D3.3 – Blockchain Traceability of Data

WP3 – BUILD: Manufacturing Data Quality

i4Q has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 958205.

Document Information

GRANT AGREEMENT NUMBER	958205	ACRONYM	i4Q	
FULL TITLE	Industrial Data Services for Quality Control in Smart Manufacturing			
START DATE	01-01-2021	DURATION	36 months	
PROJECT URL	https://www.i4q-project.eu/			
DELIVERABLE	D3.3 – Blockchain Traceability of Data			
WORK PACKAGE	WP3 – BUILD: Manufacturing Data Quality			
DATE OF DELIVERY	CONTRACTUAL	June 2022	ACTUAL	June 2022
NATURE	Report	DISSEMINATION LEVEL	Public	
LEAD BENEFICIARY	IBM			
RESPONSIBLE AUTHOR	Yoav Tock (IBM)			
CONTRIBUTIONS FROM				
TARGET AUDIENCE	1) i4Q Project partners; 2) industrial community; 3) other H2020 funded projects; 4) scientific community			
DELIVERABLE CONTEXT/ DEPENDENCIES	This document is D3.3. Its relationship to other documents is as follows: • D1.1 Project Vision Guide Document: vision of the i4Q^{BC} solution. • D1.4 Requirements Analysis and Functional Specification: requirements for the i4Q^{BC} solution. • D1.9 Requirements Analysis and Functional Specification v2: requirements for the i4Q^{BC} solution. • D2.7 i4Q Reference Architecture and Viewpoints Analysis v2: the role of the i4Q^{BC} solution in the reference architecture. • D3.11 i4Q Blockchain Traceability of Data v2: update of this deliverable			
EXTERNAL ANNEXES/ SUPPORTING DOCUMENTS	None			
READING NOTES	None			

ABSTRACT	The Blockchain Traceability of Data (i4Q^{BC}) solution aims to enhance the level of trust that different solutions and components can place on data. It provides services of immobility and finality of data, serving as the source of truth, enabling trust in data by providing the possibility for full provenance and audit trail of data stored. This deliverable explains the design choices for selection of Hyperledger Orion as the i4Q Blockchain Infrastructure. Hyperledger Orion is a centralised, trusted blockchain database that provides tamper-evidence, provenance, data lineage, authenticity, and non-repudiation through data centric Application Programming Interfaces. This document presents Hyperledger Orion and includes detailed deployment instructions. The code is available at the project's repository, so a blockchain database server is ready for use by the i4Q ecosystem.

Document History

VERSION	ISSUE DATE	STAGE	DESCRIPTION	CONTRIBUTOR
0.1	10-May-2022	ToC	Table of Contents available	IBM
1.1	27-May-2022	1st Draft	First Version Technical Deliverable	IBM
1.2	08-Jun-2022	Internal review	Review and comments	CERTH, UPV
1.3	22-Jun-2022	2nd Draft	Second Version Technical Deliverable after revision	IBM
1.4	30-Jun-2022	Final Doc	Final quality check and issue of final document	CERTH

Disclaimer

Copyright message

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS/ACRONYMS

ACL	Access Control List
AI	Artificial Intelligence
API	Application Programming Interface
BC	Blockchain
BCDB	Blockchain Database
BFT	Byzantine Fault Tolerance
CA	Certification Authority
DB	Database
DBMS	Database Management System
GA	Grant Agreement
GDPR	General Data Protection Regulation
HLF	Hyperledger Fabric
IPR	Intellectual Property Rights
JSON	JavaScript Object Notation
MVP	Minimum Viable Product
PoW	Proof of Work
RA	Reference Architecture
REST	Representational State Transfer
RO	Read Only (access)
RW	Read and Write (access)
SDK	Software Development Kit
TX	Transaction
UC	Use Case
WP	Work Package

Executive summary

The purpose of this document is to specify the blockchain platform as a first step towards making use of blockchain capabilities within the i4Q project. It encapsulates the initial activities and starting point of Task 3.2 – "Manufacturing Data Trustiness and Traceability". The content of this report takes into account the results of D2.7 [5], which is the result of multiple tasks from WP1 & WP2 as reported therein.

The Blockchain Traceability of Data (i4Q^{BC}) solution aims to enhance the level of trust that different solutions and components can place on data. Thus, it shall serve as one of the cornerstones of data storage services to be consumed by different solutions. This solution shall provide services of immobility and finality of data, serving as the source of truth, enabling trust in data by providing the possibility for full provenance and audit trail of data stored. Thus, the main functionality offered by this solution is comprised of Data trust traceability, enabling a full audit trail of assets and data.

This deliverable explains the design choices for selection of Hyperledger Orion instead of Hyperledger Fabric as the i4Q Blockchain Infrastructure. Hyperledger Orion is a centralised, trusted blockchain database that provides tamper-evidence, provenance, data lineage, authenticity, and non-repudiation through data centric Application Programming Interfaces, with transactional semantics and very simple well-known programming model. As this deliverable shows, it has the right fit to the project's requirements, and has advantages over the initially proposed Hyperledger Fabric blockchain platform. In addition, this document includes detailed deployment instructions, and the code is available at the project's repository, so a blockchain database server is ready for use by the i4Q ecosystem.

Document structure

Section 1: Contains a general description of the **i4Q^{BC}**, providing an overview and the list of features. It is addressed to final users of the i4Q Solution.

Section 2: Contains the technical specifications of the **i4Q^{BC}**, providing an overview and its architecture diagram. It is addressed to software developers.

Section 3: Details the implementation status of the **i4Q^{BC}**, explaining the current status, next steps and summarizing the implementation history.

Section 4: Provides the conclusions.

APPENDIX I: Provides the PDF version of the **i4Q Blockchain Traceability of Data** web documentation, which can be accessed online at: http://i4q.upv.es/3_i4Q_BC/index.html

1. General Description

1.1 Overview

Blockchain (BC) technology enables data traceability, which is a capability provided by i4Q^{BC}. Blockchain provides the underlying technology for ensuring immutability, finality, and provenance contributing to non-repudiation of the stored state. i4Q^{BC} provides primitives within or on top of the base blockchain infrastructure to provide these capabilities in a smart manufacturing environment, for storing and querying manufacturing related data, configuration changes decisions and AI accountability.

This section provides the background necessary to understand BC technologies, and introduces the main concepts and constructs, followed by the introduction of BCDB, which is the BC solution used in the i4Q project. Eventually, the section presents a comparison with Hyperledger Fabric, which was a candidate for the BC platform for the i4Q project.

1.2 Blockchain overview

1.2.1 Main concepts

A BC can be defined as an immutable ledger for recording transactions, representing a single source of truth for business interactions. The ledger is an append-only transactions log, where each transaction represents a state change. Transactions are grouped into blocks and a hash chain is built over the blocks, *i.e.*, each block stores also the hash of the previous block (see **Figure 1**).

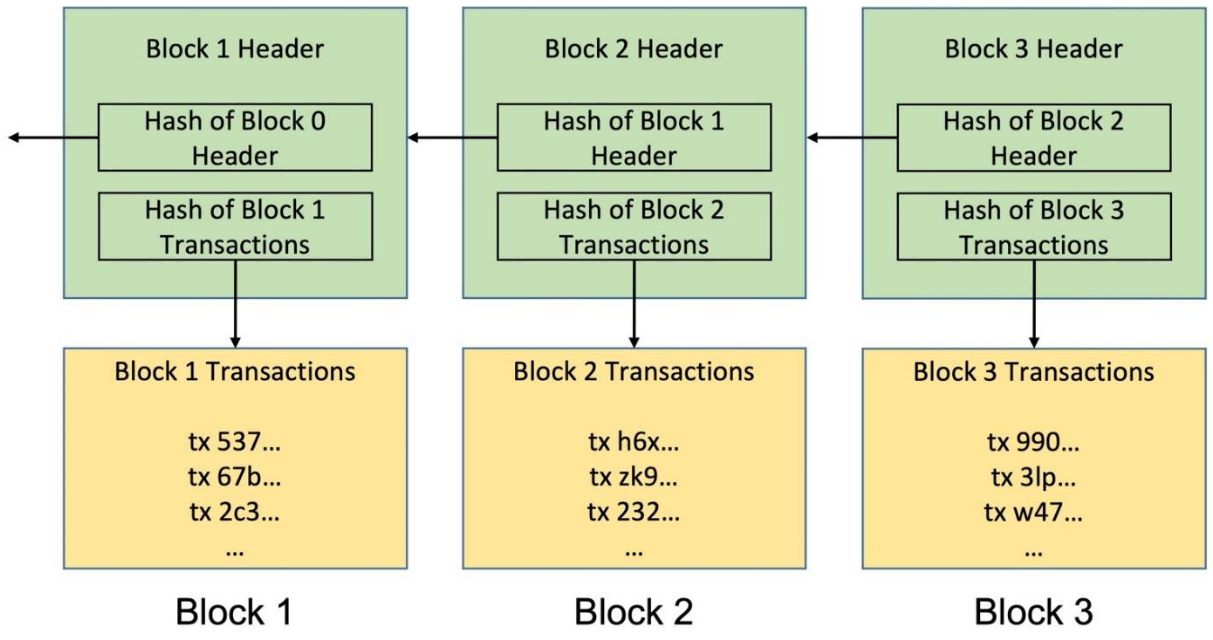


Figure 1. Data structure of a blockchain.

The BC structure is maintained in a way which ensures *provenance*, *immutability*, and *finality* of the transactions. Provenance is the capability to trace the source, and all the subsequent changes that were applied to an entity. Immutability means that a transaction that was recorded in the BC cannot be altered in any way (or at least without parties being able to identify an altered piece of information). Finally, the finality property ensures that once a transaction was recorded in the BC it cannot be removed.

1.2.2 Blockchain technologies

A BC can be maintained by a distributed network of mutually untrusting peers. Every peer maintains a copy of the ledger. The peers execute a consensus protocol to validate transactions, group them into blocks, and build a hash chain over the blocks. This process forms the ledger by ordering the transactions, as is necessary for consistency.

