DATA TRUST CHANGE TRACKING AND NOTIFICATION SYSTEM

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
In Partial Fulfillment of the Requirements
For the Degree of Master of Science
In the Department of Computer Science
University of Saskatchewan
Saskatoon

By

SAMBAVAN THARMARAJAH

PERMISSION TO USE

In presenting this thesis/dissertation in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying this thesis/dissertation in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors. They supervised my thesis/dissertation work or, in their absence, the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying, publication, or use of this thesis/dissertation or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis/dissertation.

DISCLAIMER

Reference in this thesis/dissertation to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement, recommendation, or favouring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis/dissertation in whole or part should be addressed to:

Head of the Department of Computer Science
Department of Computer Science
University of Saskatchewan
176 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

ABSTRACT

The rapid growth and advancement of digital information exchange in this Internet age have transformed data into the new-age currency of the economy, with companies and organizations clamouring for access to this precious resource. This surge in data acquisition and control has brought to the forefront the critical issue of TRUST in data handling and dissemination. Data governance's lack of transparency and accountability has eroded user confidence, leading to data privacy and misuse concerns.

To address these challenges, this thesis explores the concept of a federated data trust repository, which aims to provide a transparent, collaborative, and trustworthy framework for data sharing. Drawing on trust, governance, and accountability principles, the federated data trust model allows data producers to retain ownership and control over their data while enabling secure sharing with trusted entities (i.e., members of the data trust). It begins by examining the current data-sharing landscape and identifying the drawbacks of centralized and distributed architectures in ensuring transparency, provenance and auditability. It then articulates the research goals of establishing a data trust framework to restore trust between users and owners. The proposed federated data trust architecture encompasses data management, security, governance, and audit principles, focusing on enabling effective data sharing while maintaining control and accountability. The critical components of this architecture are data governance mechanisms, audit and provenance tracking capabilities leveraging blockchain technology, a change tracking and notification system, and a proxy interface for secure resource access. The architecture is designed to be flexible, adaptable to different use cases, and scalable to accommodate diverse data requirements.

Benefits of the federated data trust model include enhanced data privacy, increased data quality, greater flexibility, better governance, cost savings and increased innovation. In conclusion, the federated data trust model offers a promising solution to address the trust deficit in data sharing, promoting transparency, collaboration, and accountability in this digital economy. By fostering trust between the producers and consumers, the federated data trust framework paves the way for a healthier and more sustainable future for data-driven services and innovations for the greater good.

ACKNOWLEDGEMENTS

First and foremost, I wish to express my profound gratitude to my supervisor, Professor Ralph Deters. His unwavering dedication to innovation, profound technological insights, consistent feedback, and patient guidance have been pivotal in guiding me through the completion of my thesis and nurturing a deep understanding of the subject matter.

I am equally indebted to Professor Julita Vassileva for her invaluable contributions to my academic journey. Her wealth of knowledge, adeptness in fostering confidence during presentations, and unwavering support have significantly shaped my growth and development as a scholar.

A special acknowledgment is reserved for Professor Chris Zhang for generously lending his expertise as a member of my examination committee and providing invaluable feedback that has undoubtedly enriched the quality of my work.

To my parents, whose steadfast belief in the pursuit of knowledge transcending age barriers has served as a perpetual wellspring of inspiration, I owe a debt of gratitude. Likewise, to my beloved wife and children, whose unwavering support, encouragement, and gentle challenges have propelled me forward in my academic pursuits, I extend heartfelt appreciation. This achievement stands as a testament to our collective dedication and perseverance.

I also deeply appreciate the encouragement and motivation bestowed upon me by my ICT work colleagues and friends, whose unwavering belief in my capabilities has been a constant source of strength throughout this academic endeavor.

To my esteemed colleagues at the Multi-User Adaptive Distributed Mobile and Ubiquitous Computing (MADMUC) Lab, I extend sincere thanks for your steadfast support and invaluable feedback, which have undoubtedly contributed to the refinement of my research.

Lastly, I extend my appreciation to all individuals within the Department of Computer Science for their unwavering assistance and support, which have been indispensable throughout my academic journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ABAC	ATTRIBUTE-BASED ACCESS CONTROL
ACL	ACCESS CONTROL LIST
AI	ARTIFICIAL INTELLIGENCE
AMQP	ADVANCED MESSAGE QUEUING PROTOCOL
API	APPLICATION PROGRAM INTERFACE
AWS	AMAZON WEB SERVICES
DML	DATA MANIPULATION LANGUAGE
DPO	DELEGATED PROOF OF STAKE
ERD	ENTITY RELATIONSHIP DIAGRAM
IOT	INTERNET OF THINGS
JMS	JAVA MESSAGING SERVICE
M2M	MACHINE TO MACHINE
MQTT	MESSAGE QUEUE TELEMETRY TRANSPORT
ML	MACHINE LEARNING
MSP	MEMBERSHIP SERVICE PROVIDER
OLTP	ONLINE TRASNACTION PROCESSING
PBFT	PRACTICAL BYZANTINE FAULT TOLERANCE
POA	PROOF OF AUTHORITY
POC	PROOF OF CAPACITY
POS	PROOF OF STAKE
POW	PROOF OF WORK
QOS	QUALITY OF SERVICE
RBAC	ROLE-BASED ACCESS CONTORL
REST	REPRESENTATIONAL STATE TRANSFER STYLE
TPS	TRANSACTIONS PER SECOND
URI	UNIVERSAL RESOURCE INTERFACE

CHAPTER 1

INTRODUCTION

The Industrial Revolution of 1800 showed the complexity of delivering essential services. The fast sprawl of urbanization and social structure added complexity to providing vital services through centralized networks of facilities. Changes brought earlier taught us the critical reason these still exist: the inherent trust vested in these systems by the users. Our social behaviour and structure have changed rapidly in the internet age. The information that took days to reach now takes less than seconds. The distance measure has been modified to the extent of time on these transactions. This thirst for real-time knowledge of society and the environment propelled rapid growth in information flow.

With this rapid boom, the data has become the "new oil" (Budzyn 2019, Economist 2017, Kuneva 2009) which is the essential commodity of the new economy. This data is our new resource. There is a voracious appetite for data from companies and organizations. With the growth of data explosions and acquisitions comes the fundamental nature of "Trust." According to the Merriam-Webster dictionary, "Trust" means the "belief that someone or something is reliable, good, honest, and effective." The lack of trust is evident from the survey conducted by KPMG. Its survey identified that though 95 percent of the corporate leaders said they have better data policies, 66 percent accepted that they could do better (Lucas 2021). The most exciting facet of this survey is that eighty-six percent of the participants have concerns about data privacy, and sixty-six percent said they have concerns about the data collected (Lucas 2021). Big companies like Alphabet (parent of Google), Amazon, Meta (parent of Facebook), and Twitter hold the keys to the data kingdom. The users of these platforms pay for the services with the data they generate. These are then sold to third-party entities to create advertisements to target those who use the services. Though selling the data to third parties is necessary to balance the cost of providing those services, the need for more transparency to the data producers is a growing concern. These companies define and determine how this data is disseminated. As solution providers, these for-profit organizations control the data and reap benefits from the producer and the consumer. There is an apparent erosion of user confidence in the companies that provide services in these areas. Four out of five present-day users believe their generated information is being used and sold to third parties without their consent. Hence, there is an urgent need to address this trust more transparently so the producer and consumer confidence can be regained for a healthy future of new services. This lack of trust and transparency of operation can be overcome using data trust. For discussion, we consider the research data sharing among the academic research groups more transparently and collaboratively.

CHAPTER 2

PROBLEM DEFINITION

With the arrival of Big Data and the use of Artificial Intelligence (AI) and Machine Learning (ML) to analyze large volumes of data, there is a data proliferation. However, the organizations and enterprises that have these assets are not transparent about what they do or how they profit. Because of the lack of trust, they are also protective of what, how, and with whom they share the data (Matttioli 2017).

2.1 CURRENT ARCHITECTURE DRAWBACKS

In present-day centralized or distributed architecture systems, when data is shared with individuals or groups, the path that data exchange takes to reach the consumer is never straightforward Figure 2.1. When the data producer produces his data and puts it for sharing, the consumer has no clear lineage. This creates a lack of transparency and lack of provenance. The data could be changing hands from one consumer to the other, with or without alterations; this causes a lack of auditability. The more extensive and diverse the organization or the business, the more complex these interactions become. The scalability and management become more challenging. There are no defined rules to govern this activity.

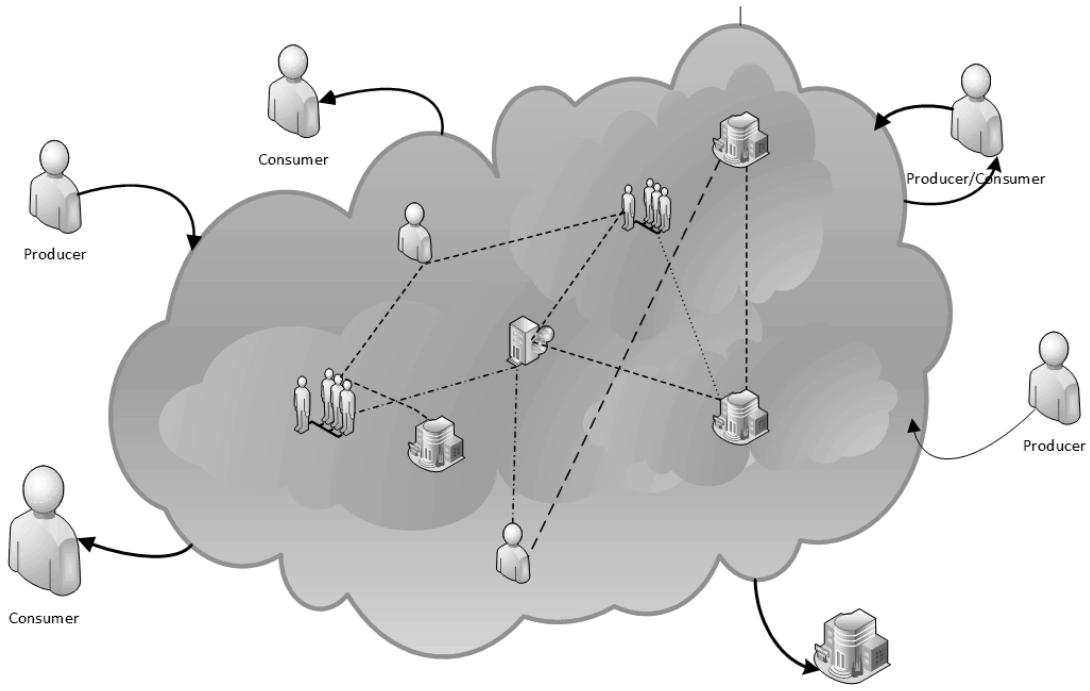


Figure 2.1: Current Data Exchange Processes

2.2 RESEARCH GOALS

This lack of trust between users and businesses, and businesses and businesses, brings us to the research goals. Companies and organizations can address this erosion of trust by introducing a *data trust* framework. With data trust, businesses could efficiently manage the data using transparency, governance, and viable auditing. Though there are several different forms of data trust, like public data trust, private data trust, and all possible flavours that can be architected in between, we narrow our focus to the federated data trust model. In a federated data trust model, the policies can be defined based on a greater good with mutual benefits Figure 2.2. Though we envision this as a more generalized use case, for extended discussion, we consider academic research data sharing between the academic research groups internally and externally between the participating members.

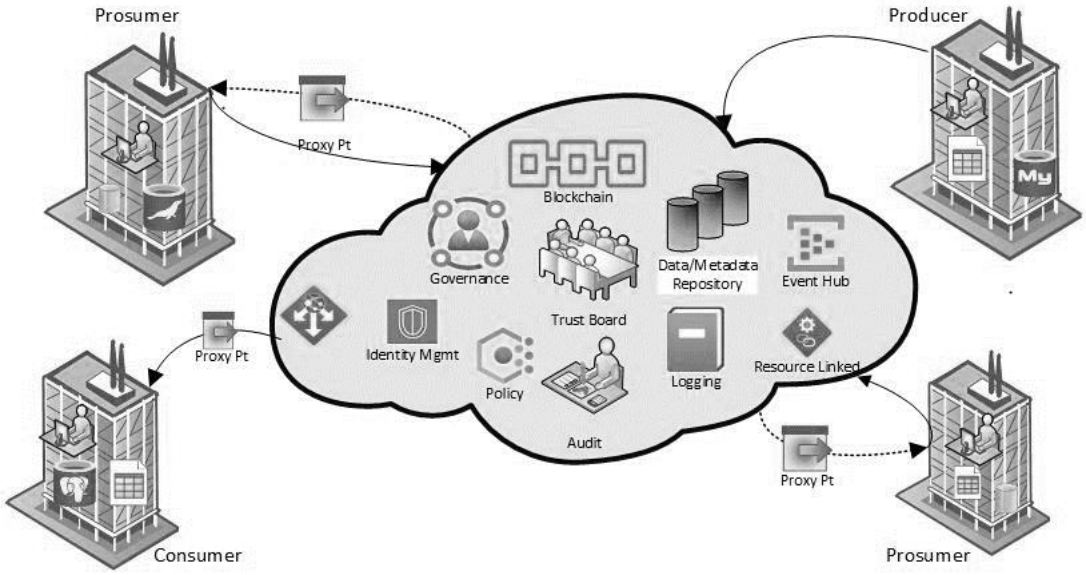


Figure 2.2: Federated Data Trust Birds Eye View

In our proposed model, the data from sources like data lakes[1], data pools[2], data puddles[3], databases, and filesystems will remain in the custody of the data producers. When they agree to share the data, they become members of our proposed data trust model, deciding upon the consortium members' terms and conditions of data-sharing policies. The data trust is a central intermediatory entity that enforces governance, provenance, and audits. In our model, the trust does not possess the raw data unless and unless otherwise

[1] Data lake - Wikipedia
[2] What is a Data Pool? | Lentiq Blog
[3] Introduction to Data Lakes - The Enterprise Big Data Lake [Book] (oreilly.com)

agreed upon by the consortium. Instead, the trust shall record and share the access point URI in an acceptable Web 3.0 URI form while the data repository keeps track of this information.

The owner also provides the metadata information on the data source. The metadata repository keeps track of this information on the locally distributed database. An encrypted hash of the data and metadata repository records are stored in the blockchain. Finally, the requestor (i.e., consumer or prosumer) is granted access, meeting all the criteria set by the data access policy. The blockchain monitors the data access. The owner can set the data access policy in compliance with the access policy standards set by the trust; when a consumer requests access to the data source, the consumer's credentials are validated to allow the authentication. Then, the consumer's subject entity, the data's object entity, the environmental restrictions, and consumer credentials are used with the policy assessment to grant authorization to access the data source URI from the data repository. Upon approval, the consumer is issued a client-side proxy API to the data source. This client-side proxy interacts with the server-side proxy to fulfill client requests—we carry out all these interactions between these components using the web API. At this point, we narrow our attention to the goal of tracking the data changes at different points of the processes. For this, we will incorporate an event system to track any changes to the data source Figure 2.3. When the producer registers his data source with the trust, they will be the prosumer on the queue. When a consumer approved of access to the data wants to be notified of any changes to the data source, they can subscribe to the event corresponding event system using the data source ID. At this point, they will also start receiving the change notifications.

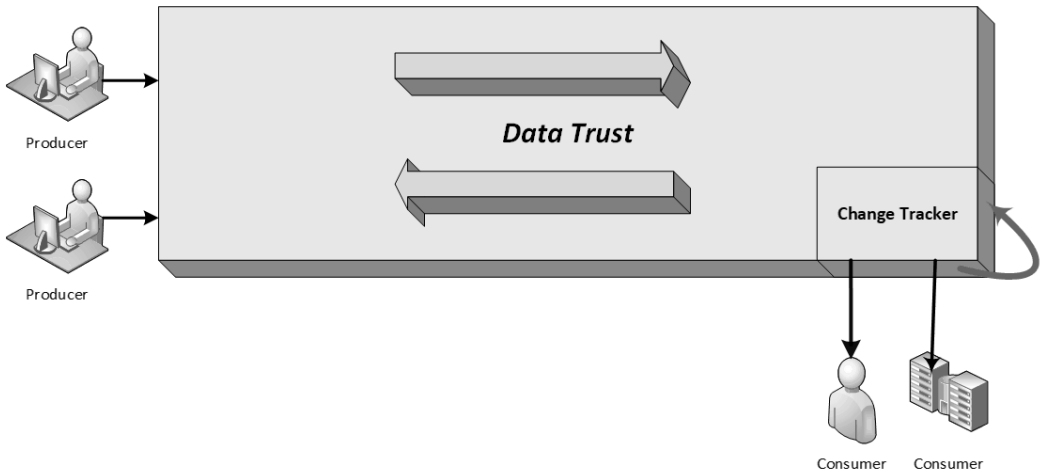


Figure 2.3: Thesis Premise

By finding answers to some of the below-mentioned questions, we look forward to adding some richness to the already maturing data trust framework.

1 How can a create, update, and delete action on the data be tracked and informed of its change state to the reliant stakeholders?

2 How can this change tracking complement governance, provenance, and auditing?

3 Test the throughput and latency that could create a bottleneck in introducing our intermediatory process.

CHAPTER 3

LITERATURE REVIEW

This chapter discusses the background information in the following areas of our interest: data trust, blockchain, message queuing technologies, etc.

3.1 DATA TRUST

The word "data trust" has become the mainstream policy-making acronym in Canada, the United Kingdom, and the European Union in the past few years. These countries experiment with a data governance framework that would facilitate the growth of their data-sharing needs in Artificial Intelligence.

3.1.1 What is Trust?

The Roman Law in 400 BC was the first to develop this concept of trust as a "testamentary trust," created as a will, which came into effect after the person's death. Later, in the 16th century, the British common law adopted it and introduced the "inter vivos" trust, a "living trust" that was funded and managed during the Trustor's lifetime (Trust Law n.d.). This common law became the cornerstone for future trust laws.

According to English common law, trust is a legal relationship in which the holder of the right "settlor," who entrusts it to another person or an entity, the "trustee" to hold and manage solely for the benefit of another, the "beneficiary" (Trust Law n.d.). Keeping this as a baseline proposition, we discuss the concept of data trust. The data is the asset, the data producers are the settlors, and the data consumers are the beneficiaries.

3.1.2 What is a Data Trust?

The term "data trust" was initially envisioned by Sir Tim Berners-Lee, the inventor of the World Wide Web. Sir Tim Berners-Lee observed that huge companies like Facebook, Google, Amazon, Twitter, and Netflix capture enormous amounts of data but do not share them for the greater good (Finley 2017). Through Solid open-source technology (Berners-Lee n.d., Sambra, et al. 2016), Sir Tim Berners-Lee envisions demonopolizing the Internet by facilitating users with options to store their data on private servers called "Pods" [4]and allowing social media services to access that user information with the consent of users (Inrupt

[4] About Solid · Solid (solidproject.org)

n.d.). He hoped to facilitate the users with personal empowerment on data. The solid-Intrup[5] project provided the underpinning technological vision for this data trust. With this underwork, Sir Tim Berners-Lee and the artificial intelligence expert Sir Nigel Shadbolt co-founded the not-for-profit organization Open Data Institute (ODI) in 2012 (Open Data Institute 2012), with the goal "*to work with companies and governments to build an open, trustworthy data ecosystem*" (Open Data Institute 2012). During this course of time, other similar organizations sprung, namely Nesta (Nesta n.d.), Element A.I. (Element AI n.d.) and GovLab (NYU GovLab n.d.).

To augment Sir Tim Berners-Lee's idea, Jack M. Balkin and Jonathan Zittrain proposed the concept of "*information fiduciaries*" (Information Fiduciaries and the First Amendment 2016, Balkin and Zittrain 2016). Balkin and Zittrain drew inspiration from the U.S. Digital Millennium Copyright Act of 1998 (Balkin and Zittrain 2016). Here, they draw parallels to how the act facilitated a safe harbour for businesses to publish the content and take down the allegedly infringing content; likewise, they put forth the idea that the technology "companies could take on the responsibilities of information fiduciaries: They would agree to a set of fair information practices, including security and privacy guarantees, and disclosure of breaches" (Balkin and Zittrain 2016). Data trust of this approach would protect the individuals' ownership of data, and the binding rules limit companies on providing access to the users' data only upon users' consent. Unfortunately, this approach puts much trust in tech companies, who stand to reap profit from their actions, which may not be a good solution for the public good and individuals. Further to this discussion, Dame Wendy Hall and Jerome Pesenti emphasize that data trust is essential to the growth of AI in the U.K. The report recommendation: "To facilitate the sharing of data between organizations holding data and organizations looking to use data to develop A.I. Government and industry should deliver a program to develop Data Trust – proven and trusted frameworks and agreements – to ensure exchanges are secure and mutually beneficial" (Hall and Pesenti 2017). Their vision of data trust is not a legal entity, instead "a set of relationships underpinned by a repeatable framework, compliant with parties' obligations, to share data in a fair, safe and equitable way" (Hall and Pesenti 2017). With no legal ties, the data trust using this principle can fall short of their fiduciary duties with no legal repercussions to their actions.

Sylvie Delacroix and Neil Lawrence envision a data trust framework where "the trust will encapsulate a particular set of aspirations, reflected in the trust. Bounded by a fiduciary obligation of undivided loyalty, data trustees will exercise the data rights held under the trust according to its particular terms." (Delacroix and Lawrence 2018). Unlike the other models proposed earlier, Sylvie and Neil point out a need for a legal

[5] Home · Solid (solidproject.org)

entity (i.e., trust), which can be either for-profit or non-profit enterprises of various varieties, where individuals have the right and technical information to shop for a trust that meets their needs. Sylvie and Neil pointed out that this kind of small, focused trust can provide better safety, security, governance, and data portability with this Trust model. Though several public and private sector organizations and businesses have conceptualized it as the framework to follow, this model has some shortcomings. First, there needs to be an ecosystem with the required legal frameworks, legislation, and laws for this framework to succeed. Which countries are grappling to figure it out? Next is the suggestion of data erasure and portability. Data portability and erasure are the most complex areas to implement; as the model grows with different subsidiaries, the provenance tracking gets harder and harder.

Sean McDonald and Keith Porcaro, who draw their inspiration from the ownership and licensing of intellectual property, propose a "civic data trust" model, where they envision "a trustee organization to own the code and the data generated by a technology and licenses it to a for-profit company that commercializes it" (McDonald and Porcaro 2015). Sean and Keith entrust the fiduciary responsibilities to be carried out by both the trustee and the commercial enterprise. Doing so establishes the needed checks and balances between the two entities.

The Open Data Institute, actively working on this concept, defines "*data trust as a legal structure that provides independent stewardship of data*" (Hardinges, Defining a 'data trust' 2018, Hardinges, What is a data trust 2018). The intertwined criticality of trust and trustworthiness of the data institutions and data trust is well illustrated in the "Desing trustworthy data institutions" (Hardings, et al. 2020). Sightline Innovation, a for-profit company, defines data trust as "*a governance framework with an architecture that supports the legal and technological infrastructure to ensure trust over the data and derivative data assets throughout their lifecycle. It provides a legally binding framework for stewardship over the data assets for the benefit of the people or the group of organizations within the Trust*" (What is a data trust? n.d.).

As we can see from a handful of works we have analyzed above, there is a broader endorsement of the concept of data trust, but there is a lack of a clear definition. This lack of clarity is one thing that constitutes the prevalent implementation of data trust.

3.1.3 Core components

In this section, we explore different core functionalities that constitute the architecture of a data trust framework.

This thesis draws references to these core components from Kieron O'Hara's "Data Trusts: Ethics, Architecture, and Governance for Trustworthy Data Stewardship" (K. O'Hara 2019) and a Canadian study (Paprica, et al. 2020).

- Legal: A legal framework should focus on authority, collection, holding, and data sharing. As O'Hara points out, this legal framework should not constrain data trust. This also should not be one size fits like Sylvie and Neil put out (Delacroix and Lawrence 2018).

- Governance: A data trust should have a stated purpose and transparency. It should have a governing body that is accountable and adaptable to its governance.

- Management: Data trust management should be agile and adaptive. O'Hara suggests that a well-defined management structure should encompass the following (K. O'Hara 2019).
    - *Discovery*: The potential users should be able to discover the data's existence, properties, and quality.

    - *Provenance*: The data quality is assessed through the metadata gathered on the data and its properties. Potential users need access to this data to evaluate its quality. The systems within which this data is accessed should also generate new provenance on the data based upon the operation.

    - *Access controls*: Data controllers take the sole custody of access control. The producer needs to engage with the controller to define the terms and conditions for sharing. This puts the data controller liable for any data breaches.

    - *Access*: Users must be provided a means to access the data. This access level needs to be determined by the access policies. For example, an access policy could choose the access to be unconditional or limited in quantity or time or to a redacted, anonymized data version.

    - Identity management: The trust model shall facilitate the data controllers' identification of those who accessed or attempted to access the data over time.

- *Auditing*: A record of data use needs to be tracked and recorded transparently to be checked for compliance with the law and trust agreements.

- *Accountability*: Sound auditing shall enable the data controllers to be accountable for their actions. This also must put those receiving the data and anyone else using the data responsibly. Thus, a suitable accountability procedure is drawn to articulate the punishments.

- *Impact*: There ought to be a constant assessment of data's value, use, and misuse through the records kept in the data trust.

- Data user agreement: The agreements should clearly state the provisioning and de-provisioning of data users' rights and access (Paprica, et al. 2020). The contract should also highlight the different data classifications, their level of sensitivity, and the prohibition on re-identification or de-identification through some linking. The agreement should also include the consequences of violating compliance with the deal.

"Data Protection by Design: Building the foundations of trustworthy data sharing" (Stalla-Bourdillon, et al. 2020) proposes a three-layer methodology: (a) data layer – where interested parties make plans to create a data pool; (b) access layer: where pooled data are made available for discovery through data trust; and (c) process layer: where pooled data are approved for (re)use by the data trust. All these studies showcase and emphasize the need for sound governance, provenance, and a well-defined process to be present for the success of the data trust. Stuart Mills further classifies the data trust architecture into three categories: Collector-centric, Data-centric, and Generator-centric models (Mills 2019). In Generator centric data trust, "Instead of pooling data, data generators collectivize as individuals, and the data trust governs the data collector's access to the data generators themselves. Thus, the generator-centric data trust does not steward data; its stewardship obligations protect individual members." (Mills 2019). This thesis focuses on federated data sharing, which resembles the Generator-centric model illustrated in Figure 3.1.

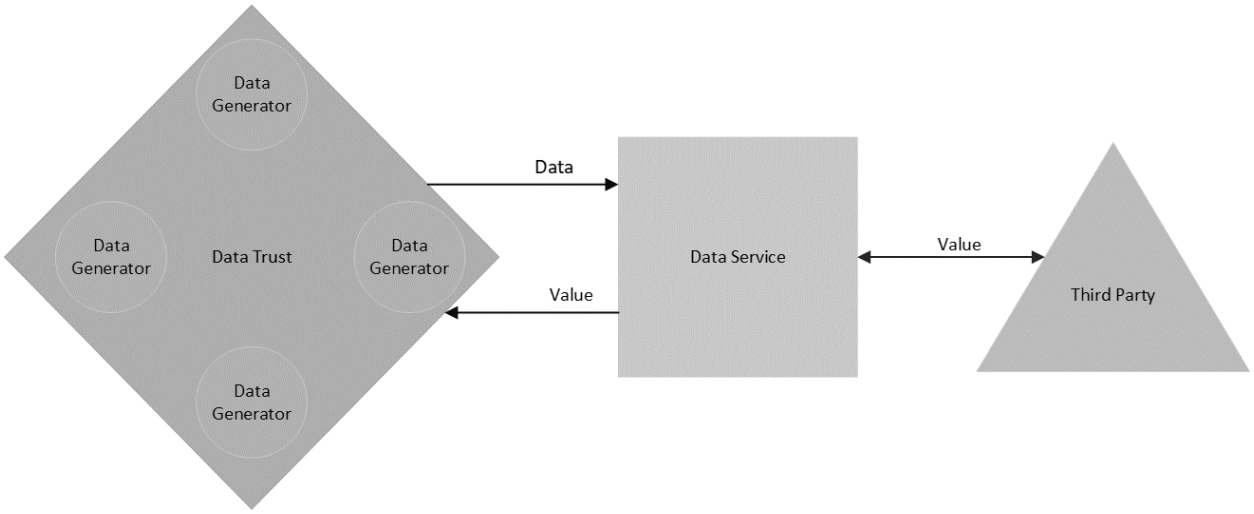


Figure 3.1: Generator-Centric Data Trust by Mills

3.1.4 Different forms of Data Trusts

Several forms of data trusts can be architected and used for different purposes. Data trust architecture can be classified into three categories: In the following sections, we touch on each design structure and see how they benefit the community they are intended for. Lau Jia Jun et al., in their working paper "Basics of private and public data trust," explores the different legal definitions and the challenges (Lau Jia Jun, Penner and Wong September 23, 2019).

3.1.4.1 PUBLIC DATA TRUSTS

Public data trusts are defined as sharing information for the greater good. Where trust members agree to share their data to achieve the greater good for society. When they decide to share their information, they knowingly share information that can be utilized for any purpose or derived purpose by the consumer. The role of data trust is to protect that information and make sure that it is used responsibly and in compliance with data use policies. The notable ones are OpenDataTrust, DataKind,[6] and Open Knowledge International (OKI)[7]

[6] DataKind - Welcome
[7] Our local group – OPEN KNOWLEDGE CANADA (okfn.org)

3.1.4.2 PRIVATE DATA TRUST

Private data trusts are owned and operated by public/private institutions to guard the privacy of the data that the individual or an organization shares. The role of the trust is to provide the defined services to its approved members in a legal, ethical and authorized manner. These trusts are typically maintained by for-profit organizations or governments that seek to protect the data. One example is data trust in the healthcare systems. Examples, Health Data Exchange[8], Health Data Hub of Frace[9], Financial Data Exchange[10].

3.1.4.3 FEDERATED DATA TRUST

Federated data trusts are data-sharing agreements between organization members, allowing members to share their data while retaining control and ownership of their data. This design facilitates the member organizations' collaboration while protecting the security and privacy of their data. One can observe these trusts to exist in governmental organizations, where the trust is established with some technical and legal compliances on the data access and use policies, preserving the privacy of the data in their custody. Susha et all, explains this under the framework "Data Collaboratives", to illustrate how the public and private sectors can use the model to share data for a greater public good (Susha, Janssen and Verhulst 2017). For example, Open Data Exchange[11], Data Commons[12], Data For Democracy[13], Chicago Data Collaborative,[14] and Data Driven Detroit[15].

A data trust reduces friction by (i) using a legal agreement to reach a shared understanding with data producers (i.e. contributors) as to their contents and allowed uses, (ii) connecting rich datasets to the expertise required to create differential privacy mechanisms appropriate for each use case, (iii) ensuring that data is used responsibly by consumers, and (iv) communicating (Young, et al. January 2019). Our thesis's focus is defined on the premise of the federated data trust models. In the next section of this work, we will analyze the blockchain's role in the data trust architecture.

[8] Secure Patient Portal (healthdataexchange.com)
[9] Page d'accueil | Health Data Hub (health-data-hub.fr)
[10] Home (financialdataexchange.org)
[11] https://www.opengovpartnership.org/members/canada/commitments/CA0032/
[12] https://www.datacommons.org/
[13] http://d4dcommunity.org/
[14] Chicago Data Collaborative
[15] https://datadrivendetroit.org/

3.2 BLOCKCHAIN

Blockchain is a distributed ledger technology (DLT) that uses the public ledger to record transactions without a need or a dependency on a third-party validation system to validate each transaction. It offers security, immutability, transparency, and traceability of transactions in a distributed network. It's prominence came to the forefront with the introduction of BitCoin[16]. Even though this technology was new, its inception can be traced back to different pieces like Ralph Merkle's Merkle tree, which provide the data structure for verifying individual records (Merkle 1987), David Chaum's vault system, which provides the foundation for the tamper-resistant data storage and access (Chaum 1979). The other technologies that contributed to the development include the works of Stuart Harber and W. Scott Stornettas, "How to timestamp a digital document," which provided a practical solution to tamper-proof a digital document and store it in a block. (Haber and Stornetta 1991). The concept of Peer-to-peer technology drew fame from Napster. In their work to combat junk mail, the early works of Cynthia Dwork and Moni Naor laid the foundation of Proof of Work (PoW) (Dwork, Naor and Wee 2005). This PoW was later assented to by the works of Adam Back's "Hashcash[17]" (Back 2002) and Hal Finney's "Reusable Proofs of Wok (RPoW)" (Finney 2004). In 2008, Satoshi Nakamoto introduced the Bitcoin and electronic cash transaction system, encompassing the technologies mentioned earlier (Nakamoto 2008).

