause a bottleneck performance to the blockchain network in terms of throughput or latency. Scheduling and queuing strategies can promise a smooth operation as intended. For example, chapter 5 transacts with the smart contract only when there is an incident or upon billing due date.

7.1.4. Compliance Assessment in the Context of IoT

Both chapter 4 and chapter 5 experiment a blockchain-based compliance assessment in the context of IoT. Both of chapters extend the SLA data manager to account for properties needed for the compliance assessment and penalty enforcement. For example, the extension accommodate

incidents received from the monitoring tool. Both chapters employ Hyperledger Fabric as an underlying blockchain platform. However, as Table 1.1 illustrates, they differ in terms of the hypothetical IoT scenario, SLA coverage, communication protocols and type of implementation of both the IoT scenario and monitoring mechanism.

For exploratory purposes, Chapter 4 limits the SLA coverage to a cloud-based IoT component that acts as an MQTT broker for IoT clients. The narrow focus helps limit the scope of cofactors influencing the service provider's compliance status. Thus, the analysis focuses on faults at the cloud side and excludes others beyond the direct control of the cloud provider, such as network and physical layers. Notwithstanding, it can be challenging to determine whether a breach was because of the MQTT broker or otherwise. For example, a client disconnection can be graceful due to the client's intent or because of a failure at the cloud side. For that, the pilot study relies on MQTT specifications to intercept fault errors relevant to the cloud provider.

Chapter 4 considers a more complex SLA that covers a simplified end-to-end IoT system. We recognise that it is still a challenge to determine the root cause of a failure even the entire IoT infrastructure is responsible for one services provider. For example, consider a case when the IoTSP fulfils its duty by delivering a fire alert within the promised duration. However, the firefighting station can claim otherwise, whether maliciously or in good faith. Therefore, we limit the responsibility of the IoTSP up to when it reports the fire incident and relive it from failures at the firefighting station system. Nonetheless, the SLA can be enhanced by obligating the IoTSP to retry reporting the incident several times if it does not receive an acknowledgement. Moreover, it can be difficult to assert whether downtime is a fault of the service provider or a natural cause (e.g. fire damage). For example, downtime at the edge may not necessarily be the service provider's fault.

The design of the assumed SLA in Chapter 5 is a product of several experiments, and we hope it can help provide more insight into the complexity of IoT systems. For example, the SLA considers assigning different max tolerance rates to various breaches. For that, it does not limit the coverage to typical quality requirements such as availability and latency but extends to functional requirements such as whether there is a fire. The tolerance to breaches of a non-functional requirement depends on the criticality of functional quality requirements. For instance, downtime during a fire event is less tolerable than when there is a fire event. The monitoring mechanism must maintain the visibility of the end-to-end IoT system. Moreover, it must observe both functional and non-functional quality requirements in the SLA and provide them to the compliance assessment smart contract. Subsequently, the compliance assessment smart contract determines whether the service provider's performance reaches the intolerable stage and takes further actions based on that.

7.1.5. Reliability and Performance

Table 1.1 shows the difference between chapter 4 and chapter 5 in terms of the employed blockchain platform and deployment settings. Both of them employ Hyperledger Fabric but with different realises and deployment settings. Recall that chapter 4 reveals the phenomena of

transaction failures when subjecting the compliance assessment smart contract to a high rate of consecutive incidents from the monitoring side.

Therefore, they cause the compliance assessment smart contract to conduct several updates on the same asset (performance report). As a result, there is a high chance of transactions that attempt to manipulate the same asset in the same block. Hyperledger Fabric employs MVCC protocol which rejects all transactions, but one, due to a conflict in the read-write sets. This was tested using an earlier version release of Hyperledger Fabric (v1.4). Note that this issue only appears on the surface in the worst-case scenario, when the service provider continuously experiences failures that would cause the monitoring tool to ceaselessly submit transactions to the compliance assessment smart contract. We experimented with the same approach using the latest releases of Hyperledger Fabric. However, there is no significant performance improvement regarding the compliance assessment because they still employ the MVCC protocol, whether the blockchain network is deployed locally or at a scalable infrastructure such as cloud hosting.

The high rate of transactions can be mitigated by applying some workaround techniques such as incident silencing and grouping from the monitoring side. However, we opt to address this on the blockchain side because such techniques may not suit every scenario. For instance, the issue will persist when concurrent monitoring submits transactions simultaneously. Therefore, this thesis sought to increase the dependability of the smart contract and resilience towards such extreme cases. The previous experiment in chapter 4 attempted to address the issue by increasing the rate of transactions processing and blocks generation to cope with the high rate of transactions received from the monitoring side. While it could manage to lower transactions conflicts, it poses a challenge to the dependability of the compliance assessment. Furthermore, the rate increase of block generation does not align well with security recommendations such as the fork possibility because validating nodes face to synchronously accommodate newly generated blocks [76][77].

For that, chapter 5 accounts for the conflicting read-write sets at the design of the smart contract, which demonstrates to eliminate the issue while achieving a considerable performance improvement in terms of throughput and latency. Most importantly, it satisfies the MVCC protocol without requiring specific blocks configurations such as a rate increase of block generation. That is, the design of the smart contract considers unique records for each incident processing, which completely mitigates read-write sets conflicts. It then aggregates these unique records when required (e.g. for billing purposes).

Moreover, we learn the importance of validating the dependability of the smart contract not only in testing settings but also in production. That is, successful validation in a testing environment does not necessarily mean reliability in production settings. Moreover, we cannot assume the applicability of conventional software engineering practice on blockchain-based solutions. This is evident when chapter 4 attempts to update the same record and does not account for the distributed nature of the blockchain network and MVCC mechanism. Subsequently, chapter 5 accounts for the transaction flow journey and thus successfully improved the compliance assessment approach.

By comparing the obtained performance results with those of other blockchain performance studies, we find that the smart contract's logic forms a major influence factor. Most of the related performance studies in section 2.5 ignore the role of smart contracts in performance. That is, they assume a trivial smart contract that conducts lightweight operations. While the reported performance outcomes in their studies are extraordinary, they do not apply to real-world scenarios, where the smart contract's logic is vital to blockchain performance. This thesis considers the smart contract's role in performance benchmarking and presents realistic outcomes. To sum up, blockchain scalability poses a real challenge to high-throughput applications. Therefore, a proper architecture must consider the scalability issue and, as practicably reasonable as possible, avoid overwhelming the blockchain side with the unnecessary transaction.

7.2. Thesis Generality

This thesis revisited a set of SLA-related distrusted processes associated with a typical SLA life cycle. While the ultimate goal of this thesis is to achieve a decentralised compliance assessment, it also decentralises other related matters such as SLA definition, storage, enforcement, billing and termination. Although this thesis ultimately focuses on the domain of application on IoT, its outcomes can be generalised to the concept of SLA in any domain, whether cloud computing, telecommunication, or traditional IT infrastructure.

Concerning the adopted blockchain platform, Hyperledger Fabric implements a unique blockchain philosophy in terms of the level of decentralisation, transaction lifecycle, smart contract life cycle, permissioned network, supported consensus mechanism. Therefore, it is important to recognise that the outcomes of this thesis do not necessarily apply to other blockchain platforms. However, it demonstrates that blockchain in general, regardless of the underlying platform, can improve trust mechanisms and decentralise distrusted SLA tasks. Furthermore, although Hyperledger Fabric influences the design of proposed approaches, they seem to be easily tweakable to suit other blockchain platforms. Therefore, future studies will address the extent of applicability of the current design to other blockchain platforms such as Ethereum.

The middleware proposed in chapter 6 is generic and modular such that it enables the integration of Java-based IoT simulators with a real-world blockchain environment. In principle, the middleware is implemented in Java programming language. Thus, it should be easily imported as a library to any Java-based simulator, such as those in section 2.1.2. It also can be imported to any Java-based simulator regardless of the simulation domain, whether it serves IoT or others. On the other hand, the middleware demonstrates to work with any smart contract deployed on the blockchain network. The middleware workers are loosely-coupled from the logic and methods signature of the smart contract. Therefore, it can be used beyond SLA purposes as long as Hyperledger Fabric is the underlying blockchain platform.

7.3. Threats to Validity and Future Trends

This research effort is primarily motivated by trust issues in the current SLA practice. Therefore, it transformed a set of distrusted tasks into smart contracts that operate independently in a decentralised fashion. Experiments demonstrate that blockchain can improve current trust mechanisms by leveraging blockchain features. For example, smart contract autonomy can automate SLA-related tasks in a non-repudiable manner which can help reduce workforces in terms of incident management and billing. They do not only enhance procedures but also reduce dispute chances to a minimum, thanks to the blockchain's ledger immutability, traceability and transparency.

Hyperledger Fabric implements the above-listed features following its unique philosophy. However, permissioned blockchain platforms are complex in terms of infrastructure deployment, operation and maintenance. Unlike Ethereum, the burden is not limited to the design of smart contracts. Rather, the burden extends to the infrastructure and networking. Therefore, the deployment architecture and operation scheme greatly impact the trust factor assumed in the blockchain. To clarify, consider the blockchain network in Figure 2.7. Therefore, examine the following cases:

- A permissioned blockchain network that is entirely under a single authority.
- A truly decentralised blockchain network, but one operator is assigned for operating all certificate authorities.
- improper endorsement policy enables a participant to upgrade the smart contract.
- One blockchain participant controls the majority of orderers.

Therefore, the decentralisation of Hyperledger Fabric must not be assumed. It should be carefully designed to consolidate the trust in the compliance assessment smart contract.

This thesis demonstrates shifting a set of SLA distrusted processes from single authorities to a non-repudiable environment. While the compliance assessment depends heavily on external monitoring tools, it only assumes a decentralised monitoring system in place and has not investigated in depth a proper decentralised monitoring architecture. However, this thesis suggests revisiting other studies such as Uriarte et al. [110] which seems to address the topic of decentralised monitoring. Moreover, it suggests the concept of decentralised oracles for validating monitoring feeds and ensuring deterministic smart contract execution [106] [108].

