UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

Faculty of Science and Technology
Department of Computer Science

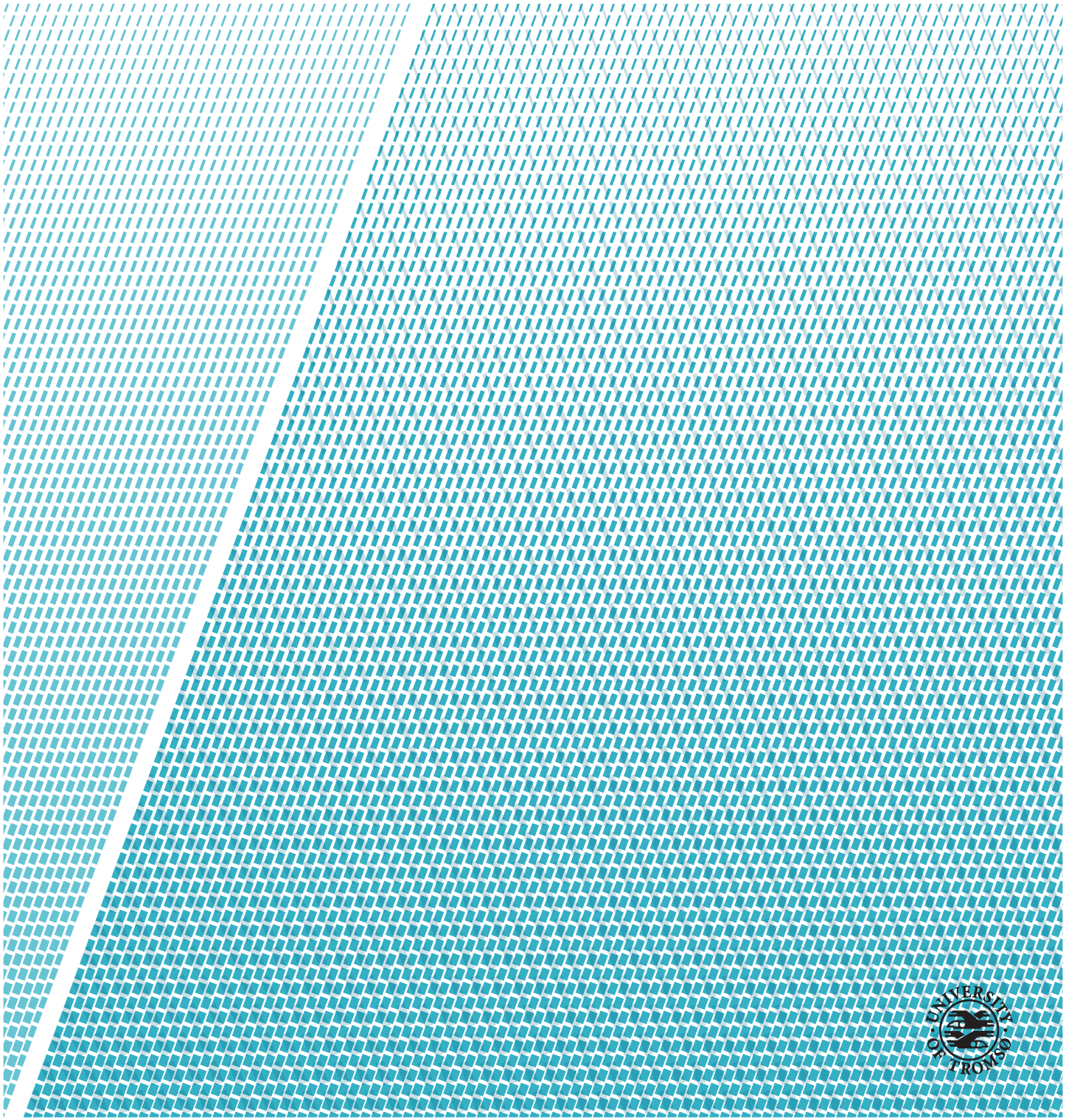
HyperProv

Blockchain-based Data Provenance using Hyperledger Fabric

—

Petter Tunstad

INF-3981 - Master's thesis in Computer Science, May 2019



"Who controls the past controls the future.
Who controls the present controls the past."
–George Orwell, 1984

Abstract

With data intensive computing helping advance state-of-the-art in varied fields, data provenance and lineage continue to remain formidable challenges in assisting with integrity and reproducibility in research and applications. This is particularly challenging for distributed scenarios, where data may be originating from decentralized sources without any centralized control by a single trusted entity. To date most of the data provenance systems are specific to particular domains, and are often centralized. Distributed ledgers such as blockchains have proved quite popular and effective in addressing trust and consensus without central control. There are a few recent proposals to employ blockchains for data provenance, however, they rely on currency in order to propose transactions using public blockchains.

We present HyperProv, a general framework for data provenance based on the permissioned blockchain Hyperledger Fabric (HLF), and to the best of our knowledge, the first provenance system that is ported to ARM based devices such as Raspberry Pi (RPi). HyperProv records the operation history and data lineage by tracking checksums, editors, timestamps, data pointers, dependencies, and more. Provenance data is retrieved and stored through a NodeJS client library to simplify interactions with the blockchain. HyperProv has a set of built-in queries using smart contracts that enable lightweight retrieval of large collections of provenance data. We evaluate the throughput, latency and resource consumption of HyperProv on x86-64 desktop machines, as well as RPi, demonstrating the feasibility of using HyperProv on RPi for tamper-proof data provenance, useful in particular for Internet of Things use cases. Our contributions to HLF for ARM devices have already generated significant uptake and attention in the community, with multiple interactions and more than 500 downloads in less than 3 months.

Acknowledgements

First and foremost I would like to thank my supervisors, Phuong Hoai Ha and Amin Khan for your advice, ideas and feedback during this project and for my time at the Green Computing Group. I miss the cake-fueled Friday meetings.

Thanks to Sunjun Mehedi and Tommy Øines from Arctic Green Computing Group for your input on machine learning model management and helpful talks around the office.

I would also like to express my sincerest gratitude for my fellow students in the class of 2014. Thank you for five great years, both on and off campus. We've come a long way since the teapots.

Finally, I would like to thank my parents for encouraging me, and for your endless love and support.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Problem Definition 2
1.2 Methodology 3
1.3 Previous Work 3
1.4 Hypothesis and Choice of Platform 4
1.5 Summary of Contributions 4
1.6 Outline 5

2 Related Work **7**
2.1 Provenance with Blockchain 7
2.2 Provenance in General 9
2.3 Other related projects using blockchain 10

3 Hyperledger Fabric **13**
3.1 Blockchain 13
3.1.1 Consensus Protocols 14
3.1.2 Smart Contracts 15
3.2 Architecture 15
3.2.1 Endorsement Policies 16
3.3 Docker 17
3.4 Node Client Libraries 17

4 Architecture and Design **19**
4.1 Provenance Metadata 21
4.2 High-level Architecture 21
4.3 Hyperledger Nodes 22

4.3.1 Chaincode Operations . . . 23
4.4 Off-Chain Storage . . . 24
4.5 Client Placement . . . 25
4.6 Availability and Consistency with Network Partitions . . . 26
4.7 System Specification . . . 27
4.7.1 Core Functionality . . . 27
4.7.2 Additional Functionality . . . 27

5 Implementation 29
5.1 Chaincode . . . 29
5.1.1 Data Pointers and Checksums . . . 30
5.1.2 Dependency Linking . . . 31
5.1.3 Identity . . . 32
5.1.4 Historic and Range Queries . . . 32
5.1.5 Pagination . . . 33
5.2 Client Library . . . 33
5.2.1 Exposed API . . . 34
5.2.2 NPM Library . . . 34
5.2.3 REST Client . . . 35
5.3 Case Studies . . . 35
5.3.1 IoT Sensor Data Client . . . 35
5.3.2 Machine Learning Models Client . . . 36
5.4 System Configuration . . . 36
5.4.1 Compose Files . . . 37
5.4.2 Certificate Authority . . . 37
5.4.3 Shared Storage . . . 37
5.5 Building Docker Images . . . 38

6 Evaluation 39
6.1 Methodology . . . 40
6.1.1 Experimental Setup . . . 40
6.1.2 CPU Throttle on 64-bit Raspberry Pi . . . 41
6.1.3 Throughput Measurements . . . 41
6.1.4 Measuring Resource Consumption . . . 41
6.2 Throughput . . . 43
6.2.1 Latency . . . 47
6.3 Resource Consumption . . . 48
6.3.1 Raspberry Pi . . . 51
6.3.2 Energy . . . 52
6.3.3 Network . . . 53
6.4 Edge Computing . . . 55
6.5 Model Metadata Tracking . . . 57

7 Discussion and Concluding Remarks 61

7.1 Discussion 61
7.1.1 Findings 62
7.2 Future Work 63
7.3 Concluding Remarks 64

Bibliography **65**

Appendices **73**

A Appendix A: System Specifications and Configuration **73**

B Appendix B: Building Hyperledger Fabric Docker Images **85**

List of Figures

3.1 Sequence of blocks and fork illustrated. 14
3.2 Transaction flow between Client, Peer and Orderers in Hyperledger Fabric. 16

4.1 The three core components in the Hyperprov System 22
4.2 Hyperledger components in Hyperprov 23
4.3 Off chain storage possibilities 25
4.4 Hyperledger Fabric in the CAP-theorem venn-diagram 26

5.1 JSON object from getDependencies for IoT sensor data visualized. 31
5.2 The software stack for Hyperprov 34

6.1 Component distribution on experimental setup. 40
6.2 Throughput and average response times for increasing load levels. (Desktop) (only provenance data) 43
6.3 Figure 6.2 extended. Throughput for load levels up to 10 000 transactions at a time. Peaking at *3276 tx/min*. 44
6.4 Throughput and average response times for increasing load levels. (RPi) (only provenance data) 44
6.5 Throughput and response times for varying data sizes. (Desktop) 45
6.6 Throughput for 1-3 concurrent client devices. (RPi) 46
6.7 Throughput and response times for varying data sizes. (RPi) 46
6.8 Latency for an increasing number of dependent records retrieved with the *getdependencies* chaincode. (Desktop) . . . 48
6.9 CPU/Memory at max load (~ 3200 tx/min). (Desktop) . . . 49
6.10 CPU and memory for peer process (Desktop) 50
6.11 CPU and memory for client process (Desktop) with external storage. 50
6.12 CPU and memory for peer process (RPi) 51
6.13 CPU and memory for client process (RPi) with external storage. 52
6.14 Energy consumption on RPi. 10-minute intervals 53
6.15 Network consumption without transactions. 54

6.16 Network consumption with and without SSHFS. 55
6.17 IoT gateways illustrated 56
6.18 Resource usage for storing 100 x 94MB ML models. 57
6.19 V8 Profile for storing 100x 94MB models 58
6.20 Network for storing 94MB models 58

List of Tables

5.1 Key operations enabled by the Hyperprov Client Library . . . 34

6.1 Latency of operations to blockchain 47
6.2 Statistics of SET operation latency data 47

1 Introduction

Over the last two decades, the size of data generated and used for research has increased significantly [1], highlighting the importance of data provenance systems [2, 3] in order to ensure the quality and integrity of information, and to counteract accidental or malicious data manipulation or corruption. Data provenance is the process that determines the lineage of data, starting from its original sources. A provenance system can be useful for verifying the integrity of data, tracking its history and recording identities of any of its editors.

Blockchain technology [4] has in the recent years attracted a lot of focus due to its ability to create a tamper-proof, shared decentralized ledger of transactions resillient to byzantine parties. Permissioned blockchains [5, 6] vary from public blockchains by placing more trust in an infrequently changing set of participants, allowing for consensus algorithms with better performance and energy efficiency. One of these permissioned systems is Hyperledger Fabric (HLF) [5], which is a new and promising enterprise-targeted blockchain framework backed by IBM and The Linux Foundation. Evaluations [6] show that HLF [5] compares well with other major private blockchains. Most other provenance solutions are developed for centralized architectures and designed for specific fields [2, 3] while recent provenance systems using blockchain rely on currency-based public blockchains [7, 8].

In this thesis, we present Hyperprov, a blockchain based provenance system using HLF to provide guarantee and lineage of data by storing provenance metadata in a tamper-proof ledger. Hyperprov records the operation history

and data lineage by tracking editors, timestamps of operations, checksums, data locations, dependencies and additional custom metadata. Provenance data is retrieved and stored through a NodeJS client library to simplify the interactions with the blockchain. Hyperprov has a set of built-in queries in the chaincode (smart contracts) that allow for lightweight retrieval of large collections of provenance data. As a distributed system, the aim of Hyperprov is to provide tamper-proof replication of provenance data while ensuring consistency and fault tolerance.

Because permissioned blockchains like HLF do not rely on any heavy computations, unlike proof-of-work blockchain systems, we believe this makes it feasible to use edge devices such as RPi or acting as gateways for devices producing data. By building and releasing Docker images for ARM we hope to pave way for other projects looking to evaluate HLF on edge devices. Our contributions in porting HLF for ARM devices have already generated significant uptake and notice in the community, with multiple interactions, and more than 500 downloads in the past 2-3 months [9].

We evaluate Hyperprov on an experimental setup both on Desktop x86-64 commodity hardware and RPi to evaluate throughput and resource consumption. To the best of our knowledge, this is the first provenance system to feature HLF's first long-term release, and also the first to run a provenance system based on HLF on ARM devices. We have compared our results on desktop machines with those reported for recent systems [7, 8]. For the RPi, our goal is to evaluate and argue for acceptable overhead of HLF-based distributed data provenance systems for IOT devices at the edge.