The blockchain records the data in a collection of transactions called blocks. The first block in the transaction is referred to as the Genesis block. The blocks are assigned an alpha-numeric string called a hash. This hash is generated based on the timestamp. The blocks are added sequentially. The list of ordered blocks forms a chain. As new data is added, a new block is added instead of changing the block's content, like in the standard database systems. When a new block is added, it uses the previous block's hash to generate its hash and records both the hashes in the header of the current block. Hence, this creates a link between the blocks to form the chain. They also use a computational process called the consensus process to validate the block's authenticity before it gets added to the chain. This ensures that the copies of its distributed Legere are in the same state at any given time. Thus preventing the possibility of data manipulation. If any manipulation is to happen, the difference in hash values recorded between parent and child blocks will cause discrepancies and fail.

Several blockchain technologies have sprung up since then. They fall under two major categories—public or permission. Public or permissionless blockchains are open to the world, allowing anyone to join the

[16] https://bitcoin.org/en/
[17] http://www.hashcash.org/

network, create transactions, and participate in the consensus. This free participatory process thus introduces a heavy computation-oriented consensus mechanism of proof of work (PoW) to sustain the extensive growth of blocks in the distributed ledger system. They also have an economic incentive. Unfortunately, this unrestricted use and high computation increase the latency of operations. Permission blockchains are created with architecture similar to that of the public; they introduce a layer of membership concept among the known identified participants, allowing only those authenticated access to the system. Because of this, a layer of authentication security allows more granular access control for reading, writing, and participation. Moreover, due to this level of membership authentication, they enjoy having a lighter consensus mechanism, allowing them to perform better than their public counterparts.

Some of the more prevalent blockchains are listed in Table 3.1.

PLATFORM	**CONSENSUS**	**PUBLIC OR PERMISSIONED**	**SUPPORTS SMART CONTRACTS**	**SUPPORTED SMART CONTRACT LANGUAGES**	**CRYPTO CURRENCY**
BITCOIN[18]	POW	PUBLIC	YES	CLARITY[19]	
ETHEREUM[20]	POW & POS	BOTH	YES	SOLIDITY[21], VYPER[22], YUL[23], FE[24], LLL[25]	EITHER
HYPERLEGER FABRIC[26]	PBFT	PERMISSIONED	YES	GO, NODE.JS, JAVA	NO
CORDA[27]	PLUGGABLE	PERMISSIONED	YES	KOTLIN, JAVA	NO
NEO[28]	DBFT	BOTH	YES	C#, F#,[29]JAVA, VB.NET, PYTHON	NEO

Table 3.1: Blockchain Platforms

The prominent blockchains currently in prevalent utilization are Ethereum and Hyperledger Fabric. After some surface-level experimentations on Ethereum and Hyperledger fabric, I chose the Hyperledger fabric because of its maturity in the permissioned blockchain realms.

[18] https://bitcoin.org/en/
[19] https://clarity-lang.org/
[20] https://ethereum.org/en/
[21] https://soliditylang.org/
[22] https://vyper.readthedocs.io/
[23] https://docs.soliditylang.org/en/v0.8.17/yul.html#yul
[24] https://fe-lang.org/
[25] https://lll-docs.readthedocs.io/en/latest/lll_reference.html
[26] https://www.hyperledger.org/
[27] https://corda.net/
[28] https://neo.org/
[29] https://fsharp.org/

3.2.1 Hyperledger Fabric

Hyperledger Fabric (Androulaki, et al. 2018) is a modular open-source permissioned blockchain system hosted by Linux Foundation[30]. This is one of the extensible blockchain systems that facilitates modular consensus protocols, memberships, and databases, which can be plugged into the baseline framework. This modularity and extensibility allow the system to be customized for different requirements. Furthermore, this was the first blockchain framework to support standard programming languages. This alleviates the need for the "*smart contract*" to be written in system-specific programming languages.

3.2.1.1 HYPERLEDGER FABRIC COMPONENTS

One must know the following essential components to understand this framework's core working, as shown in Figure 3.2.

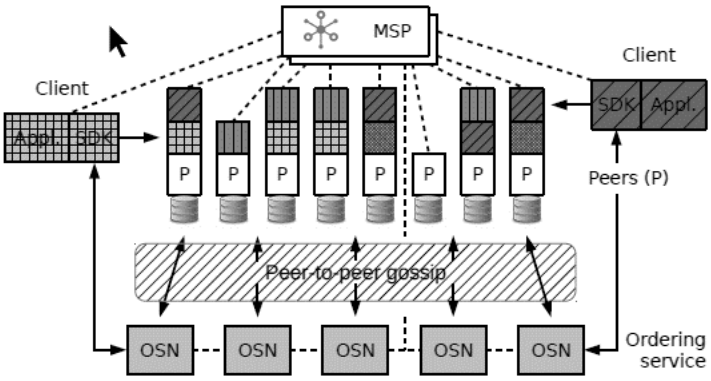


Figure 3.2: A Fabric network with federated MSPs (Androulaki, et al. 2018)

- Chaincode: In Hyperledger fabric, the smart contract is called a *chaincode*. The words smart contract and chain code are synonyms. A *chain code* is a program that implements the application logic and runs during the execution phase of the transaction. Chaincode is the central part of this distributed application. Another form of chain code exists for managing and maintaining the blockchain system. It is called a *system chaincode*.

[30] https://www.hyperledger.org/use/fabric

- Endorsement Policy: Endorsement policy determines which peers and how many of them are needed to endorse a transaction during the validation phase to call it valid. The validation phase of the transaction evaluates the *endorsement policy*. The application developers cannot alter endorsement policies. Only the designated administrators may have permission to modify the endorsement policy through system-managed functions.

- Peers: *Peers* are the nodes that execute and validate transaction proposals. Peers maintain a copy of the blockchain ledger in an append-only form. The peer maintains the transaction as a hash chain and state. The state form represents the latest state of the ledger. Not all peers are required to execute all the transactions. Based on the endorsement policy, only the subset of the endorsing peers will execute the transactions.

- Endorsement: The clients send transactions to the peers specified by the endorsement policy. Specified peers execute each transaction and record the output. This process is called the *endorsement*.

- Orderer: The orderer establishes a total order of all transactions in the fabric. Each transaction will contain the state updates and dependencies computed during the execution phase, with the cryptographic signatures of the endorsing peers. They do not participate in the execution or validation process of the transaction.

- Channels: Hyperledger fabric can support multiple blockchains. Each blockchain in this framework is called a channel. Each channel may have one or many peers as its members.

- Membership Service: Since Hyperledger fabric is a permissioned blockchain, every participant must have a valid identity using X.509 certificates. This also includes the permission attributes that describe the level of access to different information on the network. Fabric CA is the certificate authority that issues these certificates. However, this does not preclude one from using any third-party certificates. The fabrics' membership service provider (MSP) then validates these certificates and opens access to different services.

3.2.1.2 HYPERLEGER FABRIC TRANSACTION EXECUTION PROCESS

Before the advent of Hyperledger fabric, previous blockchain systems followed the order-execute architecture. Fabric blockchain is the first to introduce the paradigm of *execute-order-validate* on its transaction processing, as shown in Figure 3.3.

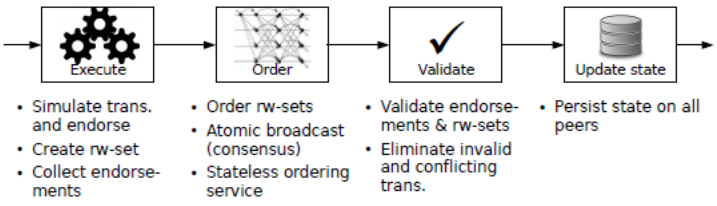


Figure 3.3: Execute-order-validate architecture of Fabric (Androulaki, et al. 2018)

The Hyperledger fabric transaction processing phase of execute-order-validate separates transaction flow into three steps: (1) *execute phase*, executes the transaction, checks its correctness, and endorses it. During this phase, the client sends the transaction proposal to one or more endorsers. The endorsers execute the operations on the specified chain code installed in the blockchain and will send the response to the client. Next, (2) *order phase* orders them through the consensus protocols, irrespective of transaction semantics. Upon receiving the proposal from the endorsing peers, the client will pass it on to the ordering. Upon receiving the transaction payload, the orderer establishes a total order on all submitted transactions per channel (i.e. blockchain). In this phase, the ordering service assembles multiple transactions into blocks. It outputs the hash-chained sequence of blocks containing the transactions. Finally, (3) the *validation phase* validates the transaction based on the application trust definitions, thus preventing any race conditions that can occur due to concurrency. Unlike all other blockchain systems, in the Fabric blockchain, the transactions are endorsed by a subset of endorsing peers. During this phase, the endorsement policy evaluates all the transactions within a block. Then, read-write conflict is evaluated for all the transactions in the block. Upon completion of the two stages of endorsing and conflict check, the ledger is updated by storing the block by appending it to the ledger, and the blockchain state is updated.

3.2.2 Consensus

A consensus mechanism is a means by which all peers agree on the transaction. Different blockchains use different consensus algorithms. These different algorithms evolved over time to more others. We will

review some of the core algorithms that form the basis of different blockchains using the survey by Nguyen and Kim (Nguyen and Kim 2018). Nugyen and Kim classify the algorithms into two categories.

3.2.2.1 PROOF BASED

In the proof-based algorithm, each node in the system must prove that it can be selected as a leader to make changes to the existing ledger, which every other node will accept. These are mainly used in open permissionless architectures. The proof-based systems can further be subclassified based on the methodology they used to define the proof.

3.2.2.1.1 Proof of Work (PoW)

Proof of work (PoW) is the consensus algorithm introduced by Cynthia Dwork and Moni Naor (Dwork, Naor and Wee 2005). This was later implemented by Adam Back's "Hashcash" (Back 2002). This was later brought to the fore by the proposal of Satoshi Nakamoto's Bitcoin (Nakamoto 2008), and later by Ethereum. It is an incentivized algorithm where all nodes try to solve a complex cryptographic puzzle by guesstimating a secret value, termed nonce. The first node successfully identifying this nonce value will broadcast its block to the network. This will prompt other miners to stop mining the nonce for that block. All the other miners will then check the validity and commit the transaction in their blockchain. Because of this, distributed mining and communication require significant computational resources.

3.2.2.1.2 Proof of Stake (PoS)

Proof of stake (PoS) was introduced to overcome the shortfalls of PoW. Though they have similar operational fundamentals, they differ regarding the mining consensus. In the PoS, the miner holds the highest on the network. The stake is defined as a digital monetary value instead of performing the hash functions. Despite computational savings, this gets the rich node richer every time. Therefore, Ethereum is moving from PoW to PoS.

3.2.2.1.3 Proof of Stake with Delegated Ownership (DPoS)

Delegated proof of ownership utilizes a voting system where the users stake tokens to vote for delegates. The elected delegate is given the responsibility for block generation and validation. The delegate receives a reward upon completing the task, which is then shared with the other delegates who voted for it. Here, the higher the voter stakes, the greater the token gains, and the higher the voting power. Because this limits the number of participatory nodes, this improves performance over the PoS model.

3.2.2.1.4 Proof of Capacity (PoC)

This solution to complex mathematical challenges is gathered in digital storage using the proof of capacity. The users can later use this to create blocks. The one who evaluates solutions quickly gets the chance to create the block (Blog 2021).

3.2.2.1.5 Proof of Authority (PoA)

This is based on the trust among the trusted parties of the blockchain. The Ethereum co-founder and former Chief Technology Officer Gavin Wood created this. Like PoW, miners compete to solve a cryptographic challenge as quickly as possible, using specific hardware and energy. However, the blocks they come across on the other end would only include the information about the block winner's identity and the reward transaction. At this point, it switches to the PoS model. Here, the validators do not stake on resources. Instead, they stake on reputation and identity because this hybrid method is faster and more efficient.

3.2.2.2 VOTING BASED

All the system nodes get to vote on every block validation in the voting-based algorithms. The blocks are validated based on the consensus policy and minimum positive vote requirements. These are mainly used in closed-circuit architectures like permissioned blockchains.

3.2.2.2.1 Byzantine fault tolerance based

As the name suggests, Byzantine fault tolerance is designed to cope with Byzantine faults, where system players must agree to a strategy to avoid catastrophic system collapse. (Blog 2021). This can further be subjected to Practical Byzantine Fault Tolerance (PBFT) and Delegated Byzantine Fault Tolerance (DBFT).

Hyperledger fabric uses the PBFT. PBFT has two kinds of nodes: the leader and validating peers. The leader node is not constant. Initially, the client sends their transaction request to their validating peers. The receiving peer will validate the transaction and broadcast it to all the peers, including the leader. After a number of transactions are received and reach the transaction block threshold, the leader will order the transactions based on the created time stamp and put them into a block. Upon this, he carries out the three phases of the executions, as shown in Figure 3.4. The first phase is pre-prepare, where the leader broadcasts his proposed block to other peers. On receiving the block, the peers will save it locally and then re-broadcast it during their preparation and commit phase to validate its authenticity. In the prepare phase, when the nodes receive the same block as the block they had stored locally from more than two-thirds of all the nodes in the network, they will proceed with the phase. In the commit phase, the same procedure is repeated as

the prepare phase, and upon the satisfactory response, the proposed block is appended to their blockchain (Nguyen and Kim 2018).

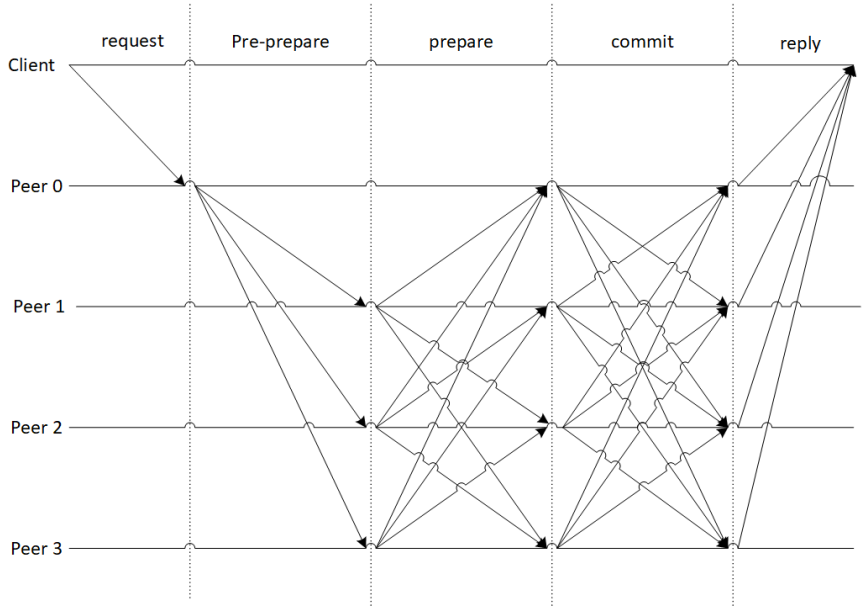


Figure 3.4: PBFT Algorithm based block creation.

3.3 MESSAGE QUEUES

Message queue technology allows applications to communicate with each other in a loosely coupled, asynchronous manner. The queues receive, buffer, store and transfer messages between applications, databases, and services. This ability to handle workload between applications and establish more reliable communication between two systems or services independently paves the path to enhance distributed systems developments.

Over the years, this technology has been used in various applications like large-scale distributed computing systems, distributed databases, web services and cloud computing solutions. A message queue service comprises a messaging server, queue manager and message store. The queue manager is responsible for receiving the messages, managing the queues, and routing the messages to the correct destinations. The messaging server is accountable for delivering the messages to the correct destination. Finally, the message store retains the messages until the recipients retrieve them. This message is a form of data packets.

The technology can be classified into two categories.

- Message Queue Brokers: Message Queue Brokers are server-side applications that provide services for receiving, storing, and delivering messages. Examples of message queue brokers include RabbitMQ[31], Apache Kafka[32], and Amazon SQS[33].

- Message Queue Libraries: Message Queue Libraries are client-side applications that provide an application interface (API) for interacting with the Message Queue Libraries. Examples of Message Queue Brokers include JMS[34] and Apache Camel[35].

3.3.1 Message Queue Technologies

The message queueing is done using different protocols and technologies. Each has its benefits depending on the purpose of utilization. We will explore some of the prominent ones to understand.

3.3.1.1 ADVANCE MESSAGE QUEUING PROTOCOL (AMQP)

Jon O'Hara developed an advanced Message Queuing Protocol (AMQP) at JPMorgan Chase in London as a cooperative open effort. He had its first deployment in the financial forum in 2007 (J. O'Hara 2007). In the meantime, a working group comprising big techs was established to create an open standard for an interoperable enterprise-scale asynchronous messaging protocol (Vinoski 2006). This open standard protocol is designed to enable the secure exchange of messages between applications and computers. AMQP is designed on the footing of store-and-forward messaging, publisher-subscriber messaging, and file transfer, Figure 3.5. To facilitate ease of use, the designers also incorporated some of the Java Messaging Service (JMS) components with some nuances. AMQP *messages* are self-contained and long-lived, and their content is immutable and opaque. The *queues* in AMQP are the core component. Messages always end up in a queue, retained until delivery or transfer. They can search and reorder messages. The *exchange* is the message delivery service of the AMQP. Choosing the carrier, the delivery form is selected

[31] https://www.rabbitmq.com
[32] https://kafka.apache.org
[33] https://aws.amazon.com/sqs
[34] Getting Started with Java Message Service (JMS) (oracle.com)
[35] https://camel.apache.org

in the exchanges. The exchanges look at the message header and determine the transfer destinations. In AMQP, different types of exchanges can serve different purposes.

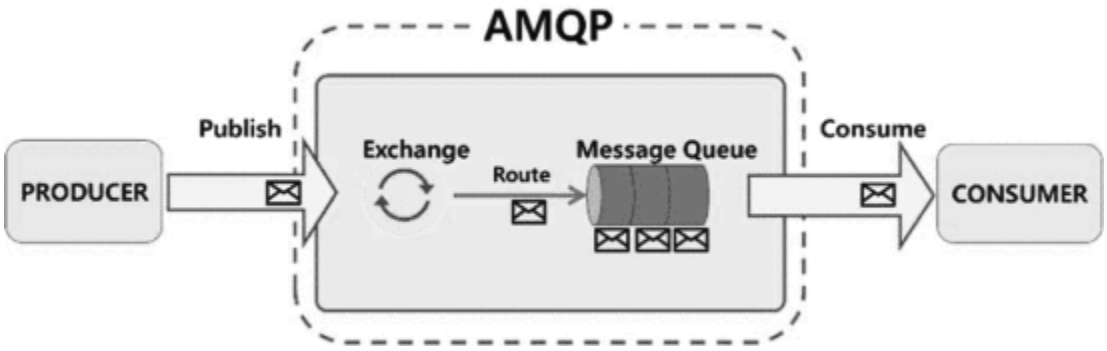


Figure 3.5: Advance Message Queuing Protocol

- *Direct Exchange*: The messages are queued to a single queue. The choice of the queue is determined by the "*routing key*" from the message header, which then matches the queue name.

- *Topic Exchange*: In the topic exchange, as the name suggests, the exchange will copy and queue the messages to all the clients who have expressed interest in the topic. This is done through a pattern matching of the topic with the routing key header.

- *Header Exchange*: In this, the exchange will compare all the headers in the message, evaluate them based on the selected predicate of the clients, and queue them.

Bindings are the arguments supplied to the exchanges to facilitate the routing of messages. AMQP works on the principle of quality of service (QoS) guarantee. Using open standards paves the path for it to be easily integrated with other technologies. One of the technologies that use this AMQP prevalently in our thesis is RabbitMQ.

3.3.1.2 APACHE KAFKA

Apache Kafka is a distributed event store and streaming platform for building real-time streaming data pipelines and applications. This was initially developed by Kreps et al. at LinkedIn (Kreps, Narkhede and Rao 2011). This was developed to provide a unified, high throughput, low latency framework to manage real-time data feeds. In this architecture, the stream of messages is classified based on a particular type called topics. The producer then publishes messages on the particular topic. These messages are stored on a set of servers called brokers. The consumer can then subscribe to one or more topics from the broker and consume them by pulling the messages from the broker (Figure 3.6). Kafka supports two types of message

topics: regular and compact. The regular topics' retention needs to be bound by time or size. In time-based retention, older topics are deleted to make space for new ones. When the space limitation is reached in size-based retention, it will delete the older topics to relieve space. In the compact topics, the messages are not expired. Instead, they are updated with the latest message based on their key. This is predominantly used to consume log messages. The message delivery state is not maintained here.

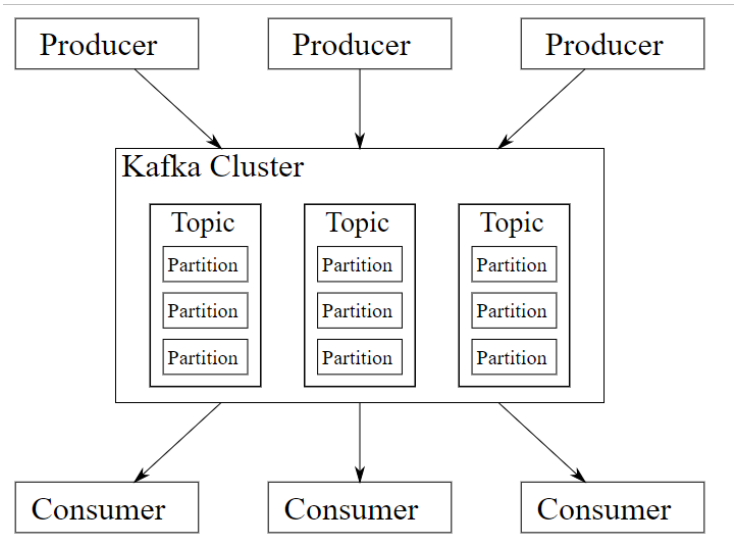


Figure 3.6: Kafka Architecture

3.3.1.3 MESSAGE QUEUE TELEMETRY TRANSPORT (MQTT)

Message Queue Telemetry Transport is an open, lightweight, publish-subscribe, machine-to-machine (M2M) protocol for M2M communication through a message queuing service. This protocol is founded on the publisher-subscriber messaging platform to distribute information between the networked applications using a message broker (Furrer, et al. 2006, Hunkeler, Truong and Stanford-Clark 2008) (Figure 3.7). Here, the publisher will send a message to the broker on a specific topic—the registered subscribers who subscribe their interest in specific topics to the broker. The broker acts as the intermediary link between the publisher and subscriber to distribute the messages it receives from the publisher to the subscriber according to the subscribed topic—this facilitates subscription to multiple topics. With lightweight, low bandwidth communication, this decoupled communication allows it to be used widely among the Internet of Things (IoT) services.

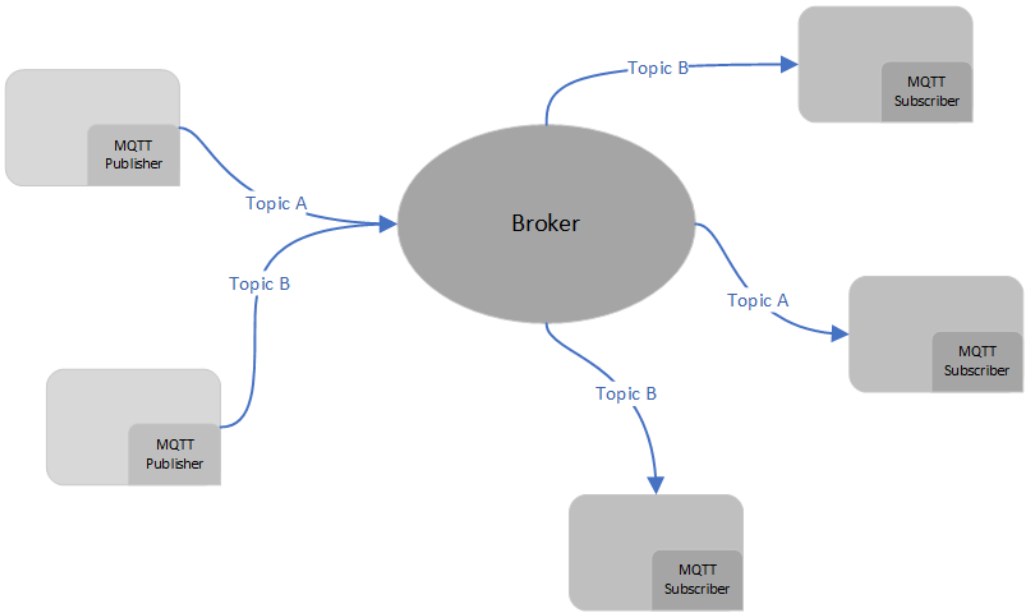


Figure 3.7: MQTT Architecture[36]

3.4 SUMMARY

This chapter explores the origins and development of Data Trust, Blockchain, and Message queuing. The literature review on Data Trust investigates its current status and potential for data sharing, highlighting ongoing debates on the implication models - public, private, or federated.

The Blockchain section covers its evolution, emphasizing the role of private blockchains, and provides an overview of consensus algorithms. Lastly, the Message Queue section discusses popular applications and protocol evolution, focusing on AMQP.

The thesis's interest in tracking data changes within the Data Trust environment using blockchain and message queues is a novel area of research. Chapters 4 and 5 will provide detailed insights into these components' design, architecture, and implementation to achieve the outlined goals from Chapter 2's problem statement.

[36] http://mqtt.org

CHAPTER 4

DESIGN AND ARCHITECTURE

We look to develop a general framework of data trust architecture in a federated model. Although, as mentioned earlier, our focus of this thesis was part of the architecture of this framework, we look forward to contributing some enhancement to the globally envisioned model. As discussed in Chapter 2, we will illustrate our proposed framework through academic research data sharing in intra and inter-academia.

4.1 PRINCIPLES

The principles of data trust architecture encompass the fundamental workflow with three essential layers: the data, access, and process layers. The data layer is where interested parties plan to share the data, while the access layer is where the pooled resources are discoverable through the data trust. Finally, the process layer is where the resources are approved via the data trust (Stalla-Bourdillon, et al. 2020).

In addition to these layers, the design principles consider different management principles. These include:

- Discovery: The ability to discover the available data resources.
- Provenance: The ability to access the quality of data resources by accessing metadata.
- Access Control: Data controllers retain the ability to determine who gets access to the data.
- Access: Appropriate users need to gain access to the data
- Identity Management: Data managers need to be able to monitor those who gain access to the data over time.
- Audit of Use: A record of the use of data needs to be tracked and monitored.
- Accountability: Data controllers are responsible for using data under their control.
- Impact: The data's value, use, and misuse need to be assessed using the audit records.

By considering these principles in the design of data trust architecture, the resulting system can enable effective data sharing while maintaining appropriate levels of control and governance.

One example of an architecture that incorporates these principles is the O'Hara architecture of the data trust portal, as shown in Figure 4.1. As discussed in previous sections, this architecture provides a guiding principle for formulating our proposed model. However, since data trust is still evolving, different approaches may be established as data trusts are established in various contexts.

Therefore, the design of data trust architecture should be adaptable to different use cases yet still maintain the fundamental principles of data sharing, control, and governance. This is particularly relevant for a federated data trust framework, where different parties agree to share data within agreed-upon guidelines and where the trust agreement is vital in disseminating and using data.

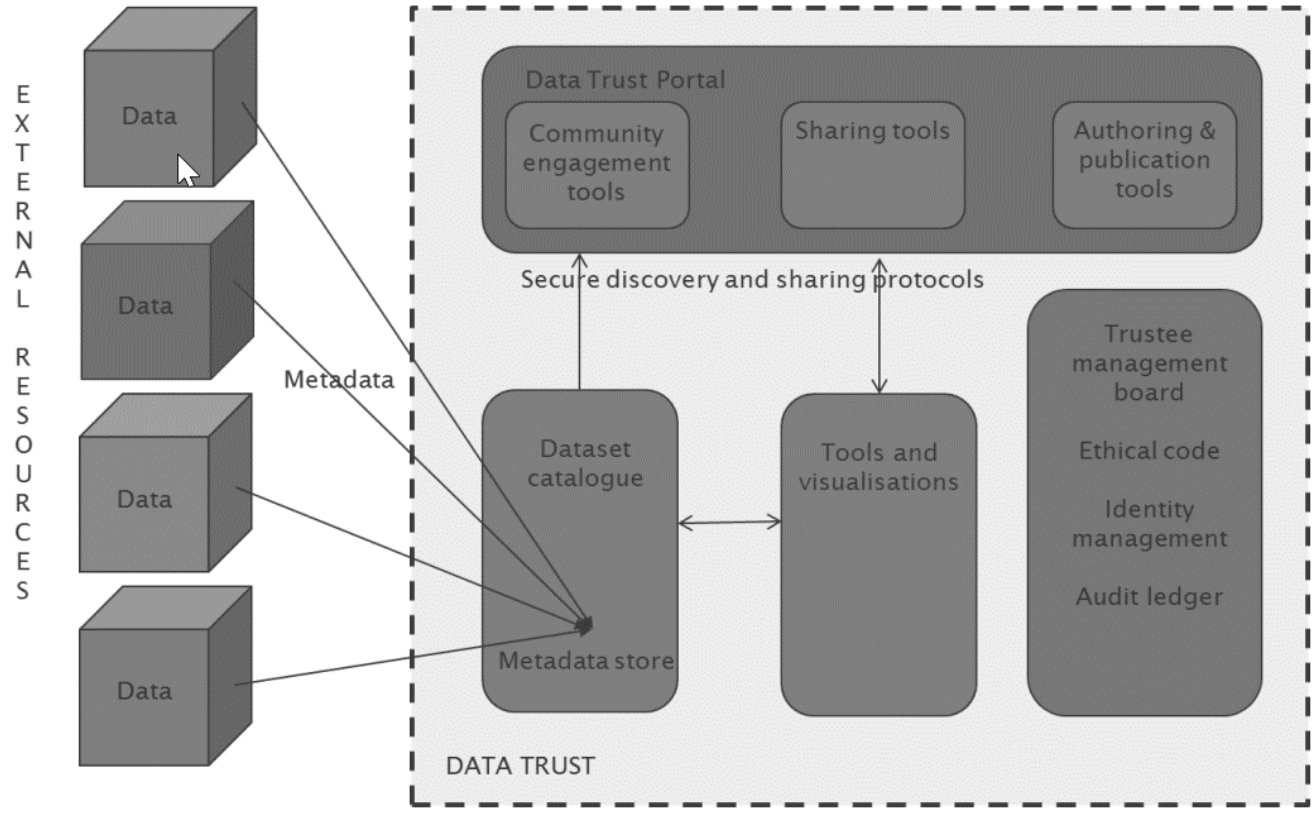


Figure 4.1: O'Hara architecture of Data Trust Portal (K. O'Hara 2019)

4.2 FEDERATED DATA TRUST

As discussed in the earlier chapters, in a federated data trust, all the federation parties agree to share their agreed data within the rules and stipulations of the data trust. In this model of data trust, the trust is vested with the responsibility to facilitate the sharing of data with sound governance, provenance, auditing, and identity management. The architecture design of this data trust, as shown in Figure 2.2, is a vision to be used by any federated members. Thus, our design or analysis will not stipulate any specific use case. Figure

4.2 depicts our detailed design model. We will explore the different components of our design elements in the following subsections. The design evolves around the basis that restful APIs deal with all access.

4.2.1 Premises

The proposed architecture assumes the following as the basis of its design.

- All transactions will be done through the restful APIs.
- The trust shall not own or store the data unless and unless otherwise agreed upon.
- The data shall reside with the producer or owner.
- The resource producer shall provide a Universal Resource Interface (URI) for the data source they intend to share.
- The resource producer or owner shall provide metadata associated with the data they intend to share.

