Graduado en Ingeniería Informática

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

TRABAJO FIN DE GRADO

Benchmarking of Blockchain Technologies used in a Decentralized Data Marketplace

Autor: Alberto Fuentes Contreras
Director: Miguel Jiménez Gañán

MADRID, JUNIO 2019
Benchmarking of Blockchain Technologies used in a Decentralized Data Marketplace

Alberto Fuentes Contreras

02-06-2019
Acknowledgement

The author would like to thank his family for the care and encouragement received during the best and worst moments in his university studies.

I really appreciate and thank the whole ConWet Lab team (specially Miguel Jiménez Gañán, Francisco Javier Soriano, Francisco de la Vega and Agustín Rodríguez), for such a great experience I had working with them, in such an innovative and ambitious project. I would like to remark how much patience they have actually had since I joined.

I would also thank everyone I met in this campus for being so awesome; such great memories I have made since I entered in 2015.

I also should not forget about all the teachers that have taught me during my studies; because they made easy what may have taken me much more time to learn by myself.

Special thanks to the FIWARE team for developing amazing products that let developers create P2P platforms easily.

I would like to thank Satoshi Nakamoto and all the Cypherpunks that originally envisioned such a new world of possibilities.

I would like to thank the whole Hyperledger community as well as the Linux Foundation for maintaining an Open Source framework for Blockchain Platforms development.

I also would like to thank the IOTA Discord Community for helping and answering me whenever I got stuck using the IOTA framework.
Contents

1 Abstract 7
1.1 Propósito del trabajo ................................. 7
1.2 Purpose of the research ............................... 8

2 Introduction and Objectives 9

3 State of the Art 11
3.1 FIWARE Context Broker ............................... 11
3.2 Blockchain Basics ................................. 14
3.3 IOTA Basics ................................. 17
3.4 Hyperledger Basics ................................. 17
  3.4.1 Hyperledger Fabric Concepts ................... 18
  3.4.2 Hyperledger Composer Concepts ................. 19
  3.4.3 Hyperledger Burrow .............................. 21
  3.4.4 Hyperledger Grid ................................. 21
  3.4.5 Hyperledger Indy ................................. 21
  3.4.6 Hyperledger Iroha ................................. 21
  3.4.7 Hyperledger Sawtooth ............................ 22
  3.4.8 Hyperledger Explorer ............................. 22
  3.4.9 Hyperledger Quilt ................................. 23
  3.4.10 Hyperledger Ursa ................................. 24
  3.4.11 Hyperledger Cello ................................. 24
  3.4.12 Hyperledger Caliper ............................. 24
3.5 BigchainDB Basics ................................. 25
  3.5.1 BigchainDB-ORM ................................. 28
  3.5.2 Using BigchainDB for data sharing ............. 29
3.6 Description of the Distributed Marketplace Application ....... 31
3.7 Vegeta HTTP Load Testing ........................... 35
G  MongoDB Benchmarking tool  157
H  Hyperledger Benchmarking Tool  158
I  Hyperledger Benchmarking Log Parser  161
J  Data Marketplace Benchmarking Tool  163
K  Data Marketplace Log Parser  164
L  Data Marketplace Docker-Compose YAML file  165
1 Abstract

1.1 Propósito del trabajo

Este documento pretende ofrecer resultados sobre el rendimiento de diferentes plataformas Blockchain; usadas para el desarrollo de la implementación del Mercado de Datos Distribuido propuesto en [1].

Las plataformas Blockchain ofrecen contabilidad y no repudio de un registro distribuido y replicado en todos los nodos de la red Blockchain, que contiene todas las transacciones mandadas por los usuarios; pese a todas las bondades del Blockchain, existe un coste computacional producido por el algoritmo de consenso entre todos los nodos, sobre la validez de las transacciones enviadas por los usuarios, por tanto, se introduce un retardo en las comunicaciones. Este retardo es el objeto de estudio.

Tras la aparición de la tecnología Blockchain en 2008 como tecnología base de Bitcoin, han surgido multitud de plataformas Blockchain, todas ellas con un caso de uso concreto; lo que ha requerido de una investigación para identificar las plataformas Blockchain que mejor se adecuaban a la arquitectura propuesta en [1], dando lugar al uso de dos Blockchains para diferentes usos: BigchainDB para el envío de datos en tiempo real y Hyperledger Fabric para almacenar los datos relacionados con la lógica de negocio.
1.2 Purpose of the research

This research is meant to provide results on the performance of different Blockchain technologies; used to develop the Decentralized Data Marketplace Application proposed in [1].

Blockchain Technologies provide accountability and non-repudiation of a registry (ledger) which is unique, distributed and replicated in every node; that contains all the transactions sent by the peers connected to the Blockchain network; but it does not come free of cost, the process that ensures consensus, between all the nodes, about the validity of transactions, introduces a delay in the communications between them. This delay is the object of study.

After the appearance of the Blockchain technology in 2008 as the core technology of the Bitcoin, plenty of Blockchain platforms have appeared, each of them with a concrete use case in mind; because of that a research to identify the Blockchain technologies that best fitted the architecture described in [1] was needed, which ended up in using two Blockchain technologies, each of them with a certain purpose: BigchainDB is used to send data in real-time and Hyperledger Fabric is used to store all the data related with the business logic.
2 Introduction and Objectives

The *Decentralized Data Marketplace* in [1] was conceived with the intention to provide a trustless platform (thanks to Blockchain traceability and non-repudiation capabilities) that outperformed other Peer-to-Peer Marketplace solutions, which are limited to only provide static Datasets; that is, the users can only download data from the databases in a synchronous way.

In order to solve that limitation of *State of the Art* Marketplace solutions; *Subscription* and *Usage based* pricing models were implemented, letting users obtain data updates asynchronously.

Many of these solutions are built up on top of a Peer-to-Peer architecture; which does not provide *non-repudiation*, the solution proposed in [1] makes use of Blockchain technologies to ensure *non-repudiation* of the payments, amount of consumed data, agreements between users...

Nowadays, Blockchain technologies have evolved into platforms with specific use cases; so that, there is a need to choose the best platforms that fit our architecture, after a research on different Blockchain products and their use cases, there was not a platform that could fit all of our needs, so that different Blockchain technologies were used in the architecture, each of them to carry out a certain task:

- In order to provide a permissioned environment, the *Hyperledger Fabric* framework was chosen.
- The Hyperledger Composer Framework provides flexibility during the development of the architecture, providing an *Object Oriented Programming* language to define Assets and a JavaScript engine to create Smart Contracts.
- In order to provide encrypted channels to let customers and sellers communicate and send data, IOTA Masked Authenticated Messaging and BigchainDB were considered. In the following chapters, it is shown that due to a bad performance of the IOTA Blockchain, BigchainDB was chosen to serve as the communication platform instead of IOTA.

The use of the *FIWARE Context Broker* is necessary to provide the *Peer-to-Peer* capabilities between users; the P2P architecture between Context Brokers is possible when a *Federation* is created between all the involved Context Brokers of a network; *registrations* are created in all the context
brokers, they contain the URL of the data provider; in order to forward the requests to them in case the user’s context broker cannot resolve the requests.

One of the main concerns of this project is providing a platform that is efficient and scales during peak traffic loads; because of that, there was a need to measure which were the most efficient technologies that best fit our needs.

One of the biggest issues during the development of the Decentralized Data Marketplace was the performance: at first using the IOTA Blockchain for the Data communication, added a big overhead to the system, due to the used Proof of Work Consensus Algorithm by the IOTA Blockchain, that increased the latency during the data distribution, so that, the biggest change to the initial planning was substituting the IOTA Blockchain technology with BigchainDB, which Consensus Algorithm is much lighter in terms of Algorithmic Complexity.

Before choosing BigchainDB as an alternative to IOTA, a research on P2P and Distributed Databases was hold to learn about the state of the art of systems that are suitable for real-time applications that use IoT data flows, without using Blockchain technologies; to make sure that Blockchain was really needed and non-repudiation could not be provided without using Blockchain.

After some research and acknowledging the need of a Consensus procedure to ensure the accountability and non-repudiation of every module of the architecture, with the speed and throughput of a Distributed Database; tests with the BigchainDB Blockchain were carried out; and all the efforts were focussed on this technology instead of IOTA, due to the better performance that BigchainDB offered with respect to IOTA.

An Introduction to the IOTA technology, and an analysis of existing Communication Protocols using the IOTA network were kept in the research, because they were documented before the IOTA performance tests.

Details about the deployment of the Software and the results after measuring the performance of the technologies are given. The software used to measure the Benchmarking metrics can also be found in the appendices.

The research also covers the analysis of different messaging protocols that use the IOTA network, their advantages, disadvantages and use cases in which they can be used.
3 State of the Art

This chapter is meant to provide an introduction to Blockchain platforms, tools and frameworks that were considered to be used during the development of the Decentralized Data Marketplace and its Performance Benchmarking.

Even though only Hyperledger Composer, Hyperledger Fabric and BigchainDB were the chosen components for the architecture, the rest Hyperledger frameworks and IOTA are discussed due to the relevance they had during the research.

3.1 FIWARE Context Broker

The FIWARE Foundation is behind the development of open platforms, frameworks and standards that provide the resources to build applications that require Big Data management and consumption, Peer-to-Peer communication and synchronization and IoT sensor integration.

The FIWARE Foundation products are called Generic Enablers; they all have different purposes and use cases that fit in every FIWARE deployment. For each of the open standards, the FIWARE Foundation provides an implementation; the community is open to write their own solutions based on the standard, these implementations can be proprietary or open.

The FIWARE Foundation is focused on making open and generic standards that fit the development of Smart Cities; in contrast with other proprietary solutions that require the burden of maintenance and adaptation to make the solutions fit different cities.

The FIWARE Context Broker is the Generic Enabler that provides Context Information consumption and management; the Context Broker serves as a middleware between the IoT sensors and the user application, that manages the data from the IoT sensors connected to each node.

All the Data obtained by each IoT sensor is stored using an Entity, whenever new data is sensed by an IoT sensor, the Entity that contains data from this IoT sensor is updated with the new generated values; previous values are overwritten and not available again after each data update.

The following is an Entity example with Weather data:

```json
{

```
In order to receive data updates, the users will subscribe to a *Dataset* offered by a seller, the subscription request will make the API create a *Subscription* on their own Context Broker. In order to make the changes to the data be sent to the subscribed customer, once the seller receives a *Subscription request*, and checks its validity, a *Subscription* will be created in the seller’s Context Broker, to automatically execute a function that will send *Data Updates* to the subscribed customer. The following is an example of a Subscription:

```json
{
  "description": "A subscription to get info about Weather",
  "subject": {
    "entities": [
      {
        "id": "Weather",
        "type": "Weather"
      }
    ],
    "condition": {
      "attrs": [
        "temperature", "rain"
      ]
    }
  }
}
```
The Context Broker also supports Queries; whenever a customer with a One Time Payment Agreement of one of the available Datasets sends a Query to the Context Broker, the Context Broker will check if the desired data is in the Context Broker as an Entity, otherwise it will check for a Registration of the data, that will forward the query to the seller’s Context Broker.

During the "Data Marketplace Application Load Testing" subsection, the accumulator server is mentioned; it is just an HTTP client that logs HTTP requests. It is used to timestamp when data updates are received on the client node; it is used as a developer tool and it is not meant for production environments. The client application that consumes the data should substitute the accumulator server in a working architecture.

The Peer-to-Peer architecture between Context Brokers appears when a Context Broker Federation is created. The Federation is achieved thanks to the Registrations, which provide information on where to find the desired data from an Entity, in case the Entity is not available in the Context Broker itself.

The following is an example of a registration, which contains the provider URL; that URL is where the Context Broker will send the HTTP request that was previously received, in order to resolve it:

```json
{
  "dataProvided": {
    "entities": [
      {
        "id": "Weather",
        "type": "Weather"
      }
    ],
```

13
3.2 Blockchain Basics

The first Blockchain architecture was proposed in [2] and gave birth to the Bitcoin network with the idea of decentralizing the monetary exchange between users. Since then, Blockchain technologies have evolved into different architectures and forms, meant to be used for many different purposes.

The nodes that compose the network need to be synchronized to make sure that they all share the same information: a copy of the ledger; to achieve that, they execute a consensus algorithm.

The consensus algorithm determines the order in which transactions are included in the ledger and ensures its validity. It is necessary to run a consensus algorithm to avoid malicious peers trying to introduce invalid transactions and to ensure that all the peers have the same copy of the ledger.

A general Blockchain taxonomy is presented below:

- Depending on the user on boarding:
  - Public: everyone can access the network by installing the required software.
  - Private: the users that desire to join the network need to have the permission from a Certificate Authority that makes sure to recognize the user with an unique identity (certificate) Private Blockchains are also called permissioned, because the user cannot join the network without the permission of the members of the network.
• Depending on the Consensus Algorithm:

  – **Proof Of Work**: consensus algorithm in which the nodes of the network apply a *Hashing* function to the created blocks and an incremental index, until a valid hash is found, in that case the node shares the found hash to the *neighbours* that will check its validity, in case it is valid a reward will be given to the user, and the *block* will be appended to the last one. This Consensus Algorithm is the most computational intensive. This kind of Consensus Algorithm is prone to causing huge amounts of electricity waste, as more powerful devices are created, more hashes can be performed per second and more electricity is needed to make these devices work.

  – **Proof Of Stake**: the nodes of the network decide which node is validating each generated block; instead of them all *mining* it until one finds a valid hash. This choice needs to be randomly distributed between all the nodes that are part of the network. This kind algorithm reduces the computational intensity of the Proof of Work algorithm, but it is also prone to being insecure, in case the choice of the validating node is not randomly distributed; as the malicious node could validate several valid transactions after an invalid one, if the same malicious node was chosen to validate a consecutive row of transactions. Proof of Stake may also reduce the incentives that users receive, as the amount of computing capabilities will not give a big advantage to those who can perform more Hashes per Second.

• Depending on the Blockchain Structure:

  – **Blockchain**: ledger data structure in which all transactions point to the previous one; it looks like a *Linked List*, but it has branches caused by *forks* (bifurcations created automatically when two nodes validate at the same time two different valid blocks).

  – **Directed Acyclic Graph**: ledger data structure introduced by the *IOTA Foundation*; the data in the ledger is not in a *Linked List* shape as in common Blockchain technologies.

  Before a transaction is appended to the network, the sending node needs to validate two unvalidated transactions in the ledger (*tips*), after that, the transaction is appended pointing to the two transactions that the node validated (the new transaction
is added as a tip, until a node validates it and becomes a valid transaction). This makes the structure to be acyclic and directed; they all point to the *Genesis Transaction*.

All the *cryptocurrencies* are based on the interchange of *tokens* (funds) between *peers* using *transactions*, to achieve that in a secure way without needing an intermediary, the tokens are sent using *wallets* (public and private key pair), that sign the transactions of the users, hold the users funds and simplify the way a user shares her account (usually providing a *QR code* that represents the public key of the account).

Every single token generated in the network is uniquely identified. These tokens can be obtained in online exchanges easily, in exchange to any *fiat currency* or another *cryptocurrency*.

In case a user wants to receive funds, she provides her public key to the sender. On the other hand when the user desires to send funds, the
transaction is signed with the private key of the sender to prove that this user really made that transaction. During the Payment Validation process; the paid amount, the user’s identity and the transaction timestamp will be checked, in order to avoid cheating users attempting to fool the system with fake or double spending transactions.

3.3 IOTA Basics

IOTA is considered a Public Blockchain which runs a Proof of Work consensus algorithm. The Hashing function that is used by IOTA is called "Troika".

The "Tangle" is the name used to identify the Directed Acyclic Graph data structure that shapes the IOTA’s Blockchain. In the Tangle all the transactions get attached for validation. Before the transaction’s validity is proved, the transaction is called a "tip".

IOTA permits feeless data transactions. When the user wants to send a transaction, the node to which is connected gets two tips from the Tangle, validates them and then inserts the user’s transaction as a tip for later validation by another node in the network.

The IOTA Foundation developed a communication protocol using the Tangle. The transactions were used as messages to transport data between peers. This protocol is known as Masked Authenticated Messaging, which will be explained in detail in the Communication Protocols using the Tangle subsection.

Plenty of applications using this protocol have been developed by the community to perform M2M (Machine to Machine) communications and secure chat channels.

In addition to the MAM protocol, many different communication protocols and APIs, that use the MAM API appeared, providing alternatives that were meant for different use cases; these will be covered in the "Communication Protocols using the Tangle" subsection.

3.4 Hyperledger Basics

The Hyperledger Fabric framework offers the developer the tools to create Private Blockchain (also known as Permissioned Blockchain) solutions.
v0.20.7

Composer v0.20.7

At this point in time, there will be no further releases for Composer, here are some experimental features that were put into composer during its development. They are experimental because they may not have full automated testing, samples, are undocumented and could still have bugs, but we just wanted to ensure the community knew about them.

Figure 2: Hyperledger Composer v0.20.7 release notes.

*Hyperledger Composer* provides the tools to design complex asset models and *Smart Contract* logic that is run on the Hyperledger Fabric network.

The *Hyperledger Foundation* has many other projects on going; the Blockchain Architect should choose one of them depending on the goals and characteristics of the Blockchain network that best satisfies the needs.

At the time of writing, the *IBM Blockchain* working group stopped the development of *Hyperledger Composer*, and the v0.20.7 is the last version of the framework, as stated in the last release notes.

Blockchain applications integrated in the *Hyperledger Fabric* framework can run in one of the following ways:

- Natively, only using the *Hyperledger Fabric* framework.
- Taking advantage of the abstraction provided by *Hyperledger Composer*; and using it to create asset models, which simplifies the burden of development.

### 3.4.1 Hyperledger Fabric Concepts

*Hyperledger Fabric* provides the infrastructure to create *permissioned Blockchain* networks. *Hyperledger Fabric* uses the *Kafka Consensus Algorithm*. The Hyperledger Fabric networks are made of the following components:

- Peers: each node of the network which function is to execute chaincode and receive transactions from external client applications that use the *Hyperledger Fabric API*. 


- Orderers: dedicated to collecting transactions from all Hyperledger Fabric channels in a network, and creating blocks with them; they also validate the channel creation transactions issued by the network administrator.

- Certificate Authority (CA): provides certificates to the blockchain members.

- Ledgers: collection of ordered blocks that contain transactions.

- Channels: provides isolation and privacy to the organizations that communicate using it; each channel contains one ledger, and one ledger can only exist in one channel. Two organizations that communicate through different channels, will not be able to see each other transactions.

- Consortium: provides the organizations the ability to create agreements on which policies will be followed to validate blocks.

The common steps followed to setup a *Hyperledger Fabric* network are the following:

1. Network Creation.
2. Consortiums Definition.
3. Channels Creation.
5. Smart Contract Deployment.

Once the network is started, Smart Contracts can be updated, or added; new peers can join, new consortiums or channels can be defined. These operations will require a transaction requested by the network administrator. Only the network administrator can make these kind of changes to the network.

### 3.4.2 Hyperledger Composer Concepts

*Hyperledger Composer* was conceived as a way of providing the modularity, ease of definition and flexibility provided by the *Object Oriented Programming paradigm*. It does not only provide a layer of abstraction, but also,
adds plenty of concepts to the Blockchain Terminology, that simplify the creation of a Blockchain based application.

- Event: they act like callback transactions, to acknowledge the client application any details, that the developer needed to know about the success or failure of the chaincode execution. They are not mandatory but recommended; they provide plenty of flexibility and were widely used for the development of the Decentralized Data Marketplace Application and the Benchmarking environment.

- Transaction: messages that contain information provided by the client application, bundled into a transaction by the peer node. They have the contents of the assets that later on will be generated.

- Asset: entity that exists in the Ledger. They can be queried, modified (using transactions), and deleted if permitted by the Consortium Policy. They behave like objects in the OOP paradigm.

- Concept: they act as building components of assets; they are useful when different assets have attributes in common; they provide the developer the ability to reuse code. They cannot be searched in the ledger as the users can search for assets, because they are not individual entities.

- Connection Cards: contain the user credentials; the user needs to create one connection card to join the network. Each .card file contains 2 files: connection.json and metadata.json.

The Hyperledger Composer network data is bundled in a .bna file (Business Network Application). This .bna file contains the following files:

- logic.js: contains the chaincode of the Business Platform.
- model.cto: contains the Object Oriented descriptions of the assets, transactions and events.
- package.json: contains all the node modules dependencies needed to deploy the network.
- permissions.acl: contains the permissions that different roles have on the world state.
3.4.3 Hyperledger Burrow

Hyperledger Burrow is a Hyperledger Framework that implements the Ethereum Virtual Machine architecture running the Tendermint Consensus Algorithm, so that, it is able to run Ethereum Smart Contracts written in Vyper or Solidity in a Permissioned environment.

The project started with the intention to motivate other Hyperledger Working Groups to add the Ethereum Smart Contract support to other Hyperledger Frameworks like Fabric.

The Hyperledger Foundation announced the Hyperledger Fabric v1.3 release with this feature.

3.4.4 Hyperledger Grid

Hyperledger Grid is much more than a framework; it covers different tools that let the developer create Supply Chain Blockchain platforms.

Hyperledger Grid exists because one of the most common Blockchain use cases is to record information during the different phases of a product manufacturing.

3.4.5 Hyperledger Indy

Another common Blockchain use case, due to its immutability feature, is providing a platform aimed to share Decentralized Identities.

Hyperledger Indy aims to provide this service to even let the users of the Blockchain Platform to share their Digital Identities outside the Blockchain platform.

3.4.6 Hyperledger Iroha

Hyperledger Iroha provides the developer a framework to develop Blockchain Platforms, supporting Java, JavaScript, Python and C++. The framework also provides the tools to develop Blockchain Applications for Android and iOS.

Iroha implements Smart Contract logic with a Byzantine Fault Tolerance Consensus Algorithm.
ProstgreSQL is used internally to keep track of the Blockchain state, providing a robust search engine for Blocks, which are stored as independent files in the node; a feature which is uncommon in the Blockchain community.

The user enrollment and authorization in the network is provided by a Role Based permission protocol.

3.4.7 Hyperledger Sawtooth

*Hyperledger Sawtooth* provides a development framework to deploy Blockchain Architectures, supporting a change in the Consensus Algorithm in real time, without needing to stop the network or a redeployment.

The supported Consensus Algorithms are:

- RAFT
- PBFT: Practical Byzantine Fault Tolerance
- PoET: Proof of Elapsed Time

Hyperledger Sawtooth makes it easy to deploy Blockchain platforms in the Cloud with Security and Scalability in mind; due to its ability to process transactions concurrently.

*Ethereum* smart contracts are also supported using the *Seth* (Sawtooth Ethereum) *transaction family*.

3.4.8 Hyperledger Explorer

*Hyperledger Explorer* is a Hyperledger tool that provides the Blockchain Administrator the ability to send transactions, query the ledger, see statistics on how the Blockchain platform is working...

By now the only supported Hyperledger projects, with which Hyperledger Explorer can be integrated are:

- Hyperledger Fabric.
- Hyperledger Composer.
- Hyperledger Cello.
Hyperledger Quilt is the Hyperledger implementation of the Interledger Protocol (also known as ILP), which goal is to be the bridge between different cryptocurrencies, providing an exchange protocol.

Hyperledger Quilt offers a Java and JavaScript API. The reason of existence of the Interledger protocol is because since the appearance of Bitcoin in 2008; many different Cryptocurrencies and Blockchain platforms have appeared; and there is no communication between them, they act as "isolated islands"; if it were not for the Interledger protocol, there would not be even a way of defining the value of a cryptocurrency quantity with respect other cryptocurrencies, and an intermediary would be necessary to validate the value of each of the cryptocurrencies during an exchange.
3.4.10 Hyperledger Ursa

*Hyperledger Ursa* provides a Cryptography library meant to ease the development process. It is made up of two projects:

- Base Crypto: provides the tools to let Blockchain developers implement the transaction and block signing procedures easily.
- Z-Mix: used to create *Zero Knowledge* proofs about a cryptographic module.

3.4.11 Hyperledger Cello

*Hyperledger Cello* is a Hyperledger utility meant to ease the Blockchain Network deployment; offering a GUI (Graphical User Interface) that lets the Blockchain Administrator manage the Blockchain network and deploy new *Chaincode* on the network nodes.

By the time of writing only Hyperledger Composer and Hyperledger Fabric are supported technologies by Hyperledger Cello; the Hyperledger Foundation is looking for supporting the rest of the Hyperledger Blockchain frameworks.

Hyperledger Cello was not used to deploy the network because it is still in Incubation phase and is prone to undocumented errors.

3.4.12 Hyperledger Caliper

As mentioned later in this research, *Hyperledger Caliper* is a Benchmarking tool, that supports Benchmarking Hyperledger Fabric and Composer Blockchain platforms.

Hyperledger Caliper was not used in the Benchmarking process, because it was easier to create a Benchmarking tool from scratch than integrating the tool with the deployed network, due to the development status of the tool which at the time of writing is in "Incubation" process and there is lack of documentation.
3.5 BigchainDB Basics

BigchainDB is a Blockchain platform that pretends to behave like a database. It appeared with the intention to provide storage based in the structure of a NoSQL database, in order to provide a full Blockchain based computing environment; in which the instruction execution would be done with the Ethereum Virtual Machine, and the file based storage would be accomplished using IPFS (Interplanetary Filesystem).

The big difference of BigchainDB with respect a NoSQL Database (like MongoDB, which is the database technology that stores the transactions, blocks and assets of a BigchainDB instance), is that the BigchainDB runs a Consensus Algorithm to ensure that only validated assets are created in the BigchainDB instance.

In order to store the assets, a common database would store them with no concrete order, a timestamp would be rarely used and the asset information would not be checked following a certain validity policy. Moreover, the bare database would only store asset information, instead of creating additional tables for the transactions and blocks.