1.1 Problem Definition

The objective is to explore the landscape of provenance tracking and the capabilities of the HLF framework. The research will propose a system for data provenance, implement a prototype and evaluate for use both on desktop and RPi devices. We will deploy a system running the long-term release of HLF with all current performance, stability and feature improvements. Focus will be on general usability, sufficient performance and low resource consumption. We will evaluate the feasibility of this for IOT edge devices on RPi and compare results against recent systems on commodity desktop hardware.

1.2 Methodology

In accordance with the final report of the ACM Task Force on The Core of Computer Science [10] there are three main paradigms in the discipline on computing. These are theory, abstraction and design.

The first paradigm, **theory**, is rooted in mathematics. This consists of an iterative process involving defining a problem to study, developing one or more theorems, test in order to prove/disprove the theorems and lastly evaluate and interpret the results to determine new factors and make progress in computing.

Abstraction is rooted in experimental scientific models. This involves forming a hypothesis, constructing models and predictions, designing an experiment to collect data and lastly analyze the results.

The last paradigm, **design**, is rooted in engineering. Within this paradigm one seeks to create a system for solving a given problem. This involves stating requirements and specification of the system, designing and implementing the system and lastly evaluating and testing the system.

For this thesis, the last paradigm, *design* seem to be most fitting as we seek to construct a system to enable data provenance. We state a problem, set requirements and specifications and evaluate the system behavior based on an implemented prototype.

1.3 Previous Work

This thesis builds on previous work from a capstone project [11] where we evaluated the possibility of running an early version of HLF on RPi devices. We came across several limitations which we improve upon to create a competitive solution both in terms of functionality and performance. We found that earlier versions of HLF (v1.0) did not have sufficient functionality to implement competitive provenance features. Also, storing all data in the ledger was unfeasible due to performance limitations and the ledger growing unsustainably large. Additionally, we found that accessing network features and interacting with the ledger was too complex and would most likely discourage users from using our system. For this work we will resolve all these limitations and more to implement a new provenance framework we believe to be competitive with existing cutting edge solutions.

1.4 Hypothesis and Choice of Platform

For this thesis we want to focus on three main hypotheses which affect our choice of platform. Our first point (**H1**) is that we believe that the HLF framework (v1.4) is sufficient to provide competitive provenance features using functionality built into smart contracts. Additionally (**H2**) we believe that a system based on HLF can compare against existing provenance solutions on blockchain in terms of performance. Last but not least (**H3**) we believe that the resource overhead is sufficiently small that our system will be feasible to run on RPi for IOT devices at the edge.

To summarize we want to explore (**H1**) functionality, (**H2**) performance and (**H3**) low-overhead. To do this we will conduct evaluations on both commodity desktop hardware and RPi.

1.5 Summary of Contributions

This thesis makes the following contributions:

- We strengthen the viability of running HLF on ARM devices by building deployable Docker images from source and publishing them to Docker Hub [9].
- We research existing provenance systems to discover limitations and common traits that can be implemented and improved upon using a permissioned blockchain framework.
- We present the architecture and implementation of Hyperprov, a system consisting of HLF chaincode and a NodeJS client library to enable storage and retrieval of provenance metadata to a tamperproof blockchain ledger.
- We demonstrate that our system is capable of storing provenance data for real-world scenarios by implementing a collection of applications on top of our client library.
- We experimentally evaluate Hyperprov trough a number of benchmarks to evaluate throughput and latency both on commodity hardware desktop systems and for Raspberry Pi devices. We also evaluate the device resource usage in terms of CPU, memory, network and energy consumption.

1.6 Outline

Chapter 2 gives an overview of related work and related projects in the subjects of Provenance, Blockchain and IoT. It also provides context for comparisons and mentions throughout the thesis.

Chapter 3 describes the key framework on which Hyperprov is built, namely Hyperledger Fabric and presents relevant background material on blockchain.

Chapter 4 presents Hyperprov and how it designed to benefit data provenance and provides an overview of the system architecture.

Chapter 5 discuss the methodology and implementation of the system prototype and HLF setup.

Chapter 6 evaluates the measurements derived from the experimental setups on Desktop and RPi devices. It also evaluates the system for two real-world scenarios.

Chapter 7 Discusses the findings, outlines future work and concludes the thesis

2

Related Work

For this section we will outline some other projects using blockchain for data provenance. We will also look into other more traditional provenance systems and how other systems make use of blockchain technology beyond provenance.

2.1 Provenance with Blockchain

SmartProvenance [8] uses the public blockchain Ethereum [12] to create a set of techniques for secure storage of data trails, access control policies, voting mechanisms and penalty payments to prevent malicious changes. Every change in this system is similarly stored as a new encrypted version of the data. They are able to mimic the Open Provenance Model (OPM) [13] and create an off-chain JavaScript module for user accessibility using MeteorJS. Building a system on a public blockchain yields some guarantees that the ledger is shared across a large number of participants, however the system relies on currency to process transactions. They state that the cost of operation ranges from 0.16 USD to 1.31 USD per change made in the system.

ProvChain [7] also make use of public blockchains, to store and verify provenance of data stored in the cloud. They claim only an average of 6.49 % overhead for storing data with the system. Data is stored using Chainpoint [14] which is responsible for combining hashes of the provenance history into

a Merke-tree [15] to allow more data per block. Transactions return verifiable proofs that are irreversibly anchored in the blockchain. Chainpoint is based on the Bitcoin [16] ledger and require a transaction to be processed before data provenance is stored. This similarly to SmartProvenance [8] is costly but ProvChain [7] propose that cloud service providers charge extra for provenance capabilities and in turn use this to pay for transactions that eventually lead to currency for the Bitcoin miners, keeping the network running.

Another approach from Demichev, Kryukov and Prikhodko [17] suggests using blockchain and more specifically HLF [5] to manage provenance metadata and access control for distributed storage. The paper is targeted towards tracking large amounts of data in studies involving administratively separate organizations where funding or estimating the required storage for projects are problematic. They target a storage model between centralized and Peer-to-peer (P2P) storage, where each participating organization integrates their centralized storage pool into a unified distributed set of storage providers which can be allocated for projects when needed. At this stage they only present a design, claiming a preliminary version prototype created with Hyperledger Composer has been deployed, which may be evaluated in the future.

Vegvisir [18] is a partition-tolerant permissioned blockchain for the IOT focusing on low-network connectivity and power efficiency. They resolve network partitions by allowing them to create forks of the blockchain resulting in a graph structure of the ledger with causal ordering. This limits Vegvisir to only applications based on Conflict-free Replicated Data Types (CRDT) [19] which can be replicated across multiple hosts and updated independently while keeping strong eventual consistency, but at the cost of unique total ordering. Nodes reconcile by periodically asking neighbors for all its blocks with no successors, and if any of these blocks are not known they add them and all their parent blocks if not already present, by doing this the nodes achieve eventual consistency on nodes without successors and gain a causal history of the blocks added. Because malicious nodes may delete new blocks before being propagated, a block is considered persistent only once it has been stored by a k number of different users, this is called proof-of-witness. An android application using this system, targeted towards emergency first responders is currently under construction and may prove to be useful in low connectivity or in ad hoc mobile networks.

2.2 Provenance in General

Herschel et al. [3] defines provenance as the production process of an end product. This could possibly include the actual data, meta-data, processes, activities and/or even people involved in the production or transformation of data. The paper brings up some use cases such as food supply chains, reproducing scientific research or complex data processing e.g. when analyzing or debugging. They also describe an interesting distinction between different types of provenance, namely *provenance metadata*, *information system provenance*, *workflow provenance* and *data provenance*. This forms a hierarchy where *provenance metadata* is the most general and *data provenance* is most specific. *Provenance metadata* describes all types of general-purpose provenance that with high freedom is used to model information about the data derivation process. Following is *information system provenance* which is more specifically meta data about an information disseminating process which can be calculated based on input, output and parameters of a process. *Workflow provenance* further specializes by restricting the processes to directed graphs where nodes and edges represent functions and the data flow between them, this allows for higher resolution provenance as you can leverage all connections of the workflow graphs. The final level of the hierarchy is *data provenance* in which you track individual items and the operations applied to them. Per-object provenance usually models either the history of existing results or explain the absence of missing results.

Simmhan, Plale and Gannon [2] classify several definitions for data provenance. The most notable is the origins of data, the process in which it arrived at the database, resources and transformations applied to derive the data, what processes created it and additional metadata which describe the process. Their survey [2] gives insight into multiple aspects where provenance information could be useful. Lineage of data can e.g. be used to determine quality and reliability. Provenance can also be used to trace the audit trail of data or as a means of replicating to another system, it can be used to strengthen ownership of data or determine liability in the event of faulty records. Lastly provenance metadata could be used in the context of data discovery and or to provide context for understanding data.

The survey [2] evaluates five major works in the field of data provenance and classifies them based on characteristics. Chimera [20] is a script based provenance system for physics and astronomy which collects provenance in the form of data derivation steps to allow for on-demand regeneration of data, simple comparison and audits of derivations. MyGrid [21] provides a middleware layer for biology experiments. Provenance is modeled as workflows in a grid enviroment, this allows for features such as resource discovery, workflow-enactment and metadata/provenance management to better integrate and enhance the

bio-informatics information model. CMCS [22] is a project targeted towards collaborative and metadata-based data management. It has been used for combustion research and uses the Scientific Annotation Middleware for storing URL-referenceable files and collections. The ESSW [23] is a script-based system used by earth science researchers for metadata management and data storage. This system relies on tracing the lineage of data for error detection and for determining the quality of datasets. The sequence of operations performed by a master script forms a Directed Acyclic Graph (DAG) which can be visualized and navigated in a web-browser. Trio [24] is a proposed database system which includes data accuracy and lineage as inherent components. Data is stored as tuples, and lineage is a product of the query and source tuples automatically determined by the system from inverse queries. Common for all five systems is that they mostly rely on a relational database, are not as tamperproof as systems based on blockchain technology and are mostly specified towards specific fields.

2.3 Other related projects using blockchain

Shafagh et al. [25] makes use of blockchain to create a distributed access control and management scheme for IoT data streams. This is made possible by running a virtual chain layer [26] to add system logic on an already existing blockchain. In this case they use Bitcoin [16] because of its security, reliability and current dominance. Because of the specific IoT use case, data is structured in streams and chunked, compressed and encrypted before being stored in an off-chain storage service. Because IoT data typically has a high level of correlation in time, the data is also highly compressible, chunks are then compressed and chained together where only the top chunk needs to be stored in the blockchain due to cryptographic chaining. This makes for efficient storage-use both off chain and in the Bitcoin blockchain where storage typically is slow and costly. The devices used are "in the orders of few MHz of CPU, few 10s of KB of RAM, and few 100s of KB of ROM" [25] and would most likely be too restrained for running their own blockchain like HLF. The paper mentions however, that an "IoT gateway" also could be used as an intermediate node and cache for multiple smaller devices interacting with the off-chain storage.

Nygaard [27] presents a highly scalable architecture for a storage system built using blockchain. The system consists of three types of devices; clients, ledger nodes and storage nodes. The system uses traditional BFT algorithms to provide superior throughput and latency over other consensus methods such as proof-of-work or proof-of-stake. For this they use a blockchain engine built around Tendermint [28] to provide consensus in permissioned environments.

For storage he uses the IPFS [29] distributed P2P file system which has content-addressing so that each item has its unique content identifier. The prototype results show the time required to disseminate varying data sizes across the network for different size networks. The limitation seems to be reliant on sending data to the storage nodes, but if tested with more than one operation the number of ledger nodes may also start to play a role.

Stanciu [30] proposed an interesting use of HLF in a system for hierarchical distributed access control based on the IEC61499 Standard. He targets Edge Computing as an extension to the cloud located closer to the devices. Hyperledger [5] chaincode and Docker [31] is used to implement function blocks and Kubernetes [32] is used to orchestrate execution across all edge resources. The goal is a three-layered architecture where edge nodes can be used to do the first steps of processing to significantly reduce transfer sizes and cloud dependency. The paper includes an evaluation of HLF v0.6 on Google Cloud Platform (GCP) to measure set(invoke) and get(query) operations on two different levels of hardware. The results show that 255 invoke transactions per second increase to 291 invoke transactions per second for twice as powerful hardware. The results indicate a limitation regarding the load that can be effectively processed by the framework as twice as powerful hardware does not significantly increase the throughput. While this limitation may exist in a data center, it may not be as present using RPi devices or commodity desktop hardware.

Selimi et al. [33] have tested an active HLF deployment in a production wireless mesh network. They make use of chaincode to automatically account for resource consumption in a community mesh network where a large number of participating routers share network resources and can be economically compensated based on usage. They set up an experimental setup where RPi nodes was used to run the blockchain. Tests were run both in a lab environment and in an actual production setting where they measured transaction latency, CPU and memory utilization. Results show that endorsing nodes become a bottleneck at about 100 transaction/min. This could be used as grounds for comparison to our system on RPi. We need to keep in mind that we run different chaincode and as we know this work was done before April 2018, nodes must comprise of the 32-bit compatible HLF v.1.0 or earlier.