Existing monitoring tools such as Prometheus are not readily capable of transacting with the blockchain side. For that, Figure 5.9 considers a server that resides between the monitoring tool and the blockchain side. The server implements Fabric SDK to facilitate authentication, connectivity and communication between the monitoring side and the blockchain side. While this is sufficient for experimental purposes, the server can pose a threat to the compliance assessment such as a server downtime or single authority. Since the blockchain paradigm is still in early

stages, the industry adoption may help accommodate features needed for integration with the blockchain side.

Moreover, the following sections lists a set of suggested topics for further study and future work.

Applicability to Ethereum

While the contributions of this thesis are blockchain-based, they are largely influenced by Hyperledger Fabric philosophy. Refer to section 2.5 for key differences between Hyperledger Fabric and Ethereum. Future work will further empirically examine the influence of Ethereum on the proposed approaches in terms of advantages and limitations. In particular, a future study will consider the influence of Ethereum on this thesis's proposed solutions in terms of the following:

Permissionless Nature

Unlike Hyperledger Fabric, Ethereum is an example of a public network that must incentify anonymous nodes to participate in the network infrastructure. Therefore, transactions require execution fees to cover these nodes' participation. In this model, transactions from monitoring tools would incur a cost. The ultimate question is who should pay for the cost? Additionally, questions arise about the enterprise and government sectors' willingness to adopt such a model where any one can be part of the infrastructure and transparency is a right for anyone, even those beyond the SLA participants. Moreover, the permissionless nature of most public networks requires heavy consensus protocols such as PoW protocol, which impacts the overall performance. Therefore, future studies will empirically examine the practicality of the permissionless blockchain model for the proposed solutions.

Performance Improvement at Blockchain Infrastructure Level

The enhanced compliance assessment mitigates conflicts of read-write sets by improving the smart contract design. This enhancement paves the way for future work to investigate improving the performance at infrastructure level of Hyperledger Fabric (HLF). For example, finding optimal block configurations, which plays a vital role on throughput and latency. HLF's modularity makes it also interesting to study the impact of different aspects on the overall performance, such as network size in terms of organisations, endorsing and committing peers. There is also the ordering service and employed consensus mechanism, chaincode configurations, smart contract programming languages and others. The deployment of the blockchain network may play a role in the performance. For example, the computing capacity of each node in the network and whether they are in the same region or data centre.

Decentralised Monitoring Mechanism

Monitoring forms an integral part of the compliance assessment. While this thesis decentralises most stages of a typical SLA life cycle, it assumes a trusted decentralised monitoring mechanism.

That is, monitoring can threaten the validity of the proposed compliance assessment. Therefore, future work will consider investigating a decentralised monitoring architecture.

Proactive Enforcement

The current enforcement mechanism only reacts to violation consequences after their occurrences. However, it is still important to enforce SLA in proactive manner in order to prevent breaches before their occurrence. Therefore, a future study will also consider an approach that can actively play a role in the service provisioning and its underlying infrastructure. For instance, when the smart contract observes a service degradation in terms of latency, it may instruct an auto-scaling mechanism to add another virtual machine to the cluster.

Interoperability

Different blockchain networks and platforms operate in silos. For example, Hyperledger Fabric networks are not compatible by default with the Ethereum network. Interoperability between both networks enables a hybrid architecture that leverages the best of both.

References

[1] Y. Philipp, J. M. Butler, W. Theilmann, and R. Yahyapour, *Service Level Agreements for Cloud Computing*, P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour, Eds. New York, NY: Springer New York, 2011. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-1614-2

[2] S. Tata, M. Mohamed, T. Sakairi, N. Mandagere, O. Anya, and H. Ludwig, "rSLA: A Service Level Agreement Language for Cloud Services," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, jun 2016, pp. 415–422. [Online]. Available: http://ieeexplore.ieee.org/document/7820299/

[3] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour, "Establishing and Monitoring SLAs in Complex Service Based Systems," in *2009 IEEE International Conference on Web Services*. IEEE, jul 2009, pp. 783–790. [Online]. Available: http://ieeexplore.ieee.org/document/5175897/

[4] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu, "Utility computing SLA management based upon business objectives," *IBM Systems Journal*, vol. 43, no. 1, pp. 159–178, 2004.

[5] P. A. Laplante, T. Costello, P. Singh, S. Bindiganavile, and M. Landon, "The Who, What, Why, Where, and When of IT Outsourcing," *IT Professional*, vol. 6, no. 1, pp. 19–23, jan 2004.

[6] A. Paschke and M. Bichler, "SLA representation, management and enforcement," in *Proceedings - 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, EEE-05*, 2005, pp. 158–163.

[7] L. Wu and R. Buyya, "Service Level Agreement (SLA) in Utility Computing Systems," *Grid and Cloud Computing*, pp. 286–310, oct 2010. [Online]. Available: https://arxiv.org/abs/1010.2881v1

[8] ISO, "ISO/IEC 19086-2:2018 - Cloud computing — Service level agreement (SLA) framework — Part 2: Metric model." [Online]. Available: https://www.iso.org/standard/67546.html

[9] N. Bakalos, D. Kyriazis, E. Protonotarios, T. Varvarigou, O. Barreto, A. Juan, A. Bantouna, P. Demestichas, A. Georgakopoulos, T. Stamati, K. Tsagkaris, and P. Vlacheas, "SLA specification and reference model," 2016.

[10] S. Mubeen, S. A. Asadollah, A. V. Papadopoulos, M. Ashjaei, H. Pei-Breivold, and M. Behnam, "Management of Service Level Agreements for Cloud Services in IoT: A Systematic Mapping Study," *IEEE Access*, vol. 6, pp. 30 184–30 207, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8016558/

[11] A. Alqahtani, E. Solaiman, P. Patel, S. Dustdar, and R. Ranjan, "Service level agreement specification for end-to-end IoT application ecosystems," *Software: Practice and Experience*, vol. 49, no. 12, pp. 1689–1711, dec 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2747https://onlinelibrary.wiley.com/doi/10.1002/spe.2747

[12] S. Girs, S. Sentilles, S. A. Asadollah, M. Ashjaei, and S. Mubeen, "A Systematic Literature Study on Definition and Modeling of Service-Level Agreements for Cloud Services in IoT," pp. 134 498–134 513, 2020.

[13] S. Habib, S. Hauke, S. Ries, and M. Mühlhäuser, "Trust as a facilitator in cloud computing: a survey," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 1, no. 1, p. 19, aug 2012. [Online]. Available: http://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-1-19

[14] R. B. Uriarte, R. de Nicola, and K. Kritikos, "Towards Distributed SLA Management with Smart Contracts and Blockchain," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, dec 2018, pp. 266–271. [Online]. Available: https://ieeexplore.ieee.org/document/8591028/

[15] W. Hussain, F. K. Hussain, and O. K. Hussain, "Maintaining Trust in Cloud Computing through SLA Monitoring," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 2014, vol. 8836, pp. 690–697. [Online]. Available: http://link.springer.com/10.1007/978-3-319-12643-2_83

[16] O. F. Rana, M. Warnier, T. B. Quillinan, F. Brazier, and D. Cojocarasu, "Managing Violations in Service Level Agreements," in *Grid Middleware and Services*. Boston, MA: Springer US, 2008, pp. 349–358. [Online]. Available: http://link.springer.com/10.1007/978-0-387-78446-5-23

[17] D. Kyriazis, "Cloud Computing Service Level Agreements Exploitation of Research Results," European Commission Directorate General Communications Networks, Content and Technology Unit E2 - Software and Services, Cloud, Tech. Rep., 2013. [Online]. Available: https://ec.europa.eu/digital-single-market/en/news/cloud-computing-service-level-agreements-exploitation-research-results

[18] T. Labidi, A. Mtibaa, W. Gaaloul, S. Tata, and F. Gargouri, "Cloud SLA Modeling and Monitoring," in *2017 IEEE International Conference on Services Computing (SCC)*. IEEE, jun 2017, pp. 338–345. [Online]. Available: http://ieeexplore.ieee.org/document/8035003/

[19] A. Alzubaidi, E. Solaiman, P. Patel, and K. Mitra, "Blockchain-Based SLA Management in the Context of IoT," *IT Professional*, vol. 21, no. 4, pp. 33–40, jul 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8764077/

[20] E. J. Scheid, B. B. Rodrigues, L. Z. Granville, and B. Stiller, "Enabling dynamic SLA compensation using blockchain-based smart contracts," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management, IM 2019*, 2019, pp. 53–61.

[21] OMG Cloud Working Group, "Practical Guide to Cloud Service Agreements Version 3.0," Object Management Group, Tech. Rep., 2019. [Online]. Available: https://www.omg.org/cloud/deliverables/Practical-Guide-to-Cloud-Service-Agreements.pdf

[22] N. Neidhardt, C. Köhler, and M. Nüttgens, "Cloud Service Billing and Service Level Agreement Monitoring based on Blockchain," *EMISA Forum*, vol. 38, pp. 46–50, 2018. [Online]. Available: http://ceur-ws.org/Vol-2097/paper11.pdf

[23] J. Huang and D. M. Nicol, "Trust mechanisms for cloud computing," *Journal of Cloud Computing*, vol. 2, no. 1, p. 9, apr 2013. [Online]. Available: http://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-2-9

[24] A. Chandrasekar, K. Chandrasekar, M. Mahadevan, and P. Varalakshmi, "QoS Monitoring and Dynamic Trust Establishment in the Cloud." Springer, Berlin, Heidelberg, 2012, pp. 289–301.

[25] A. Sunyaev, "Distributed Ledger Technology," *Internet Computing*, pp. 265–299, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-34957-8-9

[26] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: www.bitcoin.org

[27] GAVIN WOOD, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[28] V. Buterin, "A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: http://www.the-blockchain.com/docs/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf

[29] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, jan 2018. [Online]. Available: http://arxiv.org/abs/1801.10228http://dx.doi.org/10.1145/3190508.3190538https://dl.acm.org/doi/10.1145/3190508.3190538

[30] H. N. Dai, Z. Zheng, and Y. Zhang, "Blockchain for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076–8094, oct 2019.