In a public or permissionless BC, such as Bitcoin, anyone can participate without a specific identity. Public BCs typically involve a native cryptocurrency and often use consensus based on "proof of work" (PoW) and economic incentives. Consensus refers to a fundamental problem in distributed computing, achieving overall system reliability in the presence of faulty or malicious processes. This often requires coordinating processes to agree on some data value that is needed during computation. Consensus protocols in BC technologies are a set of rules accepted by all participants, which translate into executable code, for block generation, and the verification, validation, and distribution of transactions to all parties.

Permissioned BCs, as opposed to permissionless ones, run a BC among a set of known, identified participants. A permissioned BC provides a way to secure the interactions among a group of entities that have a common goal, but which do not fully trust each other, such as businesses that exchange funds, goods, or information. By relying on the identities of the peers, a permissioned BC can use traditional Byzantine Fault Tolerant (BFT) consensus.

BCs may execute arbitrary, programmable transaction logic in the form of smart contracts, as exemplified by Ethereum. They are clauses and conditions that can be implemented by translating the business logic into code operating on BC data. Smart contracts are deployed within the BC network so that their content cannot be modified. Scripts in Bitcoin were a predecessor of the concept. A smart contract functions as a trusted distributed application and gains its security from the BC and the underlying consensus among the peers.

Cryptographic tools

A set of cryptographic tools, such as encryption, signatures, and access control technique, are utilised by BC technologies to allow transactions to only be registered by authorised parties.

Hash functions: are functions that allow an entity to encrypt any input data, generating a hash, which is encrypted text of a fixed length. It has several properties:

- Obtaining the original data from the generated hash is practically impossible.
- The same output hash is always obtained for a specific input.
- Any modification of the input data generates with a high probability a completely different hash.

These characteristics allow the verification of the integrity of a message and guarantee that it has not been tampered with.

Asymmetric Cryptography: also known as public key cryptography. It allows secure communication between two parties through the use of private and public keys. The private key is kept secret by the owner of the key pair and the public key is visible to all parties. A message encrypted with the private key can only be decrypted with the public key and vice versa.

Digital signature: a combination of the two previous methods, consists of applying a hash function on the data set that is to be exchanged, and encrypting the resulting hash with a private key. A signature is obtained and attached to the data submission.

1.3 Related work: Hyperledger Fabric vs. Blockchain DB

1.3.1 Introduction to Hyperledger Fabric

Hyperledger Fabric (HLF) is an open-source project of a permissioned BC infrastructure with modular architecture, allowing managing consensus and trust among different entities [1].
At a high level, the system is comprised of *peer* servers, potentially belonging to different organisations, which replicate and validate the blocks creating the transactions comprising the ledger; an *ordering service* which determines the total order of the transactions and publishes the corresponding blocks to be picked up by the peer processes; and a *client* that interacts with the system programmatically for invoking transactions or queries. A configurable sub-set of the peers is involved also in *endorsing transactions* submitted to the system; supporting consensus for inserted transactions. All entities hold verifiable security certificates issued by a Certification Authority (CA) component.

Underlying a HLF network (as in most permissioned networks) is the notion of a consortium, which is a group of organisations that agreed to set up a BC network between them, establishing the governance body and rules. Most central organisations within a BC network will deploy (or use) a CA on their behalf, and will contribute peer(s), which are server components that endorse, validate, and hold replicas of the shared ledger. In addition, an ordering service needs to be set up by the organisations, to order the transactions, cut blocks and make them available to the peers.

To bootstrap a network, there needs to be an ordering service, peer processes need to be established on behalf of organisations, and the appropriate cryptographic material should be distributed (including the certificates required to participate in the network) to each entity. Once the backbone is in place, chaincodes (smart contracts in HLF) can be installed and instantiated. A chaincode needs to be installed on each peer, which may endorse transactions for that chaincode (endorsing peers are the only ones that execute the chaincode). The chaincode needs to be instantiated on one of the peers, to create the bond between the chaincode and the channel, and run the initialisation method specific to that chaincode.

Once the chaincode is in place, users can start invoking transactions and queries on the BC channel. By using a client, the user assembles a transaction and sends it to the endorsing peers. Endorsing peers' policy is determined per chaincode, and establishes the identity of potential endorsers and the conditions that must be satisfied for a transaction to be approved. Once the client has received responses from the endorsing peers, it can evaluate whether the transaction abides by the endorsement policy and can thus go through and be sent to the ordering service to be included in a future block, or whether it needs to be dropped. Endorsing peers are the only ones that actually run the chaincode (in a simulating mode), and return the corresponding read and write sets of the transaction, namely the keys and versions of variables that were read or written by the simulated execution of the chaincode. Endorsed transactions are then sent, along with the corresponding read and write set, to the ordering service. The ordering service in turn orders incoming transactions, cuts blocks, and makes the blocks available to the peers. The peers in turn obtain a new block, validate the transactions in it, and apply the write set for the transactions that have been determined to be valid.

1.3.2 Introduction to Blockchain Database

BCDB, by Gartner's definition, is a ledger-based Database Management System (DBMS) implementation. Ledger DBMSs, also known as BCDBs, is a spectrum of work which takes any well-known DB technology and then extends it by adding BC properties to it. The main goal of BCDB is to increase the adoption rate by providing immutable ledger features as an extension to existing DB technology and simplify the overall development, deployment, and usage. Further, these BCDBs leverage the rich features and transactional processing capability already built into DBs over decades of research and development, and enable easy integration with the legacy system. The millions of developers having experience in DB technology can easily build BC applications with ledger DBMSs. By 2021, Gartner's estimation is that most permissioned BC platforms will be replaced by ledger DBMS products [2].

BCDB is a key-value/document replicated DB that provides immutable ledger properties such as tamper evident, non-repudiation, and provenance queries. In addition, BCDB provides DBMS features such as serialisation isolation level, crypto-based authentication, confidentiality and access control, and high availability using replication. However, BCDB does not support the decentralisation of trust, as it is a centralised DB, nor smart contracts, that can be simulated by special transactions. Therefore, BCDB is not a Decentralised Distributed Ledger but rather a Ledger DBMS.

BCDB is an open-source project maintained by IBM[1]. It was recently accepted to be part of the Hyperledger open-source community[2], that is under the Linux Foundation[3], and the new BCDB project's name has now changed to *Hyperledger Orion* (although still referred to as BCDB throughout this document).

1.3.3 A blockchain platform for i4Q

In the i4Q ecosystem one needs to be granted permissions in order to access the platform, and everyone is required to specify their true identity. However, this does not contradict the privacy and anonymisation also required by i4Q. Hence, the permissioned setting is suitable for the i4Q scenario.

While different i4Q users do not necessarily have to trust each other, the i4Q^{BC} as an entity is in fact trusted by all platform users. Therefore, the i4Q ecosystem can operate under the centralised trust assumption.

[1] Blockchain DB: https://github.com/hyperledger-labs/orion-server
[2] Hyperledger: https://www.hyperledger.org/
[3] The Linux Foundation: https://www.linuxfoundation.org/

BC-based technology can help improve data trust in centralised ecosystems. While efficient classical DBs do not provide out of the box capabilities needed to guarantee and prove required trust features, and the use of decentralised (BC-based) ledger technologies for centralised systems is difficult, inefficient, and costly, an extension of classical DB with BC-based features can be used to efficiently address trust requirements in such systems.

The original permissioned BC platform considered for i4Q project was Hyperledger Fabric. However, after careful consideration, discussing alternative approaches with all relevant i4Q partners, and a better understanding of the different use case BC-specific requirements, we arrived at the conclusion that the BCDB technology is most suitable for functioning as the project's BC platform. Both are open-source projects, maintained mostly by IBM, and part of the Hyperledger open-source community. Nevertheless, BCDB has a few key benefits over Hyperledger Fabric that could serve the i4Q project.

Foremost is that BCDB is a centralised trusted ledger, as opposed to the Hyperledger Fabric decentralised trust. BCDB is specifically designed to efficiently address ecosystems where there is a trusted party such as a cloud provider, government or legal bodies, or any other key organisation, while the other participants in the ecosystem do not have to trust each other.

In a centralised trust eco-system, there is no need to create a consortium to maintain a decentralised network like Hyperledger Fabric, but rather i4Q^{BC} will maintain a BCDB, which significantly reduces associated operational and maintenance costs.

Moreover, BCDB has more non-functional benefits such as easy installation and administration, standard programming model, and significantly higher efficiency. As an example, the different transaction execution flows are illustrated in the diagram (**Figure 2**) below. While in Hyperledger Fabric, transaction execution involves many different processes and a few rounds of communication, BCDB offers a straightforward approach.

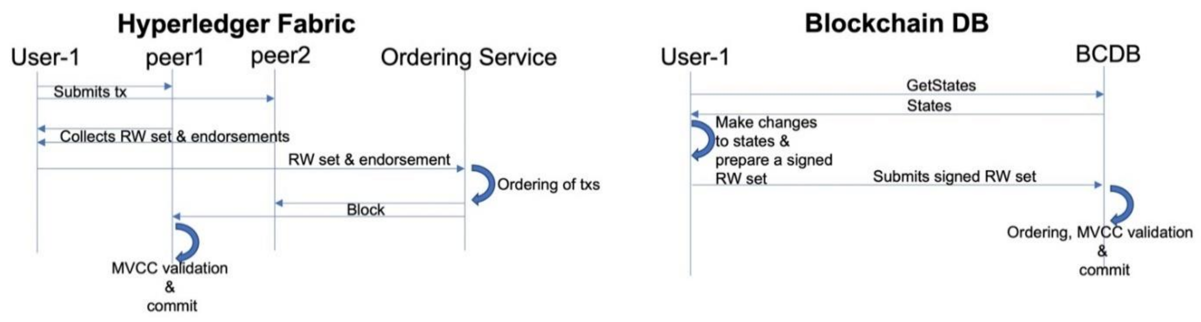


Figure 2. Transaction execution flow of Hyperledger Fabric and Blockchain DB.