3

Hyperledger Fabric

Hyperledger Fabric [5] is part of the Hyperledger collection of blockchain frameworks that are hosted by the Linux Foundation. Hyperledger was originally developed by IBM and is in many ways targeted towards business applications. Due to its focus on enterprise it can be seen as a blockchain for everything except cryptocurrency, but with many of the same features such as immutability, ordering of operations and prevention of double spending. HLF is built using a modular approach which means that consensus, endorsement and storage protocols can be easily swapped. Because HLF is open-source and has been embraced by multiple industries [34], there is a high level of flexibility, available source material and support from the community.

3.1 Blockchain

There are many descriptions of the term blockchain, but generally it can be described as a data structure in which data is always appended to the end of a list. Data is recorded in blocks and each block can hold a number of records. Each block, however, has a cryptographically secured link to its predecessor by storing its hash as part of itself. By this logic, every new block act as additional proof of every previous block as changing data in any previous block would require recalculation of all succeeding blocks. This allows the blockchain to protect the integrity of the data stored in it.

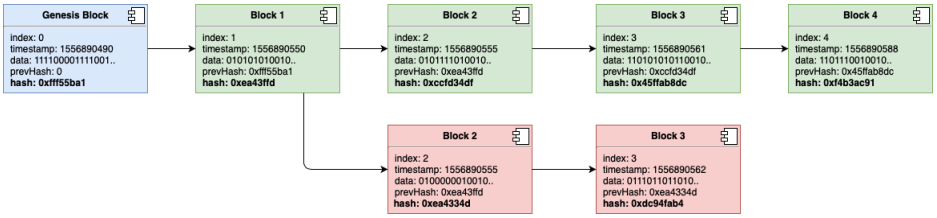


Figure 3.1: Sequence of blocks and fork illustrated.

When you start a blockchain the only thing you need to decide on is a common starting point, this is commonly referred to as the genesis block. In many implementations of blockchain, especially in public/permissionless systems, you can end up having multiple blocks linking to a common predecessor. This typically occurs if two blocks are created around the same time or during a network partitioning and is called a *fork*. The typical way to resolve this is to have all nodes choose the longest sub-chain visible to them and then simply wait until multiple blocks has been appended to your block before declaring it reliable. An example of how tedious this can be is Bitcoin [16] which suggests reliability at six blocks depth and only submits a block every 10 minutes, resulting in a full hour of wait time for a transaction to be considered *valid*. Figure 3.1 illustrates a fork happening, and while there exists only one Block 4, for some systems like Bitcoin, multiple blocks would still need to be appended before deeming the data in Block 1 trustworthy.

3.1.1 Consensus Protocols

The core component of any blockchain is the consensus protocol used to reach agreement between all participating nodes. Most commonly for Public blockchains such as Bitcoin [16] and Ethereum [12] are the use of Proof-of-Work (POW) based consensus where a collection of *miners* are financially incentivized to compute cryptographic puzzles in a race to propose new blocks, resulting in increasingly large computational barriers for recomputing blocks in the ledger. The mining industry consumes a lot of unnecessary energy doing these trivial computations, and at the time of writing (11.05.19) the estimated annual electricity consumption of only Bitcoin [16] is more than the entire country of Colombia [35]. Because of this many blockchain frameworks are actively trying to move away from POW-based consensus. An example of this is Ethereum [12] which is pursuing a switch to Proof-of-Stake (POS)-based consensus, meaning that nodes are selected for validation based on their value in the system, typically based on the amount of currency a node holds. This allows the system to remain secure as long as a majority of the currency in the system is in honest hands, as opposed to POW always requiring an honest

majority of computing power. Algorand [36] is a new approach proposing a new and improved version of Byzantine Agreement using a technique called *Cryptographic Sortition* to randomly choose users based on POS and have them compete in a voting committee for consensus with proofs from *Verifiable Random Functions* [37]. This seems promising but has to our knowledge not yet been used in a publicly available blockchain framework.

Permissioned blockchains are a new branch of systems in which nodes need to be authenticated, resulting in higher trust and a more constant set of participants. This allows for more effective and deterministic consensus protocols to be utilized, resulting in less computation overhead and in most cases no forks. Fault tolerant consensus in distributed systems can therefore utilize battle tested protocols such as Paxos [38], Raft [39] or PBFT [40]. HLF uses orderer nodes to collect transactions, and endorsers from different organizations are required to verify a transaction before it is considered valid. While prototypes often use a single orderer, production networks use multiple orderers for fault tolerance which currently supports either Kafka [41] or Raft [39] based consensus as of HLF v1.4.

3.1.2 Smart Contracts

Smart Contracts are a concept used within blockchain technology that refer to executable logic that is programmed into the core components of a blockchain network. This allows the automatic execution of certain operations and logic based on the parameters of a transaction. In Hyperledger Fabric the term *Chaincode* is used to describe a collection of *Smart Contracts*, and from this point we will refer to smart contract functionality as chaincode. Within HLF, all operations performed on the ledger both for invoking and querying is the work of chaincode. Additionally, we can write and enable our own custom chaincode operations to do specific provenance related operations or more efficient queries. This enables a lot of opportunities for chaincode developers to abstract functionality into the core of the network, and a provides a certain level of functionality guarantee as all nodes are required to run the exact same operations.

3.2 Architecture

HLF consists of three roles: client, peer and orderer. *Clients* are responsible for issuing transactions to the peers, collecting proposal responses in the event of multiple endorsing peers and sending blocks for ordering. On the other hand, *peers* are responsible for endorsement of transactions by running the

chaincode and interacting with the ledger. Chaincode containers in HLF act as an individual process but are generally considered as part of the peer component for architectural purposes. *Orderer* nodes are responsible for verifying the validity of responses and reaching consensus on how transactions are grouped together to form a new block. These blocks are then sent out to peers which in turn update their local ledgers. When this occurs, the peer emits an update event to the client, at which point a transaction is considered committed. Figure 3.2 shows the transaction flow as described in the HLF documentation [42].

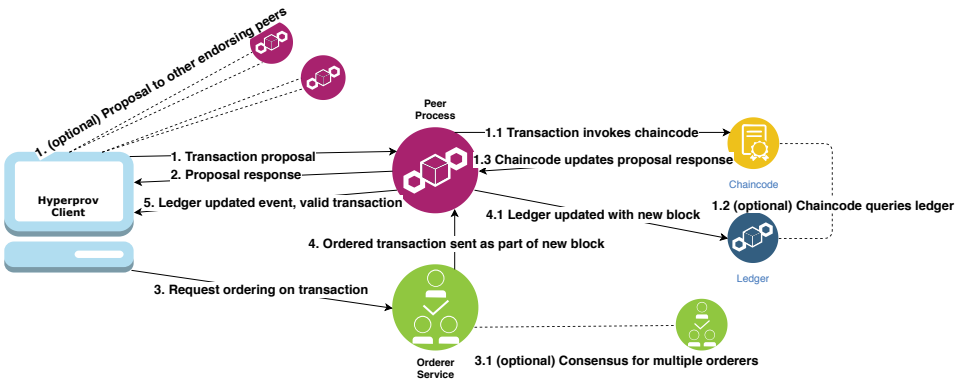


Figure 3.2: Transaction flow between Client, Peer and Orderers in Hyperledger Fabric.

3.2.1 Endorsement Policies

As Hyperledger Fabric is business centered with focus on cross organization cooperation, the endorsement policy similarly is designed around those principles. Endorsement in Hyperledger can be configured on a chaincode level when initializing and set to require peers from multiple organizations to endorse a transaction before an orderer will accept it. This assumes trust within an organization, e.g. that all peers within an organization act according to their interests. This makes HLF ideal for systems that want cross organization sharing of provenance, but also limits the attacker model to not encompass byzantine nodes within these organizations. While certificate revocation lists [43] enable certificates to be revoked if byzantine behavior is detected, any node with an ill-intentioned agenda can propose transactions to HLF as long as they have a valid certificate. However, as the ledger is immutable, data can never be deleted by byzantine nodes. Additionally, incorrect data could be automatically invalidated once a certificate is revoked if transactions are marked with certificate identifiers.

3.3 Docker

Docker [31] is virtualization software that allows its users to spin up virtualization environments called containers seamlessly on multiple devices. Docker containers are more lightweight than virtual machines because they share the host OS kernel, but because of this, images also need to be built for specific architectures. Containers are isolated and includes all dependencies and software in what is called a Docker image. HLF runs its core components such as peer and orderers in Docker environments, similarly chaincode is run in a separate Docker container along with peers. Research shows [6] that running chaincode in Docker is beneficial to many other smart code implementations relying on Ethereum Virtual Machine (EVM) [12]. *Docker Compose* is a tool used for running multiple containers using a single configuration file. These files are called compose-files and are specified using the YAML language. This can be useful for prototyping and developing systems reliant on multiple interacting docker containers. Another tool, *Docker Swarm* enables users to configure docker instances on multiple devices from a single compose file.

3.4 Node Client Libraries

The client component is as shown in Figure 3.2 a central component of HLF. Unlike the peer and orderer components, the client is not a docker image you can just enable and access. Instead *the client* refers to any application that is used to access the HLF network. This is enabled by a collection of Software Development Kit (SDK)s that include the protocols used to communicate with peer and orderer nodes. The two currently supported SDKs are for Node.js and Java, whereas the former seem to be the most maintained in terms of features and tested examples [44]. Fabric also includes a collection of unofficial or no longer maintained SDKs for Python, Golang and a REST server, which may be supported at a later point in time [44].

4

Architecture and Design

In this chapter, we explain the architecture and design decisions that went into the process of creating Hyperprov.

Blockchain enables a new range of decentralized databases that by nature allows us to trace the history of transactions appended to a shared ledger. Blockchain may not be useful for every application and many times regular centralized databases or distributed storage solutions may be more efficient due to problems with scaling and storage. However as coined by Gideon Greenspan [45] there are certain factors which may indicate that blockchain could be useful for you. If your system needs either a shared database, have multiple writers, non-trusting participants, don't want to rely on a trusted intermediary or you want automated interactions between multiple different transactions in the database, blockchain technology may be used to make your system more resilient to tampering and increase auditability. As a distributed provenance tracking system all of these apply directly or indirectly based on the application built on top of Hyperprov.

In the recent years with the expanding amount of collected data [1], we not only have to focus on security, but also on data provenance and quality. *Data provenance* refers to the metadata stored along with the collected data, this is often in regard to the source of data collected and what is collected. A helpful description of data provenance is that it helps identify who, when, what, where and how the data was derived. The granularity of provenance can vary in a wide range between different projects and while very specialized solutions

exist to track provenance data in fields like physics [20], biology [21] and earth sciences [23], we believe in a more generalized approach that can individually be adapted to target more specific fields.

Intertwined with provenance you can often find the field of *data quality*, this often relates to answering questions like if we have acceptable accuracy and precision in our measured data or if our data is specific enough. The measured metadata for data quality is often related to accuracy or tracking other factors that may have an effect on the data besides the obvious. These factors could be anything like additional sensor data such as what the pressure was during the measurements to what firmware was being run on the sensor devices. Other fields of data quality would be tracking missing values and metadata fields, or the consistency of multiple measurements across different datasets. To strengthen data quality assurance, users could choose to store additional metadata related to quality and operations could also be put in place for tracking the amount of missing data. Data quality tracking and management however is outside the scope for this project, mostly due to the wide and often use-case specific span of quality dimensions.

A system for data provenance needs to focus on provenance, albeit to have a system that users can rely on we can not compromise in terms of *data security* either. The term *data security* strongly correlates with the fields of the CIA-triad of confidentiality, integrity and availability. *Confidentiality* is vital to limit sensitive data from reaching undesired people while making sure that people who should reach it has proper access. This is often handled in permissioned blockchains by having a shared set of participants as registered members. An example of this is the organizational level CA's in HLF that supply unique identifiable certificates that is required to access the ledger. Further *integrity* is by nature already handled in blockchains once the data reaches the ledger due to its immutability. The question of integrity then becomes ensuring that data has not been tampered with during creation e.g. at sensor level or changed during transit. To combat this, we could have some sort of unique hardware fingerprints such as done here by using Physical Unclonable Functions (PUF) [46] and already common secure end-to-end communication mechanisms such as Transport Layer Security (TLS). Lastly, *availability* is important to make sure that data is accessible when it is needed by handling faulty components or unexpectedly high activity. While distributed systems and blockchains typically handle faulty nodes and high activity well, there arises other problems such as how to handle network partitions. This may for some systems require core design choices to be made in favor as done by Vegvisir [18]. Hyperprov inherits its partition tolerance from HLF which in turn uses Kafka [41] and requires a majority of orderers with communication between them, limiting availability in the event of network partitions.

4.1 Provenance Metadata

From other provenance systems we see a trend that multiple systems [20, 24, 23] trace data lineage, meaning all items used as part of creating an item is tracked, or file versions before and after an operation is linked [22]. Another trend is to track how the files changed [8, 22, 21], e.g. what operations was applied during creation or modification. Also, the user involved and responsible for the operation is often tracked [20, 8, 21, 7]. These three features form the Open Provenance Model [13] which list them as Artifact, Process and Agent respectively. We wish to retain functionality like this while still having the option for field-specific provenance data as seen in many of the reviewed provenance systems [20, 22, 23]. To do this we store the checksum of data to ensure validity, client-specified location pointers to ensure customization in terms of storage provider, the unique certificate-ID pertaining to the user who stored the data (provided by the Client Identity Chaincode Library [47]) and a list of all other data items used to form the lineage of this item. Additionally, to encompass the need for field-specific provenance data about the process, we include a custom field in which any data structure can be encoded and stored to be able to enable large record collection queries with filing based on these extra fields. A unique ID for every transaction is stored to distinguish different versions of the same object as well as a timestamp of when the transaction occurred. We choose to split the data location into two variables for increased customization on storage provider options in regard to supporting multiple different off-chain storage services simultaneously.