[31] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–12, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8643084/

[32] M. Chernyshev, Z. Baig, O. Bello, and S. Zeadally, "Internet of things (IoT): Research, simulators, and testbeds," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 1637–1647, jun 2018.

[33] K. Alwasel, D. N. Jha, F. Habeeb, U. Demirbaga, O. Rana, T. Baker, S. Dustdar, M. Villari, P. James, E. Solaiman, and R. Ranjan, "IoTSim-Osmosis: A framework for modeling and simulating IoT applications over an edge-cloud continuum," *Journal of Systems Architecture*, vol. 116, p. 101956, jun 2021.

[34] A. Alzubaidi, K. Mitra, P. Patel, and E. Solaiman, "A Blockchain-based Approach for Assessing Compliance with SLA-guaranteed IoT Services," in *2020 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE, aug 2020, pp. 213–220. [Online]. Available: https://ieeexplore.ieee.org/document/9192398/

[35] A. Alzubaidi, K. Mitra, and E. Solaiman, "Smart Contract Design Considerations for SLA Compliance Assessment in the Context of IoT," in *2021 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE, aug 2021, pp. 74–81. [Online]. Available: https://ieeexplore.ieee.org/document/9556177/

[36] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 4, pp. 2347–2376, oct 2015.

[37] M. R. Bouakouk, A. Abdelli, and L. Mokdad, "Survey on the Cloud-IoT paradigms: Taxonomy and architectures," in *Proceedings - IEEE Symposium on Computers and Communications*, vol. 2020-July. Institute of Electrical and Electronics Engineers Inc., jul 2020.

[38] IEEE Standards Association., "IEEE Standard for an Architectural Framework for the Internet of Things (IoT)," *IEEE Std 2413-2019*, pp. 1–269, mar 2020.

[39] "ISO - ISO/IEC 30141:2018 - Internet of Things (loT) — Reference Architecture," 2018. [Online]. Available: https://www.iso.org/standard/65695.html

[40] Alliance for Internet of Things Innovation, "IoT High Level Architecture (HLA)," 2016. [Online]. Available: https://aioti.eu/aioti-wg03-reports-on-iot-standards/

[41] I. Makhdoom, M. Abolhasan, H. Abbas, and W. Ni, "Blockchain's adoption in IoT: The challenges, and a way forward," *Journal of Network and Computer Applications*, vol. 125, pp. 251–279, jan 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1084804518303473

[42] OpenFog Consortium Architecture Working Group, "OpenFog Reference Architecture for Fog Computing," *OpenFogConsortium*, no. February, pp. 1–162, 2017. [Online]. Available: https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL.pdf

[43] X. Zeng, S. K. Garg, P. Strazdins, P. P. Jayaraman, D. Georgakopoulos, and R. Ranjan, "IOTSim: A simulator for analysing IoT applications," *Journal of Systems Architecture*, vol. 72, pp. 93–107, jan 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762116300662https://linkinghub.elsevier.com/retrieve/pii/S1383762116300662

[44] "ISO - ISO/IEC 20922:2016 - Information technology — Message Queuing Telemetry Transport (MQTT) v3.1.1." [Online]. Available: https://www.iso.org/standard/69466.html

[45] S. M. Kim, H. S. Choi, and W. S. Rhee, "IoT home gateway for auto-configuration and management of MQTT devices," *2015 IEEE Conference on Wireless Sensors, ICWiSE 2015*, pp. 12–17, aug 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7380346/

[46] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, mar 2012.

[47] C. Kunde and Z. Á. Mann, "Comparison of simulators for fog computing," *Proceedings of the ACM Symposium on Applied Computing*, pp. 1792–1795, mar 2020. [Online]. Available: https://doi.org/10.1145/3341105.3375771

[48] A. Markus and A. Kertesz, "A survey and taxonomy of simulation environments modelling fog computing," *Simulation Modelling Practice and Theory*, vol. 101, p. 102042, may 2020. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1569190X1930173X

[49] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, sep 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2509https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2509https://onlinelibrary.wiley.com/doi/10.1002/spe.2509

[50] M. M. Lopes, W. A. Higashino, M. A. Capretz, and L. F. Bittencourt, "MyiFogSim," in *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*. New York, NY, USA: ACM, dec 2017, pp. 47–52. [Online]. Available: https://doi.org/10.1145/3147234.3148101https://dl.acm.org/doi/10.1145/3147234.3148101

[51] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, jan 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/spe.995https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.995https://onlinelibrary.wiley.com/doi/10.1002/spe.995

[52] D. Kliazovich, P. Bouvry, Y. Audzevich, and S. U. Khan, "GreenCloud: A packet-level simulator of energy-aware cloud computing data centers," *GLOBECOM - IEEE Global Telecommunications Conference*, 2010.

[53] N. Mohan and J. Kangasharju, "Edge-Fog cloud: A distributed cloud for Internet of Things computations," *2016 Cloudification of the Internet of Things, CIoT 2016*, mar 2017.

[54] K. T. Kearney and F. Torelli, "The SLA Model," in *Service Level Agreements for Cloud Computing*. New York, NY: Springer New York, 2011, pp. 43–67. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-1614-2_4

[55] Van der Wees Arthur, C. Daniele, L. Jesus, Edwards Mike, Schifano Nicholas, and S. L. Maddalena, "Cloud Service Level Agreement Standardisation Guidelines," pp. 1–41, 2014. [Online]. Available: https://ec.europa.eu/digital-single-market/en/news/cloud-service-level-agreement-standardisation-guidelines

[56] TMForum, "Enabling End-to-End Cloud SLA Management," TMForum, Tech. Rep., 2014. [Online]. Available: https://www.tmforum.org/resources/technical-report-best-practice/tr178-enabling-end-to-end-cloud-sla-management-v2-0-2/

[57] S. K. Garg, S. Versteeg, and R. Buyya, "A framework for ranking of cloud computing services," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1012–1023, jun 2013.

[58] E. Yaqub, P. Wieder, C. Kotsokalis, V. Mazza, L. Pasquale, J. L. Rueda, S. G. Gómez, and A. E. Chimeno, "A Generic Platform for Conducting SLA Negotiations," *Service Level Agreements for Cloud Computing*, pp. 187–206, 2011. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4614-1614-2_12

[59] "ISO/IEC 19086-1:2016 - Information technology — Cloud computing — Service level agreement (SLA) framework — Part 1: Overview and concepts." [Online]. Available: https://www.iso.org/standard/67545.html

[60] A. Keller and H. Ludwig, "The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003. [Online]. Available: http://link.springer.com/10.1023/A:1022445108617

[61] T. Labidi, A. Mtibaa, W. Gaaloul, and F. Gargouri, "Ontology-based SLA negotiation and re-negotiation for cloud computing," *Proceedings - 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2017*, pp. 36–41, aug 2017.

[62] F. Faniyi and R. Bahsoon, "A Systematic Review of Service Level Management in the Cloud," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1–27, feb 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2843890

[63] H. Zhang, Z. Shao, H. Zheng, and J. Zhai, "Establishing service level agreement requirement based on monitoring," *Proceedings - 2nd International Conference on Cloud and Green Computing and 2nd International Conference on Social Computing and Its Applications, CGC/SCA 2012*, pp. 472–476, 2012.

[64] M. Hogan, F. Liu, A. Sokol, and J. Tong, “Nist cloud computing standards roadmap,” *NIST Special Publication*, vol. 35, pp. 6–11, 2011.

[65] T. S. Wong, G. Y. Chan, and F. F. Chua, “Adaptive Preventive and Remedial Measures in Resolving Cloud Quality of Service Violation,” in *International Conference on Information Networking*, vol. 2019-Janua. IEEE Computer Society, may 2019, pp. 473–479.

[66] F. Nawaz, O. Hussain, F. K. Hussain, N. K. Janjua, M. Saberi, and E. Chang, “Proactive management of SLA violations by capturing relevant external events in a Cloud of Things environment,” *Future Generation Computer Systems*, vol. 95, pp. 26–44, jun 2019.

[67] K.-W. Park, J. Han, J. Chung, and K. H. Park, “THEMIS: A Mutually Verifiable Billing System for the Cloud Computing Environment,” *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 300–313, jul 2013. [Online]. Available: http://ieeexplore.ieee.org/document/6133267/

[68] T. M. Fernandez-Carames and P. Fraga-Lamas, “A Review on the Use of Blockchain for the Internet of Things,” *IEEE Access*, vol. 6, pp. 32 979–33 001, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8370027/

[69] M. Anisetti, C. A. Ardagna, F. Gaudenzi, and E. Damiani, “A certification framework for cloud-based services,” in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, vol. 04-08-Apri. New York, NY, USA: ACM, apr 2016, pp. 440–447. [Online]. Available: https://dl.acm.org/doi/10.1145/2851613.2851628

[70] D. Romano and G. Schmid, “Beyond Bitcoin: A Critical Look at Blockchain-Based Systems,” *Cryptography*, vol. 1, no. 2, p. 15, 2017.

[71] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain technology overview,” National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep., oct 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf

[72] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, “Decentralized Applications: The Blockchain-Empowered Software System,” *IEEE Access*, vol. 6, pp. 53 019–53 033, sep 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8466786/

[73] S. Haber and W. S. Stornetta, “How to time-stamp a digital document,” *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, jan 1991. [Online]. Available: http://link.springer.com/10.1007/BF00196791

[74] S. Seebacher and R. Schüritz, “Blockchain Technology as an Enabler of Service Systems: A Structured Literature Review,” in *Exploring Services Science*, S. Za, M. Druagoicea, and M. Cavallari, Eds. Cham: Springer, Cham, may 2017, pp. 12–23. [Online]. Available: http://link.springer.com/10.1007/978-3-319-56925-3_2

[75] K. Christidis and M. Devetsikiotis, “Blockchains and Smart Contracts for the Internet of Things,” *IEEE Access*, vol. 4, pp. 2292–2303, 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7467408/

[76] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, “Untangling Blockchain: A Data Processing View of Blockchain Systems,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, jul 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8246573/

[77] A. I. Sanka and R. C. Cheung, “A systematic review of blockchain scalability: Issues, solutions, analysis and future research,” *Journal of Network and Computer Applications*, vol. 195, p. 103232, dec 2021.

