Building a Blockchain-based API Access Control 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

KHA THAI LE

ABSTRACT

API providers can expose their service and data via APIs. However, there must be an access control mechanism in place to control which client can access the APIs. Blockchain technology holds significant potential for this use case. While blockchain may introduce latency, it also offers inherent features including decentralization, data immutability, scalability, and traceability.

This thesis explores implementing a blockchain-based access control system and conducts performance evaluations. The proposed comprehensive solution features a straightforward architecture and a user-friendly web interface. It has been deployed in a cloud environment for development, testing, and performance assessments. Extensive experiments have been conducted to analyze latency and determine the system's breaking point. It can withstand 14000 client apps loading it simultaneously within the cloud environment where it was deployed.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to all those who have contributed to completing this thesis.

First and foremost, I sincerely thank Dr. Ralph Deters, who supported me during my research with his patient guidance, encouragement, and advice. Secondly, my appreciation goes to Graduate Chair Dr. Julita Vassileva. Dr. Ralph and Dr. Julita provided the knowledge needed for the thesis. They also provided unwavering support during my personal challenges, including family loss, depression during the COVID-19 pandemic, and stress from my full-time on-campus work.

Furthermore, I would like to show appreciation to my committee members: Dr. Ralph Deters, Dr. Julita Vassileva, and Dr. Chris Zhang. I would also like to thank my Multi-User Adaptive Distributed Mobile and Ubiquitous Computing (MADMUC) Lab colleagues for their friendship and support. Personally, I would love to express my gratitude to my family members in Vietnam for their love and support, as well as my dear friend in Saskatoon, Maliha Mahbub, who always encouraged and supported me.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABAC	Attribute-Based Access Control
ACL	Access Control List
AMs	Attribute Managers
APIs	Application Programming Interfaces
CH	Context Handler
CSV	Comma-Separated Values
EAG	Endpoint Access Grant
EVM	Ethereum Virtual Machine
HTML	Hypertext Markup Language
HTTP	Hyper Text Transfer Protocol
MAC	Mandatory Access Control
MLS	Multilevel Security Policies
NPD	Named Protection Domain
PAP	Policy Administration Point
PBFT	Practical Byzantine Fault Tolerance
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PIP	Policy Information Points
PoS	Proof of Stake
PoW	Proof of Work
RBAC	Role-Based Access Control
SAML	Security Assertion Markup Language
SDK	Software Development Kit
UI	User Interface
VM	Virtual Machine
XACML	eXtensible Access Control Markup Language
YAML	Yet Another Markup Language

1. INTRODUCTION

APIs can be used to expose software, service, or data over a network or the Internet. While they enable clients to interact with these resources, they also introduce concerns related to control and security. When APIs are left publicly accessible without control mechanisms, API providers face challenges [1]:

- They cannot track which client apps access their APIs.
- They cannot enforce access rights and service level agreements.
- They cannot grant or revoke access on the fly.
- They cannot trace historical events or block malicious clients.

To address these issues, API providers may implement their own access control solutions, but this can lead to centralized control, limiting scalability and flexibility. For instance, Google Maps utilizes access tokens provided to client developers for accessing its APIs, with token management centralized within Google. This centralized approach has limitations. It lacks scalability, the access control data may change over time, and it may not suit the needs of smaller developers looking to implement and offer their own APIs. Therefore, there is a demand for a decentralized, distributed, reliable, auditable, and data immutable solution that supports robust API access control.

Blockchain is a technology that inherently meets those criteria. It has gotten significant attention and has been applied in various industries, including digital assets, healthcare, medical records, IoT, and supply chain management. In these sectors, it has been used for access control [2]. However, to the best of our knowledge, its potential for API access control has yet to be extensively explored. Thus, it is a promising candidate for building the solution with room for exploration.

This exploration involves several aspects. Firstly, like any technology, blockchain would introduce performance impacts including overhead latency. These impacts must be measured, and adjustments must be made to mitigate them. Secondly, the solution's scalability can be explored, allowing the system to accommodate increased traffic and serve more clients effectively. Thirdly, determining the system's capacity for concurrent client interactions under heavy load is essential. Fourthly, having traceability is also vital. It is necessary to determine whether the system provides mechanisms for inspecting events that have occurred within the system.

This thesis aims to develop an access control system using blockchain for managing API access. The rest of the thesis is organized as follows:

- Chapter 2 defines the problems.
- Chapter 3 reviews the related knowledge, technologies, and research work.
- Chapter 4 presents the architecture, data model, design choices, and sequence diagram.
- Chapter 5 presents the technology stack of the actual implementation, code snippets, and user interface screenshots.
- Chapter 6 presents the experiments that evaluate the performance of the proposed solution.
- Chapter 7 concludes the contributions of this thesis and discusses future work.

2. PROBLEM DEFINITION

Traditional access control systems have certain drawbacks. They are centralized and have a single point of failure because a central authority manages all access decisions. The access control data stored in these systems can be modified by malicious users or system administrators. Moreover, they may suffer from low scalability and lack a comprehensive audit trail. The absence of an audit trail can impact monitoring and accountability, making it challenging to detect and address potential security breaches or unauthorized access attempts [3]. When multiple API providers implement a traditional access control system to manage their APIs, the system might resemble the following diagram:

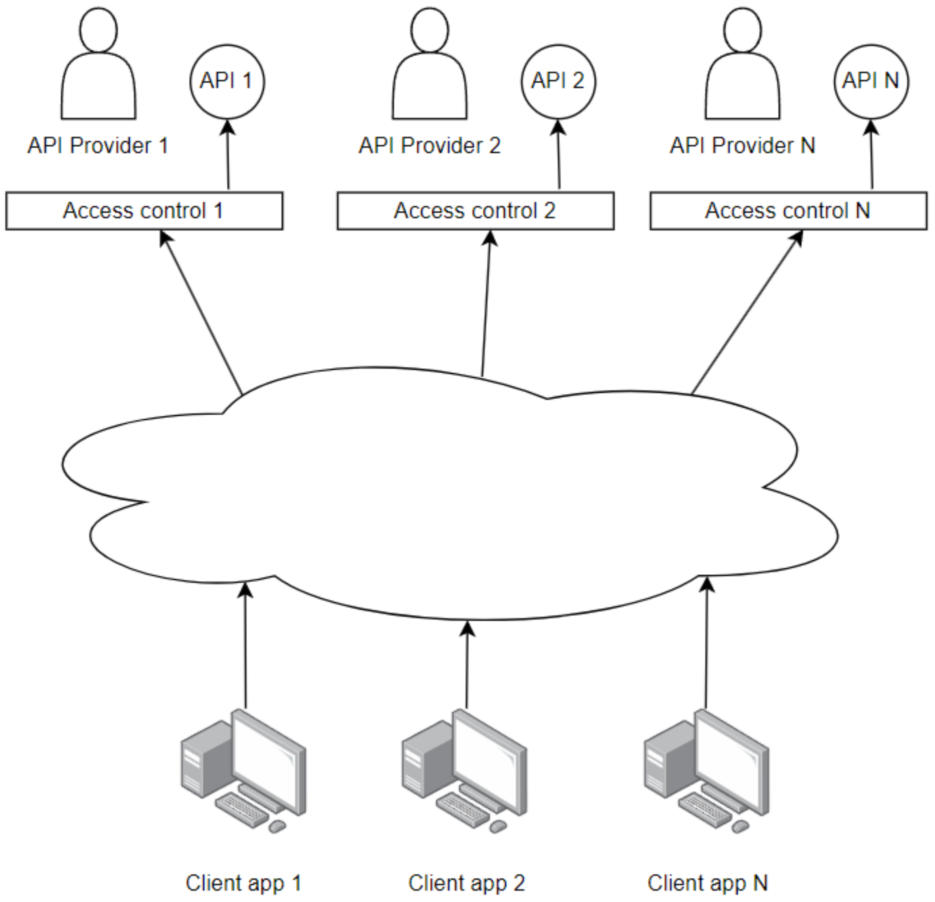


Figure 2-1 Traditional API Access Control

The diagram illustrates various API providers exposing different APIs. An API can contain multiple API endpoints. The client apps, which can be smartphone apps, web browsers, or back-end apps, represents the entities that invokes these API endpoints. In this context, a client app can be seen as accessing the APIs as a resource. The API providers implement access control solutions to manage and control which client apps can access their APIs. In this case, access control is centralized at each API provider, potentially creating a single point of failure. If the access control server for a specific API provider fails, the corresponding client apps cannot access their APIs.

A blockchain-based solution, on the other hand, can address the limitations and look like the following:

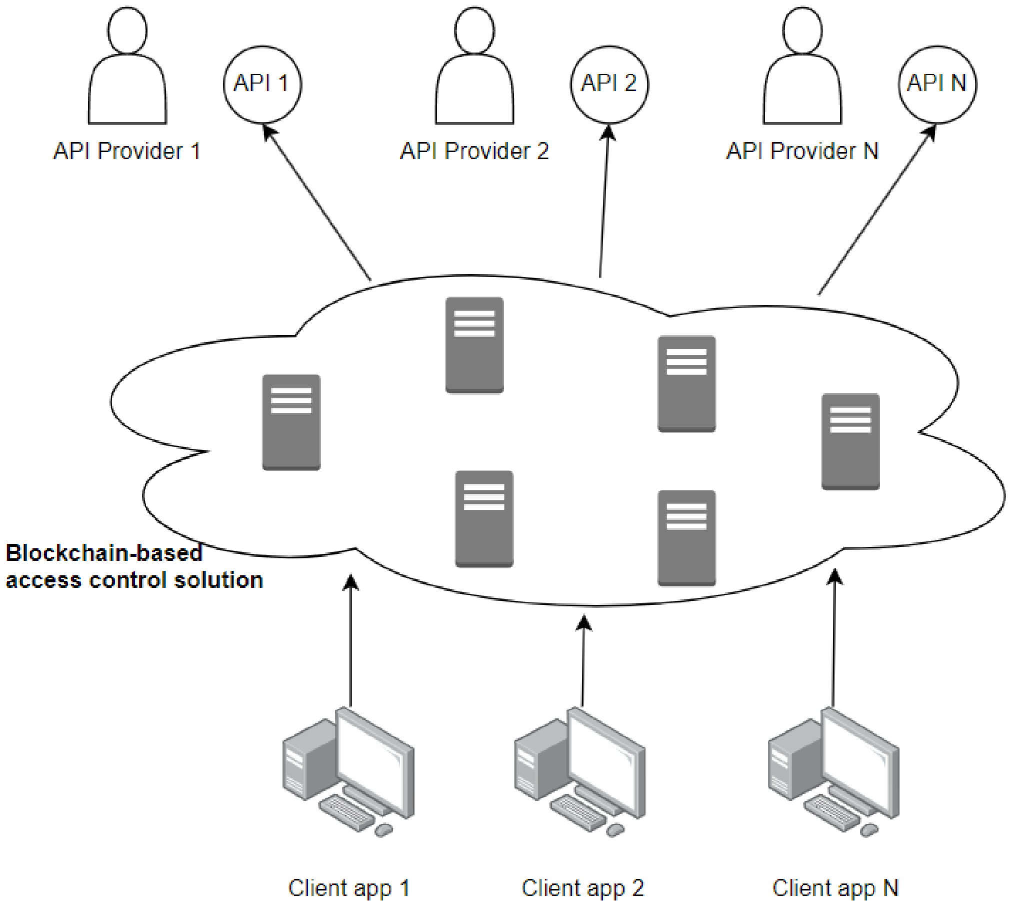


Figure 2-2 Blockchain-based API Access Control

A blockchain is a network of peer nodes. Thanks to its decentralized and tamper-resistant nature, blockchain eliminates the need for a central authority, mitigating the risk of single points of failure and enhancing overall security. The diagram shows that each API provider no longer relies on the individual node that handles access control. Instead, the providers make use of the network of multiple nodes. Moreover, blockchain's transparent and immutable ledger provides a comprehensive audit trail of all access control activities. Every access request and permission change are recorded in an unchangeable way, promoting accountability and trust among authorized parties. Thus, blockchain can be an intriguing option for API access control.

2.1 Research Objectives

The primary objective of this thesis is to develop a functional, full-stack access control system using blockchain for managing API endpoints access, with a straightforward architecture and web user interface. Subsequently, through performance evaluation experiments on the implementation, the following research questions will be addressed:

Research Question 1: What is the system's overhead latency in milliseconds when transitioning from not using the proposed solution to using it?

Understanding the impact of the proposed solution on latency is essential for system administrators, API providers, and client app developers. This question aims to quantify the increase in round-trip time when the proposed solution is used compared to its absence.

Research Question 2: What is the specific latency impact, in milliseconds, of the chosen blockchain technology?

After evaluating the overall system latency for research question 1, isolating and measuring the specific impact of the chosen blockchain technology on API response times is crucial. This question determines how much longer API responses take when enabling blockchain technology into the system.

Research Question 3: What is the system's maximum concurrent client app capacity?

Identifying the system's maximum load capacity is critical to prevent errors or timeouts when numerous client apps invoke their respective APIs. This question aims to provide a clear threshold for the number of clients the proposed solution can handle before it overloads and breaks down.

3. LITERATURE REVIEW

This chapter presents the background for building a blockchain-based API access control system. First, section 3.1 discusses the evolution of blockchain and its main concepts. Second, section 3.2 discusses Hyperledger Fabric - the specific blockchain technology used for the implementation. Then, section 3.3 presents access control concepts, including some blockchain-based access control-related works. Finally, section 3.4 covers an overview of API management.

3.1 Blockchain

This section introduces the high-level overview of blockchain using separate layers of concerns. The subsections will elaborate on each layer.

The evolution of blockchain started with the introduction of Bitcoin or blockchain 1.0, a digital currency and a technology stack. The term Bitcoin refers to components in the stack [4]:

- The underlying blockchain platform (blockchain layer).
- The Bitcoin protocol (protocol layer).
- The Bitcoin wallet, and the Bitcoin currency itself (currency layer).

The following diagram illustrates the stack:

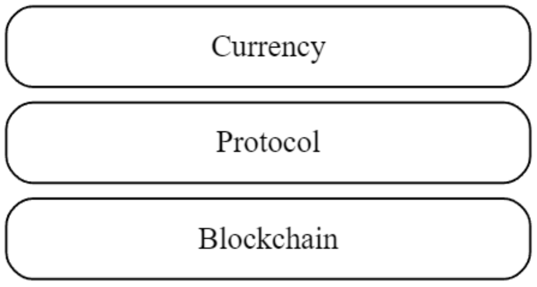


Figure 3-1 Blockchain 1.0 (Bitcoin technology stack)

Bitcoin aims to be a decentralized and trustless alternative to traditional currency. To do that, it must eliminate the need for a central authority or intermediaries and solve the double-spending problem. The blockchain platform and protocol layers work together to achieve those goals. Furthermore, thanks to the core capability of the first blockchain implementation that allows two arbitrary clients on the Internet to conduct any transaction, there were efforts to tackle more use cases besides money and payment. However, overall, they were still limited to buying and trading [4].

The next wave of blockchain came with new protocols and the smart contracts layer. More areas and markets other than currency, like banking, crowdfunding, and smart property, joined the decentralization movement. New protocols ran on the previous blockchain (Bitcoin blockchain) or new separate blockchain platforms. Anything above the protocol layer can be considered the application layer at this stage. At the application layer, anything can be on a blockchain, from the physical world to intangible and figurative things. Moreover, once they are modelled as digital assets on the blockchain, access control can be enforced on them [4].

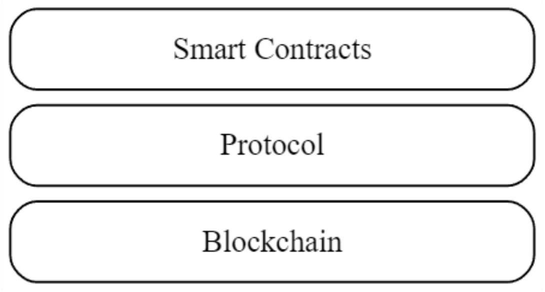


Figure 3-2 Blockchain 2.0 Technology Stack

A large ecosystem has grown around the blockchain technology stack, facilitating sophisticated blockchain-based software systems [4].

3.1.1 The Blockchain Platform Layer

The blockchain platform layer is the first layer of a blockchain stack and is responsible for decentralization and data integrity. The blockchain platform is a peer-to-peer network of nodes. Each node is a machine holding the same ledger. Each ledger contains a chain of blocks. Each block is cryptographically linked to the previous block in the chain through a hash function. Any form of data can be put inside a block's content. The rest of a block (the header) contains metadata that facilitates adding new blocks to the chain in a linear, chronological order, forming an unbroken and tamper-proof record of all transactions [2]. The following diagram describes a basic block in a ledger:

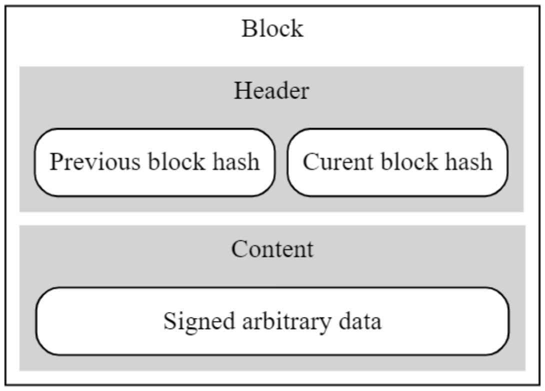


Figure 3-3 Anatomy of a Block

When a node needs to put data into a new block, it would also expect to receive the digital signature, meaning that the data creator has signed the data with their private key. The digital signature and the data are included in the block content, which gets hashed, and the resulting hash (current block hash) is included in the block header. The nature of hashing ensures that even the slightest modification to the data will result in a completely different hash value. The block header also contains the previous block's hash value as a reference. This way, we have two blocks chained together. Any tampering with the content of the previous block would produce a different hash value. During the verification process, one can detect that the previous block hash reference on the new block does not match the actual hash value of the content of the previous block. When the ledger's first block (genesis block) gets created, it does not have the previous block hash in its header [2].

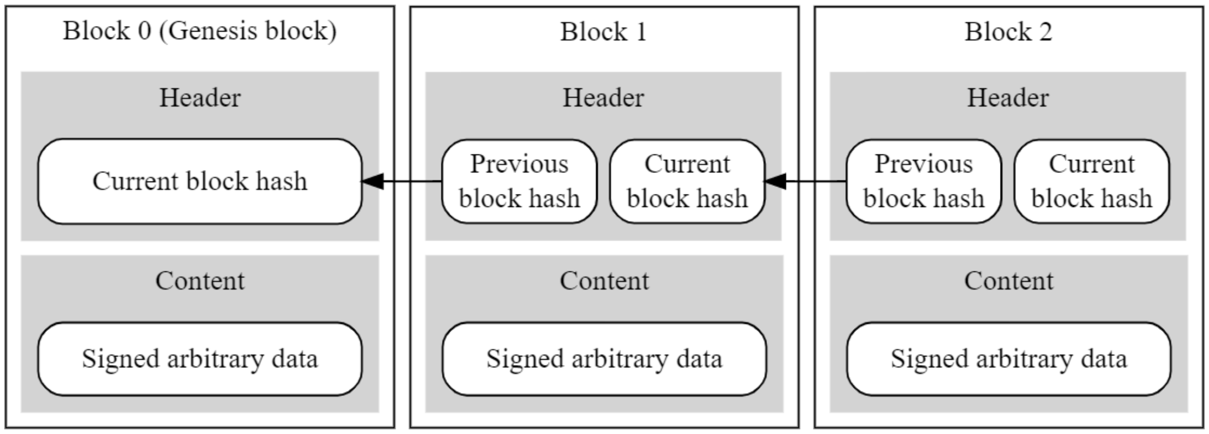


Figure 3-4 Anatomy of a Blockchain

Depending on the protocol that runs on top of the blockchain, each block header can contain more metadata, and each block content can have the corresponding data type. For instance, in the case of Bitcoin, the block header can also have a timestamp and a nonce for consensus usage. And the block content contains a set of transactions [2]

The protocol layer also regulates which node can add new blocks to the network. A new block gets broadcasted to all other nodes when it is added. All network nodes independently follow and enforce whichever rules the protocol layer regulates to ensure the new block is valid before adding it to their own ledgers. Doing so allows each node to operate autonomously without a central controlling authority, effectively addressing the centralization problem. How each node handles a new block and maintains its ledger also enforces data integrity [2].

3.1.2 The Protocol Layer - Consensus

The protocol layer is the second layer of a blockchain stack. It presents a consensus mechanism and networking rules, defining the overall goal of the stack. In a distributed network full of untrustworthy nodes and avoids using central authorities, the consensus algorithm forces all nodes to agree on a universal ledger state to preserve data consistency. The choice of which consensus mechanism to use depends on the decentralization, security, and scalability requirements. There must be a balance between those criteria, and no perfect algorithm solves everything [2].

The Bitcoin protocol uses the proof of work (PoW) consensus mechanism. The nodes with sufficient hardware can join this process and are called miners. When clients broadcast data, i.e., Bitcoin transactions, to the network, each miner receives and puts the data into a block content. The protocol then picks a random number. This number is called a nonce. The nonce and the block content are then hashed. Suppose the hash value does not comply with a particular network rule, such as having a certain number of leading zeros. In that case, the nonce is incremented or re-generated, and the miner starts hashing again. This repeating step is called solving a mathematical puzzle and requires significant computational power. When the miner finds the correct hash, it assembles the block and broadcasts the result to the network. Once other nodes verify the block and the transactions inside, all nodes append the block to their ledger, and the miner who found the solution is rewarded with a certain amount of cryptocurrency [2].

In a distributed network with no central authority and clients who can submit data to any node, a malicious user can submit two transactions that spend the exact balance to two different nodes, effectively making a double-spending attack. However, with PoW, to successfully carry out a double spending attack, the malicious user would need to modify the first transaction so that it appears the money has not been spent. Achieving this is extremely difficult because of the nature of blockchain, which guarantees that it is immutable after a block has been added. Moreover, changing one block would require re-calculating all subsequent blocks across, and that must be done to most of the network nodes so that the new forked ledger is not rejected [5].

An alternative consensus mechanism to PoW that does not require significant computational resources and is more energy-saving is Proof of Stake (PoS). In PoS, the nodes with the most currency, i.e., hold more stake, are prioritized when the network needs to select a node to commit a new block, which is rational because the more stake a node holds, the less likely it is malicious. Many protocols, such as Ethereum, started with PoW first to make sure the network has enough stakeholders, then gradually switched to PoS [2]

Another energy-efficient consensus mechanism is Practical Byzantine Fault Tolerance (PBFT). PBFT requires that the nodes in the network are known, and it contains three phases for adding a new block. First, a node is selected as a primary node, which proposes the block to other nodes. Then, the other nodes validate the proposal and communicate with each other to confirm that they all have received it. Finally, they communicate to confirm the agreement and finish adding the block to the network [2].

There are many other consensus mechanisms. One of the ones that Hyperledger Fabric can be configured to use is called Raft.

3.1.3 Types of Blockchains

Bitcoin is a public blockchain and the first one. Public blockchain refers to the blockchain platforms where each machine hosting a network node is publicly available. Any machine on the internet can join and become a node of a public blockchain. Other than Bitcoin, Ethereum is another decent public blockchain. If Bitcoin started blockchain 1.0, Ethereum was one of the pioneers of blockchain 2.0, finishing off Satoshi's plan with Smart Contracts, turning the whole stack into a foundational general-purpose cryptocurrency platform. Ethereum refers to the entire stack, blockchain, protocol, and currency. The Ethereum blockchain itself is also public [4]. Regarding the consensus mechanism, public blockchains allow all nodes to join the process, but it is typically slow [2].

On the other hand, private blockchains only allow registered and granted nodes to operate within the network and have a membership system that only allows specific clients to submit transactions. Private blockchains also implement consensus mechanisms just like public blockchains. However, since there are significantly fewer nodes, private blockchains can utilize more lightweight and resource-friendly consensus algorithms to process transactions faster. Hyperledger Fabric is one of the private blockchains [2].

Depending on the purpose and requirements, one can choose between the main types of blockchains. For instance, if low latency and high load capacity are prioritized, a private blockchain can be selected, and vice versa.

3.1.4 The Smart Contract Layer

The smart contract layer is the top layer in a blockchain stack, above the protocol layer. According to [4], as seen in Bitcoin, smart contracts were introduced in the blockchain 2.0 wave to enable more complicated and programmable instructions than simply buying and trading. The word "contract" emphasizes establishing rules that multiple parties must adhere to complete a transaction successfully. For instance, a bus driver has a "contract" with a school principal about driving a group of students around the city for sightseeing. The driver will only get paid if he adheres to specific criteria outlined in the contract. These criteria may include maintaining detailed records such as the date and time of student pick-up, the number of students transported, the time of student drop-off back at the school, etc. There may be no trust between the driver and principal, but the contract protects them, and the law will punish either of them if the contract is violated. That is how things would go with a traditional "contract." In a smart contract context, the driver and principal can use a blockchain-based app and a smart contract to automate and enforce the agreedupon terms. When the driver fulfills his obligations and meets the specified conditions, such as recording the required information accurately on the app, the smart contract automatically triggers the fund transfer. As a result, the driver's currency balance increases, reflecting his payment, while the school's currency balance decreases accordingly.

There are more characteristics of smart contracts, such as autonomy, meaning when they are deployed, they act on their own, get triggered automatically when some conditions are met, and should be able to manage the resources they need appropriately. Smart contracts are also supposed to be deployed on all peers in a blockchain network instead of one node, i.e., they are decentralized. Moreover, since they are distributed to many nodes and automated, they should also be deterministic. After all smart contracts have finished executing their programmed logic, all nodes should be in the same state, which will be verified by the protocol layer [4].

3.1.5 Wallet

In a blockchain, a wallet is software that runs on the client side and holds the following purposes:

- A unique address for identifying the user.
- A private key for signing data and transactions.
- The currency balances.

Due to the decentralized autonomy, no other than the clients manage their wallets. There are no trusted intermediaries that can help recover lost passwords or accounts. If the wallet is gone, the user loses access to all data on the blockchain [4].

3.2 Hyperledger Fabric

Hyperledger Fabric (or Fabric) is an open-source private blockchain technology from the Linux Foundation and the first blockchain that supports developing distributed apps in common programming languages [6]. Hyperledger Fabric has been used in many production use cases across different industries and for implementing the proposed solution in this paper. This section discusses how it stands out among blockchain technologies and its components.

3.2.1 The execute-order-validate pattern

Blockchain platforms before Fabric, whether private or public, follow the order-execute pattern. This pattern means the consensus protocol of the network organizes the transactions to a specific order, and then each peer processes the transactions in that order. It is simple but has many disadvantages [6]:

- The consensus mechanism cannot be changed after it has been deployed.
- The trust model cannot be changed, e.g., an order-execute network cannot switch from assuming peers to be trustless and potentially malicious to assuming peers to be trusted.
- A domain-specific language must be designed for developers to write smart contracts instead of letting them use common and stable programming languages.
- The throughput of the network is limited.
- The network is vulnerable to denial-of-service attacks, such as a smart contract that executes deliberately slow.
- All transactions must be deterministic so that all peers end up in the same state after processing them. It is difficult for smart contract developers to write such code, and the resulting code can be limited in functionality or efficiency.
- Each peer must run every smart contract, which can affect confidentiality.

Hyperledger Fabric, on the other hand, follows the three-phase execute-order-validate pattern instead. A Hyperledger Fabric application developer should be aware of the pattern and understand that due to the pattern, there are two parts of the application [6]:

- The chaincode part: the smart contract, and it runs in the execute phase.
- The endorsement policy part: select the peers to run the chaincode, e.g., half of the peers, peers A and B only, etc. Only the system administrators can set this policy, and the peers specified by the policy are called the endorsers.

After the application has been written and the endorsement policy has been in place, the three phases work as follows, starting with the execution phase [6]:

- The endorsers receive a transaction proposal from a client.
- Each endorser executes the proposal against its ledger without worrying about the consensus, i.e., it only simulates the execution, resulting in a readset and writeset (rw-sets). For example, the client submitting a transaction about "Alice transfer Bob $20" can result in a readset of [(key: 'alice', version: 1), (key: 'bob', version: 1)], and a writeset of [(key: 'alice', value: $80), (key: 'bob', value: $70)].
- The endorsers then send the rw-sets back to the client.
- The client creates a complete transaction and sends it to the ordering service.

At this stage, the ordering phase takes place [6]:

- The ordering service sorts the transactions it has received into blocks.
- The blocks are then broadcasted to the peers.

Then, for each block, the validation phase takes place, containing three steps [6]:

- Other than the chaincode that the application developers write, Fabric also has its system chaincodes. One of them is the validation system chaincode that runs here and checks the endorsements against the endorsement policy. If it detects anything invalid, the corresponding transaction is ignored.
- Rw-sets are then validated. When a peer checks the readsets' keys, it checks their versions against its ledger. If there is a conflict, the transaction is ignored.
- Each peer then adds the block to its ledger, stores the result of the validation phase, and applies the writesets to its ledger.

3.2.2 System Components

This section describes the main components within a Hyperledger Fabric network. Being a private blockchain, it makes sense for Hyperledger Fabric to have a *Membership Service.* Clients and peers in Fabric can be grouped into *nodes*, and the Membership Service issues and keeps track of their identities, credentials, and certificates. Then the *Ordering Service* is needed for the Order phase in the execute-order-validate pattern. It also manages the channels and can reconfigure them. Then, we have *Peer Gossip*, a protocol implemented by each peer and responsible for broadcasting information among the peers. Then there is the *Ledger*, which persists the state of the blockchain network and uses a simple format under the hood: tuples of (key, value, version), i.e., a versioned key-value data structure in each peer. Finally, there is the *Chaincode Execution Environment.* For each peer, the application chaincodes run inside a Docker container so that it is isolated and easy to manage, and the system chaincodes run directly on the peer host [6].

3.2.3 Consensus

Fabric offers multiple consensus protocols including PBFT, Kafka, and Solo. However, the protocol that it recommends is Raft. When a group of peers follow Raft to reach a consensus, they elect a leader, and each of them can be in only one of the three states: Follower, Candidate, or Leader [7]. The following diagram shows how the peers switch between the states:

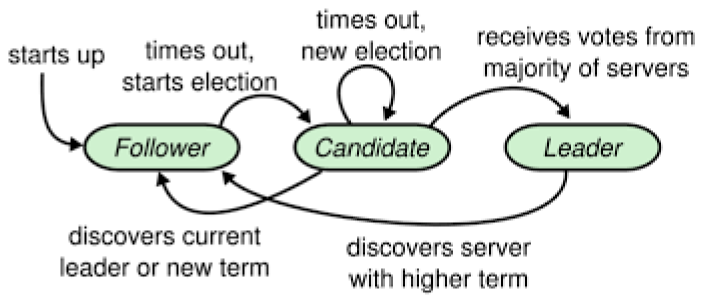

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 3-5 The three stages of the Raft consensus mechanism

As shown in the diagram [8], all peers start as followers. After not receiving anything from a leader peer for a while, they switch to the candidate state and vote for a leader. The next leader is the one that has the highest vote, and if it detects another leader with a higher term, it goes back to being a follower.

3.3 Access Control

This section discusses access control concepts since access control is needed in the proposed solution for managing access to API endpoints. An access control system is a facility that directly regulates and limits how valid clients, whether they are users or applications on behalf of users, access protected resources and functionalities of a software system. Software that does not implement access control gives clients direct access to its resources. If those clients are malicious, they can abuse the software, cause security breaches, and compromise everything [9].

There are many access control types. Different layers of concern can be used to understand them [10]:

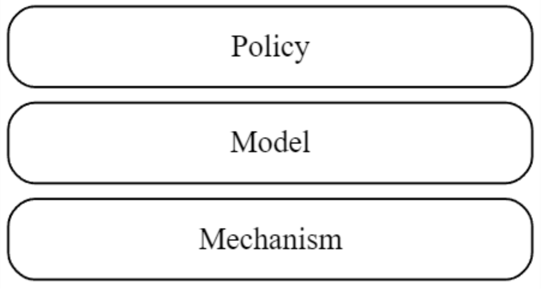


Figure 3-6 Access control layers

The diagram shows that an access control type typically has three layers. At the top, we have the policy layer for defining high-level rules. Then, there is the model layer for providing a formal representation. Thanks to the formality, the access control type can be reasoned about, analyzed, evaluated, and proved that it can provide the expected level of security. Then, we have the mechanism layer at the bottom referring to the software and hardware implementation of the high-level rules [10].

There are four common access control types:

- Discretionary Access Control (DAC)
- Mandatory Access Control (MAC)
- Role-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC)

The following subsections use the 3-layer policy-model-mechanism format to explain them. It is worth noting that one of DAC's mechanisms, *capability-based mechanism*, is related to blockchain technology. This connection will be elaborated upon in the DAC's mechanisms subsection.

3.3.1 Discretionary Access Control (DAC)

3.3.1.1 Policy

The policy layer of the Discretionary Access Control (DAC) type enforces explicit access control rules associated with the requestor's identity. A rule typically follows the "who can do what action, and on which resource?". A resource owner can grant permissions to users, and a user can share granted permissions with other users, hence the name "discretionary" [10].

3.3.1.2 Model

One of the standard DAC models is the Access Matrix model. As mentioned, a model presents formalization to reason about the system easily. In this case, the model presents the notion of a matrix. A triple of (S, O, A) indicates the state of the system, i.e. (Subjects, Objects, Access matrix). An access matrix is like a table whose rows represent subjects, columns represent objects, and each cell contains allowed actions. It is worth noticing that the model is built on the following essential elements [10]:

- Objects: what objects to protect?

- Subjects: what or who can operate on those objects?
- Actions (or rights): what actions or operations can be done on those objects?

The subjects, objects, and actions can vary depending on the context. In the context of this thesis, they can be but not limited to:

- Objects: the API endpoints
- Subjects: the client that sends HTTP requests to those endpoints
- Actions (or rights): the HTTP verbs that a client can send to and endpoints, e.g., GET, POST, PUT, DELETE

In addition to those elements, there are primitive operations that can change the state of the access matrix like [10]:

- create subject: adds a new subject with identity and attributes.
- delete subject: removes a subject with associated authorizations or attributes.
- create object: adds a new object with identity and attributes.
- delete object: removes an object with associated authorizations or attributes.
- enter action (or assign a subject to an object): grants a subject authorization to access an object.
- delete action (or remove subject from object): revokes a subject's authorization to access an object.

Within the context of this thesis, an access matrix for controlling access to API Endpoints can look like the following.

Table 3-1 An access control matrix example

	Endpoint 1	Endpoint 2	Endpoint 3
Provider X	OWN GET POST PUT DELETE	OWN GET POST PUT DELETE	OWN GET POST PUT DELETE
Client A	GET POST	GET	

	PUT DELETE		
Client B	GET	GET PUT	PUT
Client C	PUT		GET POST DELETE

The table presents three endpoint objects and four subjects (the provider and three clients) and their corresponding actions, e.g., the provider owns and is allowed all actions on the endpoints, while client B can only send GET requests to endpoint 1. From there, primitive operations can be used to build commands that can change the system's state. For instance:

```
command CREATE_ENDPOINT(api_provider, endpoint):
    create object endpoint
    enter action Own into A[api_provider, endpoint] end.

command GRANT_ENDPOINT(api_provider, client, endpoint, allowed_actions):
    if Own in A[api_provider, endpoint]
        then enter action allowed_actions into A[client, endpoint] end.
```

3.3.1.3 Mechanism

The Access Matrix model provides the notion of the matrix for more straightforward reasoning about the system. However, using a matrix for storing all subjects and objects and their access control actions may not be efficient in actual software implementation since it can waste resources and make writing and managing code difficult. Hence, there are different approaches to the Access Matrix model at the mechanism layer. One common approach is to use an authorization table, which puts non-empty cells of the matrix into a table of three columns (subject, action, object) [10]. For example:

Table 3-2 An authorization table example

Subject	Action	Object
API Provider X	Own	Endpoint 1
Client A	GET	Endpoint 1
Client A	POST	Endpoint 1
Client A	PUT	Endpoint 1
Client A	DELETE	Endpoint 1
…	…	…

Another approach is to use an access control list (ACL), which organizes rules according to the objects, meaning each object has an associated list of subjects and allowed actions, for instance:

Endpoint 1:

API Provider X: [Own, GET, POST, PUT, DELETE]

Client A: [GET, POST, PUT, DELETE]

Client B: [GET]

Client C: [PUT]

Endpoint 2: …

A third approach is the Capability mechanism, or Capability-based, which organizes rules according to the subjects, meaning each subject has an associated list of objects and allowed actions, for instance:

API Provider X:

Endpoint 1: [Own, POST, PUT, DELETE]

Endpoint 2: [Own, POST, PUT, DELETE]

Endpoint 3: [Own, POST, PUT, DELETE]

Client A:

Endpoint 1: [GET, POST, PUT, DELETE]

Endpoint 2: [GET]

Endpoint 3: [ ]

Client B: …

In a capability-based mechanism, after a subject is granted a set of privileges (e.g., API provider X is assigned a set of access rights to the three endpoints), the access control system can create a form of an access token that references the subject and the set of privileges. Then, when the user presents the token to the system, the user can use the privileges without having to authenticate that the user is the subject. In contrast, an ACL mechanism would require the authentication step [11].

Additionally, thanks to an access token, the capability-based mechanism shares similarities with blockchain technology, such as Bitcoin. In the case of Bitcoin, as long as a user possesses a cryptographic key, the user can access the digital currency associated with that key within the Bitcoin network [12].

3.3.2 Mandatory Access Control (MAC)

3.3.2.1 Policy

Mandatory Access Control (MAC) uses the following essential elements to build rules on [10]:

- Objects: passive entities storing information
- Users: human beings
- Subjects (not the same as subjects in DAC): programs or processes that operate on behalf of the users, i.e., active entities that request access to the objects.

In contrast to DAC, where permissions come from resource owners, in MAC, permissions come from the central system administrator, who assigns security labels to users, subjects, and objects. Other users cannot change the rules. Access is allowed or denied based on the security classification of the subject and object rather than on whether a specific action is authorized. There are many subtypes of MAC, such as Multilevel security policies (MLS).

In MLS, the security labels are classified into access classes. Each user, subject, and object is associated with an access class containing a security level and one or many categories and can be represented like this: (security level, categories). Security levels are hierarchically ordered. The categories, though, are not ordered. They only reflect their functional areas and facilitate finer-grained access classes. Additionally, the access class of a user is called a clearance [10].

3.3.2.2 Model

One of the fundamental formalizations of MAC models is lattice [10]. A lattice-based MAC model can look like the following example.

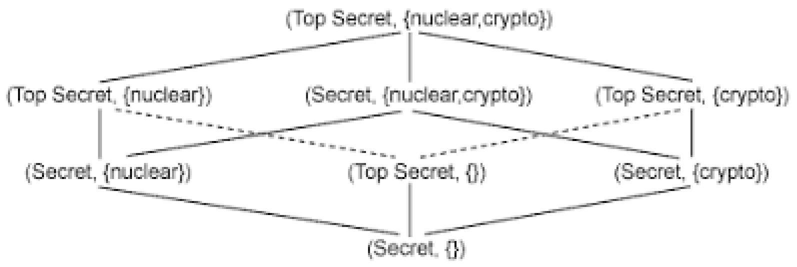


Figure 3-7 A lattice-based MAC model

The diagram shows Top Secret and Secret security levels; and None (i.e. {}), nuclear, and crypto categories. The top of the lattices represents a high level of sensitivity, and the bottom of the lattices represents a low level of sensitivity. Each point of the lattice is an access class that can be assigned to users, subjects, and objects in the system [10]. Suppose we have three users:

- Alice: (Secret, {Nuclear}).
- Bob: (Top Secret, {Nuclear, Crypto}).
- Charlie: (Secret, {}).

And three objects:

- A nuclear power plant, classified as Nuclear.
- A cryptography system, classified as Crypto.
- A public website, classified as None ({}).

Then, based on the lattices:

- Alice is authorized to access the nuclear power plant (because she has a clearance of Secret and is authorized to access Nuclear information). However, she is not authorized to access the cryptography system or the public website.
- Bob is authorized to access both the nuclear power plant (because he has a clearance of Top Secret and is authorized to access Nuclear information) and the cryptography system (because he has a clearance of Top Secret and is authorized to access Crypto information), but he is not authorized to access the public website.
- Charlie is authorized to access the public website (because he has a clearance of Secret and is authorized to access None information). However, he is not authorized to access the nuclear power plant or the cryptography system.

The set of access classes is partially ordered. In a partially ordered set, some elements can be compared, and some cannot be, meaning not all access classes can be compared because some access classes may have the same security level but different categories or the same categories but different security levels. However, when two access classes can be compared, they have a dominant relationship, e.g., the access class (Top Secret, {nuclear}) dominates the access class (Secret, {nuclear}) because it has a higher security level and the same categories. The dominance relationship also gives rise to the following properties [10]:

- Reflexivity: Every access class in the lattice is related to itself.
- Transitivity: If access class A is related to access class B and access class B is related to access class C, then access class A is related to access class C.
- Antisymmetry: If access class A is related to access class B and access class B is related to access class A, then A and B are the same access class.
- Existence of a least upper bound: Given any two access classes, A and B, a least upper bound access class C dominates both A and B.
- Existence of a greatest lower bound: Given any two access classes, A and B, a greatest lower bound access class C is dominated by both A and B.

Other MAC models build upon the lattice-based model to achieve further security goals. To achieve secrecy and integrity, two common models are Bell-LaPadula and Biba. Both models are about controlling the direct and indirect flow of data within the lattices. The Bell-LaPadula model defines the following principles [10]:

- No-read-up: A subject can read an object if the subject has a higher access class than the object's access class, i.e., the subject has higher clearance or trustworthiness, and its access class dominates the object's access class. This principle prevents lower-level subjects/objects from accessing sensitive data of the higher level.
- No-write-down: A subject can write to an object if the subject has a lower access class than the object's access class, i.e., the subject has lower clearance or trustworthiness, and the object's access class dominates its access class. This principle prevents sensitive data from flowing to lower-level subjects/objects.

The Biba model, on the other hand, defines the following principles [10]:

- No-read-down: a subject can read an object if the object has a higher access class than the subject’s.
- No-write-up: a subject can write to an object only if the subject has a lower access class than the object’s.

Secrecy is preserved by preventing the information flow from higher classes to lower classes (no-write-down). In contrast, integrity is preserved by preventing the information flow from lower classes to higher classes (no-write-up). If both characteristics must be enforced, each subject or object must have appropriate access classes for each characteristic.

3.3.2.3 Mechanism

The Bell-Padula model was initially designed to implement an access control mechanism for computer operating systems, and it makes the following assumptions [10]:

- Information is stored in objects (files)
- There are different levels of sensitivity to information.
- There are restrictions on who can access information at different levels.

The model and those assumptions fit the context of computer operating systems perfectly. Later, there were studies to expand the model for controlling database systems [10].

3.3.3 Role-based Access Control (RBAC)

3.3.3.1 Policy

There are use cases where neither DAC nor MAC can fit. However, Role-based Access Control (RBAC) can because it offers both DAC's flexible permission-granting ability and MAC's organizational constraints. When a user is granted permission to do something, the user has a privilege. RBAC is about putting a layer called "role" between the user and the privileges and appropriately grouping those privileges into roles. Furthermore, due to the notion of role, RBAC is about what a user means to the organization rather than the actual identity. The general steps are [10]:

- Define the roles.
- Assign rights or privileges to each role.
- Assign each user a role.
- Users can then access a service or resource through the role.

3.3.3.2 Model

Named protection domain (NPD) is a common model in RBAC. Since the main idea is to properly group privileges, in the NPD model, each task in a system is associated with a set of privileges needed to carry out the task [10]. For instance, a bank system can have a task "approve loan." An NPD called Loan_Approvable can be defined for the task, and it is associated with a bunch of privileges:

- Can access customer credit history data.
- Can access customer income and employment data.
- Can view loan application forms and supporting documents.
- Can set loan terms and interest rates.
- Can request additional information or documentation from the customer.
- etc.

The NPD can then be assigned to a user or another NPD. The privileges, NPDs, and users together form a directed acyclic graph. The NPDs can also be referred to as roles [10].

The granularity of a role can vary from being specific, like the Loan_Approvable example, to being general, like Bank_Staff, Bank_Supervisor, Director, etc. After users have adopted roles, they can carry out any task or access any resource regulated by the roles. It is worth noticing that some DAC models can have a notion of "group," which groups users, while a role in RBAC groups privileges.

There are some advantages to having the notion of role between users and privileges. Firstly, the management process becomes simpler because system admins assign privileges to roles and then roles to users instead of assigning privileges directly to users. Given an organization, when a new user or an application is added, the admin must assign a role corresponding to the user or application's responsibilities. Similarly, when a user quits, or an application is removed, the admin must unassign the role. Secondly, roles can form a hierarchy that naturally fits an organization. If a user is assigned a specific role, they may also be allowed to use all the "sub-roles" associated with that role. The user can do more things without asking for permission for each action. Thirdly, when a role hierarchy is appropriately built in an organization, the admin can enforce separation of duties and avoid giving a user too many privileges that can damage the system should the user turn malicious. Moreover, besides containing privileges, a role can also contain constraints [10]. For example, staff in a bank can activate the Loan_Approvable role, but the admin can limit the role so that the staff can only approve two loans a day.

3.3.4 Attribute-based Access Control (ABAC)

3.3.4.1 Policy

Attribute-based Access Control (ABAC) is comparable to RBAC. However, instead of introducing the notion of role between user and privileges, ABAC introduces the notion of attributes, which are any information or metadata associated with the subjects and objects. Furthermore, since those attributes can change in runtime, ABAC can be static or dynamic, making it versatile and fine-grained. The general steps that an ABAC policy defines are [13]:

- Define the subjects and objects.
- Define the objects’ descriptors. Each descriptor contains attributes of the corresponding object.
- Define the subjects’ descriptors. Each descriptor contains attributes of the related subject.

- Define the permissions. Each permission is an object descriptor associated with an operation.
- When a user requests access to a resource, the ABAC system checks the user's and resource descriptors. Determine if the user should be granted access based on the defined permissions.

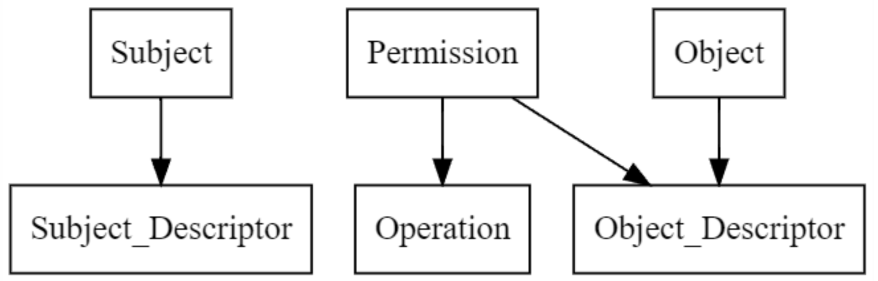


Figure 3-8 ABAC elements

3.3.4.2 Model

XACML is one of the ABAC models since it provides a formal language and a template architecture for representing and building an ABAC system. The architecture defines the following components [14]:

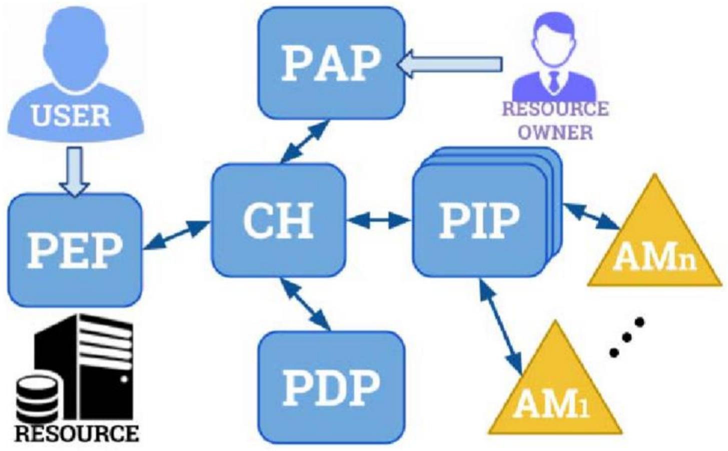


Figure 3-9 ABAC model architecture

- Policy Enforcement Point (PEP): processes access requests from the user and allows or denies access to the associated resource.
- Policy Administration Point (PAP): manages the access control rules.

- Attribute Managers (AMs): keep track of the subjects, resources, and environment attributes used to make access control decisions.
- Policy Information Points (PIPs): AMs are usually abstracted away, and the PIP is the interface to query or update the AMs.
- Policy Decision Point (PDP): evaluates and returns the result decision on whether access is allowed or denied given an access request, policy, and attributes.
- Context Handler (CH): orchestrates the system.

The architecture can be implemented as a distributed system where the components interact using HTTP requests or as a monolith system where the components are just classes invoking each other. That is up to the developer.

3.3.5 Related Works: Blockchain-based Access Control Mechanisms

3.3.5.1 RBAC mechanism

Cruz, Kaji, and Yanai [15] proposed a blockchain-based Role-Based Access Control solution called RBAC-SC. They pointed out that the role assignment entity may differ from the service provider or resource owner within some RBAC systems. In such a case, from the point of view of the role assigner, the service provider is inconsequential to it because what service is being provided does not affect the role-assigning process. On the other hand, when a user requests access to the service, the service provider must actively contact the role assigner for user role validation; failure to do so leads to a security breach in the system. The paper proposed a blockchain-based RBAC mechanism where the university (the role assigner) first submits a student role management smart contract to the blockchain. Then, when a student eats at a restaurant (the service provider), the restaurant queries the contract for the details, uses the details and follows a challenge-response protocol against the student's app to validate the student's role at the university. Ultimately, the restaurant does not have to contact the university directly. Instead, it accesses the published role data on the blockchain, i.e., the service provider gets what it needs without contacting the role assigner directly.

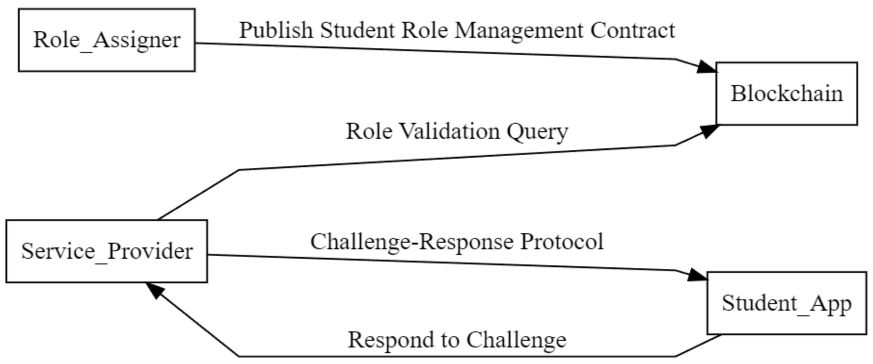


Figure 3-10 RBAC-SC mechanism design

The blockchain platform that the paper used was Ethereum. All the RBAC logic was implemented into an Ethereum smart contract, including the following functions [15]:

- addUser(): add a user to the blockchain and assign a role
- removeUser(): make a user invalid and revoke their role.
- addEndorsee(): allow a user to endorse another user by adding them to the blockchain, e.g., a student endorsing another student.
- removeEndorsee(): allow a user to remove an endorsee from the blockchain.
- changeStatus(): deactivate the smart contract and indicate that it should not be used anymore.

RBAC-SC, as presented, focuses on access control between a university and students accessing services at a restaurant. It has a limited scope with moderate traffic. And its access control decisions are infrequent, occurring mainly when students access restaurant services. On the other hand, an API access control system deals with a broader use case of regulating access to APIs offered by providers, potentially handling higher volumes of requests and with the need for real-time evaluation of access permissions for every incoming request.

3.3.5.2 ABAC mechanism

Maesa, Mori, and Ricci [14] followed the XACML model to build a blockchain-based ABAC mechanism. The components mentioned in the XACML reference architecture were implemented as follows:

- PEP: a Java component that acts as a gateway between the subjects requesting access and the resource, providing an API for handling access requests and responses.
- PAP: a Java component where the resource owner submits an XACML file containing attributes and rules for each resource. The PAP component then parses the file into smart contract code and sends the contract to the CH. After the CH has deployed the contract to the blockchain, it returns the contract address to the PAP. The PAP then keeps track of the resource and its associated contract address.
- CH: the bridge between the PEP and PAP and the blockchain. When the CH receives the smart contract from PAP, it compiles the contract into EVM bytecode, which is supposed to execute on the Ethereum blockchain. The CH sends the bytecode to the blockchain using web3j and returns the resulting address to the PAP. When the CH receives an access request from the PEP, it extracts the resource ID from the request, fetches the corresponding smart contract address from the PAP, and then submits a transaction to the blockchain to execute the deployed smart contract where the evaluation of whether the access is allowed or denied takes place. The CH then forwards the result back to the PEP.
- PDP and PIP: in this mechanism, the PDP and PIP are logically incorporated into the smart contracts on the blockchain.

This mechanism can be extended to various use cases, including API access control. However, it may result in a complex system due to several factors. Firstly, the resource owner must have ABAC and XACML knowledge beyond simple grant and revocation actions. Moreover, smart contracts are not directly written, maintained, and tested. Instead, they are generated from the XACML policy files, and submitting multiple XACML policy files leads to deploying smart contracts on the blockchain. This complexity can present challenges for users and developers implementing the system.

3.3.6 Summary

The following table summarizes the presented access control types. It selects one that fits the specific use case of this thesis, which is an access control system for API endpoints that would result in:

- A straightforward system architecture, allowing for convenient experiment setup for extensive performance evaluation.
- An intuitive user interface, allowing the API providers to grant or revoke access at their will for each endpoint.
- A straightforward smart contract implementation.

Table 3-3 Access control types summary

Access control type	Description	Fit for the use case	Explanation
DAC	Access is based on the resource owner's discretion	Yes	DAC allows the API providers to have fine-grained control over access to their API endpoints. Moreover, working out a mechanism based on the access matrix model to implement into a smart contract would be straightforward. The architecture and user interface that come with the mechanism would also be manageable.
MAC	Access is based on predefined rules or policies that are independent of the resource owner's discretion.	No	Rules and security levels must be defined and assigned up front, leaving no flexibility. From the point of view of API providers, there is no granting and revoking. All access control rules are not up to the API providers.

RBAC	Access is based on the role of the user	No	Instead of focusing on granting or revoking access to the endpoints, the API provider would also have to group the clients into roles, which may not be straightforward and may make the smart contract implementation complex due to the role managing part.
ABAC	Access is based on attributes or characteristics of the user, resource, or context.	No	While ABAC fits the context of API access control, using ABAC may result in a complex system architecture. Moreover, granting and revoking can become less intuitive for the API providers because they must submit and manage more attributes for each endpoint and client.

3.4 API Management

Controlling access to API endpoints is part of a set of practices that facilitate the creation, publishing, and monitoring of APIs throughout their lifecycle, also known as API management. Since this thesis is about implementing access control for API endpoints, this section first gives an overview of API management and then goes into the details related to access control. According to De et al. [1], API management is a comprehensive solution that supports every step in the complete life cycle of an API suite, from when it is designed to when it is built, released, maintained, and retired. The critical component within an API management tool is the API gateway. It is a facade between the clients and the back-end services, an entry to the system. Moreover, it can be a single host or a group of hosts, i.e., a group of gateway services. The following diagram illustrates the API gateway and its internal logical layers.

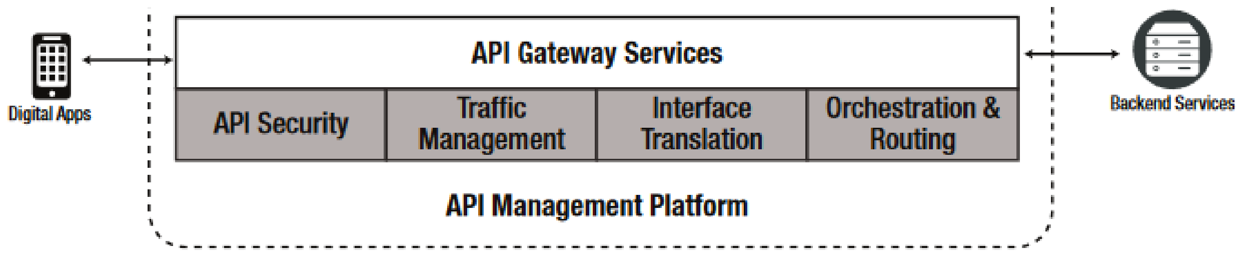


Figure 3-11 The API gateway within an API management platform

Here, De et al. [1] indicates that when a client sends a request to the gateway, the request must pass through multiple layers, including the API security layer for security enforcement. Its content can be validated and transformed to the appropriate format, and after that, it gets forwarded or routed to the appropriate back-end service. In addition, the gateway can limit the traffic and cache the response if needed to keep the load on the back-end services adequate. The gateway forms a stable communication link between the clients and the back-end services.

Regarding the API security layer, it can be broken into the following aspects [1]:

- Authentication: responsible for identifying the client's identity that makes the API call. In this case, the client is an app. The app can be a smartphone app from either of the app distributors, such as App Store and Google Play, a PC app, or a web app. The API gateway gives each app a name and a Universally Unique Identifier (UUID). The gateway also keeps track of the UUIDs and allows the system admin to revoke them if needed.
- Authorization: after identifying the app, the gateway must determine what resources or methods the app can access. Usually, the app or the developer fetches an access token from the gateway, and then the app will embed the token in requests. The access token is associated with the access privileges that the app has.
- Identity mediation: if the back-end services behind the gateway also require knowing the client's identity, the system can have a dedicated identity management system like SAML.
- Data privacy: any sensitive data should be encrypted or masked. If keys and certificates are used, they should also be managed appropriately.
- DoS protection: Denial-of-service (DoS) attacks are common for APIs, and the gateway should be able to protect them from those attacks.

- Threat detection: the gateway should be able to detect malicious requests, such as malformed formats, embedded scripts, etc. and handle them.

By concentrating on the access control of API endpoints, this thesis deals with the API security layer of the API gateway. There are generally three roles surrounding an API suite [1]:

- API providers: create, manage, and maintain APIs, making them available for consumption by API clients.
- API clients: consume APIs to build applications or services that rely on them for functionality or data.
- Product managers: responsible for strategically planning, designing, and implementing API programs. Their main goal is to ensure that the API program fits the organization's goals and meets the needs of both API providers and clients.