1.4 Features

The main features provided by this solution include:

- A replicated key-value database with transactional APIs. A data centric API provides a familiar and easy to use programming paradigm, which reduces complexity and cost compared to traditional blockchain platforms. A cluster of servers provide a fault tolerant & highly available solution.
- All stored data is bound with cryptographic data structures (e.g.: a hash-chain), which ensures immutability and tamper evidence.
- The use of digital signatures by clients and servers, for both requests, responses, and stored data, provides authentication as well as non-repudiation.
- Fine-grain privacy is achieved using key-level access control, which is enforced by the digital signatures provided by users.
- Multi-party transactions, in which more than one user is required to sign a transaction to make it valid, are supported.
- All historical changes to the data are maintained within the ledger as well as in a graph structure. The solution allows the user to execute provenance queries on those historical changes to understand the lineage of each data item.

2. Technical Specifications

In this section we describe the BCDB, its properties and trust assumptions, the high-level architecture, and detail the APIs. Finally, this chapter includes the deployment instructions for a BCDB server. More details are given on the BCDB project's website (https://github.com/hyperledger-labs/orion-server), and as previously mentioned, the project is now called Hyperledger Orion. As discussed in previous chapters, BCDB will serve as the BC platform of the i4Q project.

2.1 Blockchain properties

BCDB is a **key-value/document DB** with certain BC properties such as:

- **Tamper Evident**: Data cannot be tampered with, without it going unnoticed. At any point in time, a user can request the DB to provide proof for an existence of a transaction or data and verify the same to ensure data integrity.
- **Non-Repudiation**: A user who submitted a transaction to make changes to data cannot deny submitting the transaction later.
- **Crypto-based Authentication**: A user that submitted a query or transaction is always authenticated using digital signature.
- **Confidentiality and Access Control**: Each data item can have an Access Control List (ACL) to dictate which users can read from it and which users can write to it. Each user needs to authenticate themselves by providing their digital signature to read or write to data. Depending on the access rule defined for data, sometimes more than one user needs to authenticate themselves together to read or write to data.
- **Serialisation Isolation Level**: It ensures a safe and consistent transaction execution.
- **Provenance Queries**: All historical changes to the data are maintained separately in a persisted graph data structure so that a user can execute query on those historical changes to understand the lineage of each data item.

BCDB **DOES NOT** have the following two BC properties:

- **Smart-Contracts**: A set of functions that manage data on the BC ledger. Transactions are invocations of one or more smart contract's functions.
- **Decentralisation of Trust**: A permissioned setup of known but untrusted organisations, each operating their own independent DB nodes, but communicate together to form a BC network. As one node cannot trust the execution results of another node, ordering transaction must be done with a BFT protocol and all transactions need to be independently executed on all nodes.

2.2 High level architecture

Figure 3 presents the high-level architecture of BCDB. BCDB stores and manages the following five data elements:

1. **Users**: Storage of users' credentials such as digital certificate and their privileges. Only these users can access the DB.

2. **Key-Value Pairs**: Storage of all current/active key-value pairs committed by users of the DB.
3. **Historical Key-Value Pairs:** Storage of all past/inactive key-value pairs using a graph data structure with additional metadata such as the user who modified the key-value pair, all previous and next values of the key, transactions which have read or written to the key-value pair, *etc*. It helps to provide a complete data lineage.
4. **Authenticated Data Structure**: Storage of Merkle Patricia Tree[4], where leaf node is nothing but a key-value pair. It helps in creating proofs for the existence of a key-value pair.
5. **Hash chain of blocks**: Storage of cryptographically linked blocks, where each block holds a set of transactions submitted by the user along with its commit status, summary of state changes in the form of Merkle Patricia's Root hash, *etc*. It helps in creating a proof for the existence of a block or a transaction.

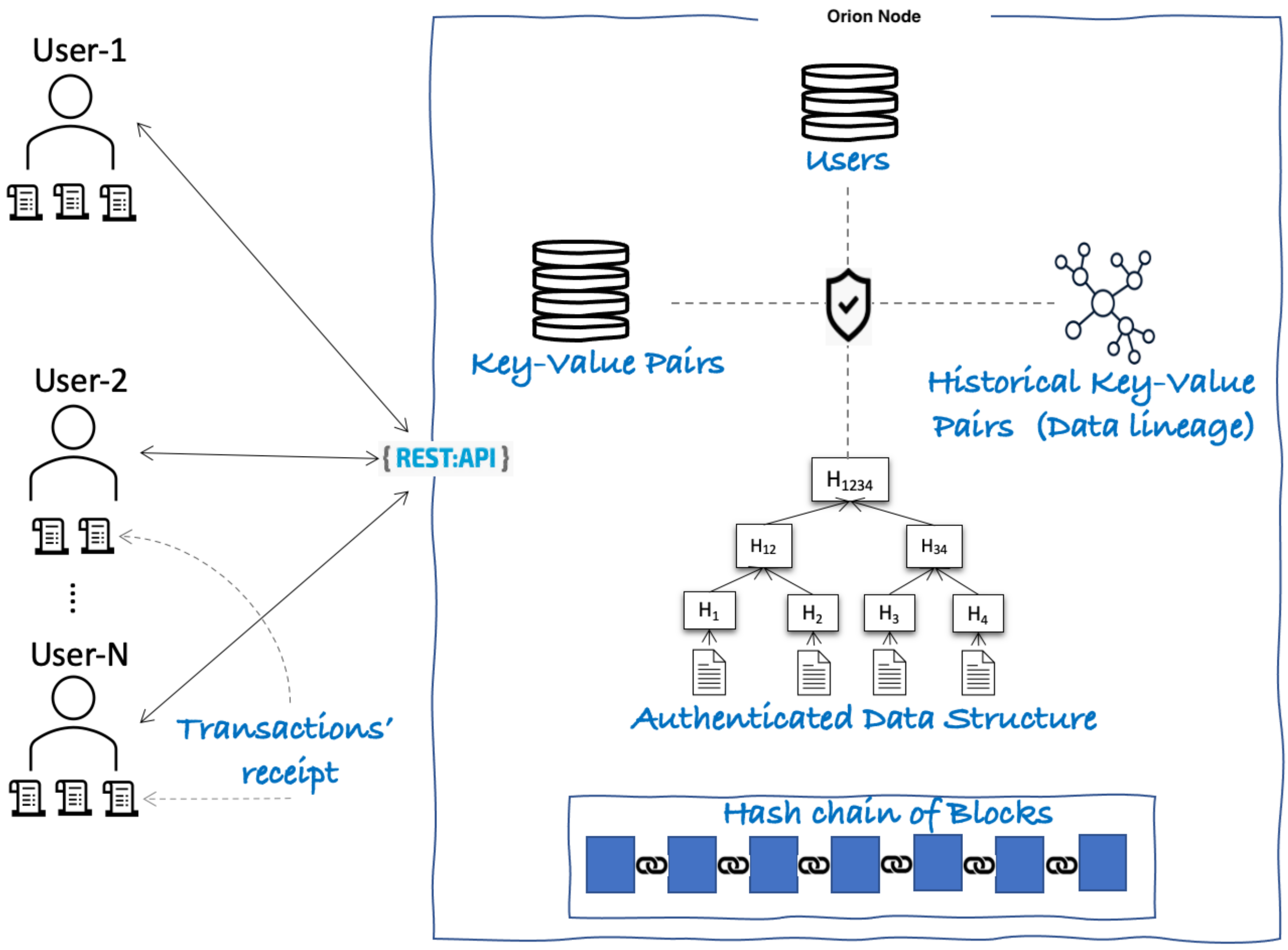


Figure 3. High level architecture of BCDB (Source: https://github.com/hyperledger-labs/orion-server).

The users of the DB can query these five data elements provided that they have the required privileges and also can perform transactions to modify active key-value pairs. When a user submits a transaction, that user receives a transaction receipt from the DB after the commit of a block that

[4] Merkle Patricia Tree: https://eth.wiki/fundamentals/patricia-tree

includes the transaction. The user can then store the receipt locally for performing client-side verification of proof of existence of a key-value pair or a transaction or a block.

2.3 APIs

REST APIs are supported to enable users to interact with the BCDB node. Every user who issues a GET or POST request must be authenticated cryptographically through digital signature. To add a digital signature on the request payload, a utility called signer is provided.

Note that a Software Development Kit (SDK) built using Golang (GO programming language) is provided, which simplifies the development by hiding much of the details such as expected data format of the request payload, signing the query and transaction, verifying the digital signature of the DB node on the response, etc. However, one can choose to use the signer utility along with the cURL to perform queries and submit transactions.

2.3.1 Queries

For each GET request, one needs to set two custom headers called UserID and Signature which will be included in the request while sending HTTP to the BCDB node.

As every user who issues a GET or POST request must be authenticated cryptographically through digital signature, the submitting user must compute a signature on the query data or transaction data and set the signature in the Signature header. The signer utility is used to compute the required digital signature.

Queries can be used to fetch data stored/managed by the DB.

2.3.1.1 Querying the cluster configuration

The cluster configuration includes node, admin, and consensus configuration (used for replication). When the BCDB server boots up for the first time, it reads nodes, admins, and consensus configuration present in the "config.yml" configuration file, and creates a genesis block. The user can query the current cluster configuration by issuing a `GET` request on the `/config/tx` endpoint.

The REST endpoint for querying the configuration is `/config/tx`, and it does not require any inputs or additional parameters from the user. Hence, the user needs to sign only their user id and set the signature in the Signature header.

Specifically, the user needs to sign the following JavaScript Object Notation (JSON) data:

```
{"user_id":"<userID>"}
```

where `<userID>` is the ID of the submitting user, who is registered in the BCDB node.