[78] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173–190, nov 2018.

[79] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, "Performance Analysis of Consensus Algorithm in Private Blockchain," *IEEE Intelligent Vehicles Symposium, Proceedings*, vol. 2018-June, pp. 280–285, oct 2018.

[80] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, pp. 305–320.

[81] D. Chaum, "Blind Signatures for Untraceable Payments," in *Advances in Cryptology*. Boston, MA: Springer US, 1983, pp. 199–203. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4757-0602-4_18http://link.springer.com/10.1007/978-1-4757-0602-4_18

[82] D. Vujičić, D. Jagodić, and S. Randić, "Blockchain technology, bitcoin, and Ethereum: A brief overview," *2018 17th International Symposium on INFOTEH-JAHORINA, INFOTEH 2018 - Proceedings*, vol. 2018-Janua, pp. 1–6, apr 2018.

[83] N. Szabo, "Smart Contracts," 1994. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[84] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart Contract Development: Challenges and Opportunities," *IEEE Transactions on Software Engineering*, pp. 1–1, sep 2019.

[85] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 1513–1520.

[86] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why Do My Blockchain Transactions Fail?" in *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: ACM, jun 2021, pp. 221–234. [Online]. Available: https://dl.acm.org/doi/10.1145/3448016.3452823

[87] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network)," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, nov 2018, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8548070/

[88] Hyperledger, "Hyperledger Blockchain Performance Metrics," Hyperledger Performance and Scale Working Group, Tech. Rep., 2018. [Online]. Available: https://www.hyperledger.org/learn/publications/blockchain-performance-metrics

[89] A. Altarawneh, T. Herschberg, S. Medury, F. Kandah, and A. Skjellum, "Buterin's Scalability Trilemma viewed through a State-change-based Classification for Common Consensus Algorithms," *2020 10th Annual Computing and Communication Workshop and Conference, CCWC 2020*, pp. 727–736, jan 2020.

[90] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*. New York, New York, USA: ACM Press, 2017, pp. 1085–1100. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3035918.3064033

[91] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads," *2017 26th International Conference on Computer Communications and Networks, ICCCN 2017*, sep 2017.

[92] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform," *arXiv preprint arXiv:1805.11390*, 2018.

[93] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance Characterization of Hyperledger Fabric," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, jun 2018, pp. 65–74. [Online]. Available: https://ieeexplore.ieee.org/document/8525394/

[94] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second," in *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency*. Institute of Electrical and Electronics Engineers Inc., may 2019, pp. 455–463.

[95] P. Yuan, K. Zheng, X. Xiong, K. Zhang, and L. Lei, "Performance modeling and analysis of a Hyperledger-based system using GSPN," *Computer Communications*, vol. 153, pp. 117–124, mar 2020.

[96] M. Kuzlu, M. Pipattanasomporn, L. Gurses, and S. Rahman, "Performance Analysis of a Hyperledger Fabric Blockchain Framework: Throughput, Latency and Scalability," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, jul 2019, pp. 536–540. [Online]. Available: https://ieeexplore.ieee.org/document/8946222/

[97] L. Hang and D.-H. Kim, "Optimal blockchain network construction methodology based on analysis of configurable components for enhancing Hyperledger Fabric performance," *Blockchain: Research and Applications*, vol. 2, no. 1, p. 100009, mar 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S209672092100004X

[98] J. Dreyer, M. Fischer, and R. Tönjes, "Performance analysis of hyperledger fabric 2.0 blockchain platform," in *CCIoT 2020 - Proceedings of the 2020 Cloud Continuum Services for Smart IoT Systems, Part of SenSys 2020*. New York, NY, USA: Association for Computing Machinery, Inc, nov 2020, pp. 32–38. [Online]. Available: https://dl.acm.org/doi/10.1145/3417310.3431398

[99] H. Zhou, C. de Laat, and Z. Zhao, "Trustworthy Cloud Service Level Agreement Enforcement with Blockchain Based Smart Contract," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, dec 2018, pp. 255–260. [Online]. Available: https://ieeexplore.ieee.org/document/8591026/

[100] A. Aldweesh, M. Alharby, E. Solaiman, and A. Van Moorsel, "Performance Benchmarking of Smart Contracts to Assess Miner Incentives in Ethereum," *Proceedings - 2018 14th European Dependable Computing Conference, EDCC 2018*, pp. 144–149, nov 2018.

[101] H. Arslanian, *Ethereum*. Cham: Springer International Publishing, 2022, pp. 91–98. [Online]. Available: https://doi.org/10.1007/978-3-030-97951-5_3https://link.springer.com/10.1007/978-3-030-97951-5_3

[102] M. Alharby and A. van Moorsel, "The Impact of Profit Uncertainty on Miner Decisions in Blockchain Systems," *Electronic Notes in Theoretical Computer Science*, vol. 340, pp. 151–167, oct 2018.

[103] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, "Untrusted Business Process Monitoring and Execution Using Blockchain," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer, Cham, sep 2016, vol. 9850 LNCS, pp. 329–347. [Online]. Available: http://link.springer.com/10.1007/978-3-319-45348-4_19

[104] E. Di Pascale, J. McMenamy, I. Macaluso, and L. Doyle, "Smart Contract SLAs for Dense Small-Cell-as-a-Service," mar 2017. [Online]. Available: https://arxiv.org/abs/1703.04502

[105] H. Nakashima and M. Aoyama, "An Automation Method of SLA Contract of Web APIs and Its Platform Based on Blockchain Concept," *Proceedings - 2017 IEEE 1st International Conference on Cognitive Computing, ICCC 2017*, pp. 32–39, jun 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8029220/

[106] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. Di Ciccio, I. Weber, M. Wöhrer, and U. Zdun, "Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World," in *Lecture Notes in Business Information Processing*, vol. 393 LNBIP. Springer Science and Business Media Deutschland GmbH, sep 2020, pp. 35–51. [Online]. Available: https://doi.org/10.1007/978-3-030-58779-6_3

[107] R. B. Uriarte, F. Tiezzi, and R. D. Nicola, "SLAC: A Formal Service-Level-Agreement Language for Cloud Computing," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE, dec 2014, pp. 419–426. [Online]. Available: http://ieeexplore.ieee.org/document/7027520/

[108] X. Xu, C. Pautasso, L. Zhu, V. Gramoli, A. Ponomarev, A. B. Tran, and S. Chen, "The Blockchain as a Software Connector," in *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, apr 2016, pp. 182–191. [Online]. Available: http://ieeexplore.ieee.org/document/7516828/

[109] M. Taghavi, J. Bentahar, H. Otrok, and K. Bakhtiyari, "A Blockchain-Based Model for Cloud Service Quality Monitoring," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 276–288, mar 2020.

[110] R. B. Uriarte, H. Zhou, K. Kritikos, Z. Shi, Z. Zhao, and R. De Nicola, "Distributed service-level agreement management with smart contracts and blockchain," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 14, jul 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/cpe.5800

[111] E. D. Pascale, H. Ahmadi, L. Doyle, and I. Macaluso, "Toward Scalable User-Deployed Ultra-Dense Networks: Blockchain-Enabled Small Cells as a Service," *IEEE Communications Magazine*, vol. 58, no. 8, pp. 82–88, aug 2020.

[112] M. S. Rahman, I. Khalil, and M. Atiquzzaman, "Blockchain-Enabled SLA Compliance for Crowdsourced Edge-Based Network Function Virtualization," *IEEE Network*, vol. 35, no. 5, pp. 58–65, sep 2021.

[113] L. De Marco, F. Ferrucci, and M.-T. Kechadi, "SLAFM - A Service Level Agreement Formal Model for Cloud Computing," in *Proceedings of the 5th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and and Technology Publications, 2015, pp. 521–528. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005451805210528

[114] A. T. Wonjiga, S. Peisert, L. Rilling, and C. Morin, "Blockchain as a Trusted Component in Cloud SLA Verification," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion - UCC '19 Companion*. New York, New York, USA: ACM Press, 2019. [Online]. Available: https://doi.org/10.1145/3368235.3368872

[115] K. Singi, V. Kaulgud, R. P. Jagadeesh Chandra Bose, and S. Podder, "CAG: Compliance adherence and governance in software delivery using blockchain," in *Proceedings - 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2019*. Institute of Electrical and Electronics Engineers Inc., may 2019, pp. 32–39.

[116] N. Kapsoulis, A. Psychas, A. Litke, and T. Varvarigou, "Reinforcing SLA Consensus on Blockchain," *Computers*, vol. 10, no. 12, p. 159, nov 2021. [Online]. Available: https://www.mdpi.com/2073-431X/10/12/159/htmhttps://www.mdpi.com/2073-431X/10/12/159

[117] J. Benet, "IPFS - Content Addressed, Versioned, P2P File System," jul 2014. [Online]. Available: https://arxiv.org/abs/1407.3561http://arxiv.org/abs/1407.3561

[118] P. Kochovski, V. Stankovski, S. Gec, F. Faticanti, M. Savi, D. Siracusa, and S. Kum, "Smart Contracts for Service-Level Agreements in Edge-to-Cloud Computing," *Journal of Grid Computing*, vol. 18, no. 4, pp. 673–690, dec 2020. [Online]. Available: https://doi.org/10.1007/s10723-020-09534-y

[119] B. Marino and A. Juels, "Setting standards for altering and undoing smart contracts," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9718. Springer Verlag, 2016, pp. 151–166.

[120] E. Androulaki, A. De Caro, M. Neugschwandtner, and A. Sorniotti, "Endorsement in hyperledger fabric," *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, pp. 510–519, jul 2019.