This thesis focuses on the API provider and API client roles. From the perspective of each role, an API management tool offers many functionalities, including:

- Authentication and authorization (or access control)
- API Discovery
- API Auditing, Logging and Analytics

3.5 Summary

This chapter shows the foundational concepts behind blockchain technology and Hyperledger Fabric, the blockchain that was chosen to implement the access control system proposed in this thesis. IBM backs Hyperledger Fabric, which has been used in many industries and companies, has a large community, and offers tooling options that can facilitate building and testing blockchain-based system applications. Hyperledger Fabric stands out from other blockchain technologies thanks to using the execute-order-validate pattern instead of the order-execute pattern. Furthermore, Hyperledger Fabric chaincode allows using familiar programming languages such as Java, Go, and JavaScript to develop smart contracts instead of requiring developers to learn and use some domain-specific language. However, Hyperledger Fabric has disadvantages and potential performance issues explored in the experiments chapter.

The various access control types are also discussed, and the Discretionary Access Control (DAC) type was chosen because it fits the API endpoint access control use case. It is rational to leave the access control decision to the resource owner (API provider) instead of the system administrators like in MAC or RBAC. Moreover, DAC can result in a straightforward system architecture, a flexible access control implementation, and a user-friendly front-end for the API providers.

API management is also discussed, highlighting the importance of having a gateway in front of any API system, and access control is part of the responsibilities of that gateway.

4. DESIGN AND ARCHITECTURE

This chapter will first present the overall system architecture for the API access control solution with the main components and their relationship with each other. Then, since the core idea is about storing and checking access control rules as data on the blockchain, the data model will be discussed with proper design so that the implementation is clean and maintainable. Thirdly, the design choices for managing the wallets for authentication and user management will be described. Finally, sequence diagrams will present the typical workflows and use cases.

4.1 Architecture

The following diagram illustrates the high-level overview of the components:

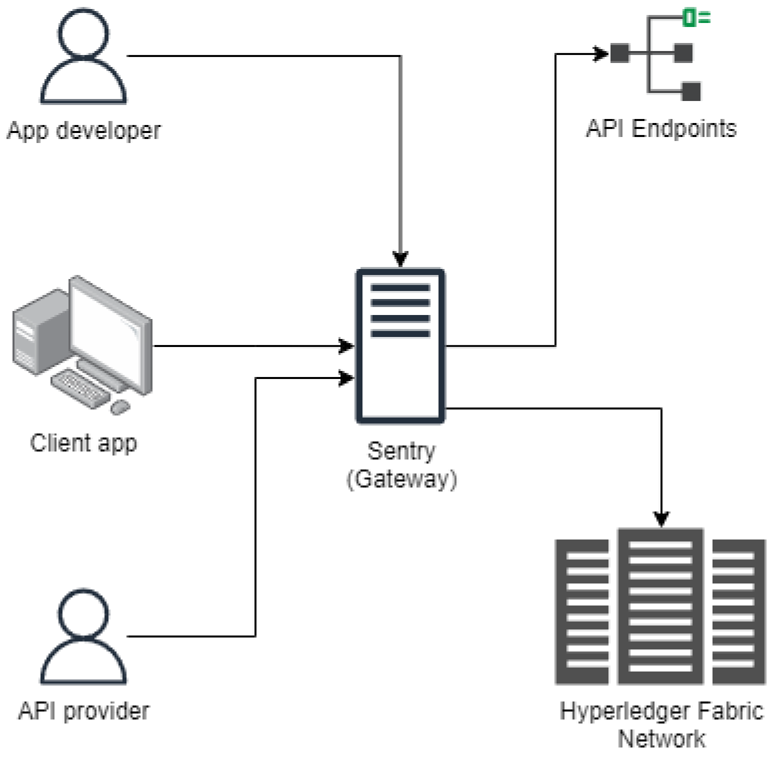


Figure 4-1 Proposed Architecture

Sentry is the component at the diagram's centre and the architecture's core. Sentry is the gateway that protects the API endpoints. API endpoints are owned and registered with Sentry by the API provider. The app developer uses Sentry to request access to the API endpoints, then embed necessary credentials and tokens into the client app so that the client app can access the API endpoints via Sentry. The client app and app developer are both forms of clients. Finally, Sentry stores and queries access control data from the Hyperledger Fabric network.

The architecture addresses the following key points:

- There must be a place to store and manage the access control rules; in this case, the Hyperledger Fabric blockchain network is responsible for that.
- The component forwarding requests to the protected API endpoints should be separate from the blockchain. Otherwise, if the forwarding logic is implemented on the blockchain, the peers would bombard the protected API points due to the nature of having a consensus mechanism, and the peers would bombard the protected API endpoints.
- The Hyperledger Fabric blockchain network manages the rules, and Sentry orchestrates the forwarding and other tasks.

4.2 Data Model Design

As mentioned in the architecture, the access control data is stored in the Hyperledger Fabric ledger, so the chaincode must be written to manage data properly. And since there are many entities that have a certain relationship to each other (i.e., client, provider, endpoint, etc.), an entity-relationship diagram can be used to design the data model for the system:

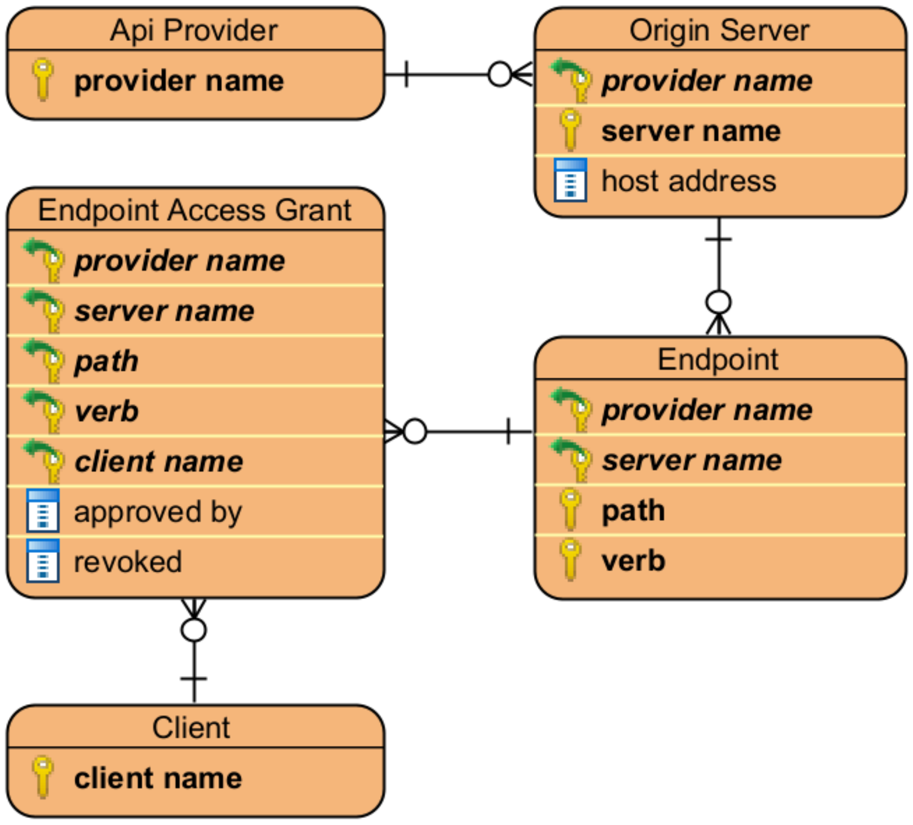


Figure 4-2 Entity-relationship diagram

The diagram presents the following entities:

Table 4-1 Entity-relationship diagram explanation

Entity	Explanation
API provider entity	This entity represents the API provider. A primary key identifies each API provider entity with the provider's name. Each provider can own multiple origin servers.
Client entity	This entity represents both the client app developer and the client app itself. The primary key is the client's name, which is formed by concatenating the app developer's name and the app's name.
Origin server entity	This entity represents a server machine that hosts the protected API endpoints. It is defined by a composite key consisting of the provider's and server's names. Each server entity stores the host address and can host multiple endpoints.

Endpoint entity	An API contains multiple endpoints. An endpoint is uniquely defined by a composite key consisting of the provider's name, server name, path (e.g., /ping), and verb (e.g., GET).
Endpoint Access Grant entity (EAG)	This entity represents that one client has been granted access to one endpoint. The relationship between the client and endpoint entities is initially many-to-many, meaning each client can access multiple endpoints, and each endpoint can be granted to multiple clients. To simplify this complex relationship, it is resolved through the EAG entity. Each client can possess multiple EAGs, while each EAG is associated with only one client. Similarly, each endpoint can be linked to multiple EAGs, with each EAG associated exclusively with one endpoint. Furthermore, each EAG contains an "approved by" field, which is initially null. When the API provider approves the EAG, the provider's name is put into the "approved by" field. Another field is "revoked," which is initially false. When an API provider revokes an EAG, the field is set to true.

4.3 Wallet Management

According to the system architecture, the API providers and clients interact with Sentry instead of directly with the blockchain network. Furthermore, it is essential to keep the users abstracted away from blockchain details when using the system to ensure a smooth user experience for them. Hence, a wallet management and user authentication mechanism are needed so that when Sentry needs to interact with the blockchain network on behalf of the user, it has the proper wallet to do so. That leads to two possible designs:

- Server-managed wallet: when new users register with Sentry, they can use regular usernames and passwords. Sentry then creates Fabric wallets and stores the users' credentials and corresponding wallets. It can use a simple file for storage, a separate database, etc. When a user signs in, Sentry verifies the credentials and retrieves the correct wallet for interacting with the blockchain.

- Client-managed wallet: only a username is needed when a new user registers. Sentry then creates a Fabric wallet, compresses it, encrypts it with symmetric encryption, and sends the encrypted wallet back to the user as a “password.” The secret key used for the symmetric encryption is stored safely at Sentry. When a user signs in, Sentry can retrieve the wallet by decrypting and decompressing the “password.” This “password” will be called an *encrypted wallet* in later chapters.

The client-managed wallet approach was chosen to keep things simple and avoid scalability overheads when having extra file storage and database involved.

4.4 Workflows

This section presents a sequence diagram of the key operations of the system.

4.4.1 Adding an endpoint

The following sequence diagram is for the use case where the API provider registers an endpoint.

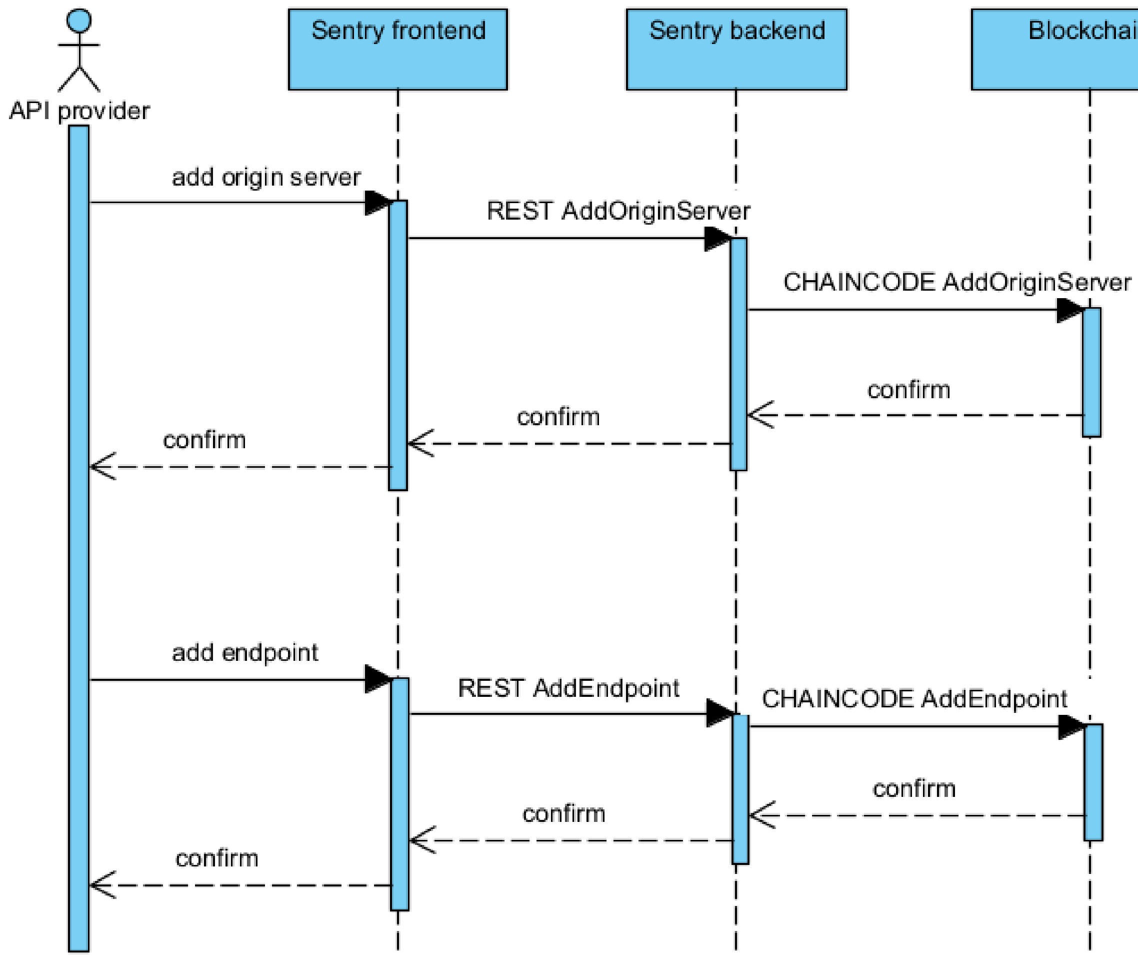


Figure 4-3 Sequence diagram: endpoint registration use case.

The diagram demonstrated the following workflow:

- The provider first uses the browser to submit a form at the Sentry front-end website.
- The front-end then sends an HTTP request to the AddOriginServer endpoint on the back-end.
- The back-end then invokes the AddOriginServer method of the chaincode deployed on the blockchain.
- After the origin server data has been added to all peers' ledgers, the blockchain returns a message to the back-end confirming that the data has been added.
- The back-end sends an HTTP response to the front-end, confirming that the origin server has been added.

- The front-end reloads the page to confirm with the provider that the origin server has been added.

4.4.2 Requesting & Granting Access

The following sequence diagram is for the use case where the client app developer has discovered on the Sentry website about the endpoint that the provider previously registered on the Sentry website.

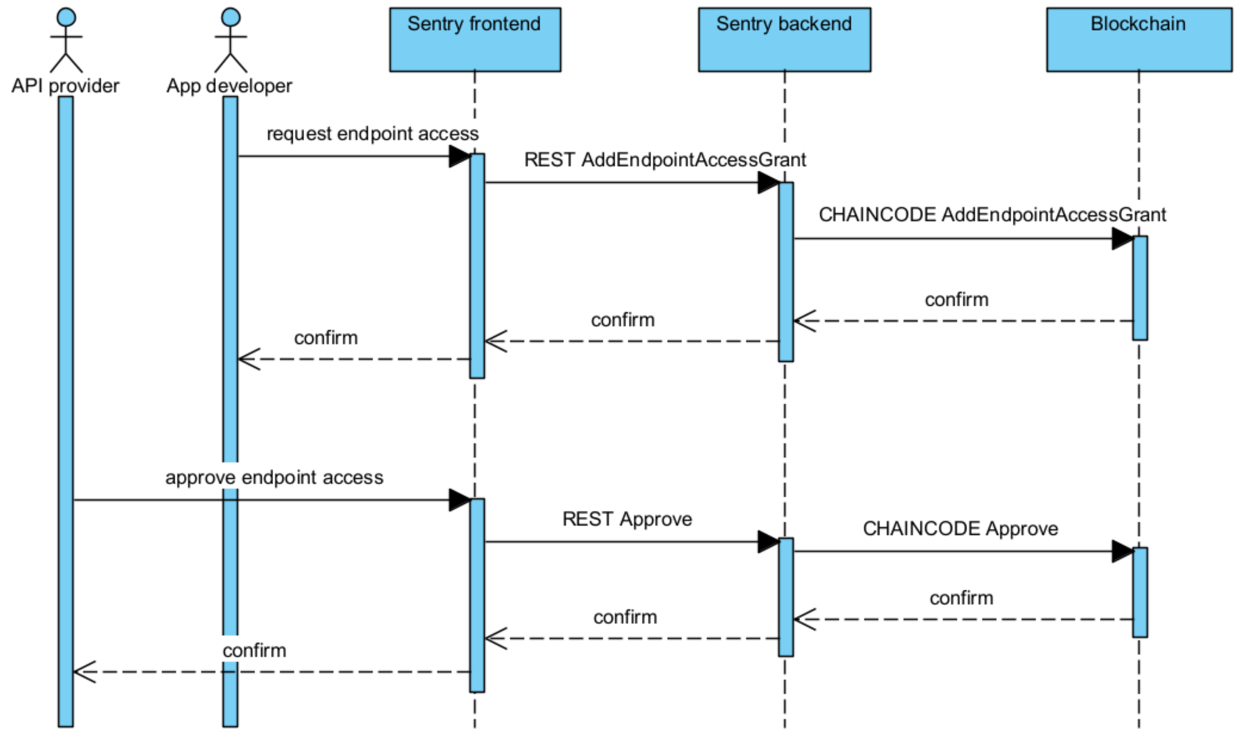


Figure 4-4 Sequence diagram: endpoint discovery use case.

The diagram demonstrated the following workflow, broken into two parts:

- The client app developer first clicks a button on Sentry front-end website to request access to an endpoint.
- The front-end sends an HTTP request to the AddEndpointAccessGrant endpoint on the back-end.
- The back-end then invokes the AddEndpointAccessGrant method of the chaincode deployed on the blockchain.

- After the data has been added to all peers' ledgers, the blockchain returns a message to the back-end confirming that the data has been added.
- The back-end sends an HTTP response to the front-end confirming that the access request to the endpoint has been added.
- The front-end reloads the page to confirm with the developer that the access request has been added and is waiting for approval from the provider.

After an access request has been added, the second part begins:

- On the provider's browser, an access request and the client app developer's information will appear. The provider then clicks a button to approve the request.
- The front-end then sends an HTTP request to the Approve endpoint on the back-end.
- The back-end then invokes the Approve method of the chaincode deployed on the blockchain.
- After the EAG grant object has been updated and saved to all peers' ledgers, the blockchain returns a message to the back-end confirming that the approval has been done.
- The back-end sends an HTTP response to the front-end confirming that the access request has been approved.
- The front-end reloads the page to confirm with the provider that the access request approval was successful.

When the client app developer reloads the page, the approved endpoint should show the ID of the EAG.

4.4.3 Accessing Endpoint

After the client app developer has been granted access to an endpoint, The developer then embeds the encrypted wallet and the EAG ID into the client app so that the client app can call the protected endpoint. The following sequence diagram describes the use case where the client app invokes the endpoint.

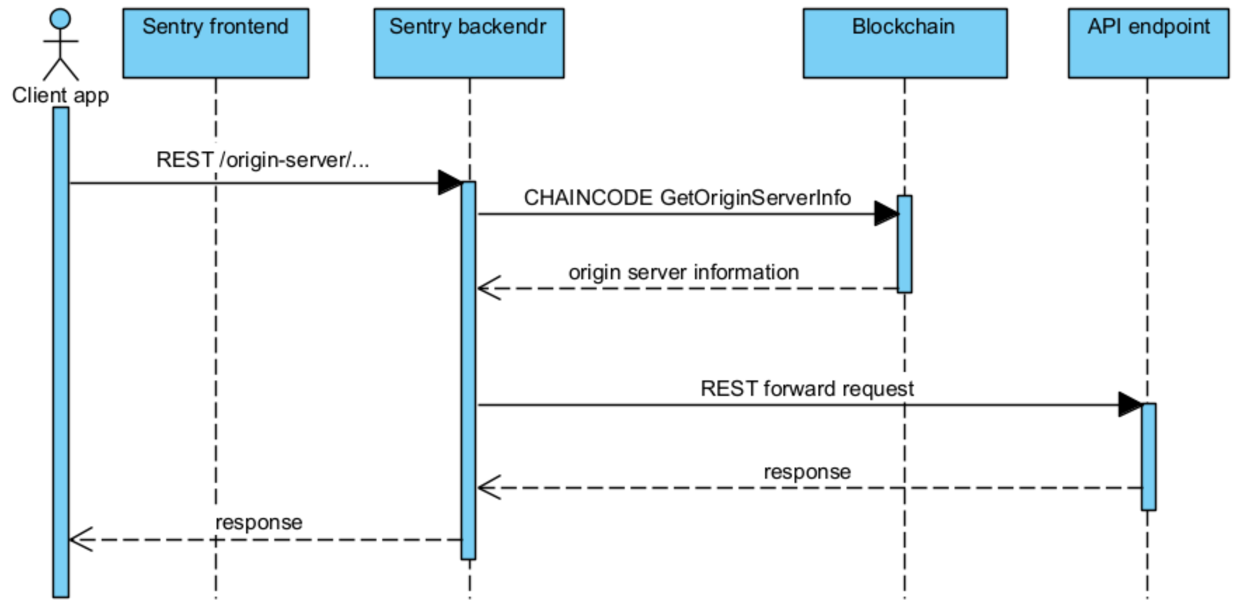


Figure 4-5 Sequence diagram: endpoint invocation use case.

The diagram demonstrated the following workflow:

- The client app attaches the encrypted wallet and EAG ID into a request's header of a request and sends it to the Sentry back-end.
- The back-end then decrypts the wallet and uses it together with the EAG ID to call the GetOriginServerInfo method of the chaincode deployed on the blockchain.
- Thanks to the wallet, the blockchain knows which client is trying to access the protected API. And thanks to the EAG ID, the blockchain can retrieve the endpoint access grant record and check if the provider has approved it. If the access is valid, the blockchain returns the origin server data.
- From the origin server data, the back-end figures out the exact address, and forwards the HTTP request of the client app to the destination.
- When the endpoint responds, the back-end forwards the response to the client app.

Throughout the process, the address of the origin server that hosts the protected API is hidden from the client.

5. IMPLEMENTATION

This chapter presents the main aspects of implementing the proposed blockchain-based API endpoint access control system. It begins by showing the tools, libraries, and technologies used for the implementation. Then, the configuration of the Hyperledger Fabric blockchain network is described. Following this, the smart contract (chaincode) is presented.

The chapter then highlights the system's practical interaction with end users through screenshots. Lastly, the exploration of the Hyperledger Fabric blockchain is depicted using a set of screenshots from the Blockchain Explorer tool.

5.1 Technology Stack

The following diagram presents the technologies used for the implementation.

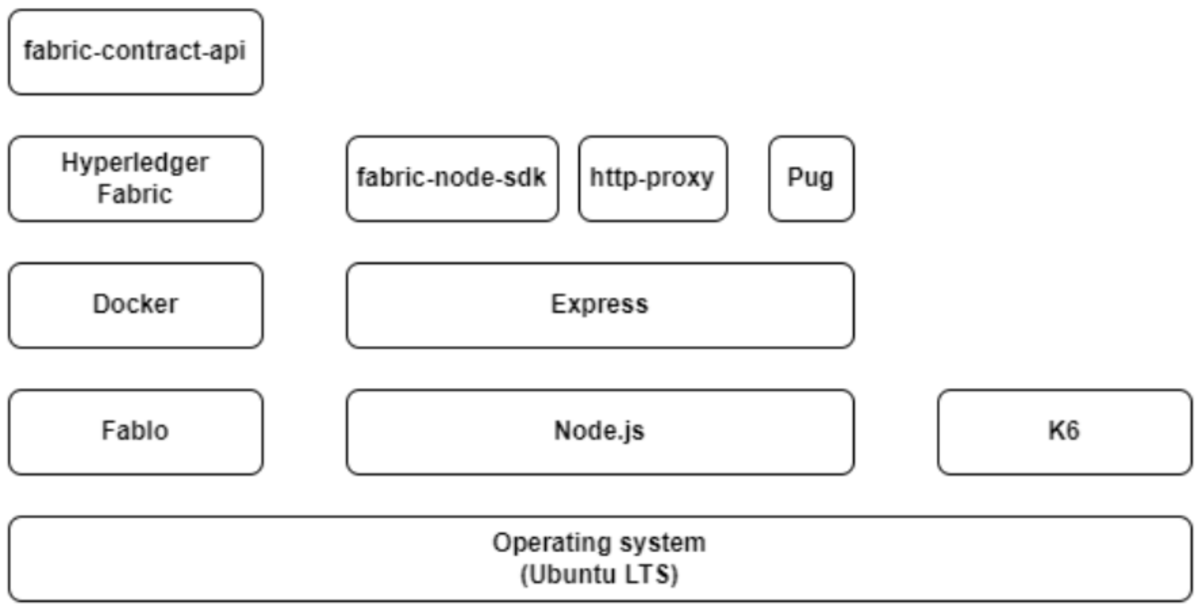


Figure 5-1 Technology Stack

Table 5-1 Technology stack explanation

Technology	Explanation
Ubuntu LTS	An operating system

Fablo	A tool that takes a single YAML file that describes the desired Hyperledger Fabric network (e.g., number of orderers and peers, consensus mechanism, channels, etc.) and generates necessary configuration files and shell scripts that create such a network that runs on top of Docker and maintain it [16]
Docker	A containerization platform widely used in software development. It allows developers to package applications, dependencies, and runtime environments into self-contained units called containers. These containers can be easily deployed and executed on any system that supports Docker, ensuring consistent behaviour across different environments. This technology simplifies the process of software development, testing, and deployment, making it a popular choice for building scalable and portable applications [17]
Hyperledger Fabric	A blockchain framework. Under the hood, Fabric utilizes Docker containers to encapsulate and isolate various components of the blockchain network, including peers, orderers, and the certificate authority. Thanks to doing so, the deployment of blockchain networks is simplified, making it easier for developers to set up and maintain distributed ledger systems.
fabric-contract-api	A JavaScript module used in Hyperledger Fabric chaincode development to simplify the implementation of smart contracts. It provides a set of classes and functions that abstract the low-level details of the Hyperledger Fabric chaincode [18]
Node.js	A server-side runtime environment built on Chrome's V8 JavaScript engine. It allows developers to execute JavaScript code outside the browser, enabling server-side application development. Node.js provides an event-driven, non-blocking I/O model that makes it highly efficient. Its extensive package manager, npm, offers a vast ecosystem of libraries and modules, simplifying the development process and enhancing code reusability [19]
Express	A web application framework for Node.js. It helps build web servers and APIs as

	well as simplifies handling HTTP requests and responses with its intuitive routing system and middleware support [20]
fabric-node-sdk	A Node.js software development kit (SDK) for Hyperledger Fabric. It enables developers to interact with the Hyperledger Fabric network, create, endorse, submit transactions, invoke the functions on the chaincode, and query the blockchain's state [21]
http-proxy	A Node.js module that facilitates HTTP proxying, allowing a server to forward incoming HTTP requests to other destinations. In this case, the destinations are the protected API endpoints [22]
Pug	A template engine for Node.js. It provides a concise syntax for generating HTML pages. In this case, those pages are the front-end user interface for the API providers and clients [23]
K6	A load-testing tool designed to assess the performance of web applications and APIs [24]

The following diagram indicates which core system component is made with which technology:

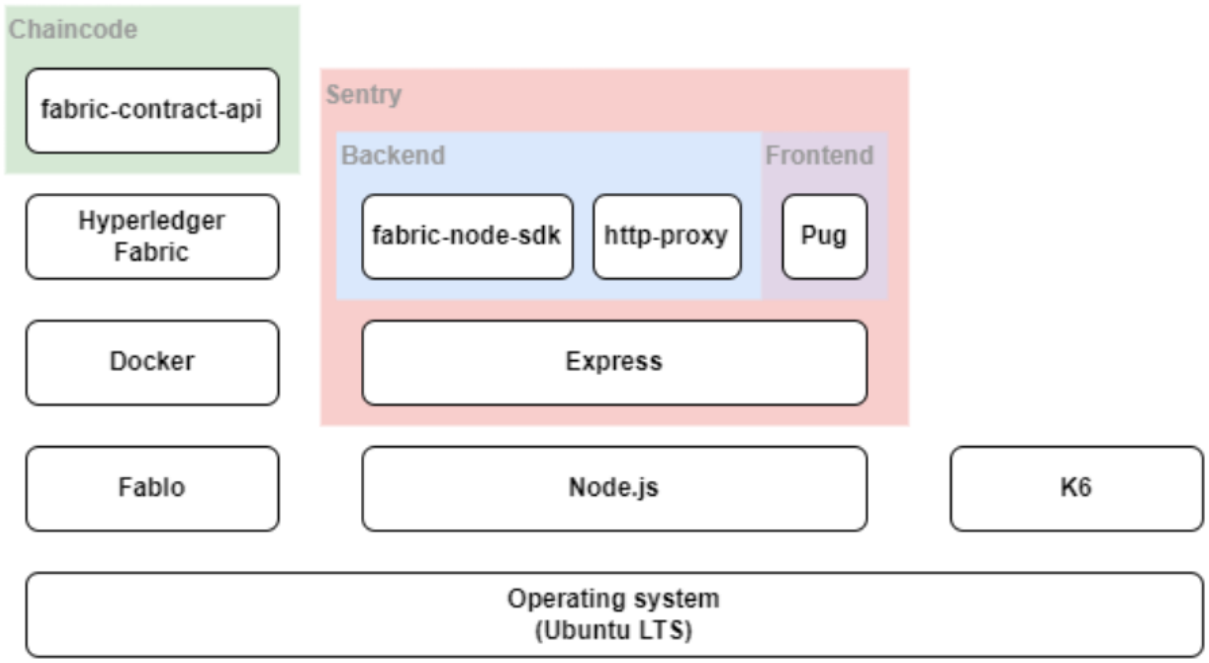


Figure 5-2 Technology Stack with Highlighted System Components

As shown in the diagram, Sentry is built on top of Express and consists of two parts:

- The front-end: this is the user interface generated by Pug for the API providers to add and manage the API endpoints and for the app developer to discover and get access to those endpoints.
- The back-end: this uses the fabric-node-sdk module to interact with the chaincode running on the peers of the Hyperledger Fabric network to manage and query access control data and then uses the http-proxy to forward the requests to the protected API endpoints.

5.2 Hyperledger Fabric Network Configuration

A YAML file was put together for Fablo to create the Hyperledger Fabric network, containing the following key properties:

- Two orderers using the RAFT consensus mechanism.
- One organization with one peer
- One channel for the organization and its peers
- One Node.js chaincode deployed on all the peers within the organization.

The number of peers in the organization is changed during the performance evaluation experiments. Every time the YAML file is changed, the whole Hyperledger Fabric network is destroyed and recreated to ensure the system's state is always clean and fresh for testing and development.

5.3 Chaincode

The following is a summarized representation of the chaincode, focusing on its main functionalities:

```
const { Contract } = require('fabric-contract-api');

class Sentry extends Contract {

   async AddClient
   async AddProvider
   async GetUser
   async AddOriginServer
   async AddEndpoint
   async AddEndpointAccessGrant
   async GetEndpointAccessGrant
   async Approve
   async Revoke
   async Enable
   async GetOriginServerInfo

}
```

Figure 5-3 Chaincode (smart contract) methods

As shown in the code, the chaincode extends from the Contract object of the fabric-contract-api module and consists of the following methods:

Table 5-2 Chaincode (smart contract) methods explanation

Method	Explanation

AddClient	Adds a new client to the system.
AddProvider	Adds a new API provider to the system.
GetUser	Retrieves user information based on the caller. A user can either be a client or an API provider.
AddOriginServer	Adds a new origin server associated with an API provider.
AddEndpoint	Adds an endpoint to an origin server.
AddEndpointAccessGrant	Grants access to a client for an endpoint.
GetEndpointAccessGrant	Retrieves information about an endpoint access grant.
Approve	Approves an endpoint access grant.
Revoke	Revokes an endpoint access grant.
Enable	Re-enable a revoked endpoint access grant.
GetOriginServerInfo	Retrieves information about the origin server related to an access grant, containing the origin server's address so Sentry knows where to forward the traffic.

5.4 Demo

This section demonstrates how the API providers and app developers use Sentry. Take the scenario where a software developer has developed a suite of APIs containing the endpoint GET /sample-get. In this case, such a developer can be called an API provider. Assuming no access control exists for the API, the provider can use Sentry to protect the endpoints. From the point of view of the provider, here are the steps they can follow:

1. Head to the register page, tick the box to verify that it is an API provider, then input a name, and click Create Account

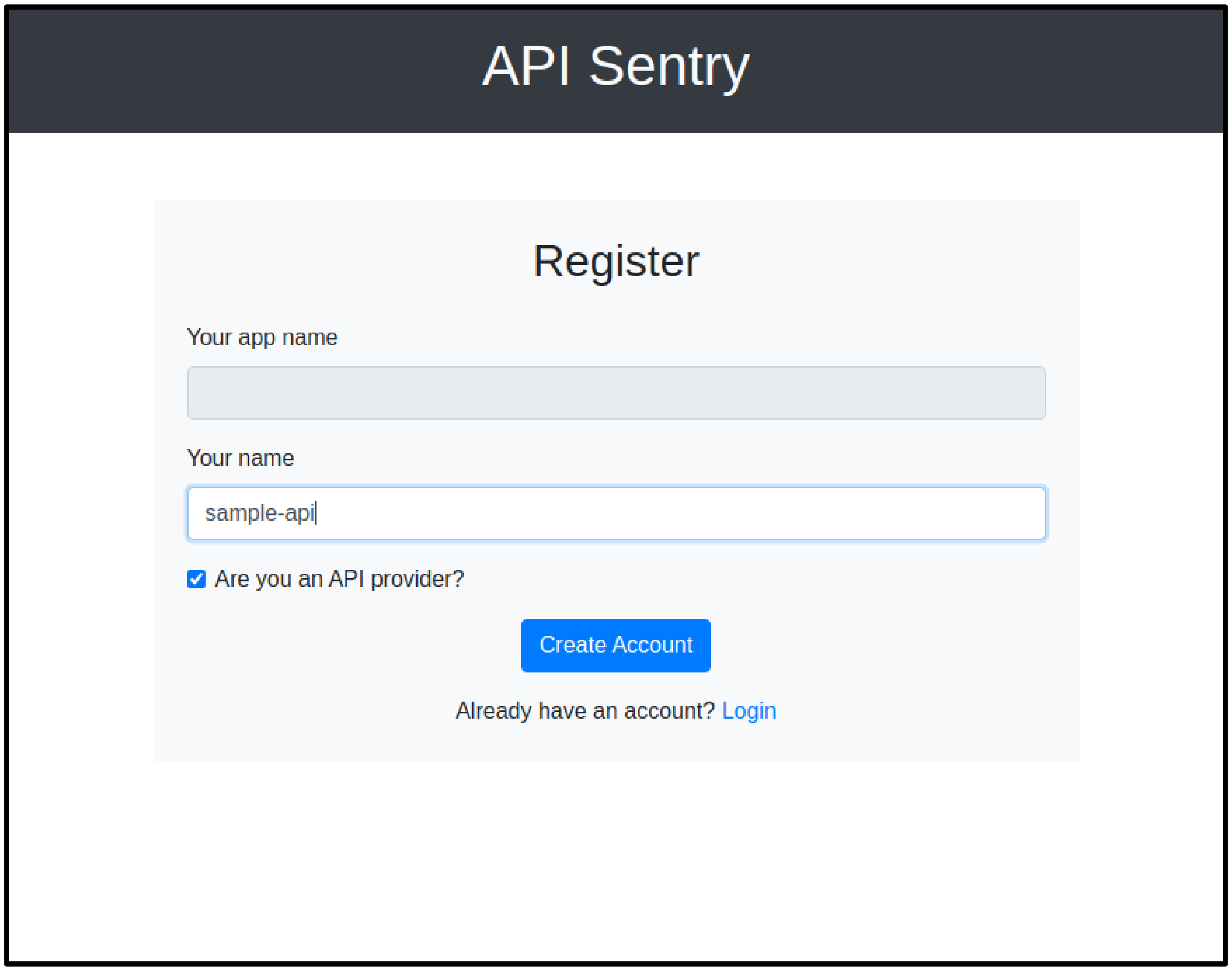
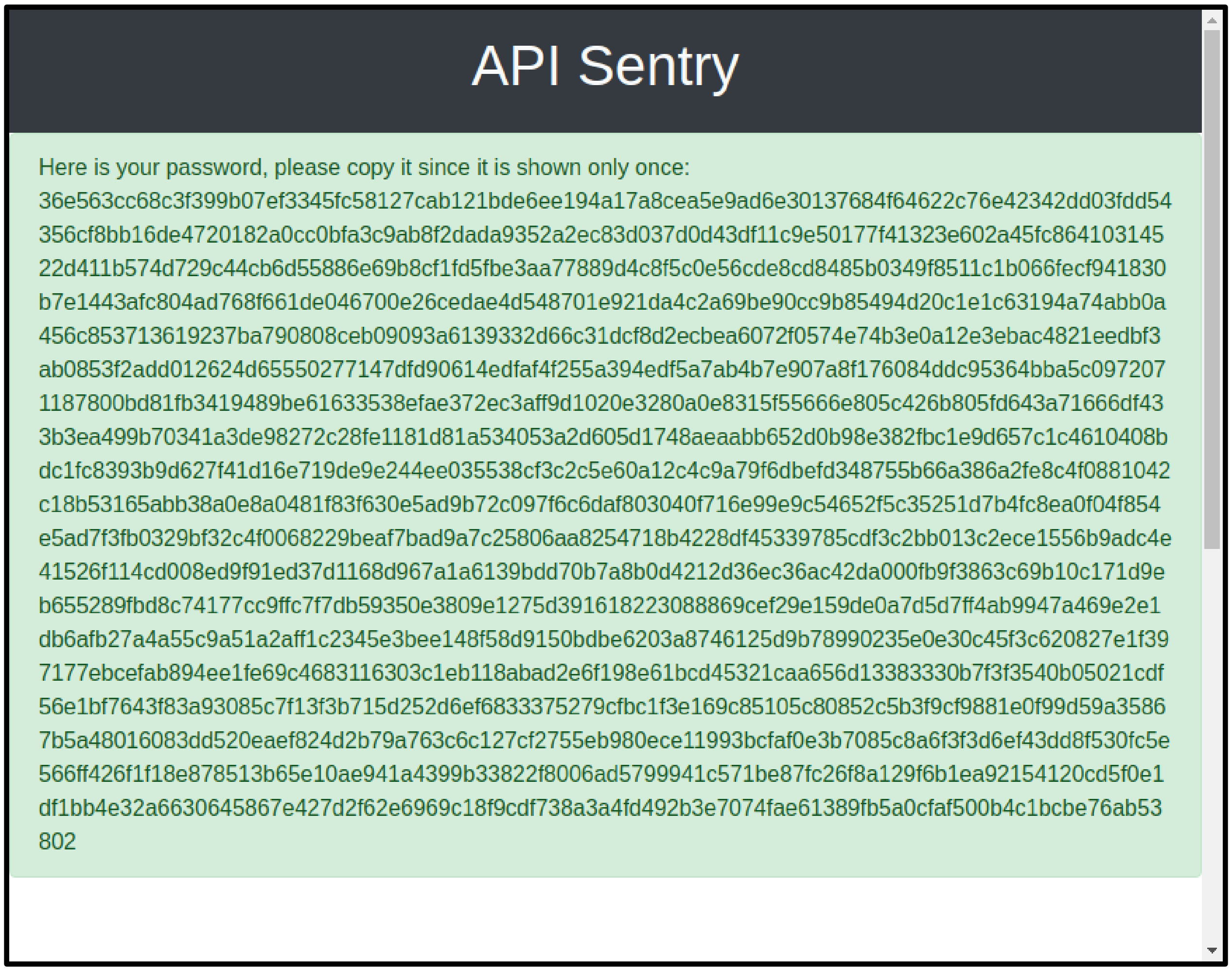


Figure 5-4 Demo: API Provider Registration step 1

2. Sentry will redirect the provider to the login page and show an encrypted Hyperledger Fabric wallet that the provider can use to log in. To the provider, it is like a password. The provider needs to copy and keep the wallet secure.



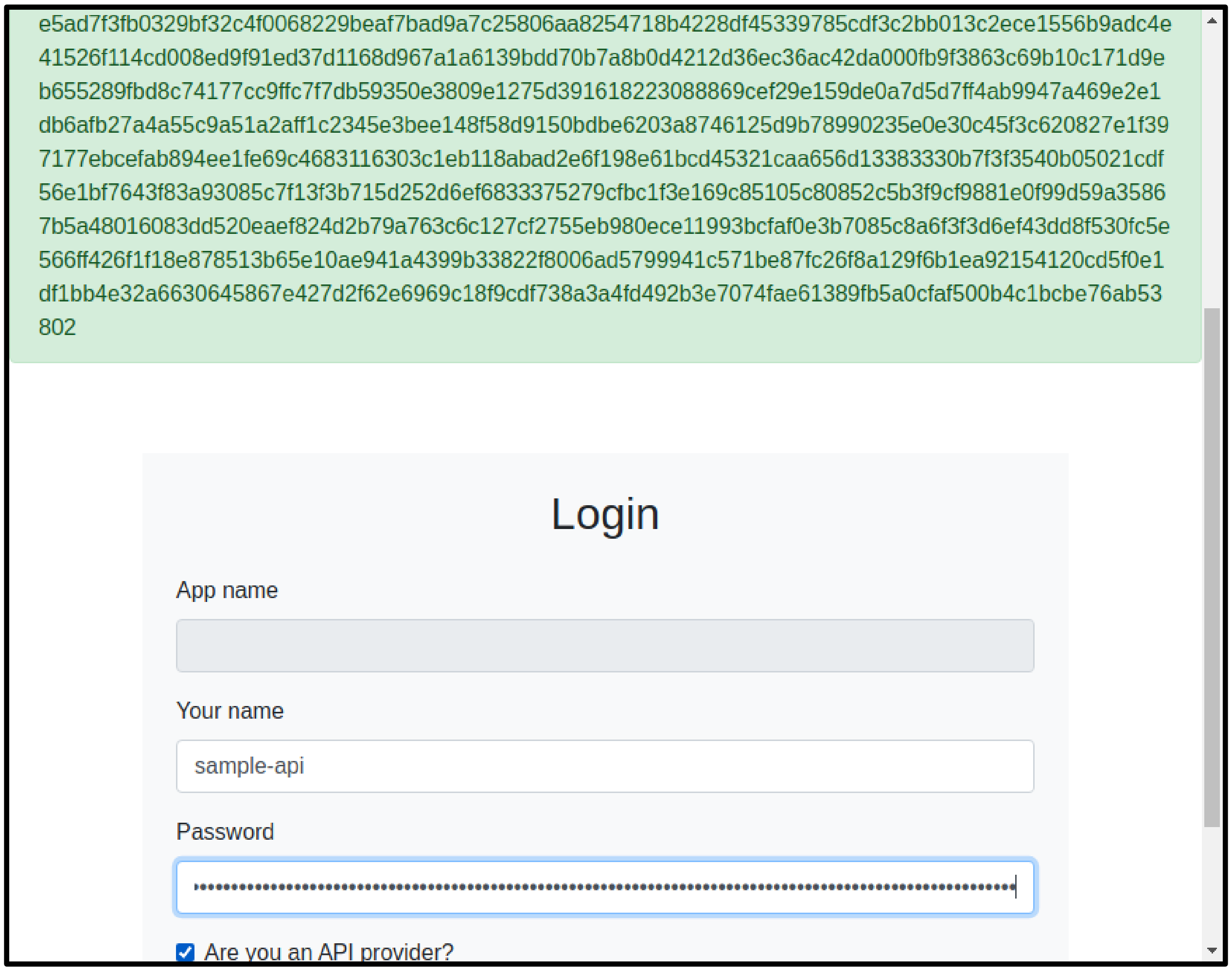
API Sentry

Here is your password, please copy it since it is shown only once:

```
36e563cc68c3f399b07ef3345fc58127cab121bde6ee194a17a8cea5e9ad6e30137684f64622c76e42342dd03fdd54
356cf8bb16de4720182a0cc0bfa3c9ab8f2dada9352a2ec83d037d0d43df11c9e50177f41323e602a45fc864103145
22d411b574d729c44cb6d55886e69b8cf1fd5fbe3aa77889d4c8f5c0e56cde8cd8485b0349f8511c1b066fecf941830
b7e1443afc804ad768f661de046700e26cedae4d548701e921da4c2a69be90cc9b85494d20c1e1c63194a74abb0a
456c853713619237ba790808ceb09093a6139332d66c31dcf8d2ecbea6072f0574e74b3e0a12e3ebac4821eedbf3
ab0853f2add012624d65550277147dfd90614edfaf4f255a394edf5a7ab4b7e907a8f176084ddc95364bba5c097207
1187800bd81fb3419489be61633538efae372ec3aff9d1020e3280a0e8315f55666e805c426b805fd643a71666df43
3b3ea499b70341a3de98272c28fe1181d81a534053a2d605d1748aeaabb652d0b98e382fbc1e9d657c1c4610408b
dc1fc8393b9d627f41d16e719de9e244ee035538cf3c2c5e60a12c4c9a79f6dbefd348755b66a386a2fe8c4f0881042
c18b53165abb38a0e8a0481f83f630e5ad9b72c097f6c6daf803040f716e99e9c54652f5c35251d7b4fc8ea0f04f854
e5ad7f3fb0329bf32c4f0068229beaf7bad9a7c25806aa8254718b4228df45339785cdf3c2bb013c2ece1556b9adc4e
41526f114cd008ed9f91ed37d1168d967a1a6139bdd70b7a8b0d4212d36ec36ac42da000fb9f3863c69b10c171d9e
b655289fbd8c74177cc9ffc7f7db59350e3809e1275d391618223088869cef29e159de0a7d5d7ff4ab9947a469e2e1
db6afb27a4a55c9a51a2aff1c2345e3bee148f58d9150bdbe6203a8746125d9b78990235e0e30c45f3c620827e1f39
7177ebcefab894ee1fe69c4683116303c1eb118abad2e6f198e61bcd45321caa656d13383330b7f3f3540b05021cdf
56e1bf7643f83a93085c7f13f3b715d252d6ef6833375279cfbc1f3e169c85105c80852c5b3f9cf9881e0f99d59a3586
7b5a48016083dd520eaef824d2b79a763c6c127cf2755eb980ece11993bcfaf0e3b7085c8a6f3f3d6ef43dd8f530fc5e
566ff426f1f18e878513b65e10ae941a4399b33822f8006ad5799941c571be87fc26f8a129f6b1ea92154120cd5f0e1
df1bb4e32a6630645867e427d2f62e6969c18f9cdf738a3a4fd492b3e7074fae61389fb5a0cfaf500b4c1bcbe76ab53
802
```

Figure 5-5 Demo: API Provider registration step 2.

3. Check the box to confirm bringing an API provider again and use the registered name and provided password to log in.



```
e5ad7f3fb0329bf32c4f0068229beaf7bad9a7c25806aa8254718b4228df45339785cdf3c2bb013c2ece1556b9adc4e
41526f114cd008ed9f91ed37d1168d967a1a6139bdd70b7a8b0d4212d36ec36ac42da000fb9f3863c69b10c171d9e
b655289fbd8c74177cc9ffc7f7db59350e3809e1275d391618223088869cef29e159de0a7d5d7ff4ab9947a469e2e1
db6afb27a4a55c9a51a2aff1c2345e3bee148f58d9150bdbe6203a8746125d9b78990235e0e30c45f3c620827e1f39
7177ebcefab894ee1fe69c4683116303c1eb118abad2e6f198e61bcd45321caa656d13383330b7f3f3540b05021cdf
56e1bf7643f83a93085c7f13f3b715d252d6ef6833375279cfbc1f3e169c85105c80852c5b3f9cf9881e0f99d59a3586
7b5a48016083dd520eaef824d2b79a763c6c127cf2755eb980ece11993bcfaf0e3b7085c8a6f3f3d6ef43dd8f530fc5e
566ff426f1f18e878513b65e10ae941a4399b33822f8006ad5799941c571be87fc26f8a129f6b1ea92154120cd5f0e1
df1bb4e32a6630645867e427d2f62e6969c18f9cdf738a3a4fd492b3e7074fae61389fb5a0cfaf500b4c1bcbe76ab53
802
```

Login

App name

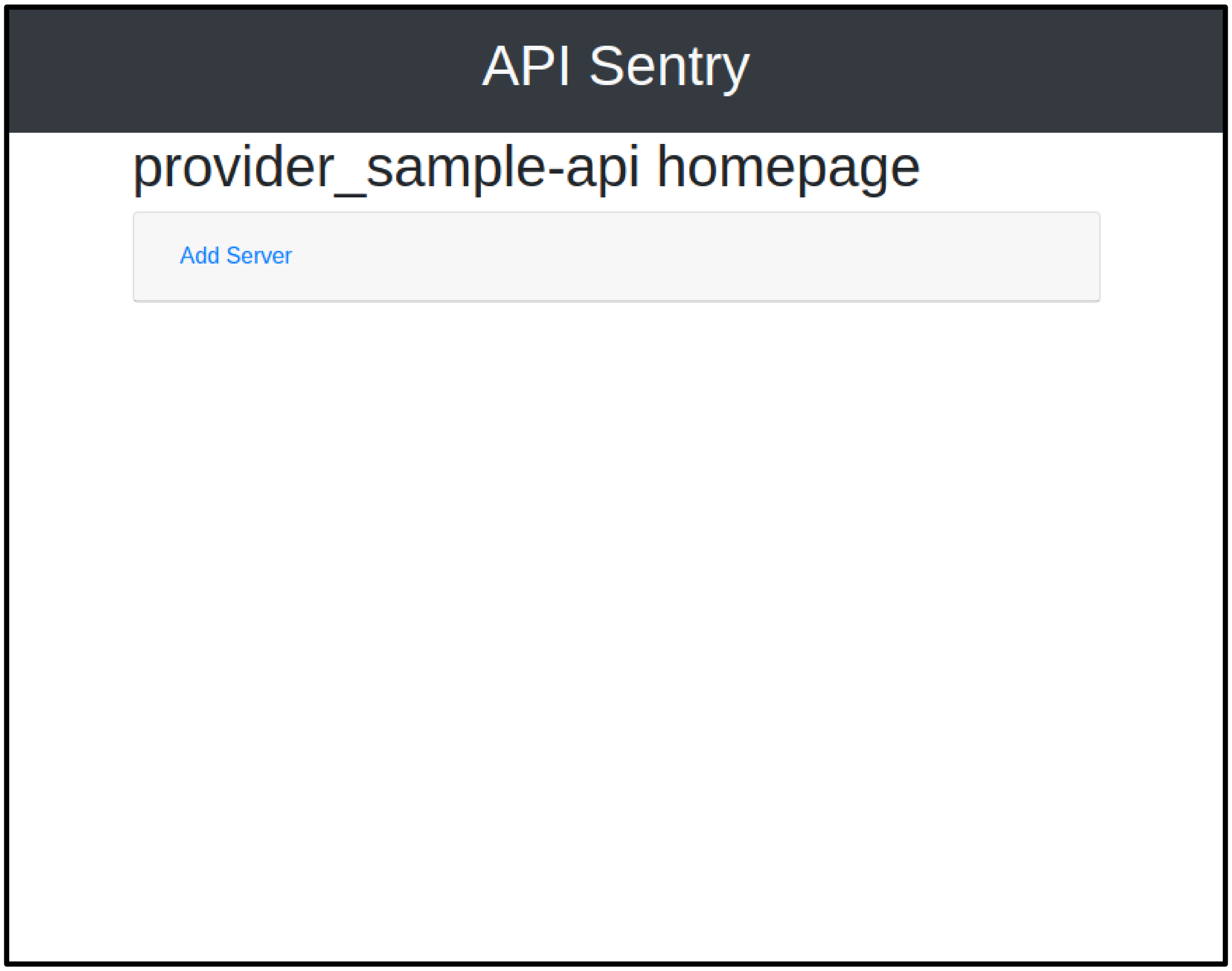
Your name: sample-api

Password: ••

[✓] Are you an API provider?

Figure 5-6 Demo: API Provider logging in.

4. On the home page, there is no server initially.

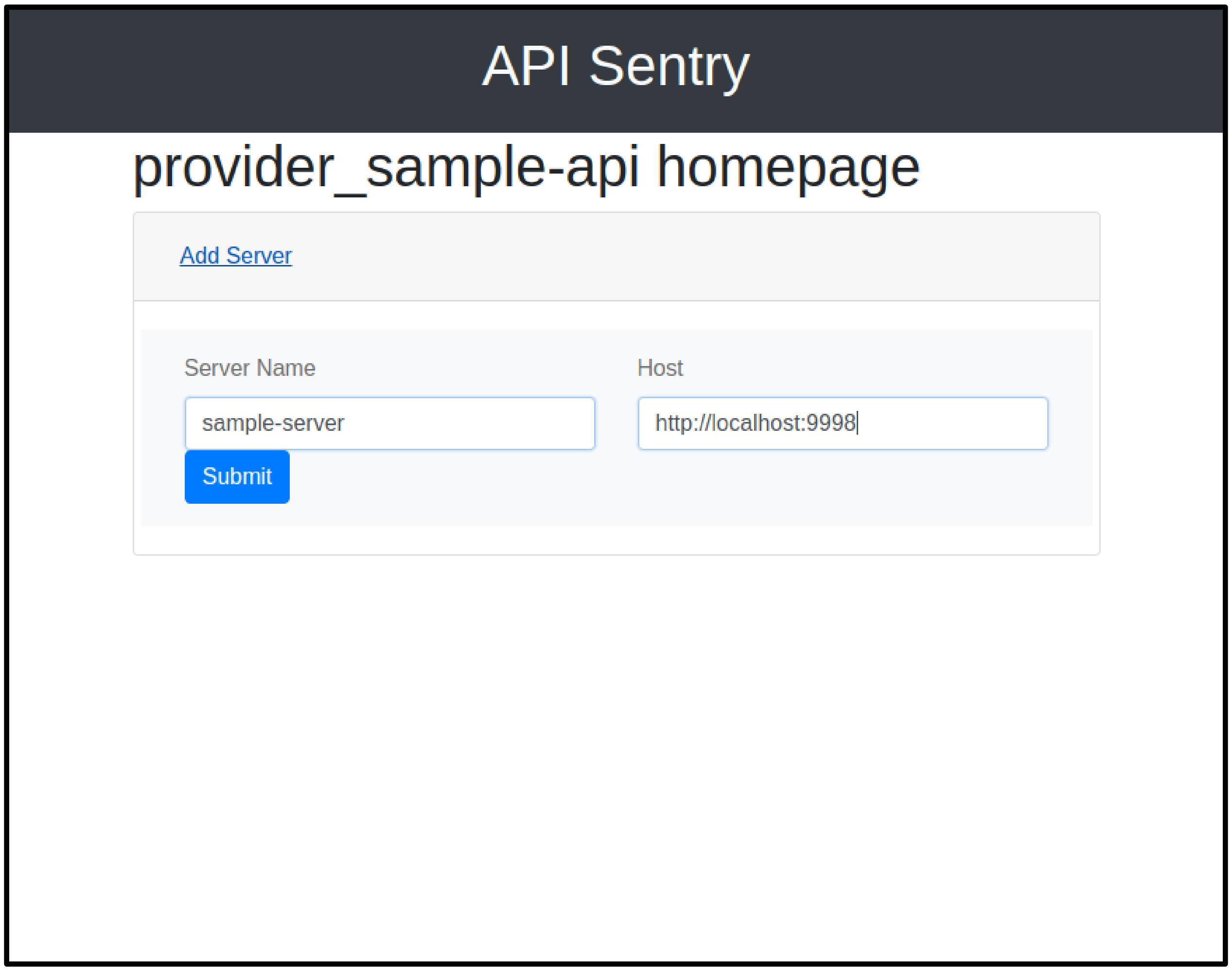


API Sentry

provider_sample-api homepage

Figure 5-7 Demo: API Provider homepage.

5. Click Add Server and fill in the details, e.g., Name the server “sample-server” and specify the host address, "http://localhost:9998.” The address is like that for demo purposes; practically, it would be something other than localhost.

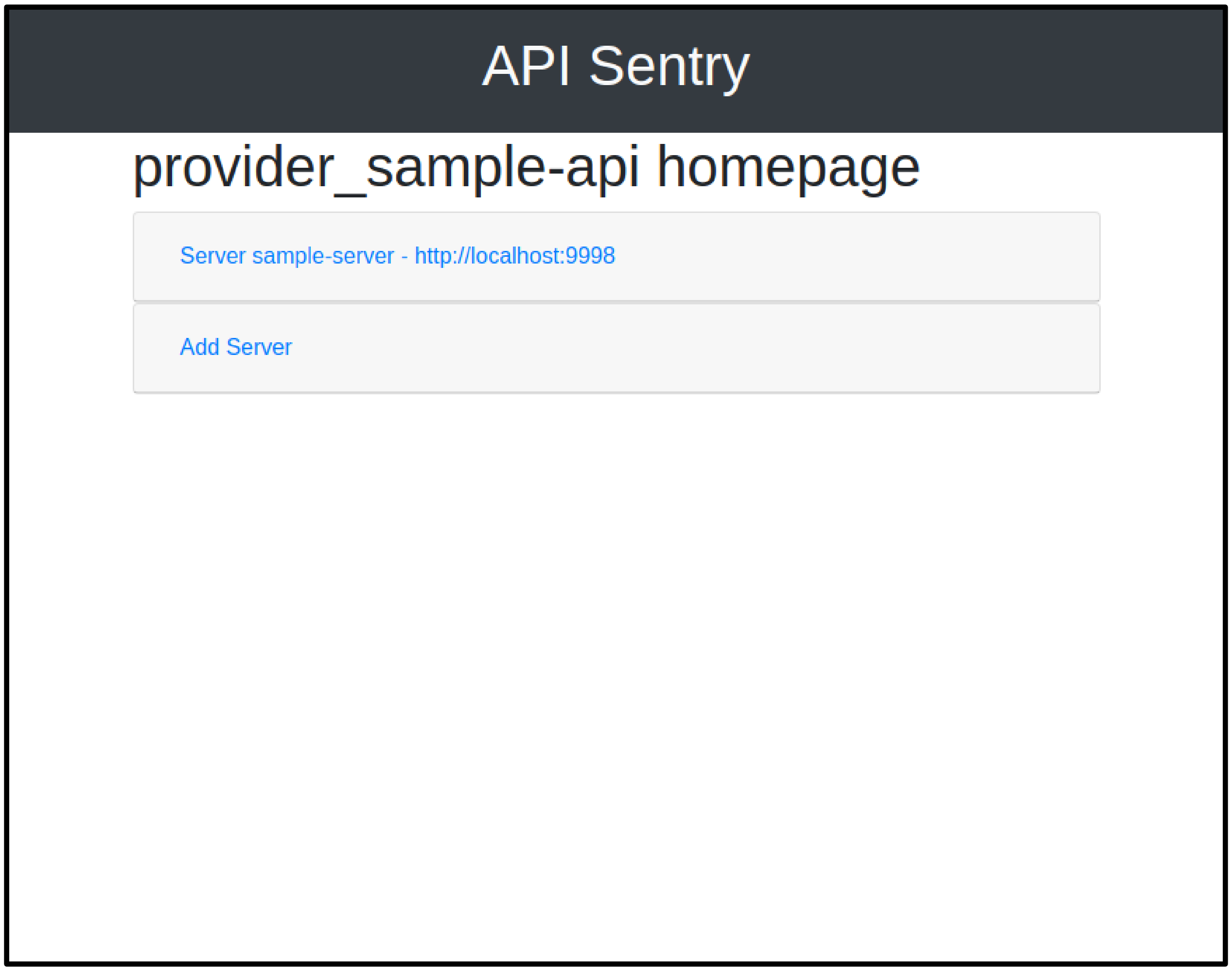

API Sentry

provider_sample-api homepage

Server Name: sample-server
Host: http://localhost:9998

Figure 5-8 Demo: API Provider adding a server.