Its advantages are the throughput it can achieve thanks to its consensus
BigchainDB is a Database in which a consensus algorithm has been configured, reducing the overhead added by other Blockchain technologies like Bitcoin, IOTA, Ethereum... which are purely Blockchain based.

It does not run a PoW (Proof of Work) algorithm like IOTA, the consensus algorithm is lighter and vote based, instead of mining based.

The main use case of BigchainDB is being a platform that helps users share their digital twins.

There are available two different APIs that use the BigchainDB platform: the bare BigchainDB API and the BigchainDB-ORM API, which will be used for our use case in the Decentralized Data Marketplace to support CRUD operations; which are necessary to model the assets as communication channels. This section discusses the BigchainDB characteristics for both APIs, the next section will cover which enhancements are done to the base API in order to provide the CRUD operations.

There are only two types of transactions in BigchainDB:

- A CREATE transaction is sent to create a new asset in the database.
- A TRANSFER transaction is sent to transfer an existing asset to a new user.

Due to its high throughput which outperforms IOTA, BigchainDB will be the choice for data transmission between users in the Decentralized Data Marketplace.

The number of transactions that make up every block is determined by the software that runs the node. In case many transactions are being attached in a period of time, blocks containing more transactions will be created; otherwise the blocks will contain less transactions. It cannot be determined using the API; so that, less freedom for stress testing is left to the developer, because the performance cannot be measured with different Block sizes, which in [7] is discussed as a parameter that can reduce the overhead of writing operations in the Hyperledger Fabric Blockchain, it seems trivial that block size affects every Blockchain technology: the smaller the Block, the more Blocks to validate and add to the ledger.

Each block in BigchainDB can be in one of the following states:

- Undecided: not enough votes from the nodes determined its validity.
Decided Valid: the majority of the votes determined the validity of the block. All its transactions need to be valid.

Decided Invalid: the majority of the votes determined the invalidity of the block. If one transaction of the whole block is invalid, the block is considered Decided Invalid. In this case the block will not be removed from the Blockchain database, the explanation comes later on.

As it is described in [11], BigchainDB handles two ledgers:

- The Transaction Set or Backlog: receives the transactions as they are sent to the Blockchain. Transactions are unordered and can be invalid.
- The Blockchain: contains a chain of ordered blocks with the transactions from the Backlog. It can contain Decided Invalid blocks.

The MongoDB instance of a BigchainDB node has separate tables for transactions, blocks and assets. The transactions database behaves like the transaction backlog, the blocks database contains all the linked blocks and the assets database contains all the created assets in the Blockchain.

There is a difference between invalid transactions and Decided Invalid blocks: Invalid transactions can be due to a malformed transaction structure; on the other hand, Decided Invalid blocks are due to an attempt for Double Spending or an attempt to TRANSFER an asset that does not exist in the Database.

The transactions before leaving the Backlog are checked for simple invalidity tests like the transaction structure. In case they are detected, they will be discarded. Otherwise, if the first validity test is passed, the transactions will be grouped into blocks; and later on the blocks may become Decided Invalid, in that case the block is sent back to the Backlog for another voting attempt.

Nodes can start voting for the validity of a Block only in case all previous Blocks are in a Decided state, to make sure no Double Spending happens.

In case a Block is voted to be Decided Invalid, it will not leave the Blockchain after the second attempt. It may impress the reader, and think that it is a security issue, but it is not, and in fact, is part of what gives BigchainDB great results in scalability and throughput.

An invalid block in the Blockchain database, will not let Double Spending. Whenever a TRANSFER transaction is sent that pretends to cause a
Double Spending (or a malicious user trying to obtain an asset without the owner permission); if it is related to an asset which was involved in an invalid transaction, it will be marked as invalid too; otherwise if the malicious user is trying to obtain an asset without permission of the actual owner, the signatures will reveal her malicious intentions and the block will be marked as Decided Invalid.

The blocks are linked and ordered before voting. The reader may also think that it is another security issue, because a voter could vote for the validity of a block after being linked, and as it is linked to the blockchain, no one could delete it; but it is not, because the votes are attached at the end of the block, and all the votes are signed by its voter. As we mentioned before it is normal to see invalid blocks linked to the Blockchain, so that, Double Spending is ensured not to happen.

The BigchainDB project, has a public repository with BEPs: BigchainDB Enhancement Proposals, which behave like RFCs (Request For Comments) documents.

The community is able to provide modifications to the architecture or software that is part of the BigchainDB framework.

3.5.1 BigchainDB-ORM

BigchainDB is meant for many different use cases, one of them involves the capability of performing CRUD operations (Create, Read, Update, Delete).

In order to support this use case, BigchainDB GmbH has created a different kind of assets (OrmObject) and implemented a different API (BigchainDB-ORM) that permits the aforementioned operations.

BigchainDB-ORM fits perfectly well to the envisioned protocol used to deliver data in encrypted channels, we have used this API instead of the base BigchainDB API for the following reasons:

- Avoid a Channel asset to keep growing forever: this is one of the main problems that many Blockchains are dealing with, one solution is providing deletion of assets containing user data, when the involved parties approve the removal.
- Data Updates on the channel: Even though the base BigchainDB API let the users add data to assets, it has to be added as metadata, the
base data of an asset cannot be modified, which could be a limitation due to the limited amount of data that the metadata field can hold.

- Ease of use: It is easier to perform the desired operations using this API than the base one.

In order to validate a block of transactions, the API provides a condition field that holds the necessary conditions that need to be fulfilled in order to change the block of involved transactions state to Decided Valid. The use of the validation conditions is needed in order to ensure that the involved seller and customer sign and approve the asset deletion (in our use case, the assets represent the channels that the seller and customer use to communicate: send queries, subscriptions and data); only they can sign it as valid; using conditions we avoid other nodes validating the deletion of the channel, avoiding undesired data loss stored in the BigchainDB.

Once the channel deletion is approved between both parties, the Channel OrmObject containing the data is removed, and the prove of existance of the channel is kept thanks to the Channel deletion transaction signed by both parties.

The Delete operation is needed in order to control the growth of the ledger; real-time data could overload the nodes after a certain amount of time due to lack of free storage, because all the nodes of the BigchainDB Blockchain need to have the same copy of the ledger, if it is not for the delete operation, the ledger contains all the transmitted data until that moment.

The Asset data structure needs to be provided to the API as a JSON Schema.

### 3.5.2 Using BigchainDB for data sharing

BigchainDB is not meant for data sharing, so that, an API that models the BigchainDB assets as Data Channels was written to wrap all the function calls that required each operation applied to the channels; it was called MarketOrm.

In order to reuse as much code as possible, after discarding IOTA for data sharing due to bad performance, the functions of the API were the same as the ones in the Masked Authenticated Messaging API.

BigchainDB asset’s structure needs to be described with a JSON Schema, and defined in BigchainDB, in order to do that the BigchainDB-ORM API
uses the `define` function; and the model is added to BigchainDB. Whenever a
new asset is created, the creation function is called on a certain model;
so that, the Blockchain can check if the sent data is in a correct format
according to the attributes of the provided `JSON Schema`.

Channels can be created by calling `createChannel` from `MarketOrm`,
which calls the `create` function from the BigchainDB-ORM API.

Whenever data needs to be sent to a channel, the `publish` function from
`MarketOrm` is used, which calls the `append` function from the `BigchainDB-
ORM API`, and new data is added to the data field of the BigchainDB asset.

One of the drawbacks of the BigchainDB-ORM API is that it does not
provide a `search` function, that lets the developer search for assets with
certain contents; this function is only provided by the BigchainDB base
API. In addition the search function in the BigchainDB base API did not
work correctly: it missed search results in many cases.

The search function is not needed at all; it can be substituted by a func-
tion that gets the channel with a certain id; which will be used to get the
channels with the id that is stored as the QueryChannelId or DataChannelId
(depending on the channel type) in the Hyperledger Agreements. The Mar-
ketOrm function `getchannel` calls the BigchainDB-ORM `retrive` function.
The `getChannel` receives the channelId, which is passed as a parameter to
the `retrive` function.

We have discussed the Creation, Update (appending new data to a chan-
nel) and Read operation; Deleting is not a common operation in Blockchain
technologies, because deletion implies that `non-repudiation` can not be achieved
(there is no way to prove that an event happened in the Blockchain if there
is no written `proof` of it in the Blockchain), in common Blockchain tech-
nologies. In some Blockchain technologies like IOTA, `snapshots` are created:
they contain only the funds balance of all the wallets in the network; the
rest of the Blockchain information is removed and the snapshot is used to
keep the previous state. Other Blockchain technologies like `Bitcoin` that do
not `summarize` the ledger, end up growing its ledger size to a huge amount
of data.

In the case of the `Decentralized Data Marketplace`, the deletion will con-
sist on deleting the `Channel Assets` and creating a `Deletion Request` asset,
which needs to be signed by the seller and the customer of the agreement to
which the Channel is part of (making sure that the customer and the seller
node sign the transaction for deletion can be achieved using the `conditions`
field of the transactions, that define the conditions that need to be fulfilled in order to consider the transaction a valid one), when the Deletion Request is signed by both, it becomes VALID, if they both agree, otherwise, it will be INVALID.

A function that is called when the Decentralized Data Marketplace is executed, monitors all the Deletion Request transactions, and removes all the Channel Assets that were related with that Deletion Request (because both the customer and the seller agree that the data was received and sent, data is assumed to be consumed in a small period of time (in seconds or minutes, but nothing like a day) since it was created by the sensors, because of the real-time nature of the data streams which are updated constantly with data updates).

The Deletion Requests are created after a certain amount of time periodically (when it is not expected that the customer makes use of the data): every day, week, month... The length can be chosen or left to an automatic function that triggers it when the channel length goes beyond a threshold.

3.6 Description of the Distributed Marketplace Application

The Data Marketplace application described in [1] provides its users a platform with which buy and sell data from Internet of Things (IoT) sensors. The devised architecture uses Blockchain technology, instead of a raw Peer-to-Peer architecture without validated consensus between the peers, in order to setup a trustless environment, in which none of its users depends on a central entity that controls the system, and trust is created from the decisions taken by all the peers that are part of the network.

After measuring the performance of the IOTA Blockchain, and all the overhead that public Blockchains add, a research on Peer to Peer systems was taken, in order to find out if there was any framework or technology that provided a lighter consensus algorithm, or relied on a different architecture with respect Blockchains, that could give us non-repudiation and a better performance. A technology that could fit our use case at the same time that could offer the capabilities that our system required was not found, so that, other Blockchain technologies were tried.

BigchainDB was the next choice, which ended up being the platform that was used to transmit data, BigchainDB offers a better performance and all the Blockchain advantages; the only problem with using BigchainDB was the
use case for which it is meant: a database with Blockchain characteristics; which does not fit our use case.

In order to solve that, assets were modeled as Communication Channels and a NodeJS API that had the same functions as the IOTA MAM API was developed, in order to reuse as much as possible the previous Decentralized Data Marketplace code that used the IOTA API.

The role of the Hyperledger Fabric network in the Decentralized Data Marketplace architecture is providing controlled user on boarding (thanks to its permissioned nature), and offering all its users a catalogue with all the available datasets that can be obtained. It is also used to store the payment information when customers buy data and the amount of consumed data in case the pricing model is Usage based or a Subscription.

The user needs to be added by an organization that is already part of the Blockchain network, that is because Hyperledger Fabric offers a Private Blockchain platform; and the User credentials need to be created by an Administrator of the Hyperledger network.

Once she is part of the network, she can create datasets, which contain metadata and descriptions about the information that the IoT sensor offers.

In order to offer the data, she needs to create an offering, which relates to a unique dataset and contains the pricing information. A Dataset is an abstract description of the kind of contents that will be delivered to the customers that buy them. These data flows are delivered in real-time, as data values change (new data values are sensed by the IoT sensors), data updates are sent to the customer in a stream-based flow.

When a user desires to acquire a data flow, an agreement is created between the buyer and the seller; which contains the information about the pricing model that the buyer chose between all the offered by the seller in the offering.

The following transactions are involved in every accepted agreement:

- MakeAgreement: request from the customer to acquire a dataset. The customer indicates in which Offering and Pricing Model from the Offering is interested.

- AttachPaymentInfo: when an offering from a seller is acquired, the seller API receives an event from the Hyperledger network, automatically the Marketplace application, updates the Agreement data with
the seller information, to which the payment needs to be done.

- **PaymentCompleted**: This transaction updates the Agreement with the information from the payment done by the customer. In the working version, the payment needs to be done outside the application, and the proof provided manually.

- **AcceptAgreement**: Once the PaymentCompleted transaction is sent, an event is created by the Hyperledger network, the Marketplace application checks that the payment is correct, checks also the date, the identity of the user and if the payment is related with a previous offering acquisition.

In the case of Pricing Models based on usage and subscription; there is a need to renew the contract, after the validity period has expired in the case of subscription based agreements, or the limit amount of supported data has been reached in usage based agreements. In order to renew the contract, 3 transactions are needed:

- **SettleAgreement**: this transaction contains the desired agreement to renew and a boolean value indicating if the customer is interested in renewing.

- **PaymentCompleted**: this transaction contains a proof of the payment, sent by the customer.

- **AcceptAgreement**: this transaction updates the state on the Agreement to ACTIVE.

This renewal process is started by the customer, if she is not interested in renewing it, there is no need to send any of the aforementioned transactions; the Hyperledger network will change the status of the agreements to PAYMENT PENDING, and whenever new data is requested, the Application will check the agreement state field in the agreement, in the case it is not ACTIVE, no answer will be given to the user; further actions may be considered in case of the user tries to request data when he has no valid agreement, as notifying her with an error banner or banning the user for a period of time in case she tries to gather data regularly.

The platform supports payments using IOTA transactions; in further versions it will support payments using fiat currency transactions and payment gateways (like PayPal), since the protocol is prepared to support them.
Figure 5: The Decentralized Data Marketplace Infrastructure.

The state of the agreement determines if the buyer is able to receive data or not having into account if the payment is valid.

Once the customer desires to receive data, she will create a subscription, in case she wants to be notified on data changes asynchronously, or will make a query, which supports a synchronous data exchange.

In [1], a formal description of the architecture, transaction flows and data structures is provided; the point is offering the user an implementation, but also leaving the freedom to the user to write their own implementation following the provided standard.

By now, there are many different Decentralized Data Marketplaces products out there in the market; they all offer pretty similar solutions, and all of them are in a Proof of Concept/testing status. By the time of writing, there is no widely used solution.

The proposed solution in [1], (which implementation is tested under stress benchmarks in this research), provides a different use case to the other State of the Art solutions: the point of this data marketplace is to provide users the capability to sell and buy streaming-based datasets instead of static ones.

In this context the concept of dynamic data is based on letting the
customer access data dynamically generated, by using queries, but also in an asynchronous Subscription based way, receiving notifications as the data changes.

This solution opens the door to future real time IoT and multimedia use cases in which users could stream and share big amounts of data in real time.

3.7 Vegeta HTTP Load Testing

Vegeta apart from being a character in the Dragon Ball fiction universe; is also the name of an HTTP Load Testing tool written in the Go programming language.

The Vegeta HTTP Load tester can be used to measure the performance of any HTTP REST API; the tool receives a file which contains all the HTTP requests that are desired to perform during the tests, as well as the duration of the tests and the rate at which the requests will be generated.

The results can be sent to an output file in different formats, which can be then processed by the tool to generate plots that show the latency that took each request to be processed.

The Vegeta tool was useful to measure the performance of the whole architecture when performing queries at different rates.

The Vegeta HTTP Load Testing utility is resilient, simple, tunable and easy to use. It is made up of 4 main commands:

- Attack: this is the main command; initiates the testing procedure. It has many flags and receives all the parameters to do the desired test, the main ones are the following:
  - -duration [duration]: It sets the duration of the attack in the form: [0-9]+[m—s—ms]. 0 makes the test to run until killed.
  - -rate [rate]: It sets the rate at which the desired HTTP requests will be sent, in the form: [0-9]+[m—s—ms].
  - -targets [targetFilepath]: It sets a file which contains the desired requests to be sent. The following is the target.txt used to achieve the query testing in this research:

    POST http://localhost:1031/v1/queryContext
The aforementioned example is basically a POST operation to the URI: `http://localhost:1031/v1/queryContext`, using the Content-Type header with value: "application/json" and the Fiware-Service header with value: "org1", the Body contents are taken from the file "queryBody.json" located at the same folder in which the command is executed ("@" is used to load a file).

- -output [output]: file where the results from the test will be saved.

- Encode: offers the capability to change the format of an output result file. It has the following flags:
  - -output [output]: sets the output file.
  - -to [format]: sets the desired format to convert the input file to, by default is set to json; it also supports: csv and gob.

- Plot: this command is used to generate graphs with the results of a test.

- Report: this command summarizes the results of common Benchmarking metrics using the results file from a test. It is interesting to remark the meaning of its flags:
  - -type [type]: specifies the desired format for the results: it supports formatted plaintext, JSON and histogram. The user can set the histogram segments, they will mark the amount and percentage of packets that took this latency to be processed.
  - -every [duration]: the results can be refreshed after a certain amount of time, letting the user to know the performance details at different points in the test.

### 3.8 Communication Protocols using the Tangle

One of the advantages of the IOTA Blockchain is its feeless transaction capability (the feeless nature is a reality because there is no mining in the network, each user who desires to send a transaction, needs to validate two unvalidated transactions (tips) before being able to send his transaction);
which provides its users the capabilities of sending monetary transactions without needing to spend monetary fees to provide rewards to the network, making small monetary quantities exchange one of its most common use cases.

Apart from monetary transactions, the IOTA Blockchain supports transactions that contain no monetary quantities, so that, it can be used as a communication platform between users.

The most common APIs and frameworks used to provide data communication over the IOTA Blockchain are discussed in the following subsections:

### 3.8.1 MAM: Masked Authenticated Messaging

The Masked Authenticated Messaging is one of the most common IOTA use cases, to let Machine-to-Machine (M2M) communication and IoT Data sharing.

The MAM is the IOTA Foundation official implementation of the aforementioned Communication Protocol over the IOTA Blockchain.

This protocol stores the messages using the Merkle Tree data structure [3]. The message’s data is stored in the signatureFragment field of IOTA’s transactions, which is encrypted.

The ”Communication Channel” is created by linking transactions that contain data; so that, a user only needs to know the address where the first message of the channel is stored and the key or keys (depending on the used permission mode) used to protect the contents of each message, to write and read from the channel.

It is not possible to overwrite or modify the contents of a message, in the case that this happens, the channel gets corrupted and it can no longer be read or written. In addition, messages can not be deleted, because they are IOTA transactions stored in the Blockchain.

The protocol implements three permission modes to access the channel contents:

- Public: in this mode the address where the message is stored is calculated by hashing a private key (root), all the users that know the root can write in the channel. As mentioned before even though the message contents are encrypted, the password to decrypt it, is the same
as the address in which it is stored, so that, if the user knows the address of the message (the IOTA ledger as the rest of cryptocurrencies is public, so that, all transactions can be seen by anyone), she can see its contents.

- **Private:** in this case the contents of the message are encrypted using a different password, only the users that know this password can decrypt the message.

- **Restricted:** this mode provides the capability of splitting channels to create subchannels. Restricted mode uses the sideKey, which can be changed whenever the owner of the channel (who writes in the channel) desires to avoid the consumers (those who read the channel messages) of its channel to keep seeing the contents of the messages.

One of the disadvantages of this Communication Protocol implementation is that, to append data to a channel, the API needs to traverse the whole channel (each message points to the next message), to get the last message and finally append to the last message the new one, setting the previously last message the address of the new last message.

### 3.8.2 MAIA: Masked Authenticated IOTA Address

This product is not meant for communication, but for making IOTA a zero knowledge coin, using the Masked Authenticated Messaging channels as IOTA wallet addresses.

One of the problems of the cryptocurrencies that do not provide the zero knowledge feature is that once the address is used to send tokens to other accounts, the address stops being secure (does not follow the one time pad requirements), because part of the Private Key can be deduced after each signed transaction (and all transactions are public); the reason behind this problem lies in how the Public Key Infrastructure (PKI) works: the public key is created from the private key, so that, each time a new transaction is signed, the private key is involved in the encryption of the transaction.

There are no security issues (regarding reuse of the private key) if an account is just used to receive funds from other users, as far as the private key is never used to sign a transaction (send funds to other users).

Almost all the cryptocurrencies are not zero knowledge; so that, the wallet software client will create a new key pair after each transaction made
(each time the user sends funds to other users), this new key pair will hold all the funds that were left in the previous account after making the transaction. So that for every single transaction made in the network, two will occur, the second one moving the left funds of the sender to her new wallet.

This software wallet feature lets the user to not worry about creating new key pairs and moving her left funds to it, but the user needs to make sure that none of her contributors (users who will send funds again to this account in a while) use the old address anymore and switch to the new one, which implies notifying them with the new wallet address.

MAIA handles this problem by using a MAM Channel as an IOTA address. As mentioned before, MAM Channels can contain several messages, each time a new one is added, the address that holds that message is a different one, so that, instead of an IOTA Address, users can provide the root address of her MAM Channel.

3.8.3 RAAM: Random Access Authenticated Messaging

RAAM provides a solution in which the message positions in the RAAM channel can be accessed without needing to traverse the whole Merkle Tree, so that, the complexity to read or write any of the message locations in the channel is $O(1)$, improving the Algorithmic Complexity of MAM Channels which have an $O(n)$ complexity to insert data in the channel, being $n$ the number of messages in a MAM channel.

This comes with the drawback that the amount of messages that can be written in a channel is predefined during the creation of the channel. The time that takes to create the channel is also longer than in the previous solutions, as all the addresses that will hold the messages need to be created at once. Because of that, the time that takes to create the channel will depend on the desired capacity of the channel.

Even though the use case of RAAM is being able to access message positions with $O(1)$ complexity, to accomplish that, channels need to have a fixed length (messages need to be linked before the channel can be used). These channels can be extended by linking the end of one to the beginning of another, solving the problem of the fixed capacity; by the time of writing this feature has not been implemented in the RAAM API yet, and will require the developer to link channels manually.

RAAM should not be used if the IOTA channel is meant to provide an
unfixed capacity; because linking many channels will affect the complexity to access any position in the channel, being the worst complexity: $O(n)$, being $n$ the amount of linked channels.

This solution also supports the Private, Public and Restricted access modes; supported by the official MAM implementation.
4 Deployments

In this section, an explanation on how to deploy each of the software components is provided. It is not trivial how to deploy the whole Decentralized Data Marketplace architecture; it was necessary to deploy the system in order to measure the performance.

At first, it comes an explanation on how to deploy each of the tested technologies; the last of the sections of this chapter will contain a docker-compose YAML file that will deploy one node of the Decentralized Data Marketplace; with all the required services.

4.1 Deploying an IOTA node

Even though IOTA has been substituted by BigchainDB; this chapter describes how the IOTA node was deployed to assess its performance. The described process to deploy an IOTA node will also work in case the reader only wants to have an IOTA node and is not interested in deploying the whole Decentralized Data Marketplace infrastructure.

The process to deploy an IOTA full node requires the following steps:

1. Getting the whole ledger of the IOTA network (if a light node is desired, only the last snapshot is needed).
2. Establish neighbor relationships with other nodes in the network.
3. Tuning the configuration of the node as desired.
4. Running the IRI software (IOTA Node Software).

The deployment was done using Docker. To fine tune the configuration of the IRI, a .INI file that is read by the IRI, is used.

The neighbours need to be added manually to the config.INI file, the IOTA Foundation is working on a solution to automatically establish the adjacencies between the peers in a secure way.

The neighbors can be found in the Discord IOTA channel, in the fullnodes section. Everyone can join the Discord IOTA channel by accessing: https://discordapp.com/invite/fNGZXvh.

This is the used config.INI file:
The Docker container was run using the following command:

docker run -d -p 14265:14265 -p 15600:15600 -p 14600:14600/udp -v 
/home/afuentes/iota_ledger:/iri/data --name iri 
 iotaledger/iri:latest latest --remote -p 14265 --config 
 config.INI

This command expects to find the config.INI file in the same directory in which it is run.

The IRI requires to expose 3 ports: 14600 and 15600 are used to communicate with the neighbours, in case an UDP communication is desired the port 14600 should be exposed, on the other hand the port 15600 is used to establish TCP connections with the neighbours. Finally the port 14265 is used to communicate with the IRI API.

The IOTA node was deployed in gemini.ls.fi.upm.es, which has the following hardware specifications:

- Processor: Intel(R) Xeon(R) CPU L5410 @ 2.33 GHz
- RAM Memory: 48 GB DDR2 @ 667 MHz
- Disk Space: 2 TB
Even though, these specifications are considered high quality by today’s standards, the performance that the IOTA node offered during the tests is unacceptable for the project needs: it will be seen during the “Results” Chapter, that all the cores achieved a 100% occupancy during the IOTA tests.

4.2 Deploying a BigchainDB Network

As mentioned before in this research, BigchainDB was chosen as an alternative to the IOTA Blockchain, due to bad performance results with IOTA’s technology; so that, BigchainDB assets will model communication channels between peers.

The Performance of BigchainDB was tested without using the BigchainDB-ORM API; the base API was used, as the performance of the system is the same one for each of the APIs, because they run on the same containers, and the ORM API is an extension that uses the same transactions, but adding a different meaning to them, in order to achieve the CRUD operations support.

In order to test the BigchainDB performance, a private BigchainDB network with one node was created. The following steps were used to achieve that.

$ git clone https://github.com/bigchaindb/bigchaindb
$ cd bigchaindb
$ make run

4.3 Deploying a Hyperledger Fabric Network

This subsection discusses the procedure to deploy a Hyperledger Fabric network. A Hyperledger Fabric network is used in the Decentralized Data Marketplace architecture to store all the Business information (agreements set by customers and sellers, pricing models offered by the sellers of their datasets, the payment information provided by the customers...).