[121] K. S. S. Wai, E. C. Htoon, and N. N. M. Thein, "Storage Structure of Student Record based on Hyperledger Fabric Blockchain," *2019 International Conference on Advanced Information Technologies, ICAIT 2019*, pp. 108–113, nov 2019.

[122] G. Chung, L. Desrosiers, M. Gupta, A. Sutton, K. Venkatadri, O. Wong, and G. Zugic, "Performance Tuning and Scaling Enterprise Blockchain Applications," dec 2019. [Online]. Available: https://arxiv.org/abs/1912.11456

[123] C. Cachin, "Architecture of the Hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. July, 2016. [Online]. Available: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf

[124] K. M. Khan, J. Arshad, W. Iqbal, S. Abdullah, and H. Zaib, "Blockchain-enabled real-time SLA monitoring for cloud-hosted services," *Cluster Computing*, pp. 1–23, oct 2021. [Online]. Available: https://link.springer.com/article/10.1007/s10586-021-03416-y

[125] P. Patel, A. H. Ranabahu, and A. P. Sheth, "Service level agreement in cloud computing," *Kno.e.sis Publications*, jan 2009. [Online]. Available: https://corescholar.libraries.wright.edu/knoesis/78

[126] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 465–483, dec 1983. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/319996.319998https://dl.acm.org/doi/10.1145/319996.319998

[127] H. Meir, A. Barger, Y. Manevich, and Y. Tock, "Lockless Transaction Isolation in Hyperledger Fabric," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, jul 2019, pp. 59–66. [Online]. Available: https://ieeexplore.ieee.org/document/8946157/

[128] P. Abhishek, A. Chobari, and D. G. Narayan, "SLA Violation Detection in Multi-Cloud Environment using Hyperledger Fabric Blockchain," *2021 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pp. 107–112, nov 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9663620/

[129] D. Magazzeni, P. McBurney, and W. Nash, "Validation and Verification of Smart Contracts: A Research Agenda," *Computer*, vol. 50, no. 9, pp. 50–57, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8048663/

[130] A.-M. Šimundić, "Measures of Diagnostic Accuracy: Basic Definitions," *EJIFCC*, vol. 19, no. 4, p. 203, jan 2009. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4975285/

[131] H. Meir, A. Barger, Y. Manevich, and Y. Tock, "Lockless Transaction Isolation in Hyperledger Fabric," in *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*. IEEE, 2019, pp. 59–66. [Online]. Available: https://doi.org/10.1109/Blockchain.2019.00017

[132] A. K. Pandey, N. D. G., and S. K., "SLA Violation Detection and Compensation in Cloud Environment using Blockchain," in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*. IEEE, jul 2021, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9580134/

[133] B. Brazil, *Prometheus: Up and Running: Infrastructure and Application Performance Monitoring*. O'Reilly Media, Inc., 2018.

[134] A. Stanciu, "Blockchain Based Distributed Control System for Edge Computing," in *Proceedings - 2017 21st International Conference on Control Systems and Computer, CSCS 2017*. Institute of Electrical and Electronics Engineers Inc., jul 2017, pp. 667–671.

[135] O. Khalaf Mohammed, O. Bayat, and H. M. Marhoon, "Design and implementation of integrated security and safety system based on internet of things," *International Journal of Engineering and Technology*, vol. 7, no. 4, pp. 5705–5711, apr 2018. [Online]. Available: www.sciencepubco.com/index.php/IJET

Appendix A. Miscellaneous

A.1. Related Achievement

Figure A.1 Nomination for best paper candidate IEEE SmartIoT Conference 2020

Figure A.2 Nomination for best paper candidate IEEE SmartIoT Conference 2021

A.2. Sample SLA-guaranteed Cloud-based IoT Services

A.2.1. Amazon Web Service (AWS) SLA

AWS IoT Core Service Level Agreement

Last Updated: March 19, 2019

This AWS IoT Core Service Level Agreement ("SLA") is a policy governing the use of AWS IoT Core and applies separately to each account using AWS IoT Core. In the event of a conflict between the terms of this SLA and the terms of the AWS Customer Agreement or other agreement with us governing your use of our Services (the "Agreement"), the terms and conditions of this SLA apply, but only to the extent of such conflict. Capitalized terms used herein but not defined herein shall have the meanings set forth in the Agreement.

Service Commitment

AWS will use commercially reasonable efforts to make AWS IoT Core available with a Monthly Uptime Percentage for each AWS region, during any monthly billing cycle, of at least 99.9% (the "Service Commitment"). In the event AWS IoT Core does not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Service Credits

Service Credits are calculated as a percentage of the total charges paid by you for AWS IoT Core in the affected AWS region for the monthly billing cycle in which the Monthly Uptime Percentage fell within the ranges set forth in the table below:

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.9% but greater than or equal to 99.0%	10%
Less than 99.0% but greater than or equal to 95.0%	25%

Less than 95.0%	100%

We will apply any Service Credits only against future AWS IoT Core payments otherwise due from you. At our discretion, we may issue the Service Credits to the credit card you used to pay for the billing cycle in which the unavailability occurred. Service Credits will not entitle you to any refund or other payment from AWS. Service Credits will be applicable and issued only if the credit amount for the applicable monthly billing cycle is greater than one dollar ($1 USD). Service Credits may not be transferred or applied to any other account. Unless otherwise provided in the Agreement, your sole and exclusive remedy for any unavailability or non-performance or other failure by us to provide AWS IoT Core is the receipt of Service Credits (if eligible) in accordance with the terms of this SLA.

Credit Request and Payment Procedures

To receive Service Credits, you will need to submit a claim by opening a case in the AWS Support Center. To be eligible, the credit request must be received by us by the end of the second billing cycle after which the incident occurred and must include:

(i) the words "SLA Credit Request" in the subject line;

(ii) the billing cycle and AWS region(s) with respect to which you are claiming Service Credits, together with the Monthly Uptime Percentage for that AWS region for the billing cycle and the specific dates, times, and Availabilities for each 5-minute interval with less than 100% Availability in that AWS region throughout the billing cycle;

(iii) your Request logs that document the errors for your claimed outage (any confidential or sensitive information in these logs should be removed or replaced with asterisks).

If the Monthly Uptime Percentage of such credit request is confirmed by us and is less than the Service Commitment, then we will issue the Service Credits to you within one billing cycle following the month in which the credit request occurred. Your failure to provide the credit request and other information as required above will disqualify you from receiving Service Credits.

AWS IoT Core SLA Exclusions

The Service Commitment does not apply to any unavailability, suspension or termination of AWS IoT Core, or any other AWS IoT Core performance issues: (i) caused by factors outside of our reasonable control, including any force majeure event or Internet access or related problems beyond the demarcation point of AWS IoT Core; (ii) that result from any voluntary actions or inactions from you or any third party (e.g. scaling of provisioned capacity, misconfiguring security groups, VPC configurations or credential settings, disabling certificates or making the certificates inaccessible, etc.); (iii) that result from capacity or availability issues of systems in the control of you or any third-party (e.g. insufficient

capacity for rules engine action targets, device message queues, etc.); (iv) that result from you not following the best practices described in the AWS IoT Core Developer Guide on the AWS Site; (v) that result from your equipment, software or other technology and/or third-party equipment, software or other technology (other than third party equipment within our direct control); (vi) that result from exceeding the published AWS IoT Core service limits as set forth in the Documentation on the AWS Site; or (vii) arising from our suspension or termination of your right to use AWS IoT Core in accordance with the Agreement (collectively, the "AWS IoT Core SLA Exclusions").

If availability is impacted by factors other than those explicitly used in our Monthly Uptime Percentage calculation, then we may issue a Service Credit considering such factors at our discretion.

Definitions

- "Availability" is calculated for each 5-minute interval as the percentage of Requests processed by AWS IoT Core that do not fail with Errors . If you did not make any Requests in a given 5-minute interval, that interval is assumed to be 100% available.
- An "Error" is:
  - any HTTP API Request that returns a 500 or 503 error code;
  - a device fails to connect to AWS IoT Core using best practices for retry and exponential back-off;
  - a MQTT Publish Message inbound (from client to service) is Published as QoS1 ("At least once") and the service does not acknowledge (PUBACK) it;
  - a MQTT Publish Message inbound (from client to service) the topic of which is subscribed to by a rule does not trigger the rule; or
  - a MQTT Publish Message outbound (from service to client) has not been delivered to a permanently connected client, successfully subscribed to the message's topic, within one hour of a Request (such Request and Error are deemed to have both occurred in the five minute interval immediately following the one hour window)
- "Monthly Uptime Percentage" for a given AWS region is calculated as the average of the Availability for all 5-minute intervals in a monthly billing cycle. Monthly Uptime Percentage measurements exclude downtime resulting directly or indirectly from any AWS IoT Core SLA Exclusion.
- "Request" is an invocation of an IoT HTTP API or the sending or receiving of a message over MQTT or Websockets.
- A "Service Credit" is a dollar credit, calculated as set forth above, that we may credit back to an eligible account.

Prior Version(s): Link

A.2.2. Google Cloud Platform (GCP) SLA

Back to Google Cloud Terms Directory (https://cloud.google.com/product-terms) > Current

IoT Core Service Level Agreement (SLA)

During the Term of the agreement under which Google has agreed to provide Google Cloud Platform to Customer (as applicable, the "Agreement"), the Covered Service will provide a Monthly Uptime Percentage to Customer as follows (the "Service Level Objective" or "SLO"):

Covered Service	Monthly Uptime Percentage
IoT Core Service	>= 99.9%

If Google does not meet the SLO, and if Customer meets its obligations under this SLA, Customer will be eligible to receive the Financial Credits described below. This SLA states Customer's sole and exclusive remedy for any failure by Google to meet the SLO. Capitalized terms used in this SLA, but not defined in this SLA, have the meaning stated in the Agreement. If the Agreement authorizes the resale or supply of Google Cloud Platform under a Google Cloud partner or reseller program, then all references to Customer in this SLA mean Partner or Reseller (as applicable), and any Financial Credit(s) will only apply for impacted Partner or Reseller order(s) under the Agreement.