In the following example, admin user is the one who submits a request to the server. Hence, the admin's private key is used to sign the `{"user_id":"admin"}`, as shown below:

```
./bin/signer    -privatekey=deployment/sample/crypto/admin/admin.key    -
data='{"user_i
d":"admin"}'
```

The above command would produce a digital signature and prints it as base64 encoded string as shown below:

```
MEUCIQCMEdLgfFEOF+vgXLwbeOdUUWnGB5HH2ULkoz15jlk5DgIgbWXuoyqD4szob78hZYiau
9LPdJLLqP3bAu7iV98BcW0=
```

Once the signature is computed, a `GET` request can be issued using the following cURL command by setting the above signature string in the Signature header:

```
curl \
   -H "Content-Type: application/json" \
   -H "UserID: admin" \
   -H                                                           "Signature:
MEUCIQCMEdLgfFEOF+vgXLwbeOdUUWnGB5HH2ULkoz15jlk5DgIgbWXuoyqD4sz
ob78hZYiau9LPdJLLqP3bAu7iV98BcW0=" \
   -X GET http://127.0.0.1:6001/config/tx | jq .
```

2.3.1.2 Querying the user information

One user can query information about another user or themselves. If the access control is defined for the user entry, it would be enforced during the query. A user information can be retrieved by issuing a GET request on the /user/{userid} endpoint, where {userid} should be replaced with the ID of the user whom information needs to be fetched.

Here, the submitting user needs to sign:

{"user_id":"<submitting_user_id>","target_user_id":"<target_user_id"}

where <submitting_user_id> denotes the ID of the user who is submitting the query, and <target_user_id> denotes the ID of the user of whom information needs to be fetched.

2.3.1.3 Checking the database existence

To check whether a DB exists/has been created, the user can issue a `GET` request on the `/db/{dbname}` endpoint, where `{dbname}` should be replaced with the DB name for which the user needs to perform this check.

For this query, the submitting user needs to sign:

```
{"user_id":"<userid>","db_name":"<dbname>"}
```

where `<userid>` denotes the submitting user and the `<dbname>` denotes the name of the DB for which the user performs the existence check.

2.3.1.4 Querying a block header

A block is a collection of ordered transactions in BCDB. The header object within a block holds the block number, root hash of the transaction Merkle tree, root hash of the state Merkle tree, and validation information.

To query a block header of a given block, the user can issue a GET request on the /ledger/block/{blocknumber} endpoint, where {blocknumber} denotes the block whose header needs to be fetched.

The submitting user needs to sign:

{"user_id":"<userid>","block_number":<blocknumber>}

where the <userid> denotes the id of the user who is submitting the query, and <blocknumber> denotes the number of the block whose header needs to be fetched.

2.3.1.5 Provenance queries

The provenance API gives the user access to the following BCDB data:

- The history of values for a given key, in different views and directions;
- Information about which users accessed or modified a specific piece of data;
- Information, including history, about the data items accessed by a given user;
- A history of user's transactions.

Usually, provenance queries are used to investigate changes of some values over time. For example, by sending `GET /provenance/data/history/{dbname}/{key}`, changes of a specific key over time can be followed. As mentioned above, BCDB supports multiple types of provenance queries.

2.3.2 Transactions

2.3.2.1 Database administration transaction

To create or delete a DB, one needs to submit a DB administration transaction.

When the DB node boots up for the first time, it would create a default DB called `bdb` and 3 system DBs named `_users`, `_dbs`, and `_config`. The `bdb` DB can be used to submit data transactions, whereas system DBs are internal to the BCDB server. The user cannot directly read or write to the system DBs.

To create or delete user DBs, the user needs to issue a:

```
POST /db/tx {txPayload}
```

where `txPayload` contains information about the DB to be created and deleted.

In queries, the `UserID` and `Signature` headers had to be set. Whereas in the transaction, both the `UserID` and `Signature` need to be passed as part of the `txPayload` itself.

One may create a new DB to store data/states by issuing a DB administration transaction. Note that the DB to be created should not exist in the node. Otherwise, the transaction would fail.

The following cURL command submits a DB administration transaction to create two new DBs named `db1` and `db2`:

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/db/tx \
   --data '{
    "payload": {
            "user_id": "admin",
            "tx_id": "1b6d6414-9b58-45d0-9723-1f31712add71",
            "create_dbs": [
                  "db1",
                  "db2"
            ]
         },
  "signature":
"MEUCIQDidxd5ScjpfYTIfVmSfC874zO0iosSyQUzRprs8j7VXgIgR7QxISwdjgXX5
8TktYXobJHwbCC3F/14rxCg0F8Ma1w="
}'
```

The payload of the DB administration transaction must contain a "user_id", who submits the transaction, "tx_id" to uniquely identify this transaction, and a list of DBs to be created in a "create_dbs" list, as shown in the above cURL command.

As all administrative transactions must be submitted only by the admin, the "user_id" is set to "admin". As there are two DBs being created, named db1 and db2, the "create_dbs" is set to ["db1","db2"]. Finally, the signature field contains the admin's signature on the payload, and is computed using the following command:

```
./bin/signer    -privatekey=deployment/sample/crypto/admin/admin.key    -
data='{"user_i
d":"admin","tx_id":"1b6d6414-9b58-45d0-9723-
1f31712add71","create_dbs":["db1","db2"]}'
```

The output of the above command is set to the signature field in the data.

Once the DB creation transaction gets validated and committed, it would return a receipt to the transaction submitter. Note that only if the TxTimeout header is set, the submitting user would receive the transaction receipt. This is because if the TxTimeout is not set, the transaction would be submitted asynchronously and the DB node returns as soon as it accepts the transaction into the queue. If the TxTimeout is set, the DB node waits for the specified time. If the transaction is committed by the specified time, the receipt would be returned.

One can delete an existing DB by issuing a DB administration transaction. Note that the DB to be deleted should exist in the node. Otherwise, the transaction would be marked invalid.

The following curl command can be used to delete two existing DBs named db1 and db2:

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/db/tx \
   --data '{
    "payload": {
            "user_id": "admin",
            "tx_id": "5c6d6414-3258-45d0-6923-2g31712add82",
            "delete_dbs": [
                  "db1",
                  "db2"
            ]
         },
  "signature":
"MEYCIQDC3t4gX4rAXmzqM8359u751vueqaSmYvBEXpCXdafeKAIhAKitFv8r89Rrr
uAlABhjcgeJPIPTEpkcc3tOZ77YmypV"
}'
```

The payload of the DB administration transaction must contain a `"user_id"`, who submits the transaction, `"tx_id"` to uniquely identify this transaction, and a list of DBs to be deleted in a `"delete_dbs"` list, as shown in the above cURL command.

As all administrative transactions must be submitted only by the admin, the `"user_id"` is set to `"admin"`. As two existing DBs are being deleted, named `db1` and `db2`, the `"delete_dbs"` is set to `["db1","db2"]`.

Within a single transaction, one can create and delete many DBs. The following command submits a transaction that creates and deletes DBs within a single transaction. This transaction will be valid only if `db3` and `db4` do not exist, and `db1` and `db2` do exist in the cluster.

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/db/tx \
   --data '{
    "payload": {
        "user_id": "admin",
        "tx_id": "1b6d6414-9b58-12d5-3733-1f31712add88",
        "create_dbs": [
            "db3",
            "db4"
        ],
        "delete_dbs": [
            "db1",
            "db2"
        ]
    },
  "signature":
"MEUCIAjEtDZ2Q6n6cteisp94ggFXk3JUOXCjhfUlftc80gf6AiEA6IPtezn06SaPW
QLfGhbx8BrFL4BI4iEIu/TDGtcaCKI="
}'
```

The signature is computed using the following command:

```
./bin/signer    -privatekey=deployment/sample/crypto/admin/admin.key    -
data='{"user_i
d":"admin","tx_id":"1b6d6414-9b58-12d5-3733-
1f31712add88","create_dbs":["db3","db4"],"delete_dbs":["db1","db2"]}'
```

2.3.2.2 *User administration transaction*

It is possible to create, update and delete users of the DB cluster using the user administration transaction, by issuing a `POST /user/tx {txPayload}`. Note that all user administration transactions must be submitted by the admin.

Adding a user

When the cluster is started for the first time, it will contain only the `admin` user specified in "config.yml". This `admin` user can add any other user to the cluster. In the below example, the `admin` user is adding two users named `alice` and `bob`, with certain privileges.

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 10s" \
   -X POST http://127.0.0.1:6001/user/tx \
   --data '{
          "payload": {
           "user_id": "admin",
           "tx_id": "7b6d6414-9b58-45d0-9723-1f31712add01",
           "user_writes": [
                 {
                       "user": {
                             "id": "alice",
                             "certificate":
"MIIBsjCCAVigAwIBAgIRAJp7i/UhOnaawHTSd
kzxR1QwCgYIKoZIzj0EAwIwHjEcMBoGA1UEAxMTQ2FyIHJlZ2lzdHJ5IFJvb3RDQTAeFw0yMT
A2MTYxMTEzMjdaFw0yMjA2MTYxMTE4MjdaMCQxIjAgBgNVBAMTGUNhciByZWdpc3RyeSBDbGl
lbnQgYWxpY2UwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAASdCmAgHdqck7uhAK5siEF/O1EI
UEIYtiR3XVEjbVhNe/6GXFShtsSThXYL9/XK6p4qF4oSy9j/PURMGnWbzSnso3EwbzAOBgNVH
Q8BAf8EBAMCBaAwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMAwGA1UdEwEB/wQCMA
AwHwYDVR0jBBgwFoAU7nVzp7gto++BPlj5KAF1IA62TNEwDwYDVR0RBAgwBocEfwAAATAKBgg
qhkjOPQQDAgNIADBFAiEAsRZlR4sDyxS//BJnYpC684EWu1hO/JU8rkNW6Nn0FFQCIH/p6m6E
LkLNQpx+1QJsWWtH/LdW94WinVylhuA4jggQ",
                             "privilege": {
                                   "db_permission": {
                                     "db1": 0,
                                     "db2": 1
                                   }
                             }
                       }
                 },
                 {
                       "user": {
                             "id": "bob",
                             "certificate":
"MIIBrzCCAVWgAwIBAgIQZOQpmvY31R8yeyy3C
lrJtzAKBggqhkjOPQQDAjAeMRwwGgYDVQQDExNDYXIgcmVnaXN0cnkgUm9vdENBMB4XDTIxMD
YxNjExMTMyN1oXDTIyMDYxNjExMTgyN1owIjEgMB4GA1UEAxMXQ2FyIHJlZ2lzdHJ5IENsaWV
udCBib2IwWTATBgcqhkjOPQIBBggqhkjOPQMBBwNCAASUDaIwGvRPPHHMzw4UFPTX5BTuPons
8Xv3AR6k/8dDJQsn09qdtKWauLLLGxiLNDY2J8S0qPzJhJVPGF6h/l9Uo3EwbzAOBgNVHQ8BA
f8EBAMCBaAwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMAwGA1UdEwEB/wQCMAAwHw
YDVR0jBBgwFoAU7nVzp7gto++BPlj5KAF1IA62TNEwDwYDVR0RBAgwBocEfwAAATAKBggqhkj
OPQQDAgNIADBFAiAFxiyZgtiTwvMFF6jKtUE5vV0YzthpWmdRiUIbUclKzQIhALQolKPJl9xm
v66wOyJTvR2q13Fb6j75M4WGcG4KfjDZ",
                             "privilege": {
                                   "db_permission": {
                                         "db1": 0,
                                         "db2": 0
                                   }
                             }
                       },
                       "acl": {
                             "read_users": {
                                   "admin": true
                             }
                       }
                 }
           ]
     },
     "signature":
"MEUCIHLCSwMzwxmnRfB6s1eON2bMfgDwFvxoSqaZ6ACXcbn0AiEA8KhjY56tSRg
9Hh9UGchhGybTV2rWl1NcsAPLyW71Vu8="
}'
```