The process used in this section will let the user deploy a simple network that can be used for development and testing purposes; it will only deploy a Peer, an Orderer and a Certificate Authority, which are the main components of a Hyperledger Fabric organization. This deployment only sets up
By the time of writing, and following the official *Hyperledger Fabric* docs, Hyperledger Fabric can only be installed on *Ubuntu 14.04 or 16.04 LTS* or a *Mac OS 10.12*.

It is recommended to avoid the *Fabric v1.1* which was much more error prone that the *Fabric v1.2*, in which all the tests were done.

*Hyperledger Composer v0.19* works with *Hyperledger Fabric v1.1*, and *Hyperledger Composer v0.20* does not work with Fabric v1.1, but with Fabric v1.2. By the time of writing the *Hyperledger Composer* docs has not been updated to the v0.20 version. It is necessary that the developer makes sure that a compatible version of Fabric is being used with Composer; otherwise, errors will arise during the deployment of the network.

The *Hyperledger Composer v0.19* docs are wrong and contain errors; the commands used to copy the certificates will not work, the issue and its solution is discussed in the Appendix B, as well as other common *Hyperledger* errors and their reasons.

All the issues with the Fabric v1.1 were solved in v1.2, to set up the same deployment on this newer version the following commands are used:

```
$ git clone https://github.com/hyperledger/composer-tools
$ cd ./.composer-tools/packages/fabric-dev-servers
$ export FABRIC_VERSION=hlfv12
$ ./downloadFabric.sh
$ ./startFabric.sh
```

If the *Hyperledger Fabric* network is not setup correctly, the *Hyperledger Composer* network will not be able to be setup and the following error will be thrown when trying to install the network with the *composer network install* command:

```
Installing business network. This may take a minute...
Error: Error trying to ping. Error: No peers available to query.
last error was Error: 14 UNAVAILABLE: Connect Failed
Command failed
```

This error may be thrown due to different reasons too.
4.4 Data Marketplace User Onboarding

Even though, the users deploy their own node with all the required services, they also need to be registered in the Decentralized Data Marketplace, in order to be able to use it.

The procedure used in order to let users enter the network is similar to how Let’s Encrypt automates the Certificate generation, the process is the following:

1. The user creates an SSL certificate; openssl can be used to generate it.
2. The user signs the certificate with the organization’s certificate that he is part of.
3. The user sends the signed certificate to the Network Administrator.
4. The network administrator checks that the certificate of the user is signed by a verified organization (organization that has entered the network before the user).
5. The network administrator creates a Hyperledger Composer participant and Identity for the user.
6. The user receives the cardname details with which he can connect to the network.

This mechanism has too many similarities with a common Certificate Services Architecture. The network administrator behaves like a Root CA, the organization in which the user is enrolled behaves like a Certificate Authority that is subordinated to the Root CA.

4.5 Decentralized Data Marketplace Node

This section discusses how to deploy a whole node of the Decentralized Data Marketplace; even though, it is not mandatory to deploy all the services to be part of the Decentralized Data Marketplace, because an organization is not forced to deploy an Orderer, as they can depend on other Orderer nodes deployed by other organisations.
On the other hand, the Peer node is mandatory; this service is responsible for receiving the transactions sent by the users of an organization to the Hyperledger Network; without that service, there would be no way to process the transactions sent by the users of an organization.

Even though, no Firewall service is being deployed in the docker-compose file as a service, it is left to each organization to choose which kind of protection they desire.

The redis service is used to store the jobs related with the transactions that create assets in the Hyperledger Network, in an asynchronous way.

The mongo service is used by the BigchainDB service, but it is also used to store the Seller details (wallet, payment method and cardname), as well as the User’s information (username and password used to get a token by the API and the user and password used to authenticate the user in the BigchainDB network).

The whole Docker-Compose YAML file to deploy a node of the Decentralized Data Marketplace can be found at Appendix L.
5 Benchmarking Definition and Setup

This section will discuss the benchmarking environments definition and setup, but before the procedure to benchmark each of the studied technologies is shown. First of all, a research needs to be done on the *State of the Art* of the Benchmarking methodologies applied on each technology.

Later on, the metrics desired to be measured and the reproduced scenarios need to be determined, to make sure the experiments can be repeated in further tests with the same conditions.

The experiments do not need to be like any previous research ever done, because the point of the benchmarking may be different to what other researchers wanted to find out; even though, it is recommended to follow common practices to provide a fair comparison between other researches and products.

Each node of the *Decentralized Data Marketplace* is meant to provide service to hundreds of users of an organization.

It was expected that an IOTA transaction would take no more than 5 seconds to be attached to the Tangle. In required computing resources terms, it was expected that only around a 15% of the capacity of the processor was used, instead of almost the 100%. In terms of how much RAM running the node would require, it was expected to be around 4 GB; which are the recommended requirements determined by the IOTA community, that is 8.33% of the used RAM in the tests.

The expected performance of the Hyperledger Fabric Blockchain, in order to fit the project needs, was that a transaction could be appended in no more than 3 seconds.

One of the unexpected results after testing Hyperledger Fabric, was that it is vulnerable to a *Denial of Service* attack, in which a malicious user floods the network with plenty of transactions; many more than the amount of transactions that the network can process before the network starts discarding transactions because they stay a lot of time waiting to be processed. Once this happens, the network is unable to recover normal activity and will keep discarding transactions for a long time. In the "Further Work" Chapter, a solution to this issue is proposed.

Finally, the expected performance of BigchainDB was accepting transactions with around 10 requests per second. The results were much more
than promising.

In terms of availability, all the technologies except IOTA were promising. There were many issues with the IOTA node during the tests; the node unsynchronised itself with the network for no reason; one of the requirements for the Decentralized Data Marketplace, is being resilient during high activity periods; the Blockchain technology is meant to provide resiliency and reliability of the stored data in the Blockchain.

5.1 IOTA

At the time of writing, no public and documented research on the performance of IOTA has been done, without having into account the performance values provided by the IOTA Foundation in the IOTA whitepaper [4].

As the Distributed Data Marketplace uses the MAM Channels that run over the IOTA network, the performance can be measured using bare IOTA transactions or using each of the different Data Communication protocols mentioned on the ”Communication Protocols using the Tangle” subsection of the ”State of the Art” Chapter.

As the MAM Communication protocols add an overhead because of the generation of the Merkle Tree and the linking of each message with the rest of the messages in the channel, which require several Hashing operations to ensure the integrity of the Merkle Tree, and the author of this research was not sure about the IOTA network offering a good enough performance for the project, after having used the IOTA Blockchain for a while; the tests were applied on bare transactions; these results provide a threshold that cannot be improved by any of the aforementioned communication protocols.

The first tests were done using public IOTA nodes; which cannot be used to stress test the IOTA network, because the public IOTA nodes have a rate-limit mechanism; the nodes register the IP of the devices that send the requests for attaching a transaction. When a threshold of permitted transactions per second is reached, the IOTA node will notify the user about 'Too many requests' with the following error code when trying to send more using the public node from the same IP device:

```
 iota.adapter.BadApiResponse: 429 response from node: {'timestamp': '2019-01-21T10:29:27.628+0000', 'error': 'Too Many Requests', 'path': '/', 'message': 'Rate limit reached or no node available. Try again later.', 'status': 429}
```
So that, it is clear that a private node needs to be used to stress test the IOTA network, because as mentioned before, only a little amount of transactions can be attached to the IOTA network using a public node before the error message appears, this amount of transactions is not enough to stress test the network and get performance information.

The first approach to better understand how the performance of the IOTA Node changed under different traffic loads, was tuning the amount of transactions in each bundle, to find the amount of transactions to introduce in each bundle that offered the best performance results. After a couple of tests it was clear that tuning the amount of transactions per bundle was not going to improve or make worse the performance; because the IOTA Node validates a pair of transactions per transaction that is desired to append, and not per bundle that is desired to append; so that the node will validate the same number of transactions before inserting a transaction; so that, distributing the transactions in more bundles or grouping many of them in one bundle will not reduce the Algorithmic Complexity of this process.

The parameters that can be tuned to perform different tests are:

- Number of transactions sent per spawned thread.
- Number of threads to send transactions in parallel (1 thread will send them sequentially).

The metrics that will be measured are the following:

- Average time to attach a transaction in the Tangle as a tip.
- Latency to attach: time between the transaction is sent and when it gets attached in the network.

Even though the simulated scenario is simple, it is the desired one, because as IOTA is a public Blockchain technology, bursts of transactions may appear at any moment because of many different reasons and the common user behaviour can not be represented precisely, as it will not affect the whole network behaviour. When studying the Hyperledger performance, the user behaviour is determined.

The transactions sent to the IOTA network had the following contents:
The amount of transactions per bundle can be chosen by the user, as a parameter of the Benchmarking tool, which code is in Appendix E.

5.2 Hyperledger Benchmarking

5.2.1 State of the Art

In the case of Hyperledger, many previous researches have been made before [5-8].

[5] studies the latency and throughput during Maximum Load tests (in which Hyperledger Caliper is tuned to send different amounts of transactions per second), in different scenarios, studying how the results change when:

- Increasing the number of peers in the network.
- Increasing the number of channels.
- Increasing the number of Smart Contracts.
- Applying more restrictive Endorsement Policies.
- Changing the amount of sent requests.


[7] models the Consensus Algorithm of Hyperledger Fabric, using a Stochastic Model; the researchers find out that the performance of a Blockchain is affected by the Block size; making bigger blocks reduces the number of validating operations, because more transactions get inside of each Block; even though, it is not the best solution in Blockchains that do not receive too many transactions. Increasing the size of the block, may be counter productive in case there are so many invalid transactions; all the transactions, except the invalid one need to be validated inside a new Block later on.
[8] analyses the Hyperledger Fabric Consensus Algorithm in depth: studying each of the small phases that takes a block to become validated. They provide an improvement on the Consensus Algorithm that can increase the whole throughput by 16 times. The research paper also studies how the chosen database to store the assets, the Block size and the endorsement policy affects the latency and throughput.

In addition to previous attempts to determining the performance of Hyperledger software, the Hyperledger Foundation has a Working Group dedicated to determining common measures and benchmarking features for Blockchain technologies, not just related with Hyperledger products; because there is a business need to know what is the overhead added by the Consensus algorithms used by Blockchain technologies.

In order to ease the Blockchain Benchmarking process, the Hyperledger Performance and Scale Working Group has also developed a tool called Hyperledger Caliper, for assessing the Blockchain performance, this tool is actually used in [5], [7] and [9]. The Hyperledger Caliper tool has been the base of custom tools for assessing other Blockchain technologies, like the Benchmarking tool used in [9], created by Huawei, that improves its capabilities adjusting the Hyperledger Caliper code to fit other testing cases. In the case of [9], the tool was modded to assess the performance of the Quorum Blockchain.

All the existing researches on Benchmarking the performance of a Hyperledger Fabric network are based on using simple Smart Contract logic that just read or store a numeric value.

The advantage of doing tests with simple Smart Contract logic is that they can be considered atomic instructions, so that, after the tests are run, metrics like "number of reads or writes per second" can be easily given, and are useful for any business cases, as any Blockchain platform will need read and write operations in their Smart Contract logic.

In order to know how is the performance of the Decentralized Data Marketplace, common user behaviours were defined, to check if the used Hyperledger platform can support the required amount of traffic.

Even though, the performance tests are focused on understanding if Hyperledger Fabric is suitable for the project needs, and the tests are not focused on how many atomic operations can the Hyperledger Fabric network process offering a good performance; the performance of the whole Smart Contract execution with our business logic, was measured; and it was clear
that the biggest overhead came from the creation of agreements: the logic needed to calculate the final price in the `MakeAgreement` Smart Contract, and the fact that an Agreement needs 3 transactions to become a Valid Agreement (active).

In the *Hyperledger Blockchain Performance Metrics* whitepaper written by the Hyperledger Foundation, it is also suggested that the Benchmarking tests should contain operations that should fail, in order to check if the system supports common traffic loads when malicious or malfunctioning nodes are part of the network and failing operations do not cause a big overhead in the nodes. These kind of operations were not introduced in the Benchmark testing of the network, because some requests made by hand with malformed data were done during the development of the Decentralized Data Marketplace and the time it took to process them was almost the same as if the requests did not contain malformed data.

### 5.2.2 Common use of the Data Marketplace

In order to understand the real requirements of the platform, and perform tests that simulated a normal day-to-day basis, an estimation was done in terms of how many *assets* and *transactions* would be created in a month; from that estimation, the rate of transactions per second that the network needs to put up with, can be obtained.

Defining a day-to-day basis use of the platform resulted in knowing the rate of how many Agreements, Datasets and Offerings would be created.

The testing environment will be made of an organization; which contains the following components: an Orderer, a Certificate Authority and a Peer.

The amount of each type of agreement is estimated having into account the use case scenarios for which the Decentralized Data Marketplace is meant; the platform takes advantage of the Subscription and Usage based agreements, because they provide an asynchronous behaviour.

Each offering is associated with a dataset. Each user can be at the same time a seller and a customer. The common use of the network will be the following:

- 70% of the peers will only produce data.
- 20% of the peers will only consume data.
• 10% of the peers will consume and produce data in one of the existing pricing models.

• 10% of the agreements will be One-time based.

• 50% of the agreements will be Subscriptions.

• 40% of the agreements will be Usage based.

• In average for every Offering, 4 Agreements will be created. Because each Offering is linked to one and only one Dataset, for each Dataset an Offering will be created.

• The common package of data will be of 128 Bytes.

• A common subscription will send 4096 updates to each subscribed user a month.

• A common usage based contract will be of 4096 B a month.

• Each accepted agreement will require 4 transactions (the complete data flow is explained in [1], the meaning of each transaction is explained in the ”State of the Art” Chapter) to the business ledger:
  
  – MakeAgreement
  – AttachPaymentInfo
  – PaymentCompleted
  – AcceptAgreement

• The renewal of each agreement will require 3 transactions:
  
  – SettleAgreement
  – PaymentCompleted
  – AcceptAgreement

From the aforementioned quantities, the normal user behaviour can be obtained: as we can see, there will be 4 times more agreements created than offerings or datasets (the amount of offerings and datasets that will be created in a regular basis is the same, because an Offering is linked to just one Dataset). So that, to simulate the normal user behaviour, during the tests for each Offering, 4 Agreements will be created and for each Dataset an Offering.
5.2.3 Benchmarking Tool

The *Hyperledger Fabric* Benchmarking environment has been developed in NodeJS; because the native Hyperledger Composer API is written in NodeJS too.

As there are many different possible testing scenarios, the interest lied in creating a flexible platform, that could be used to simulate any kind of traffic.

To measure the performance, *Hyperledger Composer events* were used. An event as described in the Section 3.4.2, is used in the Benchmarking tool, in order to know when a new transaction is appended to the Blockchain.

A listener is started before sending the transaction, when an event occurs in the Blockchain network, the listener code is executed. Before sending the transaction a timer is started, and inside the listener code, a new timestamp is taken; a `console.log()` is then executed showing the difference between both timestamps, which actually is the time it took the Network to validate the transaction and append it to the Blockchain.

The first version of the Benchmarking tool used the listeners from the Data Marketplace Client Application, in order to reduce the amount of created listeners, so that the Data Marketplace Client Application was also needed to be running.

As it is mentioned in the following paragraphs, the Benchmarking tool crashed at some point due to a big amount of listeners causing a heap overflow; because of that a new version of the Benchmarking tool only used one listener, started at the beginning of the program, its logic checked for the type of event in order to log the right transaction type, and did not use the Data Marketplace Client Application; direct Hyperledger API calls were done instead.

The Hyperledger Composer framework is aware of Listeners, and when a big amount of them is created (more than 10 listeners), warns the developer on possible memory leaks with the following output:

```
(node) warning: possible EventEmitter memory leak detected. 11 connection listeners added. Use emitter.setMaxListeners() to increase limit.
```

As the output also suggests, if the developer needs more listeners to
be deployed, it can be changed using the `setMaxListeners(NUM OF MAX LISTENERS PERMITTED)` function.

During the development process, the number of listeners can be reduced dramatically; because with just one listener, the developer can catch any kind of events, and inside the listener code, he can use conditional programming, checking for event field values; instead of creating a different listener for each of them; leading to more efficient code.

The Benchmarking tool was designed as a `Command Line` tool, which receives the following flags and proceeds creating the desired amount of assets:

- `-d`: Number of Datasets to be created.
- `-o`: Number of Offerings to be created.
- `-ao`: Number of One Time Payment Agreements to be created.
- `-au`: Number of Usage Based Agreements to be created.
- `-as`: Number of Subscription Based Agreements to be created.

The Benchmarking tool generates a file in which the Elapsed time since the start of the execution and the time it took to validate each of the transactions sent is logged. After that, the log file is parsed by a Python program (which source code is at Appendix I) that calculates the metrics values. The following is an example with all the possible log events:

```
Created a new Dataset in: 2323
Acquisition created in: 2387
Created a new Offering in: 1317
Payment Completed in: 2381
Accepted Agreement in: 2373
Elapsed Time: 621667
```

After the validation of each transaction, the elapsed time to create each asset is logged too. All the logged values are in milliseconds. The "Elapsed Time" shown in the output refers to the amount of time that passed since the beginning of the execution of the test. Each of the rest of the values are the times it took each transaction to be processed.

By the time of writing, due to the development state of the `Decentralized Data Marketplace Client Application`, the only supported Agreement flags
are ”-ao” and ”-as” because the Application does not support Usage Based Agreements (“-au”) yet.

All the Assets created had the same contents; they were taken from In memory JSON files. In order to differentiate the transactions that create these Assets, a random id was created for each of them using the uuidv4 node module.

The transactions need to be different, if two equal transactions are sent to the Blockchain the Consensus Protocol detects them as an intentional Double Spending Attack, and the following error message is shown:

```plaintext
Error trying invoke business network. Error: Peer localhost:10051
has rejected transaction
'0af07e0785e27ccc950718250f1c4dec0f3ba223dab7cabc697fa74a28c2156f'
with code MVCC_READ_CONFLICT
```

In order to solve that, the asset id which is contained in the Asset Creation transaction is randomly generated using the uuidv4 module, thanks to that, every transaction is different from the rest, and the tests can be automated.

All the results from the tests can be found in the ”Results” Chapter; it is interesting to analyse something interesting that happened during the execution of the third test:

As time passed, the assets needed much more time to be created than at the beginning of the Test. The following is part of the logs generated by the benchmarking tool:

```plaintext
Created a new Offering in: 139641ms
Accepted Agreement in: 49185ms
Created a new Dataset in: 155172ms
Created a new Offering in: 145028ms
Payment Completed in: 39485ms
Acquisition created in: 72060ms
Created a new Dataset in: 138539ms
Created a new Offering in: 145000ms
Accepted Agreement in: 67245ms
Created a new Dataset in: 164254ms
```

In comparison with normal values at the beginning of the execution:

```plaintext
Created a new Dataset in: 4061ms
```

56
At some point the program stopped and showed the following output:

```
FATAL ERROR: CALL_AND_RETRY_LAST Allocation failed - JavaScript heap out of memory
1: node::Abort() [node]
2: 0x8c21ec [node]
3: v8::Utils::ReportDOMFailure(char const*, bool) [node]
4: v8::internal::V8::FatalProcessOutOfMemory(char const*, bool) [node]
5: v8::internal::Factory::NewFixedArray(int, v8::internal::PretenureFlag) [node]
6: v8::internal::JsonParser<true>::ParseJsonArray() [node]
7: v8::internal::JsonParser<true>::ParseJsonValue() [node]
8: v8::internal::JsonParser<true>::ParseJsonObject() [node]
9: v8::internal::JsonParser<true>::ParseJsonValue() [node]
10: v8::internal::JsonParser<true>::ParseJsonObject() [node]
11: v8::internal::JsonParser<true>::ParseJsonValue() [node]
12: v8::internal::JsonParser<true>::ParseJson() [node]
13: v8::internal::Builtin_JsonParse(int, v8::internal::Object**, v8::internal::Isolate*) [node]
14: 0x1a8f8eb0697d
Abortado ('core' generado)
```

A core file was generated:

```
<--- Last few GCs --->
[12305:0x21eb710] 2050152 ms: Mark-sweep 1285.3 (1438.5) -> 1285.2 (1438.5) MB, 1392.6 / 0.1 ms allocation failure GC in old space requested
[12305:0x21eb710] 2051251 ms: Mark-sweep 1285.2 (1438.5) -> 1285.2 (1431.5) MB, 1098.5 / 0.1 ms last resort GC in old space requested
[12305:0x21eb710] 2052321 ms: Mark-sweep 1285.2 (1431.5) -> 1285.2
```
What really happened was a heap overflow; which was caused by the immense amount of listeners generated. Each module of the Application client contained the code to create a listener each time the function was called, and they were never manually closed.

This bug could not have been found if it were not for the stress test, because during the development of the Data Marketplace, the tests were done manually using REST API calls with Postman.

After this issue, the design decision was to only leave one listener to handle each event happening in the network. The Hyperledger Events contain information about the assets; and they are useful for testing purposes, because it is the only way to debug the Smart Contracts: by returning assets’ data, as well as for systems working in production environments, in order to make the system as much asynchronous as possible.

The developer needs to carefully determine which information is shared between the client application and the smart contract, due to confidentiality reasons; it is desired to introduce the least information as possible; in order to avoid users seeing confidential business information of other users or enterprises.

One of the uses of these listeners were letting the user know when a new offering owned by them was bought in the dataApp.js module.

The issue on the confidentiality was known much before performing the
stress testing, but the fact that creating listeners and not closing them manually leaked the memory was unknown until this moment.

In fact during the tests, it was clear that creating listeners took around 1 second.

Because of the fact that NodeJS is a single-threaded programming language, meant for asynchronous systems; it is not possible to deploy different threads; instead Promises need to be handled.

To simulate the Maximum Load Tests in which the transactions are sent in a certain given pace; there were two options:

- Using CronJobs.
- Using setInterval().

setInterval() was used because Cronjobs included a little delay that gave imprecise results.

The point of executing Maximum Load Tests is to know at which point the system starts losing transactions (not validating them); they get not validated and an error is thrown by the Hyperledger framework because, once the transactions are received by the node, an internal timer starts per transaction; if the transaction cannot be validated before the timer ends, then the transaction is dropped and lost.

The reason behind using timers to determine the amount of time a transaction can wait inside the "to be validated queue", is much more a security reason than a software feature. It avoids the overloading and crashing of the system, that can be accomplished by bad actors trying to cause a Denial of Service to the Blockchain.

Once the node starts dropping transactions, it is a symptom that the node will keep losing them and not recovering at all a 100% transactions validation. Only in one of the tests (which results are covered in the "Results" Chapter), had long periods of time in which it validated all transactions again.

The reason why this happens is basically that the Hyperledger Fabric framework does not seem to have a Congestion Prediction algorithm to avoid overloading; so that, it needs to be implemented as part of the Decentralized Data Marketplace Client Application.
Figure 6: Capture 1 of \textit{htop} during the execution of the test in which every 55 ms an asset was created.

Figure 7: Capture 2 of \textit{htop} during the execution of the test in which every 55 ms an asset was created. The capture was taken before the node started to lose transactions.

The Hyperledger Fabric framework does not seem to have a resource scaling algorithm either. During the tests, \textit{htop} captures were taken, and it was clear that never all the cores got 100\% CPU load. Some of the captures taken during moments in which the Hyperledger node dropped transactions can be seen in the following Figures [6-9] (these captures are from the Test in which during 10 minutes, every 55 ms a new asset was created).

If we closely look at the generated logs, we can see that at some point when the node starts to drop transactions, it gets space for more, and during a little period of time, it comes back to normal behaviour, which does not

Figure 8: Capture 3 of \textit{htop} during the execution of the test in which every 55 ms an asset was created. The capture was taken when the node was losing many transactions in a row.
last too much and comes back to dropping them iteratively.

Despite the node dropping transactions in high traffic loads periods, the Hyperledger framework is the most resilient of the tested Blockchains in this research (IOTA, BigchainDB and Hyperledger Fabric). The nodes kept running without Docker Health issues. Resiliency needs to be the most important feature of a Blockchain Platform, otherwise it can not be considered Blockchain at all.

5.3 BigchainDB Benchmarking

The Benchmarking in this case will be focused on stress testing; sending plenty of transactions with many threads concurrently and measuring how much time does it take to receive all the transactions.

The metrics that are going to be measured in order to know the performance of each of the platforms are:

- Throughput: transactions per second that can be sent in a regular basis without losing any of them due to timeouts.
- Latency of writing in the Database: amount of time it takes to write data in the ledger.

As mentioned in the "State of the Art" Chapter, the BigchainDB API does not offer the capability of determining the number of transactions that will make a block; so that, the tests can not be focused on measuring the amount of blocks that can be created at a certain rate; which can affect the performance depending on the Block Size.

The BEP-23 (BigchainDB Enhancement Proposal 23), offers a Benchmarking tool, that provides much more feedback than the developed tool in
Python. This tool was discovered when the tests were finished, so that, it was not used, instead a Python script was created to send the transactions.

In order to install it, the following process is suggested; because a small change is required in order to install it correctly:

1. Install Python Virtualenvs. (Ubuntu)

   $ sudo apt-get install python-virtualenv virtualenv

2. Create a Python Virtualenv.

   $ virtualenv bdb

3. Activate the created Python Virtualenv.

   $ source bdb/bin/activate

4. Install the BigchainDB Python driver.

   $ pip3 install bigchaindb-driver-0.6.2

5. Modify the setup.py file from the BEP-23 Benchmarking tool: change the aiohttp version from 3.0.0 to 3.0.0b0 in order to make it work. The setup.py from the Benchmark folder should look like:

   ```python
   from setuptools import setup, find_packages
   setup(
     name='bigchaindb_benchmark',
     version='0.0.1',
     description='Command Line Interface to push transactions to BigchainDB',
     author='BigchainDB devs',
     packages=find_packages(),
     install_requires=[
       'bigchaindb-driver~=0.5.0',
       'coloredlogs~=7.3.0',
       'websocket-client',
       'logstats~=0.3.0',
       'requests~=2.19.1',
       'cachetools~=2.1.0',
       'websockets~=6.0.0',
   ```
6. Install the BigchainDB BEP-23 Benchmarking tool.

```
(bdb)$ git clone https://github.com/bigchaindb/benchmark.git
(bdb)$ cd benchmark
(bdb)$ pip install -e .
```

The Benchmarking tool provides plenty of options and an output full of information. To execute it, use the following; the command will provide usage information when no arguments are given:

```
$ bigchaindb-benchmark
```

The output provided by the BEP-23 Benchmarking tool can be seen in Figure 10.