4.2.2 Gateway

Data trust gateway provides a secured and robust data access mechanism that ensures that only authorized data consumers can access the data resources. The access mechanism also ensures that data consumers can access the data resources from anywhere at any time, provided they have the necessary credentials, permissions and compliance with the access policies.

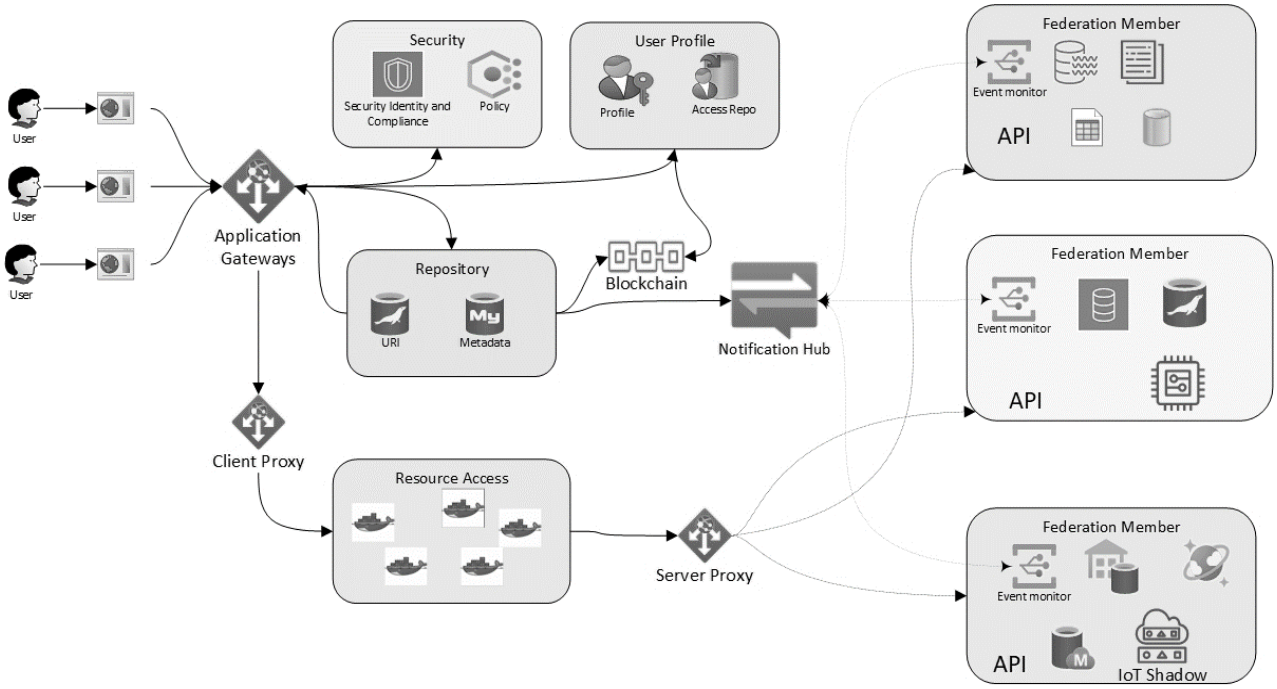


Figure 4.2 Federated Data Trust Architecture

4.2.3 Security

The security of the data layer of the data trust will assess the user authentication using the identity management system. Upon authenticating the user, authorization is carried out using role-based access control (RBAC) and attribute-based access control (ABAC).

4.2.3.1 ROLE-BASED ACCESS CONTROL

David F. Ferraiolo and D. Richard Kuhn published the first proposal of the Role Based Access Control (RBAC) (Ferraiolo, Barkley and Kuhn 1999). The premise of this model is that permission is granted to perform various actions inside an organization based on their roles, responsibilities and qualifications. In this model, object access permissions are predetermined and associated with the roles. This process is access permissions to the objects by the object owner. Then, these roles are granted to the users, authorizing the predefined access permissions on the objects or resources. Hence, permissions management is made simple as the user's responsibilities change. As this RBAC gained popularity, the Access Control List (ACL) management method was obsolete.

In our proposed architecture, we leverage this model to manage the different access grants to different management roles on the system—for example, administrator, owner, reader, auditor, etc.

4.2.3.2 ATTRIBUTE-BASED ACCESS CONTROL (ABAC)

In 2014, the National Institute for Standards and Technology (NIST) published the "Guide to Attribute-Based Access Control (ABAC) Definition and Considerations" (Hu, et al. 2014). This standard replaced the RBAC with a more robust and versatile access control system. In this model, the subject requesting access to operate on an object is granted or denied based on the assigned attributes of the subject, the object, the environmental conditions, and a set of policies specific to those attributes and conditions. For example, this can be seen in Figure 4.3.

The main components of ABAC are (Hu, et al. 2014)

- Attributes: Characteristics that define the specific aspects of the subject, object, environment conditions, and requested actions that are predefined and preassigned by an authority.
- Subject: An active entity (generally an individual, process, or device) that causes information to flow among objects or changes the system state.
- Object: A passive information system-related entity (e.g., devices, files, records tables, processes, programs, networks, domains) containing or receiving information. Access to an object implies access to the information it contains.
- Operation: The execution of a function at the request of a subject upon an object. Operations include reading, writing, editing, deleting, authoring, copying, executing, and modifying.
- Policy: The representation of rules or relationships that define the set of allowable operations a subject may perform upon an object in permitted environmental conditions.

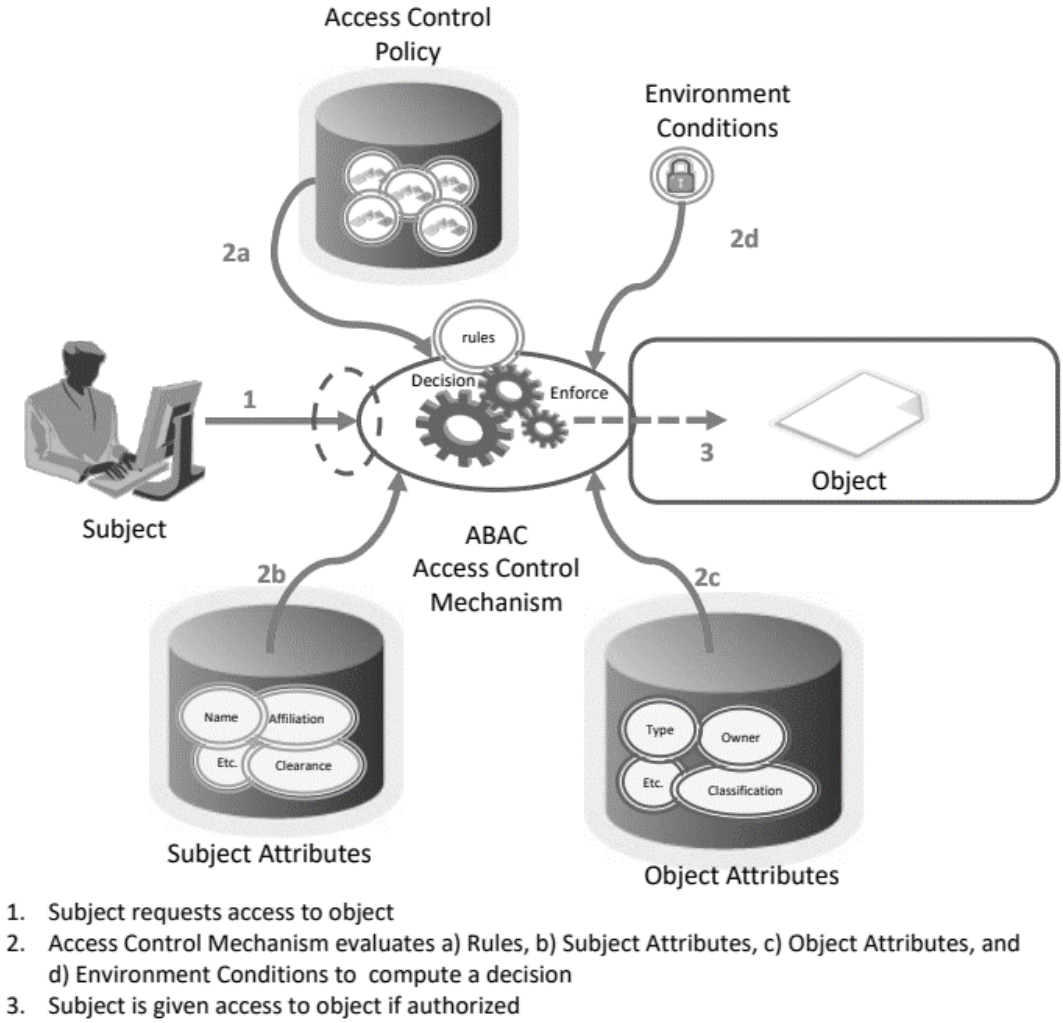


Figure 4.3: Basic ABAC scenario (Hu, et al. 2014)

We leverage this flexibility and fine-grained access capabilities to define our access policies on the resources and the requesters with other environmental criteria to fine-tune the authorization in conjunction with the RBACs. This policy is defined in collaboration with the resource owners who intend to share the resource with the membership in compliance with the governance rules.

4.2.4 Resource Layer

The resource layer of the federated data trust architecture is where the data resources are stored and managed. In this layer, the data resources are organized and made available to the data consumers through a discovery mechanism. The resource layer comprises two main components: the resource and the metadata repository.

- Resource Repository: The resource repository stores and manages the data resources. This repository organizes the data resources by type, format, resource URI and other attributes. The data resources are made available to the users through a discovery mechanism, enabling them to search and find the needed resources.
- Metadata Repository: The metadata associated with the data resources is stored and managed. The metadata includes information such as the data source, data format, data quality and data provenance. This information is crucial in enabling users to discover and evaluate the data resources before using them.

4.2.4.1 RESOURCE DISCOVERY

Resource discovery is an essential component of any data trust. The discovery mechanism enables the data consumers to find relevant resources that meet their requirements. The proposed architecture utilizes a two-stage approach for resource discovery.

The first stage involves data providers registering their data resources and metadata with the data trust gateway. This registration process involves providing a Universal Resource Identifier (URI) and metadata associated with the data resource. The metadata should include details such as data format, data quality, access rights, and the data producer's contact information.

The second stage involves data consumers querying the data trust gateway to find relevant data resources that meet their requirements. The gateway provides a search interface enabling data consumers to specify their search criteria, such as data format, quality, and access rights. The gateway then searches its database of registered data resources and returns a list of resources that meet the specified criteria.

The discovery mechanism also ensures that data resources are not duplicated and that data consumers are directed to the data source through the resource access layer. In addition, this feature ensures that data producers retain control over their data, and data consumers can trust the authenticity and provenance of the data.

4.2.4.2 DATA ACCESS

Data access is a critical component of any data trust. The proposed architecture utilizes a secure and robust data access mechanism based on RBAC and ABAC, as discussed under the security layer. Data access is granted based on the roles assigned to the users and the access policies defined by the data providers.

Data providers define access policies that specify the access rights for different roles on their data resources. For example, a data provider may define read-only access for a specific role and full access for another. Data consumers are granted access to resources based on their assigned roles and access policies defined by data providers.

4.2.5 Data Governance

Data governance is an essential component of any data trust. Therefore, the proposed architecture utilizes a data governance mechanism that ensures data resources are used appropriately and ethically. The data governance mechanism comprises the following components:

- Data Privacy: The data privacy component ensures that personal data are protected and used for the intended purpose. Data providers define access policies that specify conditions under which personal data can be used, and data consumers are granted access based on these policies.
- Data Quality: The data quality component ensures that data resources meet the required quality standards. Data providers define quality metrics for their data resources, and data consumers can access these metrics to assess the data quality.
- Data Ethics: The data ethics component ensures data resources are used ethically and responsibly. Data providers define ethical guidelines for their data resources, and data consumers must adhere to them when using them.
- Data Ownership: The data ownership component ensures that the producers own data resources and that the consumers cannot claim ownership. Data producers define ownership policies that specify the conditions under which data can be shared and used, and data consumers are granted access based on these policies.

4.2.6 Audit

The audit component of the federated data trust architecture is where the data transactions and activities are recorded and maintained. The audit component comprises a blockchain-based mechanism that enables producers and consumers to trace and verify the data transactions and activities. The audit mechanism provides the following benefits:

- Data Provenance: The audit mechanism enables consumers to trace the history of the data transactions and activities, which is crucial in ensuring data quality and integrity.
- Data Security: The audit mechanism provides a tamper-proof and immutable record of the data transactions and activities, which is crucial in ensuring data security and privacy.

- Data Accountability: The audit mechanism enables producers to hold consumers accountable for their actions and usage of the data resources.

4.2.7 Notification Layer

The notification layer is a critical component in the proposed data trust architecture. It is responsible for facilitating the dissemination of information and notifying interested parties of any changes made to the shared data resources. This is achieved through message queues, which allow members to subscribe to the notification system associated with specific resources.

When a change to a resource occurs, the responsibility of publication is vested in the resource owner. The resource owner publishes the change, and a notification message containing the change and time information is added to the message queue. The subscribed members can later read the message and carry out any necessary tasks on the resource.

In this framework, a resource queue is set up every time a resource is created. The owner is added as a publisher and subscriber, allowing them to validate the publication and dissemination of the messages to the subscribers. This means that owners of data resources become both producers and consumers of data, a concept known as prosumers.

The notification layer plays a crucial role in facilitating communication between members of the data trust. It ensures that all interested parties are notified of any changes to shared resources, allowing them to take appropriate action. It also provides transparency in the data-sharing process, as a member can monitor and track changes made to the shared resources.

4.2.8 Proxy Interface

The proposed model provides access to the resources through a resource access pool. In this model, a resource requester must provide their program code to the "trust," which will evaluate the code for safety and security before setting up a resource share in the form of docker containers in the resource access pool. The resource access point is then shared with the requestor, who can access the resources through the access point provided.

Two proxies protect the resource: a client proxy and a server-side proxy. Upon data access authorization, consumer requests are passed to the client proxy. The client proxy will directly access the corresponding

docker container. Then, the container will use the server-side proxy to process the request. The processed request is then passed through the client proxy to the consumer or the requestor. This helps to ensure that in the event of misuse, access to the resource can be revoked at the proxy levels, adding a layer of security to the access to the resources.

This approach adds a layer of security to the access of resources, making it more difficult for unauthorized parties to access or compromise the resources. By implementing this trust-based model and using proxies to control access, the system can provide secure access to valuable resources and minimize the risk of security breaches.

4.2.9 RESTful APIs

In this framework, all access to the users is facilitated through the Restful APIs. This restful API information is recorded and maintained as part of the metadata information, enabling the discovery of the resources. Furthermore, access to these APIs will pass through the gateway to handle any access controls. Finally, the resource data is disseminated to the requester upon successful compliance with the requests.

4.3 BENEFITS

A federated data trust can offer several benefits compared to a centralized data trust. Some of these benefits are:

- Enhanced Data Privacy: In a federated data trust, data remains within the control of the original data owner, and only the metadata is shared with other trust members. This limits the exposure of sensitive data to external entities, reducing the risks of data breaches and unauthorized access.
- Increased Data Quality: With a federated data trust, data producers maintain ownership and control over their data, ensuring that data quality is maintained. Data producers can ensure accuracy, completeness, and consistency, leading to higher-quality data overall.
- Greater Flexibility: A federated data trust is more flexible than a centralized data trust, as it can accommodate varying data requirements and diverse use cases. With a federated model, different data sources can be combined to provide a comprehensive view of data, enabling more effective decision-making.
- Better Governance: A federated data trust can promote better data governance by enabling data producers to retain control over their data while providing a platform for data sharing. This can

promote trust between data producers and consumers, facilitating more effective data collaboration and sharing.

- Cost Saving: A federated data trust can reduce data storage and management costs. With a federated model, data can be stored locally, reducing the need for centralized storage and processing. This can also reduce costs associated with data duplication and facilitate more efficient use of resources.
- Increased Innovation: A federated data trust can promote innovation by enabling the development of new products and services based on shared data sources. By providing a platform for data collaboration and sharing, a federated data trust can enable new insights and discoveries that may not have been possible with individual data sources.

4.4 USE CASES

A federated data trust can be applied in various contexts, including:

- Healthcare: Data trust can help share patient information among multiple stakeholders, including healthcare providers, insurers, and researchers. This would enable more accurate diagnoses, better treatment plans, and improved patient outcomes. For example, federated data trust could help identify patients with a specific medical condition, collect data from various healthcare providers, and create a centralized, anonymized data set to develop new treatments.
- Smart cities: Data trust can be used to manage and share data in smart cities. A federated data trust can help integrate data from different sources, such as traffic sensors, public transportation schedules, weather data and more, to help manage traffic flow, optimize transportation, and ensure public safety. In addition, by sharing data, stakeholders can develop more informed and effective policies to manage the city.
- Banking and finance: Banks and financial institutions could use data trust to share financial data, improve customer experience, detect fraud, and reduce operational costs. With federated data trust, banks can share transaction data, credit scores, and other financial data with partners without compromising customer privacy or data security.
- Education: Data trust can help improve educational outcomes by integrating data from various sources, such as student records, test scores, and teacher evaluations. By sharing data, education providers can develop personalized learning plans, identify improvement areas and offer better support to students.
- Environmental monitoring: Data trust can be used to monitor the environment by sharing data from multiple sources, including satellite images, weather sensors, and IoT devices. By sharing data and

analyzing it, environmentalists can detect changes in air or water quality and take appropriate actions to mitigate the impact of climate change.

- Public safety: Federated data trust can also be used for public safety, especially in emergencies or disasters. For instance, emergency services providers such as police, firefighters, and ambulatory services could share data to optimize their response times, improve communication and collaboration during emergencies, and provide better care to affected individuals.

CHAPTER 5

IMPLEMENTATION

This chapter details the implementation of our data trust framework in detail. As mentioned earlier, this deployment is based on RESTful API as the interface to interact with the application. In the following sections, we will discuss the implementation while highlighting the focus of our thesis on the change-tracking process.

5.1 DATA TRUST IMPLEMENTATION ARCHITECTURE

In the federated data trust model, the data remains in the owner's custody. The data is accessed through the RESTful API calls. In this implementation, we have divided the architecture into different application groups. These application groups are decoupled to be deployed on different servers and containers. We narrowed our implementation focus to user management, policy management, data management, proxy access management and access management modules.

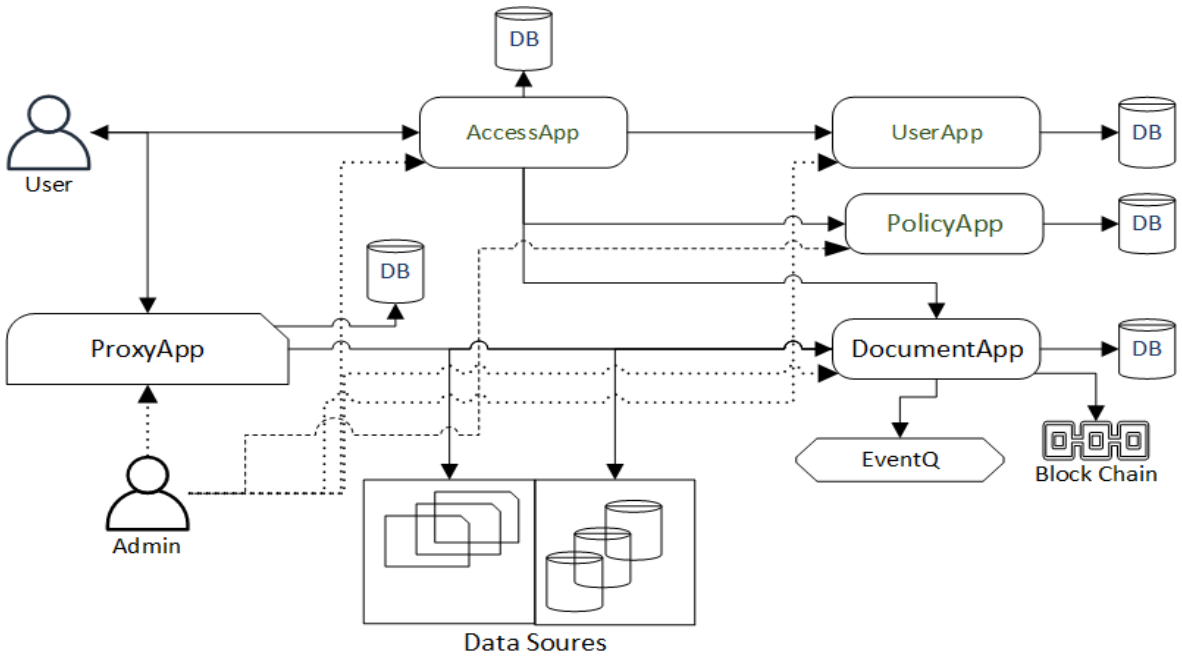


Figure 5.1: Data Trust Implementation

In this framework in Figure 5.1, the UserApp holds the user profiles, the PolicyApp maintains the policy profiles, and the DocumentApp manages the document profile. The AccessApp joins the user profile, policy

profile and document profile to define access to the document. The ProxyApp controls the proxy access list to the documents. The DocumentApp is defined in detail to illustrate event management through the message queues and change audit management through the blockchain.

5.2 SERVICE DESCRIPTIONS

Each endpoint within these services is designed to handle specific tasks, facilitating modularity and ease of maintenance.

5.2.1 UserApp Service

The UserApp Service manages the user relationship to the application. Hence, it comprises user, role, and organization endpoints to manage each sub-service.

1. User Endpoint
    - Create User: Endpoint for registering new users.
    - Update User: Allows modification to the use details.
    - Delete User: Endpoint to remove users.
    - Retrieve All Users: Endpoint to list all the users in the repository.
    - Retrieve User By ID: Endpoint to fetch a single user by user ID.

2. Role Endpoint

    This defines different roles a person or persons could have in the data trust.
    - Create Role: Endpoint for defining new roles.
    - Update Role: Allows changes to the existing roles.
    - Delete Role: Endpoint to remove a role.
    - Retrieve All Role: Endpoint to list all the role details in the repository.
    - Retrieve Role By ID: Endpoint to fetch a single role detail by role ID.

3. Organization Endpoint

    This maintains the list of organizations allowed in the data trust.
    - Create Organization: To create new organizations.
    - Update Organization: Modify the details of an existing organization.
    - Delete Organization: Remove an organization.
    - Retrieve All Organization: Endpoint to list all the organizations in the repository.

- Retrieve Organization By ID: Endpoint to fetch a single organization by organization ID.

4. User Role Endpoint

   This defines the user's role in an organization.

   - Assign Role: Assign roles to users within the organization.
   - Update Assignment: Allows the alteration of the user's role assignment in the organization.
   - Remove Assignment: Deletes existing role assignments.
   - List All Assignments: List all the mappings in the repository.
   - List By Assignment ID: Fetch a single assignment by userrole ID.

5.2.2 PolicyApp Service

PolicyApp service has the policy management endpoints to manage different policy points. They are:

1. Resource Endpoint

   This defines the list of resources that can be used in the policy management.

   - Define Resource: Endpoint to create new resource entry for the policy management.
   - Modify Resource: Allows updating the resource information.
   - Remove Resource: Endpoint to remove the resource from the resource repository.
   - View Resource: Retrieve details of specific resources.
   - View Resource by ID: Retrieve details of a specific resource.

2. Permission Endpoint

   Defines the types of access permissions that are allowed.

   - Create Permission: Create new permission.
   - Update Permission: Modify the details of the defined permissions.
   - Delete Permission: Delete permission from the repository.
   - List All Permissions: List all the permissions in the repository.
   - List Single Permission: List specific permission by id.

3. Rate Limit Endpoint

   This defines different rate limits that can be applied to a policy.

   - Set Rate Limit: Set the rate limit to the resources.
   - Update Rate Limit: Modifies the existing rate limit.

- Remove Rate Limit: Remove a rate limit definition.
- List all Rate Limits: List all the records' rate limit details.
- List Rate Limit by ID: List a specific rate limit detail.

4. Size Limit Endpoint

   Different size limits that could be applied to policies are defined.

   - Set Size Limit: Define a new size limit.
   - Modify Size Limit: Update the details of an existing size limit.
   - Delete Size Limit: Delete an existing size limit.
   - View Size Limits: List all the size limit entries in the repository.
   - View Size Limit by ID: List a specific size limit from the repository.

5. Location Endpoint

   This defines the location to which access can be limited in the policy.

   - Define Location: Create new location definitions.
   - Update Location: Modifies details of an existing location.
   - Remove Location: Deletes an existing location.
   - Retrieve All Locations: Fetch all the location entries in the repository.
   - Retrieve Location By ID: Fetch a specific location detail.

6. Policy Endpoint

   The definition of policy encompasses collecting all the above-listed endpoint data under PolicyApp to create different policy mappings.

   - Create Policy: Endpoint to define new policies.
   - Update Policy: Modifies an existing policy.
   - Delete Policy: Deletes an existing policy.
   - Retrieve All Policies: Retrieve all the policies and their details.
   - Retrieve Policy By ID: Retrieve a specific policy and its details.

5.2.3 DocumentApp Service

This has the endpoint to manage the data trust’s document resources. This comprises the following endpoint services.

1. Subscription Type Endpoint

   Defines different types of subscriptions that one can get. Example: paid or free.

   - Create Subscription Type: Define a new subscription type.
   - Update Subscription Type: Modify an existing subscription type.
   - Delete Subscription Type: Remove an existing subscription type.
   - List All Subscription Types: Fetch all the subscription types in the repository.
   - List Subscription Type By ID: Fetch a specific subscription by its ID.
   - List Subscription Type by Name: Fetch a specific subscription by its name.

2. Document Endpoint:

   This is the core of the data trust. This holds the data and metadata about the document source to which the federated member provides access.

   - Create Document: Create a new document resource.
   - Modify Document: Modify an existing document.
   - Delete Document: Remove an existing document.
   - List All Documents: List all the documents in the repository.
   - List Document by ID: List a specific document by its ID.
   - List Document by Name: Fetch a document by its name.

3. Subscription Endpoint:

   This defines the subscription mapping to the userrole and subscription type to the document.

   Create Subscription: Create a new subscription.

   Modify Subscription: Modify an existing subscription detail.

   Delete Subscription: Remove an existing subscription.

   View All Subscriptions: List the details of all the subscriptions in the repository.

   View By SubscriberId: List the subscriptions in detail by subscriber ID.

   View By Subscription ID: List the subscriptions in detail by subscription ID.

   View By Subscriber ID and Subscription ID: Fetch all the records that a subscription and subscriber ID matches.

4. Block Chain Endpoint:

   This endpoint we use internally to ensure the data changes happen when we create a new document.

   View Chain Data by Document ID: List the Chain data using the document ID.

5.2.4 ProxyApp Service

1. Proxy Access Endpoint

   This defines the access point to the proxy URIs.

   - Proxy Access URI: Proxy access to the document source data

2. Proxy Access Management Endpoint

   This provides a management entry point to override any entries created by the DocumentApp on proxy access to source data of the resource that resides outside this application.

   - Create Proxy: Create a new proxy access to the outside external resource.
   - Modify Proxy: Update the details of the proxy resource.
   - Delete Proxy: Remove the proxy entry.
   - View All Proxies: View all the defined proxies in the repository.
   - View by Proxy Access ID: View a specific proxy detail by proxy access identification.

5.2.5 AccessApp Service

1. Access Endpoint

   This defines the access audit to our application.

   - Create Access: Defines an access entry to the application.
   - Modify Access: Updates the access entry upon each access.
   - Delete Access: Remove an access entry if one exists.
   - View All Access Entries: List all the access entries in the application.
   - View Access Entry By ID: View a specific access entry.
   - Validate Access by ID: Validate access compliance by ID.
   - Check Access Exists: Check if an access entry exists by access ID.

2. User Role Policy Endpoint

   This defines the mapping of userrole to policy.

   - Create User Role Policy: Creates a new user role for policy mapping.
   - Update User Role Policy: Modifies the user role to policy mapping details.
   - Delete User Role Policy: Removes the user role policy by id.
   - View All User Role Policies: List all the user role policies defined in the repository.
   - View User Role Policy by ID: List a specific user role policy detail.

- View User Role Policy Flattened: View all the data in detail of a specific user role policy by ID.
- Check If Exists: Check if the user role policy exists.
- Check if Active: Check if the user role policy state is active.

5.3 TECHNICAL STACK

This section will discuss the software components we used to implement the project's backend. All these components run in docker containers on a single instance Google Cloud VM on an e2-medium machine in the US-central region.

5.3.1 NodeJS

NodeJS[37] is a JavaScript runtime built on Google Chrome's version 8 JavaScript engine. Its versatility in handling different backend logic and its ability to carry out asynchronous processing with a non-blocking I/O model makes it an efficient performer under load, and its scalability makes it a better choice for this implementation. We use NodeJS version 20 for our container modules. Each service implementation has an integration that interacts with PostgreSQL for database operation. In one of the core areas, DocuementAPP integrates with Hyperledger Fabric for blockchain-based auditing and RabbitMQ for event-driven architecture.

5.3.2 PostgreSQL

PostgreSQL[38] is a powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance. At the time of the inception of this project work, PostgreSQL version 13 was the latest available version. We use it as an online transaction processing (OLTP) database backend to store the transaction data of this application. The NodeJS application services interact with the database to carry out the different data manipulation language (DML) operations on the database.

[37] https://nodejs.org
[38] https://postgresql.org

5.3.3 RabbitMQ

RabbitMQ[39] is one of the most popular open-source message broker systems. Its lightweight, versatile message protocol support and ability to deploy in a distributed or federated architecture make it the number one choice for our notification system. This project uses version 3.12, running on Erlang version 25.3.2, to create and publish change notification messages so that the subscribers of the corresponding events can consume and carry out any needed action on their part. The application implementation is done on a single node, RabbitMQ. If a performance is to get any impact, one can make it a cluster with multiple nodes. Another strategic design choice is the use of the persistent queue. As a result, there are some performance penalties to endure for the safety and persistence of the messages.

5.3.4 Hyperledger Fabric Network

The project uses Hyperledger Fabric version 2.2 to support sound audibility. We have used the test-network framework of the Hyperledger Fabric for this project to evaluate the overall architectural benefits of blockchain in the system. This project has two organizations with one peer blockchain, with CouchDB, Figure 5.3. We chose to write our chain code using JavaScript. We use it to store the documents' hash for integrity verification and audit trail. Choosing to store the hash value gives the application the versatility to store the data on any blockchain without fearing data disclosure. The DocumentAPP NodeJS module interacts with it to store and retrieve the document hash in the blockchain through the DocStateList smart contract, as in Figure 5.2.

[39] https://www.rabbitmq.com

```
async CreateDoc(
    ctx,
    docId,
    changedBy,
    changeType,
    docHash
) {
    const exists = await this.DocExists(ctx, docId);
    if (exists) {
        throw new Error(`The document already exists with the given docId ${docId}`);
    }

    const changeTypeList = ["NEW", "UPDATE", "GET", "DELETE"];


    // Validate the input parameters
    if ( !changedBy || changedBy.length === 0) {
        throw new Error(`Change author cannot be empty. provide a valid changedBy entry of the user`);
    } else if (!changeType || changeType.length == 0)  {
        throw new Error(`ChangeType cannot be empty. provide a valid changeType entry`);
    } else if (!changeTypeList.includes(changeType))  {
        throw new Error(`ChangeType can only be ${changeTypeList}`);
    } else if (!docHash || docHash.length == 0)  {
        throw new Error(`Document HASH cannot be empty. provide a valid docHash`);
    }

    const dt = new Date().toUTCString();

    const docData = {
        DocId: docId,
        ChangedBy: changedBy,
        ChangeType: changeType,
        DocHash: docHash,
        docType: 'DocRepo',
        ChangeDate: dt
    };

    // We now insert the record in the alphabetical order
    await ctx.stub.putState(docId,Buffer.from(jstringify(sortKeysRecursive(docData))));
    return JSON.stringify(docData,'utf8');
}
```

Figure 5.2: Create Document function of DataStateList Chain Code Snippet

Figure 5.3: Docker image of participating nodes in the Fabric Network

5.4 DATABASE DESIGN

The Entity relationship diagram (ERD) Figure 5.4 showcases the entities and attributes that will be part of this application.