4.2 High-level Architecture

We categorize the system into two necessary components and one supplementary component. The first two are the HLF-framework running in Docker containers and the client library for interacting with these components. The final and optional component is the off-chain storage which can be skipped if a system to handle storage is already in place. The client library is responsible for initiating operations and communicating with the other components as can be seen in Figure 3.2. On invoke operations the client will put the data in storage first and then send information to the blockchain framework. On query operations the ledger will be queried first to check the data location and then subsequently retrieved from storage. The goal of this high-level architecture is to enable seamless storage of provenance metadata and checksums in a tamperproof blockchain ledger while accessing and storing data in a *pluggable* storage service.

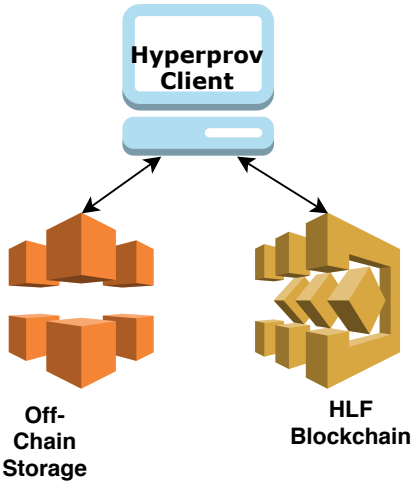


Figure 4.1: The three core components in the Hyperprov System

4.3 Hyperledger Nodes

HLF processes are configured to run on a range of nodes trough Docker. Every node is part of maintaining the ledger by running a peer process that can receive transaction proposals. Transaction proposals start at the Hyperprov Client Library via functions from the various HLF-SDK's and is signed with an eCert issued by the Certificate Authority (CA). HLF provides their own CA Docker image but can also be configured to be any X.509 capable CA. The CA only needs to be accessed when registering new certificates. Peer processes are the most fundamental element in a HLF blockchain network as they host the ledgers and chaincode. To have the most access points, endorsers and copies of the ledger we suggest running as many peers as possible, but one for each participating organization is technically enough. The ordering service is responsible for the ordering of blocks and relies on a deterministic consensus algorithm to validate blocks and their order as proposed by peers. The single orderer approach is currently deployed for our prototype, but for a production setting multiple orderers should be enabled using HLF's built-in support for Raft [39] or Kafka [41] for fault tolerance.

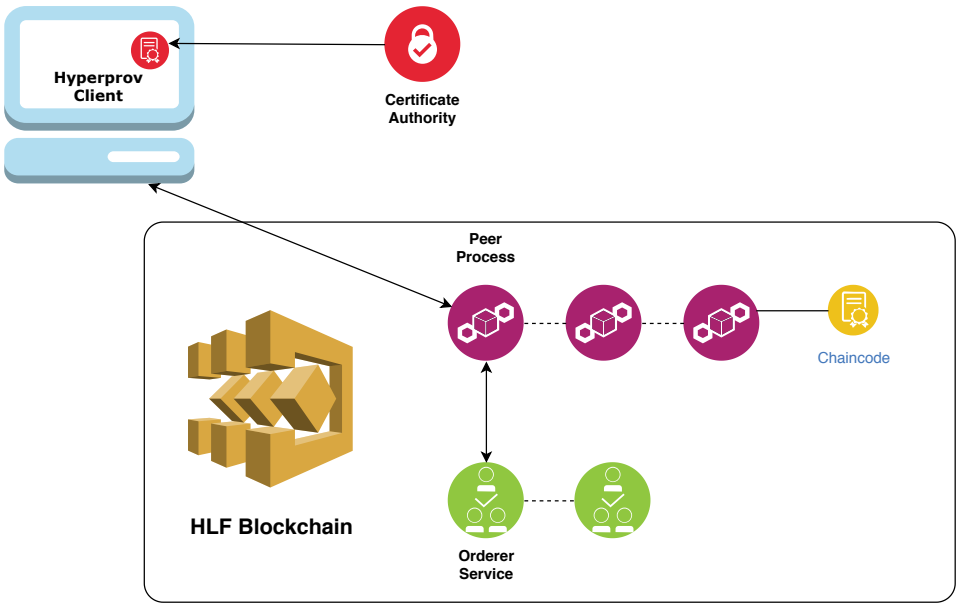


Figure 4.2: Hyperledger components in Hyperprov

4.3.1 Chaincode Operations

The peer nodes are responsible for hosting the chaincode, which is the logic that append or query data stored in the ledger. The chaincode consists of a few core operations that form the base of the Hyperprov functionality. These operations are implemented as typical functions and are mirrored and available across all peer nodes. The main functionality for the chaincode in Hyperprov is to store and retrieve data from the ledger. By the current design, the core data that is stored is the checksum of every data object, the data location, a certificate ID referring to who stored the data, a list of other objects that were used to create an object and lastly a custom field for any additional metadata such as a JSON struct as described in section 4.1. For the data to be stored in the HLF ledger, chaincode functions first need to be invoked with the data as parameters. Because Hyperprov only stores a single type of provenance metadata record, we can get by with a single set-function and instead choose not to include parameters that are not available or applicable as the space overhead is limited to the key-length of that empty field in the ultimately stored JSON-entry for that record. The chaincode can set certificate ID and timestamp automatically, so the remaining parameters needed are checksum, file-location, custom description field and list of data lineage.

As for retrieving data from the ledger, because the type of data we want is specific to the core data parameters stored in the ledger, we can design specific

functions in the chaincode to retrieve data. We want to be able to query both data items based on the key they are stored on, but also specify iterations of the same data. Additionally we want to be able to query for collections of data where the three query types we will initially support are the history of iterations on a single data item, a range query between a start and end key and also a full list of specific data items ID's which form the lineage of an item. These operations could be done easier at the client level with the ability for more custom queries, but for large searches sending an individual request for each data item would be notably slower than using built in functions or recursive queries from within the chaincode container. This results in the chaincode setting the basis for the operations supported by the client, but by no means limiting from other solutions such as having an additional database of keys for off-chain lookup.

4.4 Off-Chain Storage

Distributed ledgers implemented with blockchain have a limitation on how much data should be stored in them, both in terms of the shared ledger growing undesirably large but for HLF also in terms of the performance degradation of storing large data in the ledger [11]. To prevent this, we choose to store only provenance metadata in the blockchain ledger which for most applications is only a small fraction of the total data size. This allows data to be stored in other non-blockchain based services with the trade-off being having to compute and store checksums so the integrity of data can be verified against the immutably stored blockchain records. The choice of storage, however, does not directly affect the ledger or any chaincode functions. The only thing stored is a location field and a pointer to the individual data item, which can be interpreted however desired by the Hyperprov Client application. This allows us to quickly add other storage solutions if needed and could also help in terms of supporting multiple storage solutions simultaneously on the same ledger. As previously mentioned, the off-chain storage is a supplementary component which means it could entirely be skipped if Hyperprov is to be used as part of a system that already handles storage. Then only checksums and provenance metadata would be sent via the Hyperprov Client library without addressing any file-store operators. Because of this and the fact that you can switch between multiple storage solutions, we say that Off-chain storage is a *pluggable* component of Hyperprov.

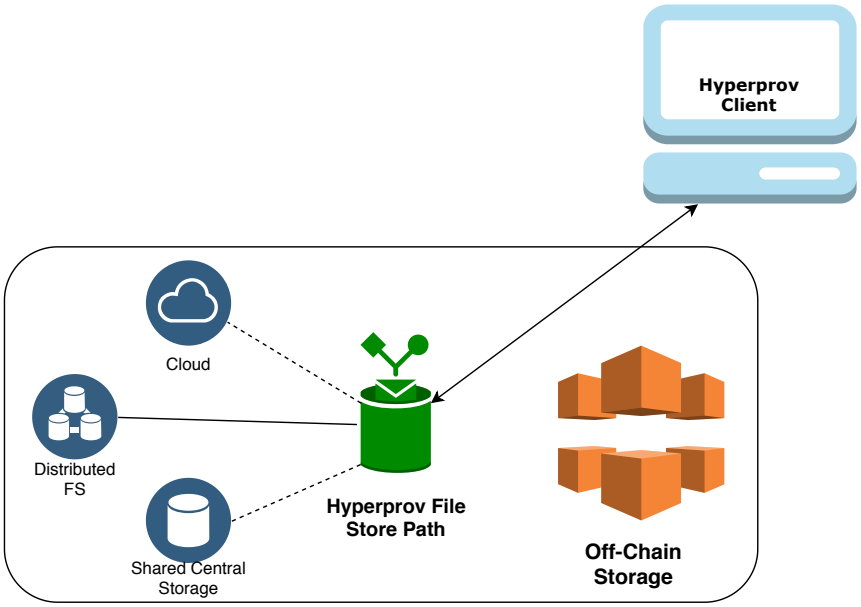


Figure 4.3: Off chain storage possibilities

4.5 Client Placement

The Hyperprov Client Library is based on the HLF SDK [48] as a way to simplify the process of interacting with blockchain technology for provenance tasks. The client library is intended as middleware for any other application that needs to store provenance information, e.g. a client application. The client application can be run either at a separate node by connecting to a peer node in the HLF network or on the peer node itself. This means that you could have a network of nodes that all have its own client application while simultaneously running the blockchain services in the background, completely separate peers and client or any other combination of the two. An important distinction to be aware of when choosing the client placement is the data transferred between client, storage and the HLF blockchain. The data stored and transferred to the ledger generally is only a fraction of the data that is stored in off-chain storage. To reduce the bandwidth the client placement should be prioritized as closely to the storage service as possible, e.g. on the same machine, LAN, or on a device with good network connectivity.

4.6 Availability and Consistency with Network Partitions

Brewer's Theorem also known as the CAP Theorem [49] is used to identify three system properties for distributed/decentralized systems, namely consistency, availability and partition tolerance. *Consistency* refers to strong or sequential consistency so that all nodes must agree on the same sequence of operations. *Availability* refers to that any node should be able to respond within a reasonable amount of time and *partition tolerance* means that you should be able to handle communication errors and recover when whole parts of your network is unreachable.

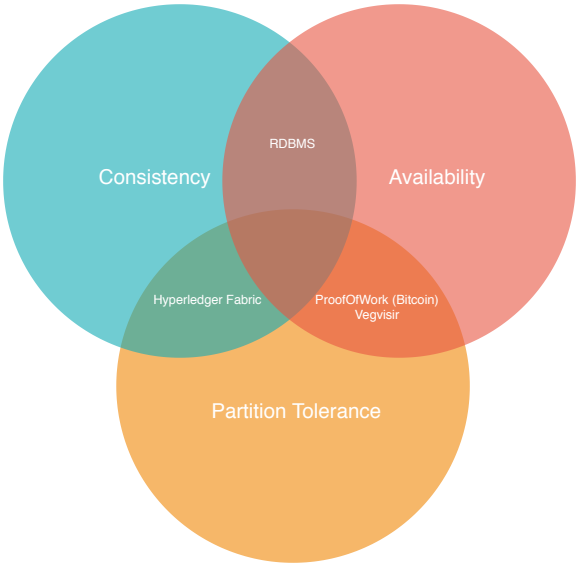


Figure 4.4: Hyperledger Fabric in the CAP-theorem venn-diagram

Because distributed systems are inherently partition tolerant, this means that in the event of a network partition the system will need to choose between strong consistency or availability. Proof-of-work based solutions such as Bitcoin [16] choose availability here, which is why forks can occur that are after some time resolved, resulting in eventual consistency. A framework with strong focus on availability during network partitions is Vegvisir [18] which uses DAGs to track only the partial ordering of events, and reconciliate by gradually comparing the outermost blocks similarly resulting in eventual consistency. HLF and inherently Hyperprov do not sacrifice consistency due to the ordering service relying on deterministic consensus algorithms [5] but can struggle with availability in the event that not enough endorsing peers are reachable or not enough orderers are available for Raft/Kafka-based [39, 41] ordering to complete.

4.7 System Specification

To end this section, we outline the functionality we want to have implemented based on our system design. We want to provide a system using HLF to give assurance of data by storing provenance data in a tamperproof append-only blockchain ledger. The system should record operations made by tracking editors, timestamps, checksums, data locations and lineage. We also want a custom field for the user to add specific data about the process. To access the data, we want to have a client library.

4.7.1 Core Functionality

Following is a list of core functionality for the provenance tracking part of Hyperprov, excluding framework-related functionality such as being able to deploy a HLF-based blockchain network to Desktop and RPi devices as well as organizing certificates and docker containers:

a) Track the location of a data item and accompanying checksum to verify its integrity.

b) Track when and who stored or edited an item based on the certificate used to invoke the transaction.

c) Track data lineage of new items by storing references to items used to create it.

d) Optional field to store additional application specific provenance metadata such as data about the creation process.

e) Store and query provenance information through a multi-purpose user-friendly client library.

4.7.2 Additional Functionality

Additionally, we list functionality that Hyperprov should support, while not being critical to the definition of the system.

a) Support for referencing individual versions of data items using unique ID's.

b) Optional support for storing data in off-chain storage trough the client library.

c) Additional chaincode-level support for full lineage, single item history or key-range queries.

5

Implementation

This chapter covers the implementation of several components of our system, most notably is the chaincode, client library and system configuration. First, we describe the base logic of our system, the Chaincode, and how it handles storing provenance and accompanying challenges. We then explain how our client library is developed to abstract away the hassle of interacting with HLF and how it handles storing provenance data. Lastly, we describe processes related to deploying a HLF network and setting up devices.