5.4 Data Marketplace Application Load Testing

After testing the performance of each component individually (BigchainDB, IOTA and Hyperledger Fabric), it is time to test the performance of the whole Decentralized Data Marketplace API.

In order to test the whole API, two types of tests were done:
• Query Maximum Load tests in which only queries are sent; the reception of the data in the customer FIWARE Orion Context Broker determines the time it took the whole system to process the query.

• Subscriptions Maximum Load tests in which using a subscription; changes on the data of the seller were done and updates were sent to the customer. When the data was received in the notification endpoint in the customer node, the time was logged.

The Vegeta HTTP Load Testing utility was used in order to accomplish the REST HTTP API Load Testing of Queries.

In order to produce the changes on the data of the seller node, a little program was written in NodeJS which sent HTTP requests with changed data (an incremented variable in the program served as the updated data) to the seller’s API. In order to get the Benchmarking metrics, the log parser written to get the metrics for Hyperledger was used with slight changes. Both pieces of code can be found at Appendix J and K.

The reason behind not using the Vegeta HTTP Load Tester for the Subscriptions is because, data delivery latency is being measured instead of how much time takes the context broker to process a PATCH request to update the data. The PATCH request ends much before the data is delivered to the customer node; making the Vegeta tool to listen to the customer node was much harder than writing a little piece of code in NodeJS and a modified parser.

It was also required to make a slight change in the accumulator-server in order to log the microseconds it took each transaction; in the base version of the accumulator-server the precision of the timestamps is up to seconds.
6 Results

In this Chapter, the results of the tests are discussed. This Chapter is made up of 4 sections, each one discusses the Benchmarking results obtained for each of the tested technologies: IOTA, BigchainDB, Hyperledger Fabric and the whole Decentralized Data Marketplace Application.

In each section appears a description of how the tests were done, the results obtained in each test are summarized in a table and plots are given to better understand how each technology behaves under different loads of traffic.

The whole output given by each Benchmarking tool can be found at Appendix A.

6.1 IOTA

The results of the Benchmarking of the IOTA network were much worse than the expected, according to the specifications and assertions of the Software provided by the IOTA Foundation.

Many technical issues happened during the tests, which proved that IOTA is not as resilient as the IOTA Foundation advertised. Between the suffered technical problems, the following must be highlighted:

- High CPU and memory usage: all the cores were running at 100% of their capabilities during 2 minutes, in order to send 1 transaction to the IOTA network, as the figures 11 and 12 of `htop` evidence. The fact that a 100% of the resources was needed to send an IOTA transaction makes it impossible to run any other services in the node; there would be no CPU left to execute the Hyperledger Fabric containers (Orderer, Peer and Certificate Authority). The IOTA Node executed on an Ubuntu 16.04 server. The specifications of the hardware in which the tests were done are the following:

  - Processor: Intel(R) Xeon(R) CPU L5410 @ 2.33 GHz
  - RAM Memory: 48 GB DDR2 @ 667 MHz
  - Disk Space: 2 TB
Figure 11: *htop* of the node when it is not sending transactions

Figure 12: *htop* of the node when it is sending transactions
- Poor documentation: the IOTA’s APIs documentation and configuration procedure, is incomplete (more features exist that are not documented), and poor (information on the core API methods is based on Medium posts made by community members and not the IOTA Foundation).

- Common desynchronization and difficulties to synchronize with the network: once the node was installed and running, it could not synchronize itself with the neighbours and a manual download of the ledger from an official IOTA repository was needed, by the time of writing the compressed ledger weights 24.7 GigaBytes, in case it is downloaded from a repository in the Internet, the user is responsible of downloading it from an untrusted site. As the tests were done using transactions that did not contain monetary funds and the used wallet was empty, the risk of being scammed implied no real damage.

- Manual neighbor discovery: automatic neighbor discovery is part of the IOTA Foundation Roadmap. Some time ago, a system called IOTA Nelson provided this feature, which was shutdown as time passed because it made the network vulnerable to certain attacks and the performance was poor. It is hard to find neighbours near the location in which the node was deployed to perform the tests. In order to add a neighbour, both neighbours need to manually add each other. It is also necessary to add plenty of neighbours in case the user wants to make sure that the node will keep synchronized, if all the neighbours go down, the user stops receiving updates from the IOTA network.

- The owner of the node needs to trust the added neighbours; if the neighbours start sending malicious or malformed transactions, the closest neighbours to the malicious node will be the first ones receiving the wrong transactions. In case the neighbour goes down, the IOTA Software will start trying to re-establish the neighbour connections once and again, increasing the processing resources needed by the node.

- Corruption of the ledger: during the tests (which were done during a week), the system got corrupted twice, as illustrated by 13. A reboot was needed in both cases, because the IRI (IOTA Reference Implementation; that is, the IOTA’s node software) container could not resynchronize and got stuck. After rebooting the system kept corrupt and crashing, deletion of the ledger and extracting again the full ledger
was needed. The server did not suffer unexpected shutdowns, Internet connection was not down, no other programs were executed and the firewall was not modified, during this time; and the cause of this corruption is unknown.

During the tests just the IOTA node container was running on the Ubuntu 16.04 server running in gemini.ls.fi.upm.es.

The high CPU usage is caused due to how the IOTA transactions are attached to the ledger: by doing the PoW of other two tips (not yet validated transactions), before sending. This feature offers the users the ability to send feeless transactions, but affects the attachment latency, because of the high Algorithmic Complexity that validating transactions using a Proof of Work algorithm has.

The performance obtained after testing the IOTA Node was much lower than expected; proving the system to be unscalable, unresilient and incapable of supporting real-time data distribution. Each transaction required around 1 minute to be sent to the Blockchain in the case in which only 10 bundles of 1 transaction each were sent at once; when the traffic loads were increased, the time it took each transaction to be sent increased, as well as the CPU occupancy did. Because of that, IOTA was discarded as the Blockchain platform used for data delivery, that is the reason why only a few tests were performed. These are the executed tests:

- Test 1: 10 threads sending each a bundle of 10 transactions in parallel.
<table>
<thead>
<tr>
<th>Test ID</th>
<th>Transactions per bundle</th>
<th>Number of bundles</th>
<th>Throughput (tx/s)</th>
<th>Jitter (s)</th>
<th>Average Latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10</td>
<td>0.01307</td>
<td>152.4086</td>
<td>579.9919</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>0.21971</td>
<td>15.58690</td>
<td>40.78209</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>20</td>
<td>0.11985</td>
<td>238.4201</td>
<td>392.5197</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>0.66121</td>
<td>35.85039</td>
<td>93.01269</td>
</tr>
</tbody>
</table>

Table 1: Benchmarking results of IOTA

- Test 2: 3 threads sending each a bundle of 5 transactions in parallel.
- Test 3: 20 threads sending each a bundle of 5 transactions in parallel.
- Test 4: 6 threads sending each a bundle of 5 transactions in parallel.

The Table 1 is a summary of the metrics taken from the obtained results of the performed tests.

The number of transactions per bundle was kept in 5 after checking that distributing the transactions into different bundles or grouping them all in the same bundle, was not going to offer an improvement or a worse performance.

It is remarkable how unreliable and changing the IOTA Blockchain Network really is; it can be seen easily by looking at the obtained Throughput results: with a low load (10 bundles each containing a transaction) the throughput is the lowest, which may suggest increasing the amount of transactions to be sent, in order to experiment the pipelining effect (the biggest delay appears when the first transactions are being sent, as time passes, transactions appear to get inserted faster because new loads of transactions are being processed when the IOTA Node ends validating the first loads of them).

Clearly increasing the amount of sent transactions improves the Throughput a bit up to 0.66 transactions per second when 6 bundles of 5 transactions are sent at once: this performance is even under the requirements of a real-time Data Communication platform.

The IOTA Node does not scale when traffic loads increase: a Node would not be able to provide service to more than one user at the same time, it can be seen when the amount of traffic is increased up to 20 bundles of 5 transactions, in which case the average latency to process each bundle
Figure 14: Measured Throughput (tx/s) in the previous IOTA tests.

raises up to 392.51 seconds (in average it takes 78.502 seconds to append each transaction), which is far beyond the scalability values provided by the IOTA Foundation of 1000 Transactions per Second.
Figure 15: Measured Jitter in during the previous IOTA tests.

Figure 16: Measured Average Latency in (s) during the previous IOTA tests.
The Jitter plot given in Figure 15 provides insights on how sensitive to traffic loads the IOTA platform really is; under a small increase in the amount of sent transactions, the latency suffers a big change (amount of time it takes a transaction to be sent to the IOTA Network). The Decentralized Data Marketplace needs a platform for data distribution that behaves equally under different loads of traffic, otherwise, customers would receive updated data in real-time under certain conditions, when there is not a lot of traffic on the network.

As we can see in the Average Latency plot given in Figure 16, the IOTA node needs a lot of time to insert a transaction in the ledger, the optimal results were obtained when 3 bundles of 5 transactions were sent; even though it was the best result obtained, having an Average Latency of 40.78 seconds, it is not enough to fit the requirements of a platform to transmit data in real-time.

The throughput of IOTA as it can be seen in Figure 14 does not go beyond 0.66 transactions per second in the case 6 bundles of 5 transactions per bundle, are sent. In the next section, the BigchainDB obtained results will show how BigchainDB outperforms the IOTA platform, achieving a throughput almost 400 times better than the best throughput obtained with IOTA.

The measured performance does not match the performance capabilities provided by the IOTA Foundation, which claims that their platform is capable of processing 1000 transactions per second. The IOTA Foundation has been criticized several times for different reasons: from unscalability to insecurity of their platform.

6.2 BigchainDB

The tests will measure the performance with different traffic density scenarios; to achieve that, a different amount of threads will be used in each test.

The reason behind this decision is due to the fact that the BigchainDB API does not provide a parameter to tune the number of transactions inside each block; so that, it is difficult to understand what happens in different traffic density scenarios.

The benchmarking metrics that will be used to measure the performance are:
• Throughput of Transactions sent to the ledger.

• Jitter of the Latency of sending transactions.

• Average Latency of the time that each thread needs to send all the transactions.

These are the results obtained using the async_send function from the BigchainDB Python API:

• Test 1: 1 thread sending 1000 transactions.

• Test 2: 4 threads sending 1000 transactions.

• Test 3: 8 threads sending 1000 transactions.

• Test 4: 1 thread sending 10000 transactions.

• Test 5: 4 threads sending 10000 transactions.

• Test 6: 8 threads sending 10000 transactions.

After launching all the tests except the 4th test, which was executed the last one, the BigchainDB node got "Unhealthy”, in order to complete the last test, all the Docker containers where deleted and recreated; because restarting them did not solve the problem. The unhealthy docker container can be removed after issuing the following command:

```
$ systemctl restart docker.socket docker.service
```

One interesting test that was not planned at the beginning is distributing the transactions between different threads instead of increasing the number of transactions per thread; so that, tests sending 1000 and 10000 transactions, using a different amount of threads were performed. The following tests were achieved:

• Test 7: 1000 transactions are distributed into 4 threads: each thread sending 250 transactions.

• Test 8: 1000 transactions are distributed into 8 threads: each thread sending 125 transactions.
## BigchainDB Benchmarking metrics

<table>
<thead>
<tr>
<th>Test ID</th>
<th>Number of Threads</th>
<th>Transactions per Thread</th>
<th>Throughput (tx/s)</th>
<th>Jitter (s)</th>
<th>MeanTime / thread (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1000</td>
<td>50,6002</td>
<td>—</td>
<td>19.76273</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1000</td>
<td>140,093</td>
<td>0.11163</td>
<td>28.41914</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1000</td>
<td>167,513</td>
<td>0.508192</td>
<td>47.07708</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10000</td>
<td>50,3566</td>
<td>—</td>
<td>198.5834</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>10000</td>
<td>98,0985</td>
<td>0.452382</td>
<td>407.2763</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10000</td>
<td>159,453</td>
<td>1.552086</td>
<td>499.7915</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>250</td>
<td>159,049</td>
<td>0.212883</td>
<td>6.122092</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>125</td>
<td>262,907</td>
<td>0.061554</td>
<td>3.732231</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>2500</td>
<td>152,958</td>
<td>0.148688</td>
<td>65.16513</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>1250</td>
<td>260,904</td>
<td>0.278740</td>
<td>38.07361</td>
</tr>
</tbody>
</table>

Table 2: Benchmarking results of BigchainDB.

- Test 9: 10000 transactions are distributed into 4 threads: each thread sending 2500 transactions.

- Test 10: 10000 transactions are distributed into 8 threads: each thread sending 1250 transactions.

The Table 2 provides a summary of the results of the performed tests.

In comparison with IOTA, BigchainDB reached a Throughput around 400 times better, if we compare both performances using the highest measured throughput. During the tests there were no issues with respect the CPU overload that was experienced during the IOTA tests.

As we can see, Tests 7-10 proof that distributing the number of transactions between different threads increased the Throughput, by 3 times when instead of 1 thread to send 1000 or 10000 transactions 4 were used, and by 5 times when 8 threads were used instead of 1. Increasing more than 8 the number of threads to send transactions makes no sense because the CPU of gemini.ls.fi.upm.es has 8 cores.

In terms of Jitter, it is clear that the system is resilient and stable, provides the same capabilities during all the tests without big changes, in comparison with IOTA the BigchainDB’s Jitter is 153,61 times lower.

In terms of the Mean time that it took each thread to send all its transactions; it is clear that the more transactions a thread needs to send, the
Figure 17: Throughput evolution when increasing the number of sent Transactions per Thread.

more time it will take it to append them to the BigchainDB, but the mean time between the insertion of two different transactions was maintained, the maximum time was 0.05 seconds, which can be obtained from the results of the 6th test. The minimum average time between two transactions was around 0.02 seconds, which can be obtained from the results in the 1st test.

Figure 17 shows that even though the number of transactions sent by each thread is increased (the blue line represents tests that sent 10 times more transactions than the orange one), the throughput almost does not change, if we compare both lines; the BigchainDB Blockchain, provides a high throughput, but as every distributed platform, there is a limit on the amount of transactions or requests that can be processed in a period of time.

Figure 18 clearly shows the effect of distributing the amount of transactions between different threads, instead of just increasing its number (Tests 1, 4, 7-10): increasing the number of threads 4 times reduces the Average Latency time that took each thread to process the transactions in a significant manner; it suggests that the Algorithmic Complexity of the Consensus Algorithm permits the CPU to run more instances of the platform without making worse its performance.

The plot given in Figure 19, shows graphically the same effect experi-
Figure 18: Average Threat Latency evolution when increasing the number of sent Transactions per Thread.

Figure 19: Latency evolution when increasing the number of sent Transactions.
enced in the previous tests, increasing the number of transactions sent to the Blockchain, will require the system more time to finish their validation. It is clear that up to 8000 transactions per second can be sent to Bigchain without suffering a huge increase in the Latency that it takes the system to append transactions. When the number of transactions sent at once to the BigchainDB Blockchain reaches 10000 transactions, the Latency starts to grow much faster than before ending up in around 500 seconds to process 80000 transactions in the Test 6. The Decentralized Data Marketplace requirements can be fulfilled with the obtained results, because BigchainDB is meant to behave like a real-time data delivery platform, and the Channels modelled in BigchainDB are used to transmit Queries, Subscriptions and Data Updates, it seems that no more than 8000 transactions will be required by the Decentralized Data Marketplace to be processed at the same time by the BigchainDB network having into account the ”Normal User Behaviour of the Decentralized Data Marketplace” discussed in the Section ”Common use of the Data Marketplace” in the ”Hyperledger Benchmarking” Chapter.
6.3 Hyperledger

The Hyperledger tests pretended to measure how many transactions and assets of different types could be created before the Hyperledger Fabric network started to drop transactions. Transactions end up being dropped if they are not validated before a certain amount of time, in order to provide a security feature to avoid Denials of Service caused by malicious nodes.

If we try to stress more the system, the following errors are thrown for each of the transactions that cannot be validated before the internal Hyperledger timers end:

(node:16976) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). (rejection id: 1043)
(node:16976) UnhandledPromiseRejectionWarning: Error: Error trying invoke business network with transaction id 79fea62d43705f9cd634b1c991fe3f9ab1798646cd5760b7a30967a2f472d329. Error: No valid responses from any peers.
Response from attempted peer comms was an error: Error: failed to execute transaction 79fea62d43705f9cd634b1c991fe3f9ab1798646cd5760b7a30967a2f472d329: error sending: timeout expired while executing transaction at HLFConnection.invokeChainCode ...

The following 3 sections will discuss three types of tests that were performed to assess the performance of the Hyperledger Fabric Network:

- Normal User Behaviour tests in which the transactions are sent at once instead of sending them at a certain pace like in the last two sections.

- Maximum Load Testing simulating the normal user behaviour in which each 4 Agreements are created per Offering, and as many Offerings as Datasets are created.

- Maximum Load Testing creating the same amount of Datasets, Offerings and Agreements.

The "Normal User Behaviour" that is simulated during the tests discussed in the following two sections was defined in the "Common Use of the
Data Marketplace" Section in the "Hyperledger Benchmarking" Chapter; in order to measure the performance of the Hyperledger Fabric Network using values that were close to the real needs of the Decentralized Data Marketplace.

6.3.1 Normal User Behaviour Hyperledger Tests

In the section "Common use of the Data Marketplace", the rate of how many assets of different types would a normal user create, was discussed. The obtained values are used in this section to test if the Hyperledger Fabric network could put up with a real scenario.

The tests in this section were based on generating the amount of traffic load required for each test, at once, and leaving the Hyperledger Fabric Network time until all the transactions were processed. The results with the amount of time that it took the network to validate each transaction were sent to a file; which was parsed later on and the metrics values were calculated.

As the system could not handle more than 35 offerings, 35 datasets and 128 agreements at once, without discarding any transaction due to timeouts, we decided to make tests with lower values.

The following tests have been done:

- Test 1: Create 1 Dataset, 1 Offering and 4 One Time Payment Agreements (1/16 of normal use).
- Test 2: Create 4 Datasets, 4 Offerings and 16 One Time Payment Agreements (1/8 of normal use).
- Test 3: Create 8 Datasets, 8 Offerings and 32 One Time Payment Agreements (1/4 of normal use).
- Test 4: Create 16 Datasets, 16 Offerings and 64 One Time Payment Agreements (1/2 of normal use).
- Test 5: Create 32 Datasets, 32 Offerings and 128 One Time Payment Agreements (normal use).

A summary of all the Benchmarking metrics of Hyperledger can be seen in the Table 3.
Hyperledger Benchmarking metrics

<table>
<thead>
<tr>
<th>Test ID</th>
<th>N. Data</th>
<th>N. Off</th>
<th>N. Agree</th>
<th>Throughput (tx/s)</th>
<th>Avg Jitter (s)</th>
<th>Avg Latency (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4.28396</td>
<td></td>
<td>1.67675</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
<td>16</td>
<td>8.62865</td>
<td>237.570</td>
<td>4.63516</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>32</td>
<td>10.4780</td>
<td>275.194</td>
<td>8.87916</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>16</td>
<td>64</td>
<td>11.8618</td>
<td>342.772</td>
<td>17.2437</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>32</td>
<td>128</td>
<td>12.2992</td>
<td>650.735</td>
<td>34.1798</td>
</tr>
</tbody>
</table>

Table 3: Benchmarking results of Hyperledger sending certain amount of each asset.

As in the previous sections in which the performance of the platforms BigchainDB and IOTA were discussed; in Hyperledger Fabric the Throughput also flattens as the traffic load increases; it can be clearly seen in the plot of the Figure 22, because there is almost no difference in the Throughput when 48, 96 and 192 assets are created at once.

The Average Latency values provided in the Table 3, clearly show that the Average Latency is really influenced by the amount of Agreements that are created, the reason behind this phenomenon is that the overhead to create an Agreement is much more than creating Datasets or Offerings; because 3 transactions are required to create a valid agreement, instead of 1 for datasets and offerings; and the higher Algorithmic Complexity of the MakeAgreement transaction, that takes much more time to execute than any other Smart Contract in the Decentralized Data Marketplace.

The Jitter in this case provides another proof of the aforementioned phenomenon: it takes more time and resources to create agreements than datasets and offerings. That is the reason behind the high Jitter values that can be seen plotted in Figure 21. As we can see in the provided results in Table 3, the Jitter reaches 650.735 in Test 5. The Jitter measures the variance of the Latency to create each of the Assets; because the Agreements take much more time to be created (requires processing 3 transactions), and in addition the AcceptAgreement transaction that ends up changing the Agreement State into ACTIVE, may be processed much after the previous 2 transactions (MakeAgreement and PaymentCompleted), because more Datasets and Offerings were processed in between the first two transactions to create an agreement and the last.

By looking at Figure 20 one can realize that the Average Latency between
tests, increases as many times as the amount of assets; so that, as the number of assets created between the first and second test is increased by 4 times, so does the *Average Latency*. Between the rest of the tests the amount of assets created is multiplied by 2, so does the *Average Latency* again. This is an interesting result, because it shows that the *Hyperledger Fabric* Blockchain is capable of processing the assets in the same amount of time no matter the load that it is put into; unless the threshold that makes the Blockchain to start dropping transactions is reached. If the *Hyperledger Fabric* Blockchain needed more time when more requests are sent, the *Average Latency* would be increased by more than the amount of requests between tests.

Figure 20: Measured Latency in (s) during the previous tests.
Figure 21: Measured Jitter in (s) during the previous tests.

Figure 22: Measured Throughput in (assets/s) during the previous tests.
6.3.2 Maximum Load Testing Normal User Behaviour

Another kind of interesting tests are those based on controlling the frequency ("pace") at which the transactions are inserted instead of letting the benchmarking tool insert them at once. This kind of tests are called maximum load tests.

The following tests were left executing for 10 minutes; this amount of time is enough to avoid taking wrong values during the first minutes of the test execution. At first, the node is not occupied with transactions to validate, as time passes the load of the node is increased with new transactions and the previous ones that could not validate to that point (at a certain transaction rate, they are sent to the node faster than the node validates them).

We measured the number of transactions that were not validated in comparison with the ones that got validated. All the lost transactions were due to the Hyperledger Fabric internal timer expiration; no corrupted transactions have been sent during these tests.

The Normal User Behaviour of the Marketplace is creating 4 Agreements per Offering, and as many Offerings as Datasets. The transactions are sent one after the other.

The following Maximum Load tests were done:

- Test 1: Creating 1 Dataset every 1000 ms, 1 Offering every 1000 ms and 1 Agreement every 1000 ms. 4 Agreements are created in each round.
- Test 2: Creating 1 Dataset every 500 ms, 1 Offering every 500 ms and 1 Agreement every 500 ms. 4 Agreements are created in each round.
- Test 3: Creating 1 Dataset every 200 ms, 1 Offering every 200 ms and 1 Agreement every 200 ms. 4 Agreements are created in each round.
- Test 4: Creating 1 Dataset every 185 ms, 1 Offering every 185 ms and 1 Agreement every 185 ms. 4 Agreements are created in each round.
- Test 5: Creating 1 Dataset every 175 ms, 1 Offering every 175 ms and 1 Agreement every 175 ms. 4 Agreements are created in each round.
- Test 6: Creating 1 Dataset every 160 ms, 1 Offering every 160 ms and 1 Agreement every 160 ms. 4 Agreements are created in each round.
Table 4: Benchmarking results of Maximum Load Tests in Hyperledger using the normal behaviour ratio.

- Test 7: Creating 1 Dataset every 150 ms, 1 Offering every 150 ms and 1 Agreement every 150 ms. 4 Agreements are created in each round.
- Test 8: Creating 1 Dataset every 100 ms, 1 Offering every 100 ms and 1 Agreement every 100 ms. 4 Agreements are created in each round.

The Table 4 summarizes the metrics taken from the experiments in which the normal user behaviour asset creation ratio was generated in Maximum Load tests (it is considered only as a completed Agreement those which could validate the 3 transactions required to complete an agreement, so that, in the table the field "Out. Agr" = min\{AcceptedAgreements, PaymentCompletes, Acquisitions (MakeAgreement transaction)\}).

The Hyperledger Fabric network scales up to validating 7470 transactions in 10 minutes; which were the results of the Test 4. From that point, accelerating a bit the pace at which assets are created, will end up dropping 3611 transactions as seen in the Test 6 results. Whenever the network can not cope with the transactions it receives, it starts dropping them (because the internal Hyperledger timers finish, and if the transaction is unvalidated at that point, it gets dropped by the Hyperledger Fabric Orderer Node), and it takes minutes (depending on how much overwhelmed the Hyperledger Fabric Orderer is), without receiving new transactions, until the Hyperledger Fabric Orderer recovers stability.

The best throughput is achieved in the Test 4, being capable of process-
Figure 23: Measured Output Asset/s Differentiating between Asset Types and simulating a Normal Behaviour.

ing 12.44 transactions per second. From that point, in the next test the throughput halves; and ends up under 1 transaction per second during tests 7 and 8.

By looking at the results obtained during Test 5 that can be seen in Table 4, only 1 transaction is dropped, but the amount of Valid Tx is around 3300 less than in the Test 4; it seems that many of the transactions were almost going to be dropped in the following minutes after stopping the tests.