6. Click submit; Sentry will add and track the server.


API Sentry

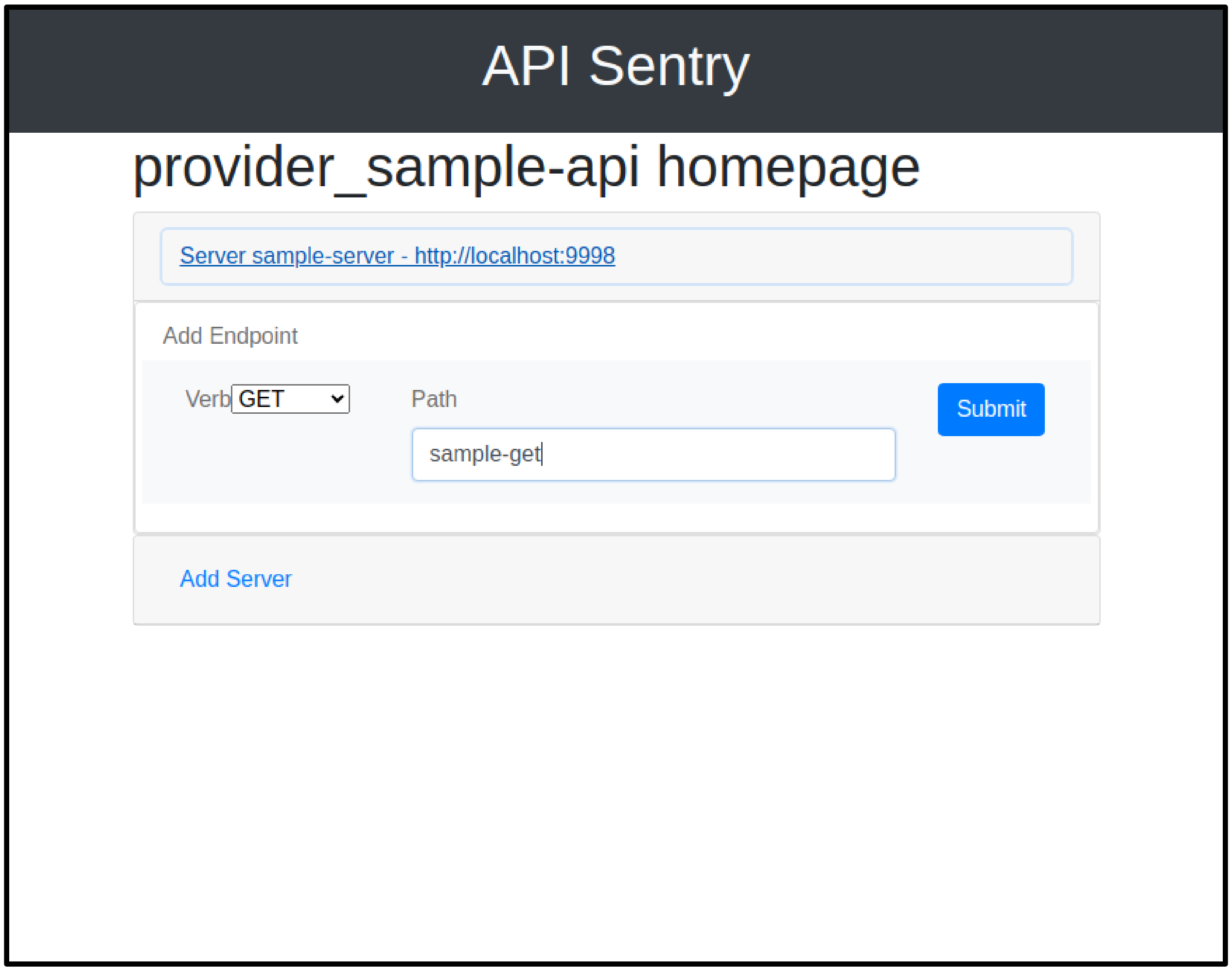
provider_sample-api homepage

Server sample-server - http://localhost:9998

Add Server

Figure 5-9 Demo: API Provider homepage showing the added server.

7. Click on the server to expand it; there will be a little form to add an endpoint, so fill in the path.


API Sentry

provider_sample-api homepage

Server sample-server - http://localhost:9998

Add Endpoint

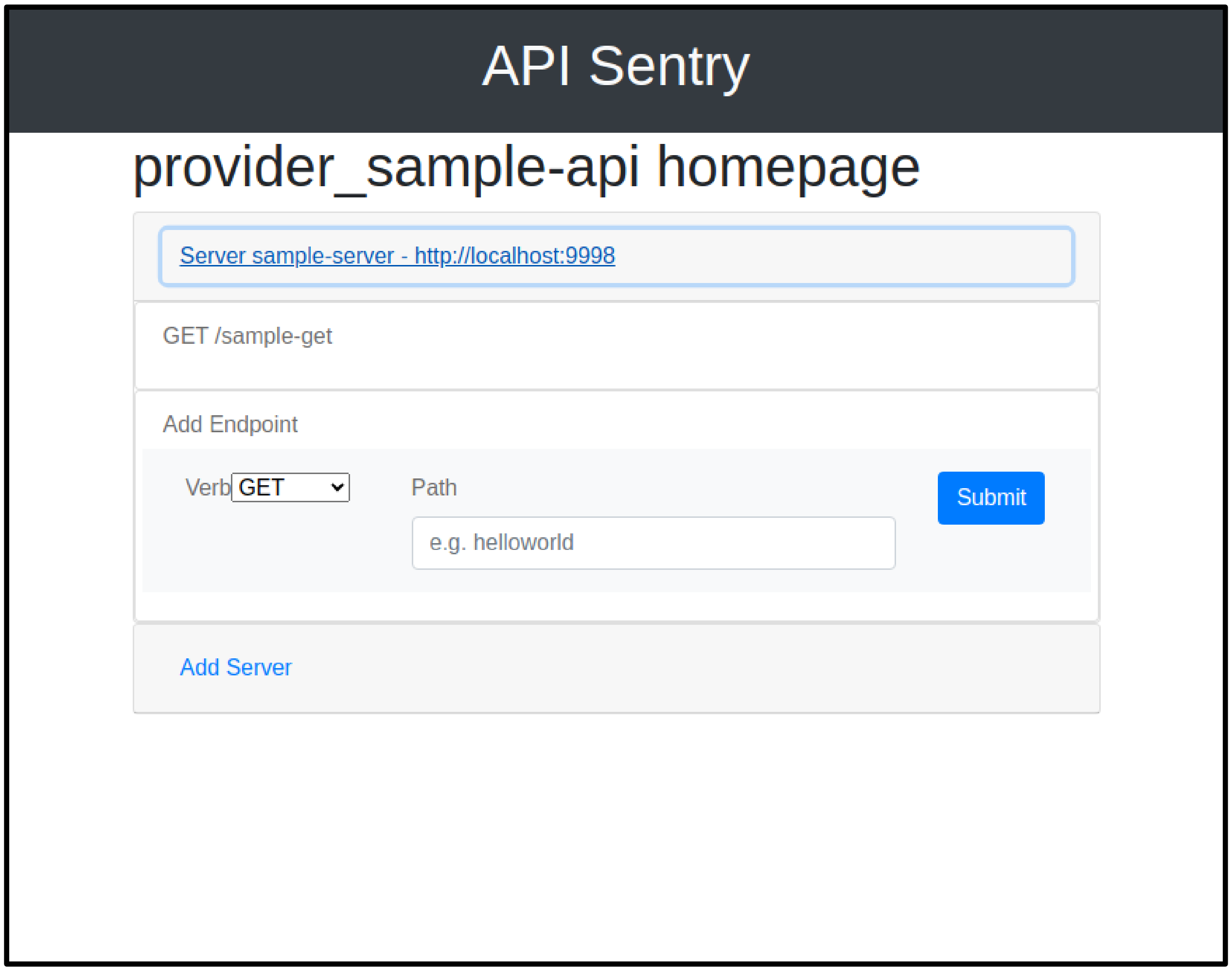
Verb GET

Path

sample-get

Figure 5-10 Demo: API Provider adding an endpoint.

8. Click submit, and the sample-get endpoint will be added.



API Sentry

provider_sample-api homepage

Server sample-server - http://localhost:9998

GET /sample-get

Add Endpoint

Verb GET

Path

e.g. helloworld

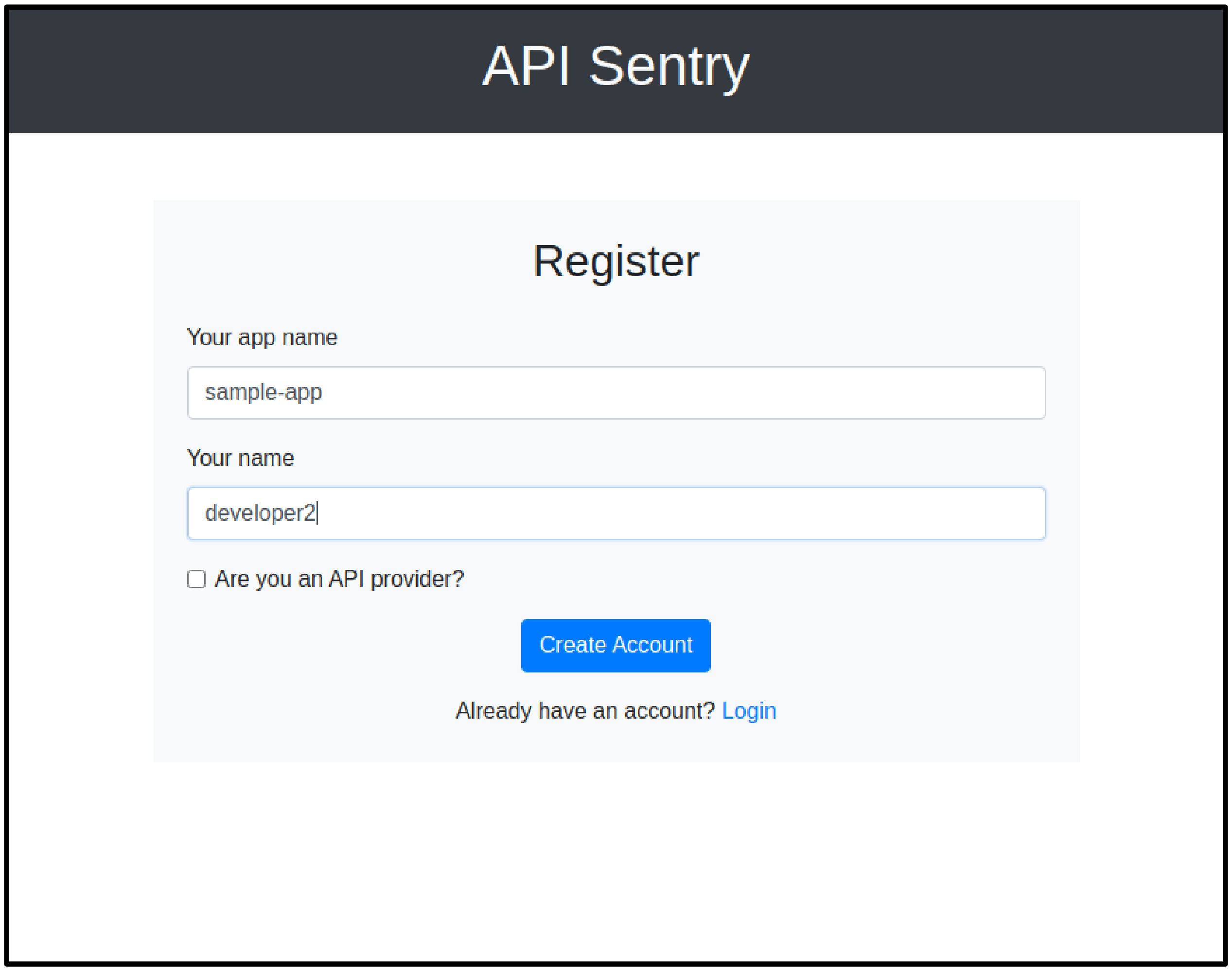
Submit

Add Server

Figure 5-11 Demo: API Provider homepage showing the added endpoint.

Suppose a developer needs to create a Node.js app that invokes the GET /sample-get endpoint from the sample-server that Sentry now protects. Here are the steps that the developer can follow:

1. Head to the register page and fill in the app and developer names.


API Sentry

Register

Your app name

sample-app

Your name

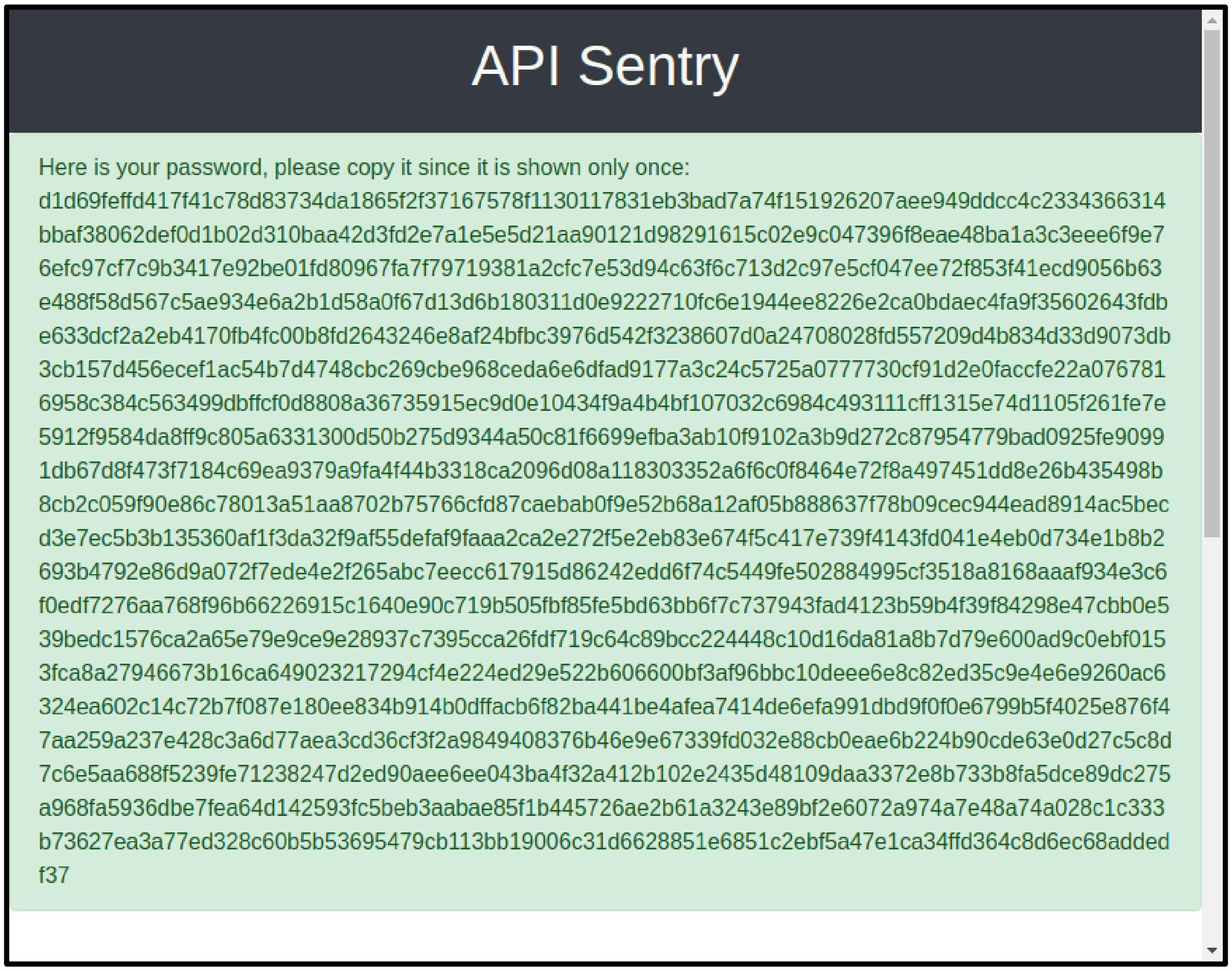
developer2

☐ Are you an API provider?

Already have an account? Login

Figure 5-12 Demo: Client app developer registration step 1.

2. After clicking submit, the developer is directed to the login page, and an encrypted Hyperledger Fabric wallet is provided.

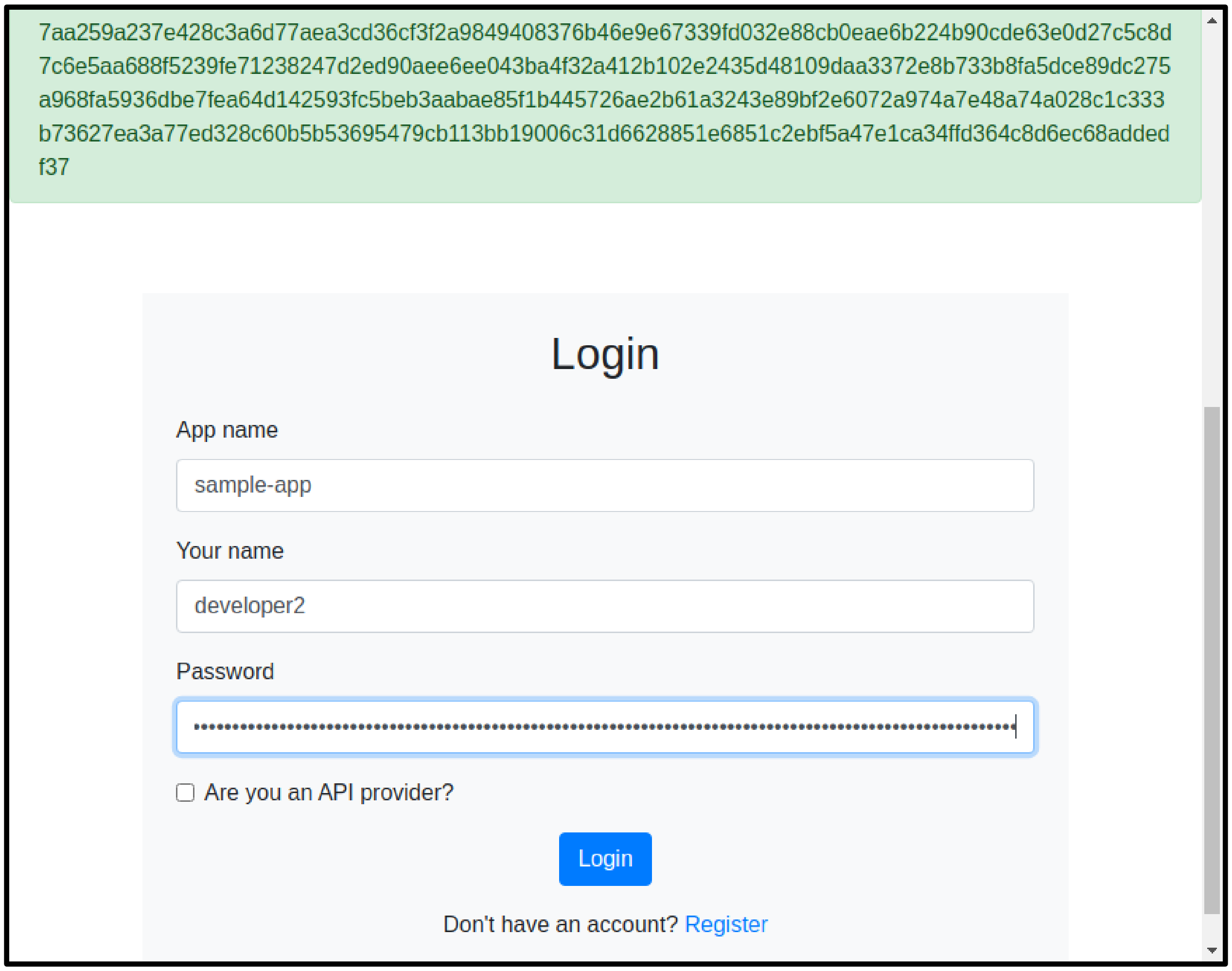


API Sentry

```
Here is your password, please copy it since it is shown only once:
d1d69feffd417f41c78d83734da1865f2f37167578f1130117831eb3bad7a74f151926207aee949ddcc4c2334366314
bbaf38062def0d1b02d310baa42d3fd2e7a1e5e5d21aa90121d98291615c02e9c047396f8eae48ba1a3c3eee6f9e7
6efc97cf7c9b3417e92be01fd80967fa7f79719381a2cfc7e53d94c63f6c713d2c97e5cf047ee72f853f41ecd9056b63
e488f58d567c5ae934e6a2b1d58a0f67d13d6b180311d0e9222710fc6e1944ee8226e2ca0bdaec4fa9f35602643fdb
e633dcf2a2eb4170fb4fc00b8fd2643246e8af24bfbc3976d542f3238607d0a24708028fd557209d4b834d33d9073db
3cb157d456ecef1ac54b7d4748cbc269cbe968ceda6e6dfad9177a3c24c5725a0777730cf91d2e0faccfe22a076781
6958c384c563499dbffcf0d8808a36735915ec9d0e10434f9a4b4bf107032c6984c493111cff1315e74d1105f261fe7e
5912f9584da8ff9c805a6331300d50b275d9344a50c81f6699efba3ab10f9102a3b9d272c87954779bad0925fe9099
1db67d8f473f7184c69ea9379a9fa4f44b3318ca2096d08a118303352a6f6c0f8464e72f8a497451dd8e26b435498b
8cb2c059f90e86c78013a51aa8702b75766cfd87caebab0f9e52b68a12af05b888637f78b09cec944ead8914ac5bec
d3e7ec5b3b135360af1f3da32f9af55defaf9faaa2ca2e272f5e2eb83e674f5c417e739f4143fd041e4eb0d734e1b8b2
693b4792e86d9a072f7ede4e2f265abc7eecc617915d86242edd6f74c5449fe502884995cf3518a8168aaaf934e3c6
f0edf7276aa768f96b66226915c1640e90c719b505fbf85fe5bd63bb6f7c737943fad4123b59b4f39f84298e47cbb0e5
39bedc1576ca2a65e79e9ce9e28937c7395cca26fdf719c64c89bcc224448c10d16da81a8b7d79e600ad9c0ebf015
3fca8a27946673b16ca649023217294cf4e224ed29e522b606600bf3af96bbc10deee6e8c82ed35c9e4e6e9260ac6
324ea602c14c72b7f087e180ee834b914b0dffacb6f82ba441be4afea7414de6efa991dbd9f0f0e6799b5f4025e876f4
7aa259a237e428c3a6d77aea3cd36cf3f2a9849408376b46e9e67339fd032e88cb0eae6b224b90cde63e0d27c5c8d
7c6e5aa688f5239fe71238247d2ed90aee6ee043ba4f32a412b102e2435d48109daa3372e8b733b8fa5dce89dc275
a968fa5936dbe7fea64d142593fc5beb3aabae85f1b445726ae2b61a3243e89bf2e6072a974a7e48a74a028c1c333
b73627ea3a77ed328c60b5b53695479cb113bb19006c31d6628851e6851c2ebf5a47e1ca34ffd364c8d6ec68added
f37
```

Figure 5-13 Demo: Client app developer registration step 2.

3. Fill in the app name, the developer’s name, and the wallet.



```
7aa259a237e428c3a6d77aea3cd36cf3f2a9849408376b46e9e67339fd032e88cb0eae6b224b90cde63e0d27c5c8d
7c6e5aa688f5239fe71238247d2ed90aee6ee043ba4f32a412b102e2435d48109daa3372e8b733b8fa5dce89dc275
a968fa5936dbe7fea64d142593fc5beb3aabae85f1b445726ae2b61a3243e89bf2e6072a974a7e48a74a028c1c333
b73627ea3a77ed328c60b5b53695479cb113bb19006c31d6628851e6851c2ebf5a47e1ca34ffd364c8d6ec68added
f37
```

Login

App name: sample-app

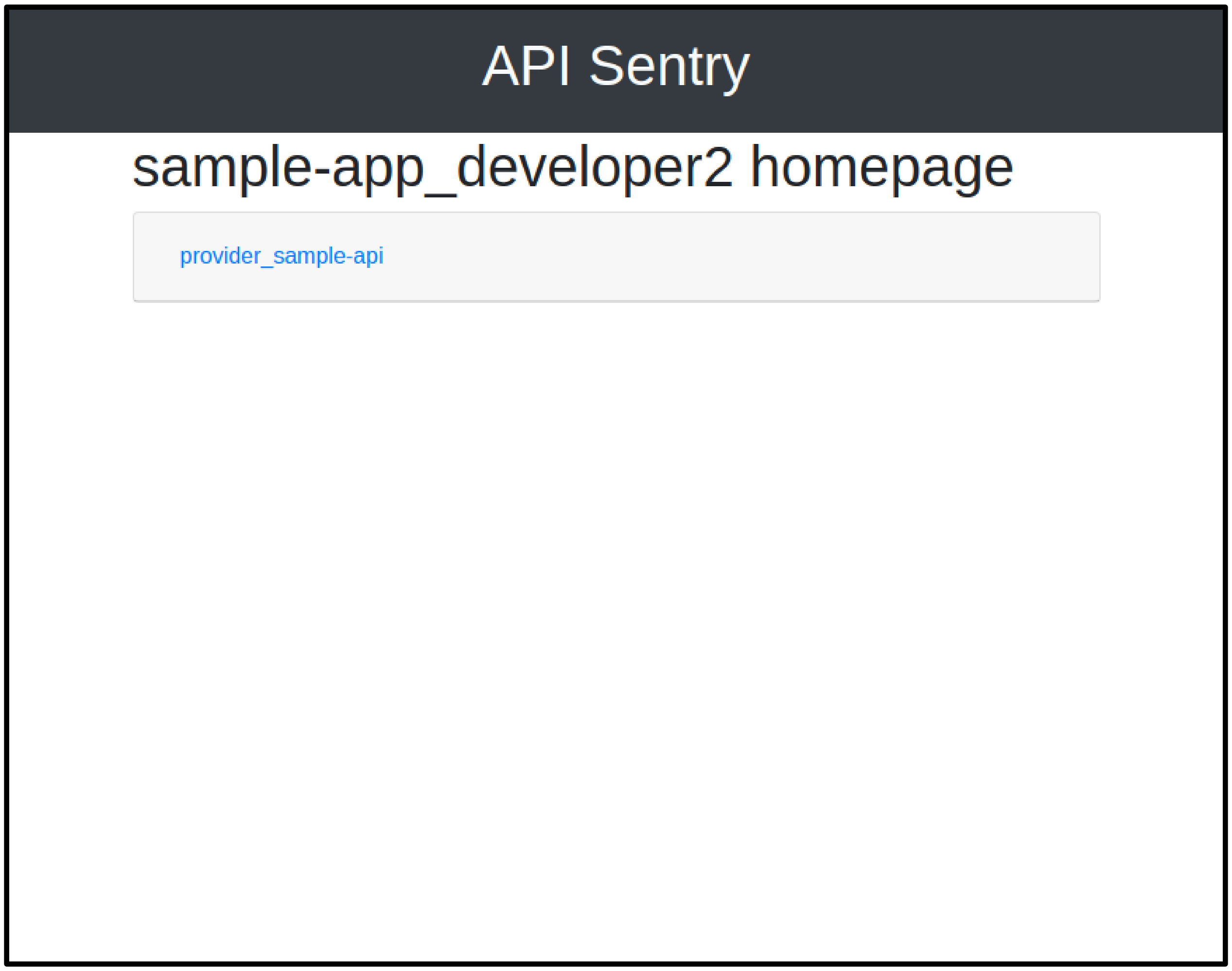
Your name: developer2

Password: ••

☐ Are you an API provider?

Figure 5-14 Demo: Client app developer logging in.

4. At the homepage, all providers registered with Sentry will be shown; here, we have the sample-api provider listed.



sample-app_developer2 homepage

provider_sample-api

Figure 5-15 Demo: Client app developer homepage.

5. Click on the provider to view the endpoint.

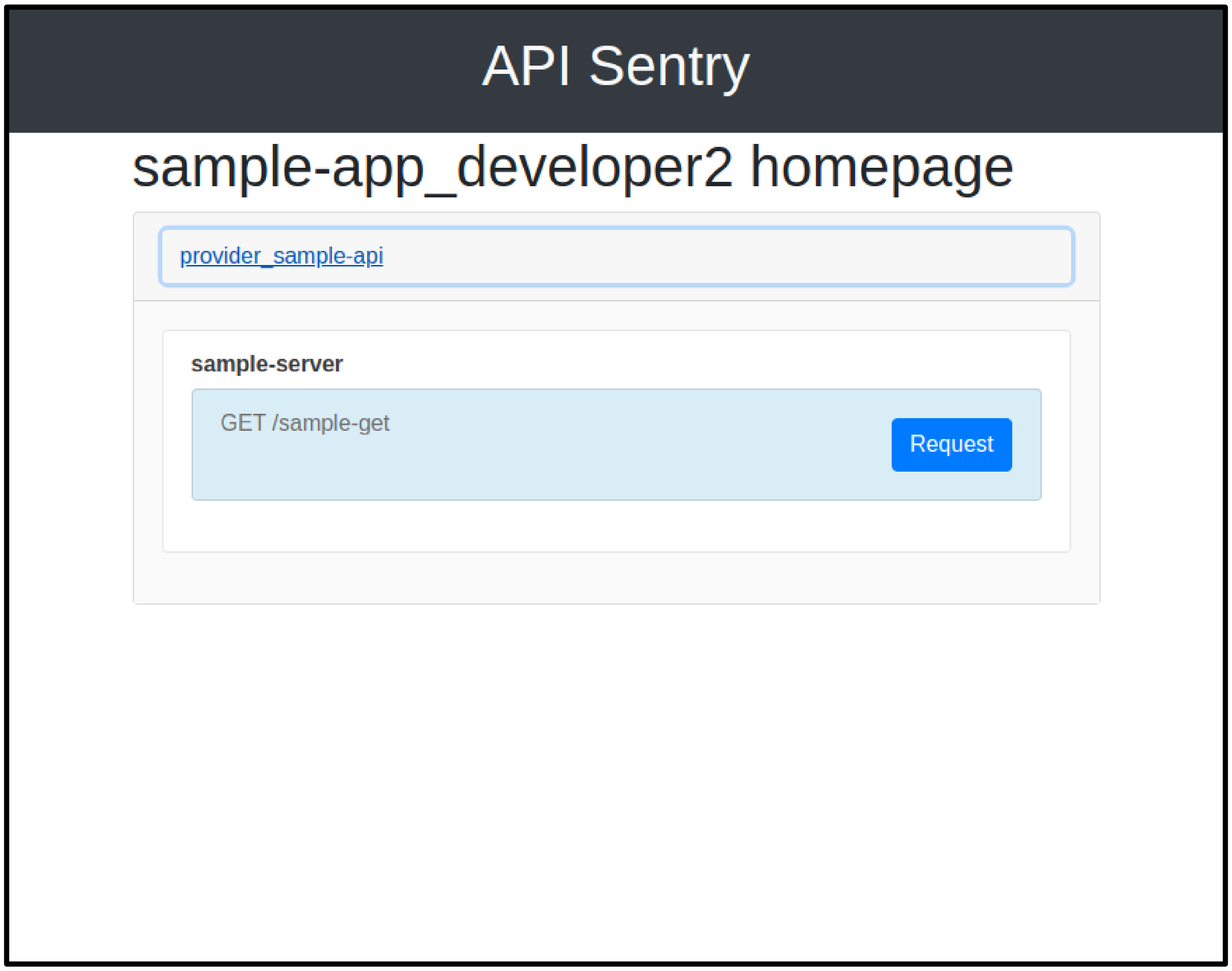


Figure 5-16 Demo: Client app developer viewing the endpoint.

6. Click the Request Access button.

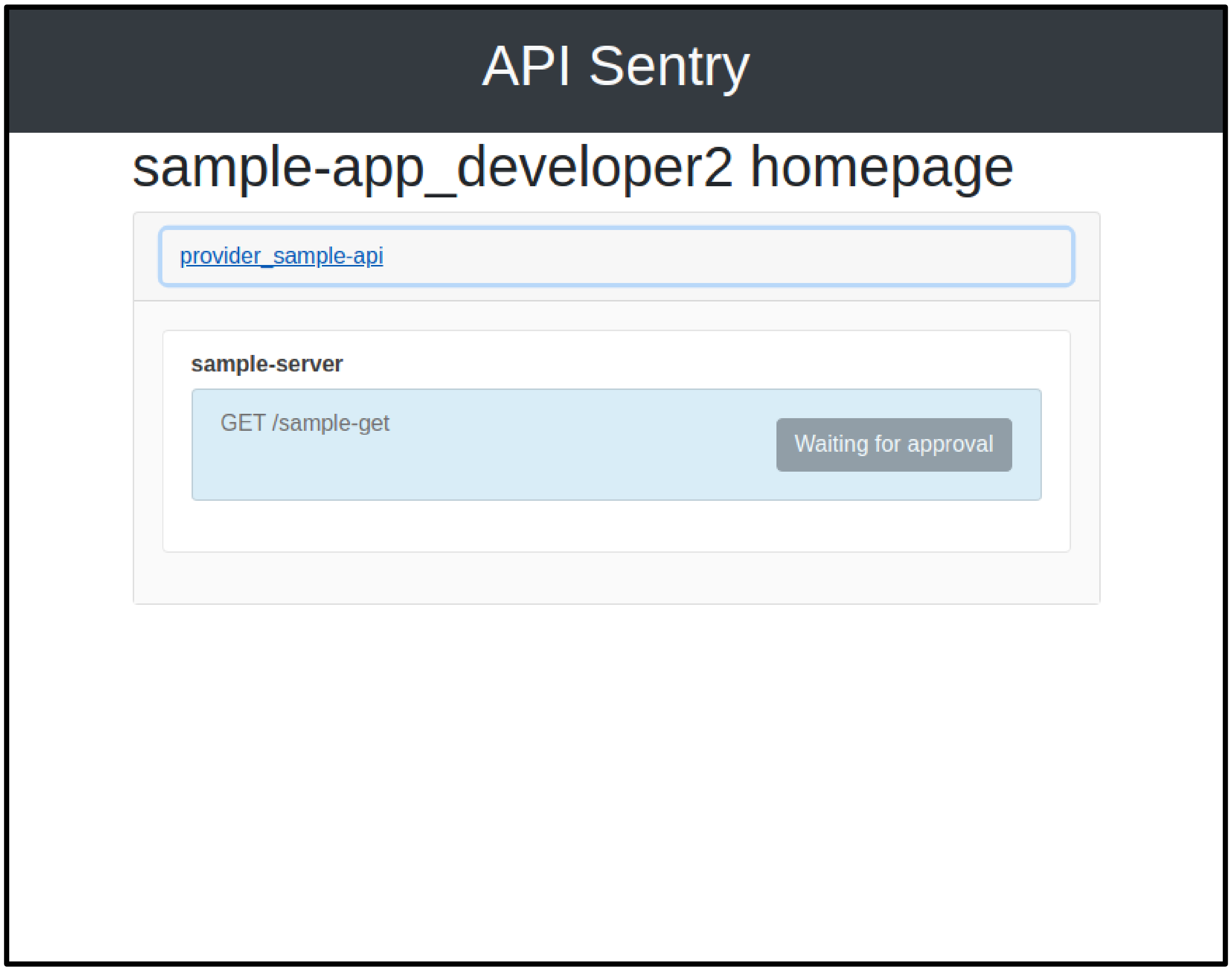
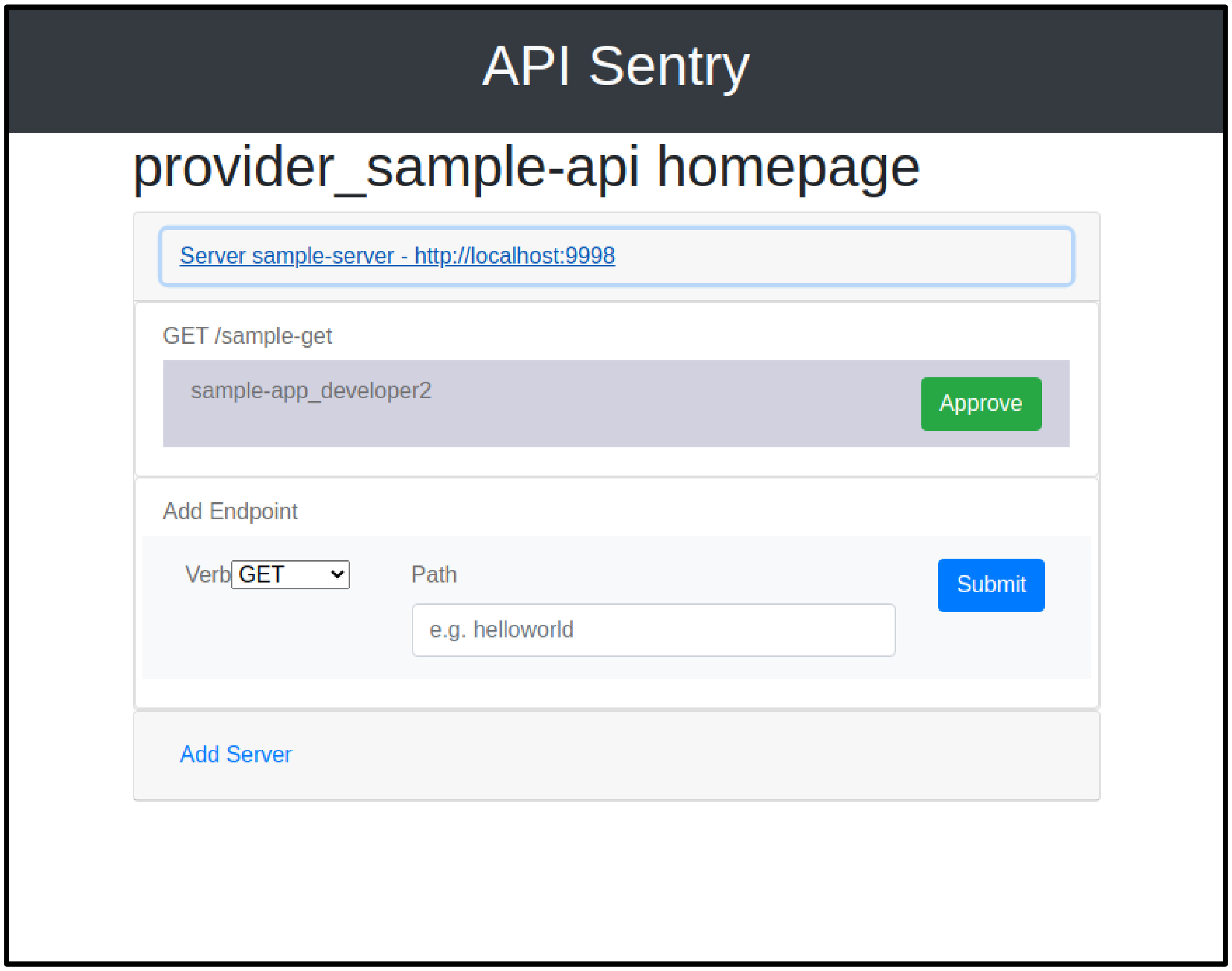


Figure 5-17 Demo: Client app developer requesting access to an endpoint.

At this stage, on the provider's side:

1. The provider will get an access request from the developer asking permission to use the GET /sample-get endpoint.


API Sentry

provider_sample-api homepage

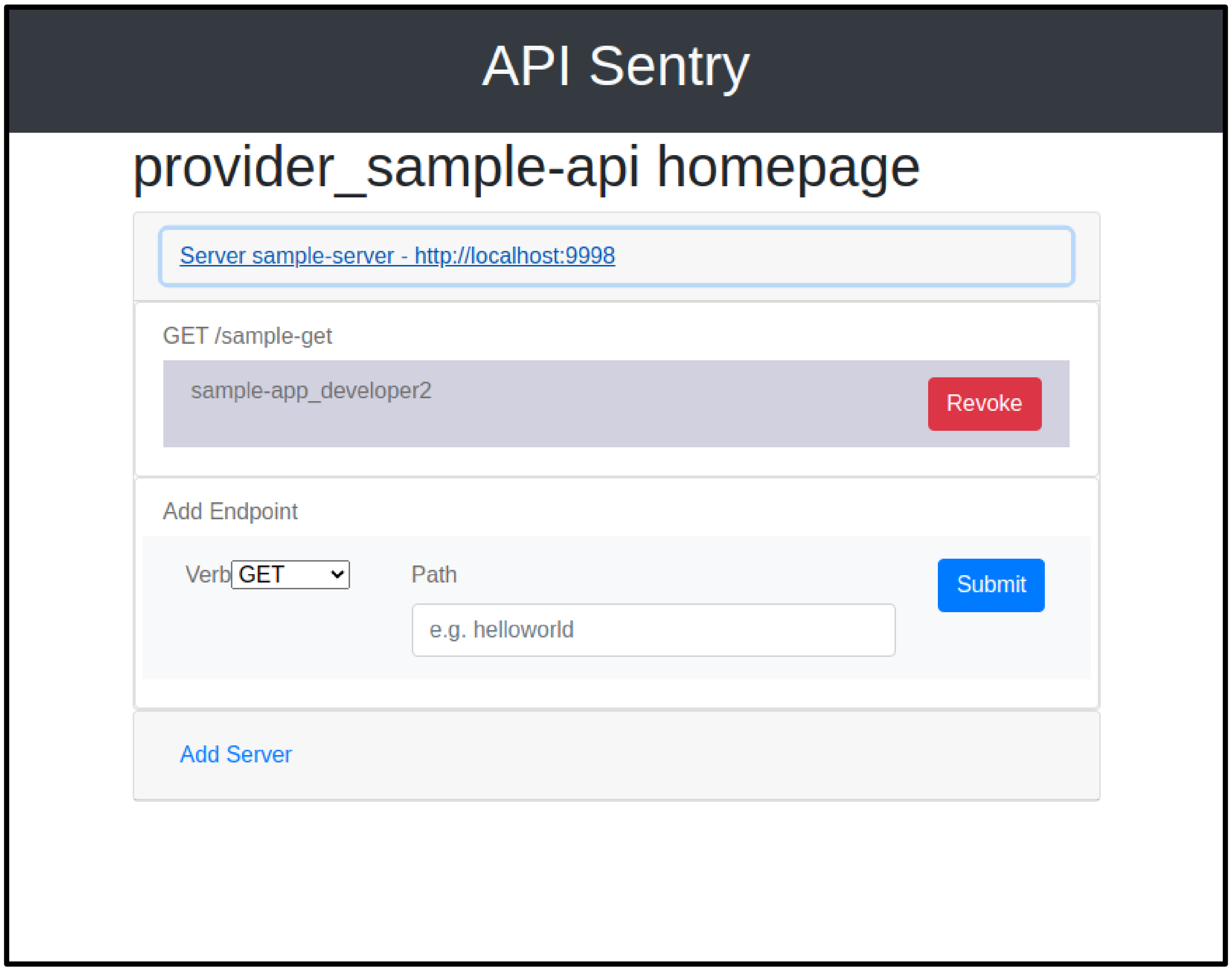
Server sample-server - http://localhost:9998

GET /sample-get

sample-app_developer2

Figure 5-18 Demo: API provider viewing the endpoint access request.

2. The provider clicks Approve to grant permission to the developer. After the endpoint has been approved, there is a “Revoke” button. When the provider does not want the sample-app client to access the endpoint anymore, they can just click the revoke button.


API Sentry

provider_sample-api homepage

Server sample-server - http://localhost:9998

GET /sample-get

sample-app_developer2

Add Endpoint

Verb GET

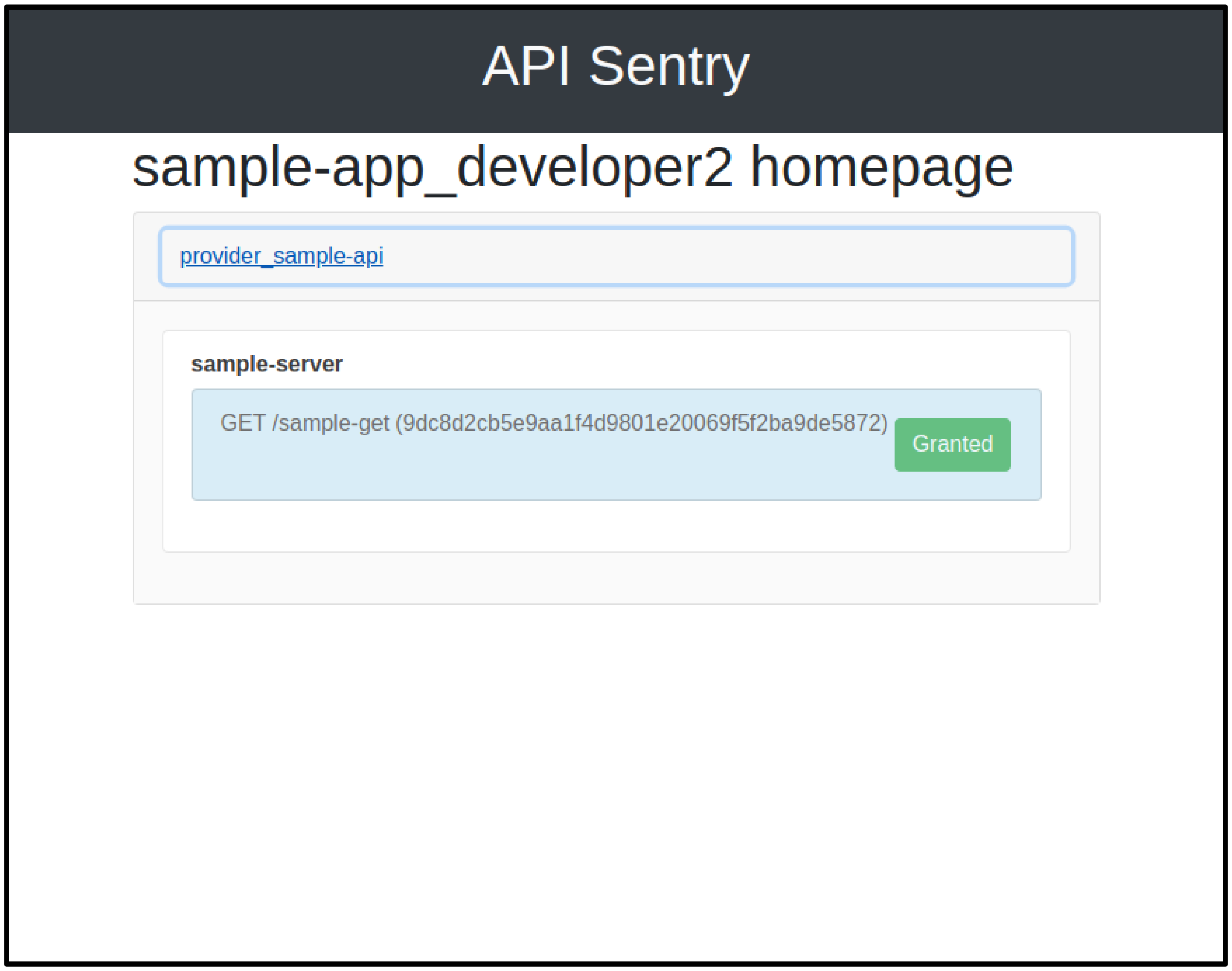
Path

e.g. helloworld

Figure 5-19 Demo: API Provider approving the endpoint access request.

Back to the developer's side:

1. The developer should see that the requested endpoint and an access grant ID in brackets have been granted.


API Sentry

sample-app_developer2 homepage

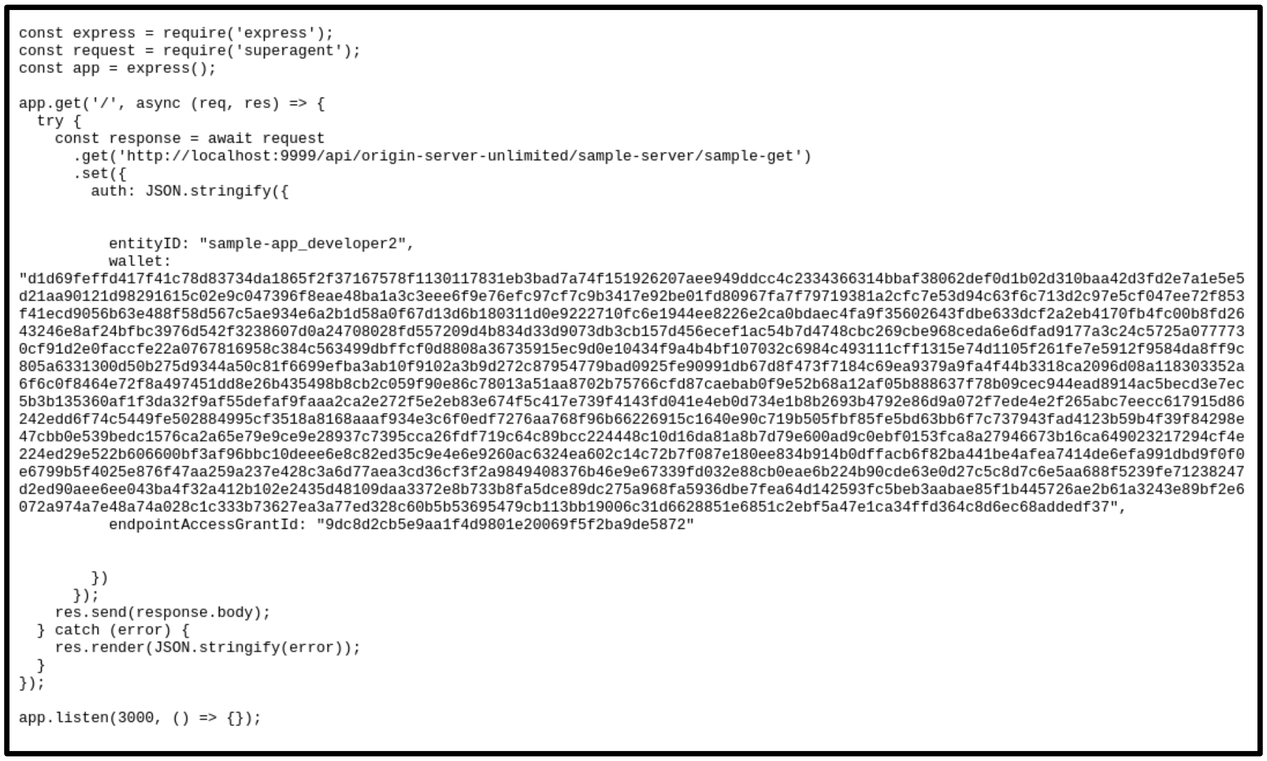
provider_sample-api

sample-server

GET /sample-get (9dc8d2cb5e9aa1f4d9801e20069f5f2ba9de5872)

Figure 5-20 Demo: Client app developer viewing the granted endpoint access.

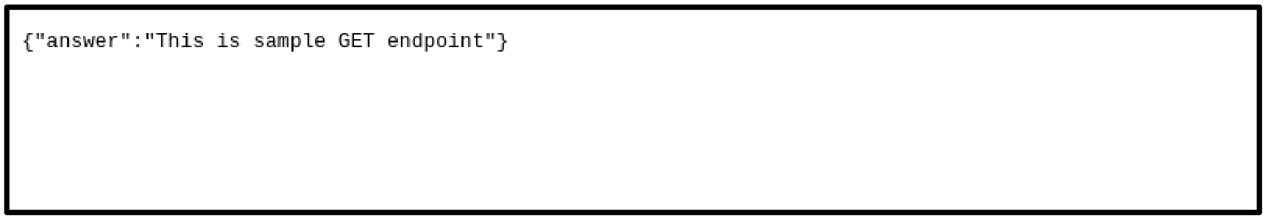
2. Embed the app name, developer name, encrypted wallet, and the access grant ID into the sample-app. Which, in this case, is a simple Node.js app



```
const express = require('express');
const request = require('superagent');
const app = express();

app.get('/', async (req, res) => {
  try {
    const response = await request
      .get('http://localhost:9999/api/origin-server-unlimited/sample-server/sample-get')
      .set({
        auth: JSON.stringify({


          entityID: "sample-app_developer2",
          wallet:
"d1d69feffd417f41c78d83734da1865f2f37167578f1130117831eb3bad7a74f151926207aee949ddcc4c2334366314bbaf38062def0d1b02d310baa42d3fd2e7a1e5e5
d21aa90121d98291615c02e9c047396f8eae48ba1a3c3eee6f9e76efc97cf7c9b3417e92be01fd80967fa7f79719381a2cfc7e53d94c63f6c713d2c97e5cf047ee72f853
f41ecd9056b63e488f58d567c5ae934e6a2b1d58a0f67d13d6b180311d0e9222710fc6e1944ee8226e2ca0bdaec4fa9f35602643fdbe633dcf2a2eb4170fb4fc00b8fd26
43246e8af24bfbc3976d542f3238607d0a24708028fd557209d4b834d33d9073db3cb157d456ecef1ac54b7d4748cbc269cbe968ceda6e6dfad9177a3c24c5725a077773
0cf91d2e0faccfe22a0767816958c384c563499dbffcf0d8808a36735915ec9d0e10434f9a4b4bf107032c6984c493111cff1315e74d1105f261fe7e5912f9584da8ff9c
805a6331300d50b275d9344a50c81f6699efba3ab10f9102a3b9d272c87954779bad0925fe90991db67d8f473f7184c69ea9379a9fa4f44b3318ca2096d08a118303352a
6f6c0f8464e72f8a497451dd8e26b435498b8cb2c059f90e86c78013a51aa8702b75766cfd87caebab0f9e52b68a12af05b888637f78b09cec944ead8914ac5becd3e7ec
5b3b135360af1f3da32f9af55defaf9faaa2ca2e272f5e2eb83e674f5c417e739f4143fd041e4eb0d734e1b8b2693b4792e86d9a072f7ede4e2f265abc7eecc617915d86
242edd6f74c5449fe502884995cf3518a8168aaaf934e3c6f0edf7276aa768f96b66226915c1640e90c719b505fbf85fe5bd63bb6f7c737943fad4123b59b4f39f84298e
47cbb0e539bedc1576ca2a65e79e9ce9e28937c7395cca26fdf719c64c89bcc224448c10d16da81a8b7d79e600ad9c0ebf0153fca8a27946673b16ca649023217294cf4e
224ed29e522b606600bf3af96bbc10deee6e8c82ed35c9e4e6e9260ac6324ea602c14c72b7f087e180ee834b914b0dffacb6f82ba441be4afea7414de6efa991dbd9f0f0
e6799b5f4025e876f47aa259a237e428c3a6d77aea3cd36cf3f2a9849408376b46e9e67339fd032e88cb0eae6b224b90cde63e0d27c5c8d7c6e5aa688f5239fe71238247
d2ed90aee6ee043ba4f32a412b102e2435d48109daa3372e8b733b8fa5dce89dc275a968fa5936dbe7fea64d142593fc5beb3aabae85f1b445726ae2b61a3243e89bf2e6
072a974a7e48a74a028c1c333b73627ea3a77ed328c60b5b53695479cb113bb19006c31d6628851e6851c2ebf5a47e1ca34ffd364c8d6ec68addedf37",
          endpointAccessGrantId: "9dc8d2cb5e9aa1f4d9801e20069f5f2ba9de5872"


        })
      });
    res.send(response.body);
  } catch (error) {
    res.render(JSON.stringify(error));
  }
});

app.listen(3000, () => {});
```

Figure 5-21 Demo: Client app developer embedding the access grant ID into the client app.

3. At this stage, the sample-app works when browsing it in the web browser.



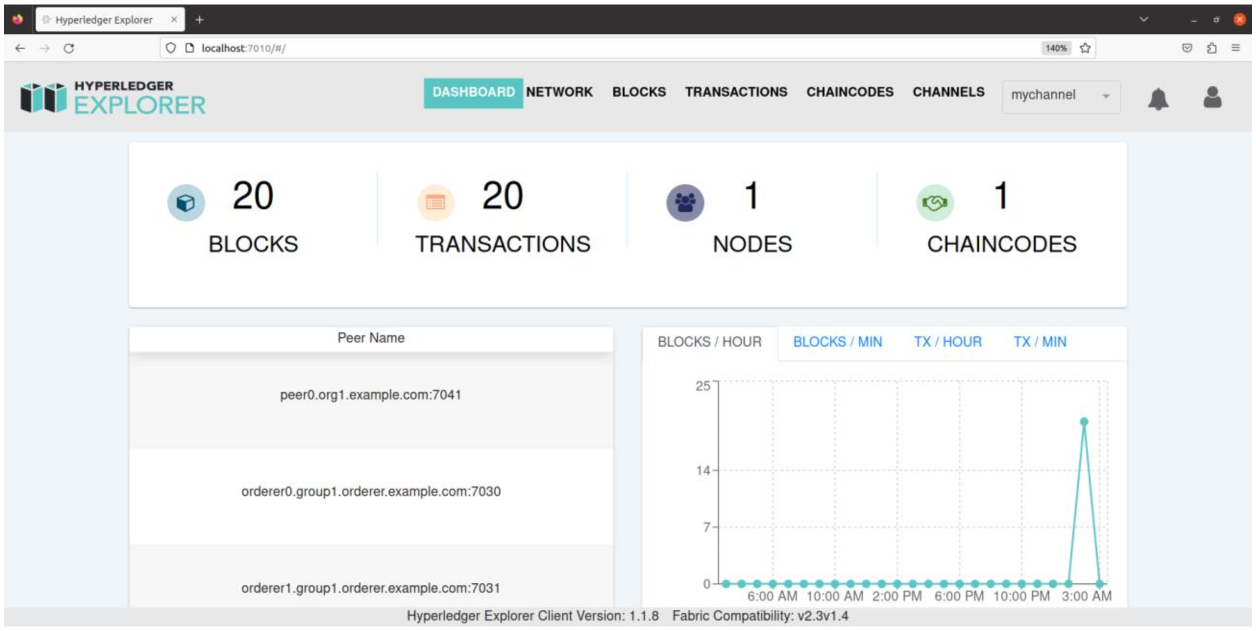
```
{"answer":"This is sample GET endpoint"}
```

Figure 5-22 Demo: The client app works and can invoke the endpoint.