The user `alice` has read only access on the `db1` DB and read-write access on the `db2` DB. These privileges are defined under `db_permission`. `"db1":0` denotes that the user has read-only privilege on `db1`, while `"db2":1` denotes that the user has read-write privilege on `db2`. In other words, 0 denotes read-only privilege and 1 denotes read-write privilege. As the access control is not defined for the user, any user can read the credential and privilege of `alice`, but only the `admin` user can modify the properties of the `alice` user.

The `bob` user has read-only privilege on `db1` and `db2`. Further, only the `admin` user can read the credential and privilege of `bob`.

Moreover, the `bob` user cannot be modified as the `"read_write_users"` section is left out empty. This means no user has permission to write to user `bob`.

Updating a user

In order to remove all privileges given to `alice`, the following steps can be executed:

1. Fetch the current committed information of the `alice` user.
2. Remove the privilege section, and construct the transaction payload.
3. Submit the transaction payload by issuing a `POST /user/tx {txPayload}`.

Note that the `user_reads` contains the version of the read information. If this does not match the committed version, the transaction would be invalidated. This is useful, because the `alice` user can be updated by any other `admin` between step 1 and 2 listed above. To ensure serialisability isolation, the read version must be passed. To be more specific, the version in the metadata section of the query result is available as shown below:

```
  "metadata": {
      "version": {
        "block_num": 4
      }
    }
```

The version is used to perform multi-version concurrency control to ensure serialisability isolation level.

The `user_reads` section says that commit this transaction only if users specified in the `user_reads` list are at a specified version as per the current committed state. Otherwise, invalidate the transaction and do not apply the changes requested by the transaction.

Note that it is not necessary to pass the version in `user_reads`. If the `user_reads` is left out, the write would be considered as a blind write.

Deleting a user

In order to delete `alice` from the cluster, the following steps can be executed:

1. Fetch the current committed information of the `alice` user, to get the committed version.
2. Add the committed version to the `user_reads`.
3. Add the `alice` user to the `user_deletes`.
4. Submit the transaction payload by issuing a `POST /user/tx {txPayload}`.

The example uses `"block_num": 5` as the version. While executing this example, the user `alice` should be queried, and the version provided in the output of the query should be used. Note that it is not necessary to pass the version in `user_reads`. If the `user_reads` is left out, the delete would be considered as a blind delete. For blind delete, steps 1 and 2 are not needed.

Within a single transaction, one can delete more than a single user. The section `user_deletes` is an array, and can be passed many users, all of whom will be deleted, if they exist. For simplicity, the example deletes a single user only.

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/user/tx \
   --data '{
          "payload": {
             "user_id": "admin",
             "tx_id": "1b6d6414-9b58-45d0-9723-1f31712add04",
        "user_reads": [
                    {
                          "user_id": "alice",
                          "version": {
                                "block_num": 5
                          }
                    }
             ],
             "user_deletes": [
                    {
                          "user_id": "alice"
                    }
             ]
          },
    "signature":
"MEUCIFDGfF7deiAexylyN/C1DINgz5TA5CbIB/w+AnjhZYYTAiEAvGtgFWWtWWQ
aGr4EWo4whcs/+pHhgYMyYeFPha8YRhg="
}'
```

Note that within a single transaction, one can do multiple operations such as adding multiple new users, updating and deleting multiple existing users.

2.3.2.3 *Data Transaction*

Data transactions are issued to store, update, and delete any state, *i.e.*, key-value pair on the ledger. By submitting a `POST /data/tx {txPaylod}`, one can perform a data transaction, where `txPayload` contains reads, writes, and deletes of states.

Storing a new state

Storing a new state with the `key1` key.

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/data/tx \
   --data '{
         "payload": {
            "must_sign_user_ids": [
            "alice"
        ],
            "tx_id": "1b6d6414-9b58-45d0-9723-1f31712add81",
        "db_operations": [
            {
                "db_name": "db2",
                    "data_writes": [
                          {
                        "key": "key1",
                        "value": "eXl5",
                        "acl": {
                            "read_users": {
                                "alice": true,
                                "bob": true
                            },
                            "read_write_users": {
                                "alice": true
                            }
                        }
                          }
                ]
            }
            ]
         },
    "signatures": {

        "alice":
"MEQCIAM/FYzdfVlQGWBPcyDMp2BRDyzQdTdusOl0M/UBCk2gAiAxns+4m30Y/Hz
lO0e0dK0HnaWhbxch5tUys0P0ME7ZPw=="

    }

}'
```

The payload contains `must_sign_user_ids`, which is a list of user ids who must sign the transaction's payload. The `db_operations` hold the `data_writes` to be applied on the specified `db_name`. The value in `data_writes` must be encoded in base64. The `acl` contains a list of users in the `read_users`, who can read-only the state, and a list of users in the `read_write_users`, who can both read and write to the state. Here, the signatures hold a map of each user in the `must_sign_user_ids` to their digital signature.

Checking the existence of a state

The query of `key1` can be submitted by either `alice` or `bob`, as both have the read permission to this key. No one else can read `key1`, including the `admin` user.

```
curl \

     -H "Content-Type: application/json" \

     -H "UserID: bob" \

     -H                                                         "Signature:
MEUCIQDm6dLmAdd0X49JygTiUkh+brZxprWSr2+hcAH+QIu3AAIgF+m7kO33Y
XyyqSbnXS9HR79wt/aL3JGhKvXFQaFBJms=" \

     -X GET http://127.0.0.1:6001/data/db2/key1 | jq .
```

The result contains the `value` associated with the key and also the `access_control`, and `version` as part of the `metadata`.

Updating an existing state

In order to update the value of `key1`, the following steps need to be executed:

1. Read the `key1` from the cluster.
2. Construct the updated value and transaction payload, including the read version.
3. Submit the data transaction.

Note that the `data_reads` should contain the version of the `key1` that was read before modifying the value of `key1`. If `data_reads` is kept empty, the `data_writes` would be considered as blind write.

```
curl \

   -H "Content-Type: application/json" \

   -H "TxTimeout: 2s" \

   -X POST http://127.0.0.1:6001/data/tx \

   --data '{

          "payload": {

             "must_sign_user_ids": [
            "alice"
        ],
            "tx_id": "1b6d6414-9b58-45d0-9723-1f31712add83",
        "db_operations": [
            {
                "db_name": "db2",
                "data_reads": [
                    {
                        "key": "key1",
                        "version": {
                            "block_num": 5
                        }
                    }
                ],
                    "data_writes": [
                          {
                        "key": "key1",
                        "value": "aXDUvZio",
                        "acl": {
                            "read_users": {
                                "alice": true
                            },
                            "read_write_users": {
                                "alice": true
                            }
                        }
                    }
                ]
            }
        ]
    },
    "signatures": {
        "alice":
"MEYCIQDFQpAI97qgNGrN/6lWM5v0Zn+ht3+4V5Mr57TIWDZFhAIhAKbatUhwr/l
asFAkTydKSrDr+trEJM3KEnRWlz2kYcTV"
    }
}'
```

Deleting an existing state

The following steps need to be executed in order to delete `key1`:

1. Read the `key1` from the cluster (to blindly delete `key1`, this step can be avoided, as the read version is not needed).
2. Construct the transaction payload including the read version.
3. Submit the data transaction.

```
curl \
   -H "Content-Type: application/json" \
   -H "TxTimeout: 2s" \
   -X POST http://127.0.0.1:6001/data/tx \
   --data '{
         "payload": {
            "must_sign_user_ids": [
            "alice"
        ],
            "tx_id": "1b6d6414-9b58-45d0-9723-1f31712add85",
        "db_operations": [
            {
                "db_name": "db2",
                "data_reads": [
                    {
                        "key": "key1",
                        "version": {
                            "block_num": 6
                        }
                    }
                ],
                    "data_deletes": [
                          {
                        "key": "key1"
                          }
                ]
            }
            ]
         },
    "signatures": {
        "alice":
"MEQCIATEMJZ2HYkQtG+ivADylvJRzaksTum3/jN0zeg96+CuAiAEYKugmTbPbHX
sjKnAWOLirNqI0WWOPcLN9jIlVaeseQ=="
```

```
    }
}'
```

Note that the `data_writes` and `data_deletes` can be used with multiple entries, along with many `data_reads` within a single transaction.