5.1 Chaincode

The chaincode in HLF is the only method for interacting with the blockchain ledger. It is also the lowest level of code responsible for storing and querying for provenance information in Hyperprov. Because of this it becomes the first step in abstracting away logic for storing provenance, this is done by grouping HLF-specific operations into common problems such as storing provenance data, retrieving it and querying data lineage. Since multiple endorsing nodes are responsible for running the chaincode we ideally want the chaincode to be lightweight. To guarantee that the chaincode always returns the same result on all endorsers, we also need the chaincode to abstain for accessing external resources. Because of the need for lightweight and deterministic operations we should move away from designs that lead to calculating checksums and accessing external storage from chaincode. The chaincode supports multiple

operations related to data provenance at this point. The operations are: storing provenance data of an item, retrieving the last provenance information on an item, requesting the checksum of an item, getting an item with its corresponding transaction ID, getting a specific version of an item from transaction ID, recursively getting all other items listed as lineage of a certain item, getting the history of a single item and retrieving a list of items with a key-range query.

5.1.1 Data Pointers and Checksums

We track file objects by storing a pointer to the location where it is stored split into two fields: location and pointer. In the current implementation location refers to the path on disk where this data item is stored, whereas pointer can refer to the unique file-name used to store the data or a position within a file referred to by the location field. By default, the client library uses `location + pointer` as the full path of the relevant file. This provides a link from the entry used for data provenance in the blockchain to the actual data stored in off-chain storage. There is a one-way link from the blockchain to the off-chain storage and provenance is only written after data has been properly stored in the blockchain. This can be referred to as *data-driven data management* as opposed to *metadata-driven data management* where the raw data is only written as a result of provenance metadata being recorded [17]. Keep in mind as there is only a one-way link from provenance metadata to actual raw data, data should never be deleted or moved in the off-chain storage without issuing an update to the provenance ledger. With other distributed storage solutions than SSHFS [50] and XtreemFS [51] it could be applicable to store the unique transaction id from the ledger operations along with the raw data item to allow for a more resilient two-way link between raw data and provenance metadata.

To verify that that data is never modified without updating the provenance log we always store the checksum of the data along with every operation. Calculating this checksum could technically be performed by the chaincode, but that would require that the chaincode container have access to external resources and also require the checksum to be recalculated by every peer responsible for endorsing the transaction. From profiling we found that for anything larger than a few kilobytes of raw data, calculating the checksum rapidly becomes a major part of the computing required for issuing a transaction. To combat this and reduce the network transfer required we instead calculate the checksum once at the client application level. Here the client library supplies both a full function to handle storing raw data, calculating hash and publishing to provenance log, or as an alternative you may supply your own calculated hash if you have a more efficient way to compute it than the JavaScript Crypto

library.

5.1.2 Dependency Linking

By comparing to other provenance solutions and projects in potential need of provenance we identify a need for tracking data lineage [20, 23, 22]. Assume you have an item **A** which is further iterated to create item **B**, and then you have item **F** which has no connection to either. Then you proceed to analyze item **B** and **F** to form a new item result **Z**. Item **Z** should now hold direct dependency links to item **B** and **F**, and also indirectly to item **A** through **B**. This is what our lineage chaincode functionality do if we store data with dependency links to items that was partial in their creation. We store dependencies as the transaction IDs delimited by colons. We use transaction ID instead of keys to be able to accurately pinpoint what provenance metadata was current at that point in time. Transaction ID's can be used to query the provenance for specific versions of an item instead of the currently latest one referred to with keys. Figure 5.1 shows the results of a dependency response for IoT data [52]. Listing dependencies from chaincode also supports a depth-specifier to limit the lineage depth as thousands of levels with dependencies may in some cases be unnecessary.

Figure 5.1: JSON object from getDependencies for IoT sensor data visualized.

5.1.3 Identity

Part of the provenance capabilities provided by the chaincode is tracking who performed an update on the data. From other systems we see a need to track users for verifying the origin of data as well as for querying on specific users to retrieve or invalidate it [20, 21, 8]. Previous iterations of HyperProv attempted to solve this problem by simply retrieving and storing the full eCert-certificate used to invoke in the transaction itself. This would be a security risk if access was not limited to certain roles which then in turn would void the concept of every participant being equal and would open up a whole new set of problems. Instead the Client Identity (CID) chaincode library [47] added in HLF v1.1 can be used to retrieve a unique userID issued by the CA which can be directly linked to certificates. The unique attributes are encoded in the certificate and can not be changed without invalidating the certificate. This allows us to store a unique string set by the CA to track origin in the chaincode. The string can be set to anything descriptive of the node which will use the certificate. If identity is ever questioned, the CA-database can be queried to check additional info specified about certificates linked to a userID.

5.1.4 Historic and Range Queries

The chaincode additionally has support for two more types of queries on provenance data, historic and range queries. The former will return the full history of changes made to the provenance information about a specific item. This could be used to query the history of editors, data locations, corresponding checksums, dependency lists or any other metadata stored in the description field of data items stored in the immutable provenance ledger. Currently this chaincode function returns all information for the complete history of a provenance item, but this could also be specified to return only a certain type of information for a specified depth. An example query of history would be, "who was the editors of this item for the past 50 updates to it?".

The other mentioned chaincode query is range queries. This type of queries is based on the built-in support for range queries which return an iterator over all keys stored in the ledger between two key strings *startKey* and *endKey*. This adds additional query capabilities for the user based on their key naming scheme. An example of this could be to label sensor nodes with something like *<owner>_<location>_<sensortype>* where an example node would be *acme_f_temperature*. Then you could query *startkey=acme_d endkey=acme_m* to get all Acmes sensors between d and m including *acme_f_temperature*. You could also do something like *startkey=acme_f_ endkey=acme_f_~* to get all sensors on location f such as *acme_f_temperature* and *acme_f_humidity*. This could potentially be very useful if used correctly but relies on key names to be

labeled in the order that searches would occur. You can not e.g. search for all humidity sensors on node f in any company because company is specified first in the key structure. This can be solved by introducing composite key queries or even better rich queries with CouchDB [53].

5.1.5 Pagination

If the number of results by the any of the queries for historic or range query surpass the maximum specified when building the HLF images in `core.yaml` variable `queryLimit` the results will only return values up to that point. This maximum is by default set to 10000. To retrieve more than this amount from a single query requires us to use pagination. Luckily Hyperledger Fabric natively supports pagination on all its query operators so if needed a wrapper function to target the paginated functions can easily be added to the chaincode lineup. These supports retrieving specific page-sizes and all queries after the first limit use a bookmark-key as their starting point, meaning that the last retrieved key becomes the bookmark used for the next paginated call.

5.2 Client Library

The HLF framework can appear complex to interact with. Because of this the HLF team has created multiple SDKs to help users interact with the system, wheere the most established of which is the Node SDK [48]. This is the basis for our client where we further tailor the interactions with the blockchain for our chaincode and provenance in general. The client library will be the only software applications built on Hyperprov will directly interact with once nodes have initially been set up and configured, figure 5.2 illustrates the software layers. Core functionality should be storing provenance metadata in the ledger and querying the ledger to retrieve provenance records for items. Additionally, the client library can be responsible for storing files in the off-chain ledger and calculating hashes, but to remain feasible for a larger audience of use cases we support this as optional functionality as well. The client still boasts a wide range of functionality such as registering users with the CA, directly accessing any function implemented in the chaincode, storing raw file objects in the blockchain encoded as Base64, Storing/Retrieving data from off-chain storage while verifying its checksum against provenance metadata and more functionality used for the benchmarks in Evaluation. Table 5.1 show the core functionality while additional functions are described in Appendix A.

5.2.1 Exposed API

The HyperProv Client Library enables the use of the HyperProv system for a wide range of functionality with only a few limited operators listed in Table 5.1. The library splits into two modes of operations whereas the first one uses the `Post` and `Get` methods to directly access the HLF chaincode functionality. The latter uses `StoreData` and `GetData` to also handle everything in regard to storing data in the external file storage and retrieving data to form actual file objects.

Function	Required Input	Output
Init	Certificate, Channel, ChaincodeID, PeerURL , OrdererURL	Success/Failure
Post	Key, Checksum, Path, Dependency List, Custom Provenance Data	ID/Failure
Get	Getfunction, Key/ID/Startkey-Endkey	Query Result
InitFS	StorePath	Success/Failure
StoreData	File, Key, Dependency List, Custom Provenance Data	ID
GetData	Key	File, ID

Table 5.1: Key operations enabled by the Hyperprov Client Library

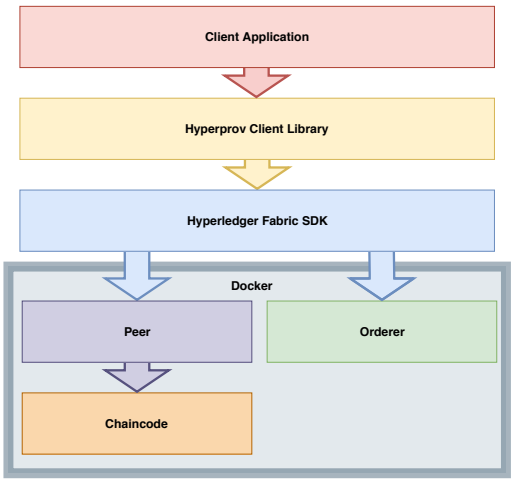


Figure 5.2: The software stack for Hyperprov

5.2.2 NPM Library

The client library is supplied as a Node.js Package Manager (NPM) package which can be imported into any Node.js-project as well as other JavaScript projects. The accessibility of the client library being a NPM package allows potential users to quickly download the client and start incorporating provenance features into their applications.

5.2.3 REST Client

Hyperprov comes with a few examples of how to use the client library, one of which is the REST Client application. The application introduces a restful API using a HTTP listener receive requests for multiple of the provenance operations enabled by the client library and chaincode functionality. This allows data to be sent from any HTTP enabled service to be stored in Hyperprov and then be received later with a simple HTTP-request, preferably authenticated. The restful service only currently supports receiving and sending provenance metadata to the blockchain, but the actual raw data could also potentially be received or sent to this client either in multipart/form-data or Base64 encoded. Ideally, the REST client could also be set up to handle any off-chain storage, completely abstracting Hyperprov data storage to a couple of authenticated HTTP requests.

5.3 Case Studies

To evaluate different use cases and possibly discover useful new features and limitations of our system we decided to look into two different use case scenarios. The first will be tracking frequently updated data in the kilobytes range like you would typically have for simple sensor data measurements. This type of use case we target towards the more lightweight like the RPi. The other use case regards larger data stored in desktop environments which can be stored frequently but typically is less frequent and could also be not as evenly distributed over time.

5.3.1 IoT Sensor Data Client

Because we are among the first to run HLF on RPi devices we believe to have a unique opportunity to provide provenance tracking to systems that may have previously struggled to run in edge environments where low cost and low power devices such as the RPi are typically deployed. To create an application to test with IoT data the first step was to actually get some IoT data. The choice fell on the Global Surface Summary of the Day (GSOD) [52] which contains measurements such as temperature, pressure, visibility, wind and snow depth from about 9000 stations around the world every day with the most complete data being from 1973 to present. The GSOD-data can then be parsed and stored using the HLF client library. Every 30 measurements are batched together for a total about 4KB, every third batch is also "analyzed", which in reality means that the previous version and previous analyzed data are retrieved and concatenated to form a new analyzed object. The analyzed

objects are stored with a dependency linking to both the previous analyzed object as well as the current newest data object. An example of five levels recursive dependencies from the IoT data stored with HyperProv can be seen in figure 5.1.

5.3.2 Machine Learning Models Client

As a rising field in a number of sectors Machine Learning is a relevant technology for any new research. Due to the compute intensiveness and randomness aspect of model training we think there may be a need for provenance data to both verify the validity of trained models as well as for providing reproducibility. Multiple of our provenance features and especially the ability to trace the lineage and history of iterating data could make the case for adding an additional tamperproof storage layer for this type of data production. For further exploration of Hyperprov we created another example application to check for and store gradually changing machine learning models produced by the ImageAI library [54]. This library is for our example being used to train models on the IdenProf dataset [55] containing images of identifiable professionals such as chefs, doctors and police. Our application uses Hyperprov to store the iteration history of training models with dependency links to the training and test sets used for any particular model. We do this by every run checking the current test and training sets, and by keeping a list in persistent storage of what models have already been stored before scanning for new models. Models are stored in off chain storage with their provenance metadata being pushed to blockchain using the Hyperprov client library. We suggest this example application could be invoked after every training epoch, session or periodically from a cron job to push provenance information to the blockchain.

5.4 System Configuration

A substantial part of the time spent implementing Hyperprov consisted of tasks related to system configuration. This in terms of setting and configuring dependencies, docker compose, docker swarm, required certificates, start scripts, chaincode updates, off-chain storage, external network access and creating custom docker images for RPi. As well as multiple issues related to updating from HLF V1.0 to HLF V1.4. The experimental setup on RPi featuring unsupported 64-bit Linux was the root of many problems related to missing libraries, software and kernel issues. For future projects using Hyperledger for RPi we would like to see improved 64-bit support for RPi devices or the ability to run HLF on 32-bit ARM devices.