5.5 Blockchain Explorer – System Traceability

The Blockchain Explorer is used to demonstrate that every interaction with the smart contract deployed on the Hyperledger Fabric network is recorded in the underlying ledger of each peer node. It is a web-based app for browsing the blocks inside the blockchain [25].

When browsing the Hyperledger Fabric blockchain created for the demo in the previous chapter, the blockchain-explorer tool's dashboard showed that there were 20 blocks with 20 transactions committed; there was one node and one chaincode deployed on the network:

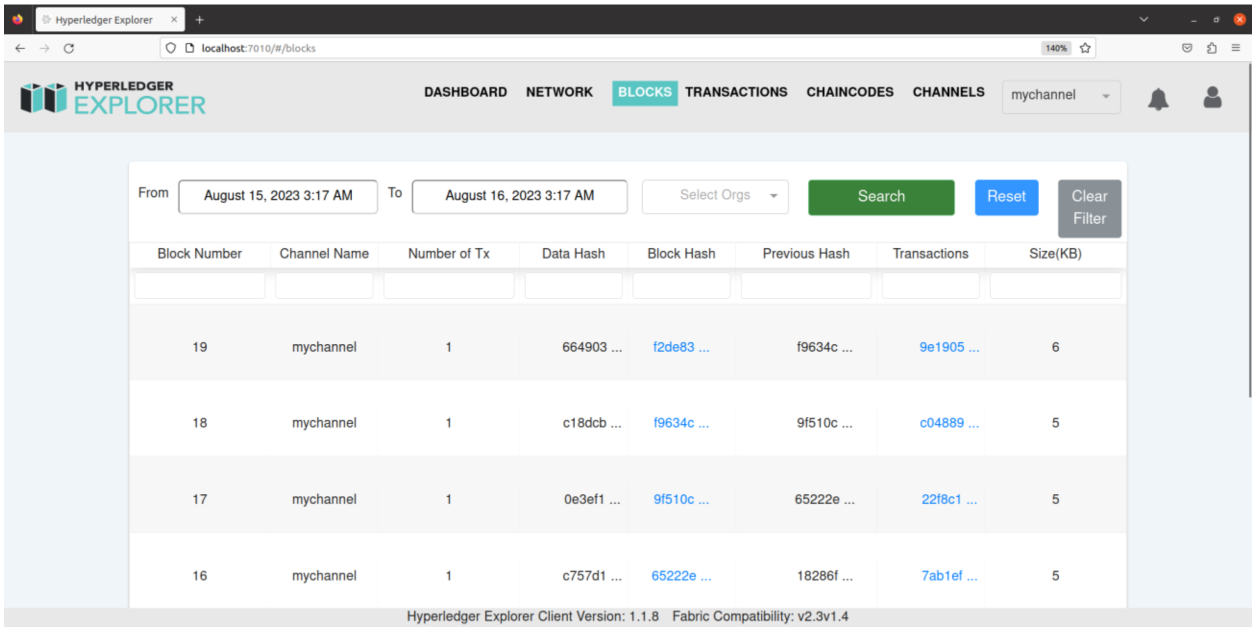


Peer Name
peer0.org1.example.com:7041
orderer0.group1.orderer.example.com:7030
orderer1.group1.orderer.example.com:7031

Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-23 Blockchain Explorer dashboard.

On the blocks section, all blocks in the blockchain can be viewed:



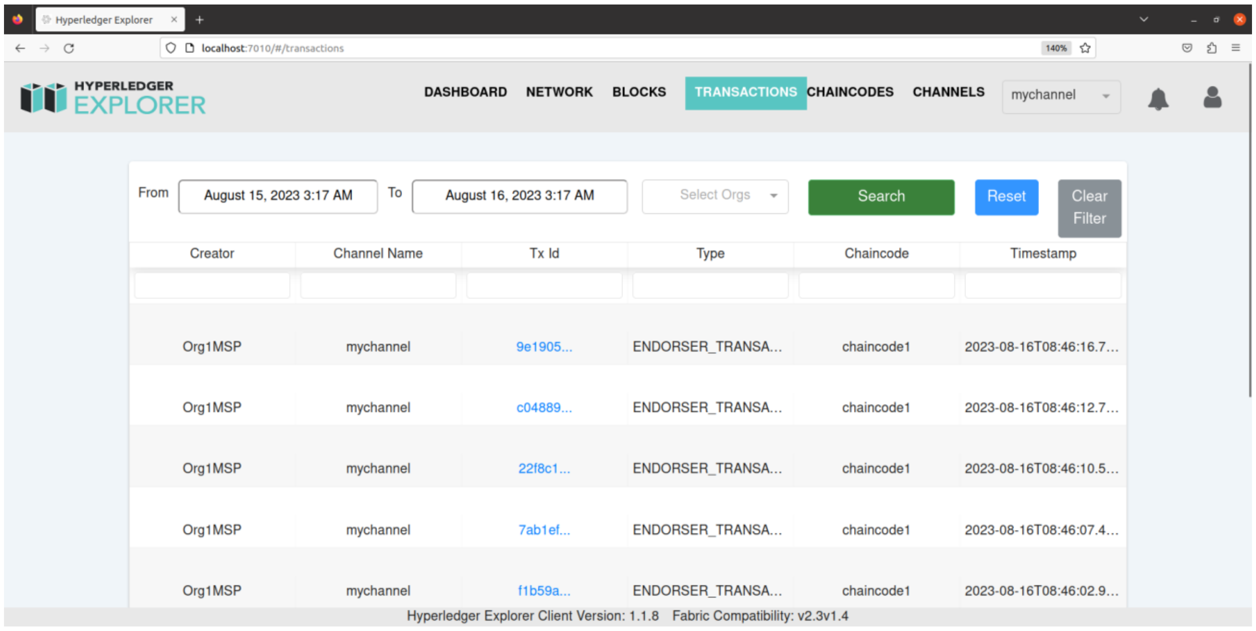
From August 15, 2023 3:17 AM To August 16, 2023 3:17 AM

Block Number	Channel Name	Number of Tx	Data Hash	Block Hash	Previous Hash	Transactions	Size(KB)
19	mychannel	1	664903 ...	f2de83 ...	f9634c ...	9e1905 ...	6
18	mychannel	1	c18dcb ...	f9634c ...	9f510c ...	c04889 ...	5
17	mychannel	1	0e3ef1 ...	9f510c ...	65222e ...	22f8c1 ...	5
16	mychannel	1	c757d1 ...	65222e ...	18286f ...	7ab1ef ...	5

Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-24 Blockchain Explorer Blocks view.

In the transactions section, all transactions in the blockchain can be viewed:

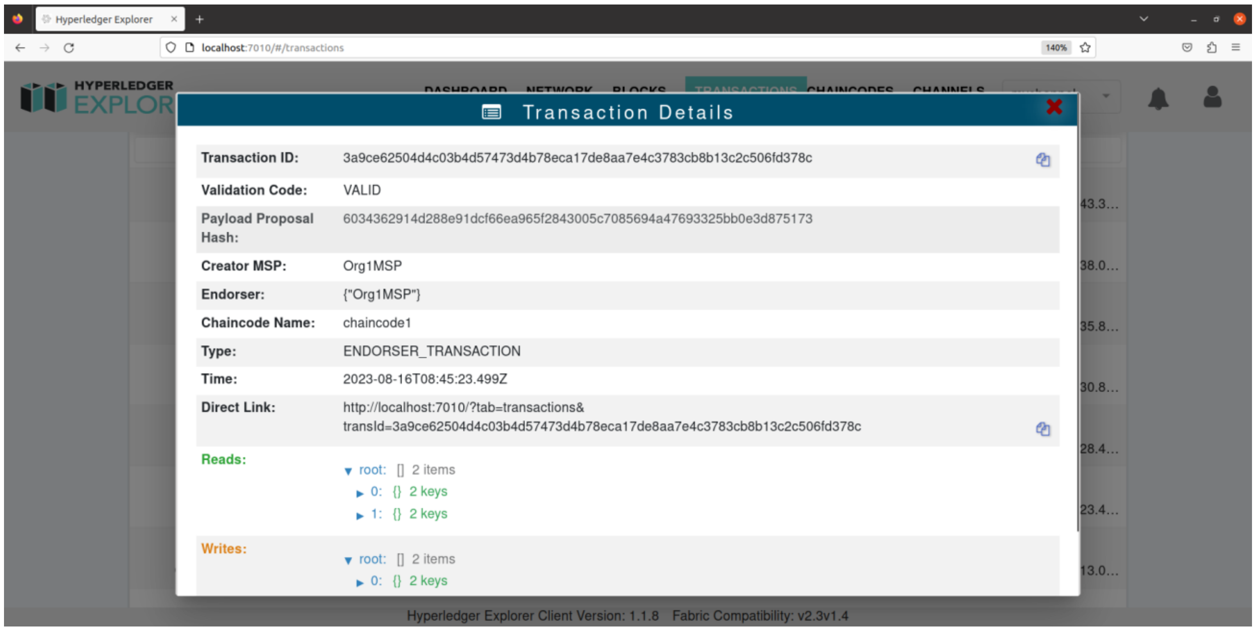

From August 15, 2023 3:17 AM To August 16, 2023 3:17 AM

Creator	Channel Name	Tx Id	Type	Chaincode	Timestamp
Org1MSP	mychannel	9e1905...	ENDORSER_TRANSA...	chaincode1	2023-08-16T08:46:16.7...
Org1MSP	mychannel	c04889...	ENDORSER_TRANSA...	chaincode1	2023-08-16T08:46:12.7...
Org1MSP	mychannel	22f8c1...	ENDORSER_TRANSA...	chaincode1	2023-08-16T08:46:10.5...
Org1MSP	mychannel	7ab1ef...	ENDORSER_TRANSA...	chaincode1	2023-08-16T08:46:07.4...
Org1MSP	mychannel	f1b59a...	ENDORSER_TRANSA...	chaincode1	2023-08-16T08:46:02.9...

Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-25 Blockchain Explorer Transaction view.

In this case, each block contains one transaction. The following is a transaction in a block near the beginning of the blockchain:


Transaction Details

Transaction ID:	3a9ce62504d4c03b4d57473d4b78eca17de8aa7e4c3783cb8b13c2c506fd378c
Validation Code:	VALID
Payload Proposal Hash:	6034362914d288e91dcf66ea965f2843005c7085694a47693325bb0e3d875173
Creator MSP:	Org1MSP
Endorser:	{"Org1MSP"}
Chaincode Name:	chaincode1
Type:	ENDORSER_TRANSACTION
Time:	2023-08-16T08:45:23.499Z
Direct Link:	http://localhost:7010/?tab=transactions& transId=3a9ce62504d4c03b4d57473d4b78eca17de8aa7e4c3783cb8b13c2c506fd378c

```
Reads:
  ▼ root: [] 2 items
    ▶ 0: {} 2 keys
    ▶ 1: {} 2 keys
Writes:
  ▼ root: [] 2 items
    ▶ 0: {} 2 keys
```

Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-26 Blockchain explorer inspecting a transaction.

When inspecting its reads and writes values, we see that this transaction was about registering the API provider.

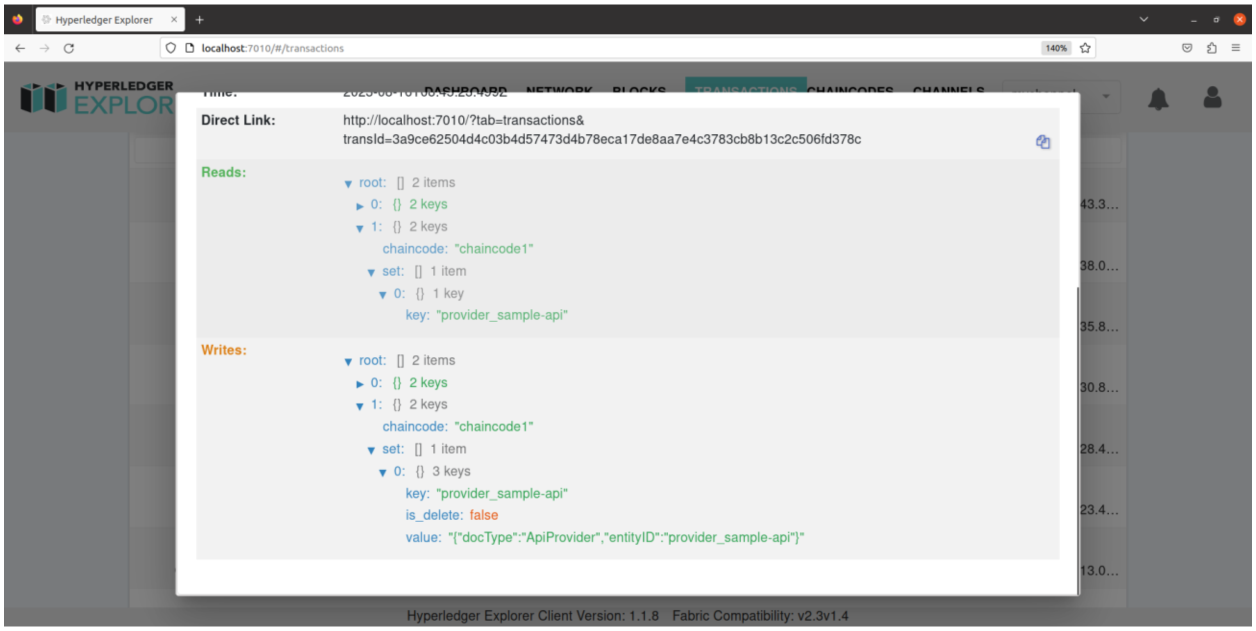

```
Direct Link:   http://localhost:7010/?tab=transactions&
               transId=3a9ce62504d4c03b4d57473d4b78eca17de8aa7e4c3783cb8b13c2c506fd378c
Reads:         ▼ root:  [] 2 items
                 ▶ 0:  {} 2 keys
                 ▼ 1:  {} 2 keys
                     chaincode: "chaincode1"
                   ▼ set:  [] 1 item
                     ▼ 0:  {} 1 key
                         key: "provider_sample-api"
Writes:        ▼ root:  [] 2 items
                 ▶ 0:  {} 2 keys
                 ▼ 1:  {} 2 keys
                     chaincode: "chaincode1"
                   ▼ set:  [] 1 item
                     ▼ 0:  {} 3 keys
                         key: "provider_sample-api"
                         is_delete: false
                         value: "{"docType":"ApiProvider","entityID":"provider_sample-api"}"
```
Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-27 Blockchain explorer viewing the API provider registration event.

There is another transaction showing the step where the API provider added an endpoint in the demo:

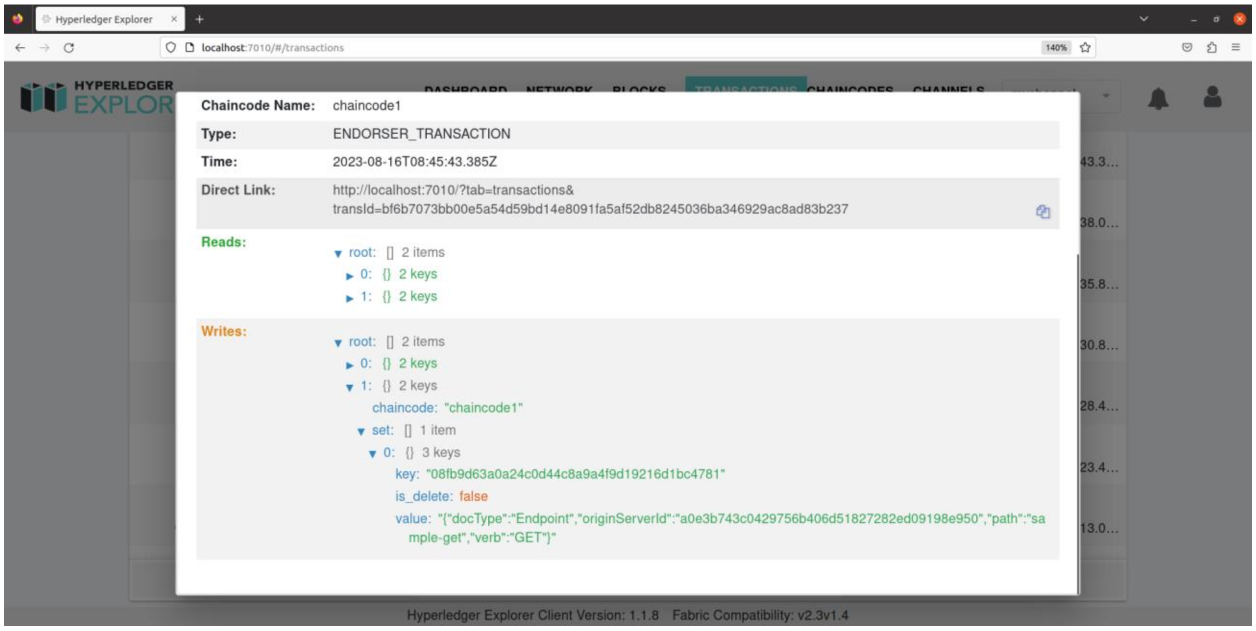

```
Chaincode Name:  chaincode1
Type:            ENDORSER_TRANSACTION
Time:            2023-08-16T08:45:43.385Z
Direct Link:     http://localhost:7010/?tab=transactions&
                 transId=bf6b7073bb00e5a54d59bd14e8091fa5af52db8245036ba346929ac8ad83b237
Reads:           ▼ root:  [] 2 items
                   ▶ 0:  {} 2 keys
                   ▶ 1:  {} 2 keys
Writes:          ▼ root:  [] 2 items
                   ▶ 0:  {} 2 keys
                   ▼ 1:  {} 2 keys
                       chaincode: "chaincode1"
                     ▼ set:  [] 1 item
                       ▼ 0:  {} 3 keys
                           key: "08fb9d63a0a24c0d44c8a9a4f9d19216d1bc4781"
                           is_delete: false
                           value: "{"docType":"Endpoint","originServerId":"a0e3b743c0429756b406d51827282ed09198e950","path":"sa
                             mple-get","verb":"GET"}"
```
Hyperledger Explorer Client Version: 1.1.8 Fabric Compatibility: v2.3v1.4

Figure 5-28 Blockchain explorer viewing the event where the API provider added an endpoint.

Similarly, there are transactions about requesting access to the endpoint, granting access to the endpoint, etc.

6. PERFORMANCE EVALUATION

This chapter presents the experiments to evaluate the system's performance and answer the research questions. First, the hardware setup is presented, followed by the strategy and tools used for the experiments, followed by the experiments and graphs.

6.1 Methodology

6.1.1 Hardware Setup

The system was deployed to virtual machines (VM) running on a private cloud infrastructure. Then, a suite of performance evaluation scripts was run against the system to record statistics in CSV format. From the statistics, the graphs were plotted. The performance evaluation scripts suite contains the following:

- K6 scripts: these are written using K6, an open-source tool from Grafana labs for load-testing and validating performance and reliability [24]. These scripts create and manage virtual client app to load the system with traffic.
- Preparation scripts: these are written using the Mocha test framework to bring the system to a certain state. After that, the K6 scripts take over and load test the system.

The following diagram presents the VMs created on the private cloud and the corresponding system components deployed:

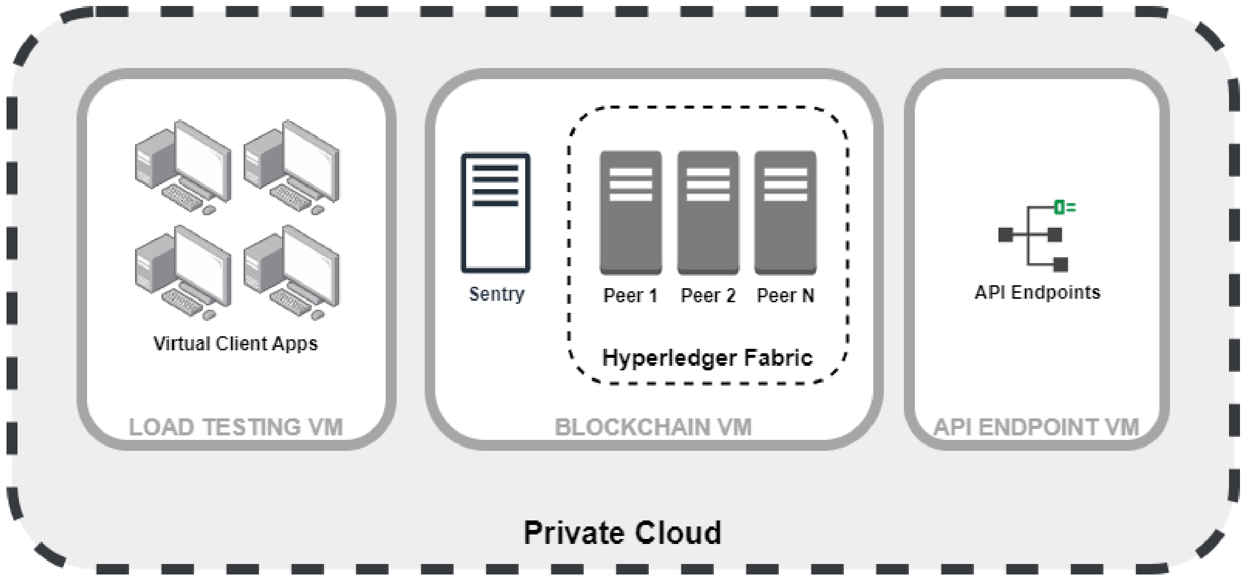


Figure 6-1 Hardware setup for experiments.

As shown in the diagram. There are three VMs in total:

- The Load-Testing VM is for running the tool K6 which creates and manages the virtual client apps for load-testing the system.
- The Blockchain VM is for running Sentry and the Hyperledger Fabric network; the network can be configured to have up to 9 peers, depending on the experiment.
- The API Endpoint VM is where the server with the protected API endpoints is deployed.

The following table shows the hardware specifications of the VMs:

Table 6-1 Experiment VMs' Hardware Specifications

VMs	Hardware specifications
Load-Testing VM	OS: Ubuntu CPU: 8 cores Memory: 32GB

Blockchain VM	OS: Ubuntu CPU: 4 cores Memory: 8GB
API Endpoint VM	OS: Ubuntu CPU: 4 cores Memory: 8GB

6.1.2 System Preparation for Accessing Endpoints

The primary focus of all experiments is on the “Accessing endpoint” workflow described in Chapter 4, where the client apps send requests to the protected endpoints. This workflow determines the overall efficiency because the system must serve the most traffic. To get a fresh system to the state where the workflow can occur, preparation scripts send HTTP requests to Sentry and carry out the following prerequisite steps of the “Accessing endpoint” workflow:

- Register an API provider account.
- Register the VM that hosts the endpoints to be a protected server.
- Register the endpoints.
- Register a client account.
- Request endpoint access for the client account.
- Approve the endpoint access request.

After the access has been granted, the credentials are stored for the virtual client apps created by K6 to use so that they can send requests to the protected endpoints via Sentry.

6.2 Experiments

6.2.1 Experiment 1: Latency

Latency in this context means how long it takes for the client app to send a request to one of the protected API endpoints and receive a response. This experiment aims to determine two key aspects: the overhead latency of the system and the latency specific to the Hyperledger Fabric component within the system, which correspond to *research questions 1 and 2,* respectively.

To address research question 1, virtual client applications were used to simulate user interactions and generate load, and the round-trip time of these interactions was measured and compared. To begin with, the Blockchain VM was bypassed, meaning that the load was sent to an API endpoint on the API Endpoint VM directly *(Direct API access scenario)*. After collecting the first set of statistics, the Blockchain VM was enabled, and the loading process was repeated, with the load passing through both the Blockchain VM and the API Endpoint VM *(Blockchain-enabled access control scenario)*. Once research question 1 was addressed, research question 2 was explored. The Blockchain VM remained enabled to do this, but the Hyperledger Fabric component was bypassed *(Blockchain-bypassed access control scenario)*. Once again, virtual client applications were used to load the system. By comparing the roundtrip time statistics obtained from both the full system and the system with the Hyperledger Fabric component bypassed, research question 2 was addressed.

6.2.2 Overhead latency of the system

The detailed steps for the Direct API access scenario were as follows: K6 running on the load-testing VM initiated with one virtual client app. The virtual client app went through multiple iterations for 60 seconds. In each iteration, it sent a request to the API endpoint and reported the round-trip time once it got the response. Every round-trip time was parsed and stored, resulting in a CSV file of thousands of iterations and their corresponding round-trip time after 60 seconds, which was used to plot scatter charts (notice that the dashed line highlights the zone between Q1 and Q3):

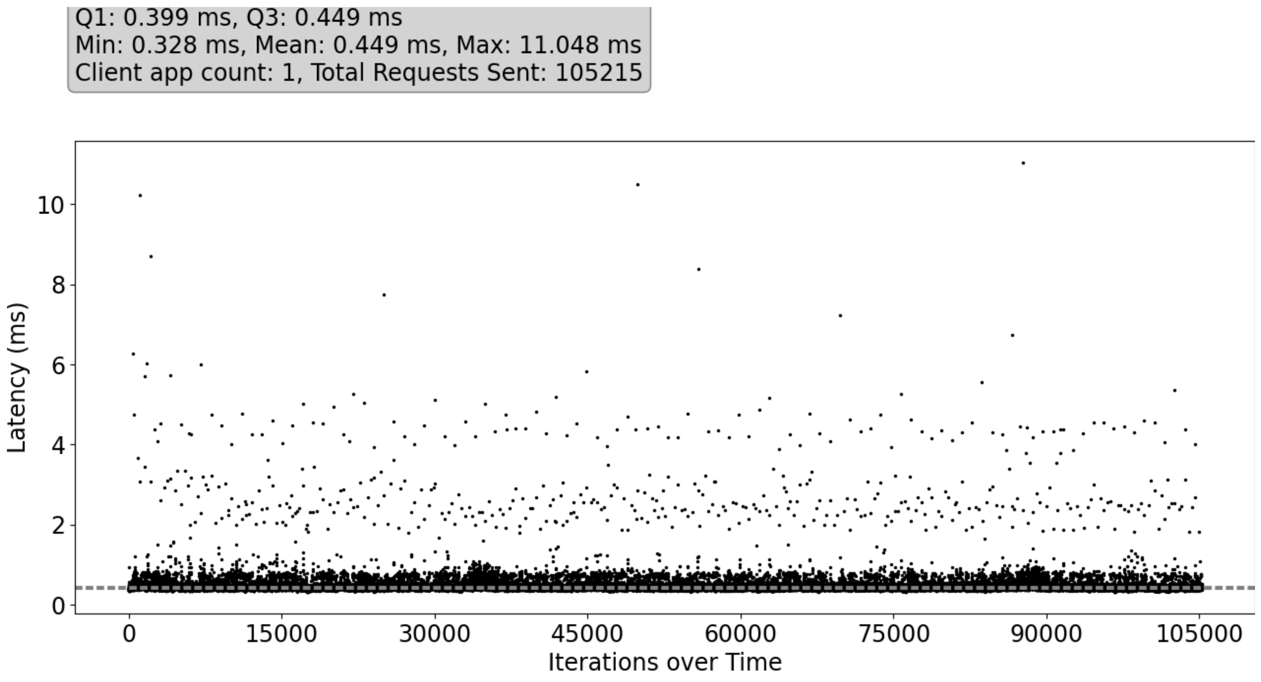


Figure 6-2 Direct API access latency for one client app

The testing procedure was repeated with more sets of virtual client apps, resulting in the following scatter charts:

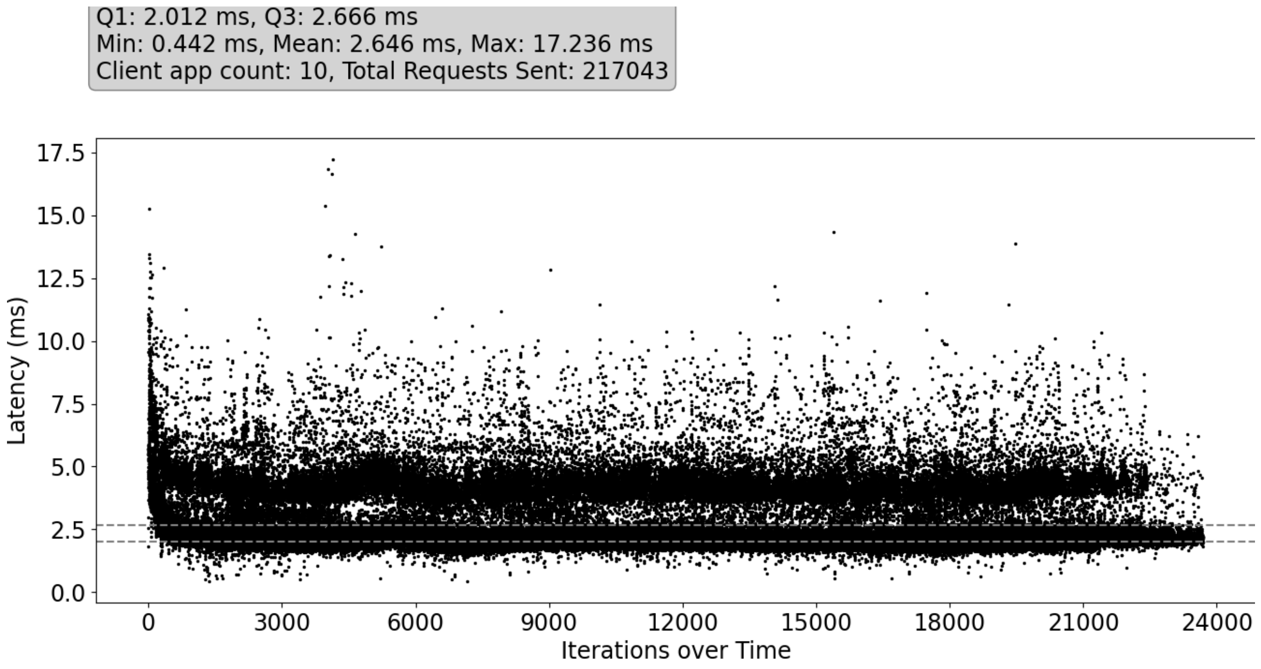


Figure 6-3 Direct API access latency for ten client apps

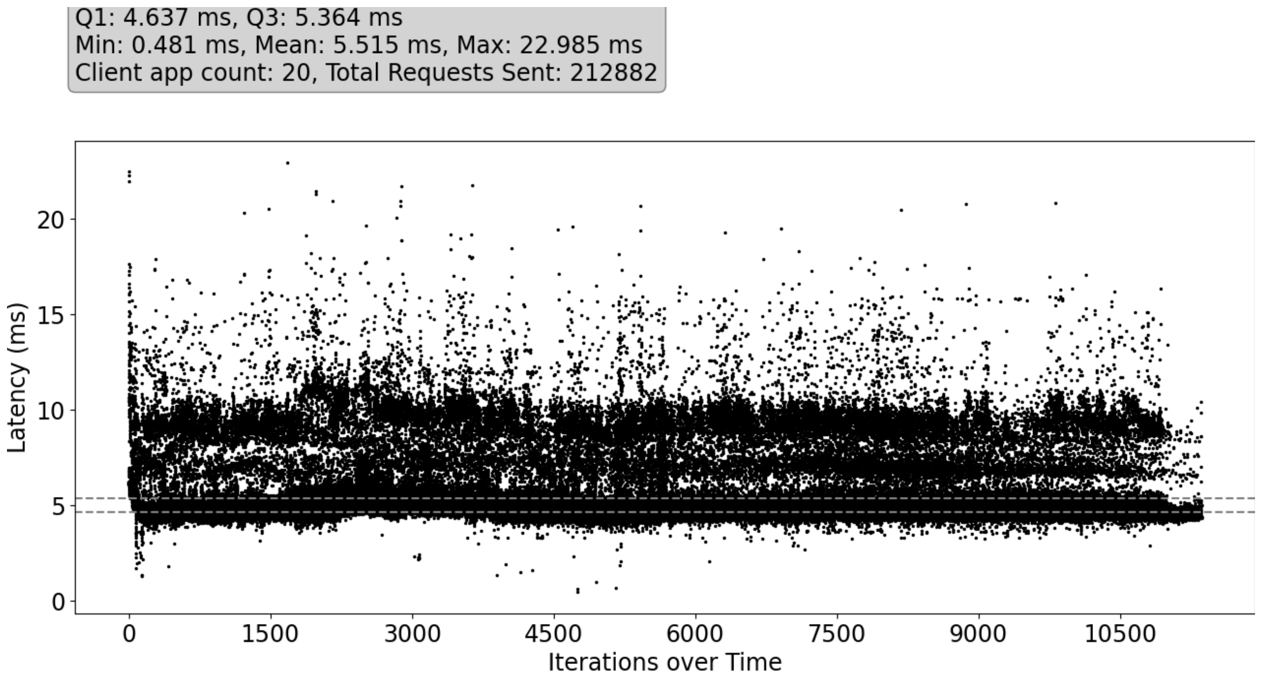


Figure 6-4 Direct API access latency for 20 client apps

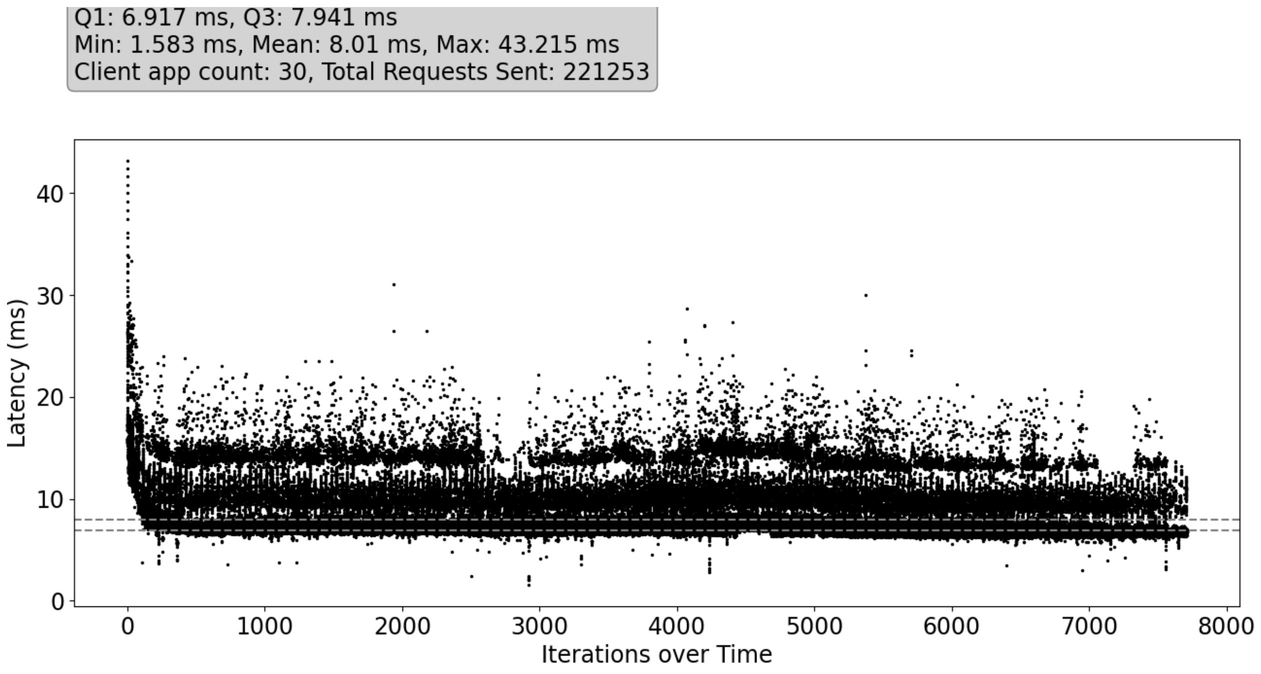


Figure 6-5 Direct API access latency for 30 client apps

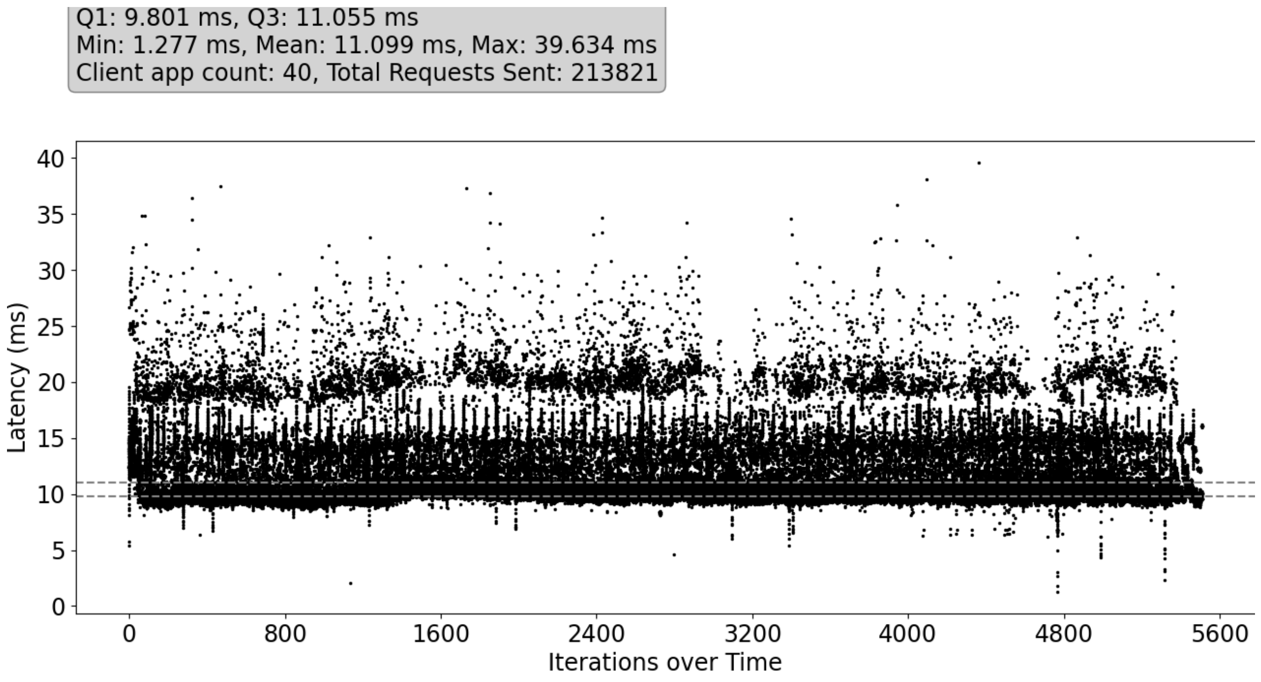


Figure 6-6 Direct API access latency for 40 client apps

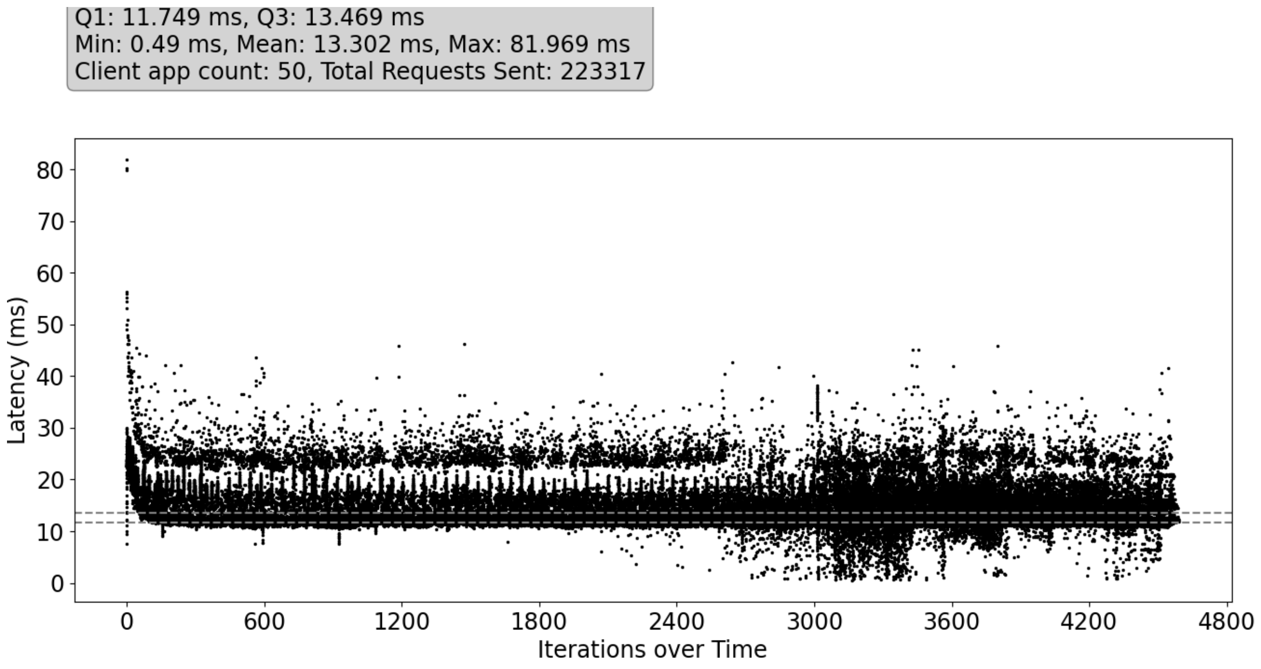


Figure 6-7 Direct API access latency for 50 client apps

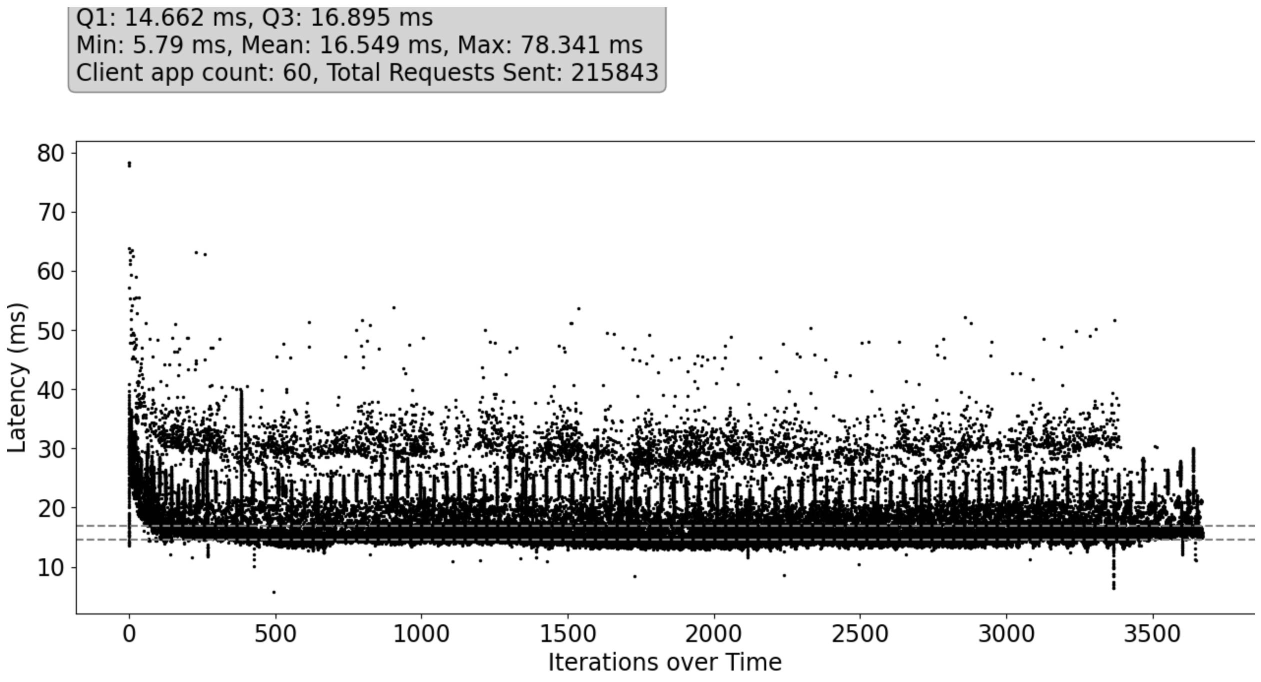


Figure 6-8 Direct API access latency for 60 client apps

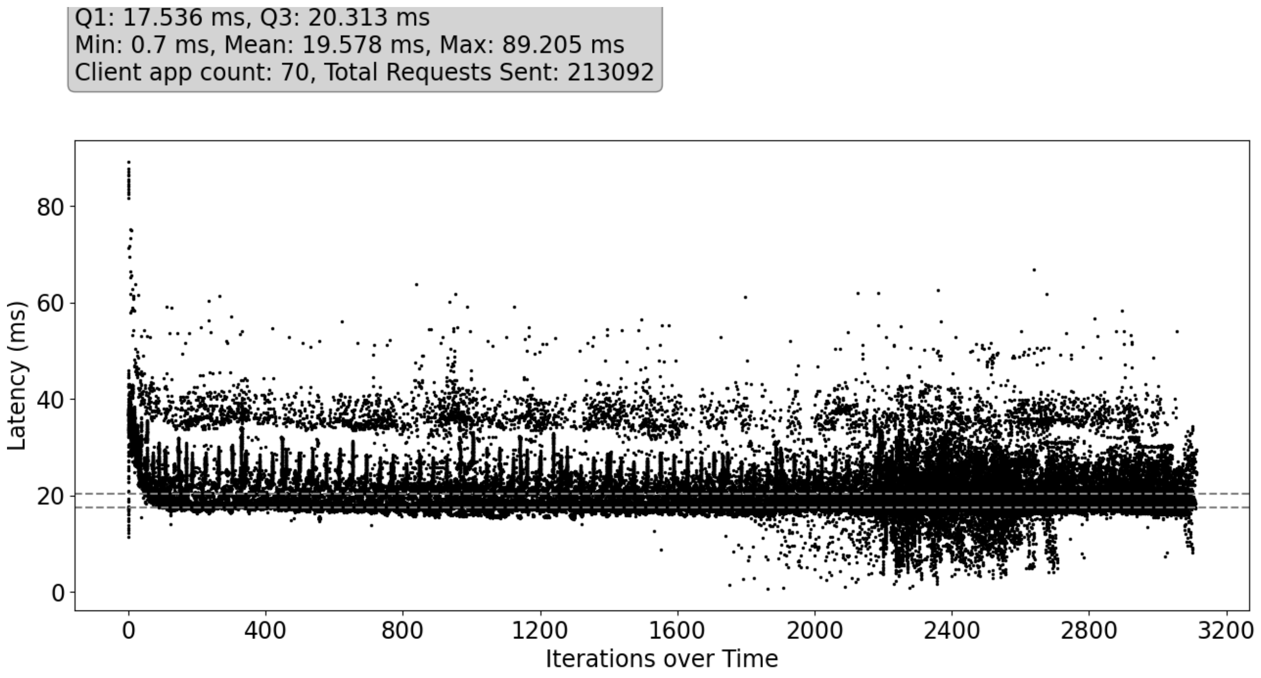


Figure 6-9 Direct API access latency for 70 client apps

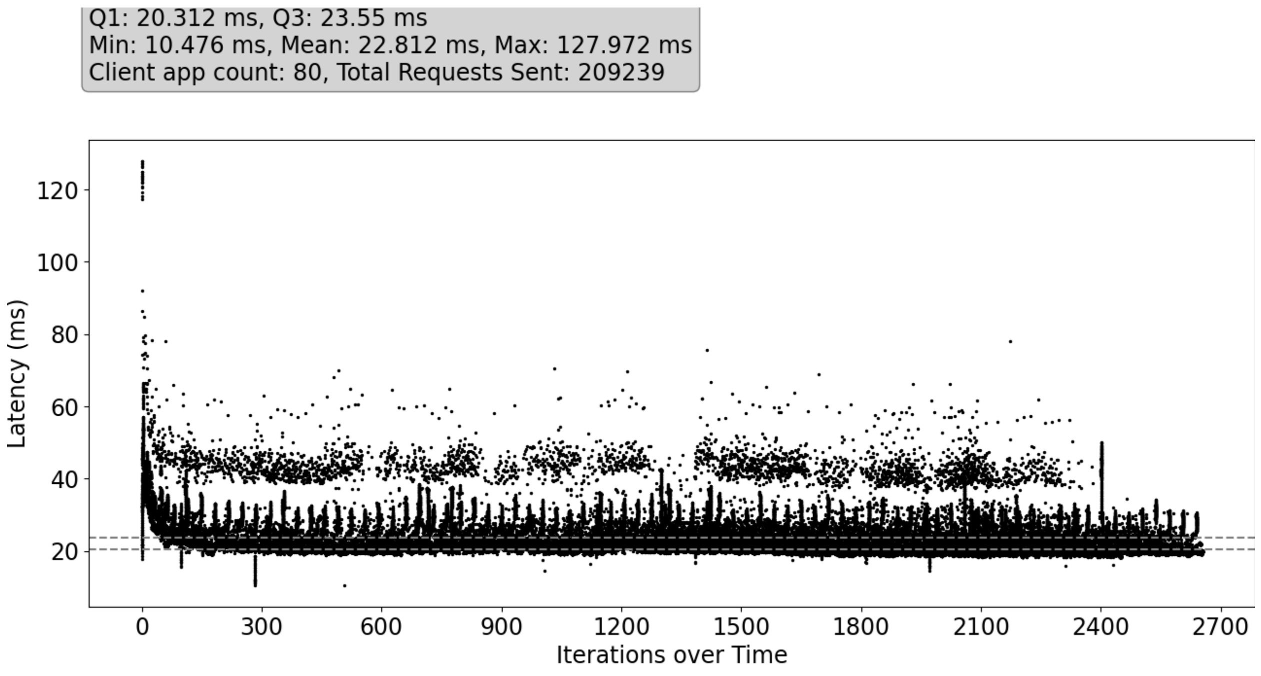

Q1: 20.312 ms, Q3: 23.55 ms
Min: 10.476 ms, Mean: 22.812 ms, Max: 127.972 ms
Client app count: 80, Total Requests Sent: 209239

Figure 6-10 Direct API access latency for 80 client apps

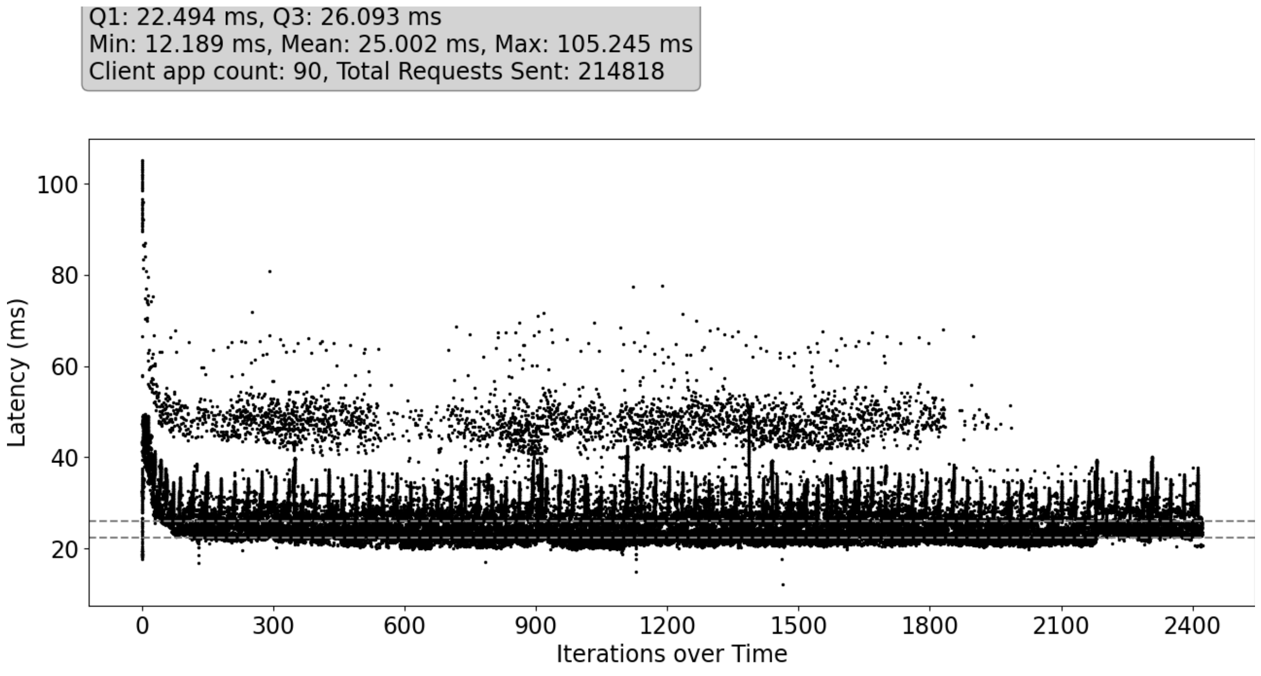

Q1: 22.494 ms, Q3: 26.093 ms
Min: 12.189 ms, Mean: 25.002 ms, Max: 105.245 ms
Client app count: 90, Total Requests Sent: 214818