Definitions

The following definitions apply to the SLA:

- **"Back-off Requirements"** means, when an error occurs, the devices are responsible for waiting for a period of time before issuing another request. This means that after the first error, there is a minimum back-off interval of 1 second and for each consecutive error, the back-off interval increases exponentially up to 32 seconds.
- **"Covered Service"** means the IoT Core Service.
- **"Downtime"** means more than a 10% Error Rate for the IoT Core device manager or protocol bridge component. Downtime is measured based on server-side Error Rate.

- "**Downtime Period**" means a period of five or more consecutive minutes of Downtime. Partial minutes will not be counted towards any Downtime Periods.
- "**Error Rate**" means:
  - for the IoT Core device manager component and IoT Core protocol bridge (HTTP) component, the number of Valid Requests that result in a response with HTTP Status 50x and Code "Internal Error" divided by the total number of Valid Requests during that period; and
  - for the IoT Core protocol bridge (MQTT) component, the number of Valid Requests that result in device disconnections as reported in Google Stackdriver metrics (or other similar metrics made available to Customer), divided by the total number of Valid Requests during that period.

  Repeated identical requests do not count toward the Error Rate unless they conform to the Back-off Requirements.
- "**Financial Credit**" means the credit amount based on the percentage of the monthly bill for the Covered Service in the table below.

Monthly Uptime Percentage	Percentage of the monthly bill for the Covered Service that will be credited to future monthly Customer bills
99% to < 99.9%	10%
95% to < 99%	25%
< 95%	50%

- "**Monthly Uptime Percentage**" means total number of minutes in a month, minus the number of minutes of Downtime suffered from all Downtime Periods for the Covered Service in a month, divided by the total number of minutes in a month.
- "**Valid Requests**" are requests that conform to the Documentation, and that would normally result in a non-error response.

Customer Must Request Financial Credit

To receive any of the Financial Credits described above, Customer must notify Google technical support (https://support.google.com/cloud/contact/cloud_platform_sla) within 30 days from the time Customer becomes eligible to receive a Financial Credit. Customer must also provide Google with identifying information (e.g., project ID and device registry IDs) and the date and time those errors occurred. If Customer does not comply with these requirements, Customer will forfeit its right to receive a Financial Credit. If a dispute arises with respect to this SLA, Google will make a determination in good faith based on its system logs, monitoring reports, configuration records, and other available information, which Google will make available to Customer at Customer's request.

Maximum Financial Credit

The total maximum number of Financial Credits to be issued by Google to Customer for any and all Downtime Periods that occur in a single billing month will not exceed 50% of the amount due by Customer for the Covered Service for the applicable month. Financial Credits will be made in the form of a monetary credit applied to future use of the Service and will be applied within 60 days after the Financial Credit was requested.

SLA Exclusions

The SLA does not apply to any: (a) features or services designated Alpha or Beta (unless otherwise stated in the associated Documentation), (b) features or services excluded from the SLA (in the associated Documentation), or (c) errors: (i) caused by factors outside of Google's reasonable control; (ii) that resulted from Customer's software or hardware or third party software or hardware, or both; (iii) that resulted from abuses or other behaviors that violate the Agreement; (iv) that resulted from quotas applied by the system or listed in the Admin Console; or (v) that resulted from Customer use of the Covered Service in a way which is inconsistent with the Documentation, including invalid request fields, unauthorized users, or inaccessible data.

PREVIOUS VERSIONS *(Last modified January 13, 2020)*

July 23, 2018 _ (/iot/sla-20180723)

A.2.3. Microsoft Azure SLA

SLA for Azure IoT Hub

Updated: 04/2019

For IoT Hub, we promise that at least 99.9% of the time deployed IoT hubs will be able to send messages to and receive messages from registered devices and the Service will able to perform create, read, update, and delete operations on IoT hubs.

No SLA is provided for the Free Tier of IoT Hub.

In this article:

Introduction

General Terms

The SLA details

Introduction

This Service Level Agreement for Azure (this “SLA”) is made by 21Vianet in connection with, and is a part of, the agreement under which Customer has purchased Azure Services from 21Vianet (the “Agreement”).

We provide financial backing to our commitment to achieve and maintain Service Levels for our Services. If we do not achieve and maintain the Service Levels for each Service as described in this SLA, then you may be eligible for a credit towards a portion of your monthly service fees. These terms will be fixed for term of your Agreement. If a subscription is renewed, the version of this SLA that is current at the time the renewal term commences will apply throughout the renewal term. We will provide at least 90 days' notice for adverse material changes to this SLA. You can review the most current version of this SLA at any time by visiting https://www.azure.cn/support/legal/sla/ .

General Terms

一、Definitions

1. "Claim" means a claim submitted by Customer to 21Vianet pursuant to this SLA that a Service Level has not been met and that a Service Credit may be due to Customer.
2. "Customer" refers to the organization that has entered into the Agreement.
3. "Customer Support" means the services by which 21Vianet may provide assistance to Customer to resolve issues with the Services.
4. "Error Code" means an indication that an operation has failed, such as an HTTP status code in the 5xx range.
5. "External Connectivity" is bi-directional network traffic over supported protocols such as HTTP and HTTPS that can be sent and received from a public IP address.
6. "Incident" means any set of circumstances resulting in a failure to meet a Service Level.
7. "Management Portal" means the web interface, provided by 21Vianet, through which customers may manage the Service.
8. "21Vianet" means the 21Vianet entity that appears on Customer's Agreement.
9. "Preview" refers to a preview, beta, or other pre-release version of a service or software offered to obtain customer feedback.
10. "Service” or “Services" refers to a Azure service provided to Customer pursuant to the Agreement for which an SLA is provided below.
11. "Service Credit" is the percentage of the monthly service fees for the affected Service or Service Resource that is credited to Customer for a validated Claim.
12. "Service Level" means standards 21Vianet chooses to adhere to and by which it measures the level of service it provides for each Service as specifically set forth

below.

13. "Service Resource" means an individual resource available for use within a Service.
14. "Success Code" means an indication that an operation has succeeded, such as an HTTP status code in the 2xx range.
15. "Support Window" refers to the period of time during which a Service feature or compatibility with a separate product or service is supported.
16. "Virtual Network" refers to a virtual private network that includes a collection of user-defined IP addresses and subnets that form a network boundary within Azure.
17. "Virtual Network Gateway" refers to a gateway that facilitates cross-premises connectivity between a Virtual Network and a customer on-premises network.

二、Service Credit Claims

1. In order for 21Vianet to consider a Claim, Customer must submit the Claim to Customer Support within two months of the end of the billing month in which the Incident that is the subject of the Claim occurs. Customer must provide to Customer Support all information necessary for 21Vianet to validate the Claim, including but not limited to detailed descriptions of the Incident, the time and duration of the Incident, the affected resources or operations, and any attempts made by Customer to resolve the Incident
2. 21Vianet will use all information reasonably available to it to validate the Claim and to determine whether any Service Credits are due.
3. In the event that more than one Service Level for a particular Service is not met because of the same Incident, Customer must choose only one Service Level under which a Claim may be made based on the Incident.
4. Service Credits apply only to fees paid for the particular Service, Service Resource, or Service tier for which a

Service Level has not been met. In cases where Service Levels apply to individual Service Resources or to separate Service tiers, Service Credits apply only to fees paid for the affected Service Resource or Service tier, as applicable.

三、SLA Exclusions

This SLA and any applicable Service Levels do not apply to any performance or availability issues:

1. Due to factors outside 21Vianet’s reasonable control (for example, a network or device failure external to 21Vianet’s data centers, including at Customer's site or between Customer's site and 21Vianet’s data center);
2. That resulted from Customer's use of hardware, software, or services not provided by 21Vianet as part of the Services (for example, third–party software or services purchased from the Azure Store or other non–Azure services provided by 21Vianet);
3. Due to Customer's use of the Service in a manner inconsistent with the features and functionality of the Service (for example, attempts to perform operations that are not supported) or inconsistent with published documentation or guidance;
4. That resulted from faulty input, instructions, or arguments (for example, requests to access files that do not exist);
5. Caused by Customer's use of the Service after 21Vianet advised Customer to modify its use of the Service, if Customer did not modify its use as advised;
6. During or with respect to Previews or to purchases made using 21Vianet subscription credits;
7. That resulted from Customer's attempts to perform operations that exceed prescribed quotas or that resulted from throttling of suspected abusive behavior;
8. Due to Customer's use Service features that are outside of associated Support Windows; or

9. Attributable to acts by persons gaining unauthorized access to 21Vianet’s Service by means of Customer's passwords or equipment or otherwise resulting from Customer's failure to follow appropriate security practices.

四、Service Credits

1. The amount and method of calculation of Service Credits is described below in connection with each Service.
2. Service Credits are Customer's sole and exclusive remedy for any failure to meet any Service Level.
3. The Service Credits awarded in any billing month for a particular Service or Service Resource will not, under any circumstance, exceed Customer's monthly service fees that Service or Service Resource, as applicable, in the billing month.
4. For Services purchased as part of a suite, the Service Credit will be based on the pro-rata portion of the cost of the Service, as determined by 21Vianet in its reasonable discretion. In cases where Customer has purchased Services from a reseller, the Service Credit will be based on the estimated retail price for the applicable Service, as determined by 21Vianet in its reasonable discretion.

The SLA details

Additional Definitions

1. “**Deployment Minutes**” is the total number of minutes that a given IoT hub has been deployed in Azure during a billing month.
2. “**Maximum Available Minutes**” is the sum of all Deployment Minutes across all IoT hubs deployed in a given Azure subscription during a billing month.
3. “**Message**” refers to any content sent by a deployed IoT hub to a device registered to the IoT hub or received by

the IoT hub from a registered device, using any protocol supported by the Service.