5.4.1 Compose Files

To easily start up and shut down the prototype network used in the Hyperprov experiments we used Docker compose to organize the deployment of multiple containers to our test nodes accessible through a shared Docker swarm network configuration. This allows relatively simple management of our network to allow for quickly clearing, changing roles and updating chain-code during development. The compose file is responsible for configuring all required variables, deploying containers and calling a script for initializing and testing the blockchain. For a non-experimental setup the single Docker compose file and swarm network would be replaced by role specific compose files on each new node joining the network.

5.4.2 Certificate Authority

While the certificates used to set up and run HLF can be generated using the supplied *cryptogen* and *configtxgen* binaries, access from the client library requires eCerts that are best generated from the *fabric_ca_client*-library. To do this we need an appropriate certificate authority to issue the certificates. While it is stated that any X.509 capable CA can be used, we decided to go with Hyperledger's own CA docker image started with a customized compose file to enable persistent storage of the certificate database. All certificates issued require a unique identifier which will be stored in the ledger on transactions issued using that certificate. The Hyperprov client library can be used to register new user certificates with the fabric-CA using methods `registerAdmin` and `registerUser`.

5.4.3 Shared Storage

For shared storage we currently use SSHFS [50]. The choice was largely influenced by limited availability on aarch64 for our RPi system. SSHFS is a filesystem in userspace client that interacts with a remote filesystem trough SSH with SFTP as the underlying protocol. Performance evaluation [56] show that SSHFS compare well to the established NFS [57]. SSHFS is easy to set up and allows for quick swapping between storage servers for baseline measurements and debugging. To provide fault-tolerance in a deployed network we would swap SSHFS for something more resilient like the distributed filesystems XtreemFS [51] or OpenAFS [58] run directly on the nodes, or a cloud service provider like Amazon EFS [59].

5.5 Building Docker Images

As HLF does not officially support the RPi there were no official or public images for the ARM architecture available when beginning this project. There are some existing images for RPi running v1.0 in 32-bit mode or v1.2 in 64-bit mode, but none complete or with the improvements of v1.3 including key-level endorsement policies and the first long term support release v1.4. To do this we had to build docker images from source code on the 64-bit ARMv8 architecture which is only supported on a small number of unofficial operating systems such as Debian Buster 64-bit for RPi. This can be a tedious process due to missing libraries or bugs on ARM, slow build times on RPi and its constrained memory. To our knowledge we are the first to build HLF v1.4 docker images for ARM with 10 of 11 docker images published to Docker Hub [9], with the remaining image (Javaenv) being published later thanks to outside help [60]. See *Appendix B* for further details on building HLF for ARM.

6

Evaluation

During implementation the focus has been on providing the required functionality while keeping chaincode lightweight and without access to external resources. A substantial part of the functionality is therefore placed at the client level while the peer remains an integral component at the center of communication (see Figure 3.2).

In this chapter, we evaluate HyperProv with focus on performance in terms of throughput and response times. We also evaluate the resource consumption in terms of CPU, memory, network and for our RPi devices also energy efficiency. We think performance and resource consumption overhead is essential in the evaluation of a system based on the premise of being an addition to already existing systems. We use custom client-side benchmarks for performance, Linux-based resource measuring tools for resource consumption and a physical power-meter to check energy usage on RPi. We run benchmarks for varying load levels and transactions sizes and compare running Hyperprov both with and without attached storage.

6.1 Methodology

6.1.1 Experimental Setup

The experiments are run on two different setups of the same network. The first setup consists of desktop-grade nodes and the other consists of RPi nodes. The desktop setup has 4 machines: 2 Intel Xeon™E5-1603 v3 CPU @ 2.80 GHz 4C4T, 1 Intel Core™i7-4700MQ CPU @ 2.40GHz 4C8T and 1 Intel Core™i3-2310M CPU @ 2.10GHz 2C4T. These four nodes form the desktop network each running peer docker containers, whereas one *Xeon* machine run the orderer. All nodes run Ubuntu 16.04, are on the same LAN and are equipped with SSD storage with approximately 500 MB/s read and write speeds. These nodes run the official docker images provided by HLF.

The next setup consists of 4 ARM-based Raspberry Pi 3B+ 1.4GHz 4C4T Cortex-A53 devices interconnected on the same network switch. The nodes run the yet unofficial RPi 3 Debian Buster 64-bit OS as the newer HLF versions require 64-bit support. Due to the lack of other supported docker images we have compiled our own images for the ARM64 architecture used to run Hyperprov on these nodes [9].

Measurements were performed with client on the 2.8GHz 4C4T Xeon E5 CPU for desktop and compared against the 600MHz (Originally 1.4GHz see 6.1.2) 4C4T Cortex-A53 CPU on the RPi. For measurements involving off-chain storage we run SSHFS [50] on the remaining node after client, orderer or accessed peer. Figure 6.1 show the layout used for measurements.

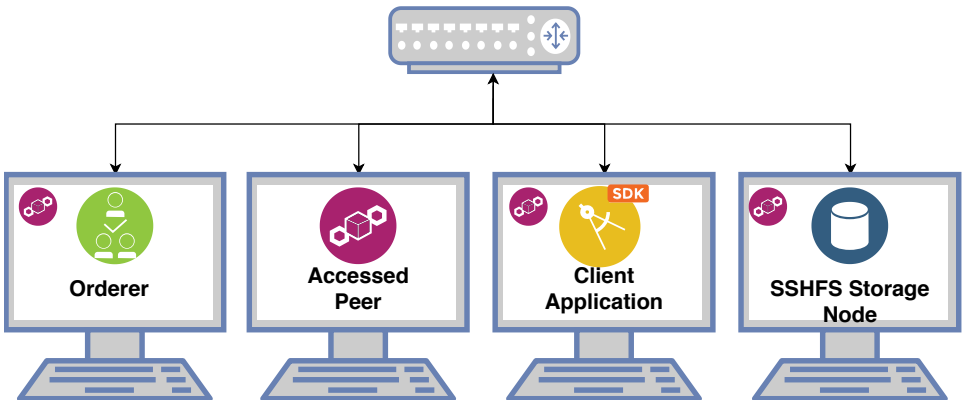


Figure 6.1: Component distribution on experimental setup.

6.1.2 CPU Throttle on 64-bit Raspberry Pi

Due to an error with the 4.9.x - 4.18.0 *aarch64* kernels on RPi, the CPU scaling governor responsible for modifying frequency does not work properly [61]. This causes the device to be stuck in *powersave* mode where the frequency is constantly set to the lowest setting (600MHz). Because we need 64-bit OS support to run HLF we are now limited to reduced frequencies until either HLF adds support for 32-bit systems or the Linux kernel for aarch64 on RPi fixes frequency scaling. This limits the maximum throughput we can measure on RPi devices, but also limits the energy used to what would be the case running devices with the *powersave* setting on the CPU governor.

6.1.3 Throughput Measurements

Measuring throughput was done by a benchmark-application built on top of the Hyperprov client library to push batches of transactions to the network. The application would run timers for every batch and individual timers for each transaction. For varying data sizes or batch sizes the application would measure multiple samples and calculate the average total time, transaction response times, transactions per minute and track the number of failed transactions occurring. The application would then be set to either store data only in the blockchain or both in the blockchain and off-chain storage. These are the results you can see in figures like 6.2 and 6.7. The single transaction latency measurements in tables 6.1 and 6.2 were derived from measurements made by another application that sends transactions to HLF 100 times and records the time of each individual transactions to calculate averages, standard deviation and more.

6.1.4 Measuring Resource Consumption

For statistics on resource consumption we did measurement on a set of load levels that reflect the middle and edge cases of load that could likely be applied. For desktop we measured 20 Transactions of 100MB data stored over a 10-minute span (2 tx/min), 120 times 50KB (12 tx/min) and 15000 times 50KB (1500 tx/min). For RPi we reduced the load to account for the less capable hardware and did 20 times 10MB (2 tx/min), 120 times 50KB (12 tx/min) and 3000 times 50KB (300 tx/min), also over 10 minutes.

Measuring CPU/RAM Usage

The CPU and RAM usage was measured on a per-process level with the psrecord tool [62]. Psrecord allows us to record the specific usage of any process, with maximum utilization on these systems being 400 % percent. We saw no limiting factor on the Orderer or Chaincode processes as we moved towards our maximum load, and therefore display them only briefly in Figure 6.9. Instead we focus on measuring the peer process which is responsible for running the blockchain and handling incoming transactions, as well as our NodeJS client application using the Hyperprov Client Library to issue transactions, calculate checksums and store data when off-chain storage is involved.

Measuring Network Traffic

Network traffic was measured using the *speedometer* network measuring tool for Linux-systems [63]. The tool was used to measure total received and sent network traffic over a time span of about 2 minutes, then for more visible graphs cut down to ~ 40 seconds. During this time all other significant network activity from the measuring machine was disabled and only the HLF processes and SSHFS communication seemed to be using traffic based on manual monitoring with the *iftop* Linux tool.

Measuring Energy Consumption

Energy measurements were only performed on the RPi devices as they are most relevant there. To measure energy on desktop we would use a framework like Heartbeats [64] which relies on the Model Specific Registers of the X86 CPU to measure power usage. Measurements on RPi were measured during the same intensity runs as described in section 6.1.4. The measurements were manually derived from an ODROID Power Meter V3 located in between the device and power source over 10 minutes. While software measurements are generally better and more accurate, without access to model specific registers on our ARM64 RPi devices, we think manual measurements are sufficient for a general idea of relative energy drain.

6.2 Throughput

We start by measuring throughput for different levels of load intensity. To do this we change the number of transactions submitted together before waiting for all to complete. From Figure 6.2 we can see as we increase the batch size the throughput grows exponentially with large steps and diminishing returns evening out around 2000 transactions per batch. The benefit from higher batch size comes from the ability for more messages to be ordered in the same block if received within the same timeout (about two seconds). This allows higher throughput to be reached if a large number of blocks are filled up quickly. The limitation of about 3000 transactions per minute or approximately 50 transactions per second indicates that our desktop hardware can handle about five blocks of 10 transactions per second on average. Also, as we increase the batch size, the individual transaction response times increase as seen in Figure 6.2. This is likely due to the increasing congestion of items waiting to be handled by peers and approved by the orderer. As the numbers shown here are the averaged times of every transaction on a batch, the first transactions compute quickly however as the queue grows individual response times increase accordingly. The throughput however does not fully peak until between 8000 and 10000 transactions per batch as shown in figure 6.3, where we measured a peak performance of 3276 tx/min. We believe, however, that batch sizes of this scale are unreasonable for most applications and not worth it due to the diminishing returns and increase in response times.

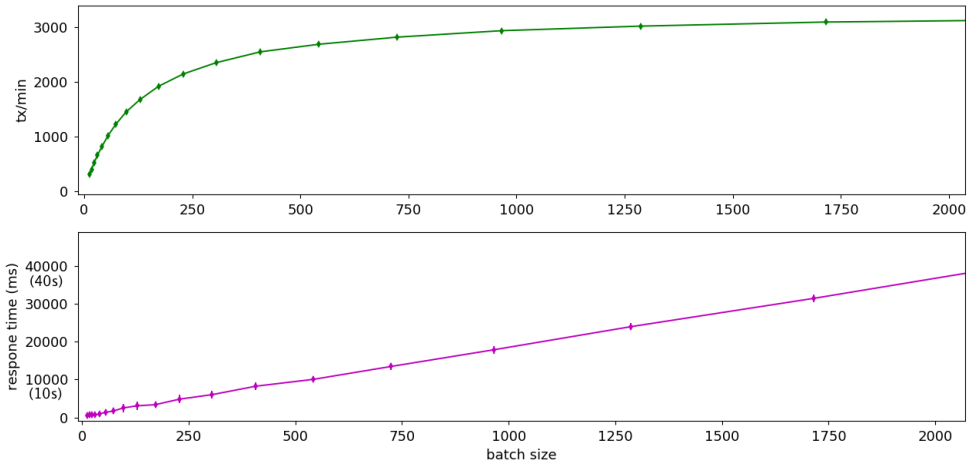


Figure 6.2: Throughput and average response times for increasing load levels. (Desktop) (only provenance data)

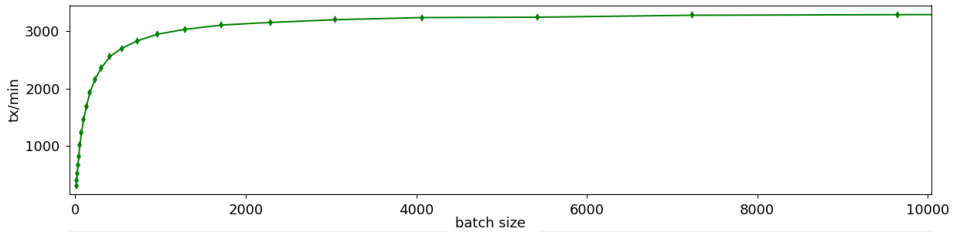


Figure 6.3: Figure 6.2 extended. Throughput for load levels up to 10 000 transactions at a time. Peaking at *3276 tx/min*.

We did similar measurements on RPi and figure 6.4 show the same trends as figure 6.2 while reaching its peak at a lower batch size, generally lower throughput and substantially higher response times. Due to the earlier peak efficiency and especially the high response times we believe that batch sizes of anything above 200 may be less than optional for the limited hardware of a single endorsing RPi device.