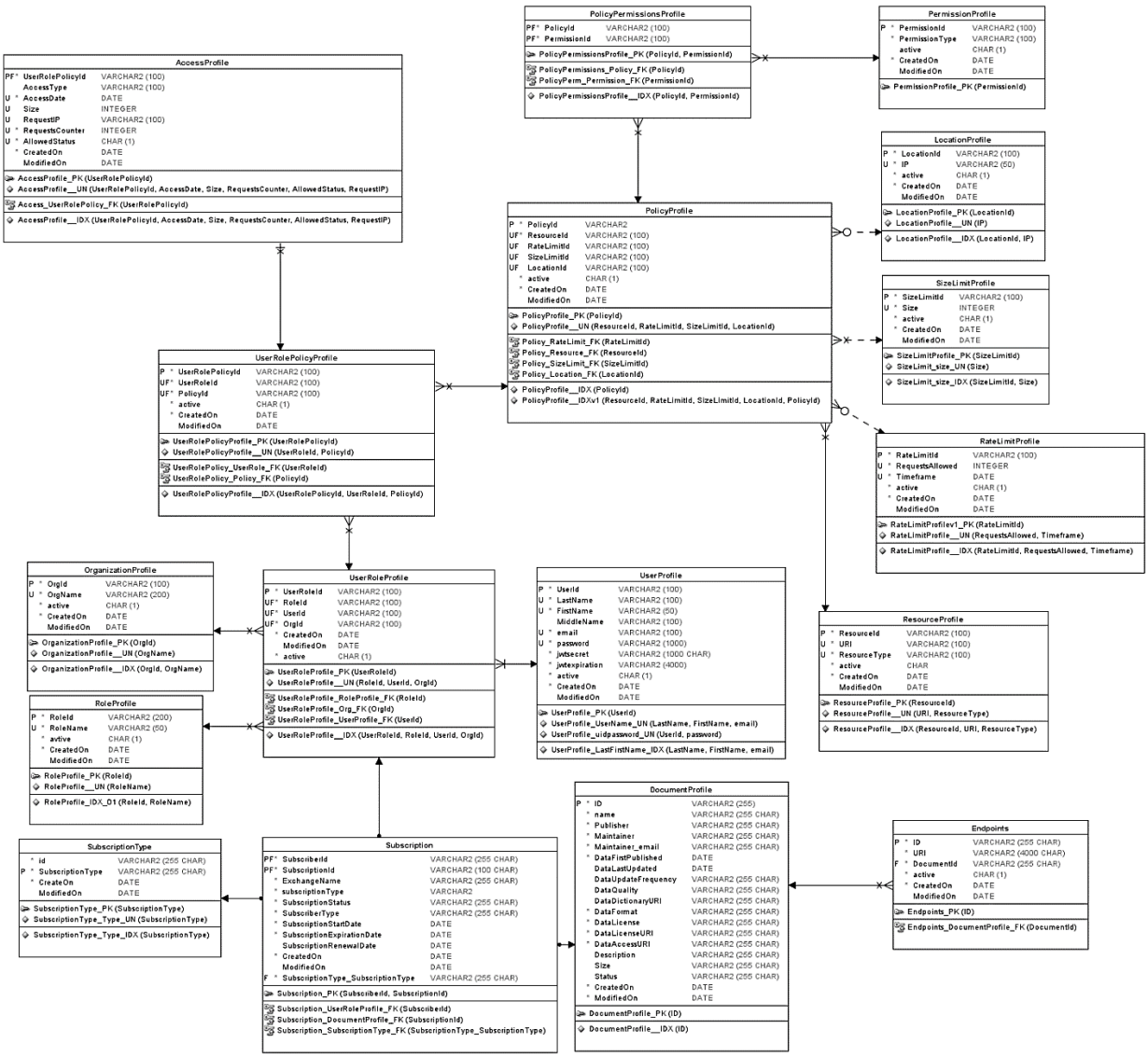


Figure 5.4: Data Trust Entity Relationship Diagram

5.5 INTEGRATION

In this section, we describe the integration of the backend services that facilitate the operation of this implementation. We had chosen Docker for its flexibility to simulate disconnected application services. All the services mentioned above in Section 5.2 run in docker containers, as shown in Figure 5.6. To successfully operate the application, some application entries need to be populated to facilitate access. They

hold a list of organizations that are allowed to participate in the network. The Role records the list of roles a user can inherit to carry out the duties in Trust. For simplicity, we have defined three roles: ADMIN, OWNER, and VIEWER. They are self-explanatory in their ability to access or control different services in this application. One can also define the policy to allow access to different service entry points. These policies can be extended to enforce the limitations on proxy access to the document, which we will discuss in detail below.

As you can comprehend from the above database design Figure 5.4, these different applications have their dependency established to render various services. Since our vision is to have all these services operated independently, it also added a layer of complexity regarding communication between services. We address this limitation by using an HTTP client for NodeJS called Axios[40]. We use this to carry out the dependency and access validation between the application services, as seen in Figure 5.5.

```
/**
 * @description Get UserRolePolicyMap by Id
 * @param id
 * @returns data object
 * @memberof AccessPolicyService
 */
async getAccessPolicyById(id: string): Promise<any> {

    try {

        if (id === undefined) {
            throw new Error('id is undefined');
        } else if (id === null) {
            throw new Error('id is null');
        }

        const baseUrl = process.env.ACCESSAPP_URL || "http://localhost:2020";
        const response = await axios.get(`${baseUrl}/api/v1/userrolepolicymap/view/${id}`);

        if (response.status !== 200) {
            throw new Error(`AccessPolicyService  Error, getAccessPolicyById: ${response.statusText}`);
        }

        return response.data;
    }
    catch (error) {
        console.log("Internal Server Error:", error);
        throw new Error(`AccessPolicyService  Error, getAccessPolicyById: ${error}`);
    }
}
```

Figure 5.5: Axios Call Code Snippet

[40] https://axios-http.com

Figure 5.6: Docker Containers Running Application, Database and RabbitMQ

This thesis is centered on documentapp services. Even though it can be stated that the Hyperledger Fabric and RabbitMQ technology stack can be included in each component of the other services, we are demonstrating its use through this application stack. Whenever a new document needs to be created, it goes through the process as illustrated in Figure 5.7. The application receives a client post request. Upon receipt, it carries out some of the following operations. Document object data that is received contains the dataAccessURI. This denotes the location of the source data that the application owner manages. This is replaced with the proxy URI to facilitate access to the source through proxy. The process expects the header to contain two header data attributes: admin and subscriberId. The admin identifier ensures that only the trust-authorized administrator can create the document in the trust repository. The subscriber ID is used to assign ownership to the document that gets created by the administrator. This information is recorded in the publisher attribute of the document. Upon asserting the validity of the two identities we had gathered from the header, the next step will be to create the document with the dummy proxy URI defined for the dataAccessURI. Upon successfully creating the document, we fetch it using the new document ID to assert its correctness from the newly created document. We first determined if the document had been approved for access. The process uses this approval status and the source dataAccessURI to create a proxy endpoint access to the document. Once the proxy endpoint is created, the object with the proxy endpoint ID is returned. It then uses this proxy endpoint ID to define the new proxy access URI in the document service. This information is updated in the new document dataAccessURI attribute. The newly updated document object is then encrypted. This encrypted hash data is then stored in the Hyperledger fabric blockchain using the sample chain code shown in Figure 5.2. Upon successfully creating the block in our two organizations' blockchain Figure 5.9, the process flow will create a new exchange name docExch-documentId in RabbitMQ. The process creates a new exchange for every document created in the repository to facilitate ease of access and management Figure 5.10. This is a strategic design choice. The process then proceeds to create a subscription object in the subscription repository. The subscription endpoint resource will determine who has a subscription to access the data source in the data trust. It also helps to determine who has access to which document source and which exchange in the RabbitMQ. Upon successfully creating the subscription, the process will continue to publish a message with the information of document ID and document change state Figure 5.13. In this process, we always make the owner the subscriber to the exchange in the RabbitMQ so that the owner can monitor all the changes that happen to their document information in the data trust by subscription to the exchange. The new document information is returned to the client as a JSON object after completing the above-stated sub-operations. The development used POSTMAN to conduct the RESTful API calls to different services to test their operational integrity. Figure 5.8 illustrates the request-response of this POST method on document resources.

Upon receiving a PUT request from the client for the update operation of this document, we will ensure the access requirements have been complied with. Upon completing this check, the document is fetched from the repository to which the changes are applied. The process checks for any update to dataAccessURI. If any changes to it are observed, the corresponding proxy endpoint access services are called to update the new changes to the data access URI. The document changes are then updated to the repository, and a newly updated document is fetched. The newly updated data is passed through encryption, and the hash data is updated in the blockchain. A change notification message is published to the document exchange to notify the subscribers of the document change. This process can be seen in Figure 5.11, Figure 5.12, and Figure 5.13.

The DELETE operation request will validate the access compliance on the system for the given requestor. Upon completion, it will commence carrying out the bottom-up removal operation. Here, it will first begin by identifying all the subscriptions to the document ID and removing the subscriptions. Then, it continues to remove the exchange to keep from the RabbitMQ. Upon removal, the process will continue to send a delete request to the blockchain so that an audit is in place. Next, the process will continue to remove the endpoint proxy access record and the document Figure 5.14.

The GET operation is more of a lightweight operation that will send a request to the backend server. The backend server will check the access compliance, fetch the record by ID from the repository, and return the JSON result to the client.

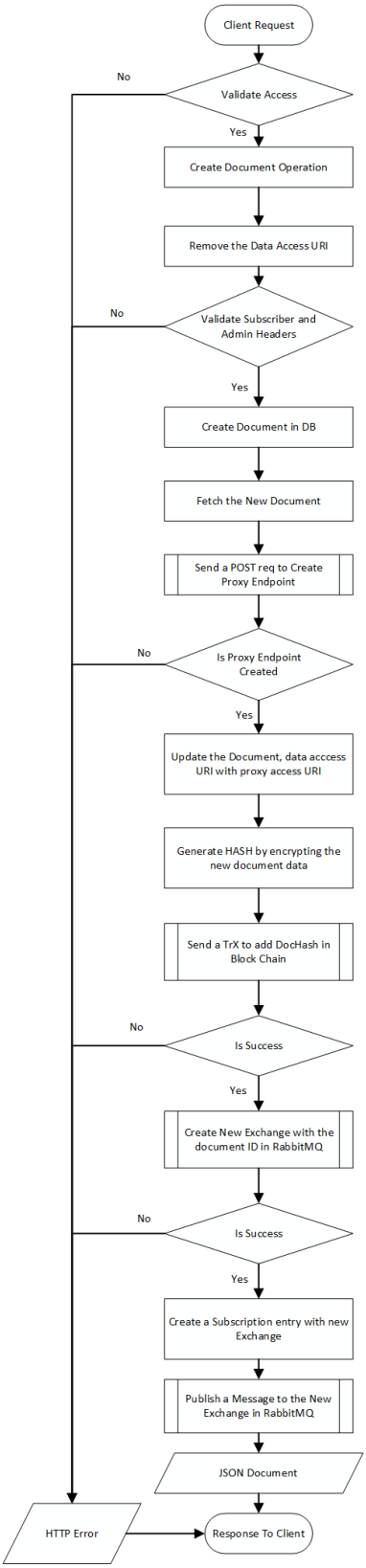


Figure 5.7: Document POST method workflow

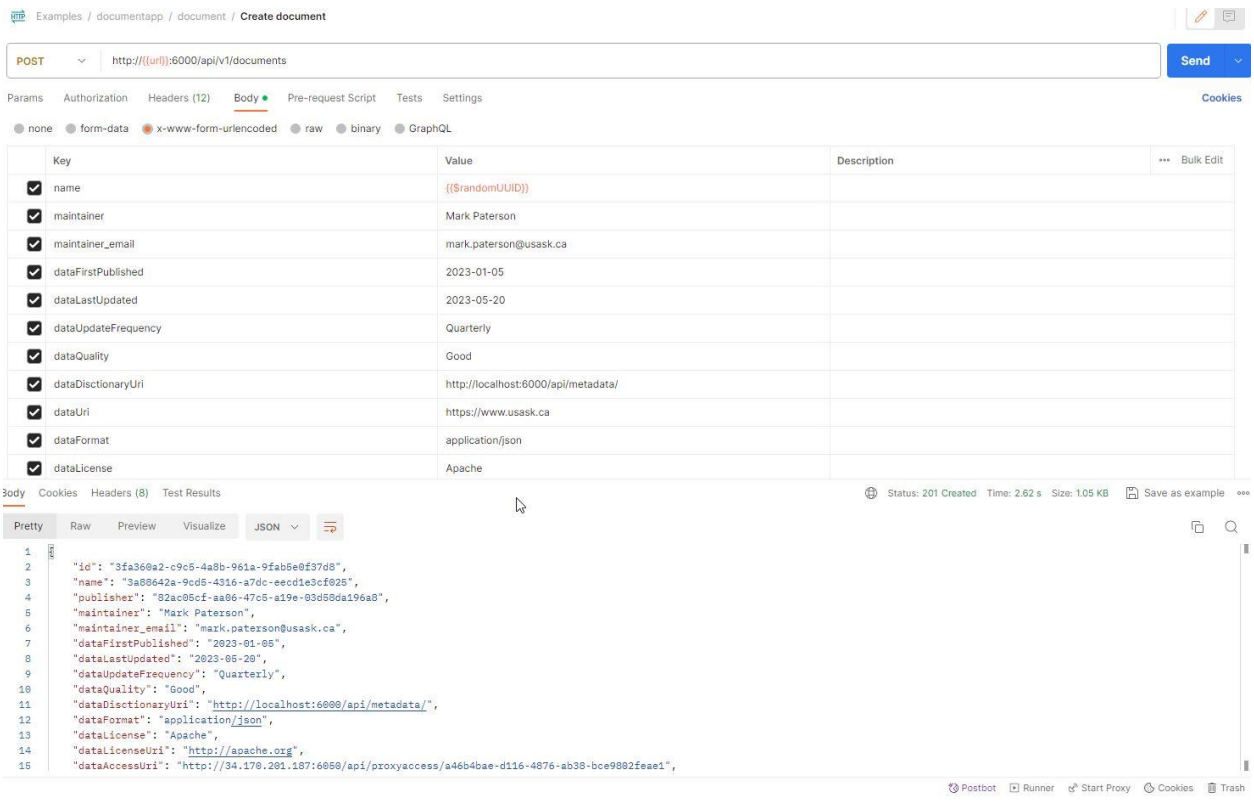


Figure 5.8: Document Resource POST method

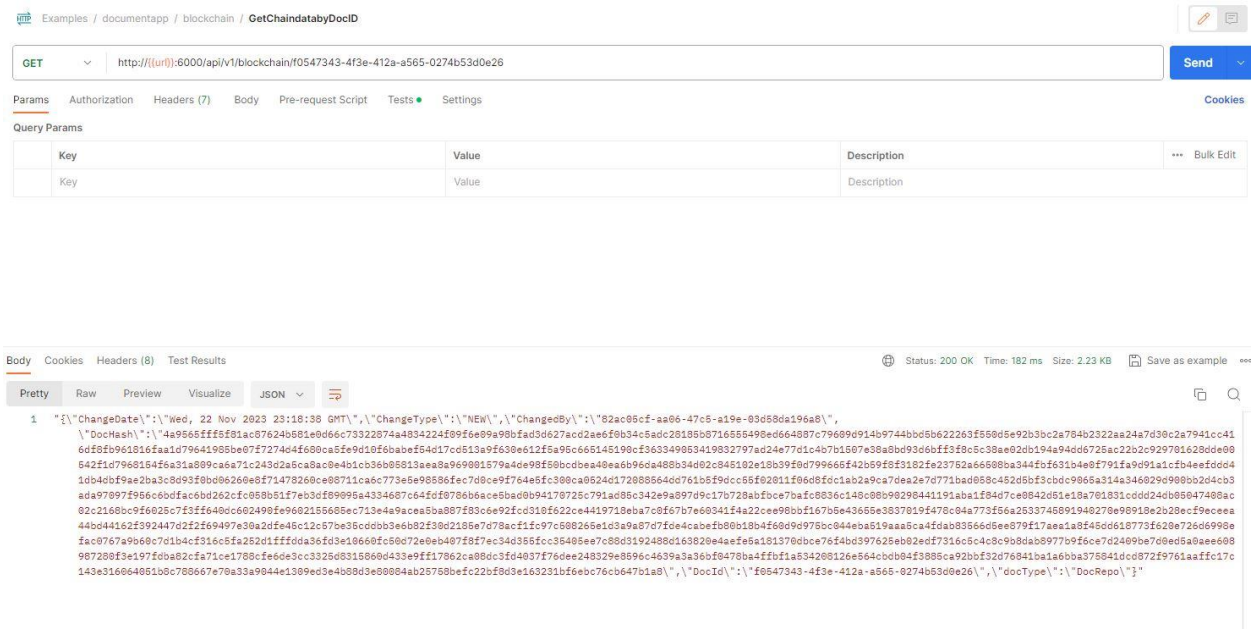


Figure 5.9: Hash of the Document in Block Chain

/	**docExch-5fa39e6d-80a2-4f03-baa4-eb5d1d2e9f57**	direct	D	0.00/s	0.00/s
/	**docExch-b5ffd628-0f3a-4239-91d4-62e780624e25**	direct	D	0.00/s	0.00/s
/	**docExch-f0555051-af45-498b-974c-881fbf2acfa4**	direct	D	0.00/s	0.00/s
/	**docExch-fa368675-cd7c-4ed5-a62f-58ed74a65e7d**	direct	D	0.00/s	0.00/s

Figure 5.10: RabbitMQ Exchange

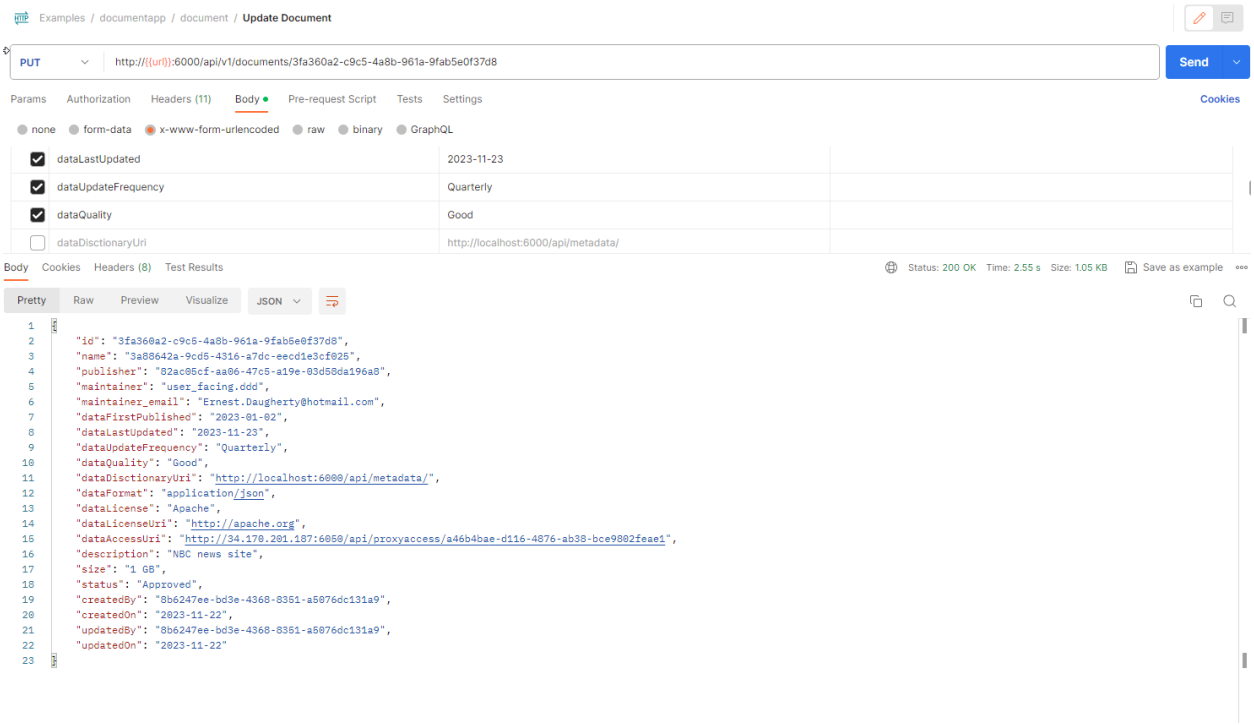


Figure 5.11: PUT method to Document resource.

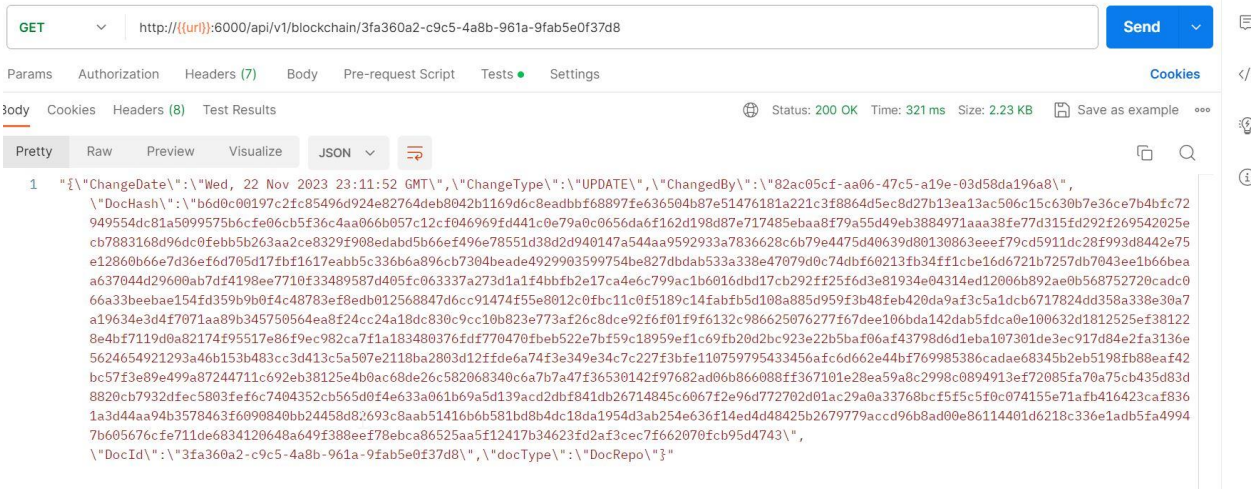


Figure 5.12: Blockchain record after update.

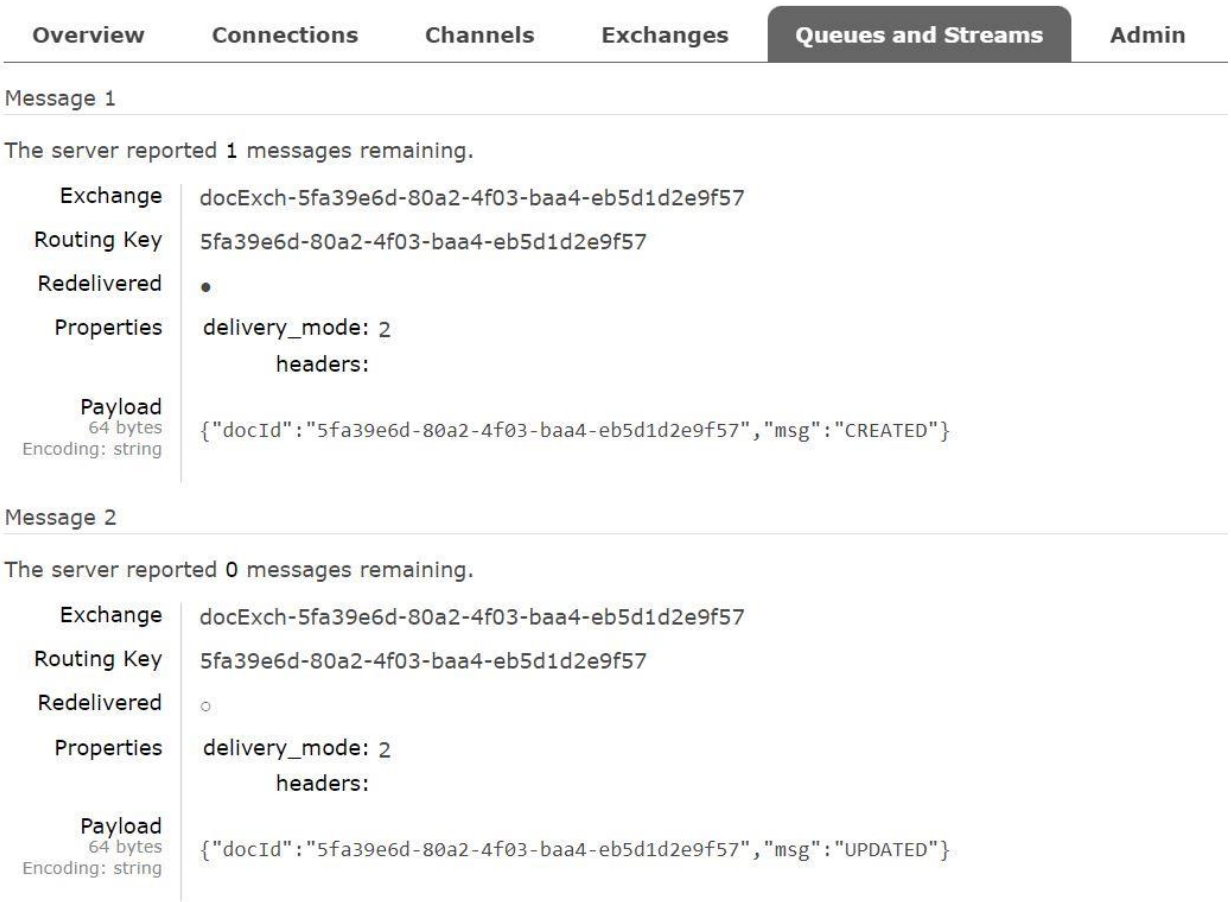


Figure 5.13: Messages in the Exchange

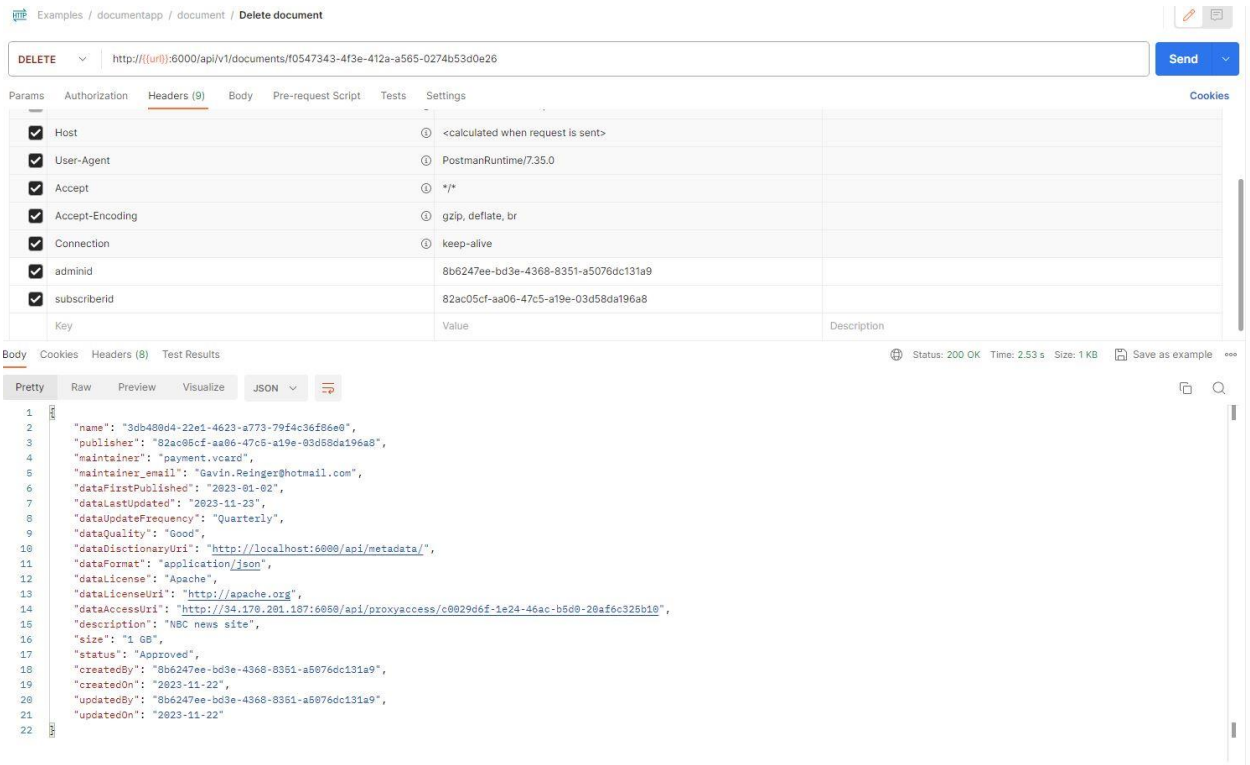


Figure 5.14: Delete request response on document resource.

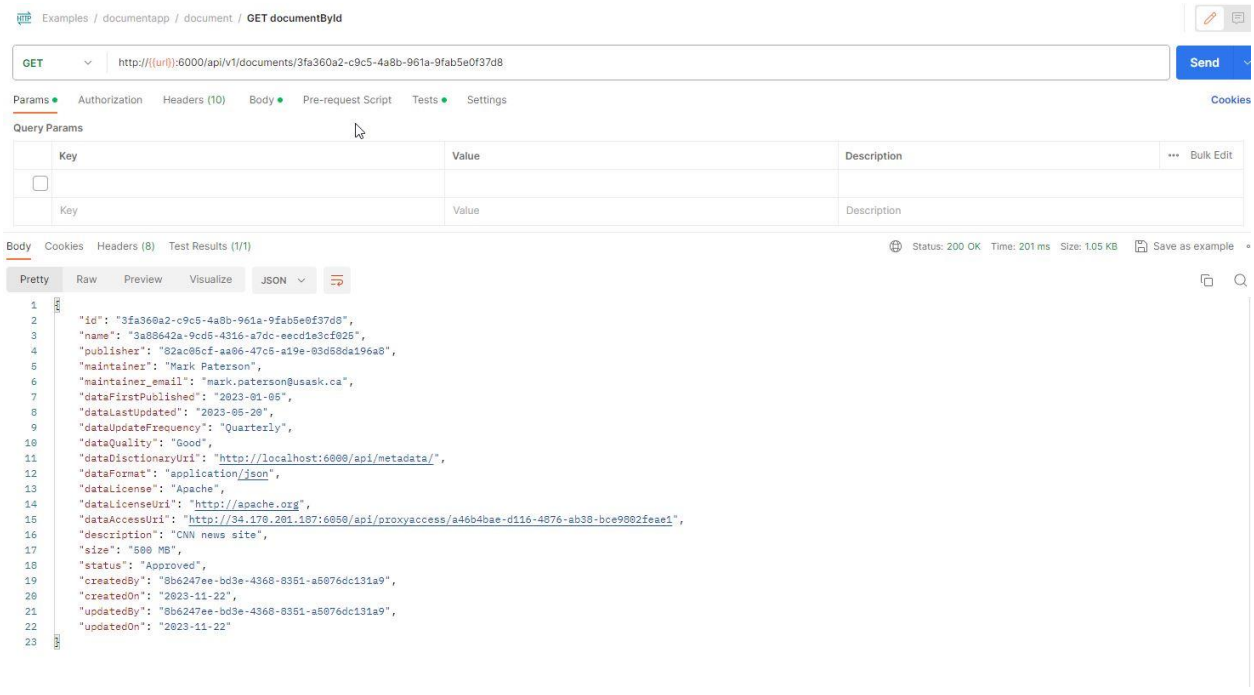


Figure 5.15: Get a request-response on a document resource.