2.4 Deployment

The latest version of the BCDB server code can be found in the GitHub repository (https://github.com/hyperledger-labs/orion-server).

BCDB can be easily deployed and operated as a service within a cluster. BCDB is an open-source project contributed and currently maintained by IBM, and it can be updated from time to time to enhance existing capabilities, fix issues, and introduce new features.

2.4.1.1 Prerequisites

To build an executable binary file, the following are the prerequisites which should be installed on the platform on which the BCDB will be deployed:

- **Go Programming Language:** The DB uses the Go Programming Language for many of its components. Go version 1.15.x is required.
- **Make:** To build a binary file and execute unit-test, make utility is required.
- **Git:** To clone the code repository.
- **cURL:** To execute queries and transactions provided in the tutorial.

2.4.1.2 Build

To build the executable binary, the following steps need to be executed:

1. To clone the repository, first, create required directories using the command `mkdir`.
2. Change the current working directory to the above created folder by issing the command `cd`.
3. Clone this repository with `git clone`.
4. Change the current working directory to the repository root directory by issuing `cd bcdb-server`
5. To build the binary executable file, issue the command `make binary` which will create a `bin` folder in the current directory. The `bin` folder will hold two executables named `bdb` and `signer`.
6. Run the `bdb` executable by issuing the command `./bin/bdb`.

For additional health check, one can run `make test` to ensure all tests pass.

2.4.1.3 Build and start BCDB node inside Docker

Build process includes two steps – crypto materials generation and docker image build.

The minimal set of crypto materials includes 3 sets of certificates:

1. `admin` and `user` – for DB users
2. `node` – for server node
3. `CA` – for Certificate Authority

To create a minimal set of cryptographic materials run:

```
./scripts/cryptoGen.sh sampleconfig
```

To generate a docker image, after generating the crypto materials, run:

```
make docker
```

To invoke a BCDB docker container run:

```
docker run -it –rm -v $(pwd)/sampleconfig/:/etc/bcdb-server -p 6001:6001 -
p 7050:7050 bcdb-server
```

2.5 Reference Architecture Diagram

Figure 4 presents the mapping of the i4Q^{BC} solution in the reference architecture of i4Q (see D2.7 [5]). The Orion database itself is most likely to be deployed in the platform tier, either on premises or in the cloud, as a service. The users of the i4Q^{BC} solution use the Orion-SDK and may reside in the platform-tier or even in the edge-tier. This solution shall be utilized by various additional components at different levels, this service shall bel be analysed in terms of its strengths:

- **Platform Tier:** One of the sub-components to be used is “Data Brokering and storage”, enabling various components to access data exhibiting different characteristics in a unified manner.
- **Edge Tier:** The mapping to “Data collecting", enables the introduction of data stemming from multiple levels, including the edge to be integrated within this solution. This capability will enable specifically support configuration changes tracking at the edge tier.

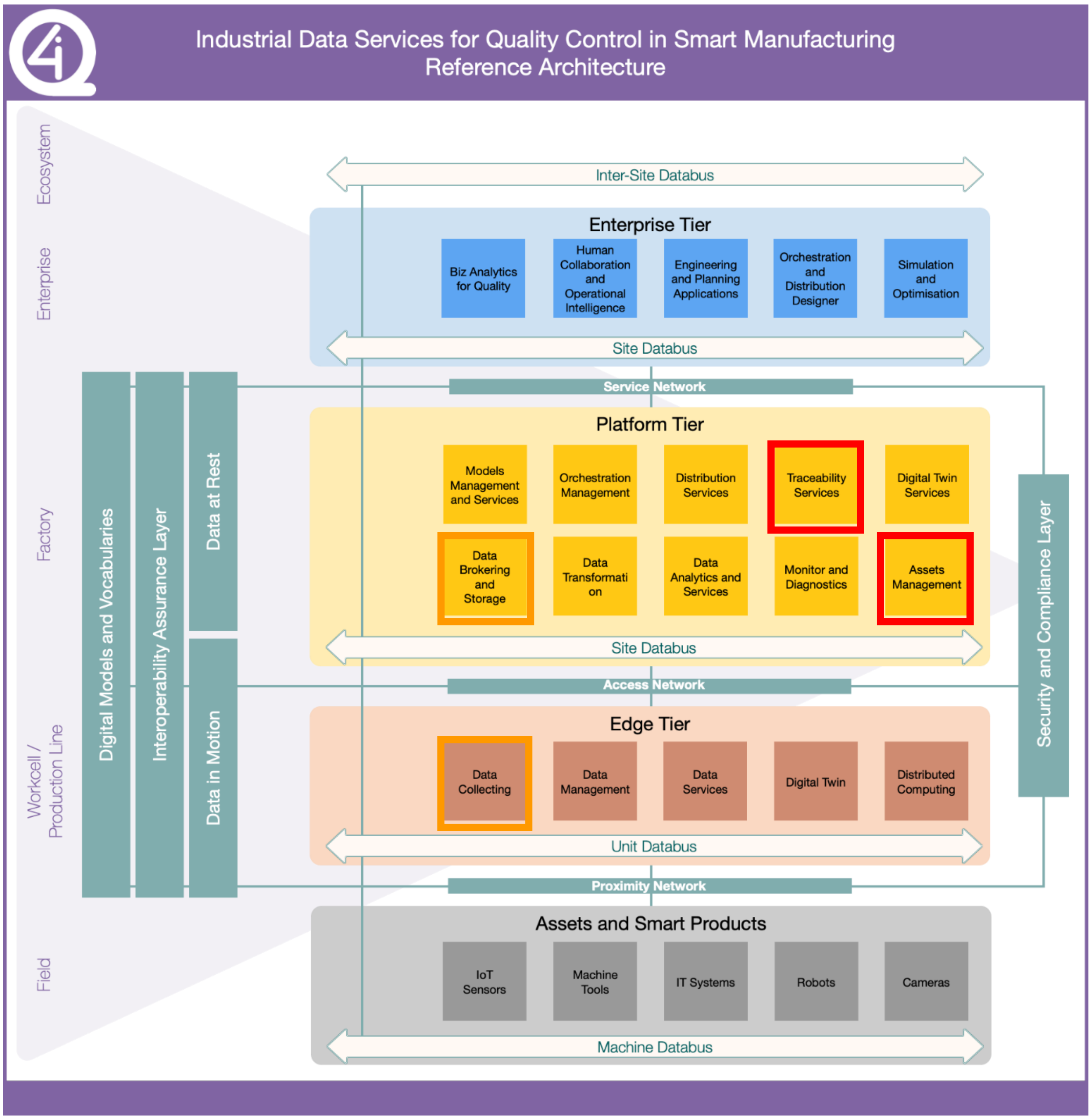


Figure 4. i4Q RA mapping with i4Q-BC

3. Implementation Status

3.1 Current implementation

The current implementation includes two components:

- A blockchain database server.
- A client SDK in Golang.

These two components constitute an MVP solution that is published in open source and readily available to any i4Q use-case for integration.

3.1.1 Solution features mapping with user requirements

i4Q^{BC} is defined in Deliverable 1.4 [3] as a solution that provides easy, trusted and traceable access to data coming from many different sources. It shall enhance the level of trust in the platform by employing a blockchain based data service, to support data traceability in the data that flows directly to the blockchain, thus serving as a single point of truth, preserving provenance and supporting non-repudiation. Deliverable 1.9 [4] maps the requirements of i4Q^{BC} to its functional components.[5] Requirements including the i4Q-BC solution are presented below.

The first requirement from i4Q^{BC} is to provide blockchain functionality, thus increasing trust, and providing a single source of truth for important data records. This requirement is addressed by the cryptographic layers that envelope the key-value store in Hyperledger Orion. The blockchain features presented by Orion are: immutability, tamper evidence, authenticity, and non-repudiation. Immutability and tamper evidence are implemented by employing a set of cryptographic data structures: a hash chain, hash skip-list, Merkle tree, and Merkle-Patricia trie. Authenticity and non-repudiation are implemented by the pervasive use of signatures, for both TXs and queries from the clients, as well as for server responses and stored blocks.

The second requirement from i4Q^{BC} is to expose interfaces to allow transaction submission and queries. This is achieved by the Orion server exposing data centric REST APIs, which allow the user to submit TXs and execute queries. The programming interface is made even more accessible and easy to use by the introduction of a client-side golang SDK, which exposes a transactional data centric API that hides most of the complexities of the REST API from the user. Moreover, the golang SDK makes accessing the cluster easier, and automatically manages replica selection and failover.

The third requirement from i4Q^{BC} is to provide traceable access to data. This is achieved the provenance API for exploring data lineage. The provenance API allows the understanding of all transformations the data underwent along the way: how, when and by whom.

[5] See i4Q D1.9 – Requirements Analysis and Functional Specification v2, p61.

3.1.2 The Hyperledger Orion server

The Hyperledger Orion server implements a replicated blockchain database, and is available as open source: https://github.com/hyperledger-labs/orion-server

It can be deployed native on a Linux server or as a docker container and supports cluster deployments (see Section 2 for more details). The feature set that is available addresses the requirements specified in Task 3.2:

- Authenticity / Non-Repudiation – Proving that the data received exactly as it was sent by the source, preventing authorship disputes
- Tamper- Evidence – Detection of any changes to the protected object
- Provenance / Data lineage – Recording & understanding of all transformations the data underwent along the way: how, when and by whom.

In addition, Orion also supports Multi-signature transactions, a mechanism to approve transaction only when signed by several designated parties.

The Orion DB is mapped for deployment in the platform tier of i4Q.