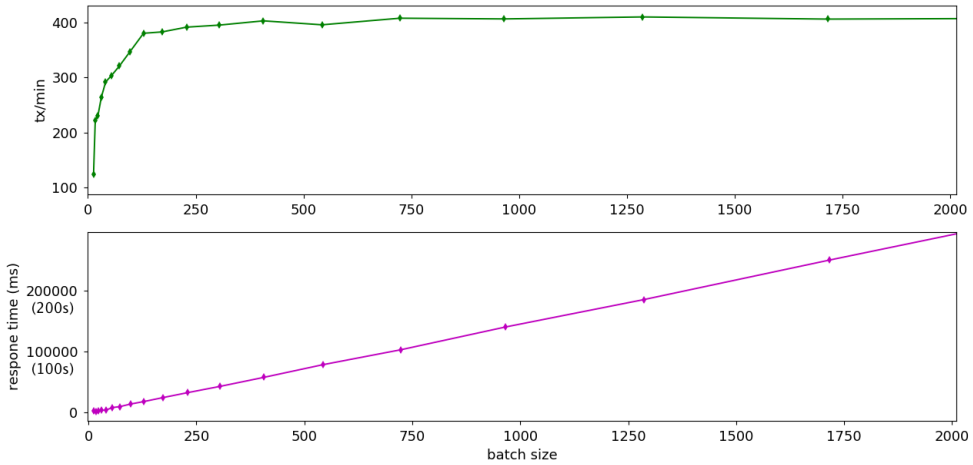


Figure 6.4: Throughput and average response times for increasing load levels. (RPi) (only provenance data)

Figure 6.5 shows how increasing the size of data items impact both throughput and response times when off-chain storage is involved. When only provenance data is stored performance is not affected by increasing data size as the data is never actually stored in the blockchain. On the other hand, when off-chain storage is involved we must now account for both the time to store data to SSHFS, but also the time to calculate the checksum of the data object. The results show gradually degrading performance as size increases, however about 500 transactions per minute of a 6 MB object is not bad with average transaction

response times of about two seconds. It is especially good considering this is the work of a single client device while a network of multiple clients and endorsers should each be capable of similar throughput. The limitation here may seem like a hardware limitation, but only from the client application and peer process limiting in conjunction with one another. What this means is that while there is a limitation on storing data, computing checksums and storing it in the ledger, none of these operations seem to use the full extent of the host device resources. Instead it is limited by the time required to do these operations in sequence for each data object, resulting in lower throughput and longer response times.

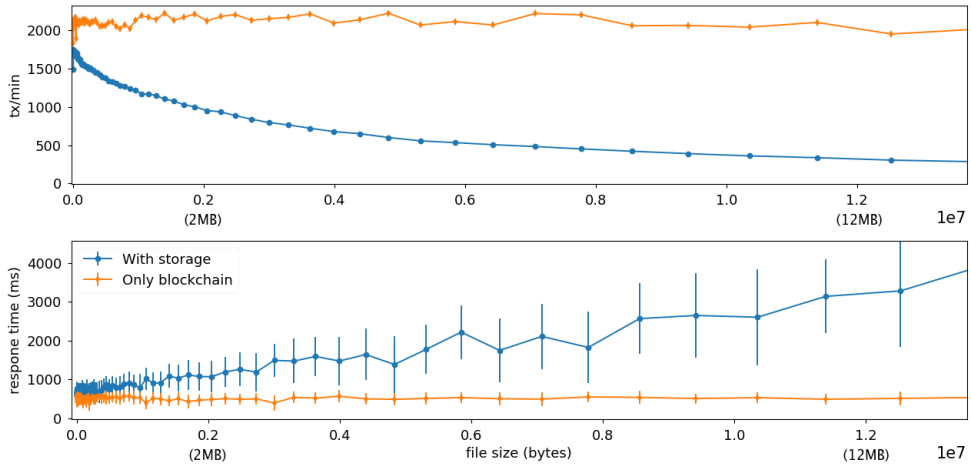


Figure 6.5: Throughput and response times for varying data sizes. (Desktop)

To show this limitation we did a series of measurements with one, two and three clients simultaneously sending transactions to their own local peers to be issued on the same endorsing peer and orderer. To have uniformity in hardware we did this on RPi devices, with orderer and endorsing peer both on the fourth device. Results in figure 6.6 show how multiple clients can be used to achieve overall higher throughput, at the cost of per-device throughput. This is because our network of four devices have only a single orderer and endorsing peer node where all transactions need to pass. With more devices we could do an in-depth analysis of how Hyperprov and HLF v1.4 performs for a large network of devices. For now, we will make do with the results from evaluations such as Blockbench [6] where they fail to scale beyond 16 nodes due to the old consensus protocol implementation over-saturating servers, although this should be fixed in newer releases. Similarly, Selimi et al. [33] indicate endorsers as a bottleneck, which should be reduced by deploying multiple endorsers with a suitable endorsement policy.

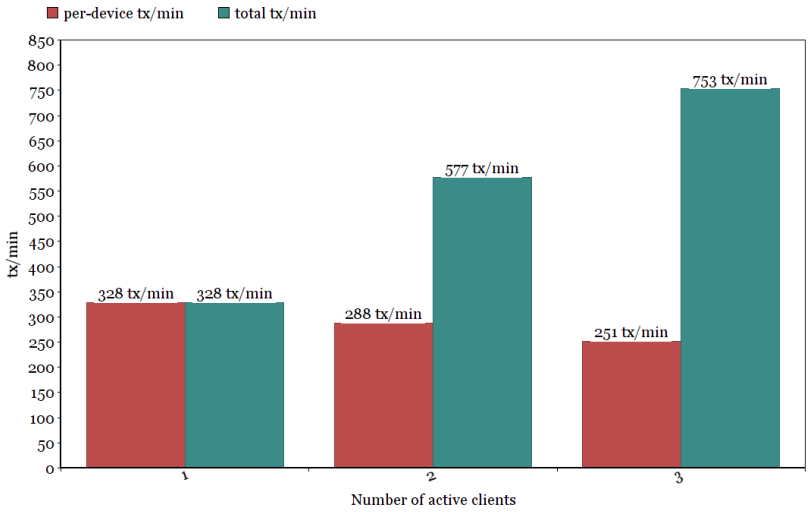


Figure 6.6: Throughput for 1-3 concurrent client devices. (RPi)

The same trend as figure 6.5 can be seen on the RPi setup, however with about 1/5 of the performance as shown in Figure 6.7. The trend is similar but due to the RPi being more limited by its hardware the results are lower and more varying. Slightly below 400 transactions per minute the RPi has about 1/5 of the performance compared to desktop, which correlates with an expected four- or five- times performance difference from 600MHz to 2.8GHz for these measurements.

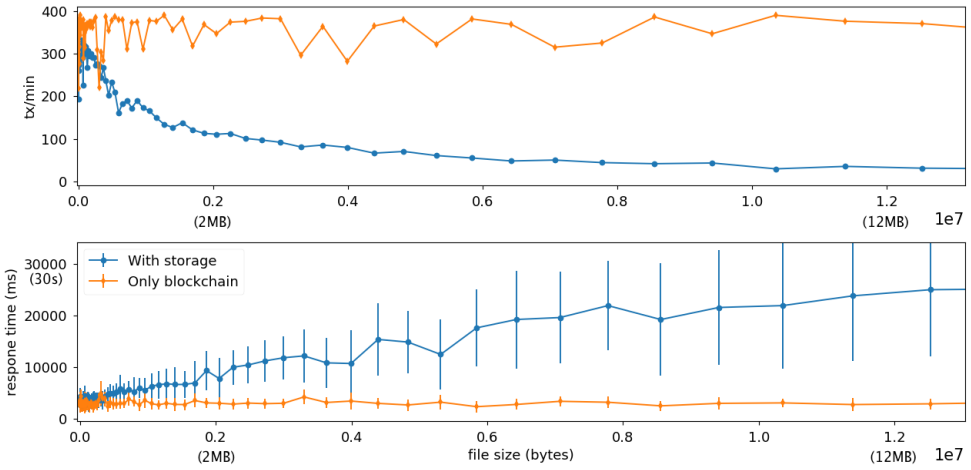


Figure 6.7: Throughput and response times for varying data sizes. (RPi)

6.2.1 Latency

We measure the responsiveness of the system by sending requests to the peer-process through the Hyperprov client library every 10 seconds. The results shown in Tables 6.1 and 6.2 are the average of 100 operations. We are able to submit a transaction with confirmation in *2146 ms* on Desktop and *2492 ms* on RPi. Retrieving operations for all get operators seems to hover around 100ms both on Desktop and RPi. The data lineage functionality *getdependencies* implemented in our chaincode have been tested with 10 recursive dependencies returned in 102ms or less, with latency within a margin of 5 % for all other query operations.

	set	get	-withid	-fromid	-keyhistory	-byrange	-dependencies(10)
Desktop	2146 ms	100 ms	101 ms	101 ms	102 ms	101 ms	102 ms
RPi	2492 ms	102 ms	101 ms	101 ms	102 ms	102 ms	103 ms

Table 6.1: Latency of operations to blockchain

To investigate why the set-operation performed worse on RPi we extended the measurements to show how the times spread out compared to desktop. We see that the standard deviation on RPi is the cause. Both resource measurements and throughput measurements on RPi also fluctuate more than its desktop counterparts. In table 6.2 we here see slightly higher lows, but also highs in the range of 4556 ms, explaining the higher averages on RPi.

	AVERAGE	STDEV	MEDIAN	MIN	MAX
Desktop	2146	29	2143	2095	2400
RPi	2492	297	2412	2357	4556

Table 6.2: Statistics of SET operation latency data

To explore the performance degradation of our built in queries, we tested the lineage tracking functionality on IoT data [52] with up to 1000 recursively linked dependencies similar to what is displayed in figure 5.1. Figure 6.8 show that every increase of about 60-70 depth levels, the time to respond increases by almost exactly 100 ms. We could not find any documentation or configuration setting explaining this phenomenon, but we think it might be due to limitations imposed by the HLF chaincode framework because of either memory limitations of a single query or being limited to a certain number of recursive calls before having to issue another transaction. Which in turn would cause the time to increase by almost the exact time of a transaction for each step of 60-70 depth levels.

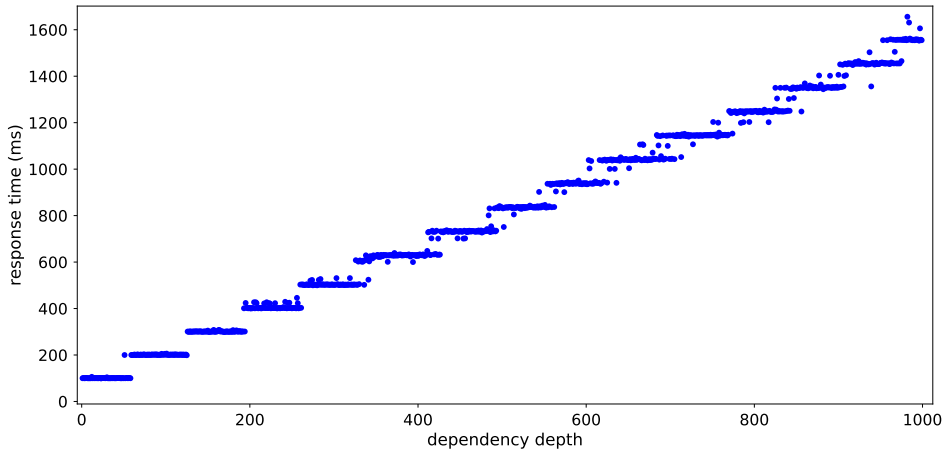
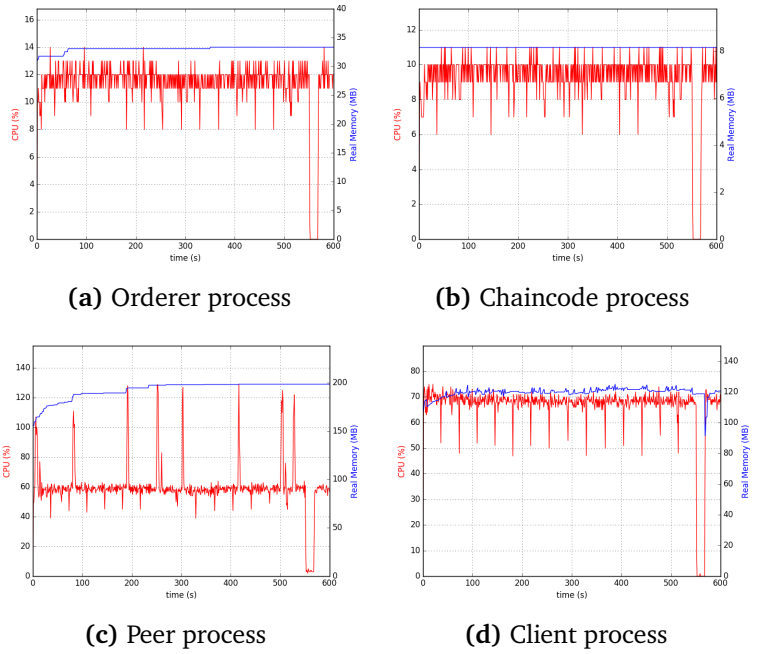


Figure 6.8: Latency for an increasing number of dependent records retrieved with the *getdependencies* chaincode. (Desktop)

6.3 Resource Consumption

As a means of measuring the overhead and understanding the effect of hardware, we performed a series of resource consumption measurements, starting with the CPU and memory usage of the involved processes, then network and energy consumption. Our system consists of four measurable components, namely the peers, orderers, chaincode and client. In Figure 6.9 you can see the CPU and Memory usage of a 3200 T/min load issuing transactions of 1KB in batches of 2000. These initial measurements show how peer and client components resource utilization depend heavily on the data size and throughput, while both the orderer and chaincode claims a relatively moderate 12 and 10% CPU usage or below in all our testing and less than 50 MBs of memory combined, see Figure 6.9 (a) and (b). Because of this we will focus mainly on the peer and clients resource utilization for the remainder of this section. The client process steadily hovers around 70 % CPU and 120 MB of RAM. This because the client has multiple responsibilities for each transaction such as connecting to the peer, invoking chaincode/proposing transactions, handling the proposal response and request that the transaction is ordered. Additionally, if the client uses off-chain storage it would need to calculate the checksums stored in the ledger and store the actual data in off-chain storage. The figures display a maximum load scenario whereas for real world usage the resource utilization may be lower, see Figures 6.10 and 6.11.