Figure 6-11 Direct API access latency for 90 client apps

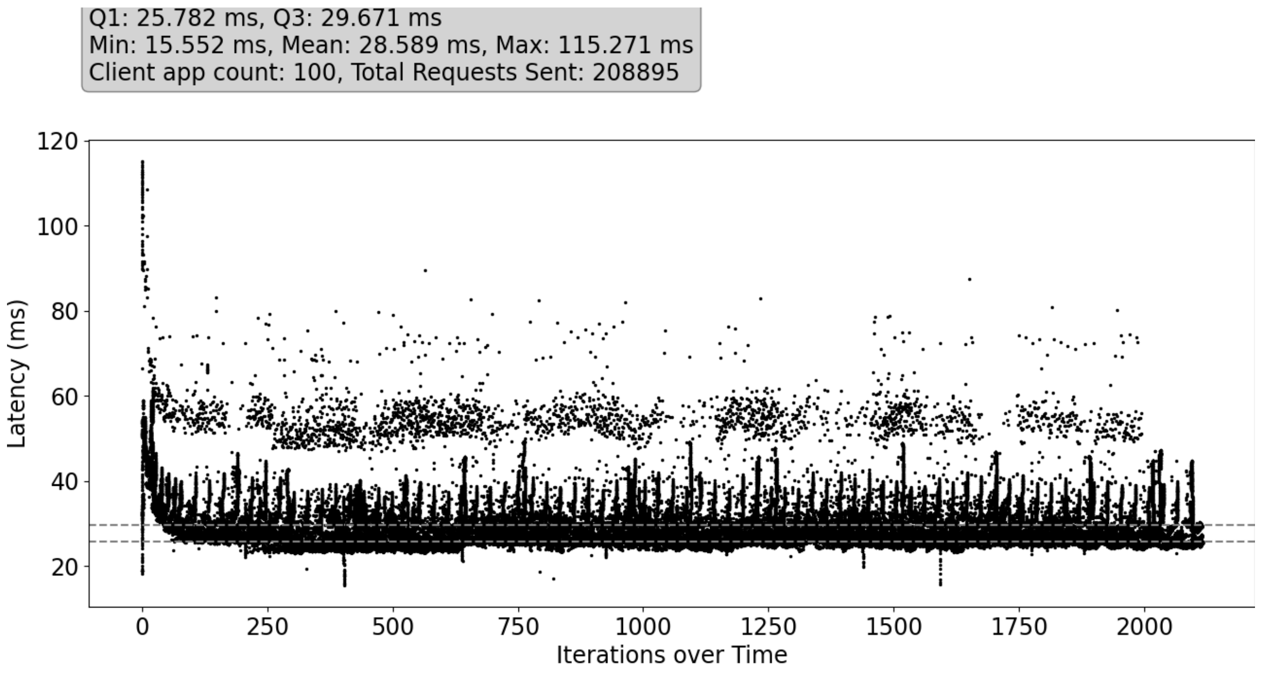


Figure 6-12 Direct API access latency for 100 client apps

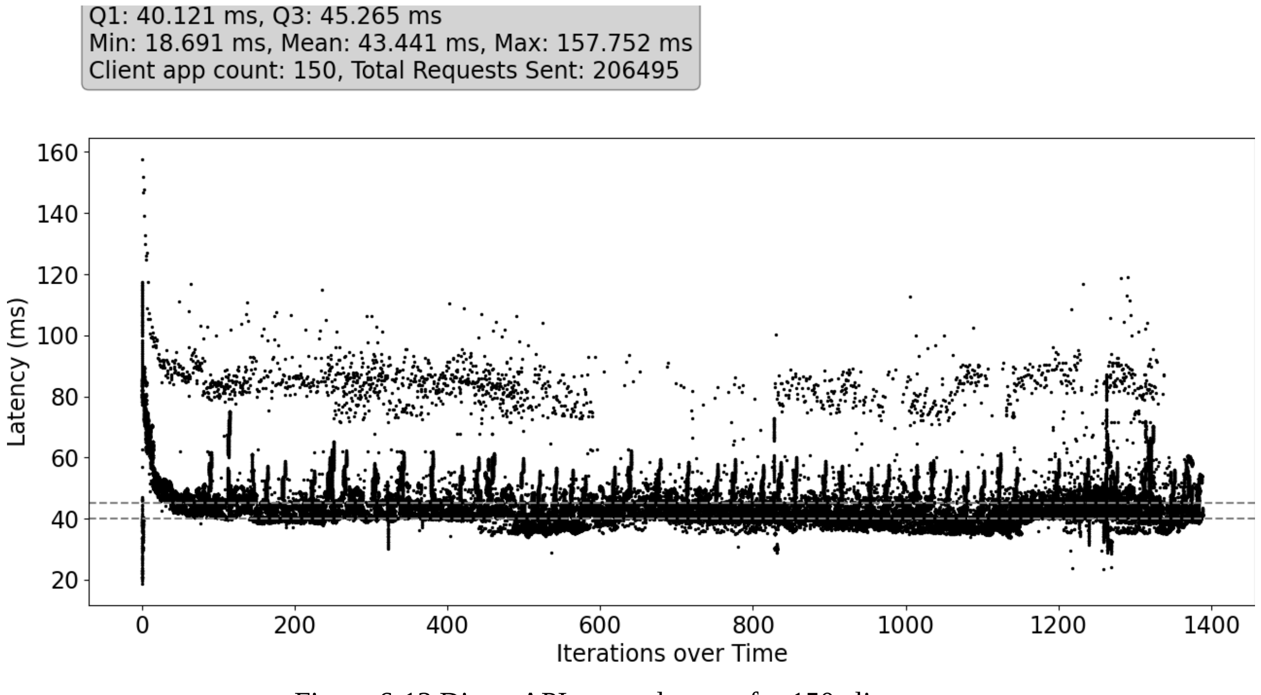


Figure 6-13 Direct API access latency for 150 client apps

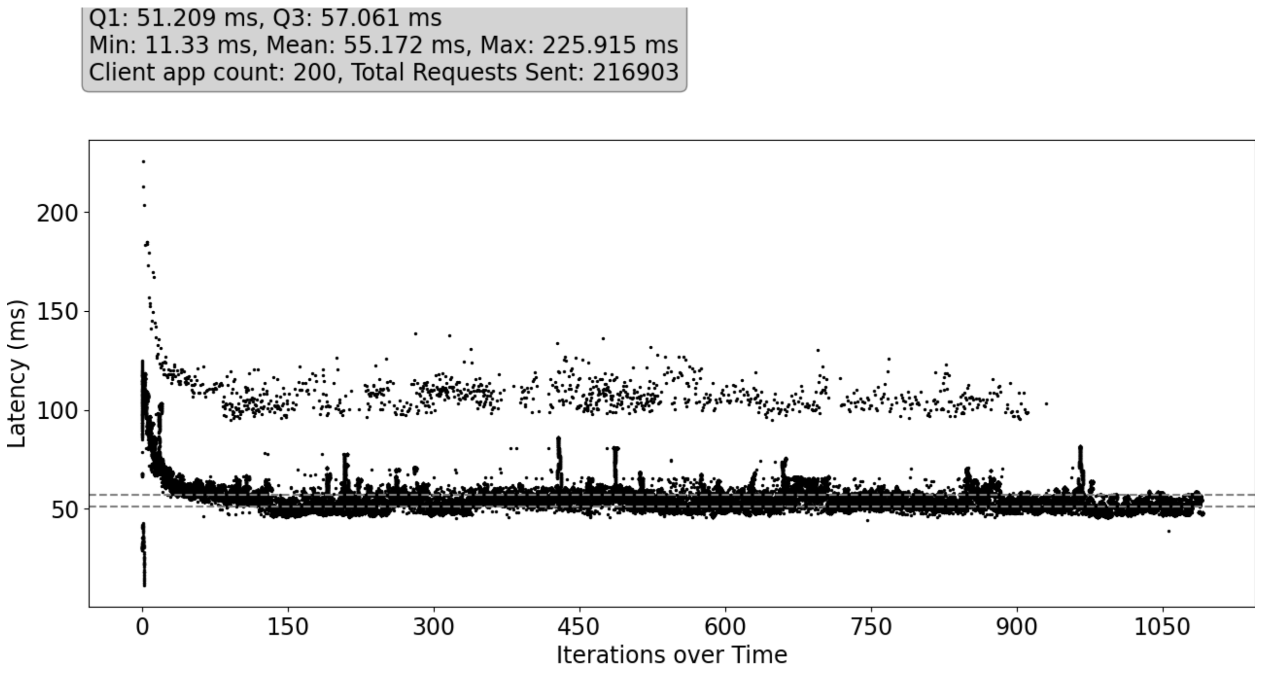


Figure 6-14 Direct API access latency for 200 client apps

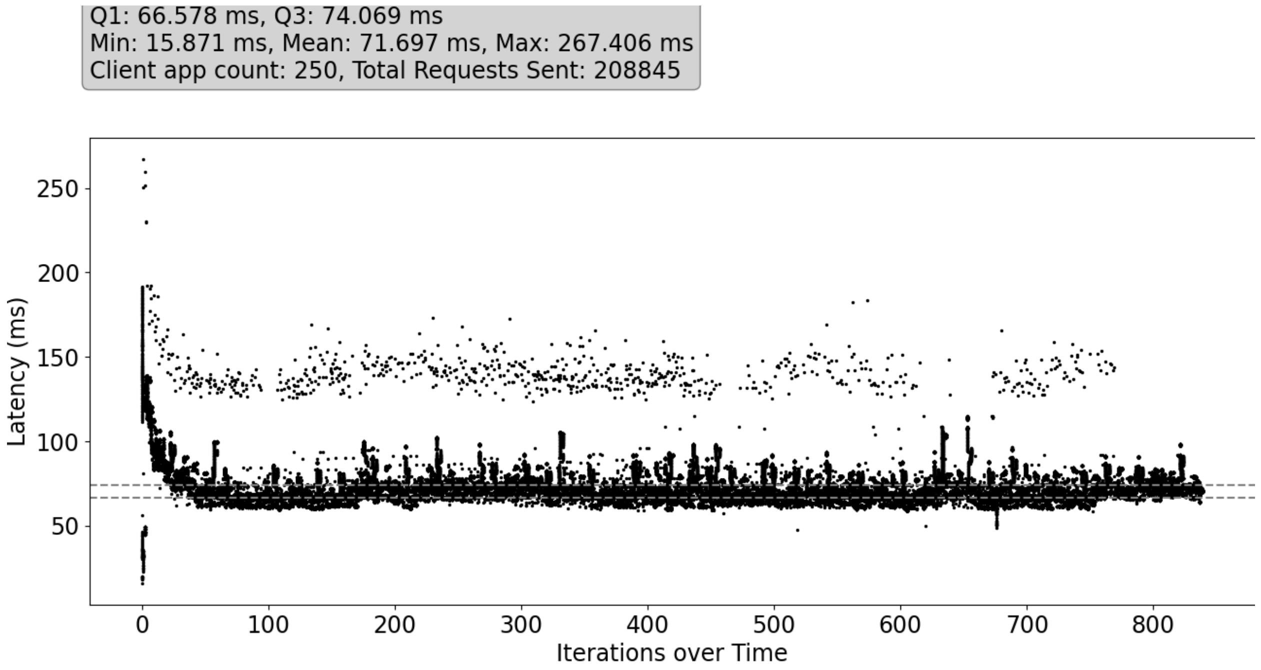


Figure 6-15 Direct API access latency for 250 client apps

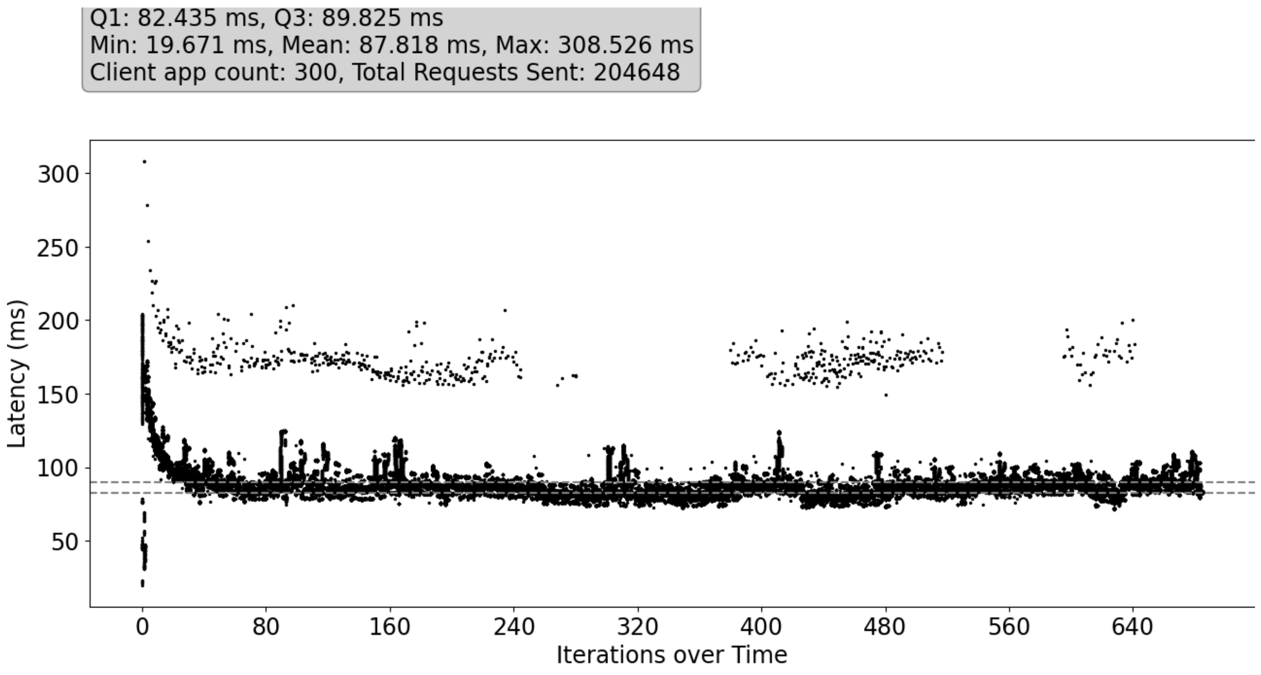


Figure 6-16 Direct API access latency for 300 client apps

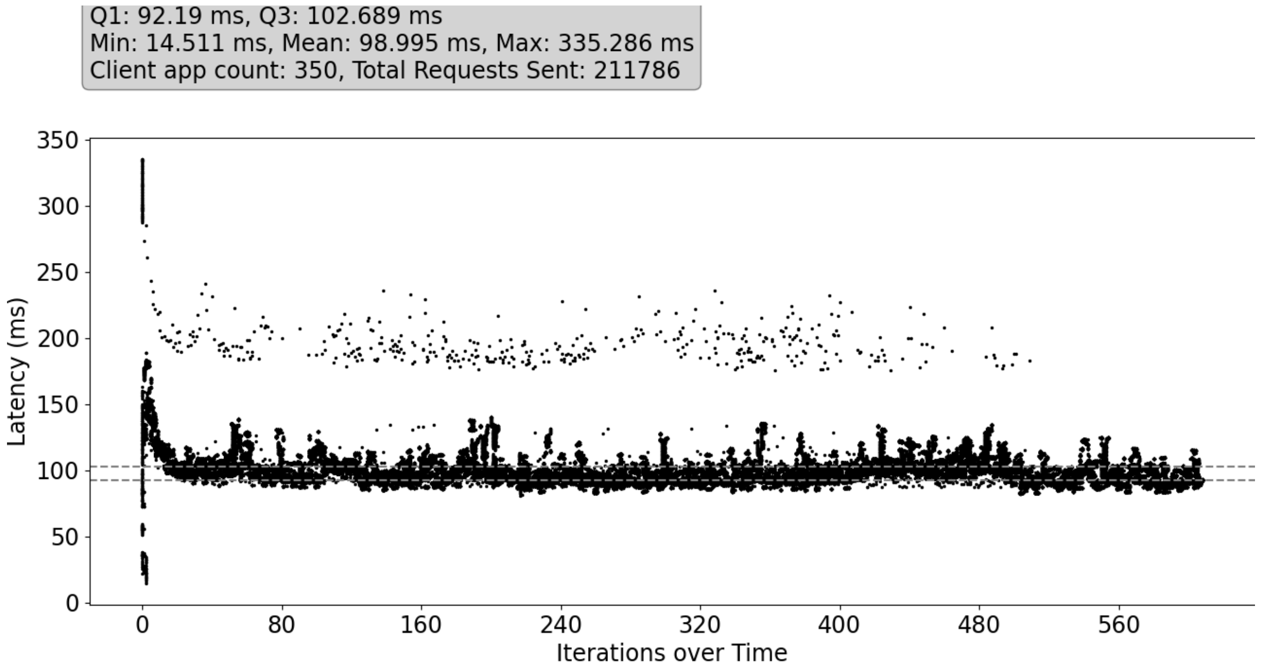


Figure 6-17 Direct API access latency for 350 client apps

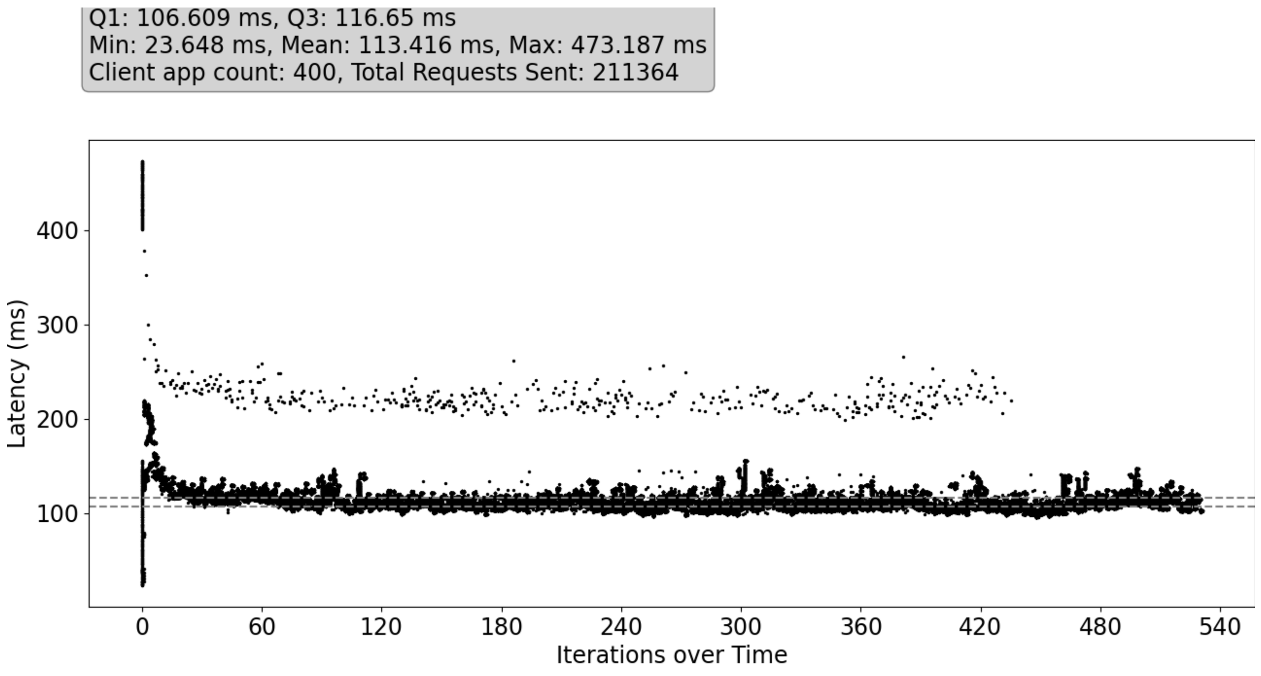


Figure 6-18 Direct API access latency for 400 client apps

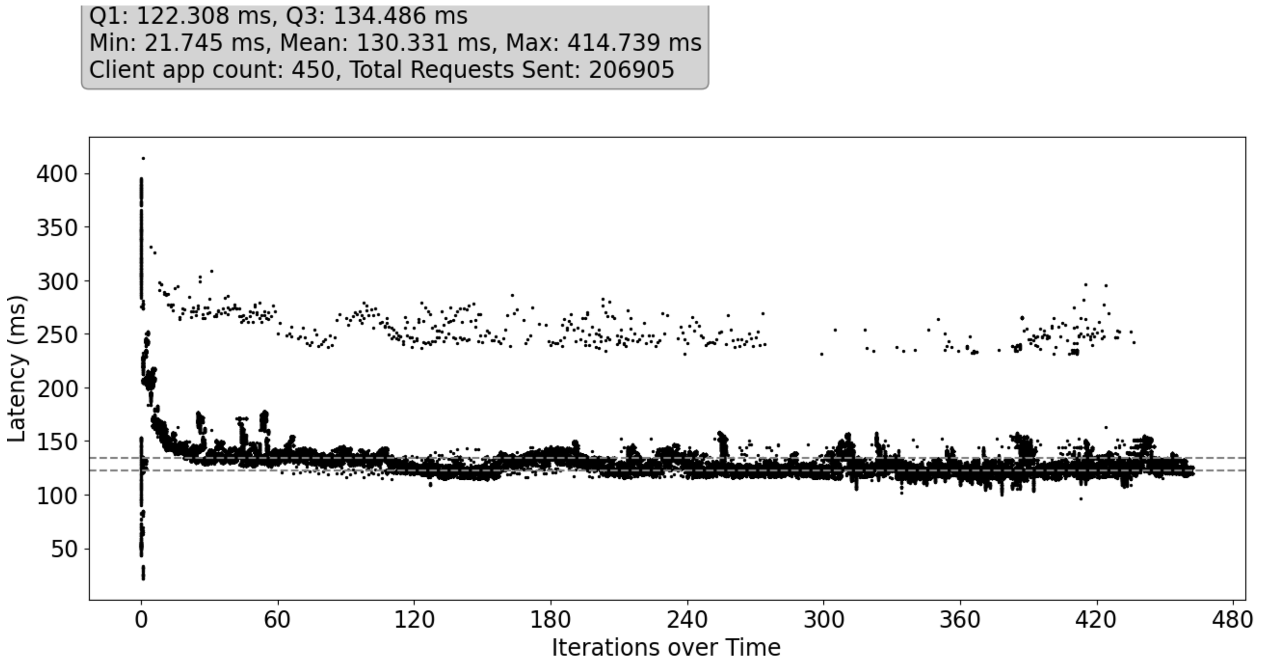


Figure 6-19 Direct API access latency for 450 client apps

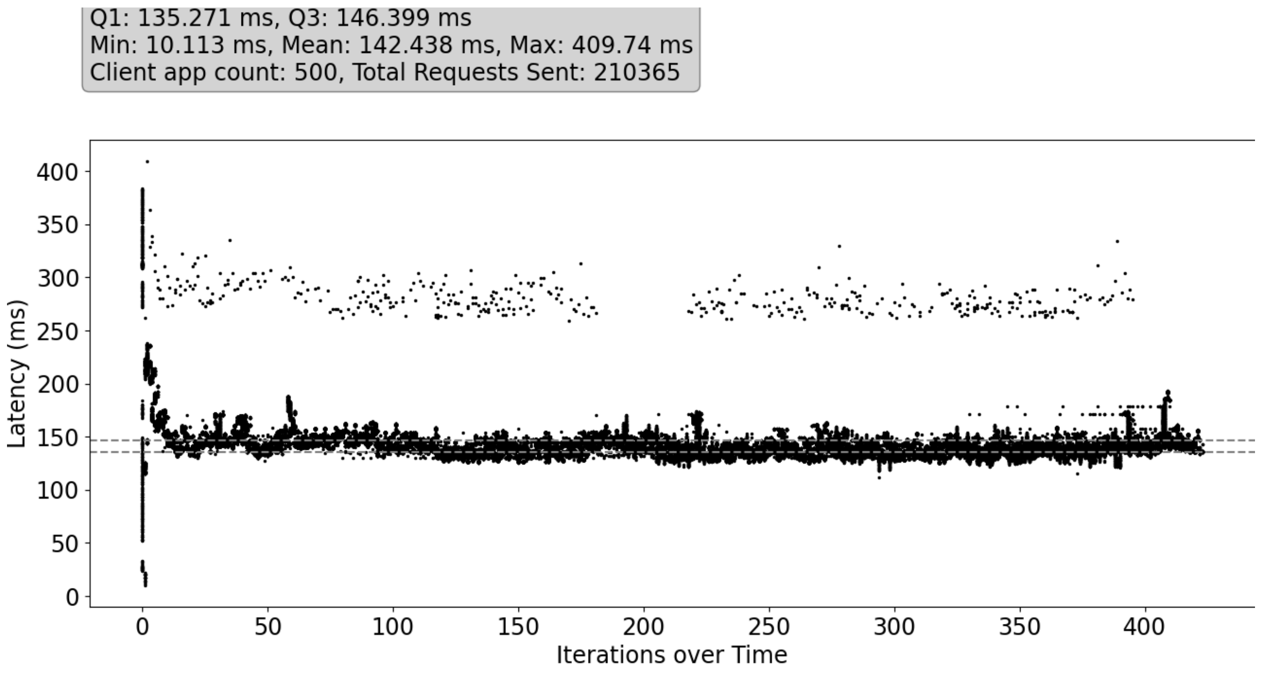


Figure 6-20 Direct API access latency for 500 client apps

At this point, the Blockchain-enabled access control scenario took over. With the three VMs in the private cloud environment, the Blockchain VM was enabled and configured with one peer node in the Hyperledger Fabric network. Subsequently, once access to the protected API endpoints had been granted, K6 running on the load-testing VM started loading the system. After it has finished loading the system with multiple sets of virtual client apps, the following scatter charts were plotted:

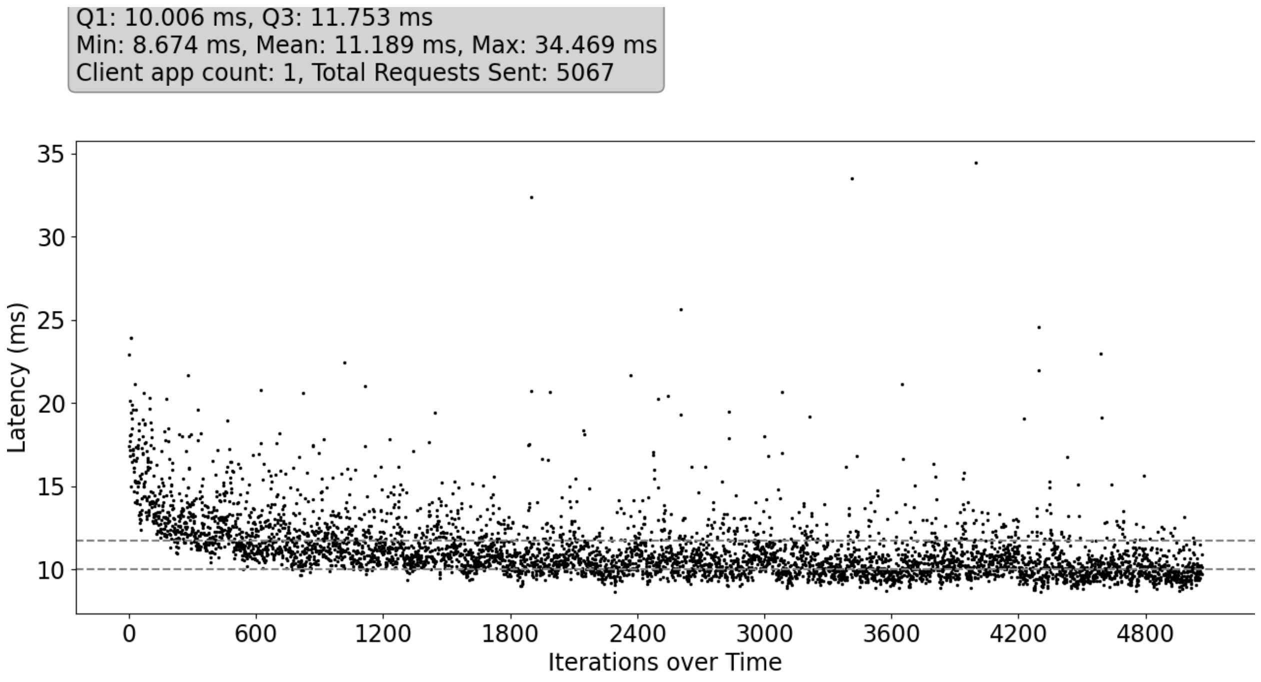


Figure 6-21 Blockchain-Enabled Access Control latency for one client app

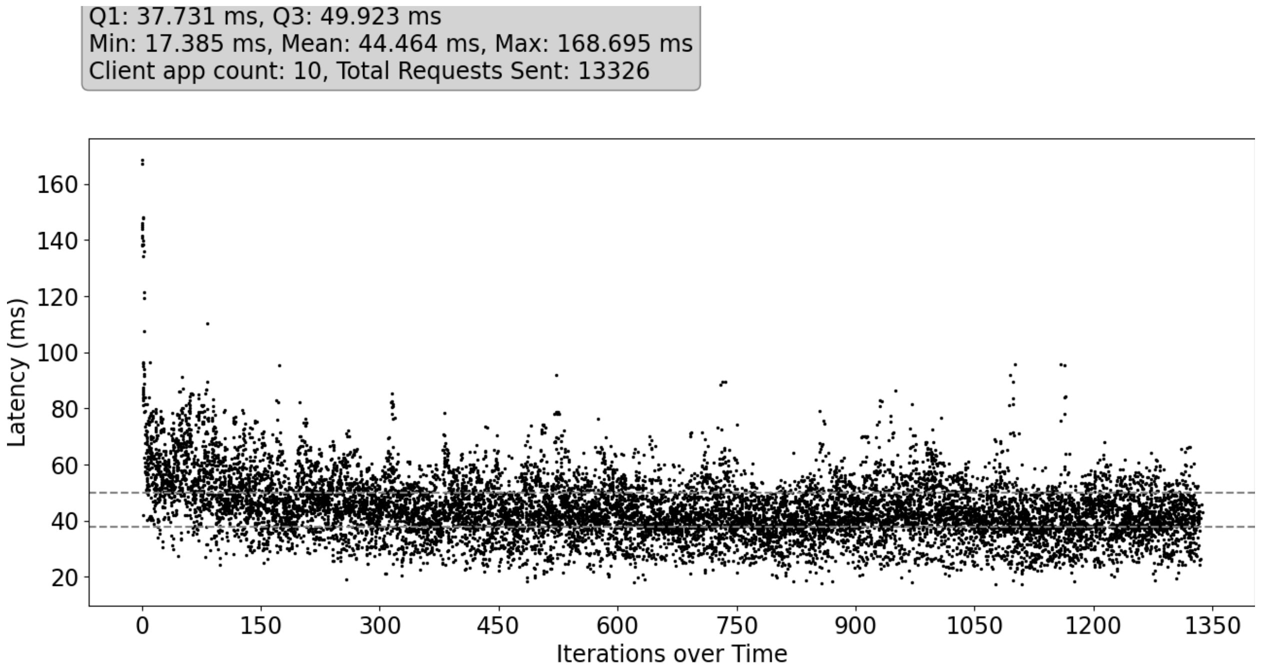


Figure 6-22 Blockchain-Enabled Access Control latency for ten client apps

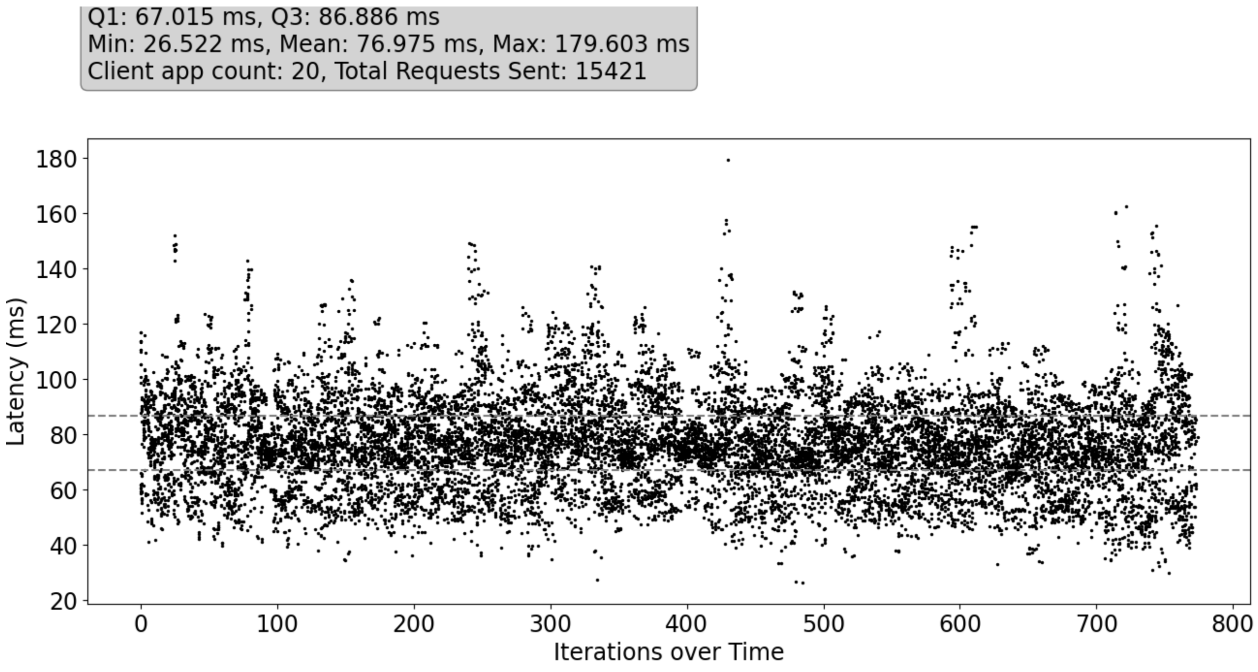


Figure 6-23 Blockchain-Enabled Access Control latency for 20 client apps

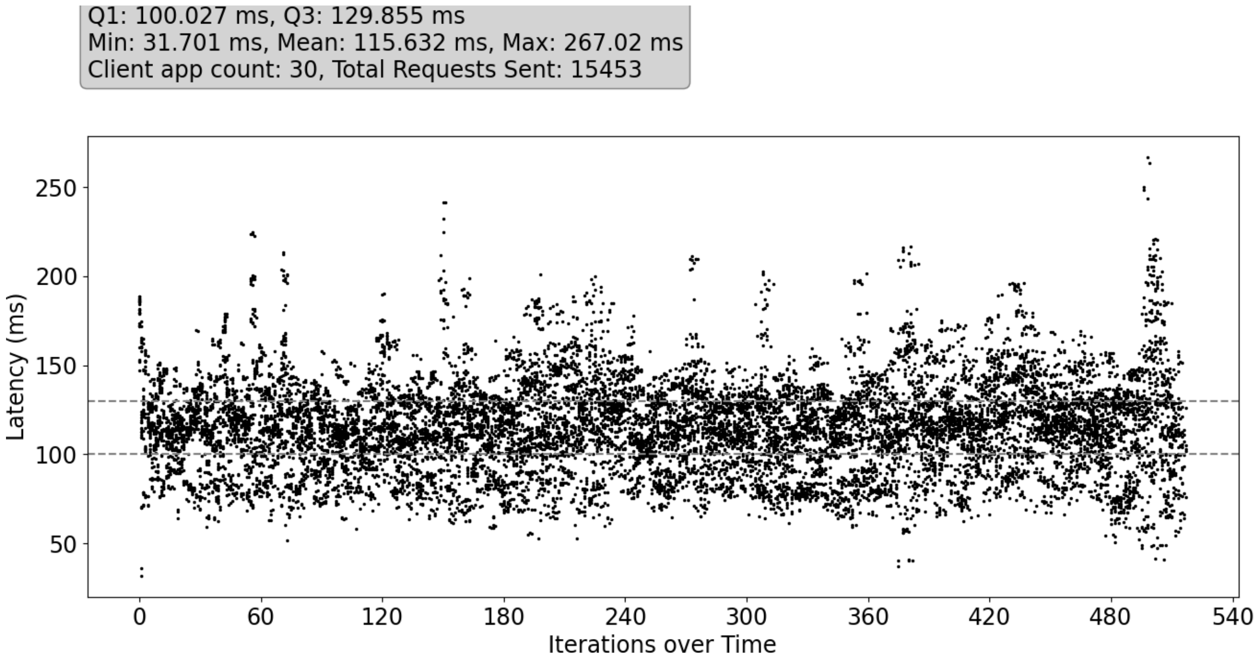


Figure 6-24 Blockchain-Enabled Access Control latency for 30 client apps

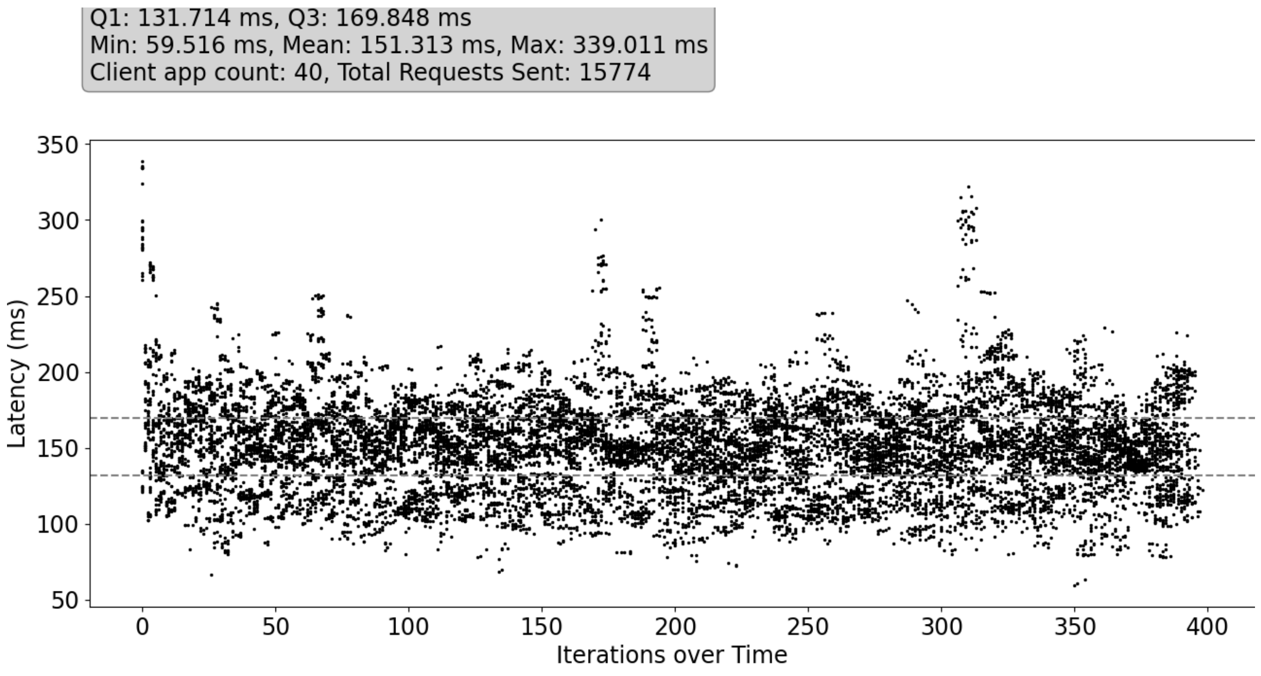


Figure 6-25 Blockchain-Enabled Access Control latency for 40 client apps

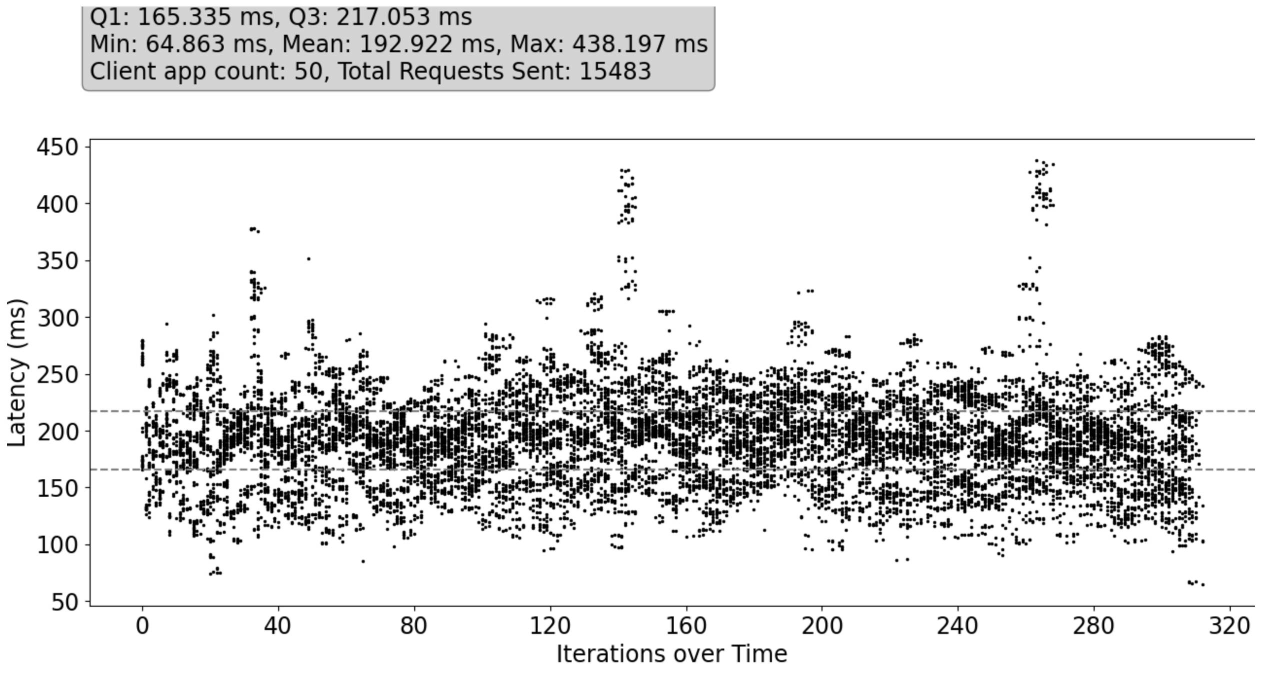


Figure 6-26 Blockchain-Enabled Access Control latency for 50 client apps

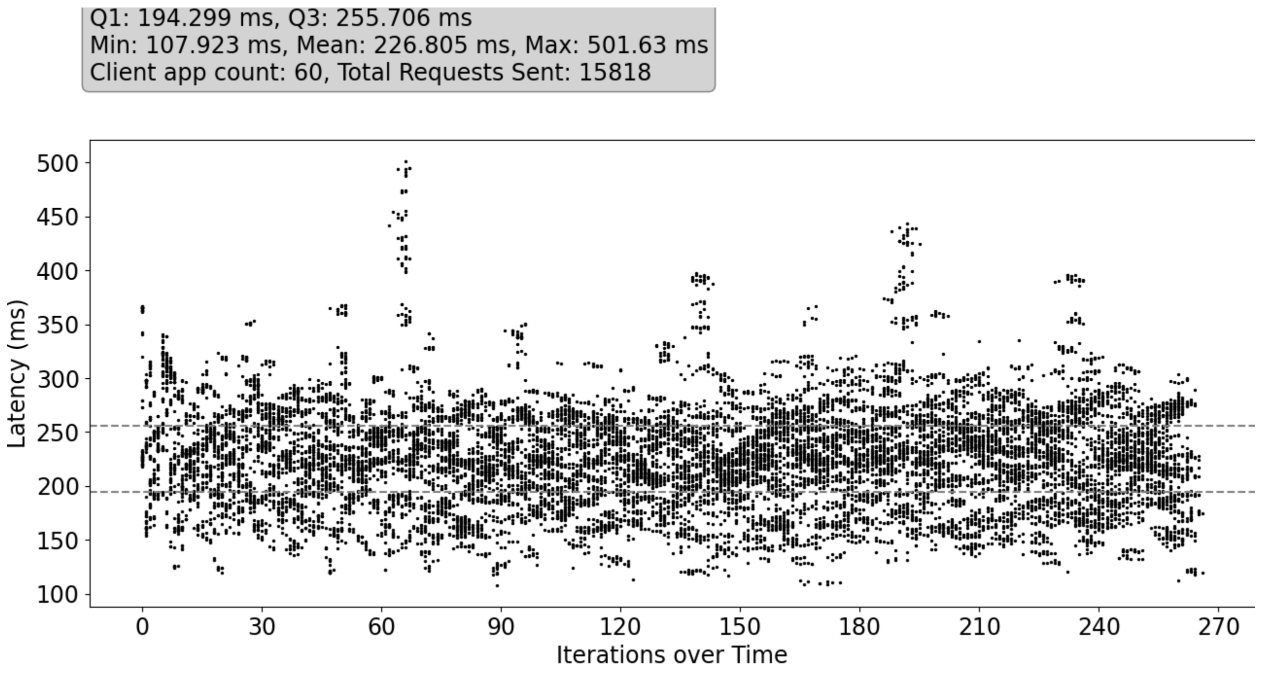


Figure 6-27 Blockchain-Enabled Access Control latency for 60 client apps

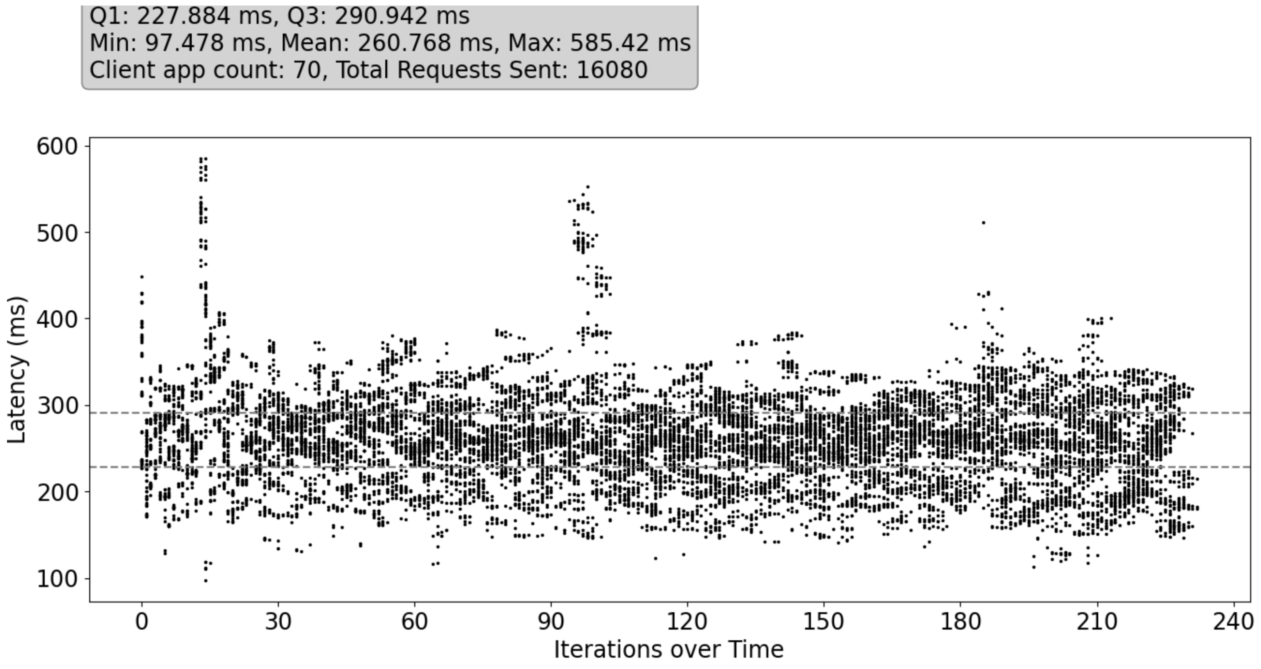


Figure 6-28 Blockchain-Enabled Access Control latency for 70 client apps

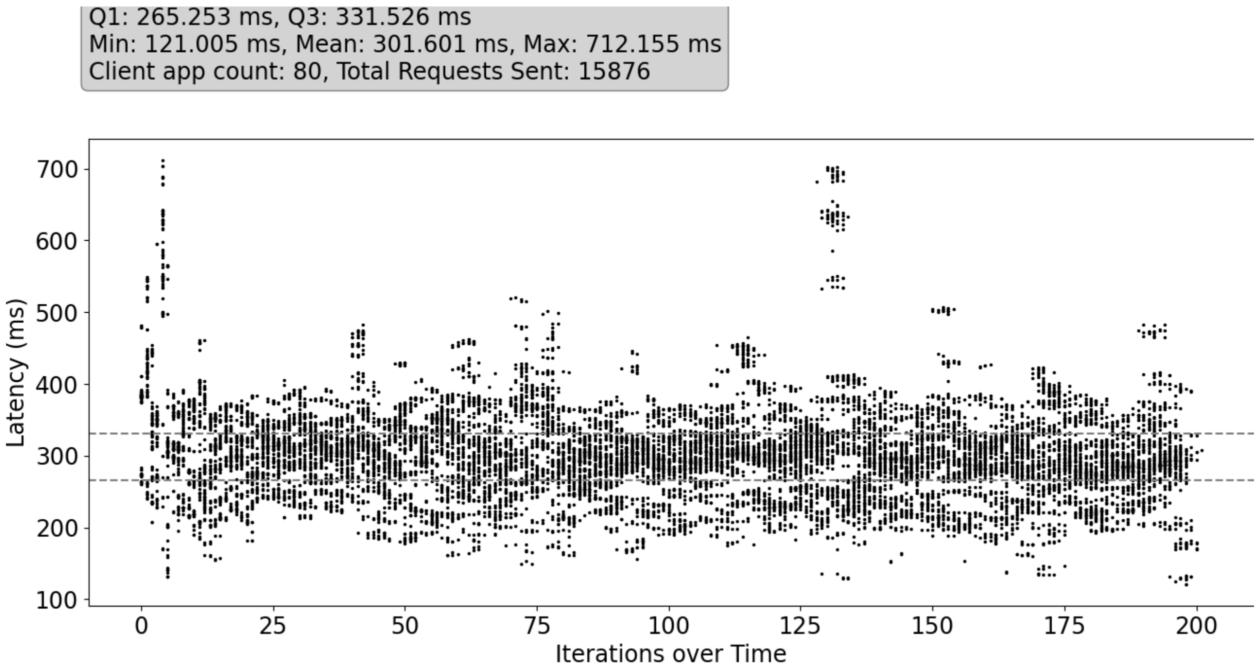


Figure 6-29 Blockchain-Enabled Access Control latency for 80 client apps

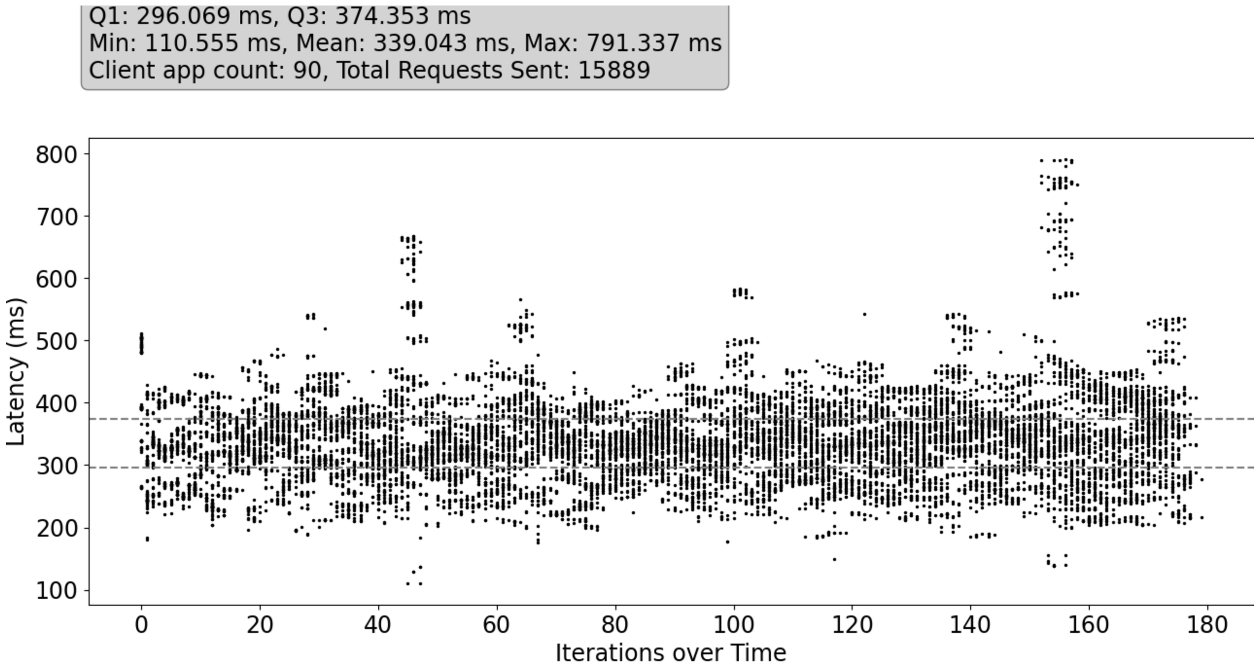


Figure 6-30 Blockchain-Enabled Access Control latency for 90 client apps

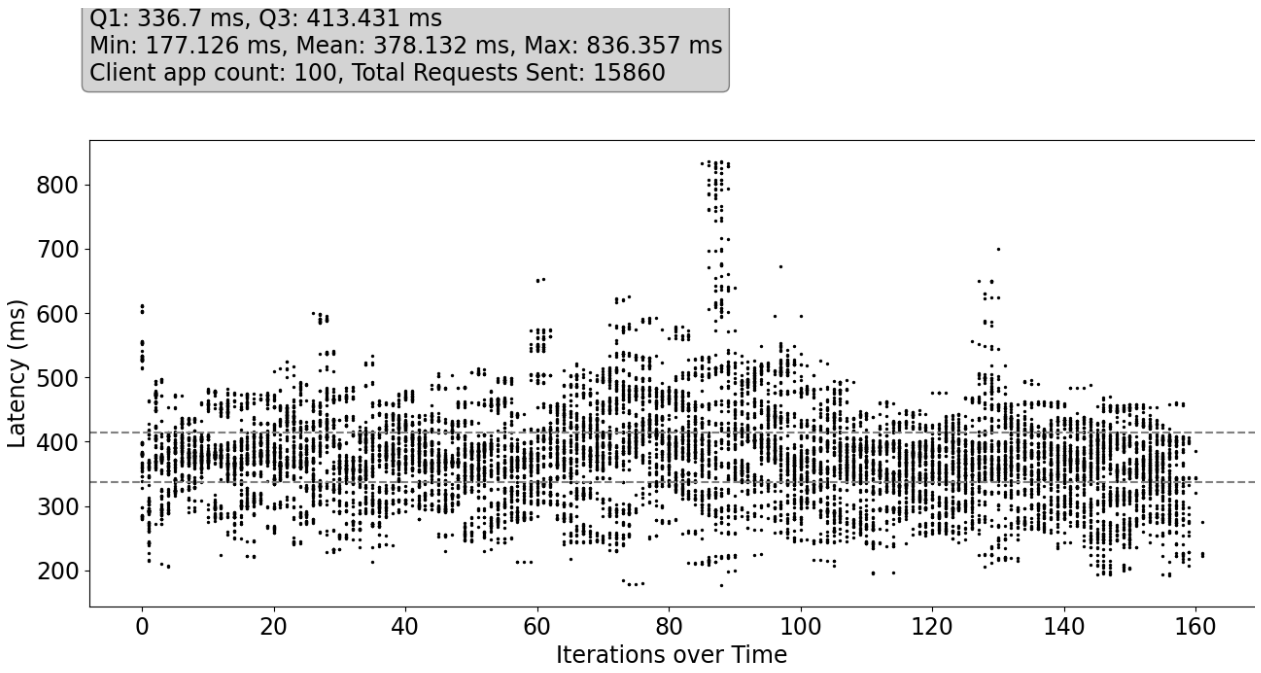


Figure 6-31 Blockchain-Enabled Access Control latency for 100 client apps

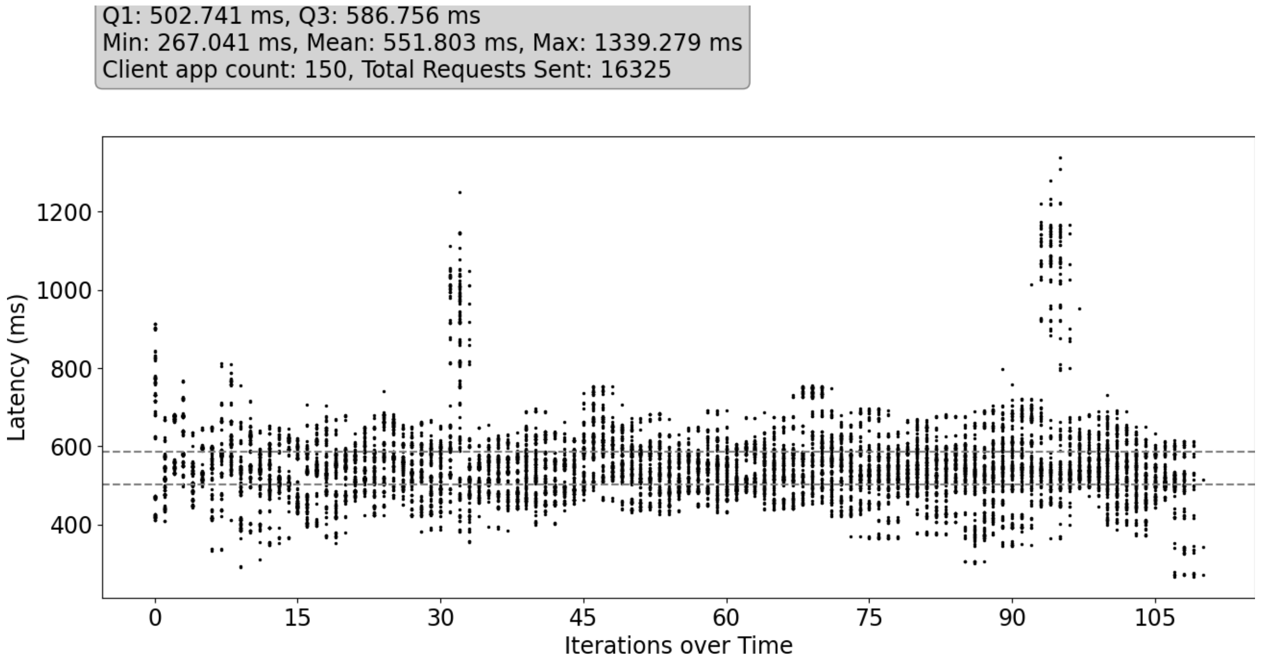


Figure 6-32 Blockchain-Enabled Access Control latency for 150 client apps

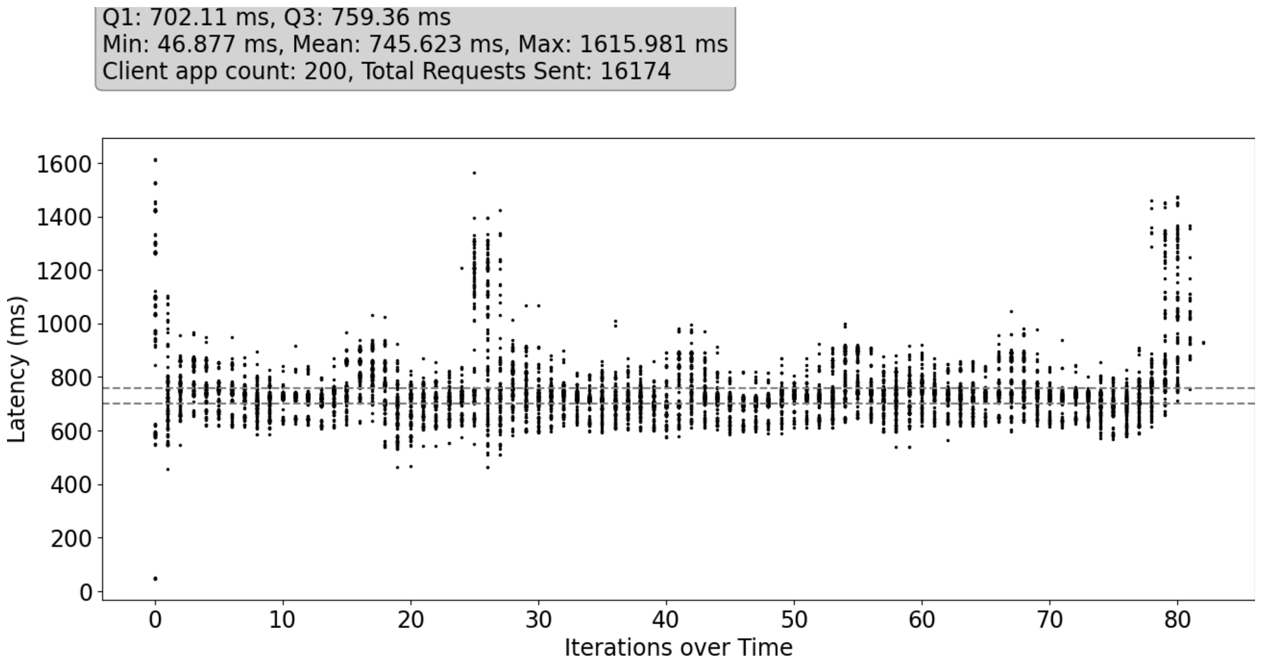


Figure 6-33 Blockchain-Enabled Access Control latency for 200 client apps

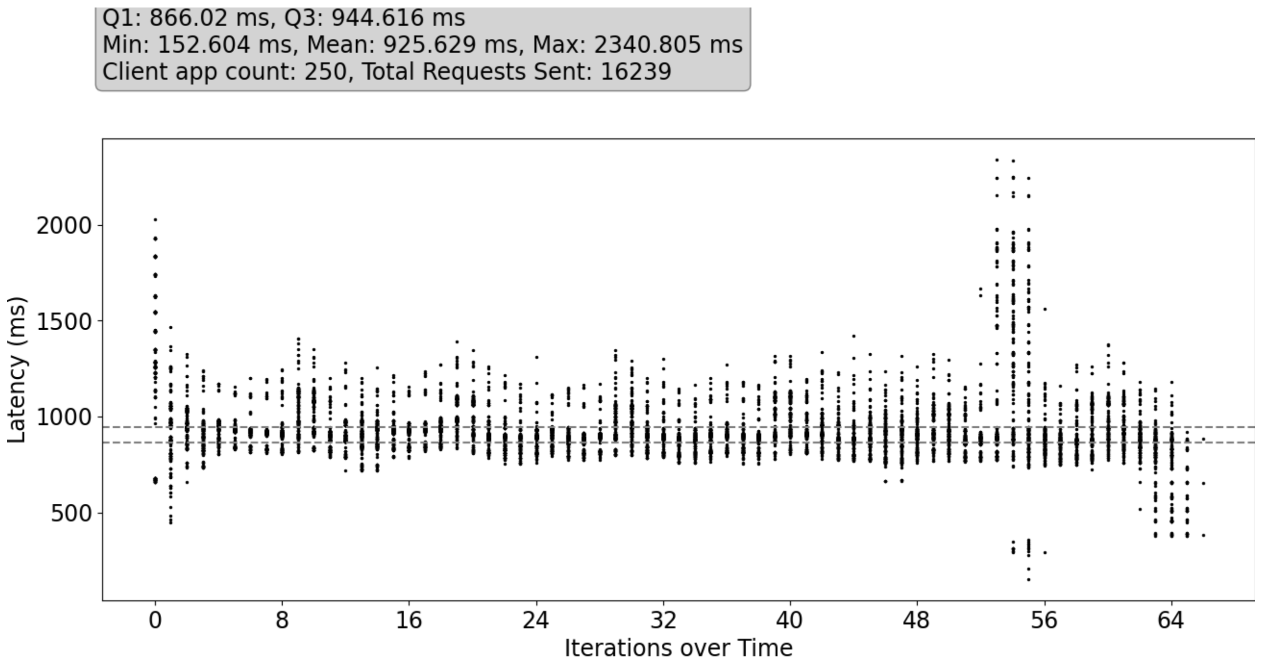


Figure 6-34 Blockchain-Enabled Access Control latency for 250 client apps

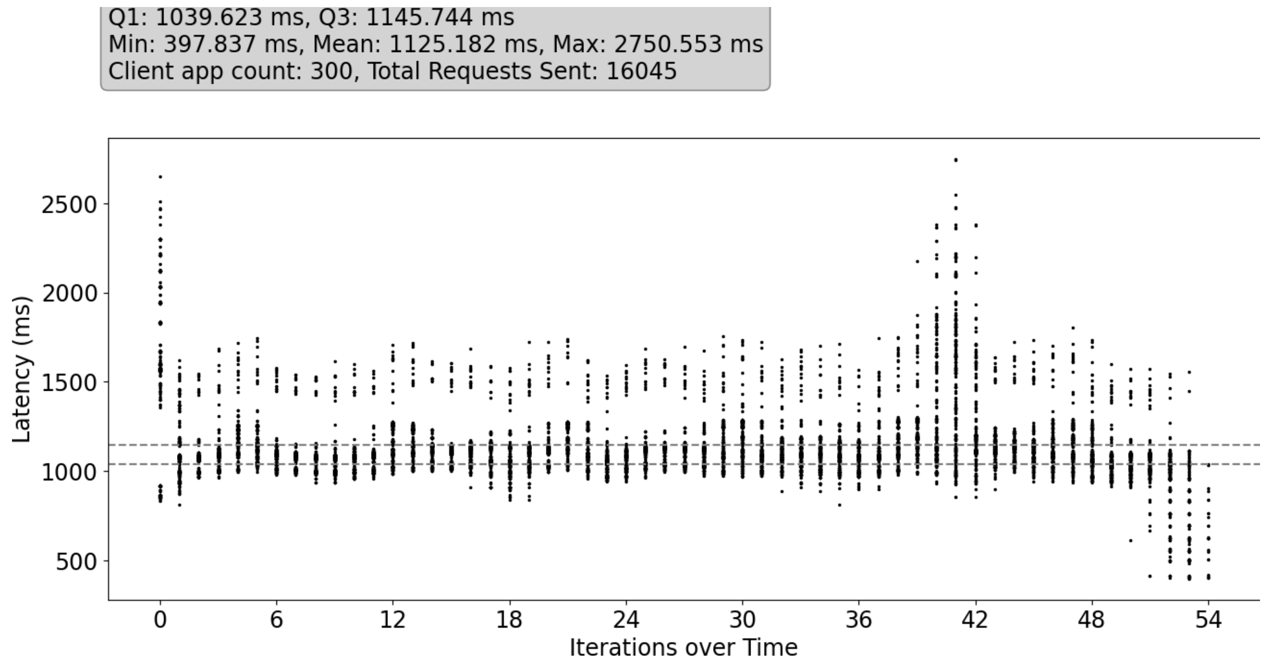


Figure 6-35 Blockchain-Enabled Access Control latency for 300 client apps

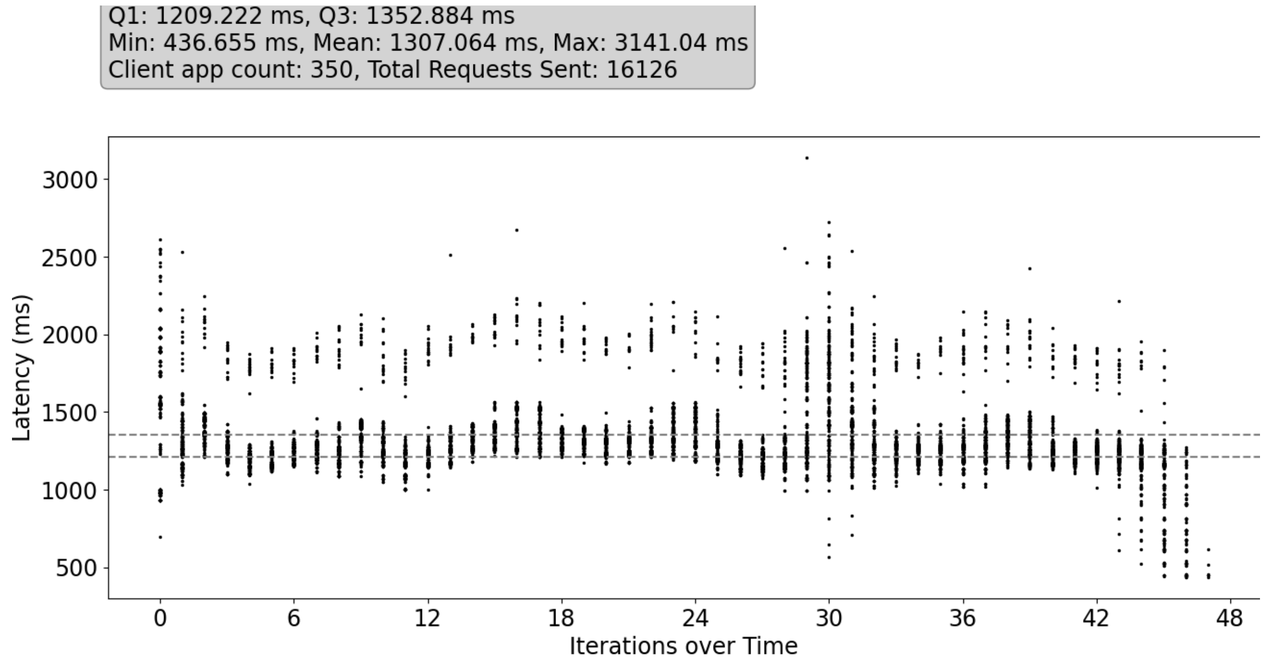


Figure 6-36 Blockchain-Enabled Access Control latency for 350 client apps

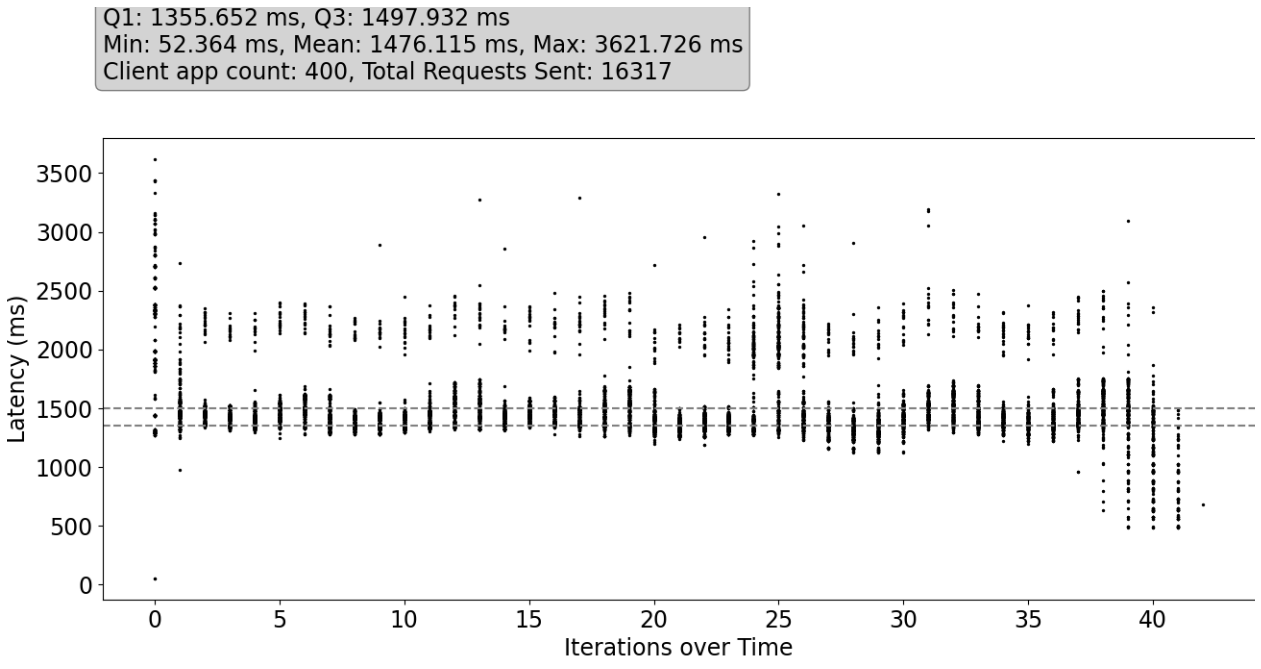


Figure 6-37 Blockchain-Enabled Access Control latency for 400 client apps

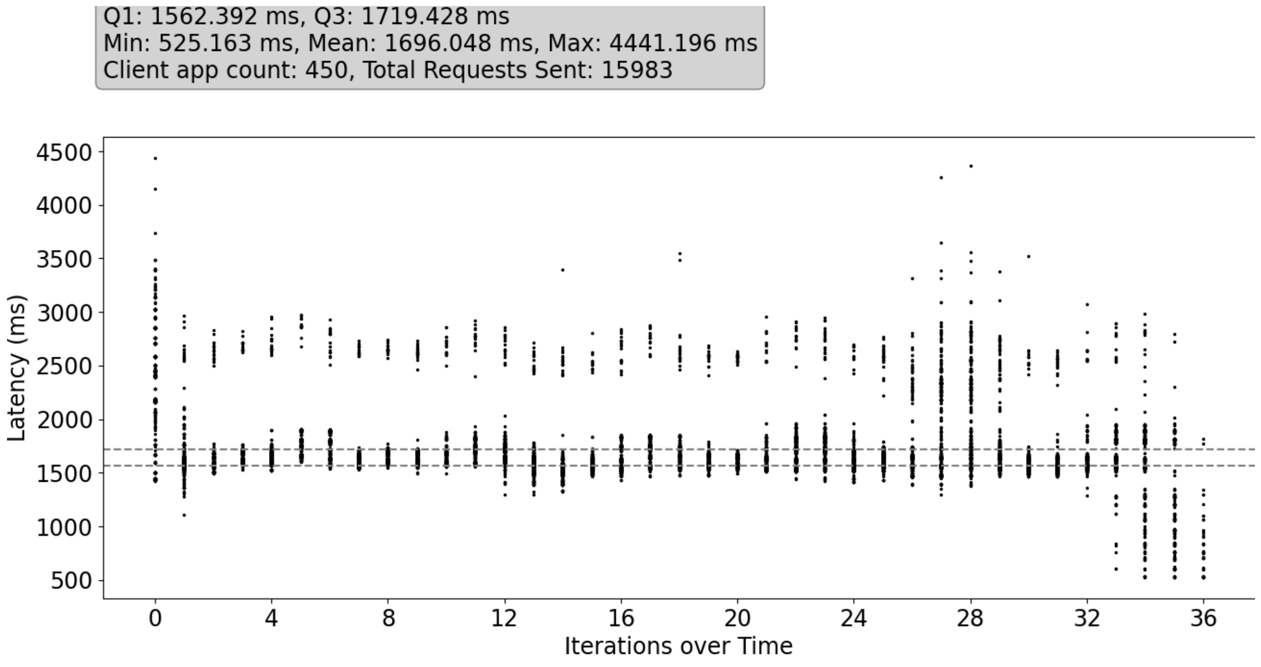


Figure 6-38 Blockchain-Enabled Access Control latency for 450 client apps

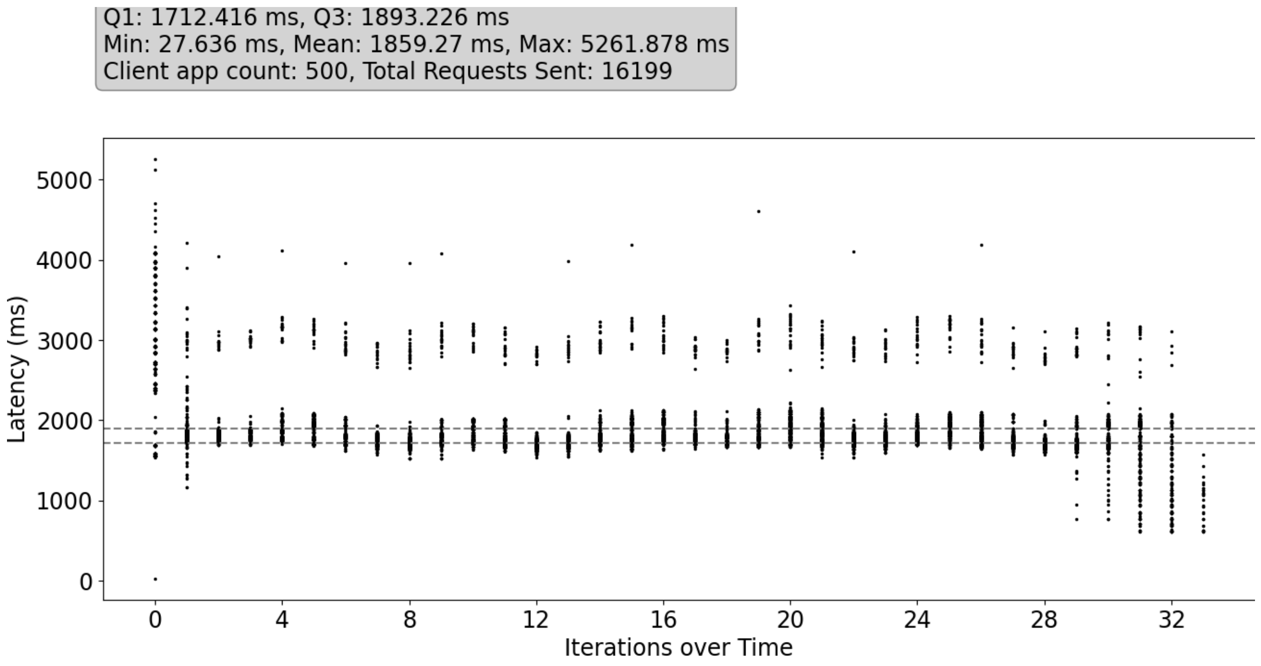


Figure 6-39 Blockchain-Enabled Access Control latency for 500 client apps

Using the Q1 and Q3 values from the Direct API Access and Blockchain-enabled access control scenarios, the following tables can be constructed:

Table 6-2 Q1 Latency comparison: Direct API Access vs Blockchain-enabled access control

Client app count	Q1 Latency (ms) of Direct API Access	Q1 Latency (ms) of Blockchain-enabled access control	Q1 Latency Difference (ms)
1	0.399	10.005	9.606
10	2.012	37.731	35.719
20	4.637	67.015	62.378
30	6.917	100.027	93.11
40	9.801	131.714	121.913
50	11.749	165.335	153.586
60	14.662	194.299	179.637
70	17.536	227.884	210.348
80	20.312	265.253	244.941

90	22.494	296.069	273.575
100	25.782	336.699	310.917
150	40.121	502.741	462.62
200	51.209	702.11	650.901
250	66.578	866.02	799.442
300	82.435	1039.623	957.188
350	92.19	1209.222	1117.032
400	106.609	1355.652	1249.043
450	122.308	1562.393	1440.085
500	135.271	1712.416	1577.145

Table 6-3 Q3 Latency comparison: Direct API Access vs Blockchain-enabled access control

Client app count	Q3 Latency (ms) of Direct API Access	Q3 Latency (ms) of Blockchain-enabled access control	Q3 Latency Difference (ms)
1	0.449	11.754	11.305
10	2.666	49.923	47.257
20	5.364	86.886	81.522
30	7.941	129.855	121.914
40	11.055	169.848	158.793
50	13.469	217.053	203.584
60	16.895	255.706	238.811
70	20.314	290.942	270.628
80	23.55	331.526	307.976
90	26.093	374.353	348.26
100	29.671	413.432	383.761

150	45.265	586.756	541.491
200	57.061	759.36	702.299
250	74.069	944.615	870.546
300	89.825	1145.744	1055.919
350	102.689	1352.884	1250.195
400	116.65	1497.932	1381.282
450	134.486	1719.428	1584.942
500	146.399	1893.226	1746.827

The tables present latency statistics for two scenarios: direct API access and protected API access (Blockchain-enabled access control). It compares the Q1 and Q3 latency values for these scenarios. A line chart can be plotted from the Q1 latency difference and Q3 latency difference from the two tables to visualize the trend:

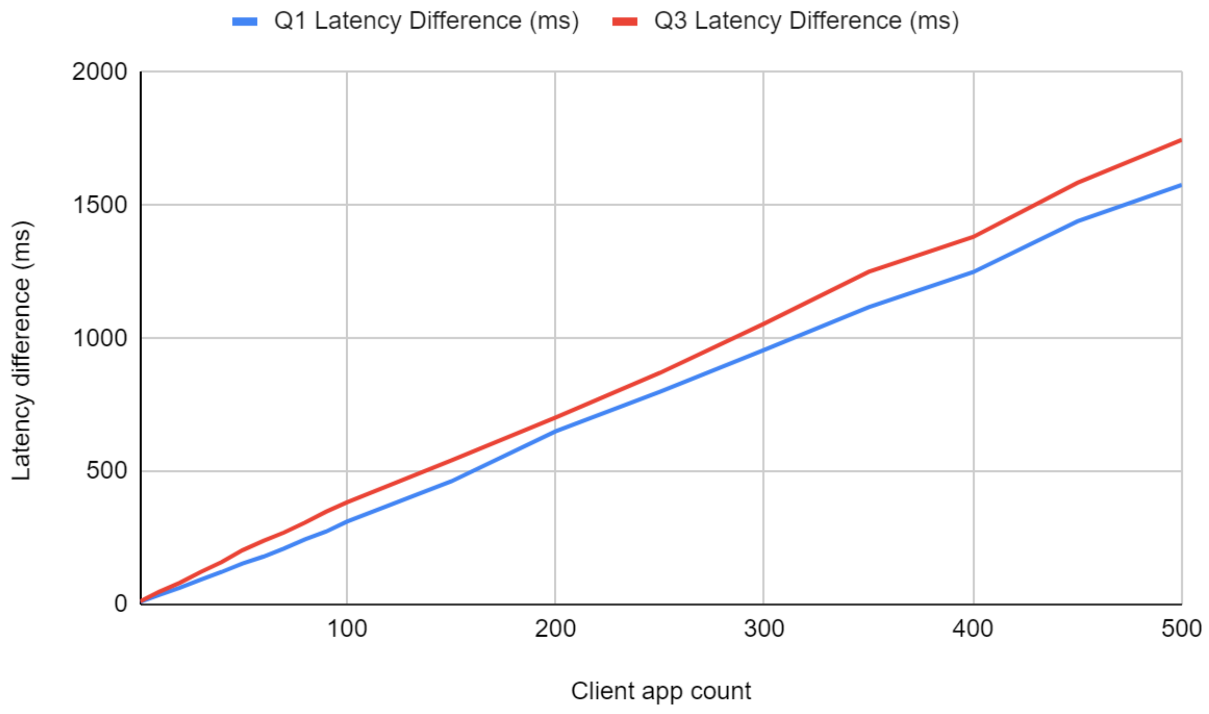


Figure 6-40 Latency comparison: Direct API Access vs Protected API Access

The two tables and the chart provide an answer to *research question 1*. It can be anticipated that when transitioning from openly accessible APIs to having them controlled by the proposed blockchain-based solution, the more client apps accessing their APIs at once, the more significant the latency becomes. When 500 client apps hit the system simultaneously, compared to the openly accessible APIs scenario, the blockchain-based solution added an overhead latency of around 1577.145 to 1746.827 ms. As the number of client apps increased, the overhead latency increased linearly.

6.2.3 Overhead latency of Hyperledger Fabric

With the statistics of how the Direct API access scenario and how the protected API access scenario (Blockchain-enabled access control scenario) handled the load available, Sentry on the Blockchain VM was then configured to bypass Hyperledger Fabric and forward the requests to the API Endpoint VM without running any access control logic (Blockchain-bypassed access control scenario). Effectively, Sentry was just a reverse proxy in this case. Then, K6, running on the load-testing VM, started loading the system. After it had finished loading the system with multiple sets of virtual client apps, the following scatter charts were plotted.

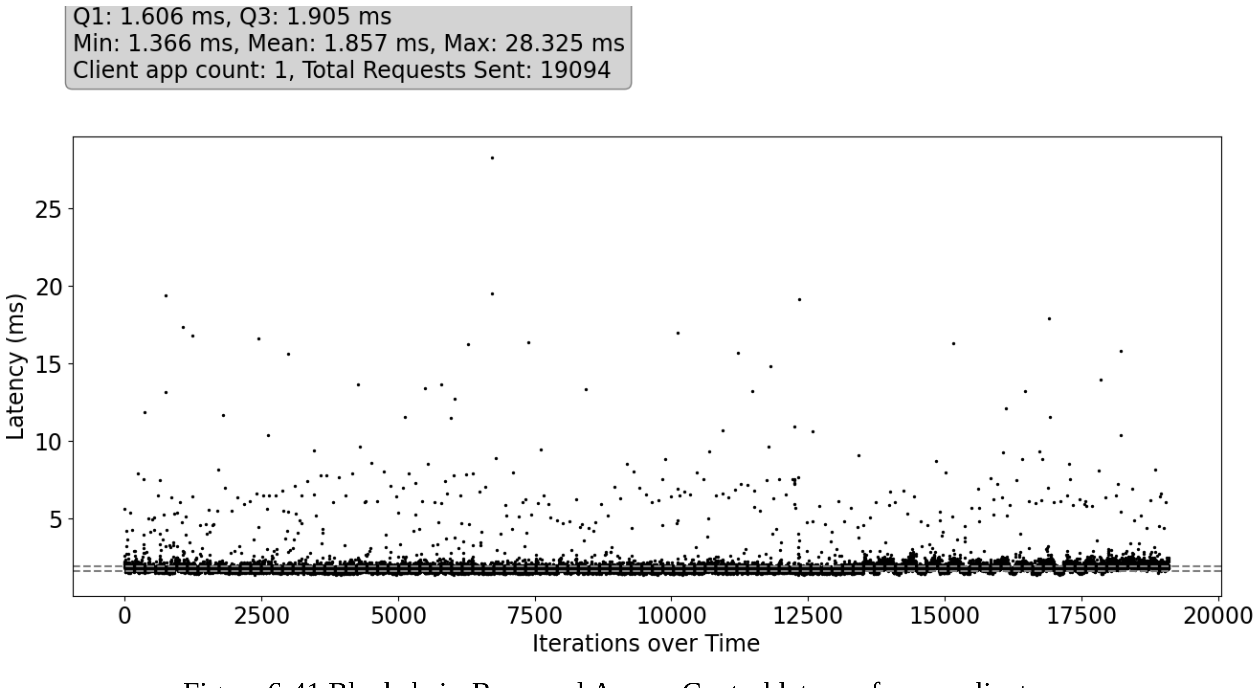


Figure 6-41 Blockchain-Bypassed Access Control latency for one client app

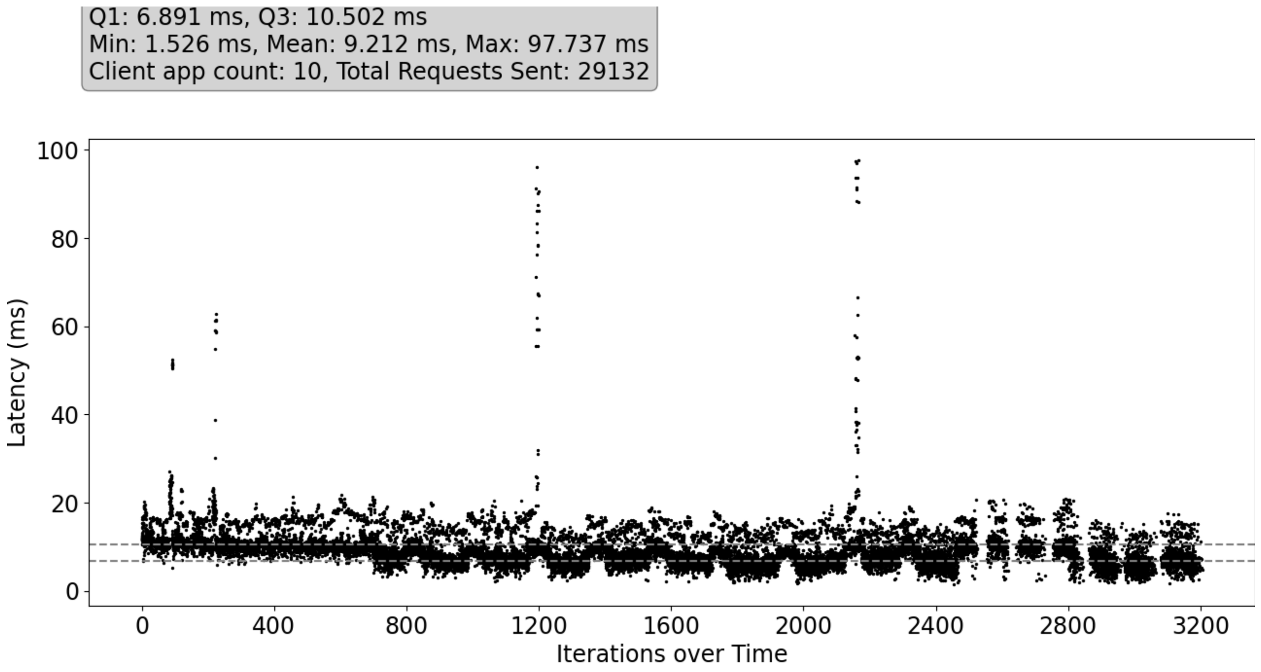


Figure 6-42 Blockchain-Bypassed Access Control latency for ten client apps

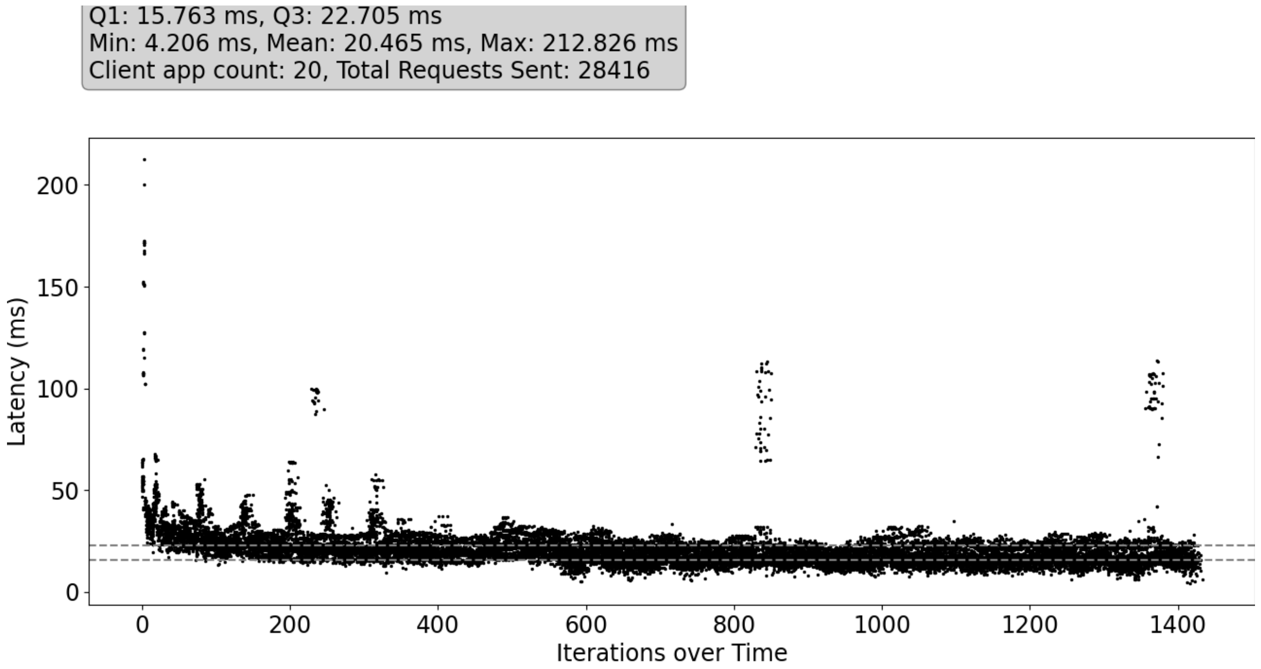


Figure 6-43 Blockchain-Bypassed Access Control latency for 20 client apps

Q1: 23.68 ms, Q3: 30.265 ms
Min: 3.373 ms, Mean: 28.057 ms, Max: 145.739 ms
Client app count: 30, Total Requests Sent: 28392

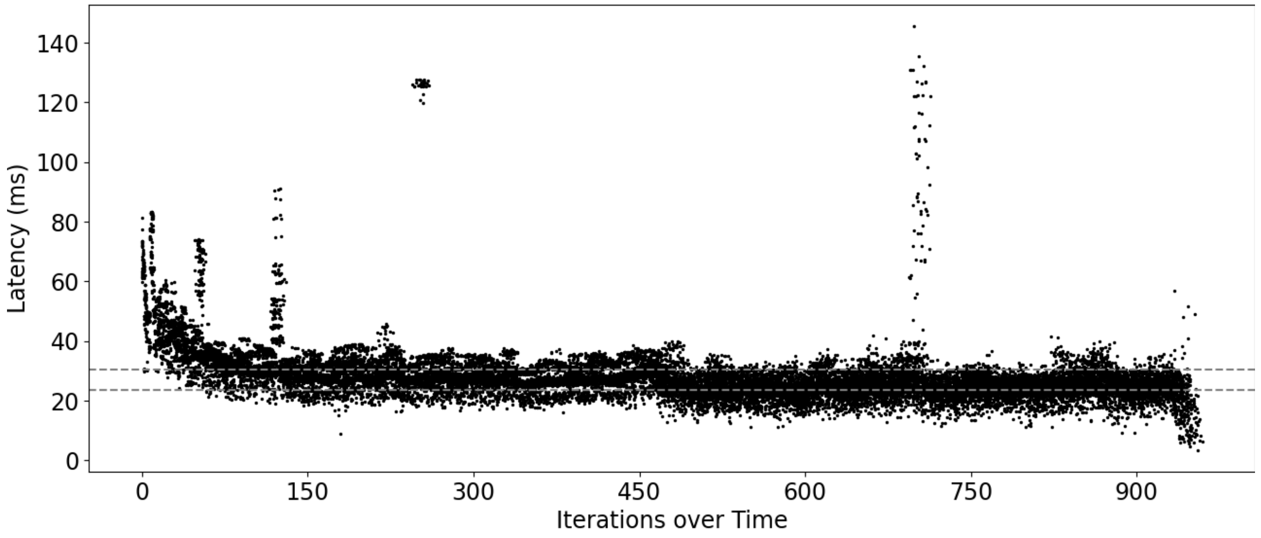


Figure 6-44 Blockchain-Bypassed Access Control latency for 30 client apps

Q1: 33.242 ms, Q3: 41.466 ms
Min: 2.018 ms, Mean: 38.286 ms, Max: 220.713 ms
Client app count: 40, Total Requests Sent: 28512