3.1.3 The Hyperledger Orion client SDK

The Hyperledger Orion SDK-Go implements a client SDK that exposes a rich and easy to use transactional database API. It is available as open source: https://github.com/hyperledger-labs/orion-sdk-go

The client SDK can be used to easily integrate the Orion DB into any solution. It can be used in solutions that reside in the platform tier and/or in the edge tier of i4Q,

3.1.4 Demonstration

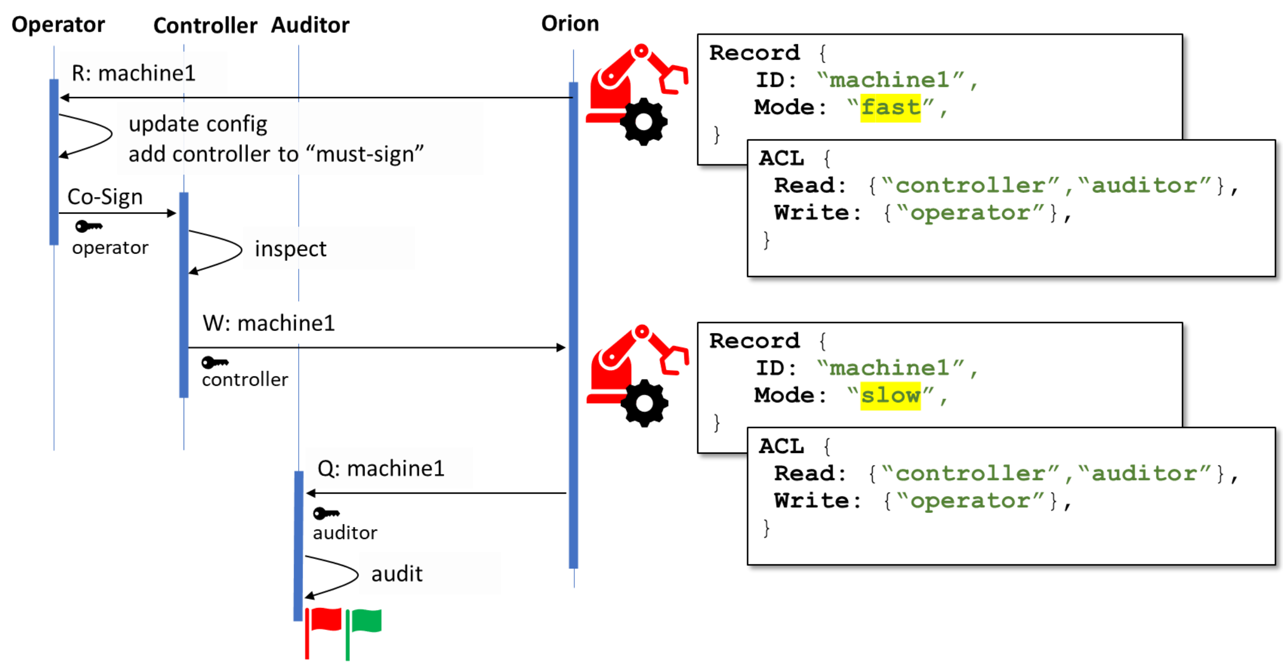
The capabilities of Hyperledger Orion for manufacturing quality were demonstrated using a simple application that saves all the changes to machine and production line configuration in to the blockchain database.

Data records are machine configuration files, in JSON format.

The actors are:

- Operator: manages multiple machines by changing their configuration, has RW access to machine configuration. The operator initiates a machine configuration change, prepares it, and passes it for review to the Controller.
- Controller: reviews and approves operator requests to change machine configuration. The controller is a required signatory on each configuration change request and has to approve and commit each request to the database. The controller has RO access to machine configuration, so he cannot change the configuration by itself.
- Auditor: verifies and audits the configuration changes, periodically or upon request. The auditor has RO access to machine configuration records and can execute full provenance queries, tracing the activity of both the operator and controller.

In the demo it was shown how to setup the database, manage users (operators, controller, auditor), execute configuration change transactions, execute operator change transactions, and execute various provenance queries, tracing the activities of the actors and evolution of the data.



Operator
Controller
Auditor
Orion
R: machine1
update config
add controller to "must-sign"
Co-Sign
operator
inspect
W: machine1
controller
Q: machine1
auditor
audit
Record { ID: "machine1", Mode: "fast", }
ACL { Read: {"controller","auditor"}, Write: {"operator"}, }
Record { ID: "machine1", Mode: "slow", }
ACL { Read: {"controller","auditor"}, Write: {"operator"}, }

Figure 5. Changing machine configuration and conducting an audit.

Figure 5 presents the flow of changing machine configuration and conducting an audit:

- **Operator** prepares a transaction
- Reads existing machine configuration record, captured within the Read-Set
- Changes machine configuration
- Adds controller to "Must-Sign" set
- Co-signs and gives the TX to the controller
- **Controller** inspects the TX, Co-signs, and submit
- Validates details
- Optional: provenance query? operator, machine, policies?
- Co-signs and submits the TX to Orion
- **Auditor** executes provenance queries
- Validates policies
- Detects anomalies
- Raises a flag

3.2 Next developments

The next steps in the development of the i4Q^{BC} solution include:

- Performance measurements and optimization of Hyperledger Orion.
- Publishing a paper about Hyperledger Orion, including aspects of the architecture, uses cases and performance.
- Integrating more closely with one of i4Q's use cases.
- Publishing a paper or blog post about the role of Hyperledger Orion in i4Q and in manufacturing quality applications in general.

3.3 History

Version	Release date	New features
Orion server v0.1.0	June 24 2021	API ready, single node
Orion server v0.2.3	April 5 2022	Cluster deployment, including docker

4. Conclusions

This deliverable describes the project's ambition concerning BC technology, and listed the BC specific requirements derived from Task 3.2. It then introduced the main BC technology concepts and constructs.

The document compared between two BC technologies, Hyperledger Fabric and Hyperledger Orion (BCDB), with respect to the project's requirements, and reasoned about what BC infrastructure is the best choice for i4Q. As detailed in this deliverable, i4Q^{BC} is an entity within this project that is trusted by all platform users, therefore i4Q can operate under the centralised trust assumption. Hence, it was agreed by the relevant parties that instead of the initially suggested Hyperledger Fabric, the i4Q^{BC} platform is be based on BCDB, which is specifically designed to efficiently address the centralised trust ecosystems like i4Q. This change provides several significant benefits for the i4Q project when compared to Hyperledger Fabric, such as easy installation and administration, standard programming model, significantly higher efficiency, and much lower operational costs, while fulfilling all use-case requirements.

Finally, this deliverable described the Hyperledger Orion properties and trust assumptions, the high-level architecture, its APIs, and included deployment instructions for a Hyperledger Orion server.

References

[1] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., D.Caro, A., Enyeart, D., Ferris, C., Laventman, G, Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., & Yellick, J. (2018). Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference* (pp. 1-15).

[2] Gartner. (2020, June 22). *Gartner Identifies Top 10 Data and Analytics Technology Trends for 2020* [Press release]. Retrieved from: https://www.gartner.com/en/newsroom/press-releases/2020-06-22-gartner-identifies-top-10-data-and-analytics-technolo

[3] i4Q deliverable D1.4 – Requirements Analysis and Functional Specification.

[4] i4Q deliverable D1.9 – Requirements Analysis and Functional Specification v2.

[5] i4Q deliverable D2.7 – i4Q Reference Architecture and Viewpoints Analysis v2.

Appendix

The PDF version of the **i4Q Blockchain Traceability of Data** web documentation, which can be accessed online at: **http://i4q.upv.es/3_i4Q_BC/index.html**

i4Q Blockchain Traceability of Data

General Description

The Blockchain Traceability of Data (i4Q^{BC}) solution aims to enhance the level of trust that different solutions and components can place on data. Thus, it shall serve as one of the cornerstones of data storage services to be consumed by different solutions. This solution provides services of immobility and finality of data, serving as the source of truth, enabling trust in data by providing the possibility for full provenance and audit trail of data stored. Thus, the main functionality offered by this solution is comprised of data trust traceability, enabling a full audit trail of assets and data.

The Blockchain Traceability of Data (i4Q^{BC}) solution is based on the Hyperledger Orion blockchain database. Hyperledger Orion is a centralised, trusted blockchain database that provides tamper-evidence, provenance, data lineage, authenticity, and non-repudiation through data centric Application Programming Interfaces (APIs), with transactional semantics and very simple well-known programming model.

The Hyperledger Orion server implements a replicated blockchain database, and is available as open source: https://github.com/hyperledger-labs/orion-server It can be deployed native on a Linux server or as a docker container and supports cluster deployments.

The Hyperledger Orion SDK-Go implements a client SDK that exposes a rich and easy to use transactional database API. It is available as open source: https://github.com/hyperledger-labs/orion-sdk-go The client SDK can be used to easily integrate the Orion DB into any solution.