4. "**Device Identity Operations**" refers to create, read, update, and delete operations performed on the device identity registry of an IoT hub.
5. Downtime : The total accumulated Deployment Minutes, across all IoT hubs deployed in a given Azure subscription, during which the IoT hub is unavailable. A minute is considered unavailable for a given IoT hub if all continuous attempts to send or receive Messages or perform Device Identity Operations on the IoT hub throughout the minute either return an Error Code or do not result in a Success Code within five minutes.
6. Monthly Uptime Percentage: The Monthly Uptime Percentage is calculated using the following formula:

```
Monthly Uptime % = (Maximum Available Minutes – Downtime
```

7. Service Credit:

Monthly Uptime Percentage	Service Credit
<99.9%	10%
<99%	25%

8. Service Level Exceptions: The Free Tier of the IoT Hub Service is not covered by this SLA.

Monthly Uptime Calculation and Service Levels for IoT Hub Device Provisioning Service

1. **"Maximum Available Minutes"** is the total number of minutes for a given Device Provisioning Service deployed by the Customer in a Microsoft Azure subscription during a billing month.
2. **"Downtime"** is the total number of minutes within the

Maximum Available Minutes during which Device Provisioning Service is unavailable. A minute is considered unavailable for a given Device Provisioning Service if all continuous attempts to register a device or perform enrollment/registration record operations on the Device Provisioning Service throughout the minute either return an Error Code or do not result in a Success Code within two minutes.

3. **Monthly Uptime Percentage**: The Monthly Uptime Percentage is calculated using the following formula:
4. Monthly Uptime % = (Maximum Available Minutes – Downtime) / Maximum Available Minutes X 100
5. The following Service Levels and Service Credits are applicable to Customer's use of IoT Hub Device Provisioning Service
6. Service Credit:

MONTHLY UPTIME PERCENTAGE	SERVICE CREDIT
<99.9%	10%
<99%	25%

Follow us via WeChat Public Account

Appendix B. Description of the IoT-based Fire Mitigation System

B.1. Description of the Example IoT System

The source code of the implementation of both the IoT system and our blockchain-based approach is publicly available under GNU GPL V3.0 License on GitHub[1]. We hope they altogether form a base that can help other interested researchers and industry alike to advance the usage of blockchain for mitigating disputes in the context of IoT. It also can be used to experiment using a real IoT system and industry-standard monitoring system.

This section describes a basic IoT ecosystem (an IoT-based fire fighting system) that we designed and implemented for examination purposes, and to clearly define the scope of this study.

B.1.1. Fire detection and Alert Processing

the IoT-based fire mitigation system presented in Figure 5.3 is centred around fire detection and alert processing. Therefore, this system is event-driven in nature, such that once a threshold is reached, an alert is triggered. For that, sensors readings are evaluated in real-time against certain thresholds to help forming a decision on whether to trigger a set of actions. For example, if the reading of the flame sensor reached the specified threshold, An alert will be issued to notify the fire station. Following, we overview main workflow of defecting, forming a decision and reporting fire events processes at both levels, the edge and IoT server.

B.1.1.1. Alerting Logic at The Edge-Side

In our presented scenario, the edge layer is one of the responsibilities assigned to the IoTSP. The edge computing unit conducts the logic presented in Figure B.1.

The main task for it is to ceaselessly observe the environment in real-time (e.g. every second) and investigate, reasonably enough for demonstration purposes, whether a fire alert should be issued. Once a fire event is detected, it instantiates a fire alert and informs the IoT server for a possible fire event. That is, the alert will not be confirmed immediately in order to prevent false positive. To form a decision on the validity of alert, the edge will subsequently undergo an evaluation processes within a specified duration. As a result, the edge will follow the initial alert with a confirmation or cancellation to the IoT server.

[1] https://github.com/aakzubaidi/BlockchainQoT

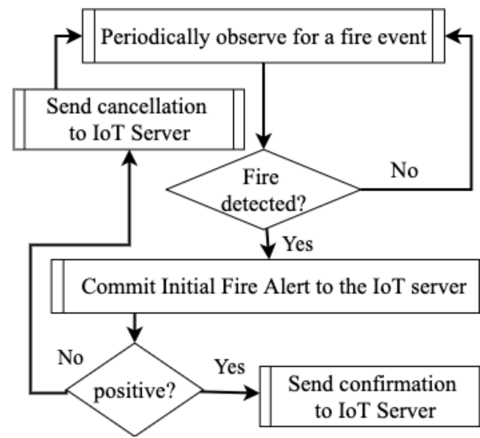


Figure B.1 Fire detection and alerting at edge level.

B.1.1.2. Alerting Logic at The Server-side

The responsibility for the IoT server is also assigned to the IoTSP. One of the IoT server tasks is to listen for fire events. Whenever it receives an initial alert, it expects to receive further exchange within the specified duration about the same alert; as to whether confirm or discard the initial alert. If confirmation is received, then the IoT server must report the fire event to the fire station; otherwise, it will discard it. In th event that no further exchange is sent by the edge computing unit, it will assume the worst case scenario as a safety measure. For instance, the reported fire has been extended to edge computing unit or the Internet gateway. In this case, the IoT server will take the initiative to self-confirm the alert and, which will trigger the action of sending it to the fire station.

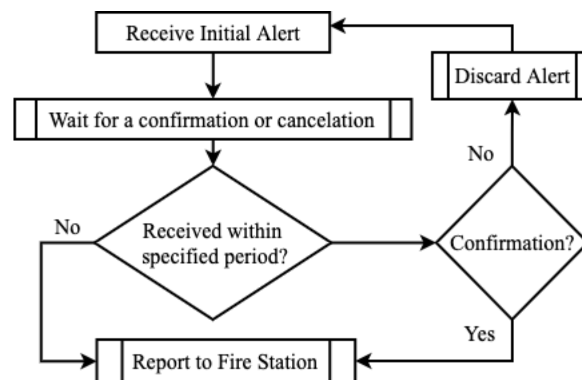
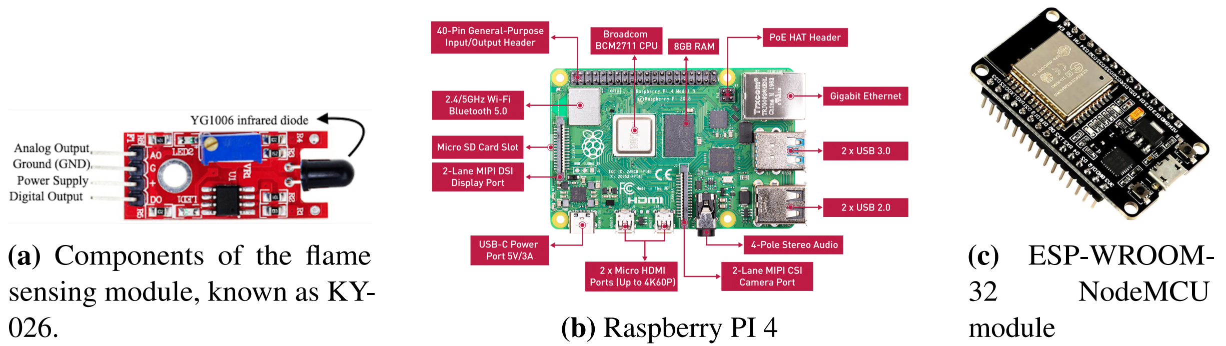


Figure B.2 Alert handling at IoT sever level

B.1.2. Fire Detection

For this study, we employed a real flame sensing module, conventionally known as KY-026. Although it does not scale to industrial level in terms of specification or coverage, we deem it sufficient for the purposes of detecting infrared flames radiation. For that, it depends on optical detection method using YG1006 infrared diode, typically able to identify wavebands within the range of $\approx$0.7 µm to $\approx$1.1 µm [135].



(a) Components of the flame sensing module, known as KY-026.

(b) Raspberry PI 4

(c) ESP-WROOM-32 NodeMCU module

Figure B.3 A set of hardware components considered for the edge layer

Figure B.3a illustrates both the pinouts and the infrared diode of the flame sensing module. In our study we do not make use of the analog output since the digital output is sufficient for our purposes, which emits signals that are either true or false as illustrated in Table B.1. That is, once the infrared diode detects a flame we expect the digital output to emit true; otherwise false.

Table B.1 Mapping flame states to digital outputs

Active Flame	Infrared Radiation	Logical Output
Yes	High	True
No	Low	False

B.1.3. Edge Computing Unit

We understand that, the flame sensing modules, presented in section B.1.2 is capable of observing its environment and emitting data about the flame activities. While essential, it lacks several capabilities such as sufficient processing, memory and storage. It is also short of connectivity and communication methods required for publishing machine-readable data over the internet. For example, but not limited to,

- such a sensing module is not IP-enabled, and thus does not support transfer protocols such as TCP or UDP and typical application protocols such as HTTP or MQTT.
- it lacks connectivity mechanism such as Ethernet, Wifi, Bluetooth, ZigBee, LoRa etc.
- it does not support machine-readable representation protocols such as XML or JSON.

Such shortcomings can be complemented by physically connecting to external capable components. To this end, a wide variety of alternatives can be employed, for bridging the gap and expanding further capabilities. Alternatives can includes, but not limited to, Microcontroller-based development boards (μCU), Single-Board Computer (*SBC*), extended computing units, gateways, etc. Depending on distinctive edge requirements, these can be selected individually or combined to serve different purposes.

Table B.2 Related analysis factors between Raspberry PI4 and ESP-WROON-32

Factor	Raspberry PI 4	ESP-WROOM-32
Type	*SBC*	μCU
Processing unit	Quad core Cortex-A72 ARM 64-bit @ 1.5GHz	Dual-core Xtensa LX6 32-bit @ 240 MHz
Memory	8GB – LPDDR4	512 KB SRAM
Persistence storage	SD CARD	N/A
Operating systems	Unix-like	FreeRTOS
GPIO support	yes	yes
Wifi Connectivity	802.11b/g/n/ac 2.4/5GHz	802.11 b/g/n only 2.4 5GHz
Output voltage	3.3V and 5V	only 3.3V

B.1.3.1. Hardware Selection

in the presented scenario, an internet connectivity gateway (e.g. a typical WiFi router) and reasonable computing units such as *SBC*, or μCU, are sufficient for implementing the the edge layer of our scenario. Popular examples of *SBC* and μCU are Raspberry PI (RPI4), as shown in Figures B.3b, and ESP32 Nodemcu as shown in Figure B.3c; respectively. See both in and However, we selected the former over the latter because it provides adequate capabilities needed for demonstrating our monitoring approach, as will be discussed in section 5.3.3.