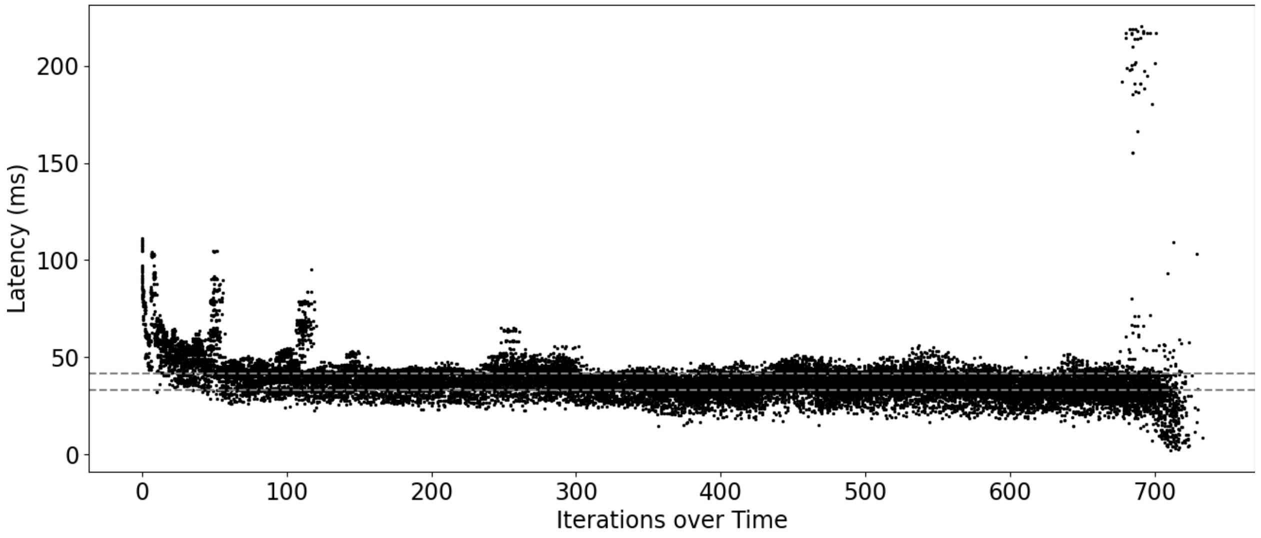


Figure 6-45 Blockchain-Bypassed Access Control latency for 40 client apps

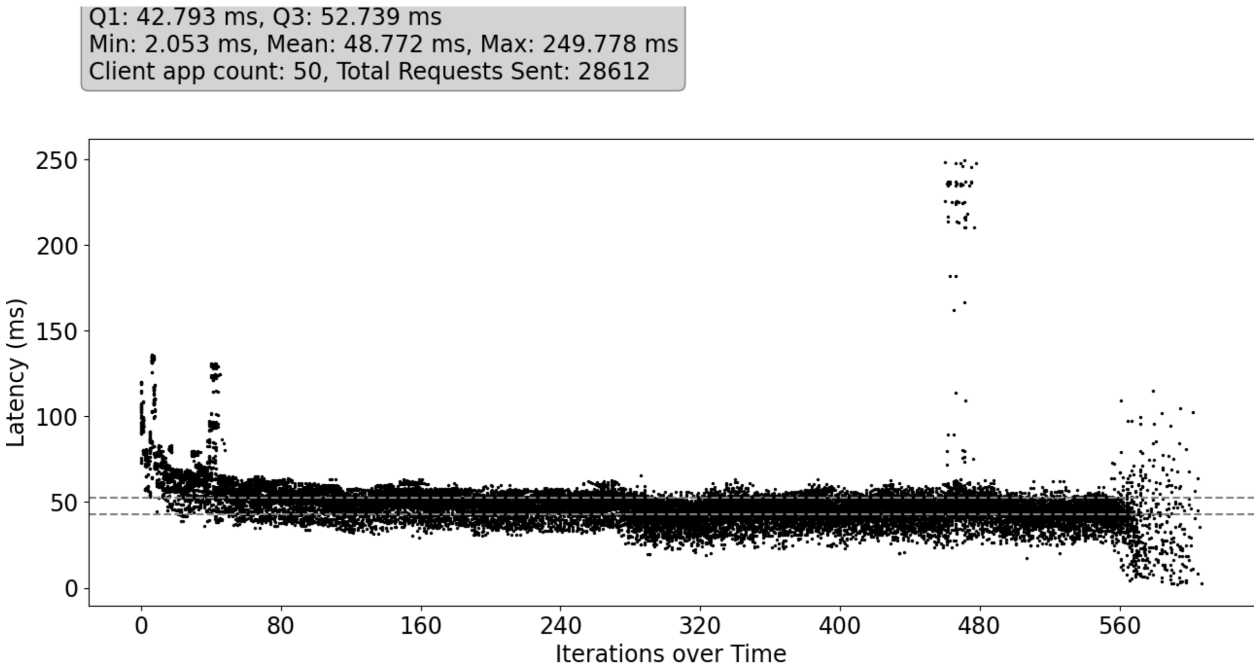


Figure 6-46 Blockchain-Bypassed Access Control latency for 50 client apps

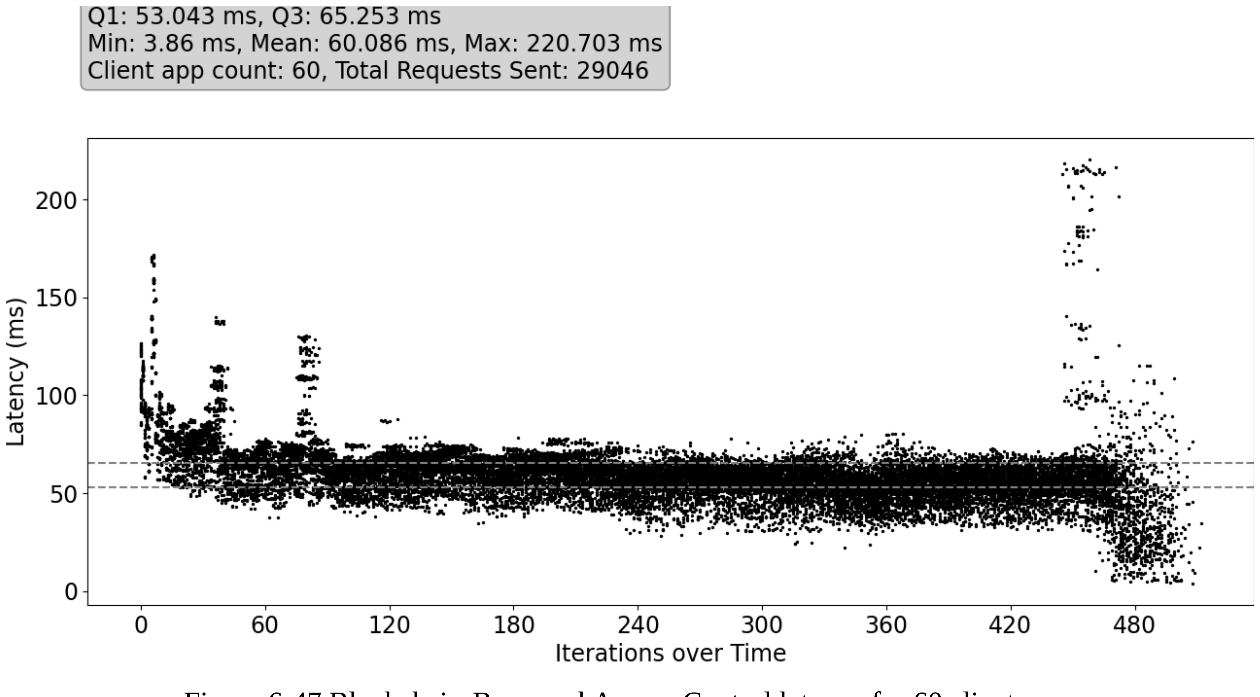


Figure 6-47 Blockchain-Bypassed Access Control latency for 60 client apps

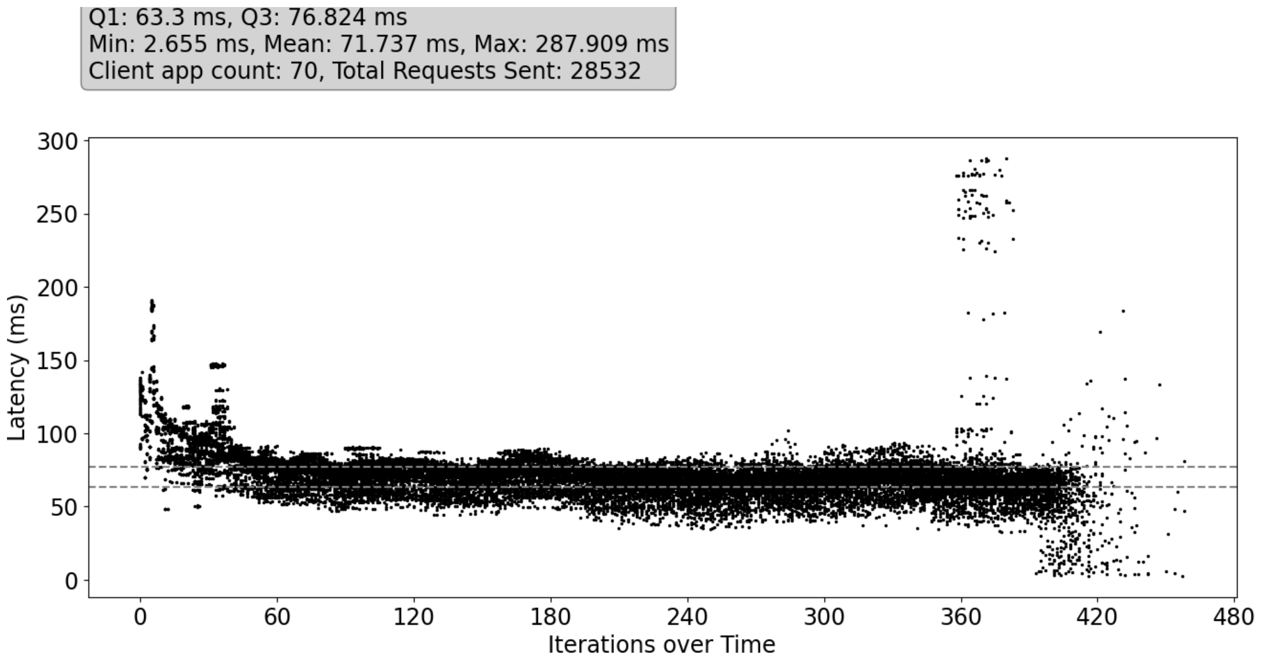


Figure 6-48 Blockchain-Bypassed Access Control latency for 70 client apps

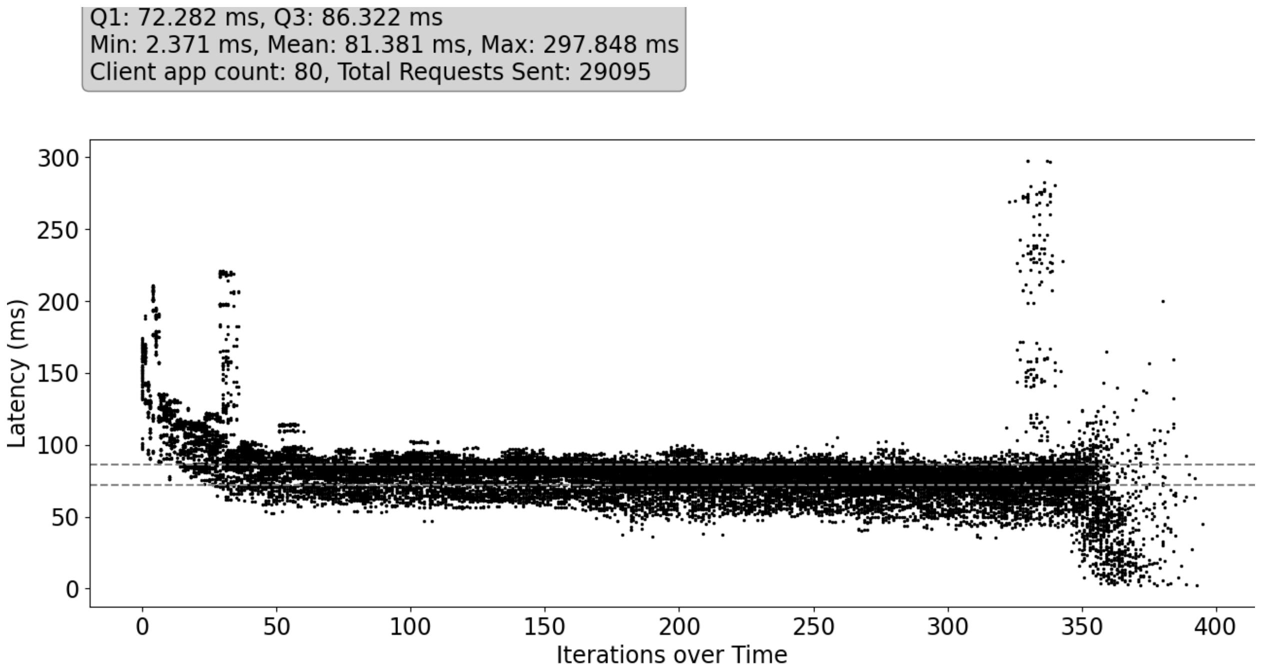


Figure 6-49 Blockchain-Bypassed Access Control latency for 80 client apps

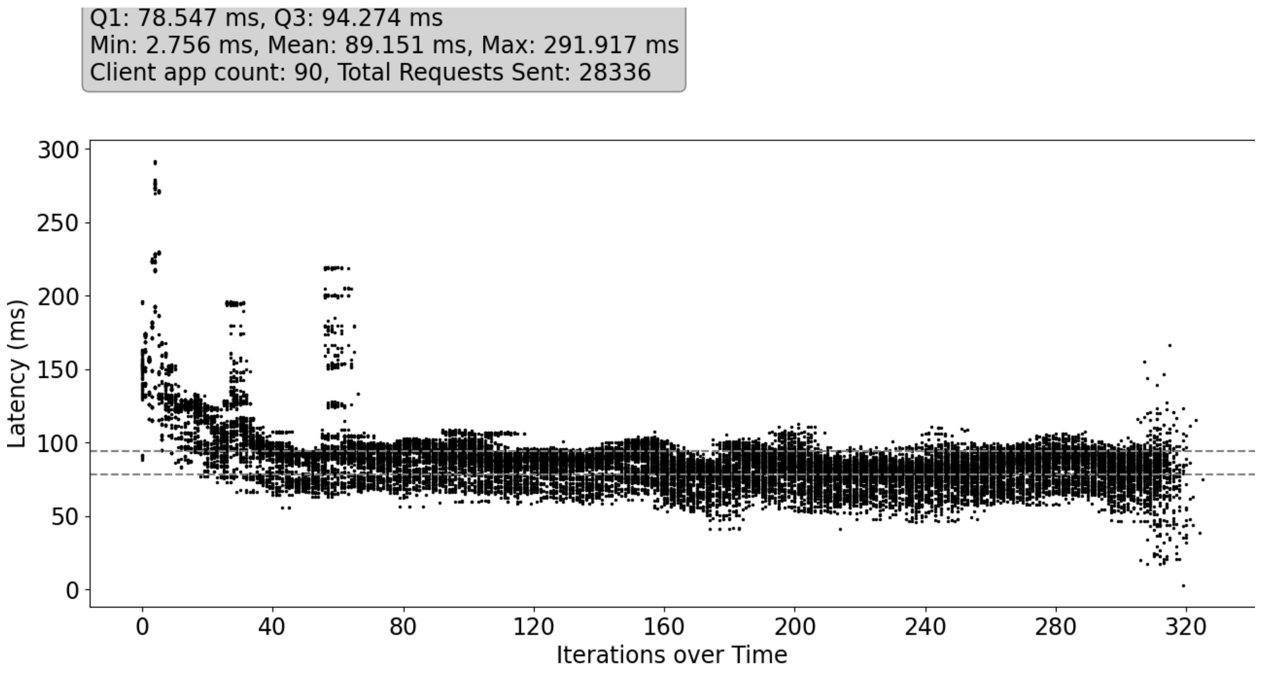


Figure 6-50 Blockchain-Bypassed Access Control latency for 90 client apps

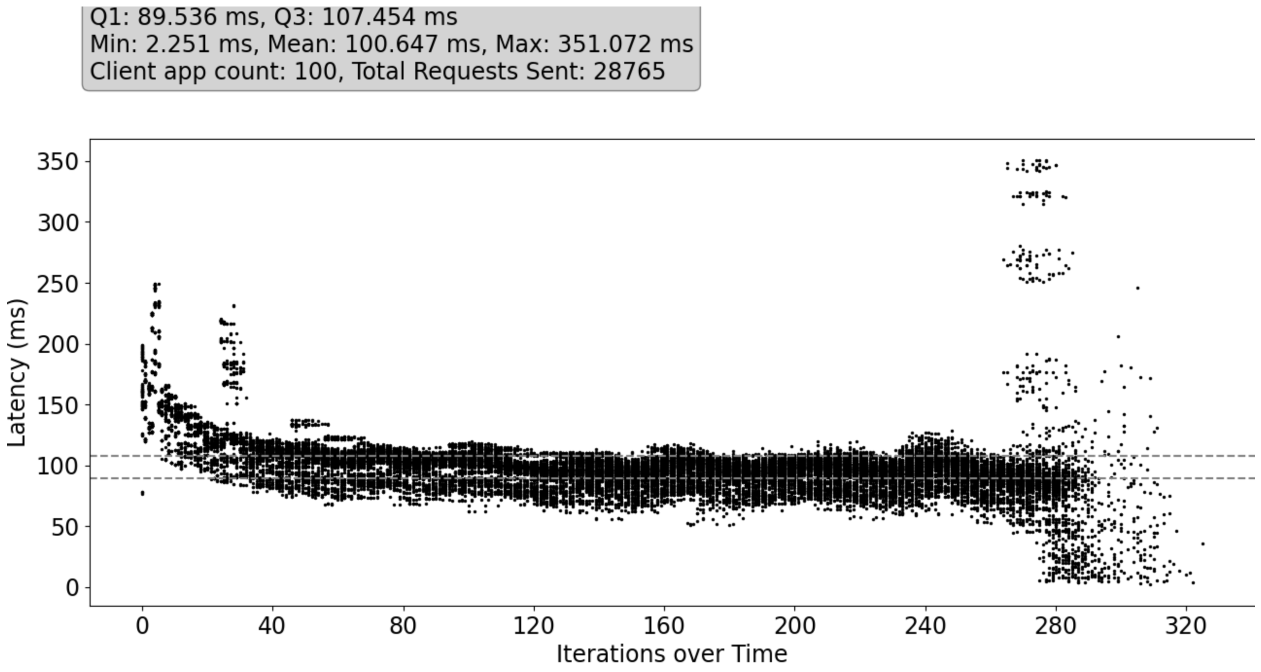


Figure 6-51 Blockchain-Bypassed Access Control latency for 100 client apps

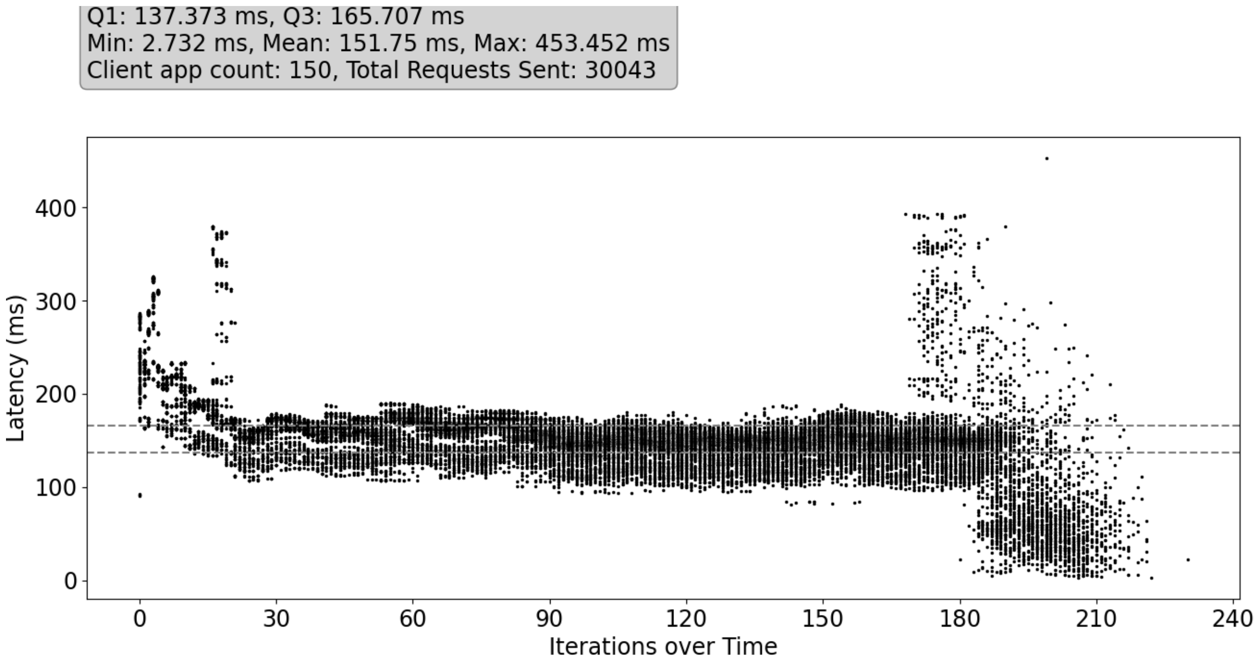


Figure 6-52 Blockchain-Bypassed Access Control latency for 150 client apps

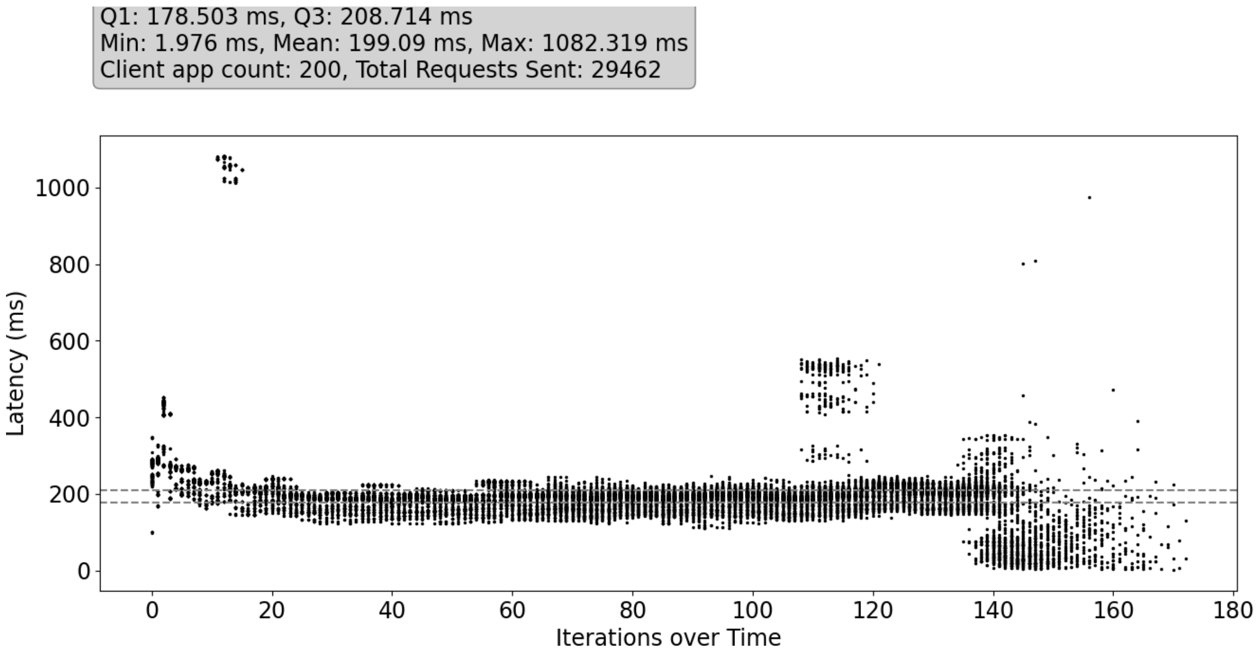


Figure 6-53 Blockchain-Bypassed Access Control latency for 200 client apps

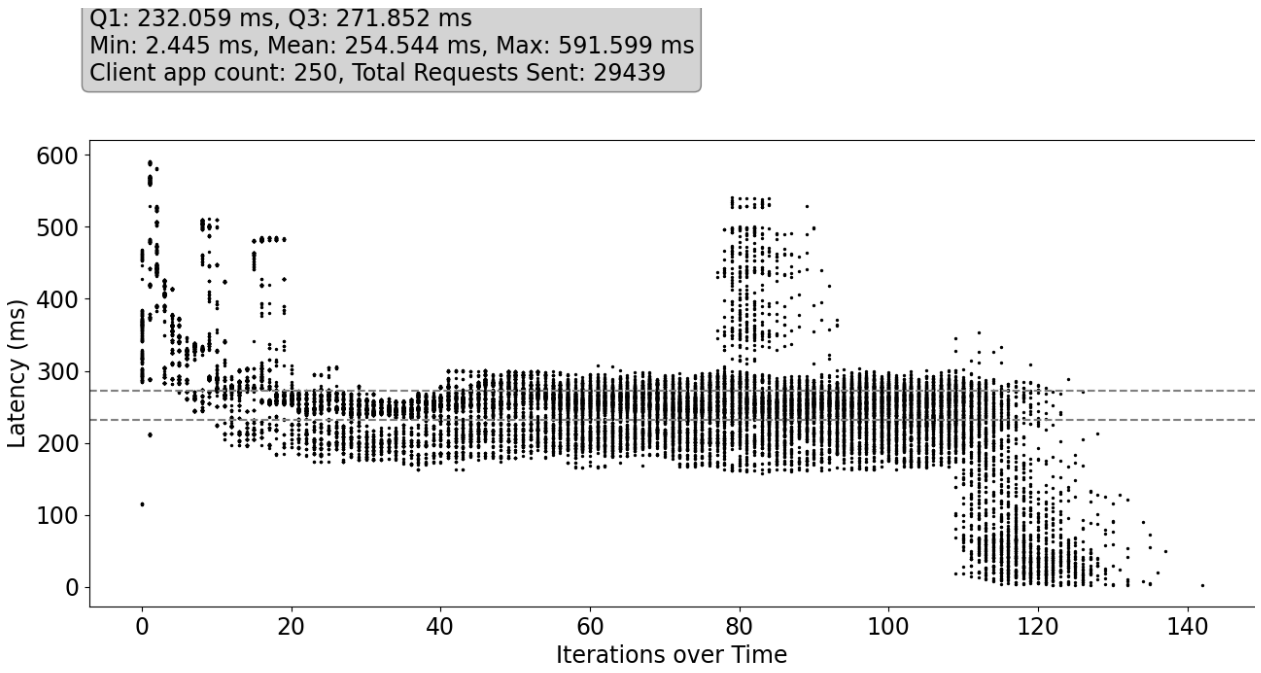


Figure 6-54 Blockchain-Bypassed Access Control latency for 250 client apps

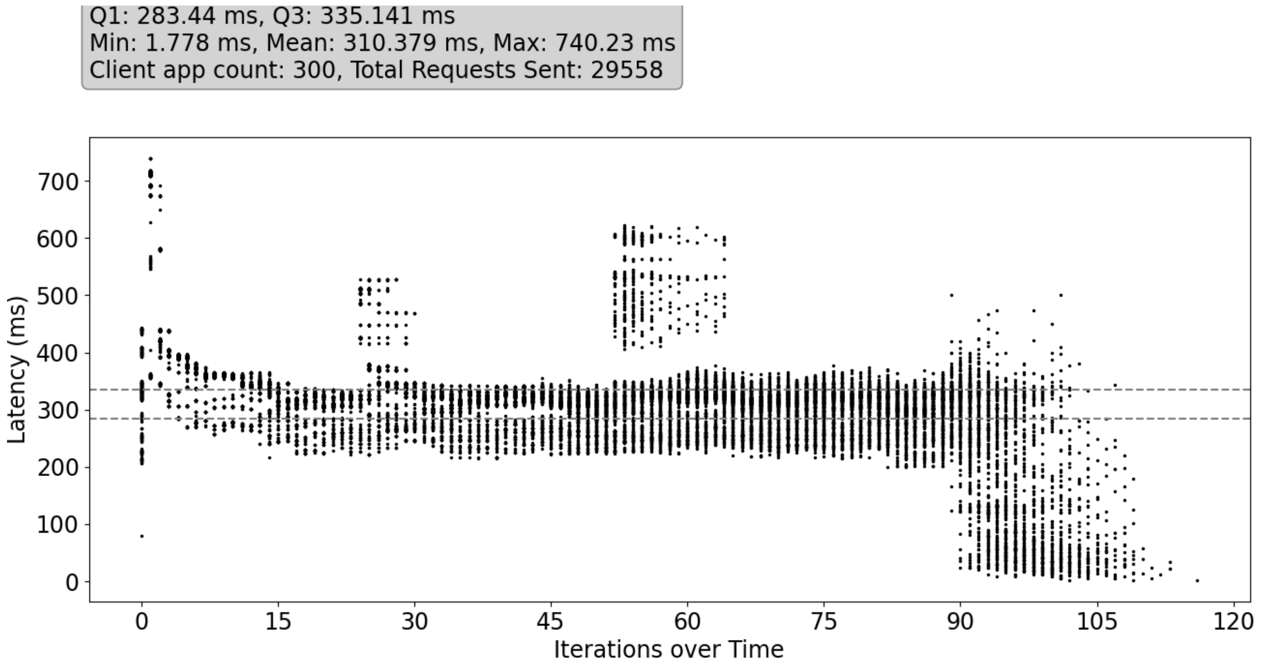


Figure 6-55 Blockchain-Bypassed Access Control latency for 300 client apps

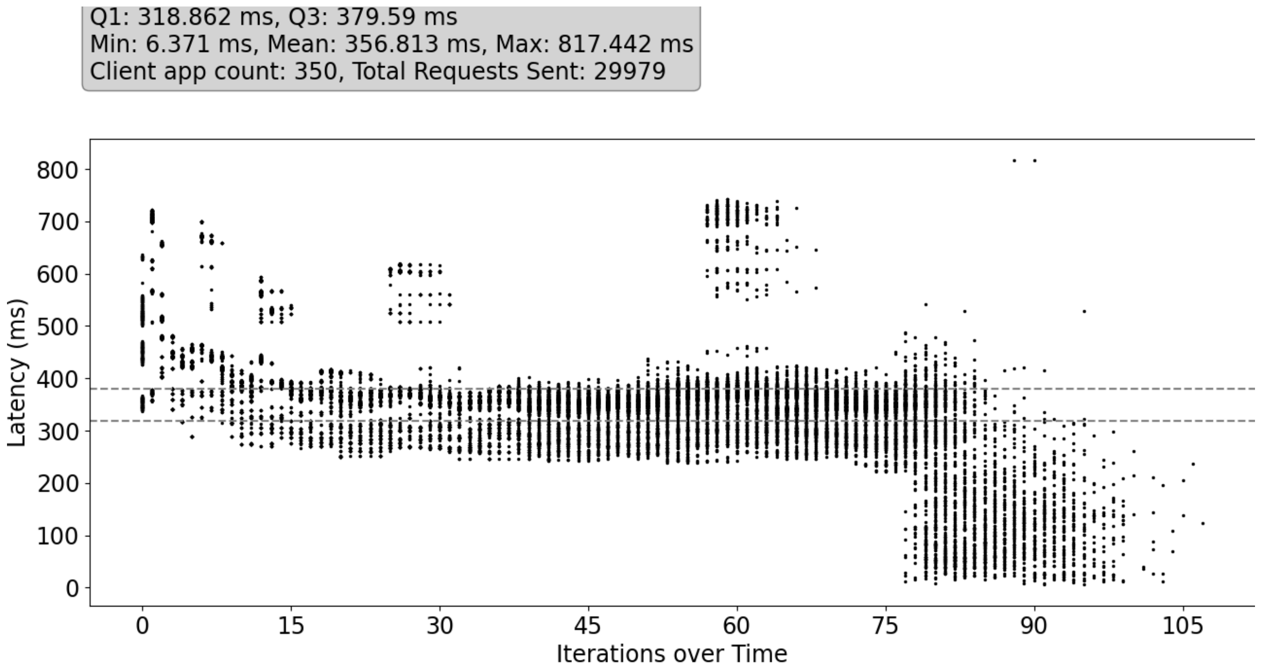


Figure 6-56 Blockchain-Bypassed Access Control latency for 350 client apps

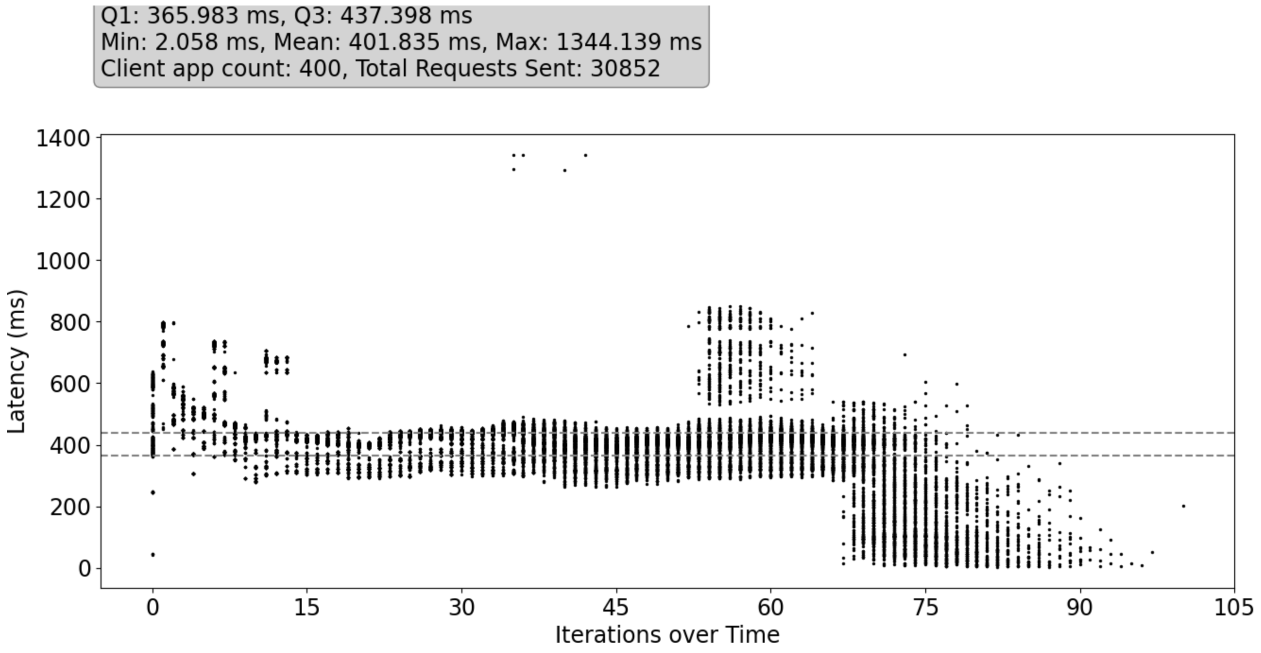


Figure 6-57 Blockchain-Bypassed Access Control latency for 400 client apps

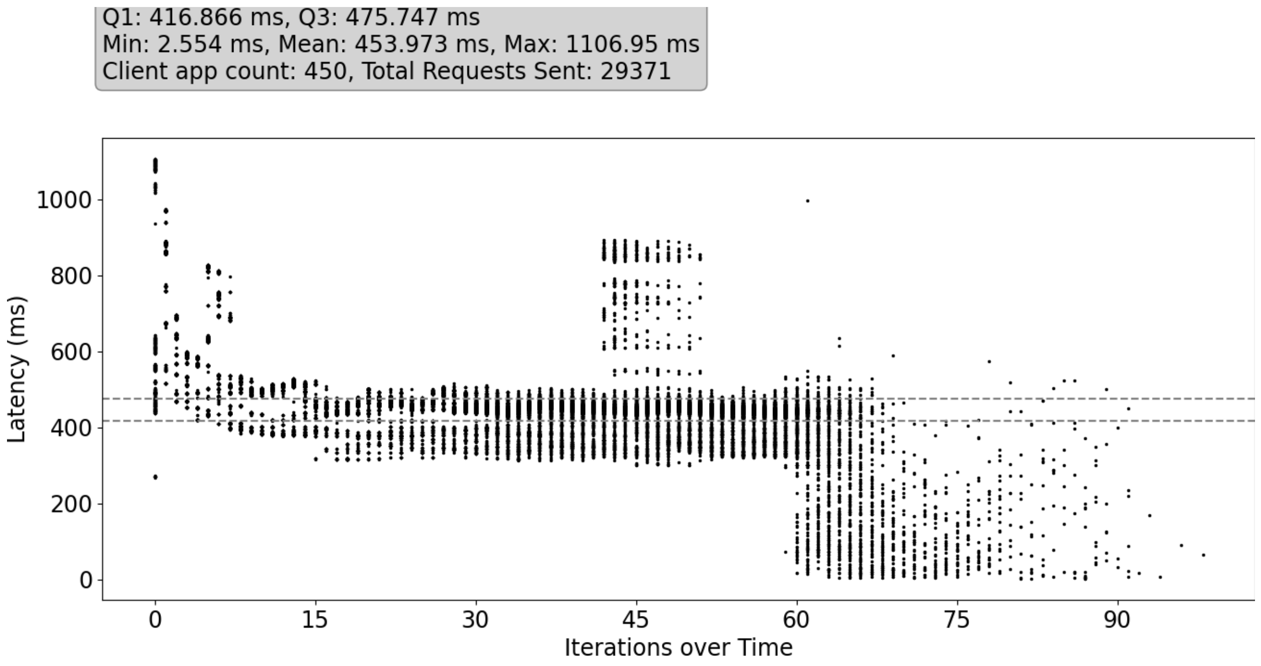


Figure 6-58 Blockchain-Bypassed Access Control latency for 450 client apps

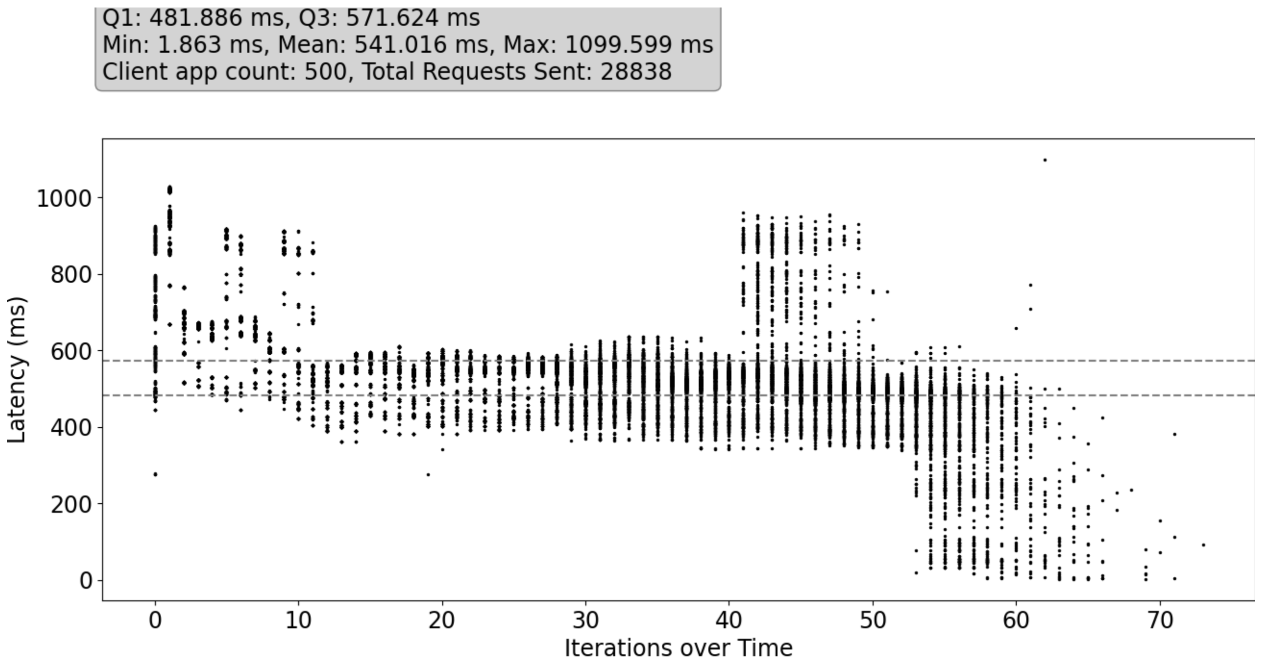


Figure 6-59 Blockchain-Bypassed Access Control latency for 500 client apps

Using the Q1 and Q3 values from the Blockchain-enabled access control and Blockchain-bypassed access control scenarios, the following two tables can be constructed:

Table 6-4 Q1 Latency Comparison: Hyperledger Fabric bypassed vs. Hyperledger Fabric enabled

Client app count	Q1 Latency (ms) of Blockchain-bypassed access control	Q1 Latency (ms) of Blockchain-enabled access control	Q1 Latency Difference (ms)
1	1.606	10.005	8.399
10	6.891	37.731	30.84
20	15.763	67.015	51.252
30	23.679	100.027	76.348
40	33.242	131.714	98.472
50	42.793	165.335	122.542
60	53.043	194.299	141.256
70	63.3	227.884	164.584
80	72.282	265.253	192.971
90	78.548	296.069	217.521
100	89.536	336.699	247.163
150	137.373	502.741	365.368
200	178.503	702.11	523.607
250	232.059	866.02	633.961
300	283.44	1039.623	756.183
350	318.861	1209.222	890.361
400	365.983	1355.652	989.669
450	416.866	1562.393	1145.527
500	481.886	1712.416	1230.53

Table 6-5 Q3 Latency Comparison: Hyperledger Fabric bypassed vs. Hyperledger Fabric enabled

Client app count	Q3 Latency (ms) of Blockchain-bypassed access control	Q3 Latency (ms) of Blockchain-enabled access control	Q3 Latency Difference (ms)
1	1.905	11.754	9.849
10	10.502	49.923	39.421
20	22.705	86.886	64.181
30	30.266	129.855	99.589
40	41.466	169.848	128.382
50	52.739	217.053	164.314
60	65.253	255.706	190.453
70	76.824	290.942	214.118
80	86.322	331.526	245.204
90	94.274	374.353	280.079
100	107.454	413.432	305.978
150	165.707	586.756	421.049
200	208.714	759.36	550.646
250	271.853	944.615	672.762
300	335.141	1145.744	810.603
350	379.591	1352.884	973.293
400	437.398	1497.932	1060.534
450	475.747	1719.428	1243.681
500	571.625	1893.226	1321.601

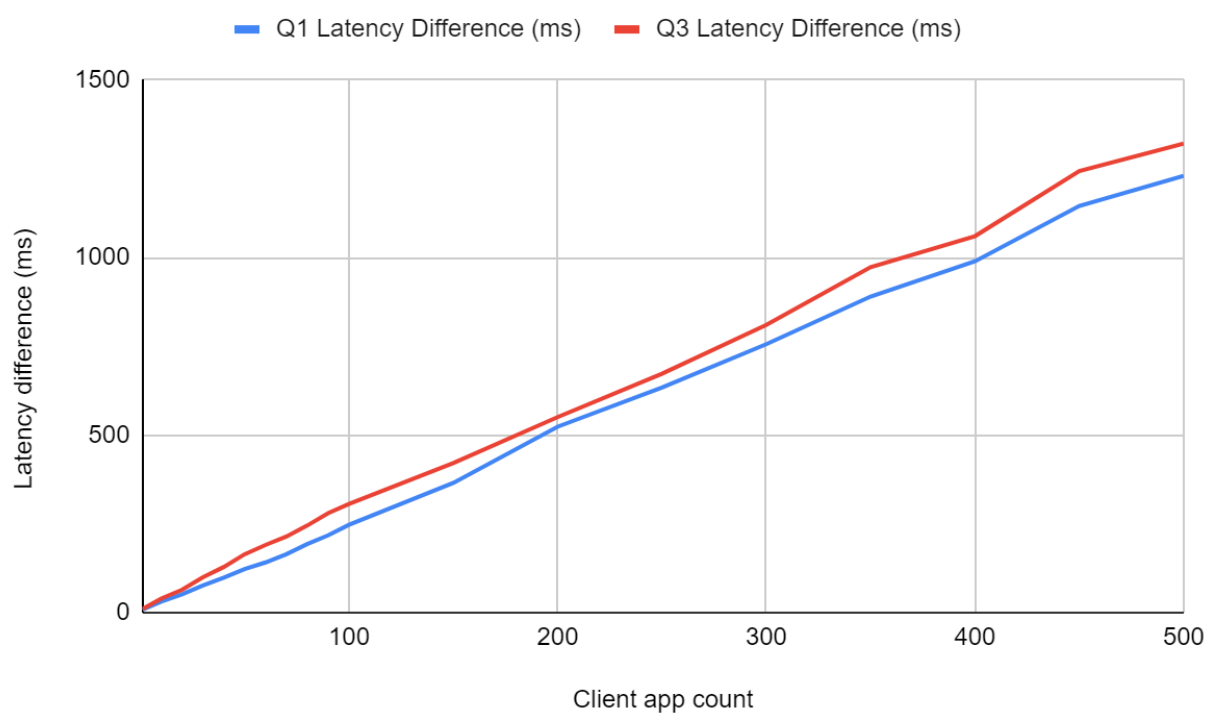


Figure 6-60 Latency comparison: Hyperledger Fabric bypassed vs enabled.

The two tables and the chart provide an answer to *research question 2*. It can be anticipated that when using Hyperledger Fabric, the more client apps accessing their APIs at once, the more significant the latency becomes. When 500 client apps hit the system simultaneously, Hyperledger Fabric added an overhead latency of around 1230.53 to 1321.601ms. As the number of client apps increased, the overhead latency increased linearly.

6.2.4 Experiment 2: Breaking point

This experiment aims to determine the system's breaking point, which is the point at which the error rate approaches 100%, indicating that none of the requests from the client apps receive a successful response (*research question 3*). A set of virtual client apps was utilized to generate load on the system to achieve this.

The process proceeded as follows: with the three VMs in the private cloud environment, the Blockchain VM was configured with one peer node in the Hyperledger network. Subsequently, once access to the protected API endpoints had been granted, K6 running on the load-testing VM initiated with a set of 500 virtual client apps. These virtual client apps sent requests to an API endpoint for 60 seconds, generating a significant traffic load on Sentry and the Hyperledger Fabric network. After 60 seconds, K6 reported the error rate, recorded in a CSV file. K6 then increased the number of virtual client apps to 1,000 and continued this process until it reached 30,000. After completing the last set of client apps, a bar chart was generated from the CSV file.

The entire process was repeated three times, resulting in three error rate bar charts.

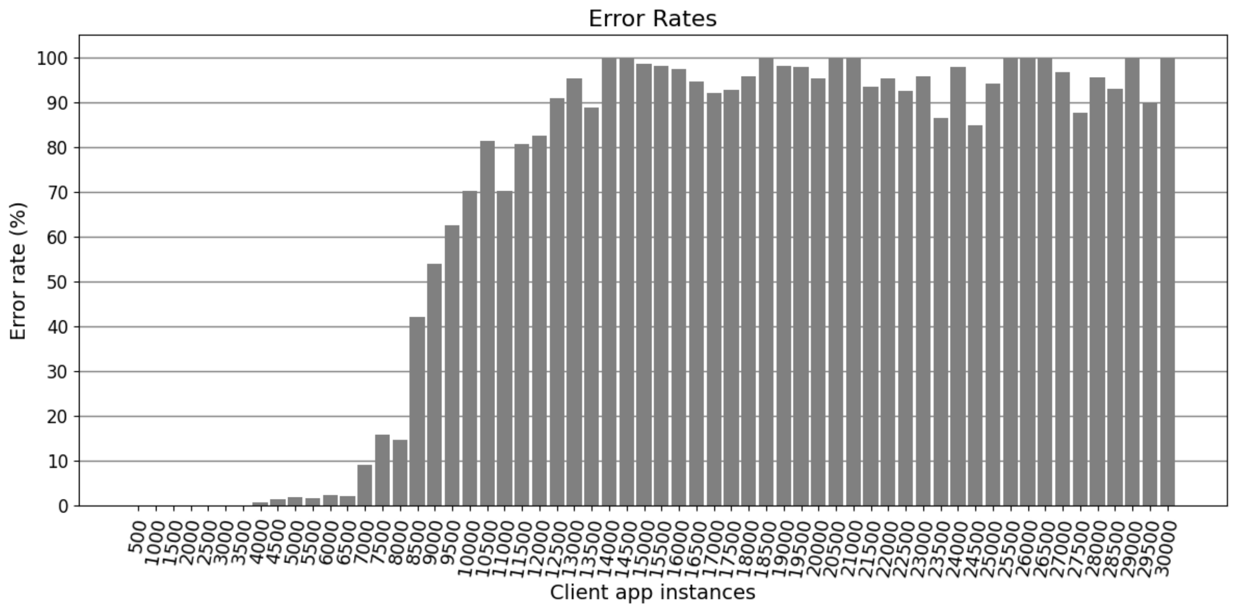


Figure 6-61 Error Rate vs. Number of Client Apps, with 1-peer Hyperledger Fabric network (run 1)

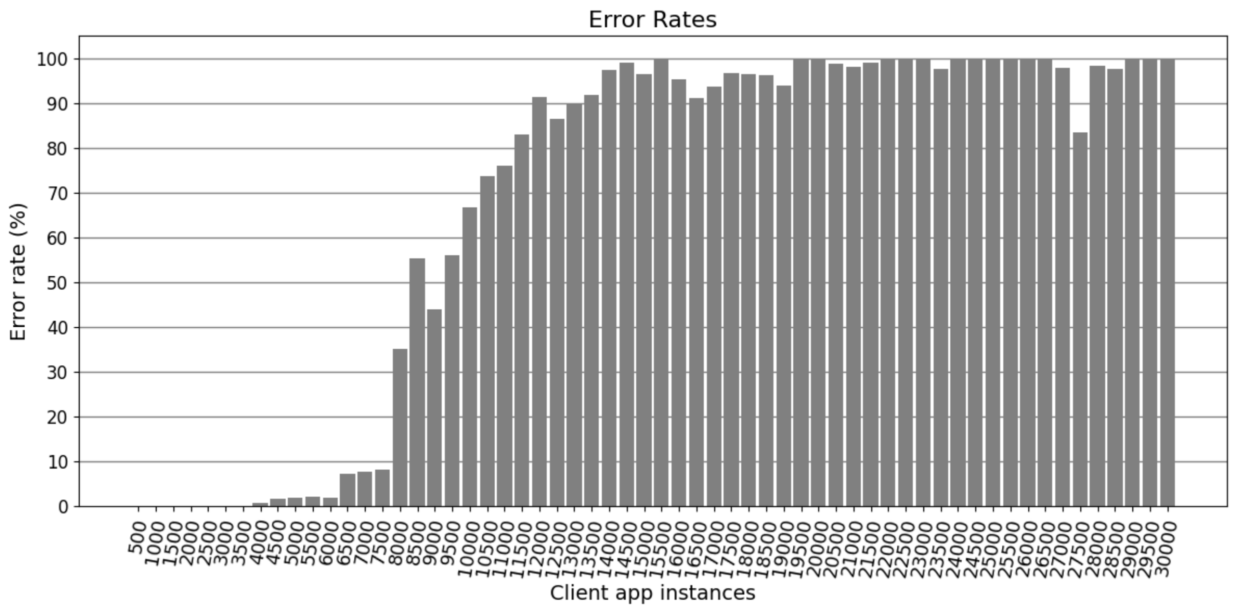


Figure 6-62 Error Rate vs. Number of Client Apps, with 1-peer Hyperledger Fabric network (run 2)

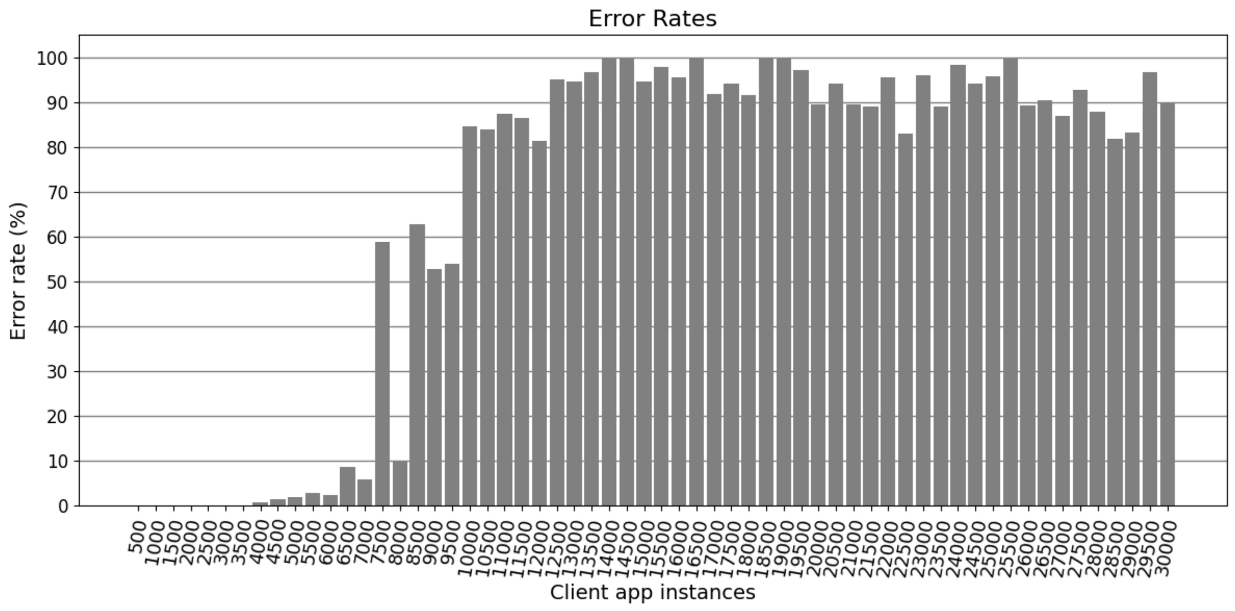


Figure 6-63 Error Rate vs. Number of Client Apps, with 1-peer Hyperledger Fabric network (run 3)

The three charts show the system can handle a substantial load without encountering errors until 4000 client apps. From 4000 to around 7000 client apps, the error rates remained below 10%. There was a remarkable jump at 8500 for run 1, 8000 for run 2, and 7500 for run 3. After that, the error rate went up quickly and hit 100% at 14000 client apps for runs 1 and 3 and 15500 client apps for run 2.

At this stage, the Hyperledger Fabric network on the Blockchain VM was re-configured to have nine peers. The same load-testing process was then repeated another three times, resulting in another three error rate bar charts:

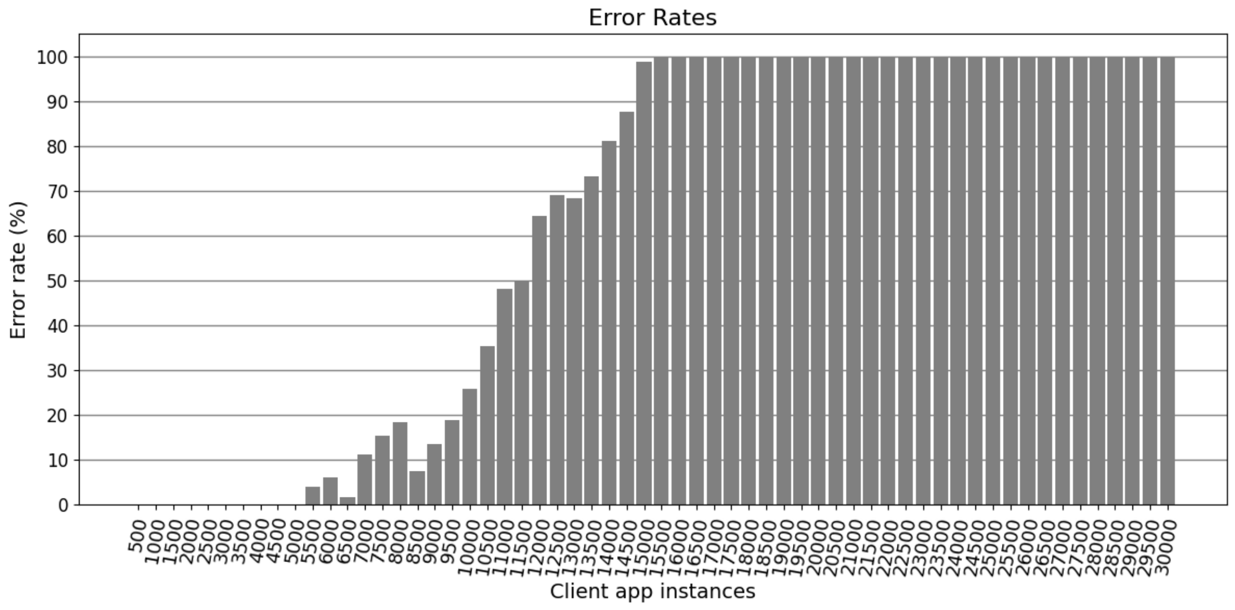


Figure 6-64 Error Rate vs. Number of Client Apps, with 9-peer Hyperledger Fabric network (run 1)

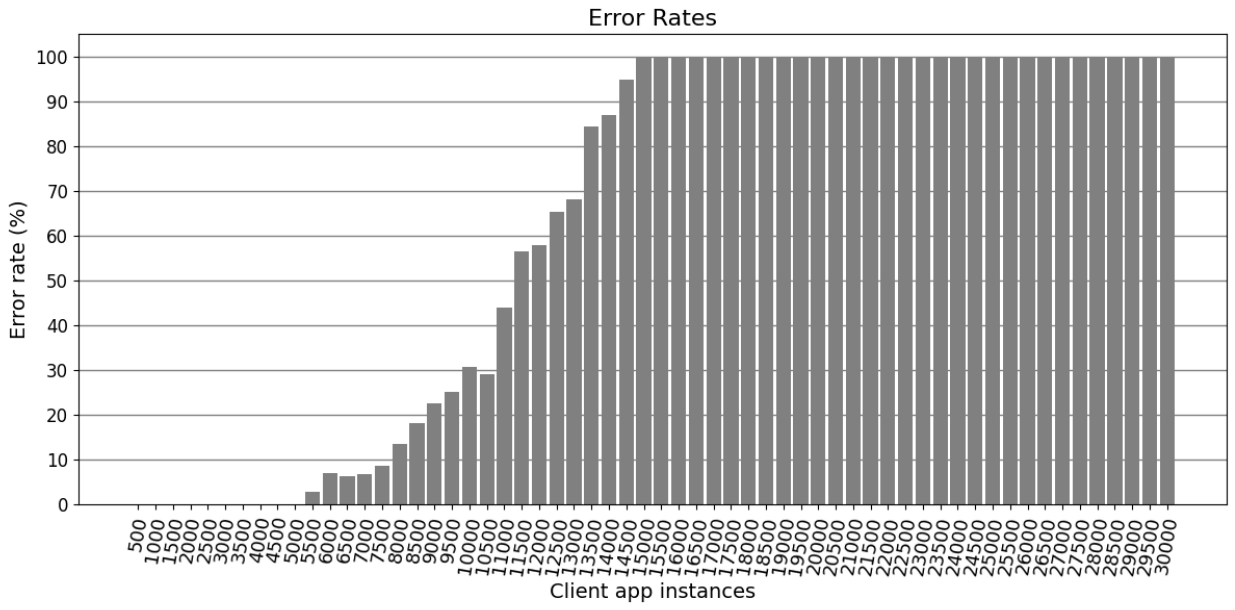


Figure 6-65 Error Rate vs. Number of Client Apps, with 9-peer Hyperledger Fabric network (run 2)

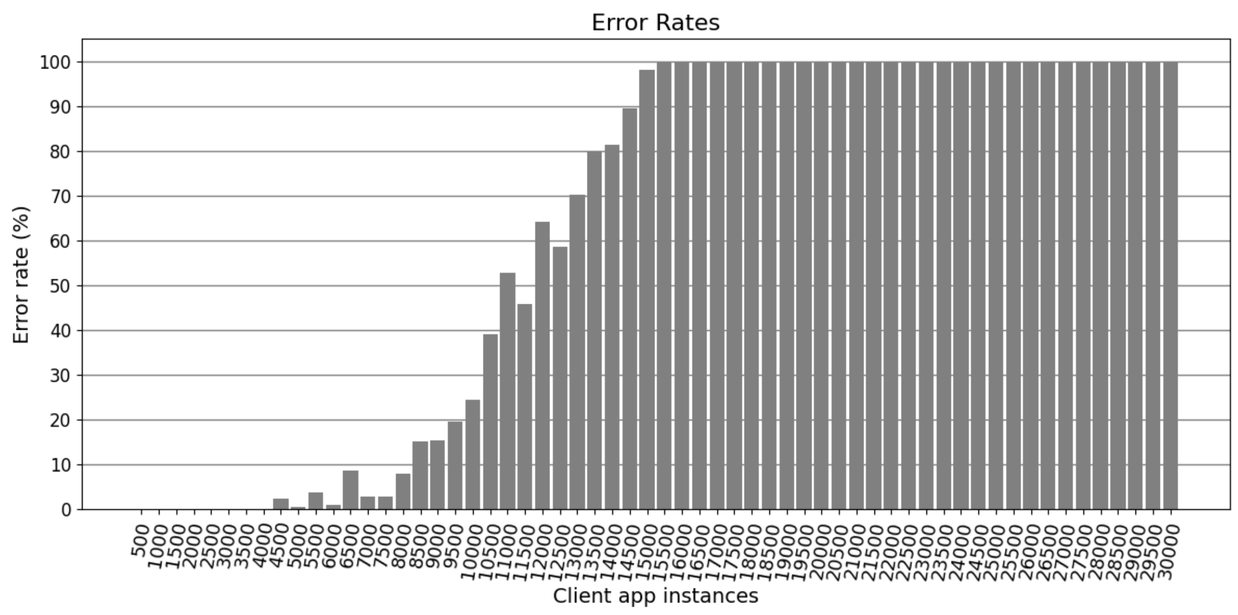


Figure 6-66 Error Rate vs. Number of Client Apps, with 9-peer Hyperledger Fabric network (run 3)

The three charts shows that the system can handle a substantial load without encountering errors until 5000 client apps for run 1 and 2 and until 4500 client apps for run 3. From 4500 to around 6500 client apps for run 1, 7500 client apps for run 2, and 8000 for run 3, the error rates remained below 10%. Beyond that point, the error rates gradually reached 100% with no drastic spike like the previous 1-peer setup. The error rate hit 100% at 15000 client apps for run 2 and 15500 for run 1 and 3.

The following table illustrates the overall trend of the six error rates charts:

Table 6-6 Comparing the error rates of the 1-peer setup and 9-peer setup

	1-peer setup	9-peer setup
Error rate at 0%	Up to 4000 client apps	Up to 4500-5000 client apps
Drastic jump of more than 20% error rate	Yes, at around 7500-8500 client apps	No
How the error rate approached 100%	Quickly	More slowly, the slope before the error rates hit 100% is less steep
When did the error rate hit 100%	At around 14000-15500 client apps	At around 15000-15500 client apps

After the error rate had reached 100%	Fluctuated and sometimes came back down to the 80-90% range	It Stayed the same after the error rate had hit 100%

The table provides the answer to *research question 3*. With the current implementation, the system can withstand around 14000 client apps before overloading and breaking down. Scaling up the Hyperledger Fabric network to nine peers made the system more resilient and soothed out the error rate slope compared to only one peer. It is worth noticing that Hyperledger Fabric supports more than nine peers. However, Fablo only supports up to nine peers. Having more than nine peers is likely to improve performance. This is discussed in the future work chapter as well.

6.3 Summary

The experiments showed that the proposed solution works, and the research questions were answered:

- *Research question 1*: What is the system's overhead latency in milliseconds when transitioning from not using the proposed solution to using it?
  Answer: For a set of 500 client apps hitting the blockchain-based solution simultaneously, the solution added an overhead latency of around 1577.145 to 1746.827 ms. As the number of client apps increased, the overhead latency rose linearly.
- *Research question 2*: What is the specific latency impact, in milliseconds, of the chosen blockchain technology?
  Answer: For a set of 500 client apps hitting the blockchain-based solution simultaneously, using the Hyperledger Fabric as the blockchain added an overhead latency of around 1230.53 to 1321.601ms. As the number of client apps increased, the overhead latency rose linearly.
- *Research question 3:* What is the system's maximum concurrent client app capacity?
  Answer: The proposed solution can withstand the concurrent load of around 14000 client apps before overloading and breaking down.

Furthermore, the first quantile Q1 and third quantile Q3 of every load-testing result from experiment 1 can be summarized in the following line chart:

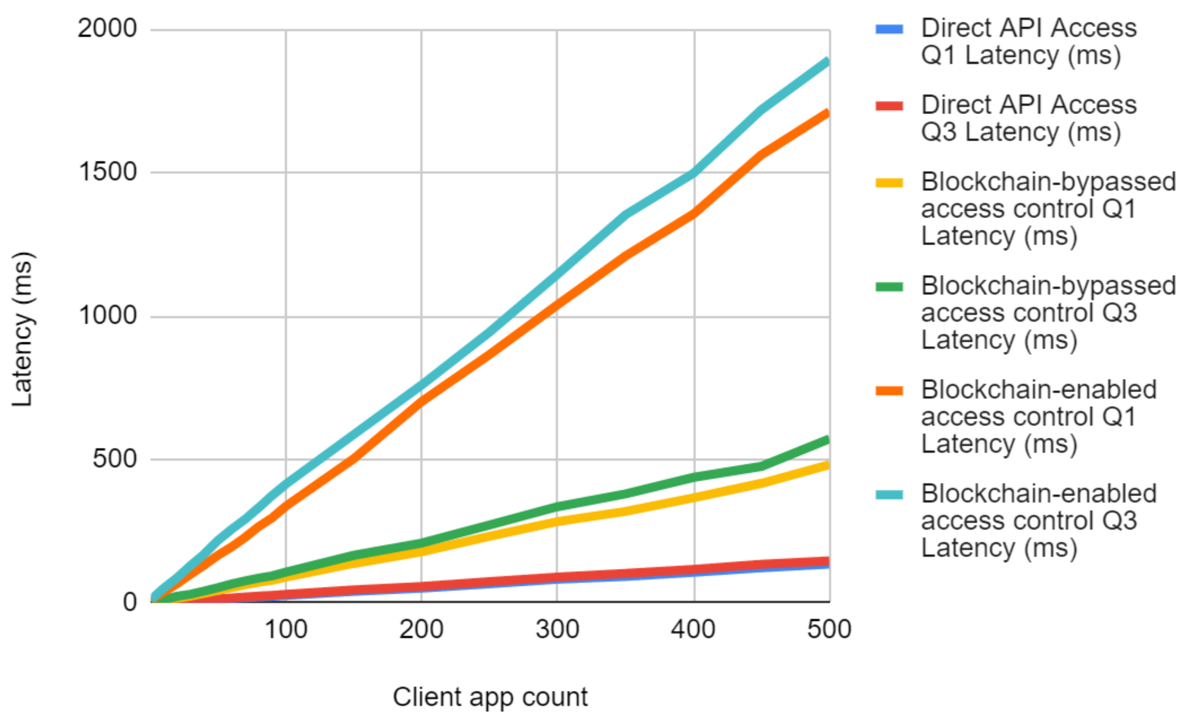


Figure 6-67 Experiment 1 summary

As shown in the chart, transitioning from direct API access to blockchain-bypassed access control, i.e., just adding a proxy hop between the client and the API, already adds significant latency for a larger number of clients. After that, the blockchain choice for access control integration, Hyperledger Fabric, in this case, added even further significant latency and can get up to nearly 2 seconds roundtrip time for worst case scenario for 500 client apps.

These findings indicate the trade-off between security and performance in adopting a blockchain-based approach for access control. While the blockchain solution inherently provides enhanced decentralization and security, it also introduces latency, which becomes increasingly prominent under high loads. This information is crucial for system architects and decision-makers when determining the suitability of such a solution for their specific use case and expected user loads. API providers and client app developers can consider using the Hyperledger Fabric-based solution depending on the performance requirements and use case.

7. CONCLUSIONS AND FUTURE WORK

7.1 Conclusion

Blockchain technology has gotten much attention in recent years. It has been used in many industries to enforce access control on assets such as cryptocurrencies, medical records, and supply chain items. Despite its widespread use, there has been limited exploration into utilizing blockchain to control API endpoint access. This area is interesting and challenging at the same time because, unlike conventional assets, API endpoints are subject to frequent and intensive access. Moreover, blockchain can introduce latency. Thus, proper implementation and performance evaluation is essential for developing a functional blockchain-based API access control solution. Employing blockchain yields the following key advantages:

- By nature, a blockchain-based system inherits an immutable data structure. New data gets appended to the blockchain, and once it is in, it cannot be changed or falsified, even by the system administrators.
- Every interaction with the blockchain is inherently recorded, providing traceability. Any participant in the blockchain network can get access and traverse the blockchain to figure out all the events that have happened.
- Hyperledger Fabric, the blockchain technology used to implement the solution proposed by this thesis, offers the ability to write smart contracts in common programming languages. Smart contracts are applications that run directly on the blockchain. Because it can be written in familiar languages instead of a domain-specific language, developers can adopt the technology, bring their experience, and maintain the code base more quickly and efficiently.

Thanks to using blockchain, the proposed solution is functional, including the following primary workflows:

- The API provider could add and manage access to the API endpoints.
- The client developer was able to request access to the endpoint.
- The client app was able to send traffic to the endpoint.

Furthermore, the Blockchain Explorer section of the implementation chapter showed a web view where all blocks and transactions that occurred on the Hyperledger Fabric blockchain could be inspected, providing system traceability.

The system architecture can be summarized in the following diagram:

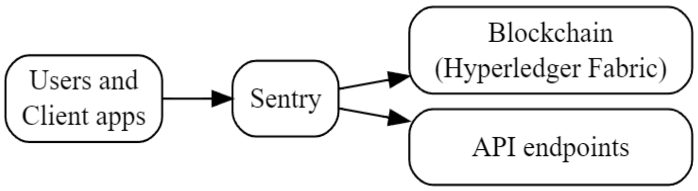


Figure 7-1 Simplified architecture.

Sentry serves as the gateway of the system. The addresses of the servers hosting the API endpoints are hidden from the users and client apps. Thus, Sentry is responsible for:

- Managing user interactions.
- Handling traffic from client apps and forwarding it to the protected API endpoints only if the client apps have been granted access to those endpoints.
- Engaging with the blockchain to store and retrieve access control data.

The core access control logic was designed based on the discrete access control type (DAC) and is written in JavaScript using the Node.js SDK provided by Hyperledger Fabric. Similarly, Sentry is also implemented in JavaScript, leveraging the Node.js runtime and the Express web framework.

Performance evaluation of the system was conducted on virtual machines (VMs) hosted within a private cloud infrastructure. The conducted experiments revealed the potential overhead latency introduced by the system, including its integration with Hyperledger Fabric. Additionally, the experiments demonstrated the system's remarkable ability to handle a substantial load of traffic generated by many client applications.

7.2 Contribution

While Bitcoin and Ethereum blockchains dominate the cryptocurrency use case, Hyperledger Fabric has been used extensively in enterprises, spanning various industries and use cases. Nevertheless, the access control use case for API endpoints has received limited attention, despite the potential of Hyperledger Fabric to address it. This thesis explored this aspect, presented an architectural proposal, developed a functional and resilient solution including a straightforward web user interface, and evaluated both the system and Hyperledger Fabric's overhead latency. Additionally, the system's traceability, an inherent characteristic of blockchain, was also demonstrated.

7.3 Limitation and Future Work

Like any work, this thesis has weaknesses, limitations, and potential areas for improvement. The following points highlight the limitations and disadvantages of the implementation and experiments, along with suggestions for future enhancements:

- *Tooling:* The tools, libraries, and utilities around Hyperledger Fabric are constrained and often challenging to use. Some tools are unmaintained, obsolete, or not stable. For instance, the Hyperledger Caliper Benchmarks tool encountered issues during setup despite following the documentation, inhibiting further experimentation. Furthermore, the tool is at version v0.5.0, which is unstable [26].
- *Hardware resources*: The hardware resources employed for the implementation and experiments were limited compared to enterprise-grade hardware. Future work could utilize more powerful machines with increased CPU and memory capacities.
- *CPU and memory monitoring:* The CPU and memory of all machines should have been monitored throughout the experiments. Future work could do this for a more comprehensive understanding of the system's resource demands.
- *Private data:* Although end users are provided with Hyperledger Fabric wallets in the proposed solution, the wallets remain encrypted. Sentry is the only component that can decrypt and use the wallets to interact with the Hyperledger Fabric network directly. However, hiding the sensitive data within the Hyperledger Fabric network will further

enhance the system’s security. Hyperledger Fabric’s private data collection feature could be explored for this purpose.

- *Access limits:* Future work could implement usage quotas or time-based restrictions that API providers at the point of granting access to specific API endpoints.
- *API providers hosting their peers:* Future work could explore the scenario in which API providers participate in the Hyperledger Fabric network by hosting their peers and having their own Hyperledger Fabric organization defined for further decentralization.
- *Evaluate all user flows:* The performance experiments focused solely on the stage after access to an endpoint had been granted, and client apps was able to initiate traffic. Future work could run performance assessments for all stages, including registration, access requests, access granting, revocation.
- *Scalability limit:* The Hyperledger Fabric network used for development and evaluation was established using Fablo, which supports up to nine peers. As Hyperledger Fabric can accommodate more peers, using an alternative tool to configure the network would enable the network to scale more. More experiments can then be conducted.
- *Other blockchain platforms:* Future work could implement the access control logic in alternative blockchain platforms, conduct comparative performance evaluations, and assess their suitability.
- *Traditional solutions:* Exploring a complete API endpoint access control system implemented using traditional access control technology and comparing its performance to the blockchain-based solution is a potential idea for future research.
- *Cold start:* Real-world software systems gradually grow and serve increasing numbers of users before facing traffic spikes. Future work could adopt a more realistic approach, allowing the system to warm up and conduct load tests over an extended period.
- *Consensus liveness:* In Hyperledger Fabric, a consensus mechanism ensures peers reach the same state. However, checking whether peers are still in consensus after the mechanism has completed is undocumented. Further experiments could be conducted to verify this scenario.

- *Virtual users:* The K6 tool used for load-testing creates virtual users. They are actually threads competing for limited hardware resources in one physical machine. Future work could employ dedicated physical machines or involve real users to generate traffic.
- *Network latency within the blockchain:* While Hyperledger Fabric peers are isolated containers, they share the same machine in this thesis. Deploying the network on distinct physical machines would better emulate network latency effects within the blockchain.
- *Chaos engineering:* K6's consistent load-testing contrasts with the unpredictability of real-world traffic. Future work could simulate network and server failures, configuration changes, smart contract updates, random traffic spikes, etc.
- *Different cloud services:* Given the diversity of cloud providers, deploying multiple instances of the system across various services and comparing performance between them offers an opportunity for exploration.

8. REFERENCES

[1] B. De and B. De, *API management*. Springer, 2017.

[2] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, “Blockchain challenges and opportunities: A survey,” *International journal of web and grid services*, vol. 14, no. 4, pp. 352–375, 2018.

[3] V. C. Hu, D. Ferraiolo, D. R. Kuhn, and others, *Assessment of access control systems*. US Department of Commerce, National Institute of Standards and Technology~…, 2006.

[4] M. Swan, *Blockchain: Blueprint for a new economy*. “ O’Reilly Media, Inc.,” 2015.

[5] S. Nakamoto, “Bitcoin: a peer-to-peer electronic cash system. Decentral,” *Bus. Rev*, vol. 21260, 2008.

[6] E. Androulaki *et al.*, “Hyperledger fabric: a distributed operating system for permissioned blockchains,” in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[7] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[8] J. Shubheksha, “Understanding the Raft consensus algorithm: an academic article summary.” Accessed: Aug. 26, 2023. [Online]. Available: https://www.freecodecamp.org/news/in-search-of-an-understandable-consensus-algorithm-a-summary-4bc294c97e0d/

[9] R. S. Sandhu and P. Samarati, “Access control: principle and practice,” *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[10] P. Samarati and S. C. de Vimercati, “Access control: Policies, models, and mechanisms,” in *International school on foundations of security analysis and design*, Springer, 2000, pp. 137–196.

[11] S. Gusmeroli, S. Piccione, and D. Rotondi, “A capability-based security approach to manage access control in the internet of things,” *Math Comput Model*, vol. 58, no. 5–6, pp. 1189–1205, 2013.

[12] “What is Capability-based Security? | by Kevin Leffew | Medium.” Accessed: Oct. 04, 2023. [Online]. Available: https://medium.com/@kleffew/what-is-capability-based-security-227c6e5483a5

[13] A. Ubale Swapnaja, G. Modani Dattatray, and S. Apte Sulabha, “Analysis of dac mac rbac access control based models for security,” *Int J Comput Appl*, vol. 104, no. 5, pp. 6–13, 2014.

[14] D. D. F. Maesa, P. Mori, and L. Ricci, “Blockchain based access control services,” in *2018 IEEE international conference on internet of things (ithings) and IEEE green computing and*

communications (greencom) and IEEE cyber, physical and social computing (cpscom) and IEEE smart data (smartdata), 2018, pp. 1379–1386.

[15] J. P. Cruz, Y. Kaji, and N. Yanai, “RBAC-SC: Role-based access control using smart contract,” *Ieee Access*, vol. 6, pp. 12240–12251, 2018.

[16] “Fablo.” Accessed: Aug. 26, 2023. [Online]. Available: https://github.com/hyperledger-labs/fablo

[17] “What is Docker | Oracle Canada.” Accessed: Aug. 26, 2023. [Online]. Available: https://www.oracle.com/ca-en/cloud/cloud-native/container-registry/what-is-docker/

[18] “Hyperledger Fabric Contract API.” Accessed: Aug. 26, 2023. [Online]. Available: https://hyperledger.github.io/fabric-chaincode-node/main/api/

[19] “About | Node.js.” Accessed: Aug. 26, 2023. [Online]. Available: https://nodejs.org/en/about

[20] “Express - Node.js web application framework.” Accessed: Aug. 26, 2023. [Online]. Available: https://expressjs.com/

[21] “Hyperledger Fabric SDK for Node.js.” Accessed: Aug. 26, 2023. [Online]. Available: https://hyperledger.github.io/fabric-sdk-node/main/module-fabric-network.html

[22] “http-party/node-http-proxy: A full-featured http proxy for node.js.” Accessed: Aug. 26, 2023. [Online]. Available: https://github.com/http-party/node-http-proxy

[23] “Pug documentation.” Accessed: Aug. 26, 2023. [Online]. Available: https://pugjs.org/api/getting-started.html

[24] “k6 Documentation.” Accessed: Aug. 26, 2023. [Online]. Available: https://k6.io/docs/

[25] “hyperledger-labs/blockchain-explorer.” Accessed: Aug. 26, 2023. [Online]. Available: https://github.com/hyperledger-labs/blockchain-explorer

[26] “hyperledger/caliper: A blockchain benchmark framework to measure performance of multiple blockchain solutions https://wiki.hyperledger.org/display/caliper.” Accessed: Aug. 26, 2023. [Online]. Available: https://github.com/hyperledger/caliper

9. APPENDIX: CONNECTION POOL FOR HYPERLEDGER FABRIC GATEWAY - ELIMINATING SYSTEM BOTTLE NECK

This appendix presents the experiments not to answer the research questions but rather to identify the bottleneck in the proposed solution and implement a connection pool for the Hyperledger Fabric Gateway to tune its performance. The system could withstand a concurrent load of up to 300 client apps without the connection pool. The connection pool helps increase the number of concurrent client apps to 17000.

The system was also deployed to three VMs on the private cloud, as described in the performance evaluation chapter. After that, different scenarios were designed so that the system's performance and weaknesses were exposed:

- Scenario 1: Direct API access
- Scenario 2: Introducing the Blockchain VM with Hyperledger Fabric bypassed.
- Scenario 3: Enabling Hyperledger Fabric
- Scenario 4: Performance of access control data retrieval
- Scenario 5: Scaling up peers in the Hyperledger Fabric network.
- Scenario 6: Connection pool for Hyperledger Fabric gateway

These scenarios were structured to escalate gradually, starting with fundamental load-testing, and systematically introducing essential components. Each scenario builds upon the previous one to pinpoint any bottleneck and part that can be tweaked for better performance. Regarding the metrics, both latency and error rates were used to trace weaknesses. Latency in this context means how long it takes for the client app to send a request to one of the protected API endpoints and receive a response. An error rate in this context means the frequency at which the response status code is not 200, indicating the proportion of unsuccessful interactions compared to the total number of requests made.

9.1 Scenario 1: Direct API access

Focus: Determining the latency when the Blockchain VM is bypassed.

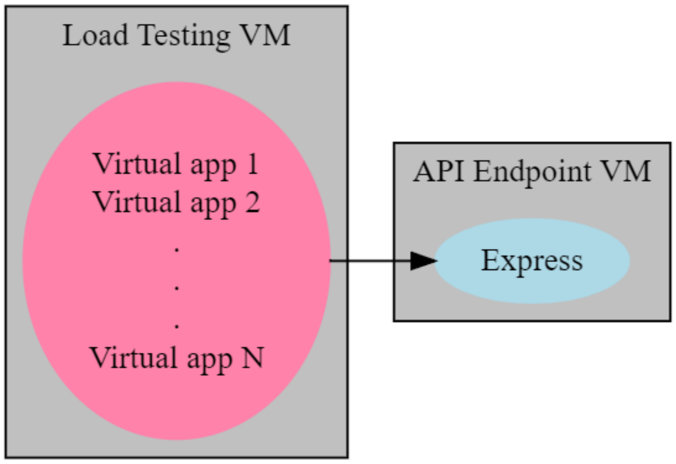


Figure 9-1 Scenario 1 Overview

The diagram shows only one component under load in scenario 1: the Express app on the API endpoints VM. After load-testing, the following scatter chart shows the latency of every request sent by one virtual client app instance:

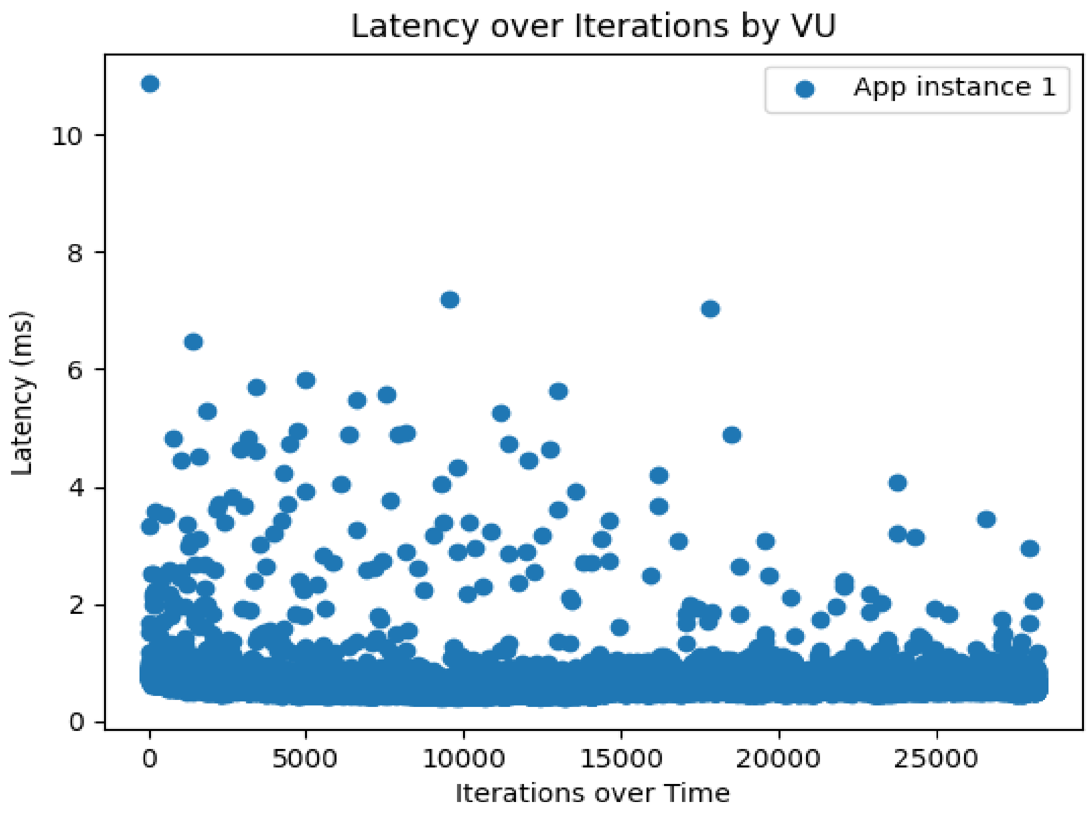


Figure 9-2 Scenario 1 latency scatter chart for one client app instance.

As shown in the chart, the client app managed to finish more than 25000 requests, and most of them had a latency of below 2ms. One request near the zeroth request reported the longest latency at more than 10ms.