The differences between the amount of requests to create assets (In Data, In Off and In Agr), with respect of the throughput of each of the assets (Out. Data, Out. Off and Out. Agr), are meaningless in the Tests 1-4, the reason behind them is the fact that at some point the test needs to be stopped and at that moment, some transactions (meaningless amount of them for testing purposes) may be finishing validation; so that, as the test is stopped, the values of these transactions, get never logged. The fact that up to Test 4 (included), the Hyperledger Fabric Network can cope with the generated traffic load, can be checked by finding 0 dropped transactions during Tests 1-4 and an increasing Throughput as the amount of Input Transactions/s was increased.

As we can see in the Figure 23, the differences in the plotted lines that
represent requested Agreements to be created and those Agreements that really end up being created and validated; has the same shape as the plotted lines of Offerings and Datasets, the only difference is that the Agreements lines are 4 times over the other because of the normal user behaviour ratio.

It is clear that the *Hyperledger Fabric* network supports up to creating 5.4 assets per second, from that point transactions start to get dropped, so that, many assets end up not being created.
6.3.3 Maximum Load Testing: Same Amount of Assets

Another interesting test of the Hyperledger Fabric platform, was doing Maximum Load Tests creating the same amount of Datasets, Offerings and Agreements, instead of trying to simulate a normal user behaviour of the Decentralized Data Marketplace which implies that 4 Agreements will be created per Offering and the same amount of Offerings and Datasets will be created: the expected user behaviour ratio of the Decentralized Data Marketplace is provided in the Section "Common use of the Data Marketplace" in the "Hyperledger Benchmarking" Chapter.

The reason behind making the following tests is understanding how much does the Agreement creation overload the network; in order to make an Active Agreement, 3 transactions are required; instead Dataset and Offering creation only involves one transaction.

Creating one type of asset after an X amount of time:

- Test 1: Creating 1 Asset every 1000 ms, 1 Offering every 1000 ms and 1 Agreement every 1000 ms.
- Test 2: Creating 1 Asset every 500 ms, 1 Offering every 500 ms and 1 Agreement every 500 ms.
- Test 3: Creating 1 Asset every 333,33 ms, 1 Offering every 333,33 ms and 1 Agreement every 333,33 ms.
- Test 4: Creating 1 Asset every 250 ms, 1 Offering every 250 ms and 1 Agreement every 250 ms.
- Test 5: Creating 1 Asset every 200 ms, 1 Offering every 200 ms and 1 Agreement every 200 ms.
- Test 6: Creating 1 Asset every 166,66 ms, 1 Offering every 166,66 ms and 1 Agreement every 166,66 ms.
- Test 7: Creating 1 Asset every 142,85 ms, 1 Offering every 142,85 ms and 1 Agreement every 142,85 ms.
- Test 8: Creating 1 Asset every 125 ms, 1 Offering every 125 ms and 1 Agreement every 125 ms.
- Test 9: Creating 1 Asset every 117.64 ms, 1 Offering every 117.64 ms and 1 Agreement every 117.64 ms.
Table 5: Benchmarking results of Maximum Load Tests in Hyperledger. The normal behaviour ratio is not taken into account. 1 of each asset type is created at the same frequency.

- Test 10: Creating 1 Asset every 111.11 ms, 1 Offering every 111.11 ms and 1 Agreement every 111.11 ms.

- Test 11: Creating 1 Asset every 100 ms, 1 Offering every 100 ms and 1 Agreement every 100 ms.

The Table 5 summarizes the performance results of the aforementioned tests.

Comparing this table results with the previous one, and having in mind that in the previous tests (those presented in the previous subsection, which generated the assets following a normal user behaviour of the network) more Agreements than Offerings and Datasets were created; it is clear that certain transactions require more time to be processed than others; that is the case of Agreements: making an agreement requires calculating the price when the MakeAgreement transaction is processed, the MakeAgreement Smart Contract requires more calculations than those executed to create Datasets and Offerings, involving plenty of additions and multiplications of floating point numbers to calculate the final price of the Agreement. In the cases of Datasets and Offerings, the logic that the creation transactions require is much lower, so that, more transactions can be processed. It can be experimentally seen when we compare the maximum amount of transactions that
Figure 24: Hyperledger Maximum Load Test Benchmarking Results not having into account the generated Asset Type

the Hyperledger Fabric network was capable of processing without dropping any of them, which in this case is 7890 transactions, against the 7470 transactions in the previous section, in which the Normal User Behaviour was simulated.

As we can see in Tests 5 and 7 two transactions are dropped, the reason of these transactions being dropped is not notorious and could be caused because two transactions during a test received the same random generated id; duplicated transactions are immediately dropped by the Hyperledger Fabric Network, to avoid Double Spending attacks.

The Figure 24 clearly shows the traffic load that the Hyperledger Fabric Network can cope with before starting to drop transactions, because the internal Hyperledger timers finish and transactions get dropped. The throughput drops dramatically from 8 created assets per second (Test 8) to 2.56 assets per second (Test 9); between those tests the received transactions to create assets increases from 8 to 8.5 assets per second (Tests 8 and 9).

When the number of assets created per second drops, it does not recover (Tests 9, 10 and 11). In the last test around 0.4 assets are created per second in average.

This behaviour can be solved if a congestion avoidance algorithm is im-
plemented to control the amount of transactions that are sent at the same time. This solution is proposed in the "Further Work" Chapter.

6.4 Decentralized Data Marketplace

The performance of the whole Decentralized Data Marketplace platform was tested in order to make sure that a real-time Data transmission could be achieved; and to find out if the platform is scalable when big amounts of data and requests are generated. The tests described in this section describe End-to-End scenarios, that use the Decentralized Data Marketplace Client Application and both Blockchain platforms (BigchainDB and Hyperledger Fabric).

One time payments and Subscriptions were tested (by the time of writing the Usage based pricing model is still not implemented).

After each test, the previous Agreements were deleted from the system, a new agreement created for each new test, and new Data and Query Channels were created; in order to measure the performance of the whole system in the same conditions, otherwise each test would use a channel that contains all the previously transmitted data; which affects the performance.

Both Pricing Models (Subscriptions and Queries) were tested using Maximum Load tests. In the case of the Query Load tests, the Vegeta HTTP Load Testing tool was used; because it eases the testing process and provides a lot of metric results and plots that represent the Latency to process each transaction.

In order to test the amount of Data Updates that can be streamed in real time, the Vegeta tool was not used, instead the accumulator-server was modified to log the whole timestamp (with a precision up to milliseconds) when the Data Updates arrived.

The Maximum Load Testing type of benchmarking was used because it was interesting to know at which frequency the system could not handle more data updates or processed queries.

6.4.1 Data Marketplace Maximum Query Load Testing

This section discusses the performed tests to know the frequency at which Queries can be sent by the customer that has established a One Time Payment Agreement, and at which frequency the customer starts not receiving
responses for the queries. This kind of Benchmarking Test is called Maximum Load Testing.

In order to measure the performance, the Vegeta HTTP Load Tester was used because it eases the measurement process and provides plenty of metrics, which requires less calculations to understand the performance of the Decentralized Data Marketplace. The Vegeta tool was provided with the contents of a HTTP Request of a Query and the needed parameters to execute each of these tests: the frequency at which the HTTP Requests are sent, the amount of time that the test needs to run and the name of the output file to which the results will be provided. All the tests were executed during 10 minutes, and for each test the frequency at which the queries were sent was changed, in order to know at which point the system cannot process more Queries and the customer starts not receiving responses.

The following tests were done to measure the performance of the Data Marketplace API processing queries:

- Sending 5 Queries per second during 10 minutes.
- Sending 10 Queries per second during 10 minutes.
- Sending 20 Queries per second during 10 minutes.
- Sending 30 Queries per second during 10 minutes.
- Sending 50 Queries per second during 10 minutes.
- Sending 100 Queries per second during 10 minutes.
- Sending 200 Queries per second during 10 minutes.
- Sending 500 Queries per second during 10 minutes.
- Sending 1000 Queries per second during 10 minutes.
- Sending 2000 Queries per second during 10 minutes.
- Sending 2500 Queries per second during 10 minutes.
- Sending 2750 Queries per second during 10 minutes.
- Sending 3000 Queries per second during 10 minutes.
- Sending 3250 Queries per second during 10 minutes.
Table 6: Benchmarking Metrics results during the Maximum Query Load testing of the Data Marketplace API.

- Sending 3500 Queries per second during 10 minutes.
- Sending 4000 Queries per second during 10 minutes.
- Sending 4500 Queries per second during 10 minutes.
- Sending 5000 Queries per second during 10 minutes.
- Sending 10000 Queries per second during 10 minutes.

The Table 6 summarizes the results obtained during the aforementioned tests.

Regarding the results provided in Table 6 it is remarkable the amount of queries that can be processed in 10 minutes, receiving a response for
each of the sent queries; Test 12 results show that 1650001 Queries could be processed in 10 minutes, and they all received a response in time. This amount of queries really fits the requirements of the Decentralized Data Marketplace; it is viable to assume that a customer will not send a query more than once a second; in that case the system could cope with 1650001 customers sending queries at the same time.

It is also remarkable the fact that the amount of not answered queries grows much faster than the increase in the sent queries, which goes is incremented by 250 in the Tests 11-15, and by 500 in the tests 15-18.

During Test 19, the amount of Unanswered Queries is around 2 times bigger than the amount of responded ones. So it is easy to predict that the system will not be able to process more than 3500 Queries per second if tests with even more Queries per second were done.

The reason no tests beyond the 10000 Queries per second were done, is basically that the Quality Requirements of the Decentralized Data Marketplace will not permit a customer not receiving the data for which she has paid to. So that, once the amount of Queries per second that the Decentralized Data Marketplace was able to handle without not answering one Query was known; the Benchmarking goals were fulfilled, but it was interesting to know at which pace the amount of unanswered Queries grows as the number
of sent Queries per second increases.

Figure 25 contains less data values plotted with respect all the results obtained in the Table 6 because for the first 12 tests the amount of dropped queries is zero; so that, it is not interesting from the Benchmarking point of view to see a plot with much more data that does not provide information itself. So that, the results from Test 12 were plotted. In addition it is interesting to plot data for $x$ values that grow linearly in the same amount; so that the results for the Tests 16-19 were omitted, because from Test 16 the amount of Input Queries/s increases by 500 instead of by 250 like in the Tests 11-15. The most interesting interval to analyse is the one measured in Tests 11-15, because it provides a fine-grained view on how the amount of unanswered queries grows; that is the reason why the increases between those tests is lower than between any other pair of Tests.

As we can see in the Figure 25 the Decentralized Data Marketplace can process up to 2750 Queries per second, without losing any of them. During the next test in which 3000 Queries are sent to the system per second, 65830 Queries are not processed and so that, lost (around 110 Queries per second are lost, as we can see the gap between the 3000 and 2890).

It is clear looking at the plot that the amount of queries per second that the platform can support ends up flattened during the tests at which 3000, 3250 and 3500 Queries per second were sent.

The Figures [26-32] were created using the vegeta plot command, they show the Latency changes that it took each query to be processed.

It is clear that the more Queries are sent per second to the Decentralized Data Marketplace by the customer, the more ”chaotic”, the latency values are. In the Figures 26 and 27 which show the Latency results of sending 5 and 10 queries per second (low amount of queries which can be processed without a big overhead by the Decentralized Data Marketplace node), we can see that almost all the Queries were processed under 2 seconds, and the spikes in the plot are meaningless.

As more Queries are sent by the customer per second, as can be seen in the Figures [28-32], the plots become more chaotic, more spikes appear showing that in those points the occupancy of the node did not let them to be processed on time.

The last two Figures (31 and 32), show a very chaotic and unpredictable behaviour with more spikes than values in a normal range of time to be processed.
Figure 26: Latency to process each query: During 10 minutes, 5 queries were sent per second.

Figure 27: Latency to process each query: During 10 minutes, 10 queries were sent per second.
Figure 28: Latency to process each query: During 10 minutes, 100 queries were sent per second.

Figure 29: Latency to process each query: During 10 minutes, 500 queries were sent per second.
Figure 30: Latency to process each query: During 10 minutes, 1000 queries were sent per second.

Figure 31: Latency to process each query: During 10 minutes, 5000 queries were sent per second.
Figure 32: Latency to process each query: During 10 minutes, 10000 queries were sent per second.

Thanks to these results, the amount of Queries per second that the Decentralized Data Marketplace can cope with is known, and the amount of permitted queries to be sent by each user can be controlled, in order to avoid the user going beyond the supported amount of queries per second. Going beyond that threshold will not give the customer any benefits, and will flood the Decentralized Data Marketplace Network, causing an overload, which will affect other users of the platform, with latency and delayed communications.
6.4.2 Data Marketplace Maximum Data Updates Load Testing

In this section, the Decentralized Data Marketplace is tested, in order to get the amount of Data Updates that can be sent in real-time and without delays to a user subscribed to a Dataset.

After each test the Agreements are removed; before starting each test, a new agreement is created and validated. Then the customer creates the subscription, the seller listens to data updates after verifying that the customer really had purchased the Dataset. For each test new Data and Query Channels are created. The measures are not taken until the channels are setup, and the seller and customer listening to the corresponding channels. The accumulator-server that will serve as the customer application that receives the data is also started before the Data Updates are created. The logs of the accumulator-server containing the timestamps of when each Data Update is received, are sent to an output file that is parsed later on.

In order to know when a data update is received by the customer, the accumulator-server provided in the FIWARE Context-Broker repository is used. It is nothing more than an HTTP REST Server that logs every HTTP Request that receives. Minor changes were done to the accumulator-server, to log the whole timestamp (with a precision up to milliseconds), because the accumulator-server as it is delivered in the Context Broker repository, logs the timestamp with a precision up to seconds; millisecond precision is required to measure the performance.

The following tests were done to measure the performance of the system sending updates to Subscribed customers:

- Send one Data Update every 125 ms.
- Send one Data Update every 250 ms.
- Send one Data Update every 500 ms.
- Send one Data Update every 1000 ms.
- Send one Data Update every 1500 ms.
- Send one Data Update every 2000 ms.
- Send one Data Update every 2500 ms.
- Send one Data Update every 3000 ms.
<table>
<thead>
<tr>
<th>Test ID</th>
<th>In Query/s</th>
<th>Out. Query/s</th>
<th>Received Updates</th>
<th>Dropped Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.33</td>
<td>0.33</td>
<td>201</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>0.4</td>
<td>240</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>0.5</td>
<td>300</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0.66</td>
<td>0.51207</td>
<td>307</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0.52939</td>
<td>317</td>
<td>281</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0.57315</td>
<td>345</td>
<td>858</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>0.58063</td>
<td>349</td>
<td>2055</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0.57121</td>
<td>343</td>
<td>4460</td>
</tr>
</tbody>
</table>

Table 7: Benchmarking metrics results of the Maximum Data Updates testing of the Data Marketplace API.

The Table 7 summarizes the performance results obtained during the aforementioned tests.

The Decentralized Data Marketplace cannot provide more than a Data Update every 2 seconds without losing Data Updates, if the amount of Queries sent per second goes even slightly beyond that (more than 1 Update every 2 seconds), some data updates will be lost (they will be received after the test was stopped with a certain delay). Delayed data updates are considered to be "dropped" because of the real-time requirements of the Decentralized Data Marketplace.

There is no point on trying to send more than 1 Data Update every 2 seconds, increasing the amount of Data Updates does not provide advantages to the customer because data will arrive certainly delayed and in addition, the network will be occupied moving data that will be worthless to the customer because of the real-time nature of the Decentralized Data Marketplace.

The maximum amount of data updates received during the tests is 349, in the 7th Test. As we can see in the results of the Tests 6-8, the amount of data updates that are received by the customer is flattened to around 340 data updates.

The Figure 33 plots the performance of the Data Marketplace API under different amounts of traffic loads.

By looking at Figure 33 it is clear that the Data Updates received by
the client in *real-time* start to flatten when 0.66 Data Updates are sent per second. The plot also clearly shows how the tendency flattens, and even though the amount of Data Updates sent per second increases, the amount of received ones does not grow that much and many of them get out of order.

The best achieved Throughput (Out. Query/s) is achieved during the Test 7; it can be clearly seen that the throughput flattens as it also happened with all the previous tested Blockchain platforms (*Hyperledger Fabric*, *BigchainDB* and *IOTA*). The throughput provided by the whole *Decentralized Data Marketplace* is the lowest if we compare it with *BigchainDB* and *Hyperledger Fabric* (the finally chosen Blockchains platforms); which makes sense, because testing the *Decentralized Data Marketplace* involves that the overhead of all the Blockchain platforms will affect the obtained performance results.
Blockchain technology is clearly being influenced by hype and technological improvements; this fact is clear after measuring the performance of different Blockchain technologies; it is also clear that they should not be used to solve every problem in which a database or decentralized database may help. Because of that, Blockchain should be considered after defining the features needed to be supported by the platform.

It is also clear that there is a need to provide Blockchain performance capabilities, in order to know how the system will behave under certain scenarios; and clarify if the platform will offer the expected results or not.

Every single Blockchain technology is very different from the rest, and they should be used for the purpose they were meant. Using a Blockchain framework that is not meant to fit in the development needs; will end up making the development team to research on other technologies; and the development of the client Blockchain application will be done again, which is basically what happened to the Decentralized Data Marketplace.

It is clear that considering IOTA as real-time data delivery platform with non-repudiation features because of its Blockchain nature, is a mistake. The obtained performance results were far from the IOTA’s capabilities provided by the IOTA Foundation, which claims that the platform is capable of processing around 1000 transactions per second. The IOTA Network as well as its Software to deploy a node are unscalable; which ends up consuming all CPU resources. In addition there is lack of documentation and materials on how to deploy, connect to the network and add neighbour nodes.

Because of discarding IOTA a lot of the client application code of the Decentralized Data Marketplace was written again using the BigchainDB framework, which clearly outperformed the IOTA platform.

Because of the differences between Blockchain technologies in the way they work and the use cases they are meant for; on the one hand a Blockchain technology with a lighter Consensus Algorithm in terms of Computational Intensity like BigchainDB was used as the platform to transmit data in real-time; on the other hand Hyperledger Fabric was used to provide a permissioned Blockchain environment, in which users needed to be authorised to be part of the network. Even though the Hyperledger Fabric Consensus Algorithm is much more intense in Algorithmic Complexity means, it is
necessary to provide security and enforce user Authentication (Hyperledger
Fabric provides the Blockchain architect the capabilities to determine the
Enforcement Policy to provide a lighter Consensus Algorithm, requiring less
valid votes to validate the transaction, in case there is a need to make the
validation process a bit faster).

The aforementioned choice of using two separate Blockchain technologies
provided more scalability and a better performance: when Data, Queries
or Subscriptions are sent through the BigchainDB platform, less validity
checks need to be done and data can be transferred faster. The Compu-
tational Complexity comes when Business Data needs to be validated in the
Hyperledger Fabric Blockchain, or a new user is added to the Hyperledger
Fabric network.

It is also clear that the security features that Hyperledger Fabric provide,
affect the performance of the platform after stress testing it, clearly showing
that the Hyperledger Fabric Blockchain can be affected by huge loads of
transactions, giving the opportunity to attackers to cause a Denial of Service.

Implementing a Congestion Prediction or Congestion Management pro-
tocol is suggested to avoid Denial of Service attacks in the Decentralized
Data Marketplace.

Even though the BigchainDB performance is much better than IOTA, it
is not still the best choice (actually a Blockchain platform meant to provide
real-time data distribution has not been found yet); because of the use cases
of BigchainDB and its inner workings, using the platform for real-time data
delivery required modelling Assets (meant to be used as Digital Twins in
the BigchainDB Blockchain), as communication channels.

As Blockchain technology is still in incubation, every week plenty of
frameworks and Blockchain platforms appear and disappear; leading to the
need of continuously measuring the performance of these new technologies;
in order to know which ones fit better for each project needs.

Even though the reader may think that only being able to send Data
Updates in real-time without delays every 2 seconds is not enough; that
delay is acceptable and present in every other Internet of Things Peer-to-
Peer Network; even without using Blockchain technologies, so that, the
Decentralized Data Marketplace provides the capabilities and performance
of a common Peer-to-Peer network but in addition, it also provides the
advantages of Blockchain technologies, like non-repudiation and no trust
required in a central entity.
8 Further Work

Improvements to the Decentralized Data Marketplace Client Application should be done, in order to avoid Denial of Service attacks to the Hyperledger Fabric Blockchain, in which a malicious user tries to overwhelm the network with high loads of traffic. This kind of attacks are possible in case a node starts flooding the network with transactions; for each transaction the Hyperledger Fabric Orderer starts a timer, if the timer ends up before the transaction is validated, the transaction is dropped. When a node floods the network with transactions, the capabilities of the Hyperledger Fabric Orderer cannot put up with that amount of requests and plenty of the following requests get dropped before validation. Using a Congestion Management and Prediction will avoid this kind of behaviour, because the Decentralized Data Marketplace Client Application will avoid users flooding the Hyperledger Fabric Blockchain.

The amount of requests sent to Hyperledger Fabric can be controlled using a Token Bucket algorithm, which will control the number of requests that will be forwarded to the Hyperledger network; in order to avoid overwhelming it. Thanks to the Token Bucket algorithm to control congestion, there would be no discarded transactions due to Hyperledger Fabric validation timeouts; and the performance of the Hyperledger Fabric network will not drop at any time.

In further versions of the Decentralized Data Marketplace, the same tests carried during this research should be done; and the metric results should be compared to the obtained in this research, in order, to understand if the Software is improved in terms of performance. Having into account the aforementioned Denial of Service issue in the Hyperledger Fabric Software; it should be checked if this behaviour is avoided when the Token Bucket algorithm is implemented.

The Decentralized Data Marketplace also requires a GUI, in order to really become a commercial product.

It could be also interesting to do fuzz testing in order to find vulnerabilities in the communication protocols used by Hyperledger Fabric, or the format support by feeding the Marketplace with malformed transactions and asset information.

Finally, as Hyperledger Composer is stopping to get support, the alternative is using the Convector framework. Which provides the same capabilities
as Hyperledger Composer using a different syntax.
9 Acronyms

- **API** Application Programming Interface
- **BEP** BigchainDB Enhancement Proposal
- **BFT** Byzantine Fault Tolerance
- **BNA** (Hyperledger) Business Network Application
- **CA** Certificate Authority
- **CPU** Central Processing Unit
- **CRUD** Create, Read, Update and Delete
- **CSV** Comma-Separated Values
- **DAG** Directed Acyclic Graph
- **DoS** Denial of Service
- **GUI** Graphical User Interface
- **HTTP** Hypertext Transfer Protocol
- **IoT** Internet of Things
- **ILP** Interledger Protocol
- **IP** Internet Protocol
- **IPFS** Interplanetary Filesystem
- **IRI** IOTA Reference Implementation
- **JSON** JavaScript Object Notation
- **LTS** Long Term Support
- **M2M** Machine to Machine
- **MAM** Masked Authenticated Messaging
- **NoSQL** No Structured Query Language
- **OOP** Object Oriented Programming
• OS Operating System
• OTP One-Time Pad
• P2P Peer to Peer
• PBFT Practical Byzantine Fault Tolerance
• PKI Public Key Infrastructure
• PoET Proof of Elapsed Time
• PoS Proof of Stake
• PoW Proof of Work
• RAAM Random Access Authenticated Messaging
• RAM Random Access Memory
• REST REpresentational State Transfer
• RFC Request For Comments
• TCP Transmission Control Protocol
• UDP User Datagram Protocol
• URL Uniform Resource Locator
• YAML YAML Ain’t Markup Language
10 Glossary

- **Access Control List**  Policy that controls access and usage of the Blockchain Platform.
- **Accountability**  Logging and tracking the customer payments.
- **Agreement**  Data Martkplace Asset that contains the details of the purchased goods by a customer.
- **Asset**  Hyperledger Composer entity that can be queried and updated using transactions data.
- **Atomic Instruction**  Instruction that is executed without being interrupted by other processes.
- **Benchmarking**  Performance measuring of a system or platform.
- **BigchainDB**  Blockchain Platform based on a NOSQL database that validates transactions using a consensus algorithm.
- **BigchainDB Enhancement Proposal**
- **Block**  Group of validated transactions in a ledger.
- **Bundle**  Block in IOTA’s terminology
- **Certificate Authority**  Entity that creates certificates for the users of a Permissioned Blockchain.
- **Chaincode**  Smart Contract synonym
- **Channel**  Medium used by two peers or organizations to communicate securely.
- **Confidentiality**  Characteristic of a system that determines that only the authorized entities have access to information.
- **Connection Card**  Hyperledger Composer file that contains the credentials of an user of the Blockchain Network.
- **Consensus Algorithm**  Algorithm that is executed by the nodes of a Blockchain network to validate Blocks.
• **Consortium**  Group of companies in a Hyperledger Fabric architecture that agree to follow a certain policy, without interfering other organizations.

• **Container**  Software that packages different services and its dependencies to run in an Operating System, and lets developers run it in any machine without needing to install anything.

• **Convector**  Alternative framework to Hyperledger Composer.

• **Core file**  Copy (dump) of the memory when a fatal error happens.

• **CronJob**  NodeJS library that lets developers configure a job to be run at certain moments using timers.

• **Dataset**  Data Marketplace asset created by the owner of a stream of data, which contains information about the characteristics of the stream.

• **Denial of Service**  Attack that pretends to overwhelm a system or platform to make it fail by creating huge amounts of requests.

• **Digital twins**  Digital representation of a physical object

• **Directed Acyclic Graph**  Structure of the IOTA ledger.

• **Distributed Database**  Database paradigm in which the information is distributed between different nodes.

• **Double Spending**  Attack to a Blockchain network that pretends to spend the same funds several times.

• **Ethereum Virtual Machine**  Software ran by a device that is part of the Ethereum Blockchain Network.