Table B.2 presents most relevant factors that we considered for running a comparative analysis between RPI 4 and ESP-WROOM-32 Nodemcu. Both can be qualified candidates for implementing the intended logic of the edge layer for many reasons, First, both of them provide General-Purpose Input/Output (GPIO) header compatible for integrating with and connecting to the flame sensing module. For instance, using their GPIOs enable data acquisition from the flame module and provide a power supply of 3.3V needed for operating it. Second, they provided connectivity to the internet via a Wifi interface.

It is a fact that, both of them can provide processing sufficient for realising an HTTP client and handling our basic edge implementation. Nevertheless, since the ESP-WROOM-32 is specifically designed to be constrained and cost-effective, it lacks native support for adequate local storage, memory and processing capabilities. It also does not support general purposes operating systems required for our approach for edge monitoring; as will be explained in following sections. For that, we opted for Raspberry PI 4 to serve as follows:

- acts as a computing hub for the flame module.
- hosts and operates HTTP client implementation.
- implement our edge monitoring approach.

B.1.3.2. Implemented Edge Logic

Being a client to the IoT server, it seeks authentication from the IoT server for itself as well as for associated assets (the flame sensor). It facilitate communication with the IoT server in a machine readable format; namely JSON. Whenever it classifies sensor data to be alarming, it must notifies the IoT server. When a first positive fire event is identified, the edge emits an initial alert to the server. In order to reduce false positives and negatives to minimum as possible, the edge does not issue a confirmed fire alert unless whichever of the following occurs first:

- a predefined quantity of consecutive positive sensor reading occur within an imposed timeframe, which increases the probability of a fire event.
- the time-frame elapses since the first positive flame reading, but no subsequent sensor readings received, possibly because of a damage occur to the sensor since the last positive reading.

Algorithm 15 Edge Layer: simple event-trigger logic

```
Input: s                                              ▷ sensor reading every second
Output: Initial || Confirmation || Discarded
 1: Intial Alert ← flase                              ▷ flag indicating an an initial alert
 2: Counter ← 0                                       ▷ number of consecutive fire alert
 3: Register to an IoT Server
 4: while True do                                     ▷ continuos operation
 5:     if s ← true then                              ▷ Fire event
 6:         Intial Alert ← true
 7:         counter++
 8:         if counter ← 1 then
 9:             emit Possible fire to IoT server
10:         end if
11:         if clock is NOT started then
12:             start clock                           ▷ seconds
13:         end if
14:         if counter ← threshold || clock ← threshold then
15:             emit Confirmed Alert to IoT server
16:             Intial Alert ← false
17:         end if
18:     else                                          ▷ No fire
19:         counter ← 0
20:         if Intial Alert ← true then
21:             Stop clock
22:             Send Discarded to the IoT server
23:             Intial Alert ← false
24:         end if
25:     end if
26: end while
```

Since IoT ecosystems are complex, this design cannot safely claim to achieve ultimate reliability measures. While the presented design does not cover such reliability measures, we

assume that it is sufficient enough for building an edge component for the purposes of this paper. That is, mitigating false positive or negatives should also handle other sources of failures such as end-user misbehaviour, electric faults, unintended disconnection between the sensing module and the computing unit, possible Internet connection issues, etc. Other reliability measures can include redundancy of sensors and computing units.

B.1.4. Data Model at IoT Server

For the purposes of this paper, we designed a data model at the server-side level, as illustrated in Figure B.4, for representing field assets (Edge computing units and sensors). We use this model for persisting associated data, and for conducting typical CRUD operations (short for: Create, Read, Update, and Delete).

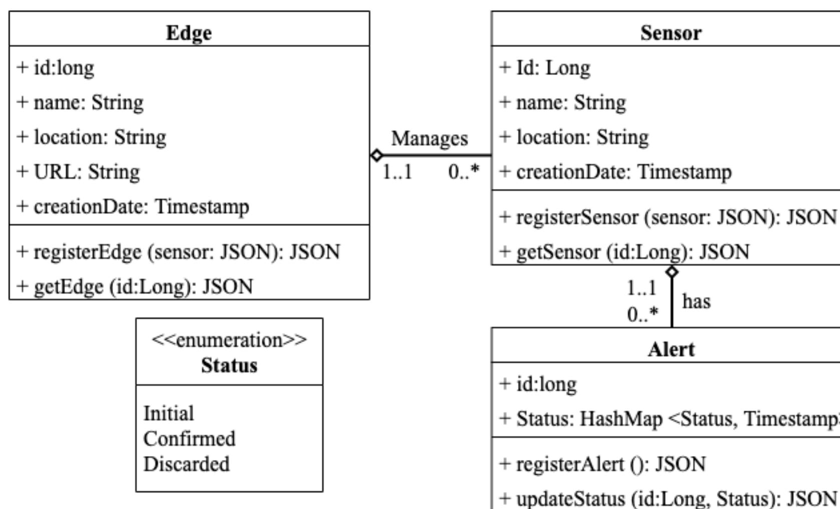


Figure B.4 Data Modelling at the IoT server side

For readability, we can perceive the data model as follows:

1. for each edge unit, it may have a set of sensors.
2. for each sensor, it may have a record of associated alerts.
3. for each alert, we keep track of its lifecycle (Initialised, Confirmed, or Discarded) as well as when a state change occurs.

B.1.5. REST HTTP API

We exposed a set of relevant CRUD functionalities as REST HTTP APIs to be consumed by the edge layer, as follows:

- registration of edge computing units and their associated assets (e.g. flame sensor).
- report a new alert (initial state).
- update the alert state to be either Confirmed or Cancelled.

B.1.6. Websocket Protocol

Since alerting requires immediate notification from the IoT server to Fire station, conventional HTTP request/response protocol is not perfect for that regard. That is, HTTP protocol would require the fire station to repeatedly request the IoT server querying for any fire events. A suitable alternative would be the concept of Websocket which maintains an open connection over TCP, enabling full-duplex communication established between the fire station and the IoT server [39]. Accordingly, this protocol enables the IoT server to actively send notifications about fire alert, and not merely responding to requests.

Appendix C. Blockchain-based Middleware for Simulated Environment

C.1. Asset and Wallet Management

Hyperledger Fabric organises assets in the form of $(key, value, version)$ data structure. For instance, quality requirements can considered as assets where *key* references a quality requirement consisting of a *value* of $Transmission_{time} \leq d$. Moreover, a performance record $pr_i \in PR$ is another form of assets under consideration of this thesis, which hold the count of newly identified compliant and breach cases. That is said, the middleware guarantees uniqueness of assets by keeping track of asset keys that are successfully processed and persisted on the blockchain side. Based on that, it generates a new unique key for every new asset such as quality requirements or performance records.

Due to the blockchain immutability, it is impossible to reuse already created keys for new assets generated by a new experiment round. An impractical workaround is by bringing down the current blockchain network and rebuilding it again from scratch. This practice can clear the storage, it is needlessly time and effort consuming. It also sacrifices all existing data.

As a solution, the middleware enables users to set a start point of asset keys to avoid conflict with existing keys at the blockchain side. For example, if the last persisted key of a quality requirement is 999, then the middleware can be instructed to start from 1000 in the new experiment. Consequently, this does not only eliminates the need for rebuilding the blockchain network for every new experiment but also preserve all existing assets. Additionally, the SLA data manager, as per Figure 3.9, provides a mechanism for resetting the storage state by deleting a set of assets with keys which are within the range from k_i to k_j.

C.2. Screenshots


```
1 {
2   "_id": "Q321",
3   "_rev": "1-f72d96ad491163aa7ba0fdd8135238e8",
4   "name": "TransmissionTime",
5   "requieredLevel": "LessThan",
6   "value": "3.0",
7   "~version": "CgMBDgA="
8 }
```


Figure C.1 Quality requirement at state storage

id	key	value
M121	M121	{ "rev": "1-7b49fd8643e3918de7138e17a764d275" }
M122	M122	{ "rev": "1-0af7d11d27554ed9d8cbcd4c53da7e91" }
M123	M123	{ "rev": "1-ac5e876886ca3ac68a28d45c77346574" }
M221	M221	{ "rev": "1-2ff3c4977f37c1d92b1c1f2a60d556de" }
M222	M222	{ "rev": "1-911464107f3a1e2b0c828baa8065692f" }
M223	M223	{ "rev": "1-03eaab9ebe215151677724bf5d6a8277" }
M321	M321	{ "rev": "1-410fc45ddbb95edd128b4273f5110683" }
M322	M322	{ "rev": "1-33521af8b7717cee021cfb515dc2c99d" }
M323	M323	{ "rev": "1-ec38b8cad00eb239786e7a319694ba02" }
M324	M324	{ "rev": "1-bdefb8ad9f92a9549ef19eeabc6ab104" }
Q121	Q121	{ "rev": "1-225fa1b8e46a5643718f1c1e514701e0" }
Q221	Q221	{ "rev": "1-5494af96cb2a391367cf36def629d23f" }
Q321	Q321	{ "rev": "1-f72d96ad491163aa7ba0fdd8135238e8" }

Figure C.2 Reported Metrics at the state storage (CouchDB)


```
1 {
2   "_id": "M322",
3   "_rev": "1-33521af8b7717cee021cfb515dc2c99d",
4   "breachCount": "9784",
5   "compliantCount": "9",
6   "qosID": "Q321",
7   "~version": "CgMBEAA="
8 }
```


Figure C.3 Reported breach and compliant metrics for each performance report pr_i at the state storage