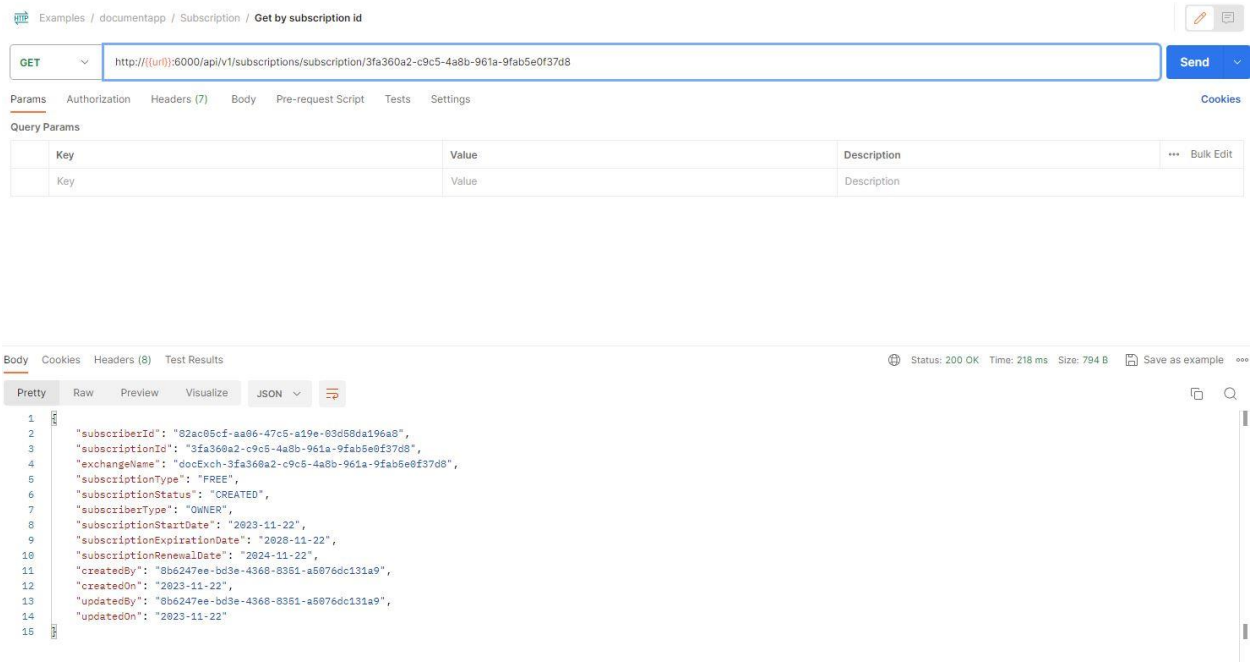


Figure 5.16: Get request and response on subscription.

The author demonstrates the implementation of a data trust architecture that stores data about the data in a secure repository, providing an additional layer of access control and security. The architecture uses a proxy to facilitate access to the data without direct access to the source. By encrypting and recording the hash copy of the document in a private blockchain, one can transparently track the data's provenance and ensure auditability. This also allows the trust to recover the data from the blockchain using the encryption hash.

Additionally, the architecture records all document subscribers, allowing the owner to monitor who has access to their data. This provides a layer of access control at the subscription level. Using a message queue in the document API, the architecture notifies all subscription owners of any document changes via the publisher and subscriber mechanism. This simplifies the automation process for subscribers, depending on the changes made to the document. Hence, through this implementation, the author has achieved research goals one and two outlined in the problem statement.

CHAPTER 6

EXPERIMENTS AND EVALUATIONS

This chapter evaluates the server-side performance of the implementation in Chapter 5 to see the impact of different choices of the components on the overall performance of academic research data sharing through data trust. The performance analysis is done through a series of load tests on the document repository endpoint to assess the performance of different requests and their impacts. The experiments simulate the four core HTTP method calls on the document endpoint to understand the various performance characteristics of the application and identify any possible bottlenecks. Through this chapter, we stand to evaluate the third research goal outlined in our problem statement. The correlation between them is shown in Table 6.1.

Research Goals	**Cases**
2.2.1: Implementation Evaluation	Sections 6.4 and 6.5
2.2.2: Evaluate key metrics	Sections 6.4 and 6.5

Table 6.1: Research Goals and Evaluations

6.1 KEY METRICS

"Throughput" and "Latency" are the key metrics to assess the implementations' performance. Throughput, a measure of the number of requests an application can handle per unit of time, is a crucial indicator of an API's ability to control the load. This is calculated using the formula as shown below.

$$Throughput\ (in\ tps) = T_{rsuccess} / T_{total}$$

Equation 6.1: Throughput

$T_{rsuccess}$ represents the total number of successful requests processed by the API, while T_{total} represents the real real-time consumed. The unit of measure transactions per second (tps) describes the results.

Latency is the time taken for a request to travel from a client to the API server, be processed by the server and send a response to the client. This measure of delay or the time-lapse to complete a request and response cycle helps assess the user experience and the overall effectiveness of the application under load. It is calculated using the formula below.

$$Latency\ (in\ milliseconds) = T_{response} - T_{request}$$

Equation 6.2: Latency

$T_{response}$ is when the first byte of the response is received from the server. $T_{request}$ is the time when the client sends the request to the server. This measure of delay is represented as a unit of milliseconds.

The assessment is carried out on each HTTP method of the primary API endpoints and the secondary components like the database and message queue of choice. Of the secondary component assessments, the blockchain of choice had been left out deliberately, as several works had been done in this area. Still, it is part of the overall component assessment.

6.2 EXPERIMENT SETUP

This section describes the setup and configuration of technologies used in the experiment.

6.2.1 Google Cloud VM Configuration

The experiment is set up with a Google Cloud virtual machine running the Debian Linux operating system. The details of the VM architecture are shown in Table 6.2. The firewall was set to open access to all the listening ports of the application so that the simulation could mimic a distributed access architecture to simulate the real-world scenario. Docker is used as a deployment framework to illustrate the distributed nature of the operation that we want to emulate.

OS and Tools	**Specifications**
Google compute engine	Machine type: e2-medium CPU platform: Intel Broadwell Minimum CPU platform: None Architecture: x86/64 vCPUs: 2CPU Memory: 4GB Custom visible cores: — Display device: Disabled GPUs: None OS: Debian 6.1.66-1 Zone: us-central1-a
Docker	Client: Version: 20.10.24+dfsg1

	API version: 1.41 Go version: go1.19.8 Git commit: 297e128 Built: Thu May 18 08:38:34 2023 OS/Arch: Linux/amd64 Context: default Experimental: true Server: Engine: Version: 20.10.24+dfsg1 API version: 1.41 (minimum version 1.12) Go version: go1.19.8 Git commit: 5d6db84 Built: Thu May 18 08:38:34 2023 OS/Arch: Linux/amd64 Experimental: false Container: Version: 1.6.20~ds1 GitCommit: 1.6.20~ds1-1+b1 runc: Version: 1.1.5+ds1 GitCommit: 1.1.5+ds1-1+b1 Docker-init: Version: 0.19.0

Table 6.2: Google VM and Docker Specifications

6.2.2 PostgreSQL Database Setup

Even though any relational database can be used, PostgreSQL was chosen considering its gain of popularity in the research world and of being an open-source database. The software specification is shown in Table 6.3. The database is deployed using docker-compose, and no alterations are made to the default configurations. Thus, the deployment has the following out-of-the-box configurations, as shown in Table 6.4.

Database	Specifications
PostgreSQL	PostgreSQL 13.12 (Debian 13.12-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit

Table 6.3: PostgreSQL Software Specifications

Parameter	Values	Description
max_connections	100	Sets the maximum number of concurrent connections.
shared_buffers	128MB	Sets the number of shared memory buffers used by the server.
effective_cache_size	4GB	Sets the planner's assumption about the total size of the data caches
maintenance_work_mem	64MB	Sets the maximum memory to be used for maintenance operations.
checkpoint_completion_target	0.5	Time spent flushing dirty buffers during checkpoint, as a fraction of checkpoint interval.
wal_buffers	4MB	Sets the number of disk-page buffers in shared memory for WAL.
default_statistics_target	100	Sets the default statistics target
random_page_cost	4	Sets the planner's estimate of the cost of a non-sequentially fetched disk page.
effective_io_concurrency	1	Number of simultaneous requests that can be handled efficiently by the disk subsystem.
work_mem	4MB	Sets the maximum memory to be used for query workspaces.
huge_pages	try	Use of huge pages on Linux or Windows.
min_wal_size	80MB	Sets the minimum size to shrink the WAL to.
max_wal_size	1GB	Sets the WAL size that triggers a checkpoint.
max_worker_processes	8	Maximum number of concurrent worker processes.

Table 6.4: PostgreSQL Database Configuration

If one needs to tune the database further, one can use an essential tool like PGTune[41], which can provide the configuration recommendations for the PostgreSQL database under different workloads. The data stored in the databases are in the text form of metadata on the data. Since this implementation expects that the research data sharing data trust does not own or manage the data. The researchers maintain it or trust members at their end.

6.2.3 RabbitMQ Configuration

The implementation uses RabbitMQ for its change tracking and notification process. The docker image detail of the RabbitMQ used is shown in Table 6.5. It is a single-node implementation without clustering.

AMQP	**Specifications**
RabbitMQ	RabbitMQ 3.12.6 Erlang 25.3.2.7

Table 6.5: RabbitMQ Specification

As discussed in Chapter 5 of the implementation, an exchange is created for every new document added to the system, with the document ID as part of the exchange name. The document ID is then used as a routing key by the consumers to consume messages from the system.

6.2.4 Hyperledger Fabric Network

This implementation uses The Hyperledger fabric network as an immutable audit component. The implementation uses the Hyperledger fabric test network to test the usability of the private blockchain network in the data trust environment. The specification of the network is shown in Table 6.6.

Hyperledger Fabric Network	**Specifications**
Hyperledger Fabric network	Number of Orgs: 2 Number of peers per Org: 1 Peer version: Peer: Version: v2.5.0 Commit SHA: bd8e248 Go version: go1.20.2

[41] https://pgtune.leopard.in.ua

	OS/Arch: Linux/amd64 Chaincode: Base Docker Label: org. Hyper ledger. fabric Docker Namespace: hyper ledger
Couch database	Version: 3.2.2 Features: [access-ready , partitioned ,pluggable-storage-engines ,reshard , scheduler] Vendor name: The Apache Software Foundation
Orderer consensus method	etcdraft
Channel	documentapp

Table 6.6: Hyperledger Fabric Network Specification

The Hyperledger Fabric test network has two organizations with one peer per organization. The chain code is written using Javascript and deployed under the document app channel. The chain code only stores the document hash value and the document change state for audit purposes.

6.2.5 REST API Endpoint Description

Though the application involves several different API endpoints to carry out various operations, as stated earlier, this experimentation focuses solely on the document endpoint. This endpoint is responsible for recording the metadata about the document and the access. Since this is a crucial part of the application and one of the core pieces, we tested using this endpoint. This endpoint incorporated the database for the storage of data. The Hyperledger Fabric stores the hash value of the data stored in the database so that audibility and provenance can be established for any needs. It uses the RabbitMQ message to create a change notification on the queue and subscribes the owner as a consumer to the queue. This will ensure that the owner can always view or review the message that goes in as part of their change to all the subscribers.

In summary, the author explored Amazon Web Services and Azure and identified Google as the better choice for its simplicity of configuration, maintenance and cost-effectiveness for the intended purpose. The choice of docker is to simulate a distributed architecture framework on a single instance during development

and evaluation to be more cost-effective and efficient. The other subsidiary components are chosen for their prevalence, ease of understanding and usage.

6.3 TOOLS USED FOR EVALUATION

For testing and evaluating different implementation components, the following tools are considered as listed in Table 6.7.

Tools	**Specifications**
Apache JMeter	Version 5.6.2
pg-bench	(PostgreSQL) 13.12 (Debian 13.12-1.pgdg120+1)
perf-test	Version 2.20 jar file

Table 6.7: Testing Tools Specifications

Apache JMeter[42] is a standalone Java program that can be run in any environment to test load, functional behaviour, and performance. It is an industry-standard tool for load-testing web applications. This paper analyzes the performance of the REST API endpoint named "document." Its different listeners allow one to capture and analyze various metrics of the web applications. For this work, the Apache JMeter is run from a Windows 11 laptop to run and grab the different metrics. Its summary and aggregate listeners capture the data used in our later analysis.

PG-Bench is a server-side load-testing tool that facilitates running benchmarks. It allows one to create and use a sample database for performance benchmarking. The program iterates a sequence of SQL commands on the sample database to provide vital statistical performance information on the PostgreSQL database. This is one of the DBA tool chests used to evaluate and tune a Postgres database. With this, one can start a baseline and adjust different database instance parameters to see the performance differences.

Perf-test is a Java-based client tool that allows one to evaluate the throughput of the RabbitMQ under different simulated workloads. To run, it uses the AMQP protocol to connect to the RabbitMQ server listening on port 5672 and generate different workloads. The thesis uses this tool to identify the throughput and latency of the message queue under other publisher and consumer settings. This tool is run on the same Windows 11 OS for various tests.

[42] Apache JMeter - Apache JMeter™

6.4 CASES OF EVALUATION

The evaluation uses the below-defined set of test cases to assess the performance efficiency of the application and its components.

6.4.1 Performance of REST API Endpoint

This experiment compartmentalizes the REST API endpoint testing into three categories to identify the role and impact of each component (i.e., database, blockchain and message queue) in the application's overall performance. For testing, the experiment cloned the standard endpoint and added the modifications so that the differences in changes were specific to the case in consideration. The testing process will carry out a series of iterations on HTTP methods of GET, POST, PUT and DELETE. The Apache JMeter, installed on Windows 11, simulates load and gathers metrics that will be used later in evaluating the implementation. The process flow is illustrated in Figure 6.1. The test is run on different concurrency rates starting from 100 to 1000 concurrent threads. No think time is used for the evaluation runs. Each iteration is repeated five times, and the average is used for the evaluation.

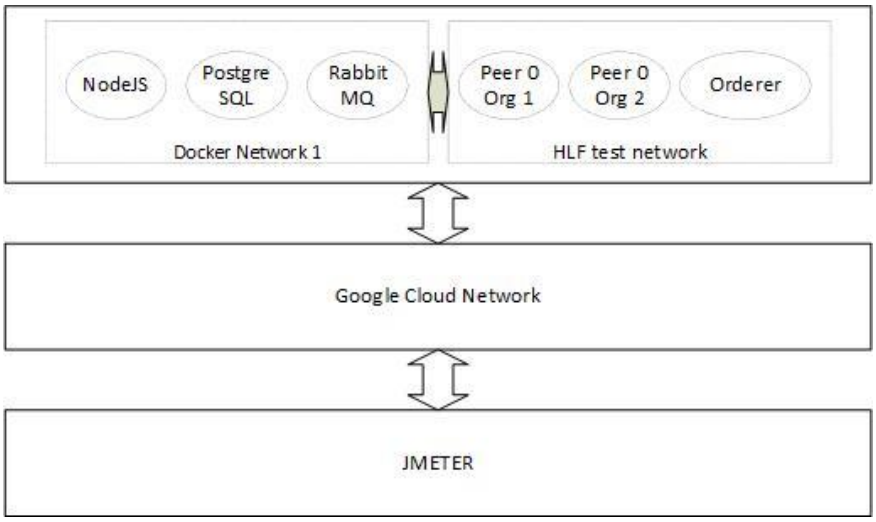


Figure 6.1: API Test Framework

6.4.1.1 CASE 1: STANDARD DOCUMENT ENDPOINT

This is the standard endpoint an application user would endure in their interaction. It incorporates the components of the PostgreSQL database for data storage, Hyperledger Fabric as an immutable audit point to store the hash value of the data, and RabbitMQ to publish a message on the state of data. Through this test case, the analysis stands to identify the performance impact of using the three core components together in one application.

6.4.1.2 CASE 2: MODIFIED DOCUMENT ENDPOINT WITH NO BLOCKCHAIN

For this test case, the experiment clones the standard document endpoint and modifies the controller and services to remove the code segment that handles the interfaces with the blockchain. The experiment evaluates the application's performance with RabbitMQ and without blockchain integration in this test case.

6.4.1.3 CASE 3: MODIFIED DOCUMENT ENDPOINT WITH NO BLOCKCHAIN AND RABBITMQ

This section of the experiment case repeats the same steps as the above blockchain case, in addition to removing the integration of the message queue component. Evaluation of this test case will assess the impact on the application's performance without the blockchain and RabbitMQ.

6.4.2 Performance Benchmark on PostgreSQL

This part of the evaluation will assess the performance of the secondary component, the PostgreSQL database, through the pgbench utility. Pgbench is run on the database running on the docker-compose container and a standalone installation on Google Cloud VM. Since the database is set to handle a maximum of 100 concurrent users, the test is carried out from 10 to 100 simultaneous users with a step-up of 10 with two transaction limits of 1000 transactions per client and 10000 transactions per client. Trying the 100000 transactions, the docker container PostgreSQL database started to crash consistently after 30 concurrent clients. The outcome of each run is the total processed transactions, average latency, transactions per second with connection, and transactions per second without connection. The latency and throughput data are then used for the comparison.

6.4.3 Performance of RabbitMQ

Performance data on RabbitMQ is collected using the perf-test tool. The tool is run as a Java command line interface on Windows 11 operating systems. The tool uses the AMQP protocol to interact with the RabbitMQ server. The first case of data sets is obtained by running a test against the RabbitMQ running on the docker container of the API network. All the containers remained operational in the docker network during this test, as shown in Figure 6.2. A similar data set is obtained by running the same tests on the stand-alone installation of the RabbitMQ server running on a Google VM instance, Figure 6.3. In both cases, the iterations are carried out in a one-to-one mapping of the producer-to-consumer ratio, starting from 10 to 100 in increments of 10. Each test is run for 300 seconds. The outcome of each run rendered the information on the number of messages sent per second, the number of messages consumed per second and

consumer latency min, median, seventy-fifth, ninety-fifth and ninety-nine percentiles represented in microseconds.

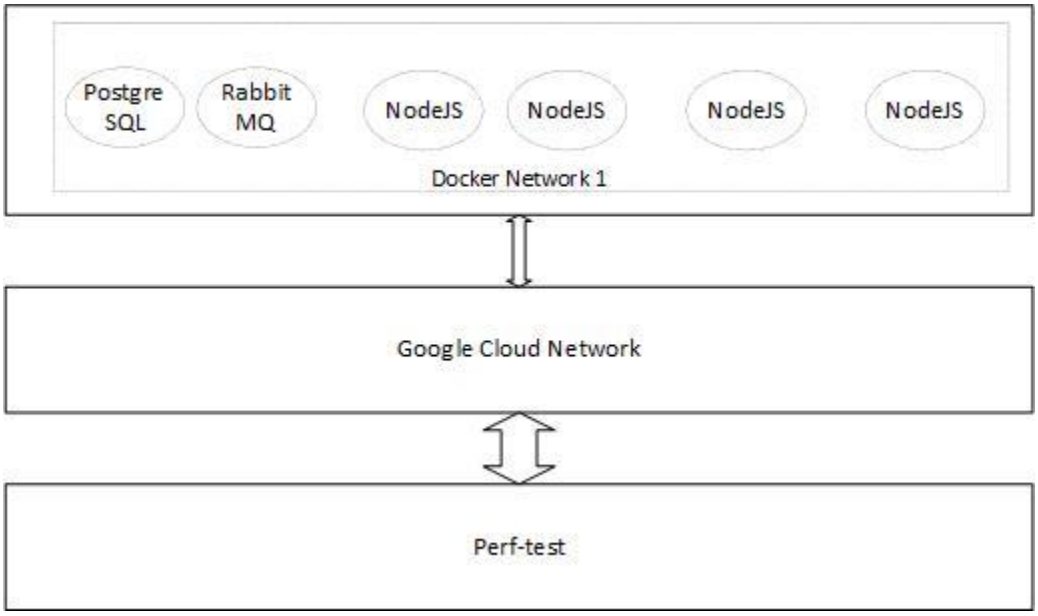


Figure 6.2: RabbitMQ test on Docker

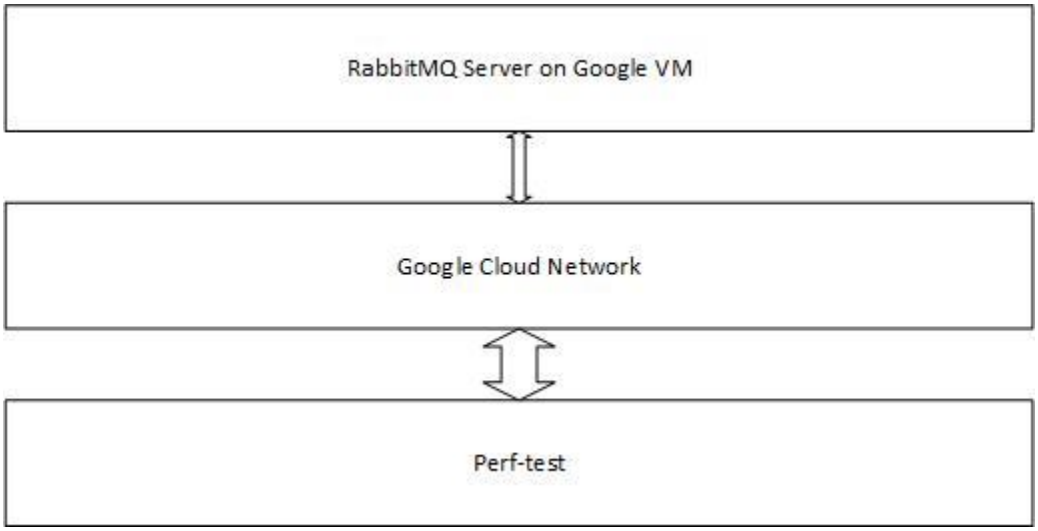


Figure 6.3: RabbitMQ test on Standalone

6.5 RESULT ANALYSIS

This section discusses the different results sets obtained by performing the previously discussed test cases.

6.5.1 Performance of REST API Endpoints

As discussed above in the test case section 6.4.1, the tests are run on HTTP method calls of GET, POST, PUT, and DELETE on the three endpoints discussed in subsections 6.4.1.1 to 6.4.1.3 using the JMeter. The GET, PUT, and DELETE tests are done with a focus on single record management. The aggregate and summary data are gathered and plotted for our analysis below. The following subsections will analyze the critical metrics under each HTTP method call.

6.5.1.1 POST METHOD

The throughput results revealed on the POST iteration, as per Figure 6.4, show a significant performance difference between the endpoint that used the database, blockchain and message queue (i.e. standard endpoint) and that of the other two. On comparing the performance of each endpoint in relevance to the total performance of each set, the common endpoint performed at 20 percent to the whole performance, while the endpoint without the blockchain performed at 35 percent and that of the one without the blockchain and RabbitMQ performed at 44 percent. This shows that the blockchain contributed 15 percent performance differences to the application post method, which is predominantly a write-oriented operation.

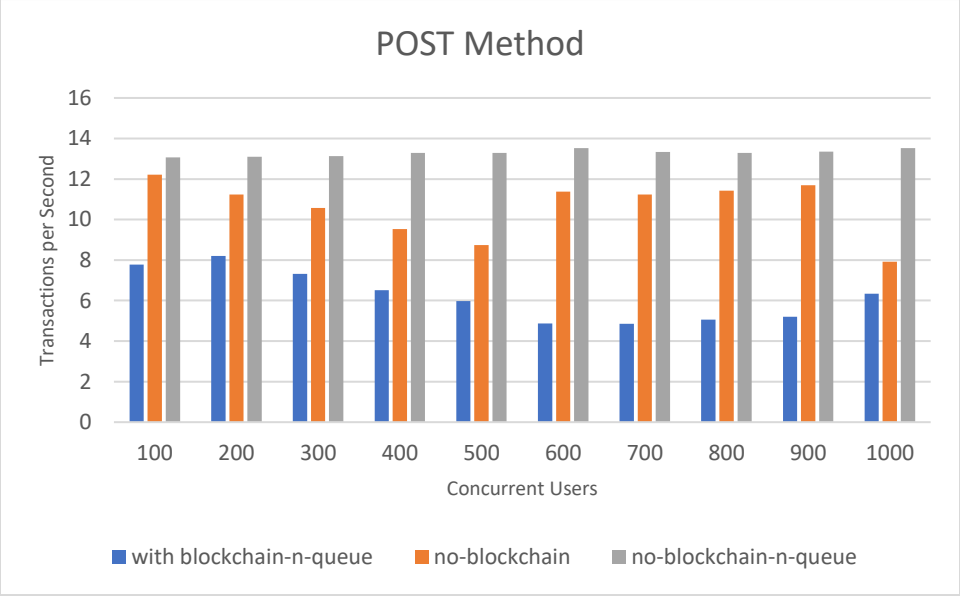


Figure 6.4: POST Method Throughput

The latency comparison charts illustrated below show the latency of the three application endpoints under different concurrencies. To display the differences, the first chart in Figure 6.5 uses the Average latency values, while Figure 6.6 uses the Median latency values. Though the average latency is the flattened depiction of all the ups and downs masking the real problem, the median will be a clear indicator of the 50th

percentile operation of the application. The standard endpoint endures higher latency than the other two. The common endpoint in this analysis consumes 43 percent of the total latencies of all three sets, while the endpoint without the blockchain but with a message queue had 32 percent. In comparison, the one without the blockchain and message queue endured the minimum of all three, with 25 percent latency on average.

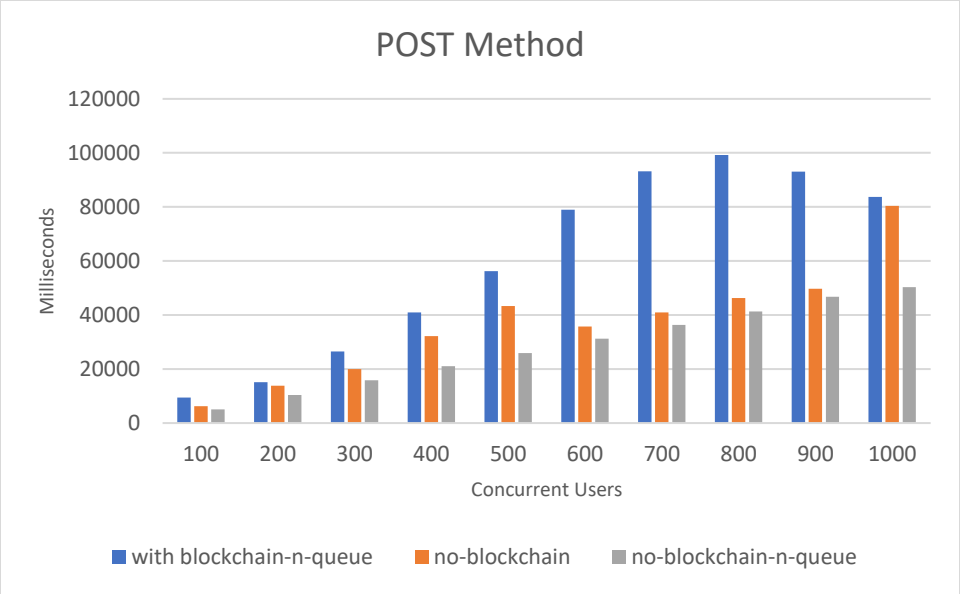


Figure 6.5: POST Method Average Latency

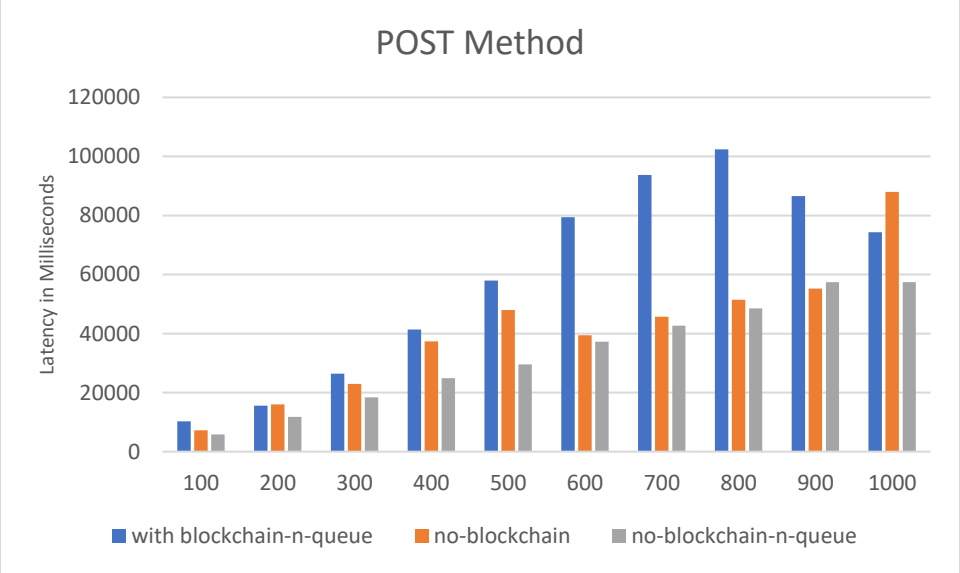


Figure 6.6: Post-Method Median Latency

As in Figure 6.7, a high error rate on standard endpoint test results is also observed. Most of them are attributed to blockchain peer consent failure. The connection failure errors are also marked on the three endpoints; this can be attributed to the server load or network responses. As the error rate increases, the latency drops because there was no delay in getting the HTTP error instead of waiting for the consent algorithm to process and send failure as a return, resulting in some wait time. This is evident from the blockchain with message queue graphs after the 800-sample set, in correlation to the error graph below.

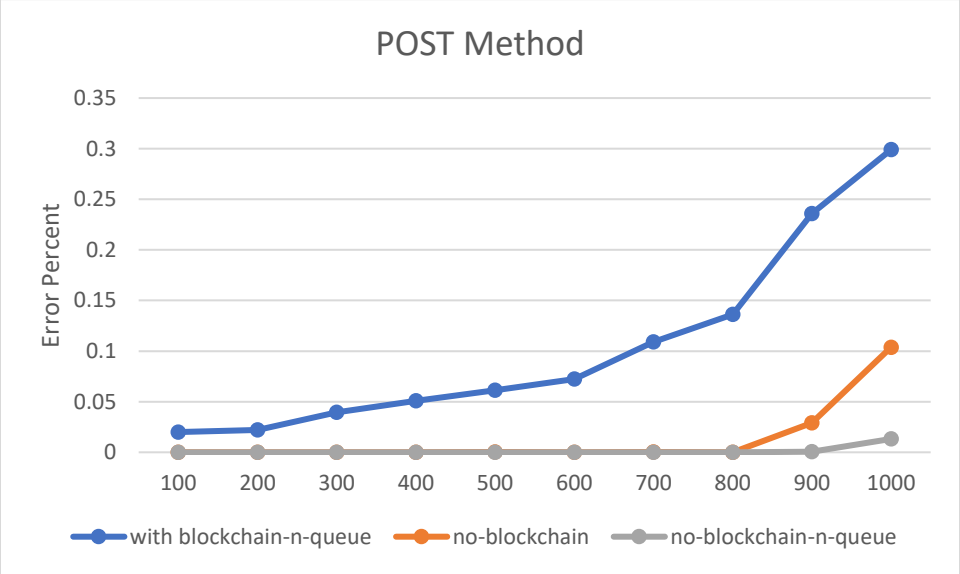


Figure 6.7: POST Method Error Percentage

6.5.1.2 PUT METHOD

The PUT method of the endpoints involved reading and writing as part of the operation. The throughput results shown in Figure 6.8 reveal that the standard endpoint throughput is at 25 percentiles compared to the total throughput of all three endpoints. The endpoint without the blockchain but message queue performed at the 35 percentile, and the one without the blockchain and message queue performed at the 39 percentile.