• **Hash**  One way function that transforms a variable length piece of data into a fixed length output; its strength is based on how hard it is to guess two inputs causing the same output.

• **Fiat Currency**  Currency which value is not determined by its properties, but by its regulation.

• **Fork**  Creation of a Branch in a Blockchain that happens when two nodes validate two valid transactions at the same time.
• **Genesis Block**  First transaction in a Blockchain network.

• **Heap Overflow**  Software design flaw that causes a process to fulfill its heap space.

• **Htop**  Linux utility that shows CPU and memory usage, and information about the processes running on it.

• **Hyperledger Event**  Message sent back to the Hyperledger client with information about the results of the execution of a Smart Contract.

• **Hyperledger Fabric**  Blockchain framework that permits the deployment of a Permissioned Blockchain architecture.

• **Hyperledger Caliper**  Hyperledger benchmarking tool.

• **Hyperledger Composer**  Hyperledger product that provides an abstraction layer to simplify the development of a Blockchain Platform.

• **Integrity**  Characteristic of a system that ensures that data is only modified by authorized entities.

• **Internet of Things**  Computing paradigm in which sensors communicate and share information.

• **Interplanetary Filesystem**  Blockchain platform meant to provide decentralized file storage to its users.

• **IOTA**  Feeless Public Blockchain platform meant for Machine to Machine communications.

• **Jitter**  Benchmarking metric that measures how deviated are the latency times from a sample.

• **Latency**  Time between an action and its result

• **Ledger**  Registry that contains all the transactions sent by its Blockchain users.

• **Machine to Machine**  Type of communication between machines, in which there is no human interaction involved.

• **Marketplace**  Platform developed to provide its users a place to exchange goods.
• **Masked Authenticated Messaging**
• **Maximum Load Test**  Testing methodology in which transactions are sent, using a constant frequency.
• **Merkel Tree**  Data structure of a MAM Channel.
• **Metadata**  Additional data of a BigchainDB asset.
• **Mining**  Activity accomplished by computers to find a valid Hash that validates a Block in PoW Blockchains.
• **MongoDB**  Open source and cross platform NOSQL database framework.
• **Neighbour**  Peer synonym.
• **Offering**  Data Marketplace asset that contains the Pricing and the Dataset information of a stream of data.
• **One Time Pad**  Characteristic of a Crypt
• **Orderer**  Type of node in the Hyperledger Fabric Architecture, which function is to order and append validated Blocks to the ledger.
• **Peer**  Node of a Blockchain network.
• **Peer to Peer**  Distributed systems paradigm in which nodes are connected between themselves without a central point.
• **Permissioned Blockchain**  Blockchain Platform that requires the permission from a Certificate Authority before accessing and using its resources.
• **Postman**  GUI based HTTP REST API Client.
• **Pricing Model**  Attributes that define the way of Payment of an Acquisition in the Data Marketplace.
• **Private Blockchain**  Synonym of Permissioned Blockchain.
• **Promise**  Returned value by a NodeJS function to let the code be ran asynchronously without waiting for the value; because NodeJS is single-threaded.
• **Proof of Work**  Consensus Algorithm in which nodes apply hashes to Blocks and an incremental value until they find a valid Hash.

• **Proof of Stake**  Consensus Algorithm in which the node validating a Block is determined by a random distribution.

• **Public Blockchain**  Blockchain platform in which anyone can join without permission.

• **Public Key Infrastructure**  Infrastructure based on generating a key pair to encrypt and sign messages.

• **Quorum**  A Permissioned Blockchain framework.

• **Repudiation**  Characteristic of a system that should be avoided if Accountability and Integrity are desired.

• **Request For Comments**  Document that suggests the developers group of a system a change in their algorithms or architecture.

• **Resiliency**  Attribute assigned to a platform or system that identifies how often it stops working or fails.

• **Scalability**  Attribute assigned to a platform or system that identifies how good it adapts to an increase on its pending tasks.

• **Smart Contract**  Code stored and executed by a node of a Blockchain network when a transaction is received with its direction as the destination.

• **State of the Art**  Common practices followed by researchers, implementers and standard organizations.

• **Stress testing**  Type of System testing in which high loads of requests are involved to check its behaviour in extreme situations.

• **Tangle**  Name of the IOTA’s DAG structure

• **Throughput**  Benchmarking metric based on measuring how much time takes the system to accomplish a certain task.

• **Timestamp**  Recorded date in a transaction, informs on the date and hour in which the transaction was sent.

• **Tip**  Unvalidated transaction in the Tangle
• **Token**  Unit with value that is shared between the users of a Blockchain platform.

• **Transaction**  Message that contains payment or data information that is sent to be attached to the Blockchain.

• **Troika**  IOTA’s Hashing Algorithm

• **Wallet**  A pair of Public and Private keys that contain the funds of an user and lets him send them or receive more.

• **Zero Knowledge**  Type of Blockchain in which the wallet Public address does not need to change, because no information of the Private address is used during the transaction signing by the sender.
A Full Listing of the Benchmarking Test Outputs

A.1 IOTA

- Test 1: 10 threads sending each a bundle of 10 transactions in parallel.

  1 bundle of 1 transactions sent in 101.467938 seconds.
  1 bundle of 1 transactions sent in 605.402478 seconds.
  1 bundle of 1 transactions sent in 607.346862 seconds.
  1 bundle of 1 transactions sent in 616.870448 seconds.
  1 bundle of 1 transactions sent in 618.153728 seconds.
  1 bundle of 1 transactions sent in 619.899701 seconds.
  1 bundle of 1 transactions sent in 620.595761 seconds.
  1 bundle of 1 transactions sent in 628.678923 seconds.
  1 bundle of 1 transactions sent in 638.683686 seconds.
  1 bundle of 1 transactions sent in 660.021966 seconds.
  1 bundle of 1 transactions sent in 662.789615 seconds.

  Sent 10 transactions using 10 threads with the

- Test 2: 3 threads sending each a bundle of 5 transactions in parallel.

  1 bundle of 5 transactions sent in 33.265483 seconds.
  1 bundle of 5 transactions sent in 26.594568 seconds.
  1 bundle of 5 transactions sent in 62.486228 seconds.

  Sent 15 transactions using 3 threads with the
  http://gemini.ls.fi.upm.es:14265 node in 68.271374 seconds.

- Test 3: 20 threads sending each a bundle of 5 transactions in parallel

  1 bundle of 5 transactions sent in 33.265483 seconds.
  1 bundle of 5 transactions sent in 26.594568 seconds.
  1 bundle of 5 transactions sent in 62.486228 seconds.
  1 bundle of 5 transactions sent in 106.004428 seconds.
  1 bundle of 5 transactions sent in 132.551028 seconds.
  1 bundle of 5 transactions sent in 211.588102 seconds.
  1 bundle of 5 transactions sent in 231.875923 seconds.
  1 bundle of 5 transactions sent in 308.021444 seconds.
  1 bundle of 5 transactions sent in 329.749698 seconds.
  1 bundle of 5 transactions sent in 323.035586 seconds.
  1 bundle of 5 transactions sent in 413.623194 seconds.
1 bundle of 5 transactions sent in 490.819584 seconds.
1 bundle of 5 transactions sent in 532.466585 seconds.
1 bundle of 5 transactions sent in 575.772903 seconds.
1 bundle of 5 transactions sent in 597.138278 seconds.
1 bundle of 5 transactions sent in 652.061102 seconds.
1 bundle of 5 transactions sent in 675.101751 seconds.
1 bundle of 5 transactions sent in 703.865784 seconds.
1 bundle of 5 transactions sent in 724.994909 seconds.
1 bundle of 5 transactions sent in 761.451821 seconds.
Sent 100 transactions using 20 threads with the http://gemini.ls.fi.upm.es:14265 node in 834.357216 seconds.

• Test 4: 6 threads sending each a bundle of 5 transactions in parallel

1 bundle of 5 transactions sent in 87.463639 seconds.
1 bundle of 5 transactions sent in 34.825387 seconds.
1 bundle of 5 transactions sent in 71.540109 seconds.
1 bundle of 5 transactions sent in 93.371046 seconds.
1 bundle of 5 transactions sent in 124.622739 seconds.
1 bundle of 5 transactions sent in 146.25323 seconds.
Sent 100 transactions using 6 threads with the http://gemini.ls.fi.upm.es:14265 node in 151.237219 seconds.

A.2 BigchainDB

• Test 1: 1 thread sending 1000 transactions

Thread 1 sent 1000 TXs in 19.762731313705444.

• Test 2: 4 threads sending 1000 transactions

Thread 3 sent 1000 TXs in 28.253794193267822.
Thread 1 sent 1000 TXs in 28.481404066085815.
Thread 2 sent 1000 TXs in 28.389081239700317.
Thread 4 sent 1000 TXs in 28.552282571792603.

• Test 3: 8 threads sending 1000 transactions

Thread 4 sent 1000 TXs in 46.40973901748657.
Thread 5 sent 1000 TXs in 46.23326086997986.
Thread 7 sent 1000 TXs in 46.793193340301514.
Thread 8 sent 1000 TXs in 47.75743079185486.
Thread 2 sent 1000 TXs in 47.3529531417847.
Thread 3 sent 1000 TXs in 47.172476291656494.
Thread 6 sent 1000 TXs in 47.40624737739563.
Thread 1 sent 1000 TXs in 47.49143147468567.

• Test 4: 1 thread sending 10000 transactions
  Sent 10000 TXs in 198.58342623710632.

• Test 5: 4 threads sending 10000 transactions
  Thread 3 sent 10000 TXs in 407.68281269073486.
  Thread 2 sent 10000 TXs in 406.7007451057434.
  Thread 4 sent 10000 TXs in 406.968636517334.
  Thread 1 sent 10000 TXs in 407.7531893253326.

• Test 6: 8 threads sending 10000 transactions
  Thread 1 sent 10000 TXs in 496.9110617637634.
  Thread 2 sent 10000 TXs in 499.95609736442566.
  Thread 6 sent 10000 TXs in 499.8045325279236.
  Thread 1 sent 10000 TXs in 498.5615575313568.
  Thread 5 sent 10000 TXs in 498.75094628334045.
  Thread 7 sent 10000 TXs in 501.2780599594116.
  Thread 4 sent 10000 TXs in 501.356949862134.
  Thread 3 sent 10000 TXs in 501.7128527164459.

• Test 7: 1000 transactions are distributed into 4 threads: each thread sending 250 transactions
  Thread 4 sent 250 TXs in 5.756422519683838.
  Thread 1 sent 250 TXs in 6.21304445724487.
  Thread 2 sent 250 TXs in 6.23151159286499.
  Thread 3 sent 250 TXs in 6.287359714508057.

• Test 8: 1000 transactions are distributed into 8 threads: each thread sending 125 transactions

116
Thread 3 sent 125 TXs in 3.6155333518981934.
Thread 2 sent 125 TXs in 3.7474279403686523.
Thread 8 sent 125 TXs in 3.7580418587309957.
Thread 6 sent 125 TXs in 3.6547281742095947.
Thread 7 sent 125 TXs in 3.7379231452941895.
Thread 4 sent 125 TXs in 3.740588903427124.
Thread 5 sent 125 TXs in 3.799983263015747.
Thread 1 sent 125 TXs in 3.803626298904419.

- Test 9: 10000 transactions are distributed into 4 threads: each thread sending 2500 transactions

  Thread 3 sent 2500 TXs in 65.00176525115967.
  Thread 2 sent 2500 TXs in 65.05246210098267.
  Thread 4 sent 2500 TXs in 65.22917866706848.
  Thread 1 sent 2500 TXs in 65.37714195251465.

- Test 10: 10000 transactions are distributed into 8 threads: each thread sending 1250 transactions

  Thread 8 sent 1250 TXs in 37.44527268409729.
  Thread 7 sent 1250 TXs in 37.88453459739685.
  Thread 3 sent 1250 TXs in 37.98678560253296.
  Thread 5 sent 1250 TXs in 38.24751162528992.
  Thread 4 sent 1250 TXs in 38.25858783721924.
  Thread 2 sent 1250 TXs in 38.32812976837158.
  Thread 1 sent 1250 TXs in 38.141833782196045.
  Thread 6 sent 1250 TXs in 38.29628038406372.

A.3 MongoDB

- Test 1: 1 thread sending 1000 transactions

  Sent 1000 tx in 0.4935538768763105.

- Test 2: 4 threads sending 1000 transactions

  Thread 1 sent 1000 tx in 1.9742801189422607.
  Thread 4 sent 1000 tx in 1.9865937232971191.
  Thread 2 sent 1000 tx in 1.9918169975280762.
  Thread 3 sent 1000 tx in 1.996716022491455.
• Test 3: 8 threads sending 1000 transactions

<table>
<thead>
<tr>
<th>Thread</th>
<th>Transactions</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 6</td>
<td>1000</td>
<td>3.894458770751953</td>
</tr>
<tr>
<td>Thread 7</td>
<td>1000</td>
<td>3.9018516540527344</td>
</tr>
<tr>
<td>Thread 4</td>
<td>1000</td>
<td>3.9266326427459717</td>
</tr>
<tr>
<td>Thread 2</td>
<td>1000</td>
<td>3.951295852661133</td>
</tr>
<tr>
<td>Thread 1</td>
<td>1000</td>
<td>3.9578065872192383</td>
</tr>
<tr>
<td>Thread 3</td>
<td>1000</td>
<td>3.9557883739471436</td>
</tr>
<tr>
<td>Thread 5</td>
<td>1000</td>
<td>3.966186046600342</td>
</tr>
<tr>
<td>Thread 8</td>
<td>1000</td>
<td>3.978071451187134</td>
</tr>
</tbody>
</table>

• Test 4: 1 thread sending 100000 transactions

Sent 100000 tx in 43.36928343772888.

• Test 5: 4 threads sending 100000 transactions

<table>
<thead>
<tr>
<th>Thread</th>
<th>Transactions</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 4</td>
<td>100000</td>
<td>201.05299258232117</td>
</tr>
<tr>
<td>Thread 1</td>
<td>100000</td>
<td>201.01920533180237</td>
</tr>
<tr>
<td>Thread 2</td>
<td>100000</td>
<td>201.5405616760254</td>
</tr>
<tr>
<td>Thread 3</td>
<td>100000</td>
<td>201.47231793403625</td>
</tr>
</tbody>
</table>

• Test 6: 8 threads sending 100000 transactions

<table>
<thead>
<tr>
<th>Thread</th>
<th>Transactions</th>
<th>Time (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 5</td>
<td>100000</td>
<td>384.05588364601135</td>
</tr>
<tr>
<td>Thread 1</td>
<td>100000</td>
<td>384.6131503582001</td>
</tr>
<tr>
<td>Thread 2</td>
<td>100000</td>
<td>385.36736249923706</td>
</tr>
<tr>
<td>Thread 7</td>
<td>100000</td>
<td>385.13842582702637</td>
</tr>
<tr>
<td>Thread 3</td>
<td>100000</td>
<td>385.7435586452484</td>
</tr>
<tr>
<td>Thread 4</td>
<td>100000</td>
<td>385.5197112560272</td>
</tr>
<tr>
<td>Thread 6</td>
<td>100000</td>
<td>385.397765901184</td>
</tr>
<tr>
<td>Thread 8</td>
<td>100000</td>
<td>385.69354724884033</td>
</tr>
</tbody>
</table>

A.4 Hyperledger

A.4.1 Normal Behaviour Given the Amount of Transactions

• Test 1: Create 1 Dataset, 1 Offering and 4 One Time Payment Agreements (1/16 of normal use):
Created 1 Offerings in 1.285 s. AverageLatency: 1.285 s.
Throughput: 0.7782101167315175 tx/s.
Accepted 4 Agreements in 5.12 s. AverageLatency: 1.28 s.
Throughput: 0.78125 tx/s.
Confirmed 4 Payments in 5.115 s. AverageLatency: 1.27875 s.
Throughput: 0.7820136852394917 tx/s.
Created 1 Datasets in 1.282 s. AverageLatency: 1.282 s.
Throughput: 0.7800312012480499 tx/s.
Created 4 Acquisitions in 13.032 s. AverageLatency: 3.258 s.
Throughput: 0.3069367710251688 tx/s.
Validated 14 Tx in 3.268 s. Whole Throughput:
4.283965728274174 tx/s.

• Test 2: Create 4 Datasets, 4 Offerings and 16 One Time Payment Agreements (1/8 of normal use):

Created 4 Offerings in 17.954 s. Jitter: 94.07266694777323.
AverageLatency: 4.4885 s. Throughput: 0.2227915784783335 tx/s.
Accepted 16 Agreements in 73.208 s. Jitter: 74.1323141416751.
AverageLatency: 4.5755 s. Throughput: 0.21855534914217026 tx/s.
Confirmed 16 Payments in 70.094 s. Jitter: 27.825348155953055.
AverageLatency: 4.380875 s. Throughput: 0.22826490141809572 tx/s.
AverageLatency: 4.42975 s. Throughput: 0.22574637394886843 tx/s.
Created 16 Acquisitions in 84.819 s. Jitter: 933.7101776425774.
AverageLatency: 5.3011875 s. Throughput: 0.18863697992195144 tx/s.
Validated 56 Tx in 6.49 s. Whole Throughput: 8.628659476117104 tx/s.

• Test 3: Create 8 Datasets, 8 Offerings and 32 One Time Payment Agreements (1/4 of normal use):

Created 8 Offerings in 72.051 s. Jitter: 694.0185130301435.
AverageLatency: 9.006375 s. Throughput: 0.11103246311640366 tx/s.
Accepted 32 Agreements in 288.376 s. Jitter: 161.26875984897325.
AverageLatency: 9.01175 s. Throughput: 0.11096623852192972 tx/s.


Validated 112 Tx in 10.689 s. Whole Throughput: 0.47806155861657 tx/s.

Test 4: Create 16 Datasets, 16 Offerings and 64 One Time Payment Agreements (1/2 of normal use):

Created 16 Offerings in 272.2 s. Jitter: 357.8787504169534. AverageLatency: 17.0125 s. Throughput: 0.05878030859662013 tx/s.

Accepted 64 Agreements in 1139.673 s. Jitter: 427.7491705738157. AverageLatency: 17.807390625 s. Throughput: 0.056156458914092024 tx/s.


Created 64 Acquisitions in 1130.625 s. Jitter: 347.64566398632394. AverageLatency: 17.666015625 s. Throughput: 0.05660585959093422 tx/s.

Validated 224 Tx in 18.884 s. Whole Throughput: 0.86189366659606 tx/s.

Test 5: Create 32 Datasets, 32 Offerings and 128 One Time Payment Agreements (normal use):


Accepted 128 Agreements in 4477.992 s. Jitter: 773.9670764529141. AverageLatency: 34.9843125 s. Throughput: 0.028584240436338428 tx/s.

Confirmed 128 Payments in 4258.044 s. Jitter:
A.4.2 Normal Behaviour Maximum Load Testing

- Test 1: Creating 1 Dataset every 1000 ms, 1 Offering every 1000 ms and 1 Agreement every 1000 ms. 4 Agreements are created in each round: (normal1s)


  Accepted 417 Agreements in 739.376 s. Jitter: 576.9916428503197. AverageLatency: 1.773083932853717 s. Throughput: 0.648410071942446 tx/s.


  Created 105 Datasets in 98.54 s. Jitter: 590.748304715138. AverageLatency: 0.9384761904761906 s. Throughput: 1.0655571341587171 tx/s.


  Validated 448 Tx in 36.425 s. Whole Throughput: 12.299245024021964 tx/s.

- Test 2: Creating 1 Dataset every 500 ms, 1 Offering every 500 ms and 1 Agreement every 500 ms. 4 Agreements are created in each round: (2normal1s)

Accepted 808 Agreements in 1045.695 s. Jitter: 611.0892317693267. AverageLatency: 1.2941769801980196 s. Throughput: 0.77269184609279 tx/s.


Validated 2827 Tx in 609.207 s. Whole Throughput: 4.640458825981973 tx/s.

- Test 3: Creating 1 Dataset every 200 ms, 1 Offering every 200 ms and 1 Agreement every 200 ms. 4 Agreements are created in each round:


Accepted 1997 Agreements in 1933.466 s. Jitter: 326.9705034768016. AverageLatency: 0.9681852779168753 s. Throughput: 1.032860158906337 tx/s.


Validated 6990 Tx in 605.634 s. Whole Throughput: 11.541624149238649 tx/s.

- Test 4: Creating 1 Dataset every 185 ms, 1 Offering every 185 ms and 1 Agreement every 185 ms. 4 Agreements are created in each round:

Created 534 Offerings in 559.309 s. Jitter: 360.3210722372485. AverageLatency: 1.0473951310861422 s. Throughput:
0.954749521284299 tx/s.

Accepted 2133 Agreements in 2424.121 s. Jitter: 351.0360588293386. AverageLatency: 1.1364842944210034 s. Throughput: 0.8799065723204411 tx/s.


Validated 7470 Tx in 600.141 s. Whole Throughput: 12.447074937389715 tx/s.

- Test 5: Creating 1 Dataset every 175 ms, 1 Offering every 175 ms and 1 Agreement every 175 ms. 4 Agreements are created in each round: (175ms1normal)


Accepted 1174 Agreements in 26649.442 s. Jitter: 93365.32778764893. AverageLatency: 227.12899659284497 s. Throughput: 0.004402784386850508 tx/s.


Validated 4104 Tx in 608.226 s. Whole Throughput: 6.747491886239654 tx/s.

- Test 6: Creating 1 Dataset every 160 ms, 1 Offering every 160 ms and 1 Agreement every 160 ms. 4 Agreements are created in each round: (160ms1normal)

Created 59 Offerings in 9386.834 s. Jitter:
165082.79682768643. AverageLatency: 159.0988813559322 s. Throughput: 0.006285399315679813 tx/s.
Accepted 239 Agreements in 40725.398 s. Jitter: 171652.61865479176. AverageLatency: 170.39915481171548 s. Throughput: 0.005868573709212123 tx/s.
Validated 832 Tx in 606.334 s. Whole Throughput: 1.372181009146774 tx/s.

• Test 7: Creating 1 Dataset every 150 ms, 1 Offering every 150 ms and 1 Agreement every 150 ms. 4 Agreements are created in each round: (150ms1normal)

Created 40 Offerings in 7083.798 s. Jitter: 211824.44340113955. AverageLatency: 177.09494999999998 s. Throughput: 0.005646688400770322 tx/s.
Accepted 159 Agreements in 29051.454 s. Jitter: 208068.85152165493. AverageLatency: 182.71354716981133 s. Throughput: 0.005473047923866392 tx/s.
Validated 557 Tx in 732.374 s. Whole Throughput: 0.7605403796420954 tx/s.

• Test 8: Creating 1 Dataset every 100 ms, 1 Offering every 100 ms and 1 Agreement every 100 ms. 4 Agreements are created in each round: (100ms1normal)

124
A.4.3 One Asset of each type Maximum Load Testing

- Test 1: Creating 1 Asset every 1000 ms, 1 Offering every 1000 ms and 1 Agreement every 1000 ms. (1 asset1s)

- Test 2: Creating 1 Asset every 500 ms, 1 Offering every 500 ms and 1
Agreement every 500 ms.(2asset1s)

   AverageLatency: 1.4588034398034397 s. Throughput: 0.6854933109663772 tx/s.
Accepted 406 Agreements in 668.263 s. Jitter:
   622.7213644461214. AverageLatency: 1.6459679802955667 s.
   Throughput: 0.6075452329397258 tx/s.
Confirmed 406 Payments in 668.278 s. Jitter:
   622.7286708563672. AverageLatency: 1.6460049261083745 s.
   Throughput: 0.6075315961321486 tx/s.
Created 407 Datasets in 466.737 s. Jitter: 622.7917624054028.
   AverageLatency: 1.1467739557739558 s. Throughput: 0.8720114325626638 tx/s.
Created 406 Acquisitions in 668.621 s. Jitter:
   622.789895812284. AverageLatency: 1.6468497536945812 s.
   Throughput: 0.607315961321486 tx/s.
Validated 2032 Tx in 613.405 s. Whole Throughput: 3.312656401561774 tx/s.

• Test 3: Creating 1 Asset every 333,33 ms, 1 Offering every 333,33 ms
   and 1 Agreement every 333,33 ms.(3asset1s)

Created 598 Offerings in 710.72 s. Jitter: 502.0152911725949.
   AverageLatency: 1.188494983277592 s. Throughput: 0.8414002701485817 tx/s.
Accepted 598 Agreements in 510.575 s. Jitter:
   502.09988213437794. AverageLatency: 0.8538043478260869 s.
   Throughput: 1.1712285168682368 tx/s.
Confirmed 598 Payments in 510.483 s. Jitter:
   502.0309872429955. AverageLatency: 0.8536505016722408 s.
   Throughput: 1.1714395974008929 tx/s.
   AverageLatency: 1.522943143812709 s. Throughput: 0.656623330908643 tx/s.
Created 598 Acquisitions in 510.992 s. Jitter:
   502.0558063208865. AverageLatency: 0.8545016722408028 s.
   Throughput: 1.170272724426214 tx/s.
Validated 2990 Tx in 601.196 s. Whole Throughput:
   4.973419650164006 tx/s.

• Test 4: Creating 1 Asset every 250 ms, 1 Offering every 250 ms and 1
   Agreement every 250 ms.(4asset1s)
Test 5: Creating 1 Asset every 200 ms, 1 Offering every 200 ms and 1 Agreement every 200 ms. (5asset1s)

Test 6: Creating 1 Asset every 166.66 ms, 1 Offering every 166.66 ms and 1 Agreement every 166.66 ms.
• Test 7: Creating 1 Asset every 142.85 ms, 1 Offering every 142.85 ms and 1 Agreement every 142.85 ms. (7asset1s)

Validated 6970 Tx in 602.105 s. Whole Throughput: 11.576054010513117 tx/s.