The following scatter chart shows the latency of every request sent by ten virtual client app instances:

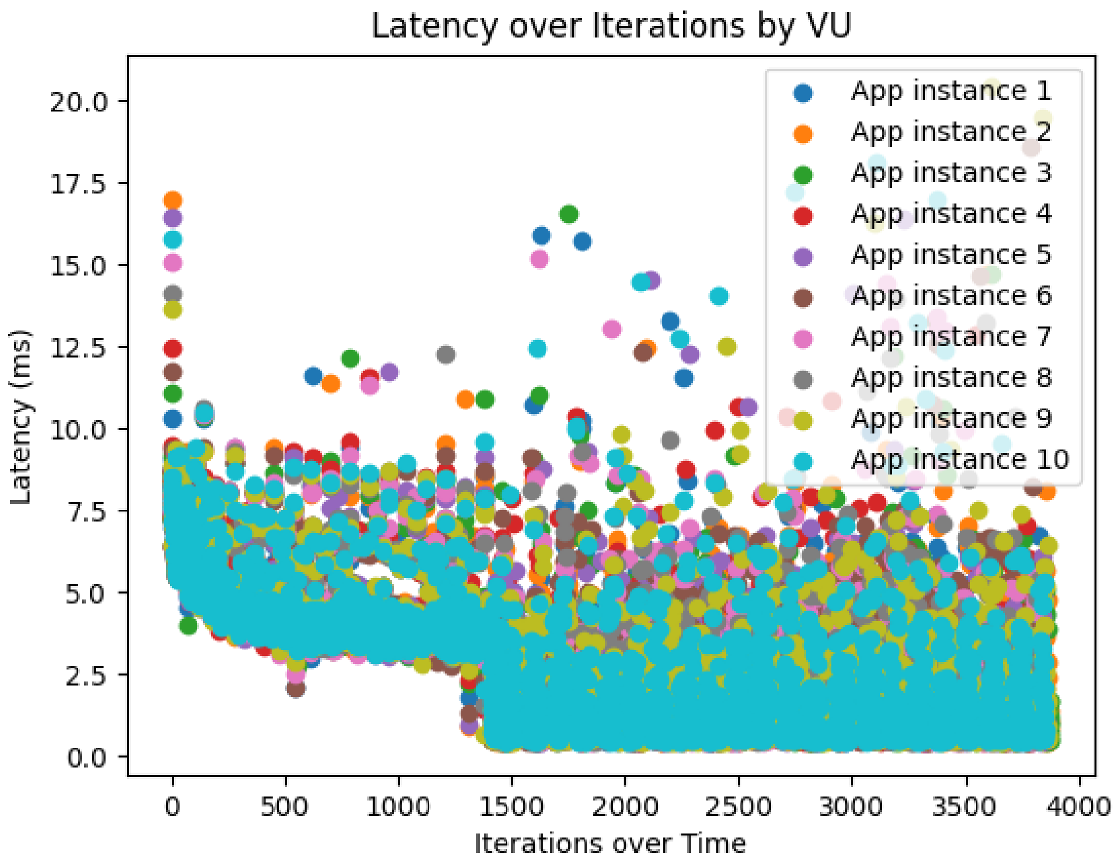


Figure 9-3 Scenario 1 latency scatter chart for ten client app instances.

The chart shows that the client app finished over 4,000 requests, most of which had a latency of below 10ms.

9.2 Scenario 2: Introducing the Blockchain VM with Hyperledger Fabric bypassed

Focus: Determining the latency when the Blockchain VM is enabled but with Hyperledger Fabric bypassed.

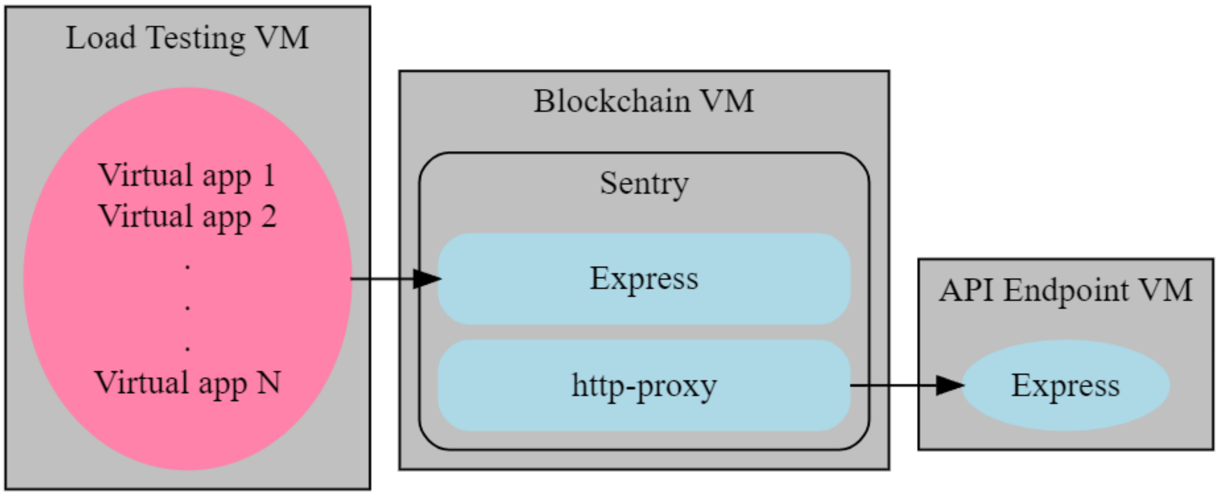


Figure 9-4 Scenario 2 overview.

Now that we have the baseline of how just the API endpoints VM handles traffic, thanks to scenario 1. In scenario 2, the blockchain gateway VM was enabled. There was no Hyperledger Fabric enabled for this scenario. There was just an Express app listening for HTTP traffic and using the http-proxy module to forward the traffic to the other Express app running on the API endpoint VM. After load-testing, the following scatter chart shows the latency of every request sent by one virtual client app instance:

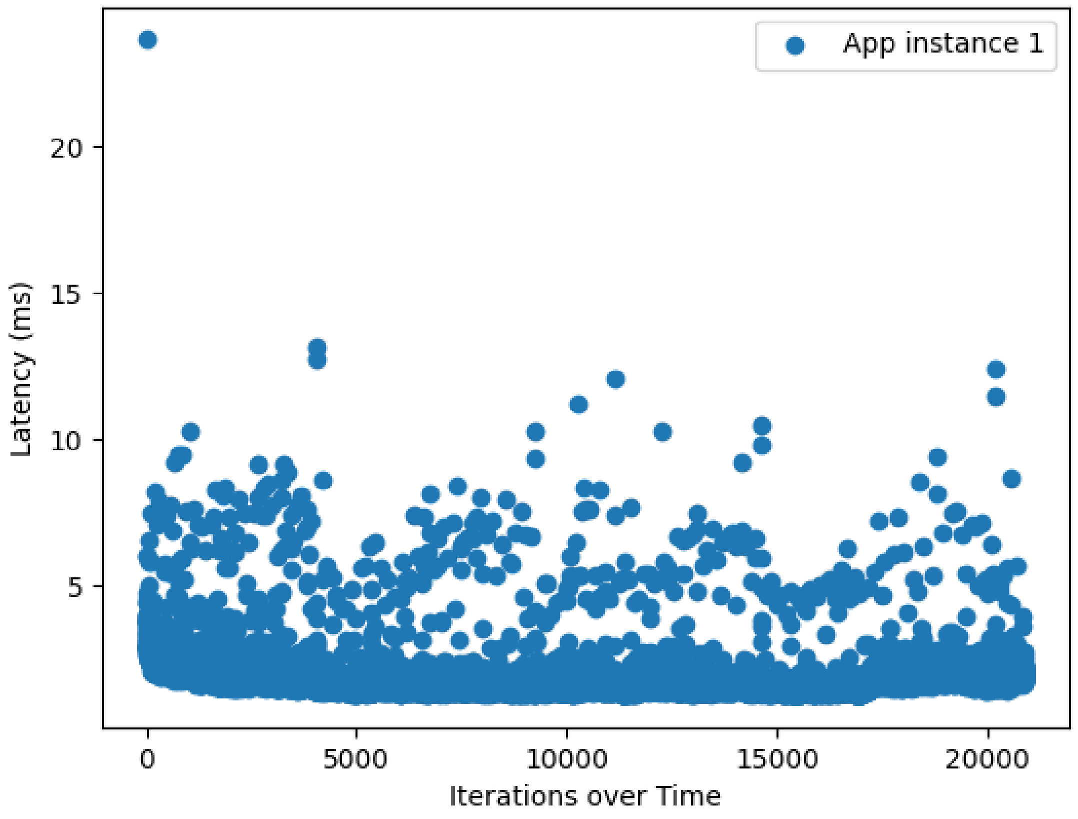


Figure 9-5 Scenario 2 latency scatter chart for one client app instance.

The chart shows that the client app finished over 20,000 requests, most of which had a latency of below 5ms. One request near the zeroth request reported the longest latency at more than 20ms.

The following scatter chart shows the latency of every request sent by ten virtual client app instances:

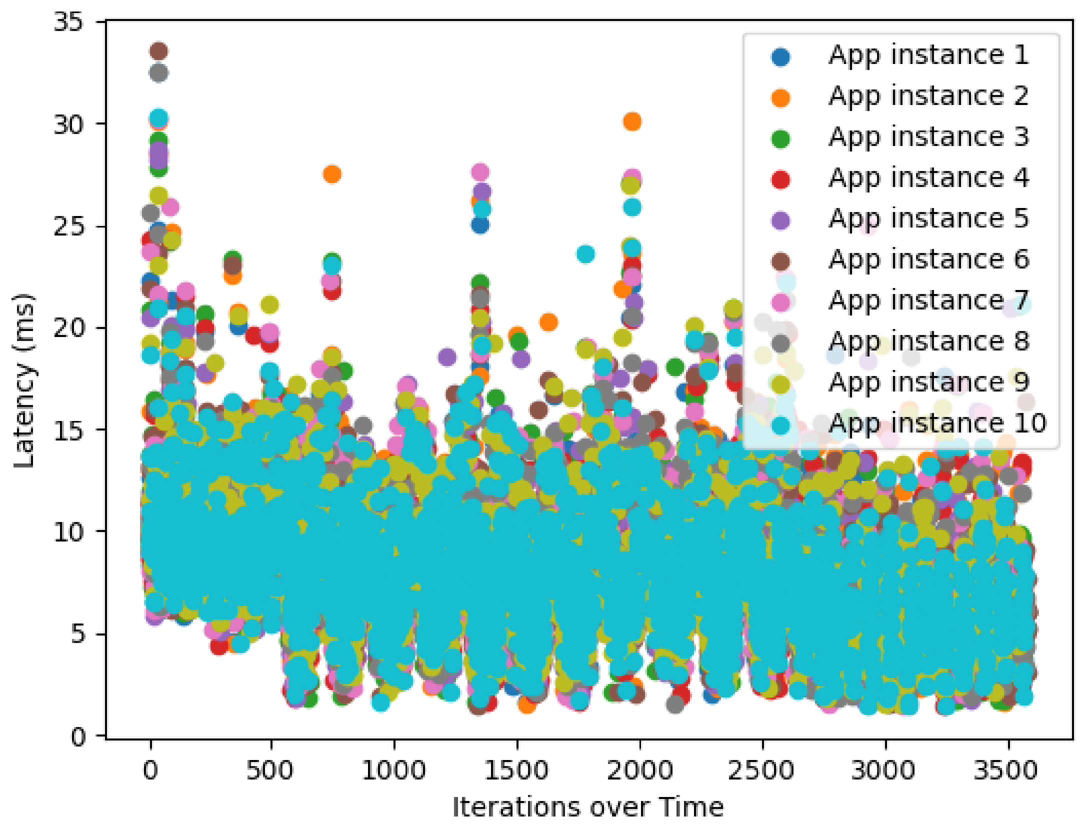


Figure 9-6 Scenario 2 latency scatter chart for ten client app instances.

As shown in the chart, the client app managed to finish more than a little more than 3,500 requests, most of which had a latency of below 15ms.

9.3 Scenario 3: Enabling Hyperledger Fabric

Focus: Determining the latency and error rate when Hyperledger Fabric is enabled and identifying the bottleneck.

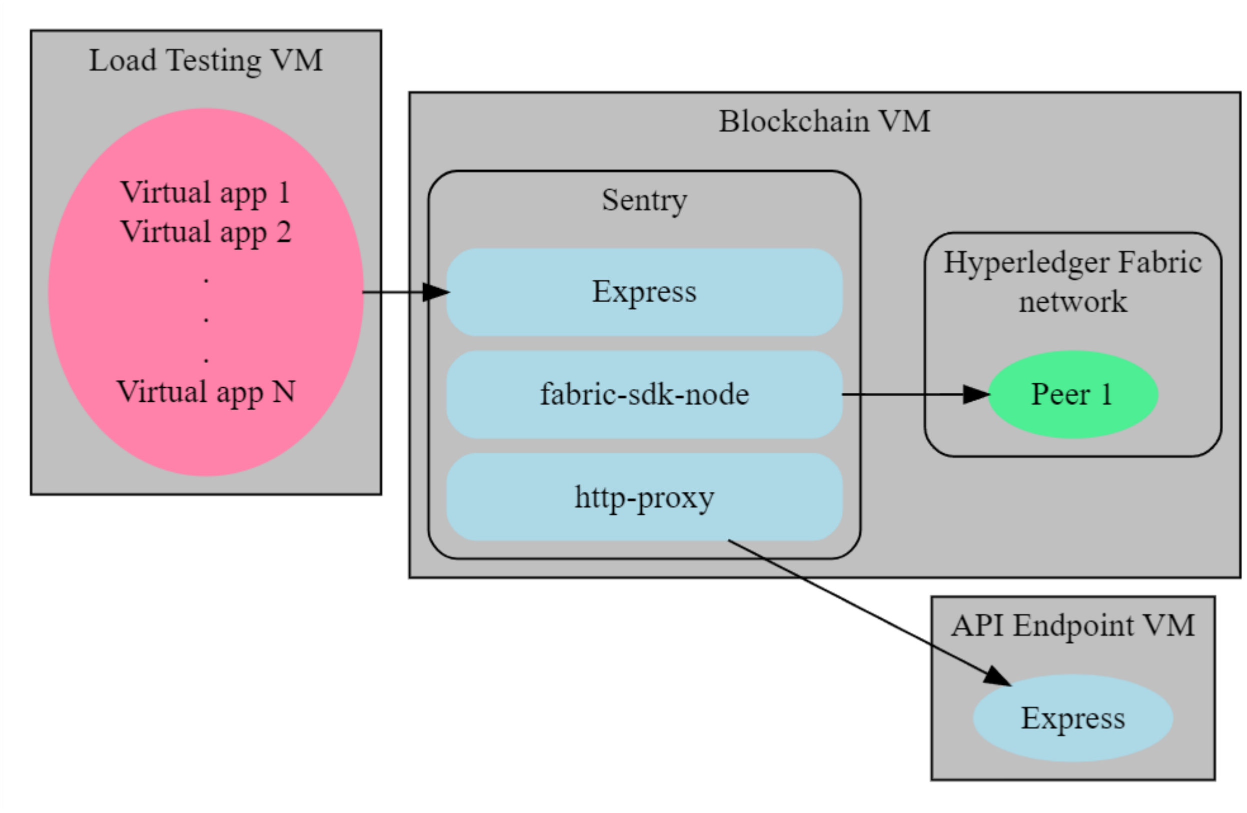


Figure 9-7 Scenario 3 overview.

In scenario 3, the Hyperledger Fabric network with one peer (Peer 1) was enabled. To process requests from the client app instances, the Express app (Sentry) retrieved and validated the access control data from Peer 1 before using the http-proxy module to forward the requests to the Express app on the API endpoints VM.

After load-testing, the following scatter chart shows the latency of every request sent by one virtual client app instance:

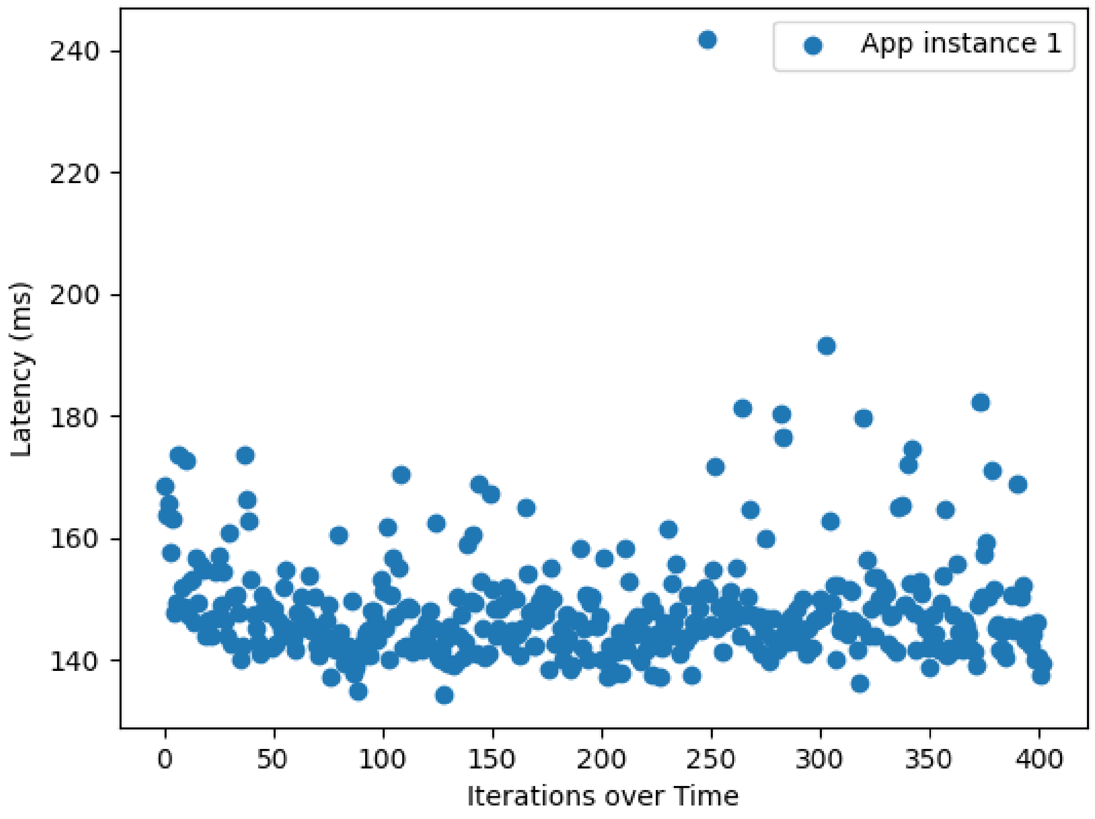


Figure 9-8 Scenario 3 latency scatter chart for one client app instance.

As shown in the chart, the client app only managed to finish around 400 requests, most of which had a latency of below 160ms. The blockchain interaction introduced remarkable latency.

The following scatter chart shows the latency of every request sent by ten virtual client app instances:

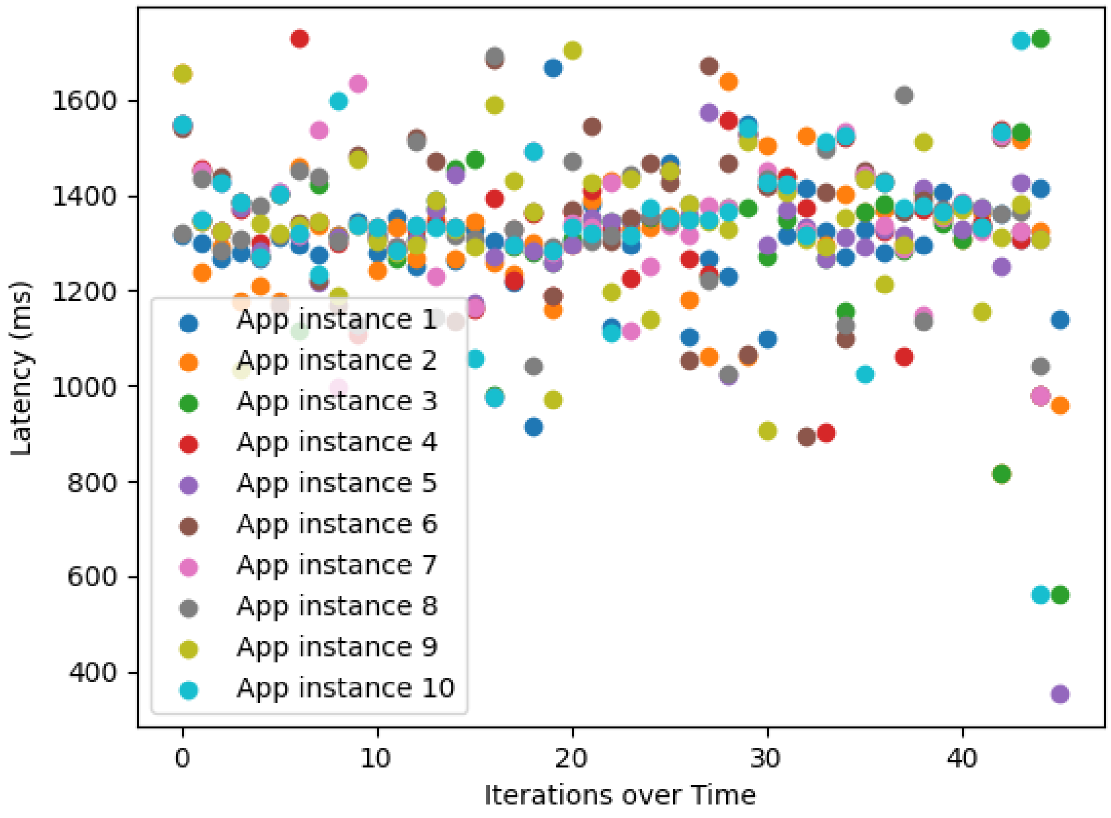


Figure 9-9 Scenario 3 latency scatter chart for ten client app instances.

The chart shows that the number of requests dropped significantly, and the latency increased.

At this stage, the load testing was repeated to determine the error rate. Growing sets of virtual client apps from 50 to 5000 were used. Each set still loaded the Blockchain gateway VM for one minute. The following bar plot is the result of the run and shows the error rates of those sets:

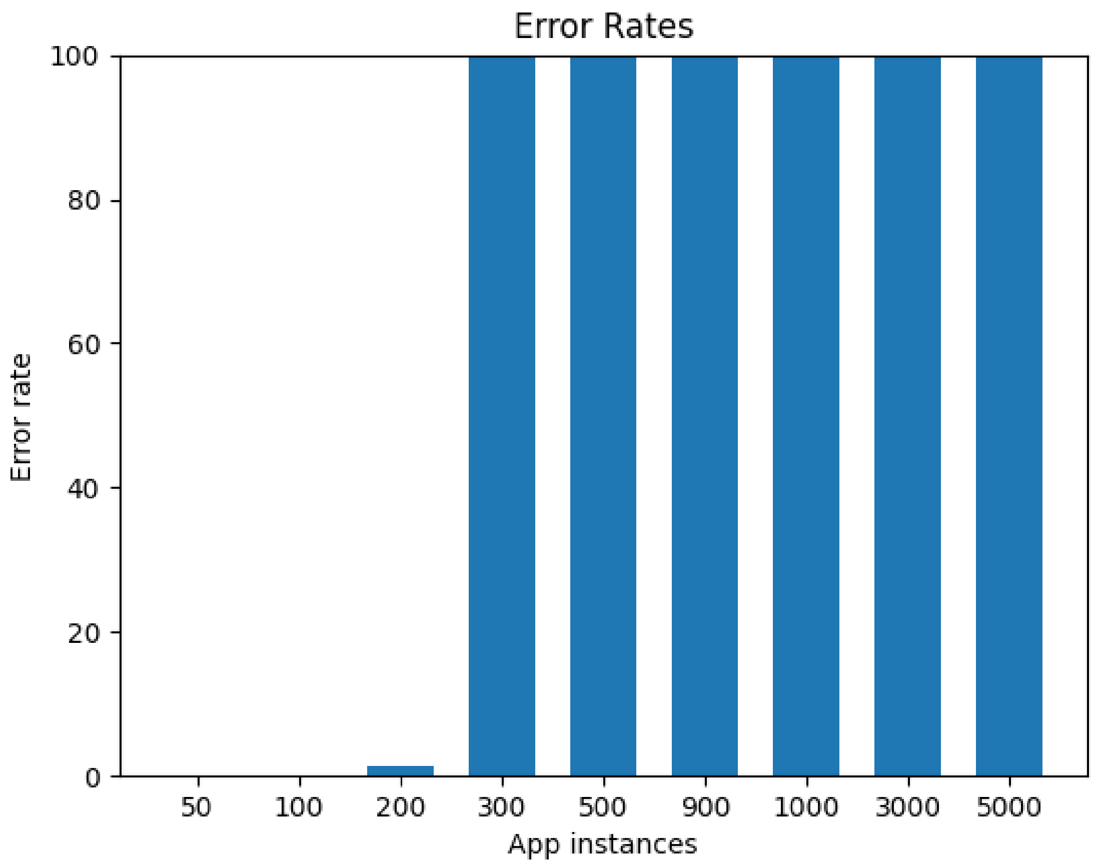


Figure 9-10 Scenario 3 error rate bar chart.

According to the chart, the system had a low error rate until 300 client apps simultaneously hit the Blockchain gateway VM. At that point, the system experienced a 100% error rate.

Thanks to the latency scatter chart and the error rate chart, the link between the Express app (Sentry) and the Hyperledger Fabric network was the system's bottleneck.

9.4 Scenario 4: Performance of Access Control Data Retrieval

Focus: Analyzing data retrieval performance without involving API Endpoint VM.

Scenario 3 helped to find that the link between the Express app (Sentry) and the Hyperledger Fabric network was the bottleneck of the system, as illustrated by the following figure:

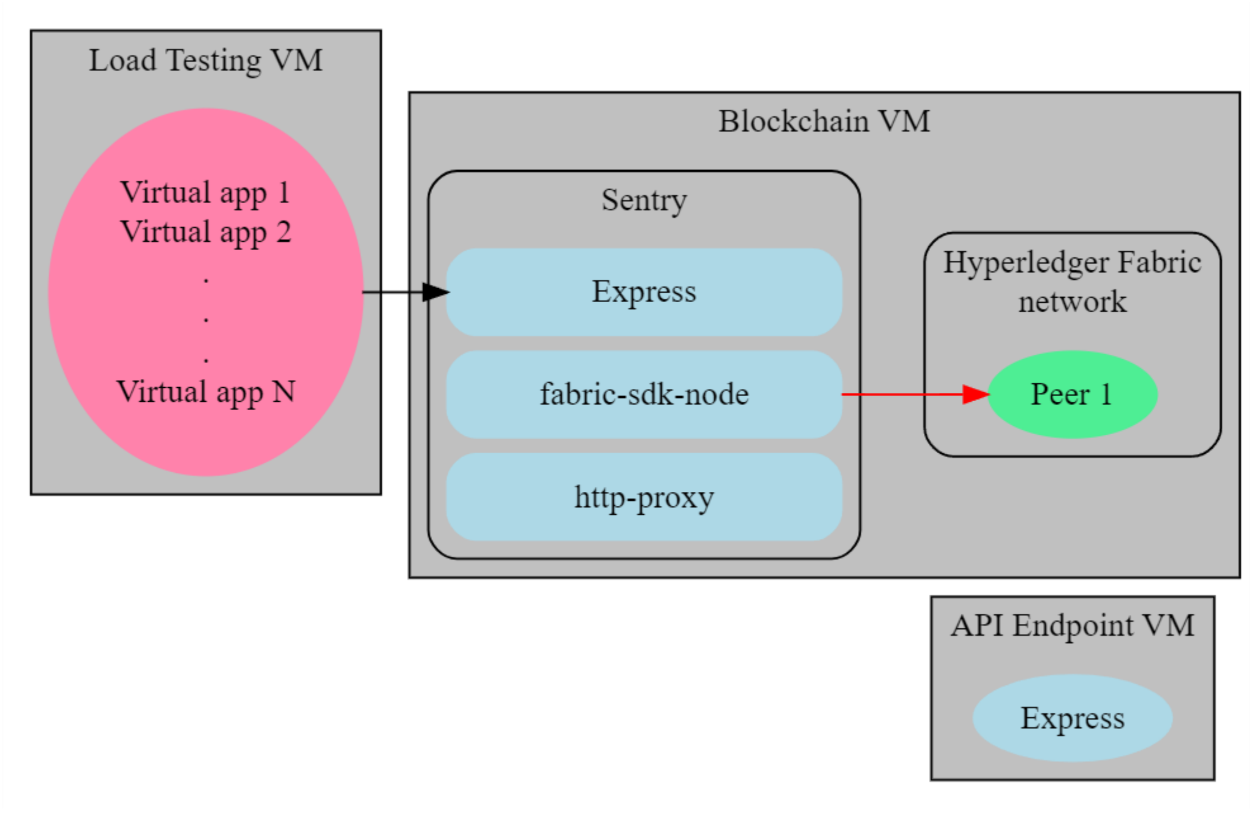


Figure 9-11 Scenario 4 highlights the bottleneck.

The API endpoints VM and http-proxy module were then deliberately omitted from scenario four so that all load-testing could focus on the bottleneck:

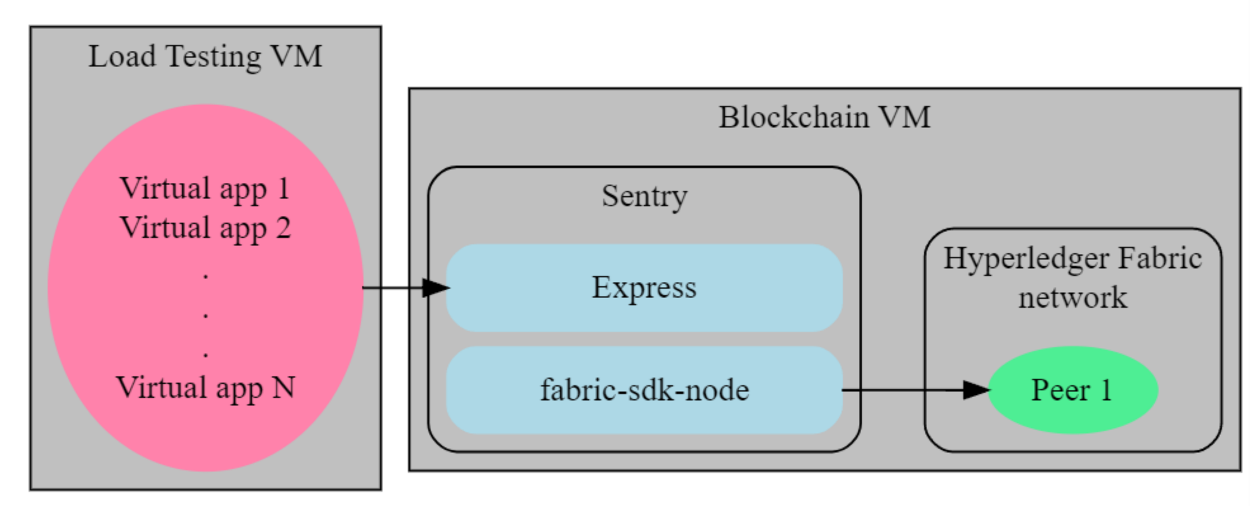


Figure 9-12 Scenario 4 overview.

After load-testing, the following scatter charts respectively show the latency of every request sent by one virtual client app instance and by ten virtual client app instances:

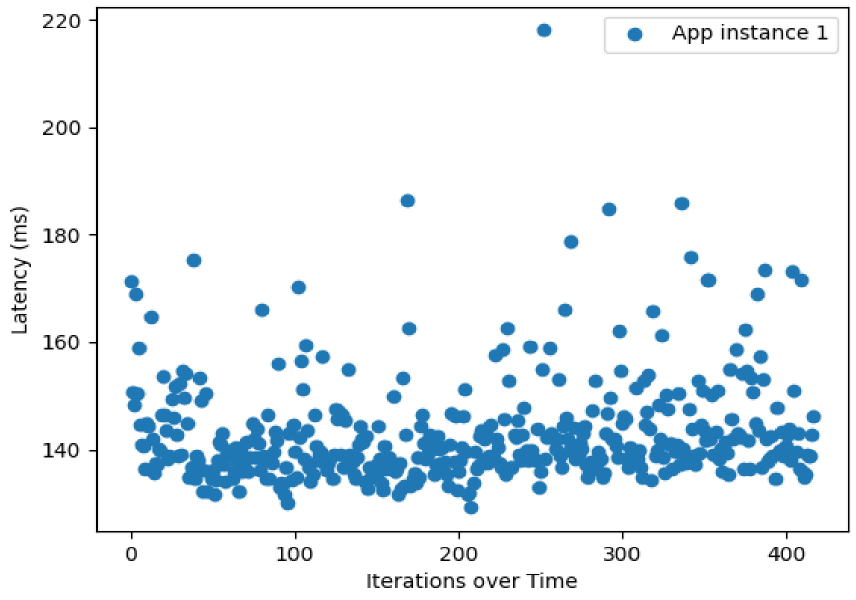


Figure 9-13 Scenario 4 latency scatter chart for one client app instance.

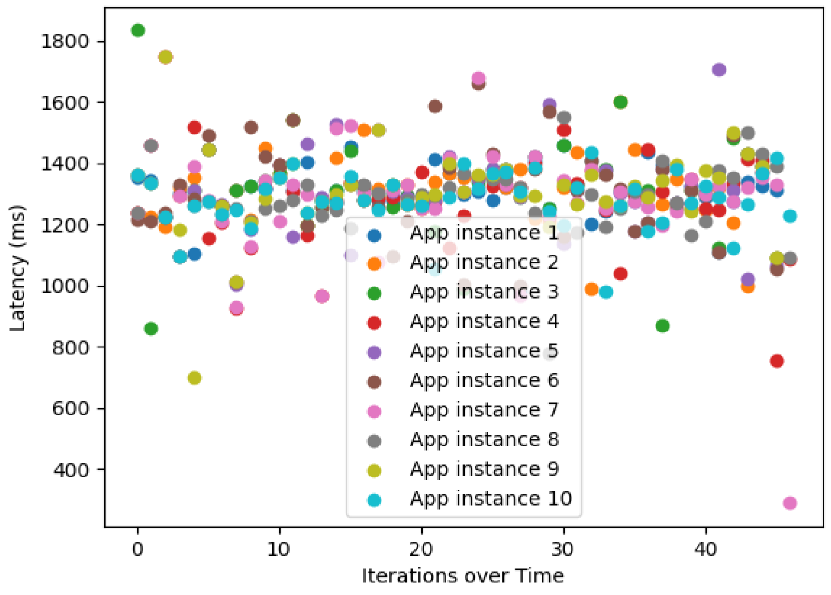


Figure 9-14 Scenario 4 latency scatter chart for ten client app instances.

The following bar plot shows the error rates derived from various sets of client app instances:

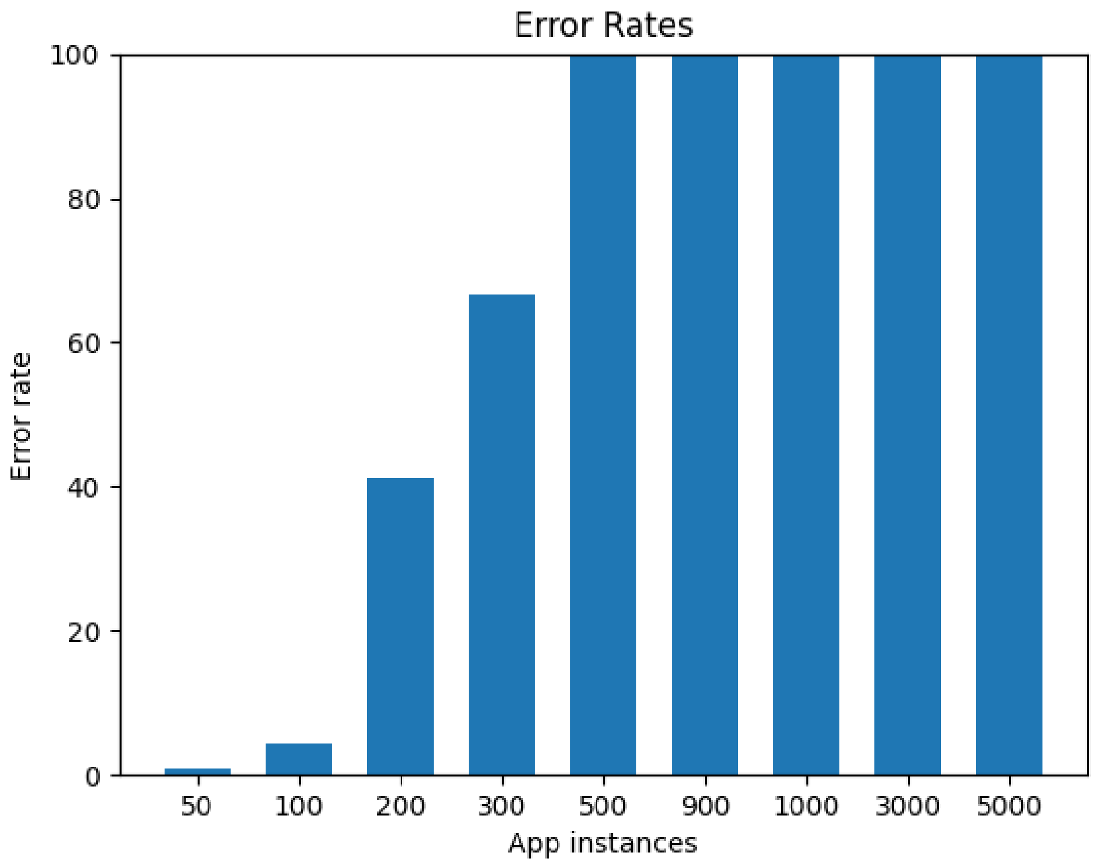


Figure 9-15 Scenario 4 error rate bar chart.

The system performance was slightly better with the API endpoints VM and http-proxy module out of the way, but overall, still unacceptable.

9.5 Scenario 5: Scaling Up Peers in the Hyperledger Fabric Network

Focus: Determining if increasing the number of peers inside the Hyperledger Fabric network can help improve the latency and error rate.

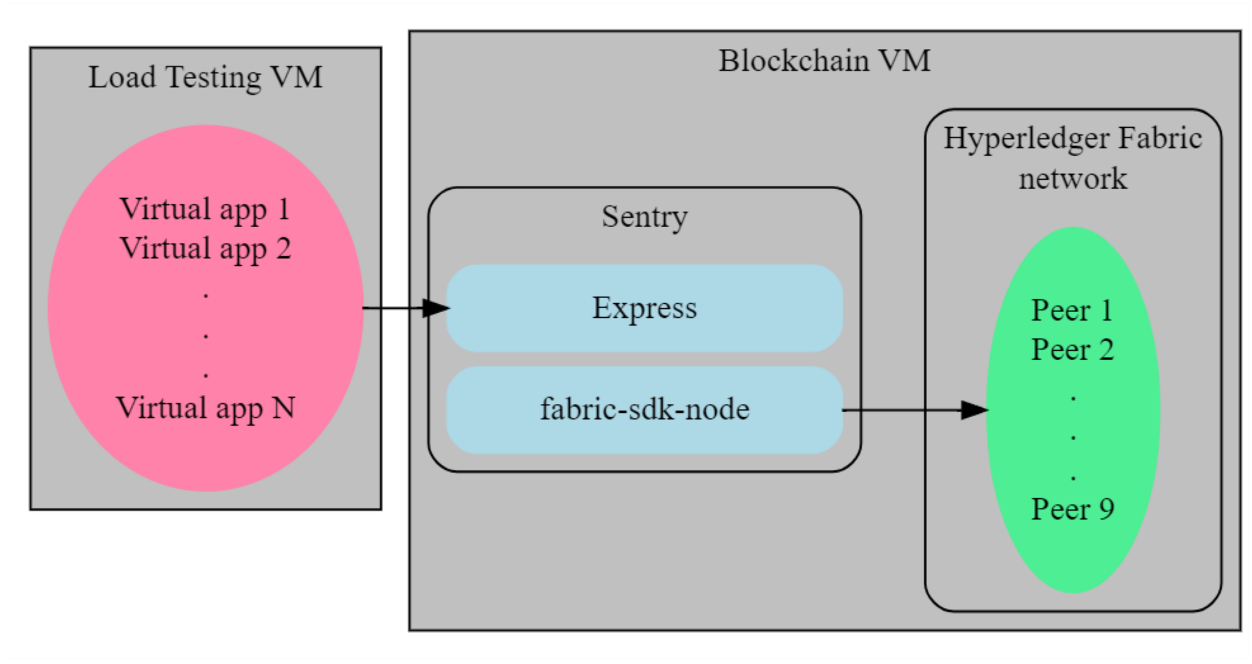


Figure 9-16 Scenario 5 overview.

The diagram shows nine peers in the Hyperledger Fabric network for scenario 5. That is the highest number of peers that Fablo allows. Fablo was the tool that helped create the Hyperledger Fabric network.

After load-testing, the following scatter charts respectively show the latency of every request sent by one virtual client app instance and by ten virtual client app instances:

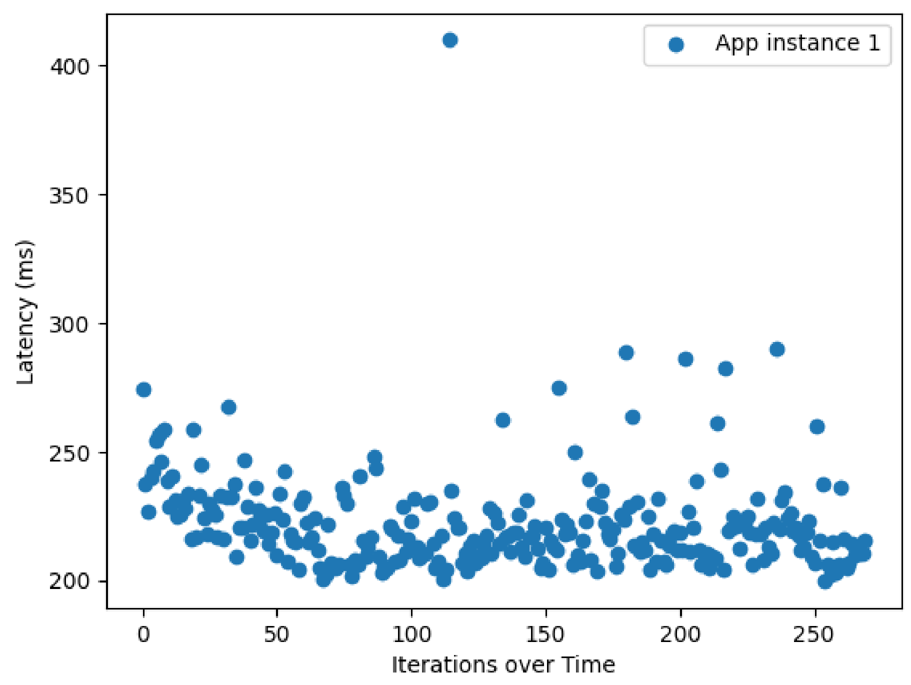


Figure 9-17 Scenario 5 latency scatter chart for one client app instance.

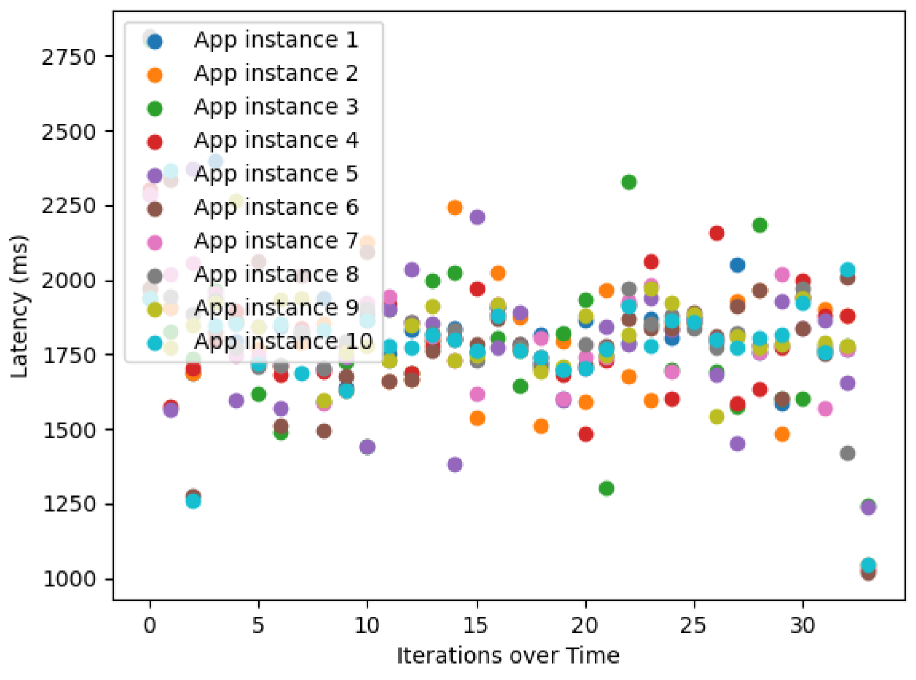


Figure 9-18 Scenario 5 latency scatter chart for ten client app instances.

The following bar plot shows the error rates derived from various sets of client app instances:

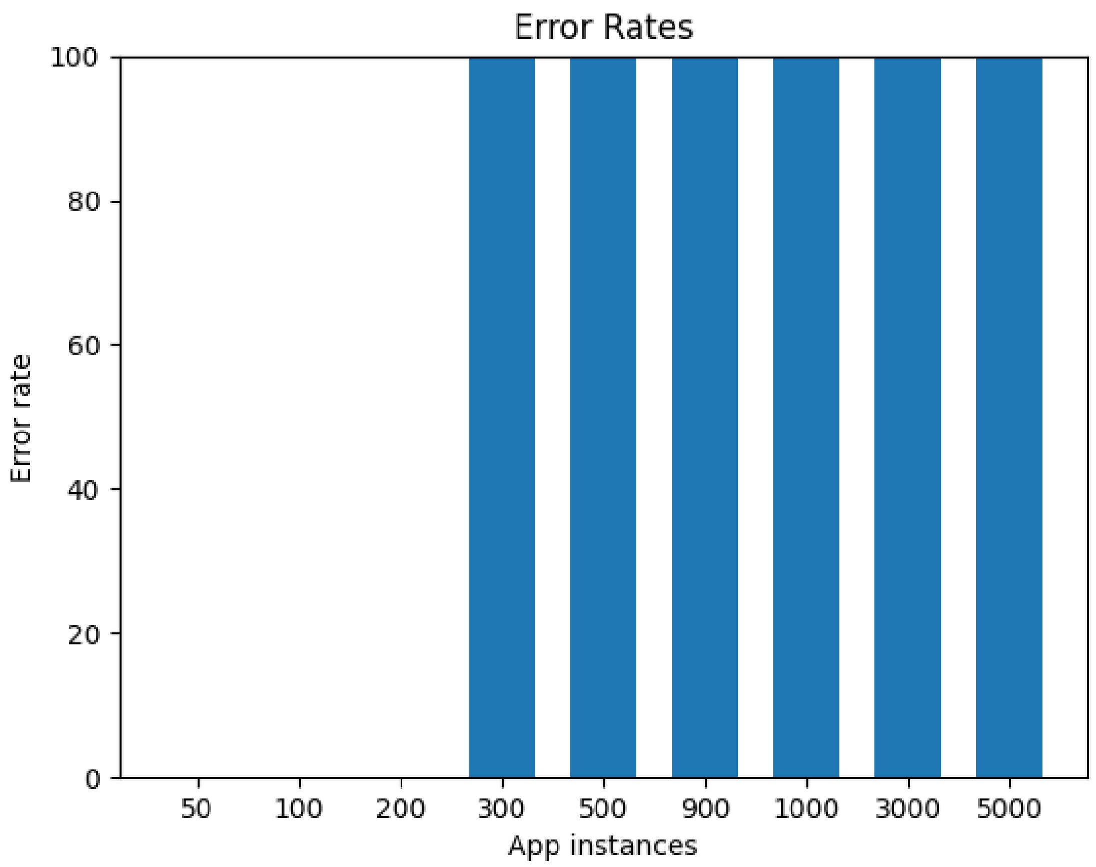


Figure 9-19 Scenario 5 error rate bar chart.

As shown in the charts, the system performance unfortunately did not improve.

9.6 Scenario 6: Connection Pool for Hyperledger Fabric Gateway

Focus: Determining if implementing a connection pool for the Hyperledger Fabric gateway can improve the latency and error rate.

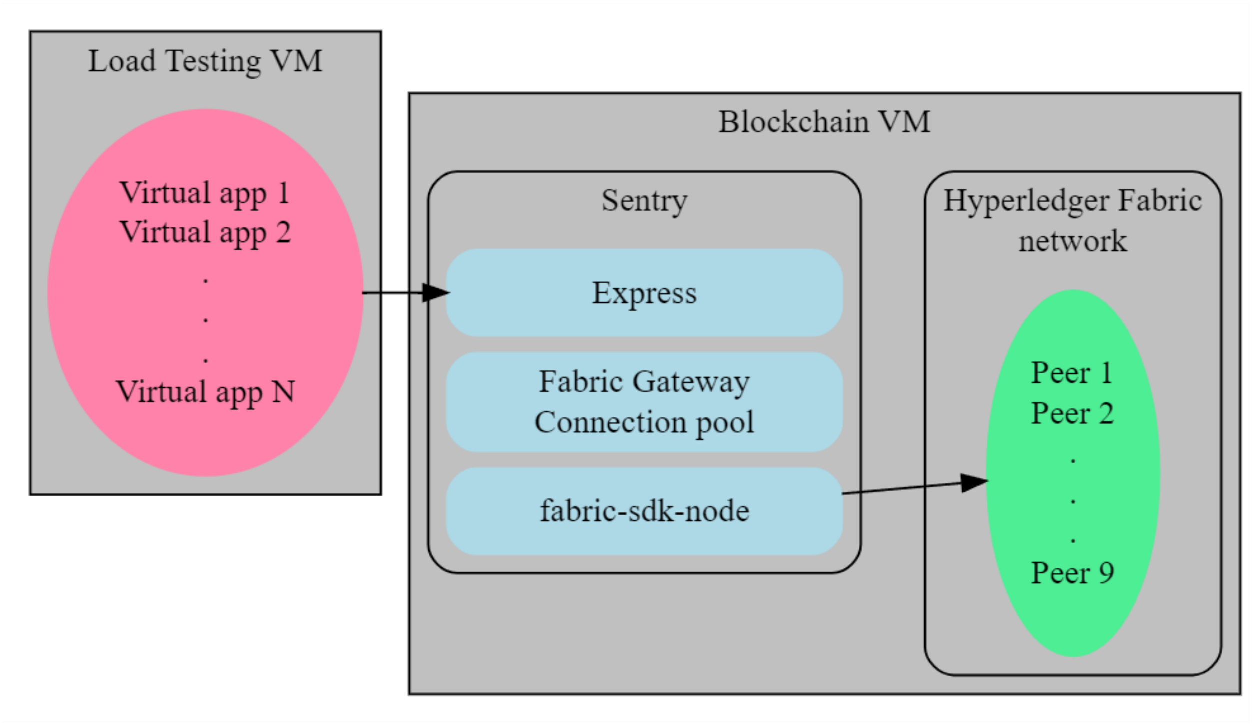


Figure 9-20 Scenario 6 overview.

After some debugging, it was discovered that the slowest part of the code was the snippet that handles the connection between the Express app and the Hyperledger Fabric network. Initially, when the Express app needs to get the access control data from the Hyperledger Fabric network, it creates a connection and closes it after the data has been retrieved. The step where the connection is created is time-consuming, which leads to re-creating a new connection for every request coming from the client app instances, slows down the overall performance. Hence, a connection pool was implemented. When the Express app receives a request from a client app containing a Hyperledger Fabric wallet, a connection is created using the wallet and saved to a pool. Then, the connection is re-used for every subsequent request of the same wallet. In this scenario, all client app instances shared the same connection object.

With the connection pool in place, after load-testing, the following scatter charts respectively show the latency of every request sent by one virtual client app instance and by ten virtual client app instances:

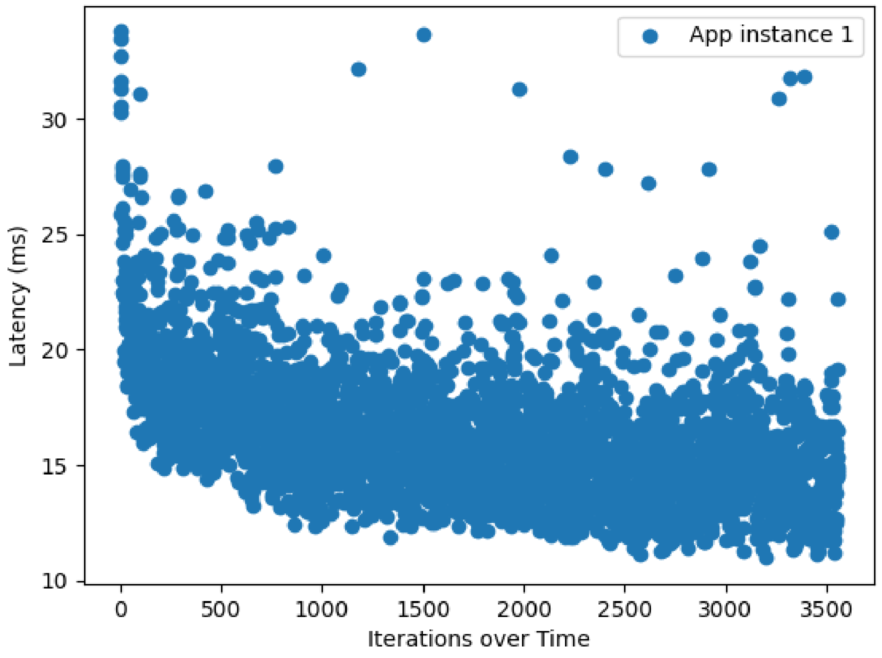


Figure 9-21 Scenario 6 latency scatter chart for one client app instance.

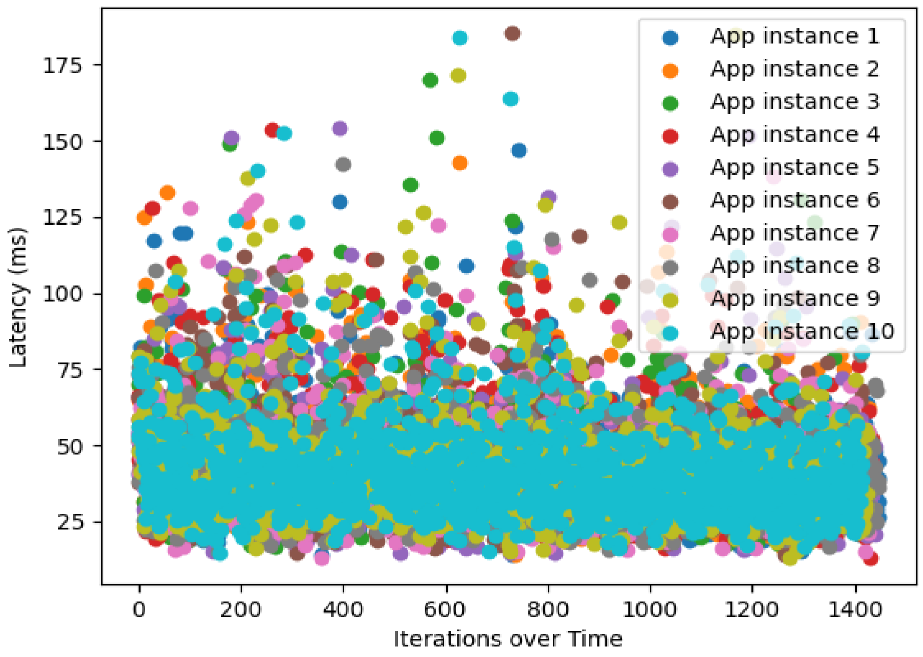


Figure 9-22 Scenario 6 latency scatter chart for ten client app instances.

The following bar plot shows the error rates derived from various sets of client app instances:

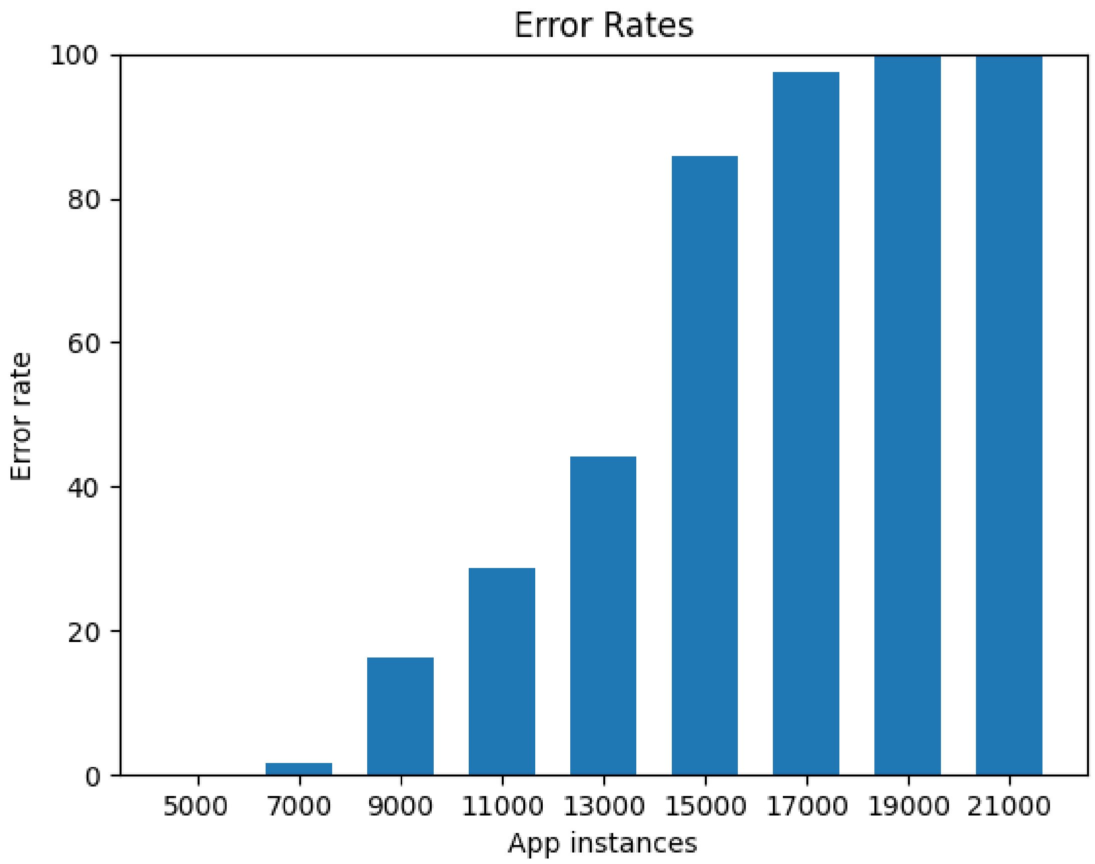


Figure 9-23 Scenario 6 error rate bar chart.

As shown in the charts, the latency significantly improved. At this point, the system could handle way more client app instances, up to 17000 instances, before the error rate hit 100% again, compared to only 300 instances in scenarios 4 and 5.

9.7 Summary

This appendix section presented multiple load-testing scenarios that helped identify the bottleneck in the proposed solution. It demonstrated how implementing a connection pool for the Hyperledger Fabric Gateway improved latency and enabled the system to handle a more substantial traffic load from a larger set of client apps. The connection pool for the Hyperledger Fabric Gateway proved to be a successful solution to the identified bottleneck, significantly enhancing system performance under heavy traffic loads.