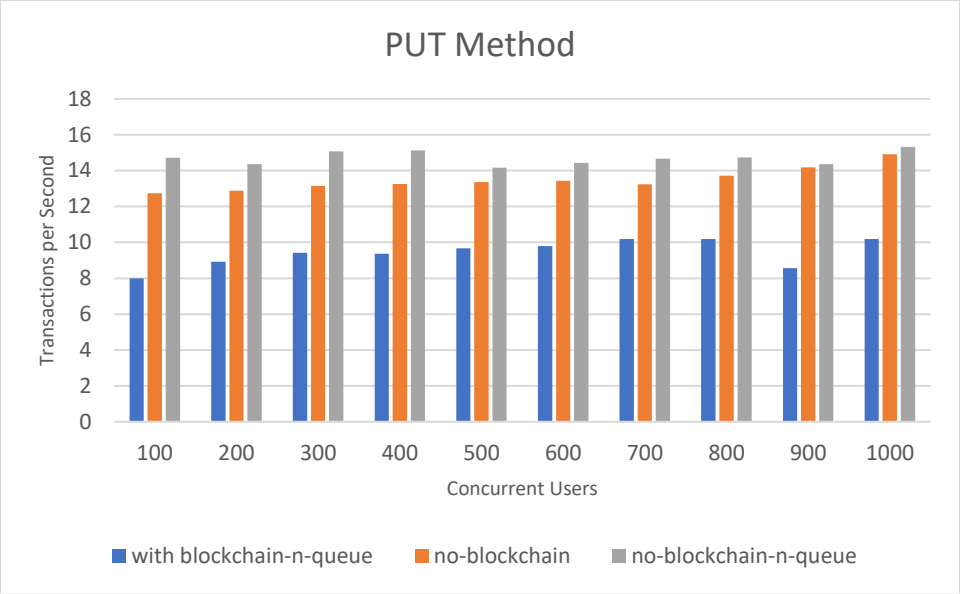


Figure 6.8: PUT Method Throughput

Regarding throughput, the latency of the endpoints shown in Figure 6.9 and Figure 6.10 shows that the standard endpoint endured higher latency than the other two endpoints without the blockchain and/or RabbitMQ. The average performance of each endpoint reveals that the common endpoint took 41 percentile degradation compared to the other endpoints. The endpoint without the blockchain but with a message queue had 30 percentile degradation, and the one without the message queue and blockchain faced 28 percentile latency. This shows that the endpoint with blockchain, due to the nature of the internal consensus, is likely to have a higher latency time amended to the application performance.

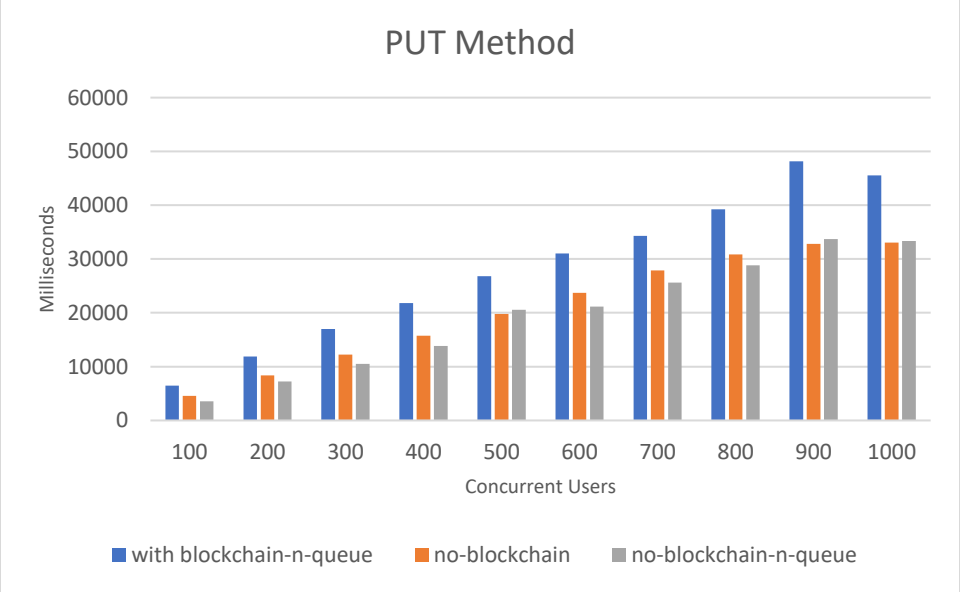


Figure 6.9: PUT Method Average Latency

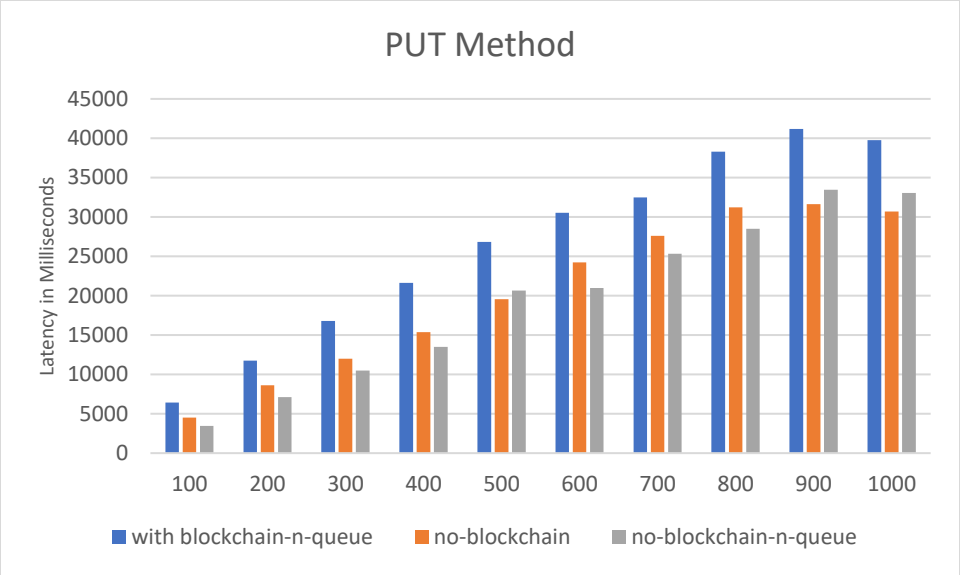


Figure 6.10: PUT Method Median Latency

It is also evident that not all the transactions are completed successfully. This is evident from the error percentage of each application endpoint during the different iteration points, Figure 6.11. The error rate is exceptionally high on the standard endpoint compared to the others. This is because, during the testing, the peers cannot ensure consensus on time for the transaction to be committed in the blockchain. Part of the error rate can also be attributed to the server workload and the resultant impact on the response time.

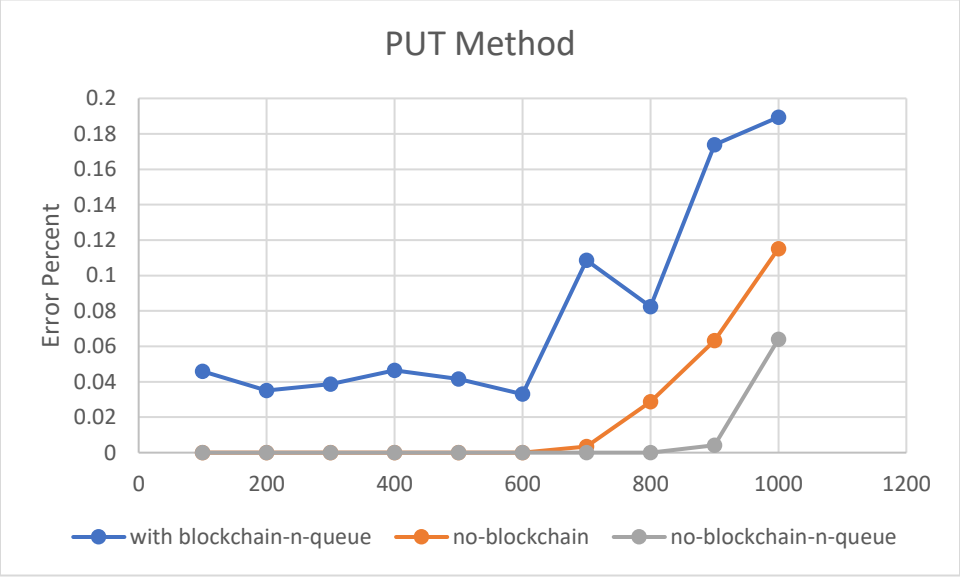


Figure 6.11: PUT Method Error Percentage

6.5.1.3 DELETE METHOD

With the DELETE, like all the POST and PUT methods, the process endures a write to the blockchain to remove the record from the repository and the exchange from the message queue in the standard endpoint operation. As seen in Figure 6.12, the throughput performance of the common endpoint is deficient compared to the ones without the blockchain in the framework. Also, it can be observed that the throughput of the service differs on the one without the blockchain and the other without the blockchain, and the message queue also has some visible differences. This is because there is a time that gets endured when deleting the exchange on the message queue, which contributes to the performance. Though this is not as significant as the blockchain, it is visible.

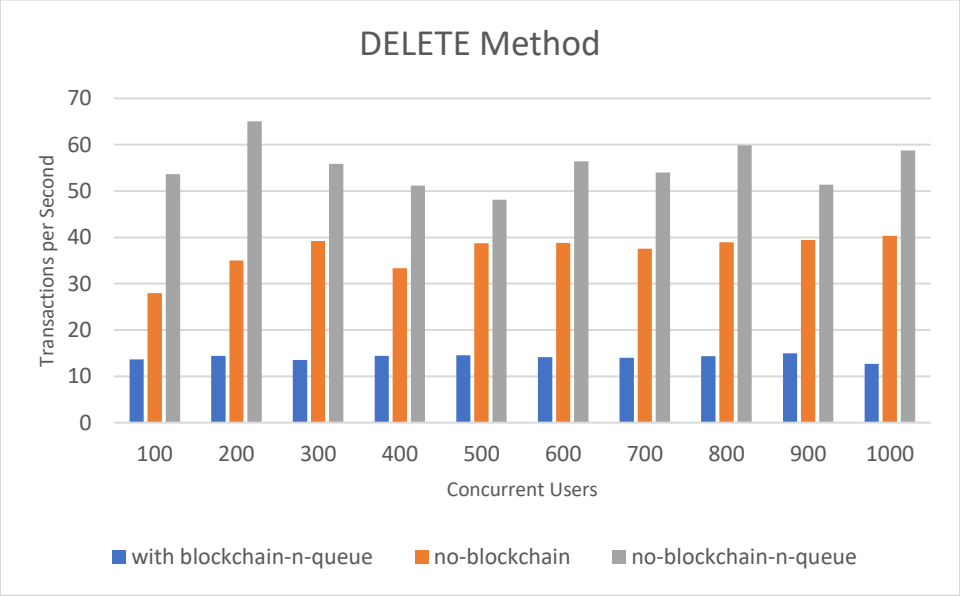


Figure 6.12: Delete Method Throughput

As observed on the earlier method calls, the latency of the standard endpoint is high compared to the others without the blockchain component. This is visible in Figure 6.13 and Figure 6.14. As the concurrency increases, the error rate increases due to the high latency on the standard endpoint, as seen in Figure 6.15.

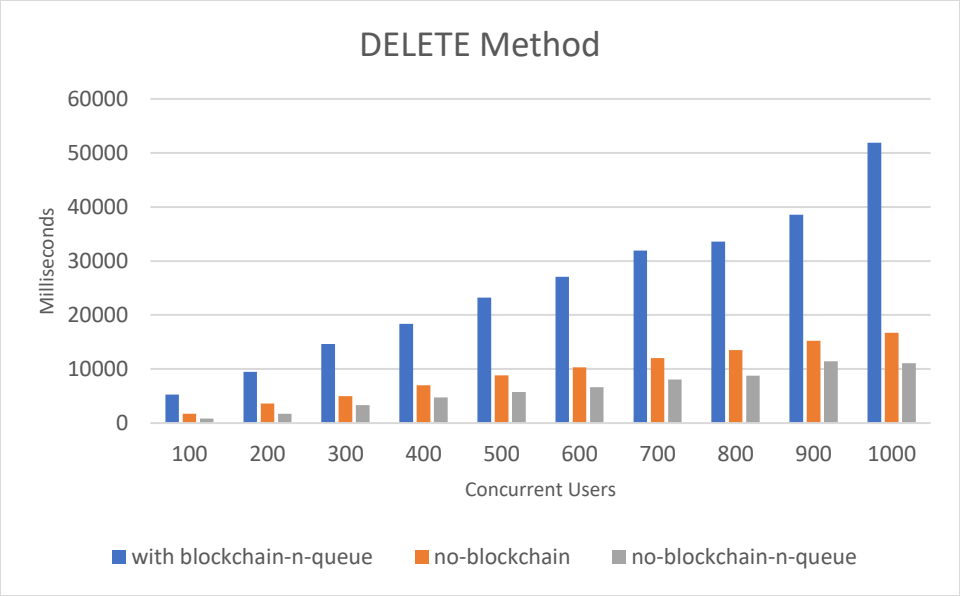


Figure 6.13: DELETE Method Average Latency

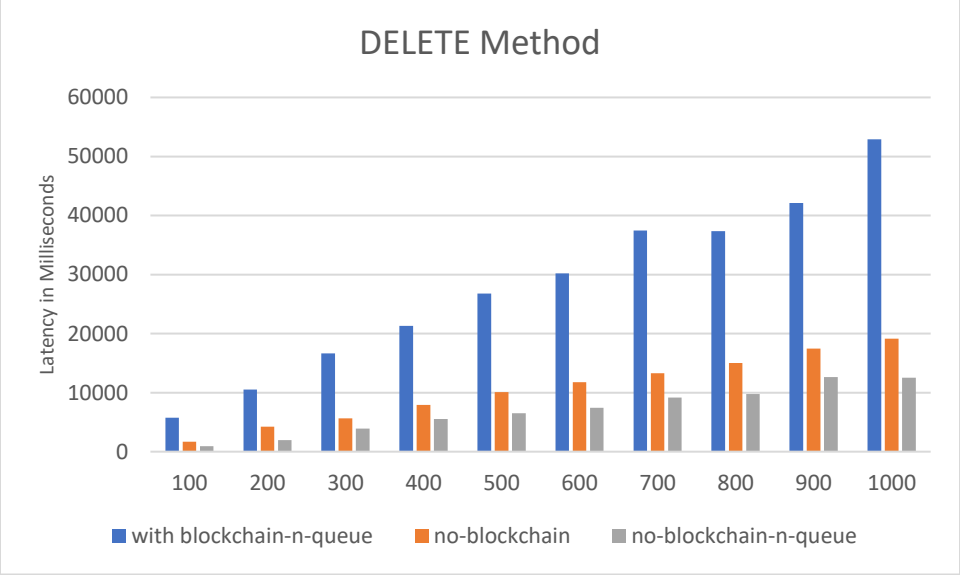


Figure 6.14: Delete Method Median Latency

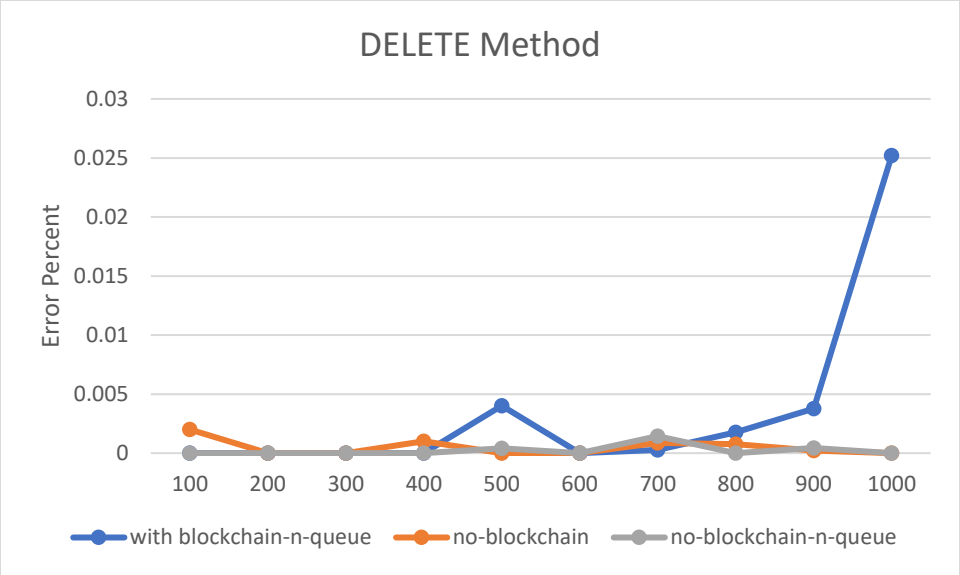


Figure 6.15: DELETE Method Error Percentage

6.5.1.4 GET METHOD

On the contrary, the GET method seems to have the closest throughput across all three endpoints testing, Figure 6.16. This can be attributed to the fact that the GET method is a highly READ-oriented operation. Also, in the GET method, the process does not read the data from the blockchain for its operation. It relies on the database repository for the operation to return the response. Similarly, the latency of the three endpoints also has the closest latency timings, as per Figure 6.17 and Figure 6.18. The error spike in Figure 6.19 for the no-blockchain-n-queue endpoint can be attributed to an anomaly due to network delays or performance.

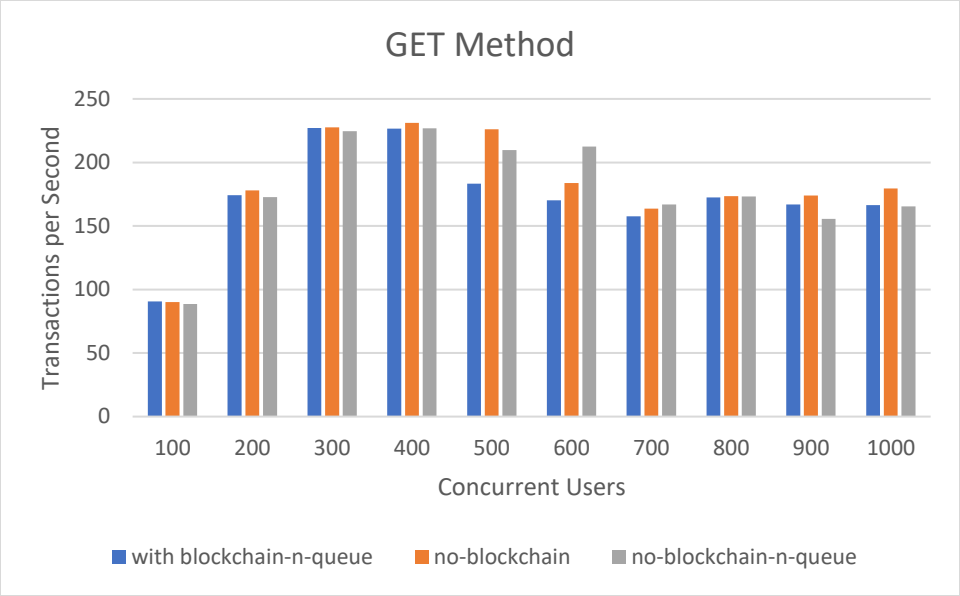


Figure 6.16: GET Method Throughput

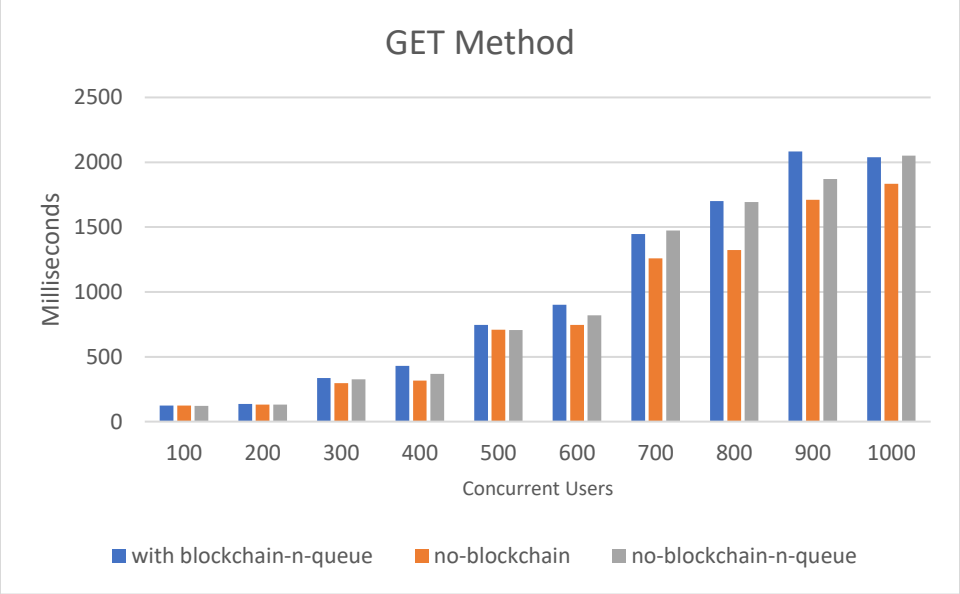


Figure 6.17: GET Method Average Latency

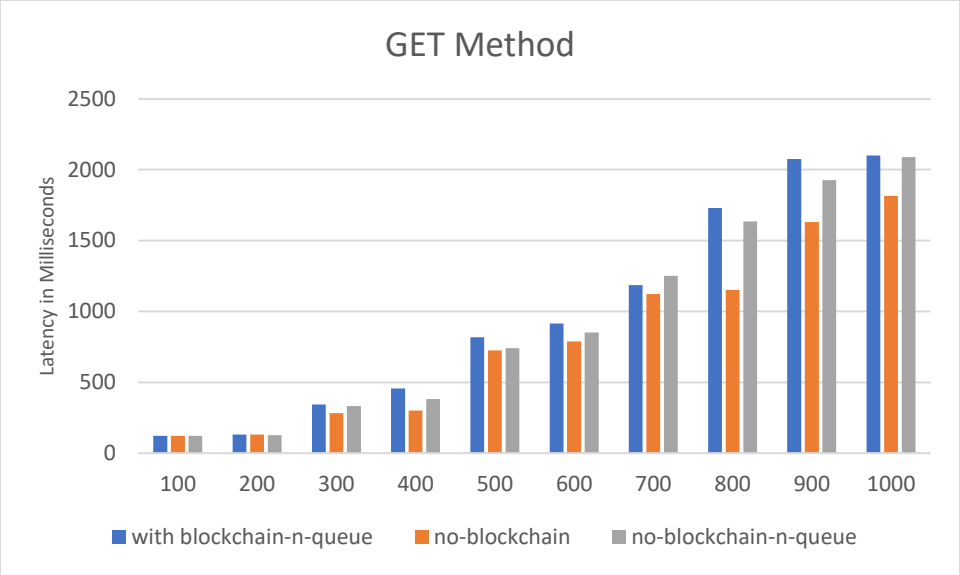


Figure 6.18: GET Method Median Latency

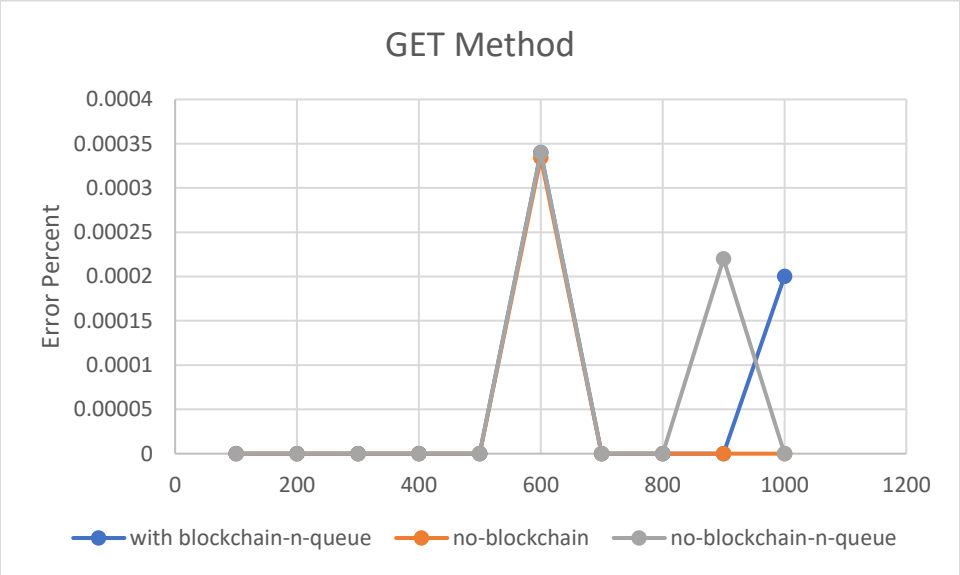


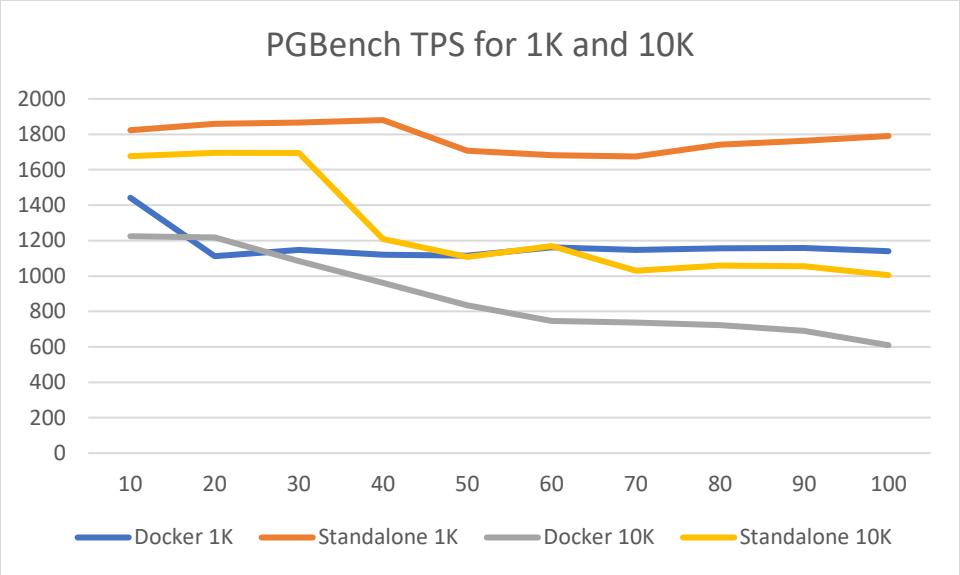
Figure 6.19: GET Method Error Percentage

6.5.2 Performance Benchmark on PostgreSQL

As mentioned in section 6.4.2, the PGbench utility is run on the database node for 1000 and 10000 transactions, and the resulting data are plotted below based on their throughput and latency. No tuning is done in either instance, as the intention is to analyze the out-of-the-box deployment capabilities of the database.

Figure 6.20 and Figure 6.21 illustrate the throughput of the operations when performed with different concurrencies of users ranging from 10 to 100. Per observation, there is a significant performance difference between the containerized database and the one running as a standalone server. Thus, attributing these differences to the architecture, the containerized database competes with the other application containers for resources. At the same time, the standalone reaps the benefit of using the VM's dedicated resources.

On analyzing the percentile of the performance between the docker and standalone installation of PostgreSQL databases, the 1K run shows that, on average, docker was reaping a throughput of 39 percent, compared to the standalone doing at 60 percent. On the 10K runs, the docker was able to generate 40 percent throughput in comparison to the standalone, reaping 59 percent. Hence, in both the runs of 1K and 10K, the average differences are close to 20 percent and 18 percent. This can purely be the result of resource availability.



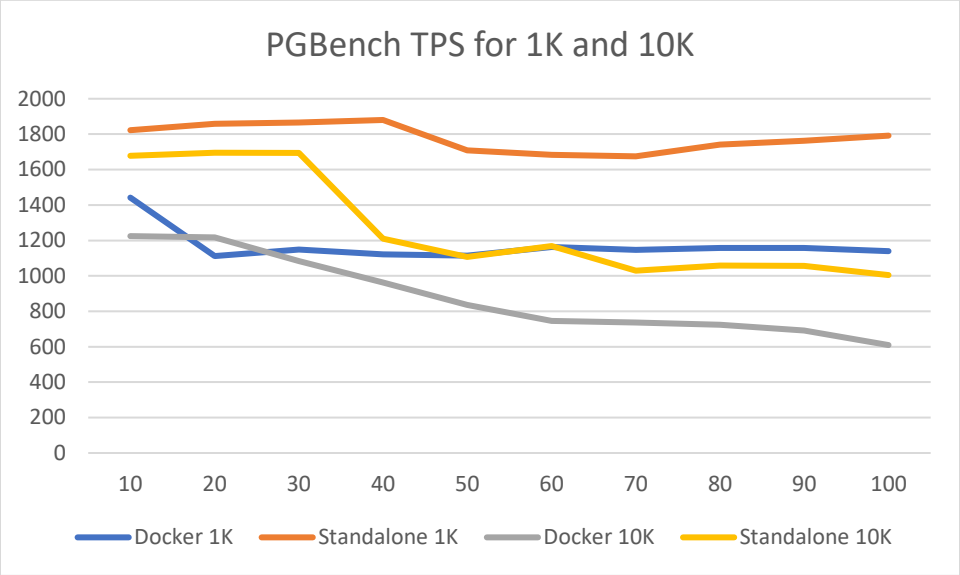


Figure 6.20: PGBench Throughput Chart between Docker vs Standalone.

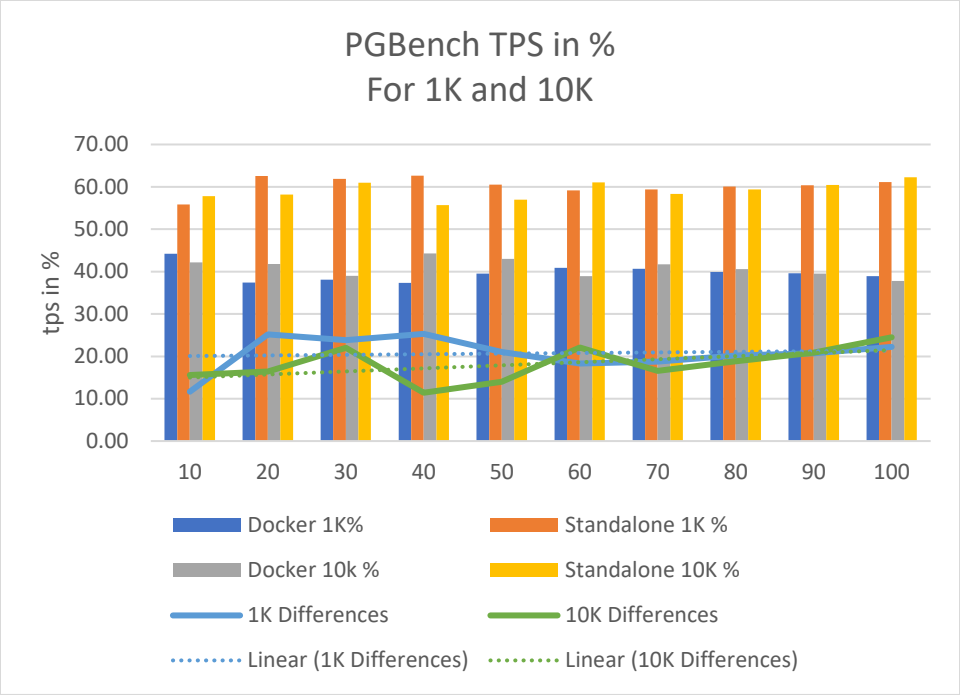


Figure 6.21: PGBench Throughput Differential Chart between Docker vs Standalone.

Regarding the latency of 1K and 10K tests, in correlation to the throughput, the latency is noticeably high for the docker container compared to the standalone, Figure 6.22, Figure 6.23. This is for the same reasons that were mentioned earlier. It can be observed that in this experiment, the latency percentile ratios are the reverse of the throughput ratios. The 1K run of docker to standalone rendered 60:39 percentile, while the 10K rendered 59:40 percentile.

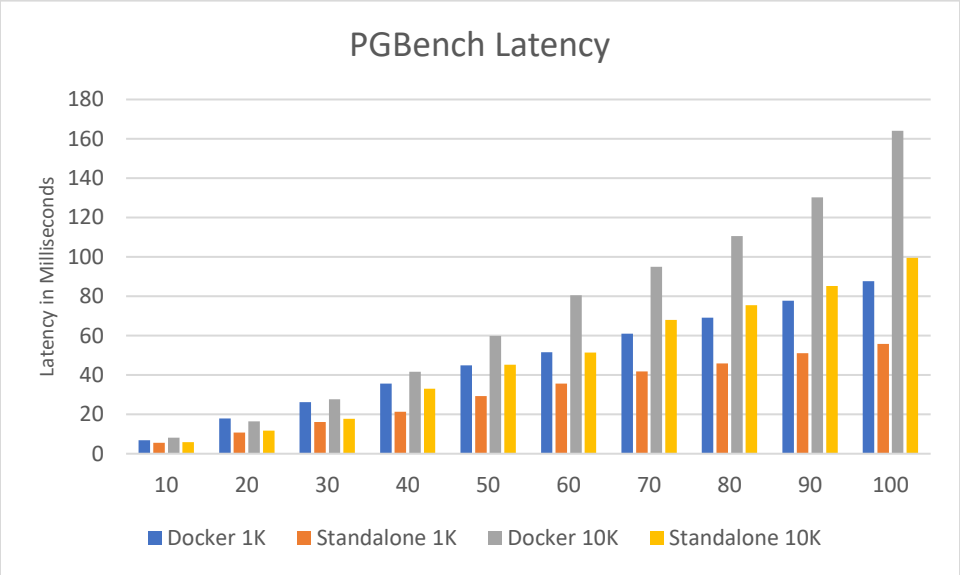


Figure 6.22: PGBench Latency between Docker vs Standalone.