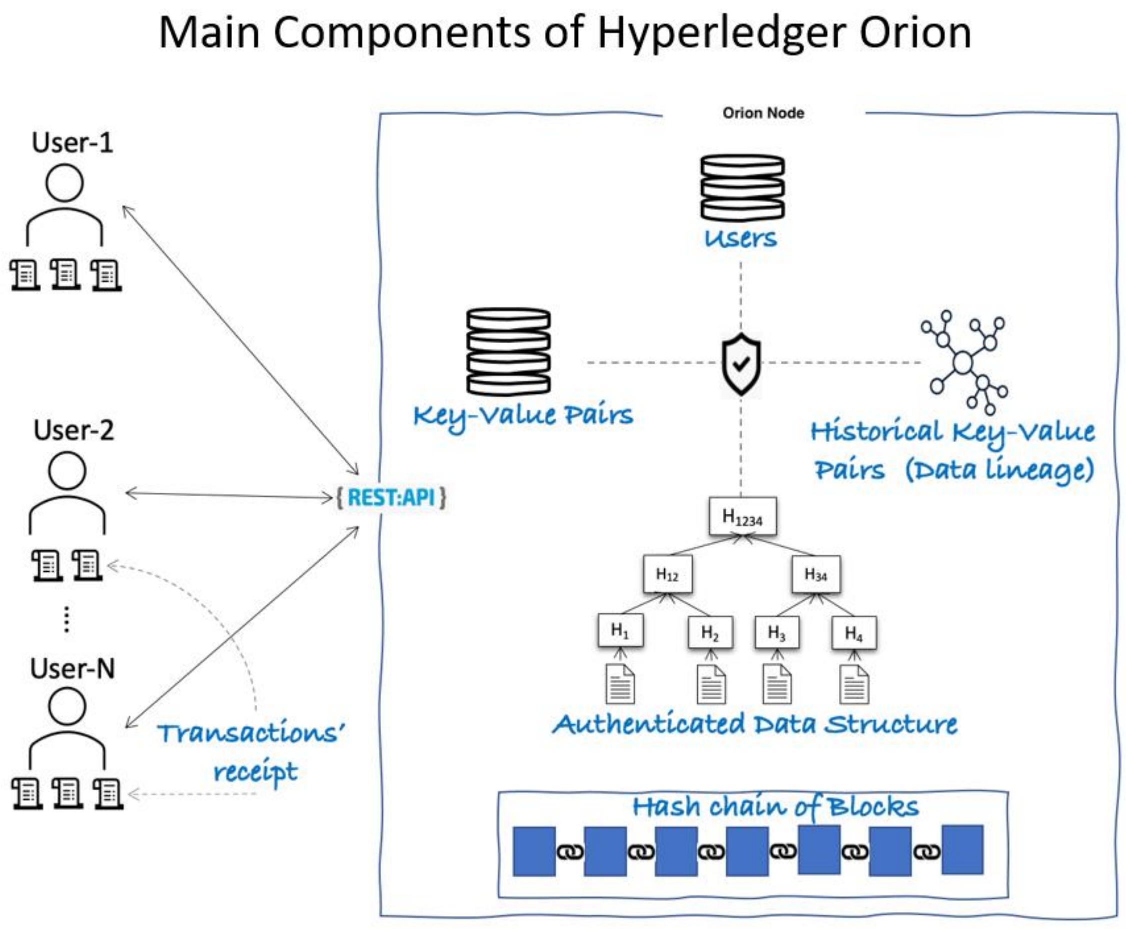
Features

The main features provided by this solution include:

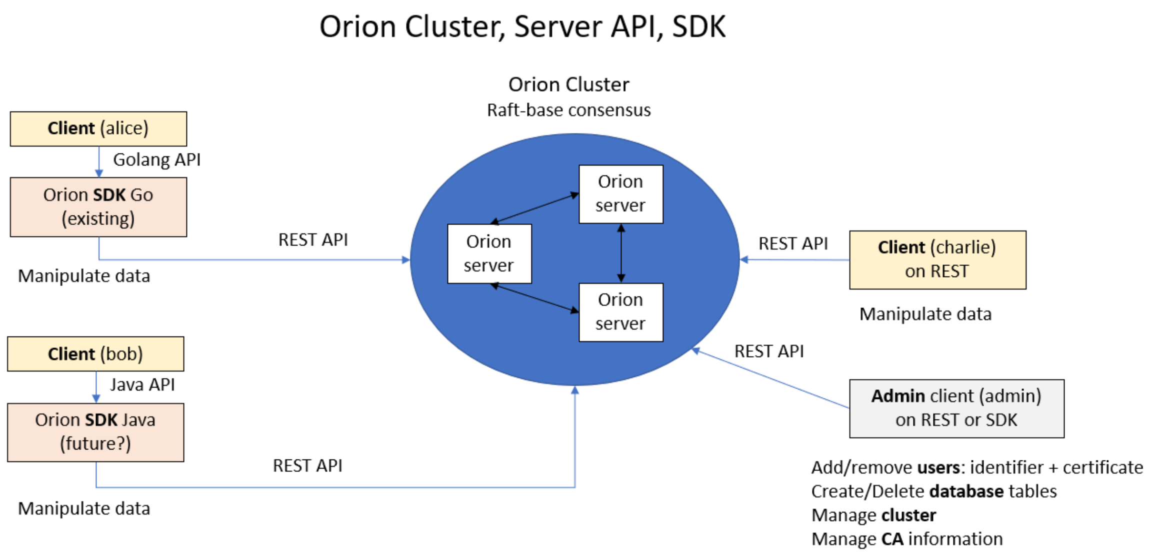
1. **A key-value database**: A replicated key-value database with transactional APIs. A data centric API provides a familiar and easy to use programming paradigm, which reduces complexity and cost compared to traditional blockchain platforms. A cluster of servers provide a fault tolerant & highly available solution.
2. **Immutability and tamper evidence**: All stored data is bound with cryptographic data structures (e.g.: a hash-chain), which ensures immutability and tamper evidence.
3. **Authentication and non-repudiation**: The use of digital signatures by clients and servers, for both requests, responses, and stored data, provides authentication as well as non-repudiation.
4. **Privacy**: Fine-grain privacy is achieved using key-level access control, which is enforced by the digital signatures provided by users.
5. **Multi-party transactions**: The ability to require that multiple users sign a transaction in order to make it valid, allowing tight controls on data mutations and multi-party agreements.
6. **Provenance**: All historical changes to the data are maintained within the ledger as well as in a graph structure. The solution allows the user to execute provenance queries on those historical changes to understand the lineage of each data item.

ScreenShots

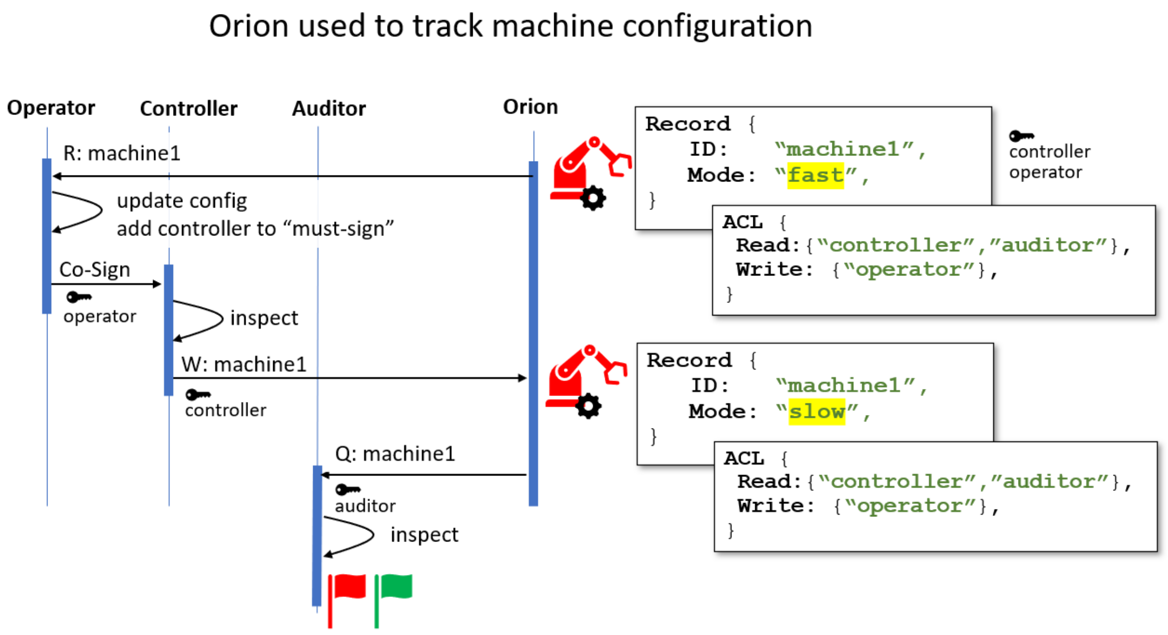
The main components of Hyperledger Orion are depicted below:


Main Components of Hyperledger Orion
User-1
User-2
User-N
{ REST:API }
Transactions' receipt
Orion Node
Users
Key-Value Pairs
Historical Key-Value Pairs (Data lineage)
H_{1234}
H_{12}
H_{34}
H_1
H_2
H_3
H_4
Authenticated Data Structure
Hash chain of Blocks

The **cluster**, **server API**, and **SDK** of Hyperledger Orion are shown below:



An **example** showing how machine configuration can be tracked using Hyperledger Orion:


Orion used to track machine configuration
Operator
Controller
Auditor
Orion
R: machine1
update config
add controller to "must-sign"
Co-Sign
operator
inspect
W: machine1
controller
Q: machine1
auditor
inspect
Record { ID: "machine1", Mode: "fast", }
ACL { Read: {"controller","auditor"}, Write: {"operator"}, }
controller
operator
Record { ID: "machine1", Mode: "slow", }
ACL { Read: {"controller","auditor"}, Write: {"operator"}, }

Commercial Information

Authors

Company	Website	Logo
IBM	https://research.ibm.com/labs/haifa/	

License

The Hyperledger Orion server and the Go-SDK are open source with **Apache License 2.0**. See: - Server license - SDK license

Pricing

Subject	Hyperledger Orion (Server, SDK)
Payment Model	One-off
Price	Free

Associated i4Q Solutions

Required

- i4Q Blockchain Traceability of Data solution has no dependency on another i4Q solutions.

Optional

- i4Q IIoT Security Handler may be used as a Certificate Authority (CA) provider.

System Requirements

- OS: Linux (tested on Ubuntu and RedHat).
- Hardware: a set of servers capable of running a database cluster. For production at least 3 are required for fault tolerance, however 5 servers are recommended.

API Specification, User Manual, and Deployment Instructions

The complete **documentation** of Hyperledger Orion, including **API Specification**, **User Manual**, and **Deployment Instructions** is specified here: http://labs.hyperledger.org/orion-server

The **sources** of Hyperledger Orion can be found here: - Server: https://github.com/hyperledger-labs/orion-server - SDK: https://github.com/hyperledger-labs/orion-sdk-go

Docker images can be found here: https://hub.docker.com/r/orionbcdb/orion-server

Hyperledger Orion was presented in a Hyperledger London meetup, see the video on YouTube.