• Test 8: Creating 1 Asset every 125 ms, 1 Offering every 125 ms and 1 Agreement every 125 ms. (8asset1s)

Created 1579 Offerings in 1746.596 s. Jitter:
335.33749980514295. AverageLatency: 1.1061405953134895 s. Throughput: 0.9040442094222133 tx/s.

Accepted 1577 Agreements in 1635.12 s. Jitter: 393.42375836515464. AverageLatency: 1.0368547875713379 s. Throughput: 0.9644552081804394 tx/s.

Confirmed 1578 Payments in 1540.11 s. Jitter: 326.16380550963277. AverageLatency: 0.9759885931558935 s. Throughput: 1.0246021388082671 tx/s.


Validated 7890 Tx in 602.981 s. Whole Throughput: 13.084989410943297 tx/s.

- Test 9: Creating 1 Asset every 117.64 ms, 1 Offering every 117.64 ms and 1 Agreement every 117.64 ms. (8halfasset1s)


Accepted 517 Agreements in 90411.86 s. Jitter: 130056.935623404554 s. AverageLatency: 174.87787234042554 s. Throughput: 0.005718276341179133 tx/s.


Created 516 Acquisitions in 90626.004 s. Jitter: 130478.11825520854. AverageLatency: 175.63179069767443 s. Throughput: 0.005693730024773022 tx/s.

Validated 2584 Tx in 603.147 s. Whole Throughput: 4.28419605834067 tx/s.

- Test 10: Creating 1 Asset every 111.11 ms, 1 Offering every 111.11 ms and 1 Agreement every 111.11 ms. (9asset1s)

Throughput: 0.006208562596121853 tx/s.
Accepted 98 Agreements in 15512.889 s. Jitter: 177329.6674412076. AverageLatency: 158.2947857142857 s. Throughput: 0.0063173274816831345 tx/s.
Created 98 Acquisitions in 16098.89 s. Jitter: 181110.5203659051. AverageLatency: 164.27438775510203 s. Throughput: 0.006087376210409538 tx/s.
Validated 491 Tx in 609.546 s. Whole Throughput: 0.805517549126727 tx/s.

Test 11: Creating 1 Asset every 100 ms, 1 Offering every 100 ms and 1 Agreement every 100 ms. (10asset1s)

Created 82 Offerings in 12810.595 s. Jitter: 176304.4185466718. AverageLatency: 156.2267829268292 s. Throughput: 0.006400951712235068 tx/s.
Accepted 81 Agreements in 12808.69 s. Jitter: 179920.7370349032. AverageLatency: 158.13197530864198 s. Throughput: 0.006323831711127367 tx/s.
Validated 416 Tx in 618.55 s. Whole Throughput: 0.805517549126727 tx/s.

A.5 Decentralized Data Marketplace API

A.5.1 Query Maximum Load Testing

• Sending 5 Queries per second during 10 minutes.
### Sending 10 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>3000, 5.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.801778545s, 9m59.800110312s, 1.668233ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.553645ms, 1.552319ms, 1.787146ms, 1.913231ms, 5.587698ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>609000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>561000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:3000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Sending 20 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>6000, 10.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.901768986s, 9m59.900126273s, 1.642713ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.475945ms, 1.431927ms, 1.740645ms, 1.961878ms, 5.834437ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>1218000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>1122000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:6000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Sending 30 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>12000, 20.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.95178222s, 9m59.950028055s, 1.754165ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.383318ms, 1.359544ms, 1.686862ms, 1.863852ms, 12.227362ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>2436000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>2244000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:12000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Sending 30 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>18000, 30.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.967973073s, 9m59.966678919s, 1.294154ms</td>
</tr>
</tbody>
</table>

131
<table>
<thead>
<tr>
<th>Latencies</th>
<th>mean, 50, 95, 99, max</th>
<th>1.327625ms, 1.303685ms, 1.608351ms, 1.869639ms, 2.391011ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>3654000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>3366000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:18000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending 50 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>30000, 50.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.981283946s, 9m59.98076132s, 1.207814ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.294788ms, 1.287705ms, 1.433622ms, 1.749467ms, 3.607921ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>6090000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>5610000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:30000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending 100 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>60000, 100.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.989988312s, 9m59.989988312s, 1.310638ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.270653ms, 1.275551ms, 1.360028ms, 1.60675ms, 2.962485ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>12180000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>11220000, 187.00</td>
</tr>
<tr>
<td>Success</td>
<td>[ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes</td>
<td>[code:count]</td>
<td>200:60000</td>
</tr>
<tr>
<td>Error Set:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending 200 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests</th>
<th>[total, rate]</th>
<th>120000, 200.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>[total, attack, wait]</td>
<td>9m59.996165315s, 9m59.995009178s, 1.156137ms</td>
</tr>
<tr>
<td>Latencies</td>
<td>[mean, 50, 95, 99, max]</td>
<td>1.243223ms, 1.264982ms, 1.343663ms, 1.484848ms, 5.742593ms</td>
</tr>
<tr>
<td>Bytes In</td>
<td>[total, mean]</td>
<td>24360000, 203.00</td>
</tr>
<tr>
<td>Bytes Out</td>
<td>[total, mean]</td>
<td>22440000, 187.00</td>
</tr>
</tbody>
</table>

132
### Sending 500 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests [total, rate]</th>
<th>300000, 500.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration [total, attack, wait]</td>
<td>9m59.99913647s, 9m59.998064282s, 1.072188ms</td>
</tr>
<tr>
<td>Latencies [mean, 50, 95, 99, max]</td>
<td>1.014677ms, 1.016772ms, 1.12194ms, 1.201323ms, 3.786723ms</td>
</tr>
<tr>
<td>Bytes In [total, mean]</td>
<td>60900000, 203.00</td>
</tr>
<tr>
<td>Bytes Out [total, mean]</td>
<td>56100000, 187.00</td>
</tr>
<tr>
<td>Success [ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes [code:count]</td>
<td>200:300000</td>
</tr>
</tbody>
</table>

### Sending 1000 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests [total, rate]</th>
<th>600000, 1000.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration [total, attack, wait]</td>
<td>10m0.000130445s, 9m59.999064685s, 1.06576ms</td>
</tr>
<tr>
<td>Latencies [mean, 50, 95, 99, max]</td>
<td>926.853s, 936.519s, 1.086099ms, 1.167084ms, 8.466962ms</td>
</tr>
<tr>
<td>Bytes In [total, mean]</td>
<td>121800000, 203.00</td>
</tr>
<tr>
<td>Bytes Out [total, mean]</td>
<td>112200000, 187.00</td>
</tr>
<tr>
<td>Success [ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes [code:count]</td>
<td>200:600000</td>
</tr>
</tbody>
</table>

### Sending 2000 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th>Requests [total, rate]</th>
<th>1200000, 2000.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration [total, attack, wait]</td>
<td>10m0.000522285s, 9m59.999506405s, 1.01588ms</td>
</tr>
<tr>
<td>Latencies [mean, 50, 95, 99, max]</td>
<td>866.915s, 874.122s, 1.038877ms, 1.184446ms, 10.95303ms</td>
</tr>
<tr>
<td>Bytes In [total, mean]</td>
<td>243600000, 203.00</td>
</tr>
<tr>
<td>Bytes Out [total, mean]</td>
<td>224400000, 187.00</td>
</tr>
<tr>
<td>Success [ratio]</td>
<td>100.00%</td>
</tr>
<tr>
<td>Status Codes [code:count]</td>
<td>200:1200000</td>
</tr>
</tbody>
</table>
• Sending 2500 Queries per second during 10 minutes.

Requests [total, rate] 1500000, 2500.00
Duration [total, attack, wait] 10m0.000451784s, 9m59.999642764s, 809.02s
Latencies [mean, 50, 95, 99, max] 831.785s, 795 s, 1.144399ms, 1.586093ms, 41.296908ms
Bytes In [total, mean] 304500000, 203.00
Bytes Out [total, mean] 280500000, 187.00
Success [ratio] 100.00%
Status Codes [code:count] 200:1500000
Error Set:

• Sending 2750 Queries per second during 10 minutes.

Requests [total, rate] 1650001, 2750.00
Duration [total, attack, wait] 10m0.000165759s, 9m59.999427754s, 738.005s
Latencies [mean, 50, 95, 99, max] 967.55s, 849.154s, 1.668827ms, 2.430528ms, 41.6753ms
Bytes In [total, mean] 334950203, 203.00
Bytes Out [total, mean] 308550187, 187.00
Success [ratio] 100.00%
Status Codes [code:count] 200:1650001
Error Set:

• Sending 3000 Queries per second during 10 minutes.

Requests [total, rate] 1800001, 3000.00
Duration [total, attack, wait] 10m0.357787347s, 9m59.999429722s, 358.358127ms
Latencies [mean, 50, 95, 99, max] 317.167108ms, 348.332582ms, 373.820883ms, 713.062145ms, 2.065327416s
Bytes In [total, mean] 352036713, 195.58
Bytes Out [total, mean] 324289977, 180.16
Success [ratio] 96.34%
Status Codes [code:count] 0:65830 200:1734171
Error Set:
Post http://localhost:1031/v1/queryContext: dial tcp 0.0.0.0->127.0.0.1:1031: socket: too many open files

• Sending 3250 Queries per second during 10 minutes.
Requests [total, rate] 1950001, 3250.00
Duration [total, attack, wait] 10m0.419421172s, 9m59.99945188s, 419.969292ms
Latencies [mean, 50, 95, 99, max] 307.513276ms, 351.182934ms, 371.265692ms, 710.831744ms, 1.446610877s
Bytes In [total, mean] 354378521, 181.73
Bytes Out [total, mean] 326447209, 167.41
Success [ratio] 89.52%
Status Codes [code:count] 0:204294 200:1745707
Error Set:
Post http://localhost:1031/v1/queryContext: dial tcp 0.0.0.0:0->127.0.0.1:1031: socket: too many open files

• Sending 3500 Queries per second during 10 minutes.

Requests [total, rate] 2100002, 3500.00
Duration [total, attack, wait] 10m0.412147988s, 9m59.999728489s, 412.419499ms
Latencies [mean, 50, 95, 99, max] 289.333994ms, 349.092627ms, 366.907807ms, 701.347405ms, 1.681938995s
Bytes In [total, mean] 356483834, 169.75
Bytes Out [total, mean] 328386586, 156.37
Success [ratio] 83.62%
Status Codes [code:count] 0:343924 200:1756078
Error Set:
Post http://localhost:1031/v1/queryContext: dial tcp 0.0.0.0:0->127.0.0.1:1031: socket: too many open files

• Sending 4000 Queries per second during 10 minutes.

Requests [total, rate] 2400000, 4000.00
Duration [total, attack, wait] 10m0.422957222s, 9m59.999756346s, 423.200876ms
Latencies [mean, 50, 95, 99, max] 254.157009ms, 342.21284ms, 359.546353ms, 674.557787ms, 1.404880326s
Bytes In [total, mean] 361749045, 150.73
Bytes Out [total, mean] 333236805, 138.85
Success [ratio] 74.25%
Status Codes [code:count] 0:617985 200:1782015
Error Set:
Post http://localhost:1031/v1/queryContext: dial tcp 0.0.0.0:0->127.0.0.1:1031: socket: too many open files
- Sending 4500 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th></th>
<th>Requests [total, rate] 2700002, 4500.01</th>
<th>Duration [total, attack, wait] 10m0.351318833s, 9m59.999678013s, 351.64082ms</th>
<th>Latencies [mean, 50, 95, 99, max] 226.09652ms, 334.552657ms, 353.452117ms, 611.222774ms, 1.68825416s</th>
<th>Bytes In [total, mean] 366990302, 135.92</th>
<th>Bytes Out [total, mean] 338064958, 125.21</th>
<th>Success [ratio] 66.96%</th>
<th>Status Codes [code:count] 0:892168 200:1807834</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Set:</td>
<td>Post <a href="http://localhost:1031/v1/queryContext">http://localhost:1031/v1/queryContext</a>: dial tcp 0.0.0.0:0-&gt;127.0.0.1:1031: socket: too many open files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending 5000 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th></th>
<th>Requests [total, rate] 3000000, 5000.00</th>
<th>Duration [total, attack, wait] 10m0.434538903s, 9m59.99981882s, 434.720083ms</th>
<th>Latencies [mean, 50, 95, 99, max] 203.52816ms, 323.63656ms, 345.1725ms, 482.878124ms, 1.35800568s</th>
<th>Bytes In [total, mean] 376004720, 115.33</th>
<th>Bytes Out [total, mean] 346368880, 115.46</th>
<th>Success [ratio] 61.74%</th>
<th>Status Codes [code:count] 0:1147760 200:1852240</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Set:</td>
<td>Post <a href="http://localhost:1031/v1/queryContext">http://localhost:1031/v1/queryContext</a>: dial tcp 0.0.0.0:0-&gt;127.0.0.1:1031: socket: too many open files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sending 10000 Queries per second during 10 minutes.

<table>
<thead>
<tr>
<th></th>
<th>Requests [total, rate] 6000000, 10000.00</th>
<th>Duration [total, attack, wait] 10m0.431130392s, 9m59.999999199s, 431.131193ms</th>
<th>Latencies [mean, 50, 95, 99, max] 101.828812ms, 0s, 312.674237ms, 324.881238ms, 1.667176827s</th>
<th>Bytes In [total, mean] 405149836, 67.52</th>
<th>Bytes Out [total, mean] 373216844, 62.20</th>
<th>Success [ratio] 33.26%</th>
<th>Status Codes [code:count] 0:4004188 200:1995812</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Set:</td>
<td>Post <a href="http://localhost:1031/v1/queryContext">http://localhost:1031/v1/queryContext</a>: dial tcp 0.0.0.0:0-&gt;127.0.0.1:1031: socket: too many open files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

136
A.5.2 Data Updates Maximum Load Testing

- Send one Data Update every 125 ms

  Received 343 Data Updates in 600.471630964355 ms. Jitter: 0.5622460525456984. AverageLatency: 1.7557650002819754. Throughput: 0.569552303204332.

- Send one Data Update every 250 ms

  Received 349 Data Updates in 601.0672330856323 ms. Jitter: 0.5410343162198715. AverageLatency: 1.7272046927748055. Throughput: 0.578970173126076.

- Send one Data Update every 500 ms

  Received 345 Data Updates in 601.9271459579468 ms. Jitter: 0.5625150391032845. AverageLatency: 1.7497882149940314. Throughput: 0.5714977340862998.

- Send one Data Update every 1000 ms

  Received 317 Data Updates in 598.7996640205383 ms. Jitter: 0.6586257228635077. AverageLatency: 1.894935645634615. Throughput: 0.5277224069871246.

- Send one Data Update every 1500 ms

  Received 307 Data Updates in 599.5232539176941 ms. Jitter: 0.6902746446166765. AverageLatency: 1.9592263199924644. Throughput: 0.5104055564156805.

- Send one Data Update every 2000 ms

  Received 300 Data Updates in 599.6489310264587 ms. Jitter: 0.7126995636786336. AverageLatency: 2.0055148194864842. Throughput: 0.49862508632873226.

- Send one Data Update every 2500 ms
• Send one Data Update every 3000 ms

Received 240 Data Updates in 601.2223372459412 ms. Jitter: 0.7389290570835394. AverageLatency: 2.515574632828206. Throughput: 0.39752348705938484.

• Send one Data Update every 3000 ms

Received 201 Data Updates in 600.9760456085205 ms. Jitter: 0.05868388262816985. AverageLatency: 3.0048802280426026. Throughput: 0.33279196643767933.

B Hyperledger Error Messages and Causes

This section is meant to provide the reason behind the common errors thrown during the deployment of the Hyperledger network, as they are not explained officially anywhere and it is hard to find solutions on the Hyperledger chat. The full listings of the errors may be different to the obtained ones, but it should be clear which kind of error raised by looking for similarities on the error output given. These errors may be raised due to different reasons not discussed here, if it is not solved after checking this appendix and applying the suggested solutions, look for an answer on the Internet or ask in: https://chat.hyperledger.org/home

• Not installed dependencies when creating the .bna file.
  – When the error happens: When trying to create the .bna file.
  – Cause of the error: In order to create the .bna file using the following command:

  ```
  $ composer archive create --sourceType dir --sourceName .
  -a conwet-network.bna
  ```

  The developer may have forgotten to install the node modules that need to be installed to create the bna file. Check that the package.json is correct and references the right libraries.

  – Full listing of the error:

  Creating Business Network Archive
Looking for package.json of Business Network Definition

Input directory:
/home/usuario/Escritorio/BlockchainBAE/bna-network

Error: npm dependency path
/home/usuario/Escritorio/BlockchainBAE/bna-network/composer-common
does not exist. Did you run npm install?

Command failed

- Error: Uncompatible Composer-Runtime and Composer-Client versions.

  - When the error happens: When the developer tries to ping a network card or connect to a Hyperledger Composer network using the Composer Client NodeJS API.

  - Cause of the error: There are two possible causes of this error:

    * The Business Network (.bna file) deployed, references the 0.19 version; it may happen when you are moving to an updated environment.

    * The composer-cli or composer-playground or composer-runtime node modules are not updated to the same version.

  - Solution: Make sure to delete all node modules, reinstall the new versions and check their version with:

    $ composer --version

Before recreating the .bna file as suggested in the following command, make sure you have no references to previous composer versions; recreating it will not solve the issue, because references to old software in the package.json file will be kept, as we can see in the package.json file, the composer-runtime-hlfv1 is referencing the 0.19.14 version:

```json
{
  "name": "conwet-network",
  "author": "author",
  "description": "Start from scratch with a blank business network",
  "version": "0.0.2-deploy.265",
  "devDependencies": {
    "browserfs": "~1.2.0",
    "chai": "~3.5.0"
  }
}
```
"composer-admin": "latest",
"composer-cli": "latest",
"composer-client": "latest",
"composer-connector-embedded": "latest",
"eslint": ""3.6.1",
"istanbul": ""0.4.5",
"jsdoc": ""3.4.1",
"mkdirp": ""0.5.1",
"mocha": ""3.2.0",
"moment": ""2.19.3"
},
"keywords": [],
"license": "Apache 2.0",
"repository": {
  "type": "e.g. git",
  "url": "URL"
},
"scripts": {
  "deploy": "./scripts/deploy.sh",
  "doc": "jsdoc --pedantic --recursive -c jsdoc.conf",
  "lint": "eslint .",
  "postlicchk": "npm run doc",
  "postlint": "npm run licchk",
  "prepublish": "mkdirp ./dist && composer archive create --sourceType dir --sourceName . -a ./dist/unnamed-network.bna",
  "pretest": "npm run lint",
  "test": "mocha --recursive",
  "start": "start-network"
},
"dependencies": {
  "composer-common": ""0.20.8",
  "composer-runtime-hlfv1": ""0.19.14"
} } 

Also make sure you have updated the .bna file. Delete and recreate it using the following command from the folder where the Business Network files are stored:

```
$ composer archive create --sourceType dir --sourceName .
   -a <BNAFILENAME.bna>
```

- Full listing of the error:
Error: Error trying to ping. Error: Composer runtime (0.19.14) is not compatible with client (0.20.8)
Command failed

- Error: Unable to read required file: metadata.json
  - When the error happens: after running the `composer card list` command.
  - Cause of the error: The composer command cannot find the metadata.json files of the different cards stored in the Hyperledger wallet. This file contains all the information that is shown after issuing the command.
  - Solution: make sure the Hyperledger Composer wallet (by default stored in: /.composer), is not corrupted, that is, has the /cards directory and inside each card directory a metadata.json appears.

  If the .composer wallet was sent to a remote server using `scp`, maybe the metadata.json file in each card folder got zipped, and the composer cannot parse (understand) its contents.

  Try deleting `/composer` and starting from zero, the wallet will be created from zero, after issuing the first `composer card create` command:

  $ rm -rf ~/.composer

- Full listing of the error

  Error: Unable to read required file: metadata.json
  Command failed

- Error: 2 UNKNOWN: transaction returned with failure: SyntaxError
  - When the error happens: after running the `composer network start` command.
  - Cause of the error: the .bna file has Syntax mistakes, so that, it cannot be instantiated. Find the errors using the composer web playground uploading your .bna file and editing it.
  - Solution: Fix the errors, export the fixed .bna file and try to install.
Full listing of the error:

Starting business network definition. This may take a minute...
Error: Error trying to start business network. Error: No valid responses from any peers.
Response from attempted peer comms was an error: Error: 2
UNKNOWN: transaction returned with failure:
SyntaxError: Identifier 'Crypto' has already been declared
Response from attempted peer comms was an error: Error: 2
UNKNOWN: transaction returned with failure:
SyntaxError: Identifier 'Crypto' has already been declared
Response from attempted peer comms was an error: Error: 2
UNKNOWN: transaction returned with failure:
SyntaxError: Identifier 'Crypto' has already been declared
Response from attempted peer comms was an error: Error: 2
UNKNOWN: transaction returned with failure:
SyntaxError: Identifier 'Crypto' has already been declared

SSL-ERROR-SSL: Handshake failed with fatal error

- When the error happens: after running the composer network start command.
- Cause of the error: this problem was solved after changing the composer version to match one that is supported by the version of the Hyperledger Fabric Software used in the deployment. Not every version of Hyperledger Composer (composer commands), are compatible with the Hyperledger Fabric Software running in the docker containers.
- Solution: the Hyperledger Fabric version used during this research was the 1.1, the version of Hyperledger Composer that is compatible with Fabric 1.1 is the v0.19 (in concrete v0.19.14). The error was raised because the composer version was the 0.20, which is not compatible with Fabric 1.1. Get the version of Hyperledger Composer by running:

```
$ composer --version
```
Check the Hyperledger Fabric version running in the containers by running:

```
$ docker image ls hyperledger/fabric*
```

All the containers should be running the same Fabric version.

- Full listing of the error:

  Starting business network definition. This may take a minute...
  E0910 10:38:01.079213962 4444
  E0910 10:38:01.081067352 4444
  E0910 10:38:01.081271737 4444
  E0910 10:38:01.081432409 4444
  E0910 10:38:01.081594169 4444
  E0910 10:38:01.081754856 4444
  Starting business network definition. This may take a minute...
  E0910 10:38:01.081915523 4444
  E0910 10:38:01.081915523 4444
  E0910 10:38:01.081915523 4444
  E0910 10:38:01.081915523 4444
  E0910 10:38:01.081915523 4444
  E0910 10:38:01.081915523 4444
Error: Error trying to start business network. Error: Unable to initialize channel. Attempted to contact 4 Peers. Last error was Error: Error: Failed to connect before the deadline
Command failed

- **ssl-transport-security.cc**: Could not load any root certificate
  - When the error happens: after running the `composer network install` command.
  - Cause of the error: The certificates of the different entities (peers, orderers...) have a wrong format or are incorrect. This may caused because the developer followed the official Hyperledger Composer docs about how to deploy a Multi Organization network [], which are incorrect. The `awk` commands will not get the certificates from the different peers and orderers in the right way, because the "new line" characters are converted to "\n" as different characters.
  - Solution: Instead of using the `awk` commands to copy the contents of the certificates to a file, it is suggested to use the following commands, which will get the contents of the certificate right. In case it does not solve your error, make sure you have inserted the certificates in the `connection.json` file that is used to setup the network, and that they are right, that is, that you did not mixed
the certificates or introduced the wrong one or with additional characters.

- Full listing of the error:

```
Installing business network. This may take a
minute...E0912 18:33:29.579030547 6736
ssl_transport_security.cc:599] Could not load any
root certificate.
E0912 18:33:29.579072084 6736
ssl_transport_security.cc:1400] Cannot load server
root certificates.
E0912 18:33:29.579088242 6736
security_connector.cc:1025] Handshaker factory creation failed with
TSI_INVALID_ARGUMENT.
E0912 18:33:29.579115435 6736
secure_channel_create.cc:111] Failed to create secure
subchannel for secure name 'localhost:7051'
E0912 18:33:29.579122785 6736
secure_channel_create.cc:142] Failed to create
subchannel arguments during subchannel creation.
E0912 18:33:29.57953521 6736
ssl_transport_security.cc:599] Could not load any
root certificate.
E0912 18:33:29.579561595 6736
ssl_transport_security.cc:1400] Cannot load server
root certificates.
E0912 18:33:29.579578363 6736
security_connector.cc:1025] Handshaker factory creation failed with
TSI_INVALID_ARGUMENT.
E0912 18:33:29.579583252 6736
secure_channel_create.cc:111] Failed to create secure
subchannel for secure name 'localhost:8051'
E0912 18:33:29.579587336 6736
secure_channel_create.cc:142] Failed to create
subchannel arguments during subchannel creation.
E0912 18:33:29.580026483 6736
ssl_transport_security.cc:599] Could not load any
root certificate.
E0912 18:33:29.580034334 6736
ssl_transport_security.cc:1400] Cannot load server
root certificates.
E0912 18:33:29.580050876 6736
security_connector.cc:1025] Handshaker factory creation failed with
TSI_INVALID_ARGUMENT.
```
Failed to create secure subchannel for secure name 'localhost:9051'.
Failed to create subchannel arguments during subchannel creation.
Failed to create secure subchannel for secure name 'localhost:10051'.
Failed to create secure subchannel for secure name 'localhost:7051'.
Could not load any root certificate.
Cannot load server root certificates.
Handshaker factory creation failed with TSI_INVALID_ARGUMENT.
Failed to create secure subchannel for secure name 'localhost:10051'.
Failed to create subchannel arguments during subchannel creation. Installing business network. This may take a minute...
Could not load any root certificate.
Cannot load server root certificates.
Handshaker factory creation failed with TSI_INVALID_ARGUMENT.
Failed to create secure subchannel for secure name 'localhost:7051'.
Failed to create secure subchannel for secure name 'localhost:10051'.
Could not load any root certificate.
Cannot load server root certificates.
TSI_INVALID_ARGUMENT.
E0912 18:33:32.936137914 6736
  secure_channel_create.cc:111] Failed to create secure
  subchannel for secure name 'localhost:8051'
E0912 18:33:32.936141840 6736
  secure_channel_create.cc:142] Failed to create
  subchannel arguments during subchannel creation.