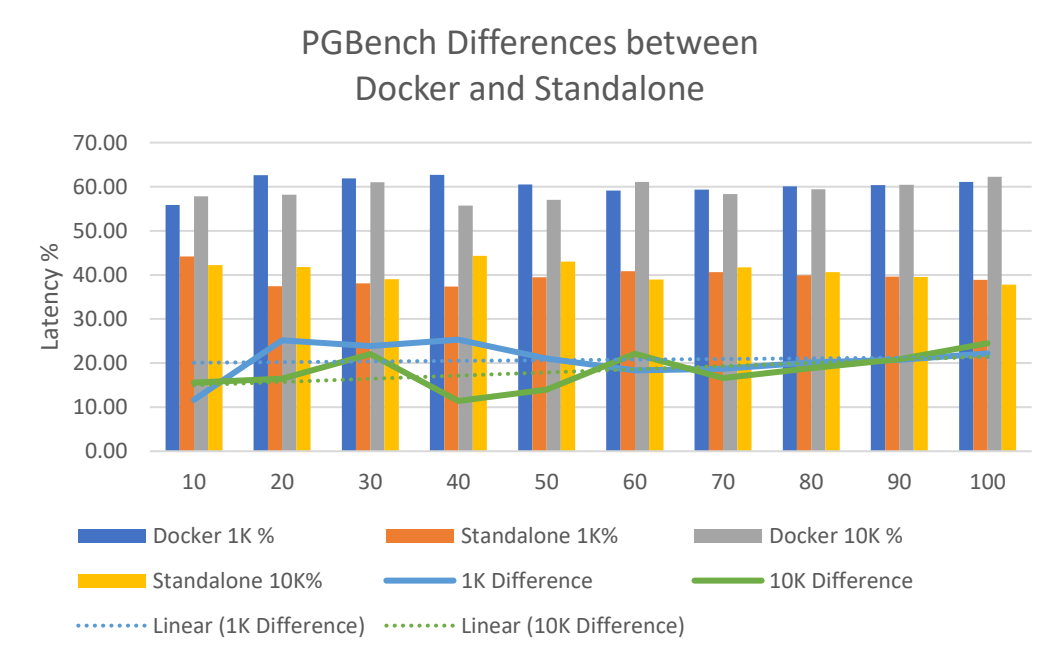


Figure 6.23: PGBench Latency Differential Chart between Docker vs Standalone

6.5.3 Performance of RabbitMQ

Using the perf-test tool on the RabbitMQ installation on docker identified the performance of the queue depletes beyond 50 high publisher and consumer concurrency operations. For these reasons, the comparison

charts below only depict the producer-consumer number limits, which are limited to 50. Based on this evaluation, it is evident that there is a 30 percent performance difference between the producers, with the standalone installation of the message queue outperforming. Similarly, on the consumer side of the throughput analysis, it was clear that the standalone install outperformed the docker with an average of 50 messages per second. This is evident from the charts shown in Figure 6.24 and Figure 6.25.

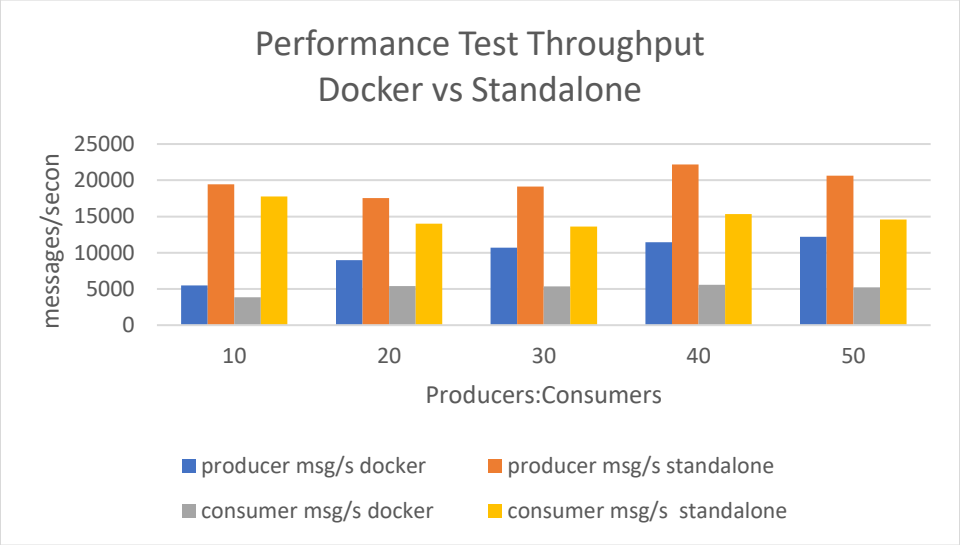


Figure 6.24: RabbitMQ Performance Throughput

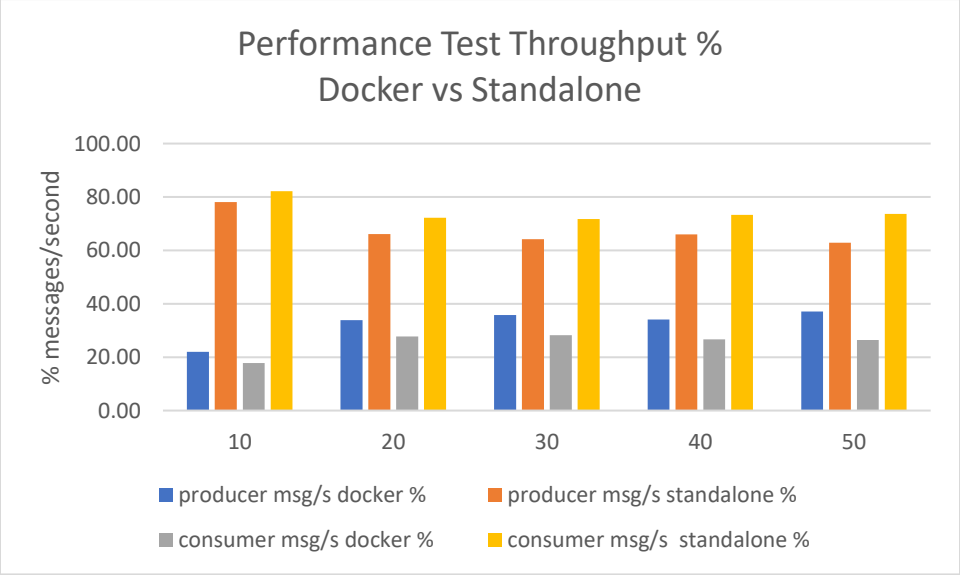


Figure 6.25: RabbitMQ Performance Throughput in Percentage

Perf-test, on its analysis, renders only the latency information for the consumer actions. Upon using this information on the different concurrency analyses between the docker and standalone, it can be observed that the throughput had 27 percent lower latency than the docker. This can be seen in Figure 6.26 and Figure 6.27. The blog post RabbitMQ 3.10 performance improvements illustrates the rabbitmq capabilities in a clustered environment (Kuratczyk 2022).

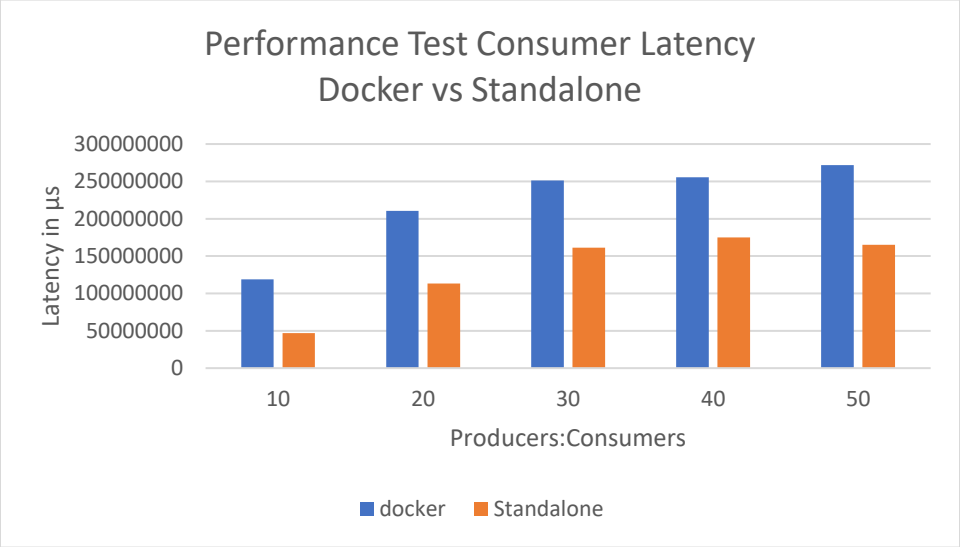


Figure 6.26: RabbitMQ Consumer Latency

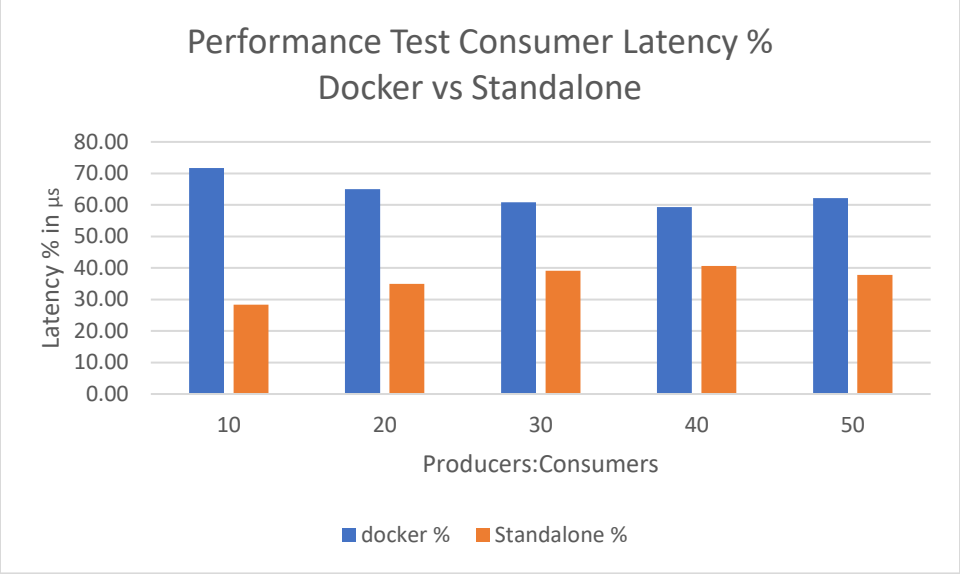


Figure 6.27: RabbitMQ Consumer Latency %

6.6 SUMMARY

This chapter delves into the server-side performance evaluation of an implementation for academic research data sharing through data trust. The load tests on the document repository endpoint are designed to assess the impact of different components on the system's overall performance and identify any possible bottlenecks. The two key metrics, throughput and latency, are used to assess the performance of the implementation. Throughput measures the number of requests an application can handle per unit of time, while latency measures the time taken for a request to travel from a client to the API server and back. Experiments are conducted on the HTTP methods of the primary API endpoints and several secondary components, including the database and message queue. The experiment setup used a Google Cloud VM running the Debian Linux operating system, with Docker as the deployment framework. It also deployed a PostgreSQL database using docker-compose, considering its open-source nature and popularity in the research world. The experiment setup and configuration are detailed in this chapter, including the specifications of the Google Cloud VM and the PostgreSQL database. This chapter offers insightful information on the implementation evaluation, which can help make informed decisions for academic research data sharing through data trust, hence achieving the research goals one and two listed in Chapter two.

CHAPTER 7

CONCLUSIONS, CONTRIBUTION, AND FUTURE WORK

7.1 CONCLUSION AND CONTRIBUTION

At the time of this research work, there was no cohesive research data-sharing framework in the research community. To mitigate this shortcoming, the author set up this data-sharing framework on the backbone of the Federated Data Trust model using a decentralized architecture. There, it highlighted that the data remains in the custody of the data owners, and the metadata on the data is managed through the data trust with the access point to actual data at the data owner site. Since the elaborative framework is beyond the scope of this thesis, the work is set to focus on the document repository, which maintains the metadata on the data. Also, it is worth mentioning that all access is done through API calls. Hence, the focus is to ensure that the core access methods to disseminate the data are operational effectively. The research goal is attained by introducing a change notification system with a message queue to notify members of the data change on the provider end.

Chapters 1 and 2 give a brief introduction and the state of the problem through a set of research goals. Chapter 3 explores the origin and evolution of the Data Trust, blockchain and message queuing technologies. It was clear that data trust was one of the best agreed-upon solutions to bring the public and private data holders to share the data for the greater public good. However, there needs to be a cohesive agreement regarding implementation; it is a work in progress. The publications of "Data Trusts: Ethics, Architecture, and Governance for Trustworthy Data Stewardship" (K. O'Hara 2019), a Canadian study (Paprica, et al. 2020), and "Data Protection By Design: Building the Foundations of Trustworthy Data Sharing" (Stalla-Bourdillon, et al. 2020) provide more comprehensive detail on the components that a data trust must have. This thesis drew its core design features from these papers. While identifying different blockchain technologies that can be used, the blockchain section of this chapter highlights the role of Hyperledger Fabric private blockchain technology, which the author intends to use in this thesis. Though various forms of it were explored with the message queue, the final choice was to go with Advance Message Queue technology using RabbitMQ.

Chapter 4 explores the design and architecture of the vision of a Data Trust system with change tracking that will facilitate a trustworthy sharing of research data in a transparent manner. In this chapter, the author proposes a vision of Data Trust architecture and identifies the different management principles contributing to the complete system framework. The blockchain is used as an immutable storage to track the changes to the data about the data. The notification hub acts as a change notification system using the message queue. As noted initially, the data remains in the owner's custody. A proxy access point is set on the server side to facilitate access to the data upon valid access control checks.

Chapter 5 details implementing a decentralized data trust framework built on NodeJS to handle the user interfaces, incorporating PostgreSQL for hosting different repository databases, RabbitMQ for change notification, and Hyperledger fabric for immutable audit and provenance tracking. With this model, the owner is added as a subscriber to the document change notification system when new data is added. Upon creating a successful account, the trust makes a proxy endpoint to access the source data at the owners' site. The owner and subscribers are notified when the state change on the data is updated. The metadata record is encrypted during this operation, and the hash is stored in the blockchain network. This provided the means to track the data provenance and auditing as stated in research goal two. Using data trust with the federated trust model, the thesis establishes the governance goal.

Chapter 6, an elaborate test using the different endpoints with different core secondary components, shows that the endpoints referred to as standard or common endpoints were less performant than those without the blockchain. This shows cases where the members need consent to commit the block, which is attributed to the overall performance of chains in the event of writing with blockchain. This delay also contributed to the number of failure rates in the testing. The performance testing also identified that using different deployments as docker images on a single VM has limitations, as the docker containers had to compete for resources. This was evident when the performance testing was done with PostgreSQL and RabbitMQ. It became more pronounced with the RabbitMQ testing, where it continued to crash the container when the concurrent producer and consumer were increased to 60 or above. The standalone install of the RabbitMQ server breezed through the operation.

Through these tests, the overall conclusion of this thesis is that in a data trust framework, immutable data storage is critical. The commonly known area of this storage is in the blockchain. Hence, a good choice of blockchain framework can resolve this limitation. With the PostgreSQL Database, no extra tuning was

done. As a result, it only allowed 100 concurrent users. One can reap better performance beyond the application's needs by hosting it as a hosted or standalone service and tuning it with some benchmark points. Regarding the message queue, it is recommended that the RabbitMQ server be hosted as a standalone or clustered server in its VM(s). This will outperform the one hosted as a docker image in a docker-compose.

7.2 FUTURE WORK

Data Trust can be entailed in a vast array of applications, opening the venue to share the data more trustfully. With the above implementation and assessment, the thesis proved that the federated data trust repository can sustain and operate under a nominal performance load. One can unlock its full potential with some of the suggested future work below.

1. Cross-Cloud solution: Most present-day applications in the cloud rely on single-cloud solutions and shift towards cross-cloud solutions. Future work should consider the decoupled deployment of this framework on a cross-cloud infrastructure.
2. Data provenance dashboards: An immense value-added proposition will be a visual aspect of the data provenance in the Data Trust.
3. Policy governance: There are no clear guiding principles to establish policy governance. Though it may have been deliberately left out, future work can provide guiding templates to define the principles.
4. State of trust at disembarkation: All past and present works on Data trust can be seen to have steps and methodologies to onboard members and data to the trust and its management. No works or references can be identified for someone disembarking the trust. Though this can be a simple topic on the surface, many intricacies need to be addressed,
    a. Like what happens if the data producer leaves?
    b. What happens to those who have subscribed to the data when the producer leaves?
    c. What is the state of the data passed out when a consumer leaves?
    d. What is the role of the Data Trust when it comes to reclamation?
5. Trust collaboration: As discussed above, data trust provides a means to collaborate data among trust members. This opens the next tier of questions: what would be the model when the data needs to be shared between one or more data trusts? What would the policy or procedure that governs them look like?

6. Digital Twins: Digital Twins are garnering popularity with data distributions. Hence, it can be used as a data dissemination point for those aggregate data.
7. Enforcement of proxy access availability: This thesis did not focus on the assurance of source data link being available for proxy access. Hence, future work can consider measures to ensure this access availability.
8. Ranking on quality and access: A ranking system can be established to facilitate the quality and accessibility of data.
9. ETags: Using this to track the change on the source. Integration of this process with the notification queue can enhance the data change tracking and notification system.
10. Client proxy: Future work can expand by using a client proxy that interacts with the server proxy to access data through some access controls.
11. Tuning of blockchain: Our work shows that the blockchain could be the bottleneck to the performance of the Data Trust on a high transaction system. Thus, exploring other fast private blockchains like Solana and assessing the performance gains or losses is recommended.
12. Queue as intermediatory point: Given the better performance timing with RabbitMQ, one can leverage the persistent message queue to put the hash in the queue and set up a consumer process to consume the queue and commit the change in the blockchain of choice at its own pace. This can alleviate the transaction failures on the application front end due to consent delays or failures.
13. RabbitMQ in cluster: Little work was put into assessing the use of RabbitMQ in a clustered environment. No work was done in a clustered form on a distributed framework. Future work can consider such a framework.
14. Streaming solutions: Kafka or MQTT was not considered for the message queue work. In the future, more work can be put into assessing these technologies' role in a data trust environment.

REFERENCES

Androulaki, Elli, Arem Barger, Vita Bortniko, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, et al. 2018. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains." *EuroSys 2018 conference.* ArXiv. Accessed January 23, 2023. doi:https://doi.org/10.1145/3190508.3190538.

Back, Adam. 2002. *Hashcash-a denial of service counter-measure.* August 1. http://www.hashcash.org/papers/hashcash.pdf.

Balkin, Jack M, and Jonathan Zittrain. 2016. "A Grand Bargain to Make Tech Companies Trustworthy." October 03. Accessed January 01, 2022. https://www.theatlantic.com/technology/archive/2016/10/information-fiduciary/502346/.

Berners-Lee, Tim. n.d. "https://solidproject.org/."

Blog. 2021. *Blockchain Consensus Algorithms: What and How?* October 27. Accessed January 23, 2023. https://www.cbcamerica.org/blockchain-insights/blockchain-consensus-algorithms-what-and-how.

Budzyn, Agnes. 2019. *Data is the oil of the digital world. What if tech giants had to buy it from us?* April 30. https://www.weforum.org/agenda/2019/04/data-oil-digital-world-asset-tech-giants-buy-it/.

Chaum, David L. 1979. *Computer Systems established, maintained and trusted by Mutually suspicious groups.* Ph.D. Dissertation, ELECTRONICS RESEARCH LABORATORY, College of Engineering, University of California, Berkeley. https://chaum.com/wp-content/uploads/2022/02/techrep.pdf.

Delacroix, Sylvie, and Neil Lawrence. 2018. "Bottom-up data Trusts: Disturbing the 'one size fits all' approach." *International Data Privacy Law.* Oxford Academic. 236–252. doi:https://doi.org/10.1093/idpl/ipz014.

Dwork, Cynthia, Moni Naor, and Hoeteck Wee. 2005. "Pebbling and Proofs of Work." *CRYPTO 2005: Advances in Cryptology.* 37-54. Accessed January 20, 2023. doi:https://doi.org/10.1007/11535218_3.

Economist. 2017. *The world's most valuable resource is no longer oil, but data.* May 06. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

n.d. *Element AI.* Accessed December 30, 2021. https://www.elementai.com/.

Ferraiolo, David F, John F. Barkley, and D. Richard Kuhn. 1999. "A role-based access control model and reference implementation within a corporate intranet." *ACM Transactions on Information and System Security* 2 (1): 34-64. https://doi.org/10.1145/300830.300834.

Finley, Klint. 2017. *Tim Berners-Lee, Inventor of the Web, Plots a Radical Overhaul of His Creation.* Wire. April 04. Accessed November 25, 2021. https://www.wired.com/2017/04/tim-berners-lee-inventor-web-plots-radical-overhaul-creation/?mbid=social_fb.

Finney, Hal. 2004. *RPOW - Reusable Proofs of Work.* August 15. https://cryptome.org/rpow.htm.

Furrer, S, W Scott, H.L. Truong, and B. Weiss. 2006. "The IBM wireless sensor networking testbed." *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006.* Barcelona, Spain: IEEE. 5-46. doi:10.1109/TRIDNT.2006.1649126.

GitHub. 2023. *GitHub Copilot for VSCode.* Accessed 2023. https://code.visualstudio.com/docs/copilot/overview.

Haber, Stuart, and W. Scott Stornetta. 1991. "How to time-stamp a digital document." *Journal of Cryptology* (Springer) 99-111. Accessed January 20, 2023. doi:https://doi.org/10.1007/3-540-38424-3_32.

Hall, Dame Wendy, and Jérôme Pesenti. 2017. "Growing the Artificial Intelligence Industry in the U.K." *Gov of UK.* October 15. Accessed November 16, 2021. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/652097/Growing_the_artificial_intelligence_industry_in_the_UK.pdf.

Hardinges, Jack. 2018. *Defining a 'data trust'.* October 19. Accessed January 7, 2022. https://theodi.org/article/defining-a-data-trust/.

—. 2018. *What is a data trust.* July 10. Accessed January 7, 2022. https://theodi.org/article/what-is-a-data-trust/.

Hardings, Jack, Jared Robert Keller, Jeni Tennison, Olivier Thereaux, and Rachel Wilson. 2020. *Designing Trustworthy Data Institutions.* Theodi.org. Accessed August 08, 2022. https://theodi.org/article/designing-trustworthy-data-institutions-report/.

Hu, Vincent C., David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. 2014. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* Computer Security, NIST Special Publication 800-162. Accessed February 17, 2023. doi:NIST.SP.800-162.

Hunkeler, Urs, Hong Linh Truong, and Andy Stanford-Clark. 2008. "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks." *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08).* Bangalore, India: IEEE. 791-798. Accessed February 3, 2023. doi:10.1109/COMSWA.2008.4554519.

2016. "Information Fiduciaries and the First Amendment." *UC Davis Law Review* 49 (4): 1183 - 1234.

n.d. *Inrupt.* https://inrupt.com.

Kreps, Jay, Neha Narkhede, and Jun Rao. 2011. "Kafka: A distributed messaging system for log processing." *In Proceedings of the NetDB*, 1-7.

Kuneva, Meglena. 2009. "European Consumer Commissioner - Keynote Speech - Roundtable on Online Data Collection, Targeting and Profiling." Brussels, March 31. Accessed December 01, 2021. https://ec.europa.eu/commission/presscorner/detail/en/speech_09_156.

Kuratczyk, Michał. 2022. *RabbitMQ 3.10 Performance Improvements.* May 16. Accessed Oct 15, 2023. https://blog.rabbitmq.com/posts/2022/05/rabbitmq-3.10-performance-improvements/.

Lau Jia Jun, Jeremiah, James Penner, and Benjamin Wong. September 23, 2019. "The Basics of Private and Public Data Trusts (September 23, 2019)." NUS Law Working Paper No. 2019/019, EW Barker Centre for Law & Business Working Paper 19/03. Accessed January 19, 2023. doi:http://dx.doi.org/10.2139/ssrn.3458192.

Lucas, Orson. 2021. "Corporate data responsibility: Bridging the consumer trust gap." *KPMG.* August. Accessed December 06, 2021. https://advisory.kpmg.us/articles/2021/bridging-the-trust-chasm.html.

Matttioli, Michael. 2017. "The Data-Pooling Problem." *Berkeley Technology Law Journal* 32 (1): 179-236.

McDonald, Sean, and Keith Porcaro. 2015. *Civic Trust.* August 4. Accessed January 7, 2022. https://medium.com/@digitalpublic/the-civic-trust-e674f9aeab43.

Merkle, Ralph C. 1987. "A Digital Signature Based on a Conventional Encryption Function." *Conference on the Theory and Application of Cryptographic Techniques.* Heidelberg, Berlin: Springer. 369-378. Accessed January 20, 2023. doi: https://doi.org/10.1007/3-540-48184-2_32.

Mills, Stuart. 2019. "Who Owns the Future? Data Trusts, Data Commons, and the Future of Data Ownership." August 23. Accessed August 08, 2022. doi:http://dx.doi.org/10.2139/ssrn.3437936.

Nakamoto, Satoshi. 2008. "Bitcoin: A peer-to-peer electronic cash system." *Decentralized Business Review*, 21260. https://www.debr.io/article/21260.pdf.

n.d. *Nesta.* Accessed December 30, 2021. https://www.nesta.org.uk/.

Nguyen, Giang-Truong, and Kyungbaek Kim. 2018. "A survey about consensus algorithms used in Blockchain." *Journal of information processing systems* 14 (1): 101-128. Accessed January 23, 2023. doi:https://doi.org/10.3745/JIPS.01.0024.

n.d. *NYU GovLab.* Accessed December 16, 2021. https://datacollaboratives.org/.

O'Hara, John. 2007. "Toward a Commodity Enterprise Middleware." 5: 48-55. Accessed January 23, 2023.

O'Hara, Kieron. 2019. "Data Trusts: Ethics, Architecture and Governance for Trustworthy Data Stewardship." doi:http://dx.doi.org/10.5258/SOTON/WSI-WP001.

2012. *Open Data Institute.* Accessed December 30, 2021. https://theodi.org/.

OpenAI. 2023. *ChatGPT*. https://chat.openai.com.

Paprica, P. Alison, Eric Sutherland, Andrea Smith, Michael Brudno, Rosario G Cartagena, Monique Circhlow, Brian K Courtney, et al. 2020. "Essential requirements for establishing and operating data trusts: practical guidance co-developed by representatives from fifteen Canadian organizations and initiatives." *International Journal of Population Data Science* 5 (1): August. doi:https://doi.org/10.23889/ijpds.v5i1.1353.

Sambra, Andrei Vlad, Essam Mansour, Sandro Hawke, Maged Zereba, Nicola Greco, Abdurrahman Ghanem, Dmitri Zagidulin, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. "Solid: A Platform for Decentralized Social Application Based on Linked Data." MIT CSAIL & Qatar Computing Research Institute. Accessed December 30, 2021. http://emansour.com/research/meccano/solid_protocols.pdf.

Stalla-Bourdillon, Sophie, Gefion Thuermer, Johanna Walker, Laura Carmichael, and Elena Simperl. 2020. "Data protection by design: Building the foundations of trustworthy data sharing." *Data & Policy* E4. doi:10.1017/dap.2020.1.

Susha, Iryna, M.F.W.H.A Janssen, and Stefaan Verhulst. 2017. "Data Collaboratives as a New Frontier of Cross-Sector Partnerships in the Age of Open Data: Taxonomy Development." *Proceedings of the 50th Hawaii International Conference on System Sciences.* The University of Hawaii. doi:https://doi.org/10.24251/hicss.2017.325.

n.d. *Trust Law.* Accessed January 7, 2022. https://en.wikipedia.org/wiki/Trust_law#:~:text=A%20trust%20is%20a%20legal%20relationship%20in%20which,to%20hold%20and%20use%20it%20for%20another%27s%20benefit.

Vinoski, Steve. 2006. "Advanced Message Queuing Protocol." IEEE Internet Computing. 87 - 89. Accessed January 24, 2023. doi:10.1109/MIC.2006.116.

n.d. *What is a data trust?* Sightline Innovation. Accessed January 7, 2022. https://sightlineinnovation.com/faq.

Young, Meg, Luke Rodriguez, Emily Keller, Feiyang Sun, Boyang Sa, Jan Whittington, and Bill Howe. January 2019. "Beyond Open vs. Closed: Balancing Individual Privacy and Public Accountability in Data Sharing." *Proceedings of the Conference on Fairness, Accountability, and Transparency.* ACM Digital Library. 191-200. doi:https://doi.org/10.1145/3287560.3287577.

APPENDIX A

JMeter Data points are acquired through tests for the app's REST API document.

REST API document endpoint (Standard/Common Endpoint)

POST Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	9483.667	10242.33	10948.33	11552.67	12148.67	5106.333	12748.67	0.02	7.77217	7.036667	6.74
200	15094.8	15516.4	21847.6	21882.4	23380.6	2228.4	23411.6	0.022	8.204572	8.364	7.5
300	26539.8	26460.4	41976.8	42051.4	42855	4194.8	43168.6	0.039334	7.321488	7.42	6.688
400	40951.8	41361	66260.6	66322	66716.8	3297	67018	0.051	6.518258	6.54	5.956
500	56215.8	57917.6	92331.2	92435	92904.2	2911.8	93332	0.0612	5.979296	5.986	5.464
600	78524.6	79378.2	132746.2	132972.2	133151	2769.6	133419	0.072346	4.890454	4.862	4.47
700	93218.8	93771.8	158504	158889.4	159137.4	4675.6	159485.8	0.109146	4.847616	5.074	4.294
800	99242.6	102409.8	172900	173220.8	173449.6	4041.4	174066.4	0.13625	5.064316	5.458	4.372
900	93123.6	86572.6	177775.8	178716.8	179039.4	2291	179276.8	0.235778	5.206898	6.62	3.952
1000	91173	74292	180325	180942	181987	8040	182106	29.90%	5.47028	7.67	3.72

Table A. 1: POST Method Aggregate Data for the Standard Endpoint

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	9483.667	0	12748.67	1766.237	2.00%	7.77217	7.036667	6.74	927.6333
200	15094.8	0	23411.6	6325.804	0.022	8.204572	8.364	7.5	1043.8
300	26539.8	0	43168.6	12709.42	0.039334	7.321488	7.42	6.688	1034.72
400	40951.8	0	67018	20511.26	0.051	6.518258	6.54	5.956	1026.26
500	56215.8	0	93332	29061.96	0.0612	5.979296	5.986	5.464	1020.18
600	78938.6	0	133954.8	41734.5	0.075334	4.877986	4.844	4.46	1011.62
700	93218.8	0	159485.8	50832.34	0.109152	4.847616	5.074	4.294	1061.76
800	99242.6	0	174066.4	57009.78	0.13627	5.064316	5.458	4.372	1092.36
900	93123.6	0	179276.8	58636.88	0.235782	5.206898	6.62	3.952	1306.3
1000	83712.8	0	169162.2	56707.93	0.2656	6.333988	8.34	4.604	1354.16

Table A. 2: POST Method Summary Data for the Standard Endpoint

PUT Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	6494.6	6421.4	9596.4	10925.2	11907.2	2005	12111	0.046	7.992374	7.778	6.672
200	11878	11744.4	19153	19865.8	21751.2	3022	21861.4	0.035	8.918688	8.722	7.448
300	16982	16791	28036.2	29311.4	31284.8	4548.2	31504	0.038666	9.421338	9.202	7.864
400	21821.4	21619.2	37606.6	39191.4	40583.8	2960.4	41829.4	0.0465	9.373146	9.118	7.826
500	26820.4	26812.6	46064.6	48589	50264.6	5161.6	50971.8	0.0416	9.663112	9.422	8.07
600	31037.6	30522	53891.8	57116.2	59236.4	3407.2	60624.2	0.033	9.801954	9.594	8.184
700	34289.8	32488	60799.4	64401	66613.4	4415.4	68196.2	0.10857	10.18829	11.34	7.852
800	39233.6	38308	69928	73883.4	76792.8	4176.6	78025.6	0.0825	10.18003	10.986	8.018
900	48152.8	41163.6	97583.6	104412.6	109538	2046	111093.6	0.173776	8.557338	10.332	6.158
1000	45564.2	39758.6	89467.8	95591.8	99728	2505	101350	0.1894	10.19472	12.52	7.226

Table A. 3: PUT Method Aggregate Data for the Standard Endpoint

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES	STANDARD ERROR
100	6494.6	0	12111	2644.47	0.046	7.992374	7.778	6.672	996.74	264.447
200	11878	0	21861.4	5467.598	0.035	8.918688	8.722	7.448	1001.56	386.6176
300	16982	0	31504	7937.886	0.038666	9.421338	9.202	7.864	999.96	458.2941
400	21821.4	0	41829.4	11149.09	0.0465	9.373146	9.118	7.826	996.4	557.4543
500	26820.4	0	50971.8	13791.82	0.0416	9.663112	9.422	8.07	998.66	616.7888
600	31037.6	0	60624.2	16474.45	0.033	9.801954	9.594	8.184	1002.44	672.5666
700	34289.8	0	68196.2	18304.43	0.10857	10.18829	11.34	7.852	1139.6	691.8423
800	39233.6	0	78025.6	21619.05	0.0825	10.18003	10.986	8.018	1104.02	764.3488
900	48152.8	0	111093.6	31932.61	0.173776	8.557338	10.332	6.158	1234.66	1064.42
1000	45564.2	0	101350	29211.29	0.1894	10.19472	12.52	7.226	1262.38	923.7422

Table A. 4: PUT Method Summary Data for the Standard Endpoint

DELETE Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	5304.25	5753	6399.75	6465.25	6540	2723	6557.25	0	13.65547	13.4575	12.2175
200	9508.8	10508.8	12603	12752.2	12881.2	3625	13044.4	0	14.39129	14.18	12.832
300	14657	16625	20561.6	20915.4	21172.4	4278.8	21470.4	0	13.50767	13.312	12.024
400	18342.2	21294.8	26004	26295.8	26599.8	3606.6	26671.4	0	14.4295	14.218	12.86
500	23223.2	26770	32559.2	33193.8	33441	4137.6	33524.4	0.004	14.53742	14.29	12.954
600	27099.4	30204.6	40132	40706.8	41180.4	4341.8	41312.8	0	14.16921	13.96	12.616
700	31908.4	37477.8	47791.4	48848.8	49388.4	4201.8	49466	0.000286	13.98379	13.784	12.456
800	33564.6	37340.2	53145	54039.8	54837.6	3605.8	55120.6	0.00177	14.38253	14.158	12.808
900	38553	42094.8	57822.2	58634.4	59325.8	4232.4	59518.4	0.00378	14.9517	14.782	13.29
1000	51890.6	52892.6	84194.2	85441.2	86619.2	4378.2	86888.6	0.0252	12.66809	12.444	11.146

Table A. 5: DELETE Method Aggregate Data for the Standard Endpoint

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	5304.25	0	6557.25	1140.228	0	13.65547	13.4575	12.2175	1009
200	9508.8	0	13044.4	2857.692	0	14.39129	14.18	12.832	1009
300	14657	0	21470.4	5517.142	0	13.50767	13.312	12.024	1009
400	18342.2	0	26671.4	7338.386	0	14.4295	14.218	12.86	1009
500	23223.2	0	33524.4	9022.628	0.004	14.53742	14.29	12.954	1006.44
600	27099.4	0	41312.8	11873.97	0	14.16921	13.96	12.616	1009
700	31908.4	0	49466	14096.43	0.00028	13.98379	13.784	12.456	1009.46
800	33564.6	0	55120.6	16008.72	0.00178	14.38253	14.158	12.808	1007.88
900	38553	0	59518.4	16732.43	0.00378	14.9517	14.782	13.29	1012.6
1000	51890.6	0	86888.6	27254.17	0.0252	12.66809	12.444	11.146	1001.68

Table A. 6: DELETE Method Summary Data for the Standard Endpoint

GET Method

Aggregate Data

100	123.4	122.2	135.4	140.4	144.2	113.8	147	0	90.73083	91.406	29.328
200	137	130.2	176.4	184.8	192.2	115.4	199	0	174.268	175.326	56.33
300	336.2	343.8	483.2	500	515.6	122.8	544.4	0	227.0326	228.05	73.386
400	428.8	456.6	616.4	637.2	652.4	123.8	673.8	0	226.695	227.526	73.278
500	744.6	817.6	1084.8	1174.4	1649.4	141.6	2011	0	183.1776	183.748	59.21
600	902.4	914.2	1527.8	1630	2019.8	142.8	5952.4	0.00034	170.1855	170.598	55.01
700	1447.2	1185.8	2525.8	3051.8	3178	152.4	4196.4	0	157.6138	158.088	50.948
800	1699.6	1729	3151.4	3252.4	3377	157.8	4291	0	172.3742	173.142	55.718
900	2083	2076.4	3624.4	3701.4	4074.2	164	5180.8	0	166.9417	168.098	53.962
1000	2039	2099.8	3528.4	3634	4126.8	167	8087.2	0.0002	166.4633	168.158	53.804

Table A. 7: GET Method Aggregate Data for the Standard Endpoint

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES	STANDARD ERROR
100	123.4	0	147	7.294	0	90.73083	91.406	29.328	1031.62	0.7294
200	137	0	199	20.236	0	174.268	175.326	56.33	1030.22	1.430901282
300	336.2	0	544.4	117.516	0	227.0326	228.05	73.386	1028.54	6.784789423
400	428.8	0	673.8	157.008	0	226.695	227.526	73.278	1027.68	7.8504
500	744.6	0	2011	341.01	0	183.1776	183.748	59.21	1027.16	15.25043082
600	902.4	0	5952.4	537.398	0.00034	170.1855	170.598	55.01	1026.66	21.93918148
700	1447.2	0	4196.4	793.174	0	157.6138	158.088	50.948	1027.16	29.97915929
800	1699.6	0	4291	942.842	0	172.3742	173.142	55.718	1028.52	33.33449859
900	2083	0	5180.8	1131.792	0	166.9417	168.098	53.962	1031.3	37.7264
1000	2039	0	8087.2	1161.798	0.0002	166.4633	168.158	53.804	1033.8	36.73927861

Table A. 8: GET Method Summary Data for the Standard Endpoint

REST API document endpoint without blockchain.

POST Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	6199	7208.6	7396.8	7410.4	7525.6	1961.6	7539.4	0	12.21924	12.624	11.202
200	13871	16052	17551.6	17593.6	17650	5136.2	17701.4	0	11.24153	11.616	10.328
300	19964.8	22920	28374.2	28414.8	28507.6	2501.2	28606	0	10.56707	10.916	9.71
400	32137.8	37338.4	45087.6	45186.6	45271.4	4709	45310.8	0	9.526654	9.844	8.752
500	43362.8	48034.8	63003.6	63093	63234	6276.6	63283.8	0.0004	8.73776	9.026	8.028
600	35719.8	39451.6	51800.6	51955.4	52047.4	4912	52097	0	11.37031	11.746	10.448
700	40979.2	45662.2	60808.2	61046.8	61189.6	2319.2	61280.4	0.000286	11.23185	11.602	10.322
800	46224	51491.6	68727.2	69005.6	69202.2	2295	69336.8	0	11.43112	11.81	10.504
900	49717.2	55200	75517.8	75835.4	76075.4	2487.6	76161.4	0.029106	11.68734	12.668	10.42
1000	80451.2	87984.8	133295.6	133787.4	134101.6	3537.2	134274.8	0.1038	7.913856	9.404	6.56

Table A. 9: POST Method Aggregate Data for the Endpoint W/O Blockchain

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	6199	0	7539.4	1573.02	0	12.21924	12.624	11.202	1058
200	13871	0	17701.4	4007.488	0	11.24153	11.616	10.328	1058
300	19964.8	0	28606	8595.622	0	10.56707	10.916	9.71	1058
400	32137.8	0	45310.8	13234.18	0	9.526654	9.844	8.752	1058
500	43362.8	0	63283.8	19262.57	0.0004	8.73776	9.026	8.028	1057.72
600	35719.8	0	52097	15686.59	0	11.37031	11.746	10.448	1058
700	40979.2	0	61280.4	19062.91	0.000286	11.23185	11.602	10.322	1057.8
800	46224	0	69336.8	21526.74	0	11.43112	11.81	10.504	1058
900	49717.2	0	76161.4	23960.1	0.02911	11.68734	12.668	10.42	1108.56
1000	80451.2	0	134274.8	46827.52	0.1038	7.913856	9.404	6.56	1224.8

Table A. 10: POST Method Summary Data for the Endpoint W/O Blockchain

PUT Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	4550.8	4495	6884.6	6909.8	6934.2	683.6	6953.4	0	12.73629	12.648	10.72
200	8378.4	8622.4	14113.8	14490.4	14536.4	624.6	14566.2	0	12.88883	12.8	10.852
300	12236.8	12012.2	21256.4	21895	21940.2	910.8	21951	0	13.1552	13.064	11.074
400	15748.2	15376	27631.4	28652	28933.4	683.8	28990.2	0	13.25912	13.168	11.16
500	19770	19550.8	34652.4	35945.8	36388	719.4	36448	0	13.37128	13.28	11.256
600	23734.2	24229.4	41259	42722.2	43497	915	43560.4	0	13.42906	13.34	11.306
700	27860.6	27589.8	49089.6	51298.8	52133.4	1117.2	52216	0.00343	13.243	13.202	11.118
800	30841.2	31204.8	55282	56927	57575.6	1058.8	57653.6	0.02875	13.7137	14.29	11.214
900	32814.6	31625.2	59317.6	61857.6	62666.2	1080.2	62807	0.063334	14.17631	15.548	11.186
1000	33075.6	30702.8	60785.8	64821.4	66130.6	1028	66276.6	0.1152	14.91143	17.674	11.084

Table A. 11: PUT Method Aggregate Data for the Endpoint W/O Blockchain

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	4550.8	0	6953.4	1727.656	0	12.73629	12.648	10.72	1017
200	8378.4	0	14566.2	4056.122	0	12.88883	12.8	10.852	1017
300	12236.8	0	21951	6323.946	0	13.1552	13.064	11.074	1017
400	15748.2	0	28990.2	8435.138	0	13.25912	13.168	11.16	1017
500	19770	0	36448	10572.71	0	13.37128	13.28	11.256	1017
600	23734.2	0	43560.4	12649.59	0	13.42906	13.34	11.306	1017
700	27860.6	0	52216	15286.62	0.00343	13.243	13.202	11.118	1021.04
800	30841.2	0	57653.6	17024.38	0.02875	13.7137	14.29	11.214	1066.8
900	32814.6	0	62807	18033.01	0.063334	14.17631	15.548	11.186	1121.68
1000	33075.6	0	66276.6	19078.84	0.1152	14.91143	17.674	11.084	1210.36

Table A. 12: PUT Method Summary Data for the Endpoint W/O Blockchain

DELETE Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	1756.2	1708.4	2063.4	2093.4	2136.2	1070.8	5947	0.002	27.9778	27.582	24.68
200	3645.8	4223.2	4760.2	4778.4	4805.4	1281.2	4840	0	35.03459	34.52	31.384
300	4986.2	5655.4	6732.6	6759.8	6821.6	1292.2	6850	0	39.24296	38.67	35.126
400	6973.4	7912.8	9556.2	9613.2	9672	1645	11985.6	0.001	33.38074	32.896	29.922
500	8846	10090.4	11923.2	11946.8	12121.4	1535	12442.8	0	38.75864	38.19	34.656
600	10323.4	11769	14316.6	14495.4	14550.4	1840.6	14587	0	38.78457	38.216	34.678
700	12009.2	13288.6	17445.8	17491.4	17635	1840	18228.2	0.00086	37.59199	37.064	33.676
800	13512.4	14997	19359.2	19405.6	19447.6	2698.6	19918.2	0.00076	38.94333	38.376	34.818
900	15225.2	17482.4	21920.2	22011.2	22103.6	2372.8	22173.6	0.00022	39.40088	38.838	35.27
1000	16705.2	19126	23860.8	23998.2	24098.4	3365.2	24224.4	0	40.29559	39.706	36.082

Table A. 13: DELETE Method Aggregate Data for the Endpoint W/O Blockchain

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	1756.2	0	5947	656.356	0.002	27.9778	27.582	24.68	1011.44
200	3645.8	0	4840	1196.516	0	35.03459	34.52	31.384	1009
300	4986.2	0	6850	1787.426	0	39.24296	38.67	35.126	1009
400	6973.4	0	11985.6	2635.404	0.001	33.38074	32.896	29.922	1009.48
500	8846	0	12442.8	3274.568	0	38.75864	38.19	34.656	1009
600	10323.4	0	14587	4073.87	0	38.78457	38.216	34.678	1009
700	12009.2	0	18228.2	5140.378	0.00086	37.59199	37.064	33.676	1009.74
800	13512.4	0	19918.2	5621.248	0.00076	38.94333	38.376	34.818	1009.08
900	15225.2	0	22173.6	6539.604	0.00022	39.40088	38.838	35.27	1009.36
1000	16705.2	0	24224.4	6966.246	0	40.29559	39.706	36.082	1009

Table A. 14: DELETE Method Summary Data for the Endpoint W/O Blockchain

GET Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	123.8	122	137.6	142.4	146.8	113.8	150.2	0	90.21893	89.602	29.516
200	131.2	130.4	146.2	148.6	152.6	114.2	159.6	0	178.015	176.798	58.238
300	297	282	449.25	464.5	480.5	120.75	492.75	0	227.5795	226.025	74.45
400	316.2	300.8	472	489	515.6	119.6	535.8	0	231.2191	229.638	75.642
500	709.8	724.6	1108	1152	1169.6	128	1343.8	0	226.0361	224.49	73.946
600	745	788.2	1105	1214.8	1570.4	125.8	5704.2	0.000334	183.6719	182.432	60.086
700	1259.6	1122.6	2177.8	2414	3034.6	152.4	3761	0	163.5418	162.426	53.504
800	1322.6	1153	2457	2832	3345.2	123.6	3683.4	0	173.3983	172.214	56.726
900	1709.4	1631.8	3190.4	3232	3876	154.2	4805.2	0	173.9827	172.794	56.918
1000	1833.8	1814.8	3358.6	3401	3902	144.2	5048.6	0	179.5027	178.276	58.724

Table A. 15: GET Method Aggregate Data for the Endpoint W/O Blockchain

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	123.8	0	150.2	8.416	0.00%	90.218926	89.602	29.516	1017
200	131.2	0	159.6	10.634	0.00%	178.015048	176.798	58.238	1017
300	297	0	492.75	111.435	0.00%	227.57951	226.025	74.45	1017
400	316.2	0	535.8	119.538	0.00%	231.219116	229.638	75.642	1017
500	709.8	0	1343.8	337.682	0.00%	226.036146	224.49	73.946	1017
600	745	0	5704.2	466.692	0.03%	183.671878	182.432	60.086	1017.54
700	1259.6	0	3761	717.51	0.00%	163.54179	162.426	53.504	1017
800	1322.6	0	3683.4	817.426	0.00%	173.398348	172.214	56.726	1017
900	1709.4	0	4805.2	1010.252	0.00%	173.982704	172.794	56.918	1017
1000	1833.8	0	5048.6	1077.742	0.00%	179.502678	178.276	58.724	1017

Table A. 16: GET Method Summary Data for the Endpoint W/O Blockchain

REST API document endpoint without blockchain and message queue.

POST Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	5058.6	5835.2	6827.2	6851.6	6871	563	6878.6	0	13.07223	13.494	12.052
200	10340.8	11721.8	14350.8	14364.6	14381.2	1669.6	14391.2	0	13.09299	13.516	12.068
300	15893.4	18436.2	22002.8	22013.6	22028	1817	22040.2	0	13.12371	13.546	12.098
400	21062	24896.4	28959.6	28972.6	28992	1451	29004.4	0	13.28657	13.714	12.248
500	25952.6	29565.8	36611.2	36635	36674.6	1738.8	36718.2	0	13.29327	13.72	12.254
600	31236.2	37261.2	43177.2	43210	43262.6	2176.2	43473.8	0	13.52532	13.962	12.47
700	36384.6	42687.2	51698	51791	51858.8	1748.8	51901.6	0	13.34317	13.774	12.3
800	41331	48532	58776.8	58834	58945.8	1685.2	58972.6	0	13.29581	13.722	12.258
900	46906.2	57456.2	66071.6	66195.8	66384.2	1872.2	66460.2	0.00044	13.34523	13.77	12.302
1000	50367.4	57429.4	72539.4	72746.8	72977.6	3467	73081.2	0.0132	13.5273	14.25	12.3

Table A. 17: POST Method Aggregate Data for the Endpoint W/O Blockchain and Message Queue

Summary Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	5058.6	5835.2	6827.2	6851.6	6871	563	6878.6	0	13.07223	13.494	12.052
200	10340.8	11721.8	14350.8	14364.6	14381.2	1669.6	14391.2	0	13.09299	13.516	12.068
300	15893.4	18436.2	22002.8	22013.6	22028	1817	22040.2	0	13.12371	13.546	12.098
400	21062	24896.4	28959.6	28972.6	28992	1451	29004.4	0	13.28657	13.714	12.248
500	25952.6	29565.8	36611.2	36635	36674.6	1738.8	36718.2	0	13.29327	13.72	12.254
600	31236.2	37261.2	43177.2	43210	43262.6	2176.2	43473.8	0	13.52532	13.962	12.47
700	36384.6	42687.2	51698	51791	51858.8	1748.8	51901.6	0	13.34317	13.774	12.3
800	41331	48532	58776.8	58834	58945.8	1685.2	58972.6	0	13.29581	13.722	12.258
900	46906.2	57456.2	66071.6	66195.8	66384.2	1872.2	66460.2	0.00044	13.34523	13.77	12.302
1000	50367.4	57429.4	72539.4	72746.8	72977.6	3467	73081.2	0.0132	13.5273	14.25	12.3

Table A. 18: POST Method Summary Data for the Endpoint W/O Blockchain and Message Queue

PUT Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	3586.2	3437.6	5654.4	5805.4	5858	400.6	5868.6	0	14.7206	14.62	13.01
200	7220.6	7136.2	12301.6	12778.2	12951	1044.8	12986.2	0	14.3583	14.26	12.69
300	10529.6	10486.6	17944.2	18753.4	19000.6	1771.4	19033.4	0	15.06774	14.966	13.316
400	13819.6	13481.8	24333.6	24987.8	25246.4	2339	25260	0	15.12947	15.026	13.372
500	20561.2	20669.8	33745	34543	34810.2	4559.4	34841.2	0	14.16751	14.07	12.522
600	21178.2	20967.8	37409.6	39092.6	40258.8	872.2	40420.4	0	14.43634	14.34	12.758
700	25614.4	25343.8	45427.6	46662.4	47185.6	1767.4	47231.8	0	14.65711	14.556	12.954
800	28854.4	28507.8	50724	53150	53584.8	2159.6	53645.8	0	14.73466	14.632	13.022
900	33717.6	33443	59522.4	61346	61852	1255.2	61947.6	0.00422	14.36402	14.356	12.644
1000	33606.8	33051	61010.2	64095	64758.6	1322.8	64906.6	0.064	15.22866	16.658	12.596

Table A. 19: PUT Method Aggregate Data for the Endpoint W/O Blockchain and Message Queue

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	3586.2	0	5868.6	1523.812	0	14.7206	14.62	13.01	1017
200	7220.6	0	12986.2	3534.612	0	14.3583	14.26	12.69	1017
300	10529.6	0	19033.4	5320.586	0	15.06774	14.966	13.316	1017
400	13819.6	0	25260	7204.218	0	15.12947	15.026	13.372	1017
500	20561.2	0	34841.2	9562.35	0	14.16751	14.07	12.522	1017
600	21178.2	0	40420.4	11694.41	0	14.43634	14.34	12.758	1017
700	25614.4	0	47231.8	14419.51	0	14.65711	14.556	12.954	1017
800	28854.4	0	53645.8	15930.53	0	14.73466	14.632	13.022	1017
900	33717.6	0	61947.6	18797.17	0.00422	14.36402	14.356	12.644	1023.22
1000	33380	0	64527.4	19009.81	0.0666	15.3167	16.814	12.634	1124.1

Table A. 20: PUT Method Summary Data for the Endpoint W/O Blockchain and Message Queue

DELETE Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	851.6	948	1027	1036.4	1051.6	333.6	1068.2	0	53.67503	52.834	48.268
200	1725.75	1965.5	2211.5	2227.5	2244.5	383.75	2253.25	0	65.07692	64.06	58.685
300	3310	3926.4	4654.2	4699.2	4742.6	439.2	4757	0	55.85733	54.984	50.368
400	4745.2	5548.2	6637.4	6675.2	6732.2	638.4	6746.6	0	51.19237	50.392	46.102
500	5755	6507	8039.4	8071.4	8110.4	770.6	10950.4	0.0004	48.13023	47.392	42.932
600	6634.4	7454.6	9496	9511.8	9542.8	640.8	9572.8	0	56.43515	55.554	50.892
700	8075.8	9169.4	11142.8	11209	11295.2	686	13495.6	0.00144	53.9865	53.114	48.792
800	8782.8	9743.2	12554.4	12621.8	12778.2	868.8	12890.8	0	59.87737	58.942	53.88
900	11468.4	12643.4	16246.8	16355	16494	1031.6	17158.4	0.00044	51.40111	50.626	47.088
1000	11069	12547	16108	16200.4	16318	827.8	16451.4	0	58.75339	57.41	52.802

Table A. 21: DELETE Method Aggregate Data for the Endpoint W/O Blockchain and Message Queue

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	851.6	0	1068.2	203.516	0	53.67503	52.834	48.268	1008
200	1725.75	0	2253.25	553.5725	0	65.07692	64.06	58.685	1008
300	3310	0	4757	1393.536	0	55.85733	54.984	50.368	1008
400	4745.2	0	6746.6	1940.35	0	51.19237	50.392	46.102	1008
500	5755	0	10950.4	2323.594	0.0004	48.13023	47.392	42.932	1008.64
600	6634.4	0	9572.8	2792.928	0	56.43515	55.554	50.892	1008
700	8075.8	0	13495.6	3223.464	0.00144	53.9865	53.114	48.792	1007.78
800	8782.8	0	12890.8	3590.31	0	59.87737	58.942	53.88	1008
900	11468.4	0	17158.4	4719.776	0.00044	51.40111	50.626	47.088	1008.72
1000	11069	0	16451.4	4795.238	0	58.75339	57.41	52.802	1001

Table A. 22: DELETE Method Summary Data for the Endpoint W/O Blockchain and Message Queue

GET Method

Aggregate Data

# SAMPLES	AVERAGE	MEDIAN	90% LINE	95% LINE	99% LINE	MIN	MAX	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC
100	121.4	121.6	127.2	128.8	131	114.6	132.2	0	88.70667	88.1	78.398
200	131.2	126.4	160.2	166.6	182.2	115.6	186.4	0	172.6248	171.444	152.564
300	327	332.8	467.6	484	501.8	124.2	511.8	0	224.5176	222.984	198.426
400	368.4	380.8	533.2	560	600.8	122	618.2	0	226.8144	225.266	200.456
500	707.2	742	1061.8	1098	1188.6	137.8	1660.4	0	209.7201	208.288	185.35
600	825.8	852	1297.2	1497.2	1880	134.8	5865.6	0.00034	171.6711	170.512	151.714
700	1472.6	1251	2894.8	3191.2	3287.8	150	3695.8	0	167.044	165.902	147.63
800	1693.8	1634.4	3227.4	3291.6	3445.2	166.2	4067.8	0	173.0951	171.912	152.98
900	1870	1926.4	3325.4	3380.8	3714.8	174.2	7767.8	0.00022	155.6601	154.61	137.564
1000	2052	2089.6	3591.2	3723.2	4545	159	5827.6	0	165.4423	164.31	146.218

Table A. 23: GET Method Aggregate Data for the Endpoint W/O Blockchain and Message Queue

Summary Data

# SAMPLES	AVERAGE	MIN	MAX	STD. DEV.	ERROR %	THROUGHPUT	RECEIVED KB/SEC	SENT KB/SEC	AVG. BYTES
100	121.4	0	132.2	4.072	0	88.70667	88.1	78.398	1017
200	131.2	0	186.4	15.946	0	172.6248	171.444	152.564	1017
300	327	0	511.8	113.26	0	224.5176	222.984	198.426	1017
400	368.4	0	618.2	135.234	0	226.8144	225.266	200.456	1017
500	707.2	0	1660.4	320.94	0	209.7201	208.288	185.35	1017
600	819	0	1982.2	406.674	0	212.5747	211.122	187.872	1017
700	1472.6	0	3695.8	866.026	0	167.044	165.902	147.63	1017
800	1693.8	0	4067.8	978.324	0	173.0951	171.912	152.98	1017
900	1870	0	7767.8	1088.31	0.000222	155.6601	154.61	137.564	1017.36
1000	2052	0	5827.6	1171.782	0	165.4423	164.31	146.218	1017

Table A. 24: GET Method Summary Data for the Endpoint W/O Blockchain and Message Queue

APPENDIX B

This section shows the data points obtained using the pgbench utility to assess the PostgreSQL databases on the docker container and that of a standalone installation.

Docker-based PostgreSQL.

1000 Transactions/Client Test

#CLIENTS	# OF THREADS	TRANSACTION /CLIENT	TOTAL PROCESSED TRANSACTIONS	AVERAGE LATENCY MS	TPS W CONNECTION	TPS W/O CONNECTION
10	2	1000	10000/10000	6.935	1441.973827	1443.056212
20	2	1000	20000/20000	17.987	1111.930349	1112.231148
30	2	1000	30000/30000	26.128	1148.185556	1148.39226
40	2	1000	40000/40000	35.686	1120.894847	1121.039495
50	2	1000	50000/50000	44.851	1114.803158	1114.967668
60	2	1000	60000/60000	51.624	1162.24621	1162.360851
70	2	1000	70000/70000	61.055	1146.498151	1146.59723
80	2	1000	80000/80000	69.156	1156.798978	1156.940163
90	2	1000	90000/90000	77.762	1157.374061	1157.452367
100	2	1000	100000/100000	87.691	1140.366489	1140.444936

Table B. 1: PGBench runs on 1000 Transactions per Client on Docker

10000 Transactions/Client Test

#CLIENTS	# OF THREADS	TRANSACTION /CLIENT	TOTAL PROCESSED TRANSACTIONS	AVERAGE LATENCY MS	TPS W CONNECTION	TPS W/O CONNECTION
10	2	10000	100000/100000	8.166	1224.534923	1224.622676
20	2	10000	200000/200000	16.434	1216.971474	1217.035723
30	2	10000	300000/300000	27.694	1083.280811	1083.299306
40	2	10000	400000/400000	41.589	961.78652	961.797651
50	2	10000	500000/500000	59.861	835.271712	835.278909
60	2	10000	600000/600000	80.496	745.380513	745.386103
70	2	10000	700000/700000	94.986	736.947654	736.951814
80	2	10000	800000/800000	110.592	723.379514	723.38302
90	2	10000	900000/900000	130.268	690.881562	690.884339
100	2	10000	1000000/1000000	164.113	609.33722	609.339159

Table B. 2: PGBench runs on 10000 Transactions per Client on Docker

Standalone installation of PostgreSQL

1000 Transactions/client

#CLIENTS	# OF THREADS	TRANSACTION/ CLIENT	TOTAL PROCESSED TRANSACTIONS	AVERAGE LATENCY MS	TPS W CONNECTION	TPS W/O CONNECTION
10	2	1000	10000/10000	5.488	1822.241216	1824.335052
20	2	1000	20000/20000	10.757	1859.303002	1860.49323
30	2	1000	30000/30000	16.073	1866.540089	1867.193248
40	2	1000	40000/40000	21.274	1880.235141	1880.65112
50	2	1000	50000/50000	29.271	1708.179221	1708.533117
60	2	1000	60000/60000	35.65	1683.051439	1683.26593
70	2	1000	70000/70000	41.791	1674.988138	1675.238372
80	2	1000	80000/80000	45.936	1741.56256	1741.766737
90	2	1000	90000/90000	51.053	1762.862968	1763.113101
100	2	1000	100000/100000	55.827	1791.23837	1791.411424

Table B. 3: PGBench runs on 1000 Transactions per Client on a Standalone

10000 Transactions/client

#CLIENTS	# OF THREADS	TRANSACTION/ CLIENT	TOTAL PROCESSED TRANSACTIONS	AVERAGE LATENCY MS	TPS W CONNECTION	TPS W/O CONNECTION
10	2	10000	100000/100000	5.962	1677.175959	1677.313599
20	2	10000	200000/200000	11.797	1695.401957	1695.476663
30	2	10000	300000/300000	17.7	1694.869838	1694.92017
40	2	10000	400000/400000	33.072	1209.477453	1209.493802
50	2	10000	500000/500000	45.151	1107.3901	1107.402101
60	2	10000	600000/600000	51.352	1168.411507	1168.422937
70	2	10000	700000/700000	67.957	1030.057657	1030.06543
80	2	10000	800000/800000	75.533	1059.14591	1059.153683
90	2	10000	900000/900000	85.234	1055.913457	1055.92114
100	2	10000	1000000/1000000	99.541	1004.613602	1004.620114

Table B. 4: PGBench runs on 10000 Transactions per Client on a Standalone

APPENDIX C

Statistical data was obtained by running the perf-test utility on RabbitMQ as a docker container as part of the application and as a standalone installation. The test was run for 300s with producers and consumers in a 1:1 ratio.

RabbitMQ in Docker

# PRODUCERS	#CONSUMERS	DURATION	SENDING MSG/S	MSG CONSUMED MSG/S	CONSUMER LATENCY MIN µS	CONSUMER LATENCY MEDIAN µS	CONSUMER LATENCY 75TH µS	CONSUMER LATENCY 95TH µS	CONSUMER LATENCY 99TH µS
10	10	300	5470	3843	7980676	118993209	121829443	125840754	129261595
20	20	300	8992	5388	17367820	210631749	219953303	230394650	243384283
30	30	300	10677	5363	13789350	251245233	270278194	287609450	292596692
40	40	300	11446	5581	2060325	255359743	276990197	292566646	298113232
50	50	300	12192	5230	3435871	271952722	284659788	296857798	299967173

Table C. 0.1: RabbitMQ perf-test run on Docker

RabbitMQ as Standalone (Used data points 10-50 in our analysis)

# PRODUCERS	#CONSUMERS	DURATION	SENDING MSG/S	MSG CONSUMED	MIN µS	MEDIAN µS	75TH µS	95TH µS	99TH µS
10	10	300	19434	17767	4820401	47051090	48494319	49782296	50546192
20	20	300	17529	13998	2856707	113203919	116914784	121225967	123506068
30	30	300	19125	13604	19907192	161363481	168693849	178829550	182069113
40	40	300	22171	15354	5518636	175166632	184282896	192283768	196885130
50	50	300	20643	14577	2626791	165295177	186520439	195806267	198620490
60	60	300	19259	11856	2646650	128518899	192134024	242077727	242077727
70	70	300	17507	12598	2726853	164309476	181192780	193194345	235147241
80	80	300	19715	12971	6015922	98745869	196275060	220868066	229856364
90	90	300	20053	12840	6655550	99078435	185091955	224971512	232766307
100	100	300	20091	12207	4690878	109791311	174219726	222142966	249828126

Table C. 0.2: RabbitMQ perf-test run on Standalone