• No valid responses from any peers. Error: 2 UNKNOWN: chaincode error

  – When the error happens: after running the composer network
    start command.

  – Cause of the error: the name or the version of the Composer
    Network given to the command do not match the name or version
    of the .bna file.
    That is caused because before starting the network, the composer
    software checks that, both the version and name match the prop-
    erties of the given .bna file.

  – Solution: make sure that the given version and name given to the
    composer network start command are the same as the given to
    the .bna file.
    To get the version of a .bna file run:

    $ composer archive list -a [.bna FILE]

    To see which chaincodes were installed in each of the entities
    (peers, orderers...), connect to the desired container and run an
    ls command in the chaincodes directory, this process can be ac-
    complished using the following commands:

    $ docker exec -ti [DOCKER CONTAINER NAME] bash
    # ls /var/hyperledger/production/chaincodes/

  – Full listing of the error:

    Starting business network trade-network at version 0.1.14

    Processing these Network Admins:
        userName: alice
        userName: bob

    147
Starting business network definition. This may take a minute...

Error: Error trying to start business network. Error: No valid responses from any peers.
Response from attempted peer comms was an error: Error: 2
UNKNOWN: chaincode error (status: 500, message: cannot get package for chaincode
(trade-network:0.1.14))
Response from attempted peer comms was an error: Error: 2
UNKNOWN: chaincode error (status: 500, message: cannot get package for chaincode
(trade-network:0.1.14))
Response from attempted peer comms was an error: Error: 2
UNKNOWN: chaincode error (status: 500, message: cannot get package for chaincode
(trade-network:0.1.14))
Response from attempted peer comms was an error: Error: 2
UNKNOWN: chaincode error (status: 500, message: cannot get package for chaincode
(trade-network:0.1.14))
Command failed

C Congested BigchainDB Docker Logs

Traceback (most recent call last):
  File "/usr/src/app/bigchaindb/backend/localmongodb/connection.py", line 68, in run
    return query.run(self.conn)
  File "/usr/src/app/bigchaindb/utils.py", line 176, in run
    last = last(*item[0], **item[1])
  File "/usr/local/lib/python3.6/site-packages/pymongo/collection.py", line 1262, in find_one
    for result in cursor.limit(-1):
  File "/usr/local/lib/python3.6/site-packages/pymongo/cursor.py", line 1189, in next
    if len(self.__data) or self._refresh():
  File "/usr/local/lib/python3.6/site-packages/pymongo/cursor.py", line 1087, in _refresh
The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "src/gevent/greenlet.py", line 766, in 
    gevent._greenlet.Greenlet.run
  File "src/gevent/baseserver.py", line 26, in 
    _handle_and_close_when_done
      return handle(*args_tuple)
  File "src/gevent/protocol.py", line 171, in 
    _handle_connection
      response = self.protocol.process(req_type, message)
  File "src/gevent/protocol.py", line 50, in 
    process
      return handler(req)
  File "src/abci/server.py", line 77, in 
    result = self.app.deliver_tx(req.deliver_tx.tx)
  File "src/bigchaindb/core.py", line 184, in 
    decode_transaction(raw_transaction), self.block_transactions)
D  BigchainDB Python driver error message trying to connect to congested node

Traceback (most recent call last):
 File "/usr/local/lib/python3.5/dist-packages/urllib3/connectionpool.py", line 384, in _make_request
 six.raise_from(e, None)
 File "/usr/local/lib/python3.5/dist-packages/urllib3/connectionpool.py", line 380, in _make_request
 httplib_response = conn.getresponse()
During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/usr/local/lib/python3.5/dist-packages/requests/adapters.py", line 449, in send
    timeout=timeout
  File "/usr/local/lib/python3.5/dist-packages/urllib3/connectionpool.py", line 638, in urlopen
    _stacktrace=sys.exc_info()[2])
    raise six.reraise(type(error), error, _stacktrace)
  File "/usr/local/lib/python3.5/dist-packages/urllib3/connectionpool.py", line 600, in urlopen
    chunked=chunked)
  File "/usr/local/lib/python3.5/dist-packages/urllib3/connectionpool.py", line 386, in _make_request
    self._raise_timeout(err=e, url=url, timeout_value=read_timeout)
    raise ReadTimeoutError(self, url, "Read timed out. (read timeout=%s)" % timeout_value)
urllib3.exceptions.ReadTimeoutError:
    HTTPConnectionPool(host='localhost', port=9984): Read timed out. (read timeout=20)

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/usr/lib/python3.5/threading.py", line 914, in _bootstrap_inner
    self.run()
  File "bigchain_send_th.py", line 55, in run
    bdb.transactions.send_async(fulfilled_token_txList[i])
  File "/usr/lib/python3.5/site-packages/bigchaindb_driver/driver.py", line 337, in send_async
    headers=headers)
  File "/usr/lib/python3.5/site-packages/bigchaindb_driver/transport.py", line 82, in forward_request
    backoff_cap=backoff_cap,
  File "/usr/lib/python3.5/site-packages/bigchaindb_driver/connection.py", line 91, in request
    **kwargs,
  File "/usr/lib/python3.5/site-packages/bigchaindb_driver/connection.py", line 120, in _request
    response = self.session.request(**kwargs)
  File "/usr/local/lib/python3.5/dist-packages/requests/sessions.py", line 533, in request
    resp = self.send(prep, **send_kwargs)
  File "/usr/local/lib/python3.5/dist-packages/requests/adapters.py", line 529, in send
    raise ReadTimeout(e, request=request)
requests.exceptions.ReadTimeout:
    HTTPConnectionPool(host='localhost', port=9984): Read timed out. (read timeout=20)
E  IOTA Benchmarking tool

from iota import *
import sys
from threading import Thread
import datetime

SEED1 = b"INSERT VALID SEED!!"
ADDRESS_WITH_CHECKSUM_SECURITY_LEVEL_2 = b"INSERT VALID ADDRESS FROM SEED!!"
api = Iota('https://pow1.iota.community:443', seed = SEED1)

class benchmark():
    def __init__(self):
        if len(sys.argv[1:]) != 4:
            print("Error Numero invalido de argumentos")
            print("Uso: python benchmark.py NUMTHREADS URI NUMBUNDLES NUMTXPERBUNDLE"
        sys.exit(1)

        self.nthreads = int(sys.argv[1])
        self.uri = sys.argv[2]
        self.nbundles = int(sys.argv[3])
        self.ntxperbundle = int(sys.argv[4])

    def sendTx(self):
        tx = ProposedTransaction(
            address =
                Address(ADDRESS_WITH_CHECKSUM_SECURITY_LEVEL_2),
            value = 0,
            tag = Tag(b'EXAMPLE'),
            message = TryteString.from_string('temperatura: 123 Fahrenheit :D')
        )

        l_bundle = [ProposedBundle() for i in range(self.nbundles)]

        for i in range(self.nbundles):
            for j in range(self.ntxperbundle):
                l_bundle[i].add_transaction(tx)
            l_bundle[i].finalize()
        api.send_trytes(trytes=l_bundle[i].as_tryte_strings(),
                depth=10)
if __name__ == "__main__":
    startProgram = datetime.datetime.now()

    bm = benchmark()
    bm.sendTx()

    th_list = [Thread(target = bm.sendTx) for ii in range(bm.nthreads)]

    for th in th_list:
        th.start()

    for th in th_list:
        th.join()

    endProgram = datetime.datetime.now()

    print("Sent {} transactions using {} threads with the {} node in
    {} seconds.".format(bm.nbundles * bm.ntxperbundle * bm.nthreads,
                        bm.nthreads,
                        bm.uri,
                        (endProgram - startProgram).total_seconds()))

F  BigchainDB Benchmarking tool

    import time
    import threading

    from bigchaindb_driver import BigchainDB
    from bigchaindb_driver.crypto import generate_keypair

    bdb_root_url = 'localhost:9984' # Use YOUR BigchainDB Root URL here
    bdb = BigchainDB(bdb_root_url) #Without auth tokens, second param
                             #for the tokens.

    alice, bob = generate_keypair(), generate_keypair()

    numTh = 8
    numTx = 10
class BigchainTh(threading.Thread):
    def run(self):
        global numTx
        assetList = []
        txList = []
        fulfilled_token_txList = []

        for i in range(numTx):
            assetList.append(
                {'data':
                    {'Temperature': 'aajd'+self.getName(),
                     'Humidity': i,
                     'Location': 'UPM-CoNWetLab'}
                }
            )
        
        for i in range(numTx):
            txList.append(bdb.transactions.prepare(
                operation='CREATE',
                signers=alice.public_key,
                recipients=[[(bob.public_key), 10]],
                asset=assetList[i]
            ))
        
        for i in range(numTx):
            fulfilled_token_txList.append(bdb.transactions.fulfill(
                txList[i],
                private_keys=alice.private_key
            ))
        
        start = time.time()

        for i in range(numTx):
            startTx = time.time()
            bdb.transactions.send_async(fulfilled_token_txList[i])
            endTx = time.time()
            #print("Sent a transaction in {} .".format(endTx-startTx))
end = time.time()

print("Thread {} sent {} TXs in {} .".format(self.getName(),
numTx, end - start))

def main():
    for i in range(numTh):
        myTh = BigchainTh(name = i)
        myTh.start()

if __name__ == '__main__':
    main()

G MongoDB Benchmarking tool

import time
import threading

from pymongo import MongoClient

client = MongoClient("172.17.0.6:27017")
db = client.assets

numTx = 100000
numTh = 8

class MongoTh(threading.Thread):
    def run(self):
        global numTx
        assetList = []
        for i in range(numTx):
            assetList.append(
                {
                    'data': {
                        'Temperature': 12,
                        'Humidity': i,
                        'Location': 'UPM-ETSIINF-CoNWetLab-uu'
                    }
                }
            )

start = time.time()
for i in range(numTx):
    result = db.reviews.insert_one(assetList[i])

end = time.time()

print("Thread {} sent {} tx in {} .".format(self.getName(), numTx, (end - start)))

def main():
    for i in range(numTh):
        myTh = MongoTh(name = i)
        myTh.start()

if __name__ == '__main__':
    main()

H Hyperledger Benchmarking Tool

const DatasetRegistry = require('../datasetRegistry')
const OfferingRegistry = require('../offeringRegistry')
const PaymentCompletedRegistry = require('../paymentCompleted')
const AgreementRegistry = require('../agreementRegistry')

const uuidv4 = require('uuid/v4');
const dataApiUri = "http://localhost:7000"

async function createNDatasets(n) {
    var data = require('./benchmark-assets/dataset')
    var req;
    var ids = []
    for(var i = 0; i < n; i++){
        ids.push(uuidv4().substr(0,6))
    }
    //console.log(ids)
    for (var i = 0; i < n; i++) {
        //console.log("UUUGH")
        var d = new DatasetRegistry()
        var jobId = uuidv4();
        data.$class = "org.conwet.biznet.CreateDataset"
        data.jobId = jobId
data.datasetId = ids[i]
req = {
  uri: dataApiUri + '/datasets/',
  body: data,
  method: 'POST',
  headers: {
    'Content-Type': "application/json"
  }
};
await d.createDataset(req, jobId)
}

async function createNOfferings(n){
  var data = require('./benchmark-assets/offering')
  var req;
  var ids = []
  for(var i = 0; i < n; i++){
    ids.push(uuidv4().substr(0,6))
  }
  //console.log(ids)
  for (var i = 0; i < n; i++) {
    var o = new OfferingRegistry()
    var jobId = uuidv4();
    data.$class = "org.conwet.biznet.CreateOffering"
    data.jobId = jobId
    data.offeringId = ids[i]
    req = {
      uri: dataApiUri + '/offerings/',
      body: data,
      method: 'POST',
      headers: {
        'Content-Type': "application/json"
      }
    }; //console.log("LLAMADA")
    await o.createOffering(req, jobId)
  }
}

async function createPaymentCompleted(){
  //console.log("pag")
  var data = require('./benchmark-assets/paymentcompleted')
  var jobId = uuidv4()
  data.jobId = jobId
var p = new PaymentCompletedRegistry();
var start = new Date()
var simulateTime = 1000
//console.log(data)
var start = new Date()
await p.completePayment(data, jobId)
}

async function createAcquisition(){
  var data = require('./benchmark-assets/onetimeaqc')
  var req;
  var a = new AgreementRegistry()
  var jobId = uuidv4();
  data.$class = "org.conwet.biznet.MakeAgreement"
  data.jobId = jobId
  data.agreementId = uuidv4().substr(0,6)
  req = {
    uri: dataApiUri + '/acquisitions/',
    body: data,
    method: 'POST',
    headers: {
      'Content-Type': "application/json"
    }
  };
  await a.createAgreement(req, jobId)
}

async function createAcceptAgreement(){
  var a = new AgreementRegistry()
  var start = new Date()
  var simulateTime = 1000
  var jobId = uuidv4();
  var start = new Date()
  await a.acceptAgreement("aydear5",jobId)
}

async function createNOneTimeAgreements(n) {
  for(var i = 0; i < n; i++){
    await createAcquisition()
    await createPaymentCompleted()
    await createAcceptAgreement()
  }
}

159
var args = process.argv.slice(2)
if (args.length % 2 != 0) {
    console.log("ERR")
    process.exit(1)
}
args = args.reduce(function(result, value, index, array) {
    if (index % 2 === 0)
        result.push(array.slice(index, index + 2));
    return result;
}, []);

for (var i = 0; i < args.length; i++) {
    if (args[i][0] == '-d') {
        createNDatasets(args[i][1])
    } else if (args[i][0] == '-o') {
        createNOfferings(args[i][1])
    } else if (args[i][0] == '-ao') {
        createNOneTimeAgreements(args[i][1])
    } else if (args[i][0] == '-as') {
        createNSubscriptionAgreements(args[i][1])
    } else if (args[i][0] == '-au') {
        createNUusageAgreements(args[i][1])
    }

//console.log(args)

I Hyperledger Benchmarking Log Parser

import sys
import statistics

f = open("log", "r")

offerTimes = []
dataTimes = []
acquiTimes = []
payTimes = []
acceptTimes = []
avgOffer = 0
gAvgData = 0
gAvgAcqui = 0
avgAccept = 0
gAvgPay = 0

for l in f.readlines():
    if "Offering" in l:
        words = l.split()
        offerTimes.append(int(words[-1][:-2]))
    avg = 0
    for elem in offerTimes:
        avgOffer += elem

    if "Dataset" in l:
        words = l.split()
        dataTimes.append(int(words[-1][:-2]))
        for elem in dataTimes:
            avgData += elem

    if "Acquisition" in l:
        words = l.split()
        acquiTimes.append(int(words[-1][:-2]))
        for elem in acquiTimes:
            avgAcqui += elem

    if "Accepted" in l:
        words = l.split()
        acceptTimes.append(int(words[-1][:-2]))
        for elem in acceptTimes:
            avgAccept += elem

    if "Payment" in l:
        words = l.split()
        payTimes.append(int(words[-1][:-2]))
        for elem in payTimes:
            avgPay += elem

print("Offer: ", format(offerTimes))
nprint("Data: ", format(dataTimes))
nprint("Acqui: ", format(acquiTimes))
nprint("Accept: ", format(acceptTimes))
nprint("Pay: ", format(payTimes))
print("Created {} Offerings in {} ms. Jitter: {}. AverageLatency: 
{}\). Throughput: {}.\".format( 
len(offerTimes), sum(offerTimes), statistics.stdev(offerTimes), 
sum(offerTimes)/len(offerTimes), 
len(offerTimes)/sum(offerTimes)))
print("Accepted {} Agreements in {} ms. Jitter: {}. AverageLatency: 
{}\). Throughput: {}.\".format( 
len(acceptTimes), sum(acceptTimes), 
statistics.stdev(acceptTimes), 
sum(acceptTimes)/len(acceptTimes), 
len(acceptTimes)/sum(acceptTimes)))
print("Confirmed {} Payments in {} ms. Jitter: {}. AverageLatency: 
{}\). Throughput: {}.\".format( 
len(payTimes), sum(payTimes), statistics.stdev(payTimes), 
sum(payTimes)/len(payTimes), len(payTimes)/sum(payTimes)))
print("Created {} Datasets in {} ms. Jitter: {}. AverageLatency: 
{}\). Throughput: {}.\".format( 
len(dataTimes), sum(dataTimes), statistics.stdev(dataTimes), 
sum(dataTimes)/len(dataTimes), len(dataTimes)/sum(dataTimes)))
print("Created {} Acquisitions in {} ms. Jitter: {}. AverageLatency: 
{}\). Throughput: {}.\".format( 
len(acquiTimes), sum(acquiTimes), statistics.stdev(acquiTimes), 
sum(acquiTimes)/len(acquiTimes), len(acquiTimes)/sum(acquiTimes)))

J Data Marketplace Benchmarking Tool

var request = require('request')
var updateBody = require('./bodyUpdate.json')

var val = 0
var orgName = "org1"

function updateData(){
  if(val == 65000)
    val = 0

  updateBody.rain.value = val;
  val++;

  var cuerpoStr = JSON.stringify(updateBody)
var brokerTransaction = {
    body: JSON.stringify(updateBody),
    method: 'PATCH',
    headers: {
        'Content-Type': 'application/json,
        'Fiware-Service': orgName,  //TODO: It is not received in the query through the channel. D:
        'Content-Length': cuerpoStr.length
    }
}

request(brokerTransaction)

setInterval(updateData, 1500)

K Data Marketplace Log Parser

from datetime import datetime
import sys
import statistics

f = open("sublog", "r")
dataTimes = []
dataLines = []

for l in f.readlines():
    if "DATA RECEIVED" in l:
        dataLines.append(l.split()[-1])

for v, w in zip(dataLines[:-1], dataLines[1:]):
    t1 = datetime.strptime(v, '%H:%M:%S.%f')
    t2 = datetime.strptime(w, '%H:%M:%S.%f')
    print("ESTO: \{\}".format(t2-t1))
    dataTimes.append(t2.timestamp() - t1.timestamp())

avgData = sum(dataTimes)/len(dataTimes)
print(avgData)
print("Received {} Data Updates in {} ms. Jitter: {}. AverageLatency: {}. Throughput: {}.").format(len(dataTimes)+1, sum(dataTimes), statistics.stdev(dataTimes), sum(dataTimes)/len(dataTimes), len(dataTimes)/sum(dataTimes)))

L Data Marketplace Docker-Compose YAML file

version: '3'

networks:
    test:
        external: true

services:
    redis:
        build:
            context: .
            dockerfile: Dockerfile-Redis
        networks:
            test:
        ports:
            - "6379:6379"

app:
    build:
        context: .
        dockerfile: Dockerfile-NodeModules
    depends_on:
        - redis
        - bigchaindb
        - mongodb
    networks:
        test:
    ports:
        - "7000/tcp:7000/tcp"
        - "8000/tcp:8000/tcp"

orion:
    image: fiware/orion
    depends_on:
mongodb:
  image: mongo:3.6
  networks:
  - test:
    ports:
    - "1026:1026"
  command: -dbhost mongodb

command: mongod
restart: always

bigchaindb:
  depends_on:
  - mongodb
  - tendermint
  image: bigchaindb/bigchaindb
  networks:
  - test:
  volumes:
  - ./bigchaindb/bigchaindb:/usr/src/app/bigchaindb
  - ./bigchaindb/tests:/usr/src/app/tests
  - ./bigchaindb/docs:/usr/src/app/docs
  - ./bigchaindb/htmlcov:/usr/src/app/htmlcov
  - ./bigchaindb/setup.py:/usr/src/app/setup.py
  - ./bigchaindb/setup.cfg:/usr/src/app/setup.cfg
  - ./bigchaindb/pytest.ini:/usr/src/app/pytest.ini
  - ./bigchaindb/tox.ini:/usr/src/app/tox.ini
  environment:
  BIGCHAINDB_DATABASE_BACKEND: localmongodb
  BIGCHAINDB_DATABASE_HOST: mongodb
  BIGCHAINDB_DATABASE_PORT: 27017
  BIGCHAINDB_SERVER_BIND: 0.0.0.0:9984
  BIGCHAINDB_WSSERVER_HOST: 0.0.0.0
  BIGCHAINDB_WSSERVER_ADVERTISED_HOST: bigchaindb
  BIGCHAINDB_TENDERMINT_HOST: tendermint
  BIGCHAINDB_TENDERMINT_PORT: 26657
  ports:
  - "9984:9984"
  - "9985:9985"
  - "26658"
command: 'start'
restart: always

tendermint:
  image: tendermint/tendermint:0.22.8
  entrypoint: ''
  networks:
  test:
  ports:
    - "26656:26656"
    - "26657:26657"
  command: sh -c "tendermint init && tendermint node
    --consensus.create_empty_blocks=false
    --proxy_app=tcp://bigchaindb:26658"
restart: always

tendermint:
  image: tendermint/tendermint:0.22.8
  entrypoint: ''
  networks:
  test:
  ports:
    - "26656:26656"
    - "26657:26657"
  command: sh -c "tendermint init && tendermint node
    --consensus.create_empty_blocks=false
    --proxy_app=tcp://bigchaindb:26658"
restart: always

ca.example.com:
  image: hyperledger/fabric-ca
  environment:
    - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
    - FABRIC_CA_SERVER_CA_NAME=ca.example.com
    - FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org1.example.com/ca.crt
    - FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/4239aa0dcd76dae4f70c9995f35054c0f6a24d6f
  ports:
    - "7054:7054"
  command: sh -c 'fabric-ca-server start -b admin:adminpw -d'
  volumes:
    - ./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledger/fabric-ca-server-config
  container_name: ca.example.com
  networks:
    - test

orderer.example.com:
  container_name: orderer.example.com
  image: hyperledger/fabric-orderer
  environment:
    - ORDERER_GENERAL_LOGLEVEL=debug
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_GENESISMETHOD=file
    - ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/configtx/genesis.block
    - ORDERER_GENERAL_LOCALMSPID=OrdererMSP

- ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/msp/orderer/msp
  working_dir:
    /opt/gopath/src/github.com/hyperledger/fabric/orderer
  command: orderer
  ports:
    - 7050:7050
  volumes:
    - ./config/:/etc/hyperledger/configtx
    - ./crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/:/etc
    - ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/:/
  networks:
    - test

peer0.org1.example.com:
  container_name: peer0.org1.example.com
  image: hyperledger/fabric-peer
  environment:
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - CORE_PEER_ID=peer0.org1.example.com
    - CORE_LOGGING_PEER=debug
    - CORE_CHAINCODE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_LOCALMSPID=Org1MSP
    - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
  # # the following setting starts chaincode containers on the same
  # # bridge network as the peers
  # # https://docs.docker.com/compose/networking/
  - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_test
  - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
  - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb:5984
  # The CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME and
  # CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD
  # provide the credentials for ledger to connect to CouchDB.
  # The username and password must
  # match the username and password set for the associated CouchDB.
  - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=
  - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric
  command: peer node start
  # command: peer node start --peer-chaincode-dev=true

167
ports:
- 7051:7051
- 7053:7053
volumes:
- /var/run/:/host/var/run/
- ./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/ms
- ./crypto-config/peerOrganizations/org1.example.com/users:/etc/hyperledger/msp/user
- ./config:/etc/hyperledger/configtx
depends_on:
- orderer.example.com
- couchdb
networks:
- test
couchdb:
  container_name: couchdb
  image: hyperledger/fabric-couchdb
  # Populate the COUCHDB_USER and COUCHDB_PASSWORD to set an
  # admin user and password
  # for CouchDB. This will prevent CouchDB from operating in an
  # "Admin Party" mode.
  environment:
  - COUCHDB_USER=
  - COUCHDB_PASSWORD=
  ports:
  - 5984:5984
networks:
- test
cli:
  container_name: cli
  image: hyperledger/fabric-tools
  tty: true
  environment:
  - GOPATH=/opt/gopath
  - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
  - CORE_LOGGING_LEVEL=DEBUG
  - CORE_PEER_ID=cli
  - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
  - CORE_PEER_LOCALMSPID=Org1MSP
  - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/p
  - CORE_CHAINCODE_KEEPALIVE=10
working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
command: /bin/bash
volumes:
  - /var/run:/host/var/run/
  - ./../chaincode:/opt/gopath/src/github.com/
  - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
networks:
  - test
#depends_on:
#  - orderer.example.com
#  - peer0.org1.example.com
#  - couchdb
References


[3] Ralph C. Merkle, *A Digital Signature based on a Conventional Encryption Function*, Elxsi, 2334 Lundy Place, San Jose, CA 95131


170
The following references are meant to provide the reader additional materials to better understand how to use, develop and deploy the Blockchain Platforms used in the Decentralized Data Marketplace.

Recommended Books:


The following links contain the official documentation of all the used platforms:


In order to better understand how IOTA and Masked Authenticated Messaging protocol the author recommends having a look at:


To learn how to deploy different Hyperledger Blockchain Networks, the author recommends the following free course:
For further details on how the different Communication Protocols using the IOTA Tangle work:


[23] *RAAM*, https://blog.usejournal.com/random-access-authenticated-messaging-45a5f40f2532

Este documento está firmado por

<table>
<thead>
<tr>
<th>Firmante</th>
<th>CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fecha/Hora</td>
<td>Sat Jun 01 14:33:25 CEST 2019</td>
</tr>
<tr>
<td>Emisor del</td>
<td>EMAILADDRESS=<a href="mailto:camanager@fi.upm.es">camanager@fi.upm.es</a>, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES</td>
</tr>
<tr>
<td>Certificado</td>
<td></td>
</tr>
<tr>
<td>Numero de Serie</td>
<td>630</td>
</tr>
<tr>
<td>Metodo</td>
<td>urn:adobe.com:Adobe.PPKLite:adbe.pkc7.sha1 (Adobe Signature)</td>
</tr>
</tbody>
</table>