(a) Orderer process

(b) Chaincode process

(c) Peer process

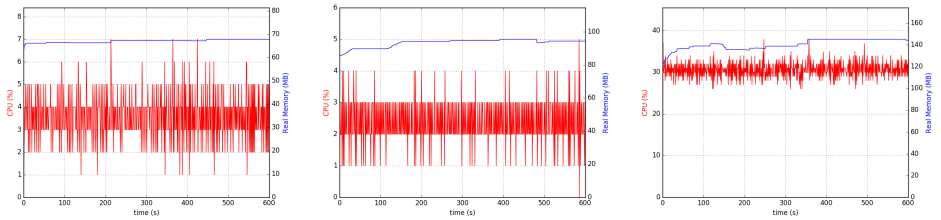
(d) Client process

Figure 6.9: CPU/Memory at max load (~ 3200 tx/min). (Desktop)

Similarly to the client, the peer process resides at around 60% CPU usage. As the core component in any HLF network it plays a substantial role in transaction proposal as it needs to coordinate responses to and from all other components while also maintaining the ledger data. In Figure 6.9 (c) you can see about 7-8 spikes in CPU-usage over the 10-minute span, this occurred only in some of our measurements on full load. We think this is due to the peer sometimes falling behind on communication with client, chaincode and orderers and therefore consuming more at certain intervals in an effort to catch up. In Figure 6.9 (c) about 550 seconds in, we pause transactions for a bit, and we can see how the peer is the only process with any discernible consumption at idle state with a 0-3% CPU utilization.

To evaluate how the resource consumption is at different levels of intensity we measure both peer and client for three levels of throughput. These levels were selected to showcase (a) large data size with low throughput, (b) small data size with low throughput and (c) small data size with high throughput. The measurements show plots of CPU and memory usage over 10 minutes with one second granularity. If we look at how the peer process behaves across different load intensities in Figure 6.10, we see similar behavior across the lower intensities in (a) and (b) but increasing to high throughput (c) of 1500 tx/min, we see a substantial change to 30 % CPU usage. Similarly, as we increase the

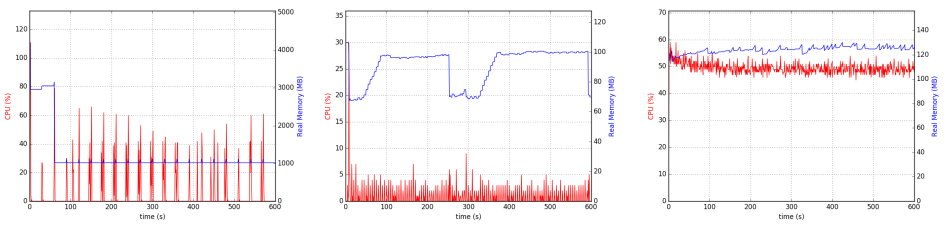
load intensity, the memory consumption increases as more transactions must be handled by the peer simultaneously.



(a) 2 tx/min 100MB **(b)** 12 tx/min 50KB **(c)** 1500 tx/min 50KB

Figure 6.10: CPU and memory for peer process (Desktop)

To evaluate performance on the client application we enabled off-chain storage to see if data size affected the CPU and memory consumption. For a small number of large transactions in Figure 6.11 (a) we can see how the CPU usage spikes about 20 times in between 40-60 %, this compared to the mostly sub 5 % CPU of a larger number of transactions in (b) strengthens the claim that the client library's calculation of checksums may become a limit for large file sizes. Large files also consume increased memory, as storing and calculating the checksum of these large files seem to consume about 1GB of memory in 6.11(a). This, however, is client specific and depends on the application making use of the Hyperprov client library. For the application used to measure this we store data in between transactions to avoid latency related to randomly generating a 100MB file for each transaction. The throughput on the other hand has a clear effect on the client as seen from (c) where high intensity with small files of 50KB has the CPU utilization at around 50 %. These results combined with the full load measurements without off chain storage in Figure 6.9 (d) show that the client can be affected and limited by large file sizes, high intensity or a combination of both.



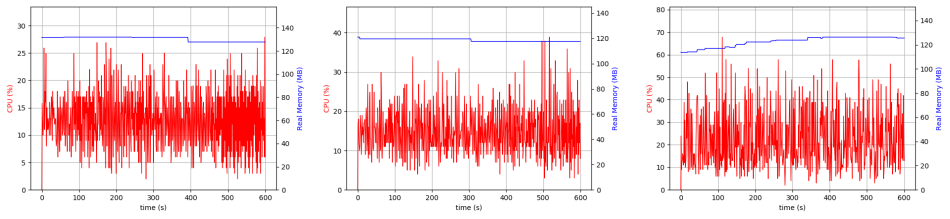
(a) 2 tx/min 100MB **(b)** 12 tx/min 50KB **(c)** 1500 tx/min 50KB

Figure 6.11: CPU and memory for client process (Desktop) with external storage.

6.3.1 Raspberry Pi

To evaluate RPi system relative to its hardware we did similar resource measurements on RPi devices with two modifications. We lowered the size of measurement (a) from 100MB to 10MB and the high throughput of measurement (c) from 1500 tx/min to 300 tx/min. This was done to more realistically fit the hardware limitations of these devices while maintaining measurement (b) of 12 tx/min 50KB for direct comparison.

From Figure 6.12 (a) and (b) we can see that the difference between 2 tx/min and 12 tx/min are more apparent than on the desktop setup, even on peers that should be unaffected by the modified size. Compared to the desktop the same throughput levels result in 5-6 times higher CPU consumption running on RPi, which need to be considered relative to their clock speed difference ratio of 4.6. For measurement 6.12 (c) we can not directly compare but see that 300 tx/min has approximately the same CPU consumption as 1500 tx/min does on desktop, albeit with noticeably more fluctuating consumption, which again is a trend seen across all measurements comparing RPi to desktop.

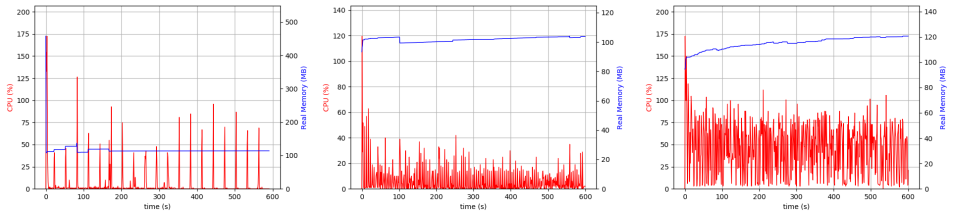


(a) 2 tx/min 10MB **(b)** 12 tx/min 50KB **(c)** 300 tx/min 50KB

Figure 6.12: CPU and memory for peer process (RPi)

The client application on RPi similarly was measured with off-chain storage enabled. Figure 6.13 show similar trends to Figure 6.11 across all measurements. The difference between running client on desktops and on RPi seems to be the overall higher CPU % which is relative to clock speed difference.

Where comparable, the memory consumption seems to be equal on clients between RPi and Desktop. For peers the memory consumption seems to be overall higher on RPi, which may be related to the slower hardware requiring the peers to hold on to pending transactions longer. With memory consumption between 200MB and 300MB for all components combined, memory may become a limitation for systems utilizing RPi devices for additional computation besides provenance, with only 1GB available.



(a) 2 tx/min 10MB **(b)** 12 tx/min 50KB **(c)** 300 tx/min 50KB

Figure 6.13: CPU and memory for client process (RPi) with external storage.

6.3.2 Energy

High energy consumption is a concerning problem with PoW based blockchains like Bitcoin [35]. While permissioned blockchain don't suffer similarly thanks to the lack of mining, if they are to be viable for IoT and edge devices energy consumption becomes important again. Since we run our system on RPi devices, we want to measure the overhead of running Hyperprov for different load intensities. Figure 6.14 shows the average energy consumption of RPi devices running both peer and client processes over a 10-minute span. The first measurement show that the RPi consumes about 2.71 Watts in idle state with no transactions. We did multiple similar measurements with HLF/docker disabled and ended up with approximately 2.71W on average. Thus, running Hyperledger Fabric without any transactions barely consumes power as reflected by the idle CPU usage in Figure 6.10 (c). Once we start increasing throughput to 20X and 120X transactions, power moves up to around 2.79W, which is about a 2.95 % increase in power consumption from idle. For the highest load level of 3000X transactions there is about a 10.7 % increase in power consumption from idle. The final bar shows the maximum power consumption at 400 % CPU load is 3.64 W based on our device, while the Raspberry Pi 3 B+ is rated at a maximum consumption of 5.1 W. This is due to the issue with CPU frequency scaling for 64-bit OS on RPi mentioned in section 6.1.2. All measurements in Figure 6.14 reflect the power consumption of our RPi devices running in *powersave* mode.

Just for fun, our highest throughput of 3000 transactions over 10 minutes consumes 0.503 watt-hours, if we compare this to the average network consumption of a Bitcoin transaction [16] at 468 kilowatt-hours [35], we are more efficient by *11 orders of magnitude*. However, this does not account for the global scale of Bitcoin. Assuming that HLF would scale as large and handle byzantine adversaries with a constantly changing set of participants, if every Bitcoin miner (~1 Million) instead had a RPi running Hyperledger Fabric it would still improve energy consumption by six orders of magnitude on a per-transaction

basis.

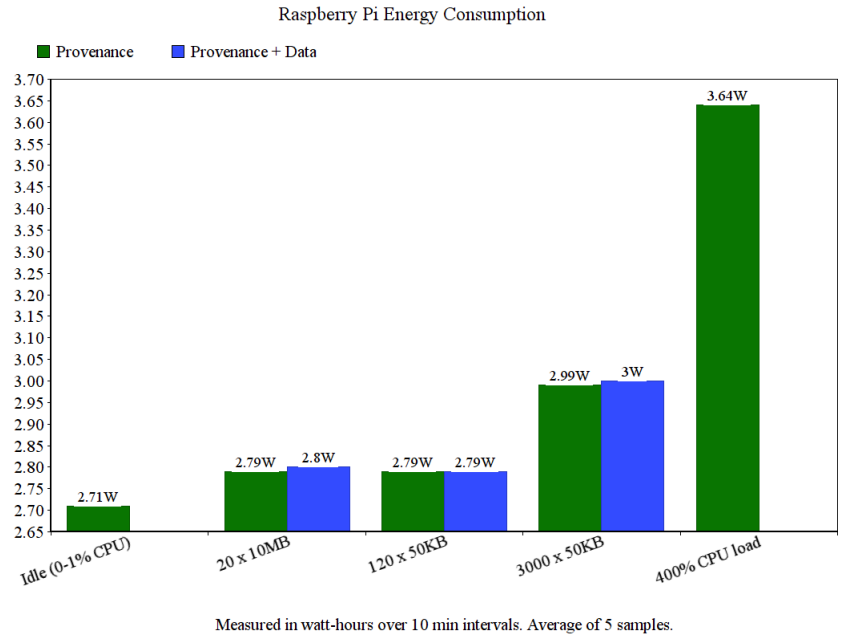


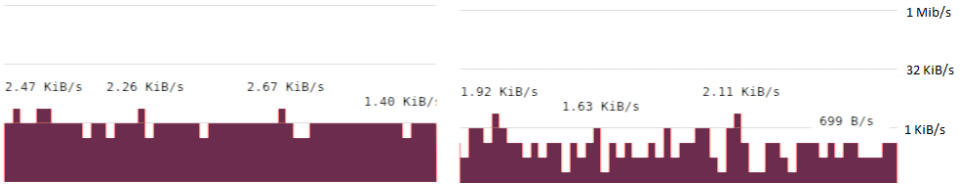
Figure 6.14: Energy consumption on RPi. 10-minute intervals

6.3.3 Network

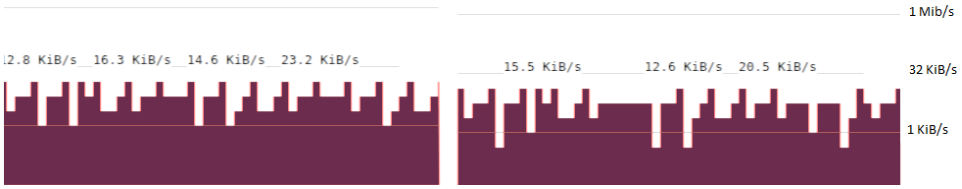
To measure the overhead in terms of network resources consumed we measure on a setup with a peer and client running locally, while publishing transactions to an external peer, orderer and storage service. By running a peer locally, we are able to account for ledger updates received while measuring client to peer traffic on transferred traffic. This way we are able to account for all factors of potential data transfer.

Our baseline measurements were run without Docker or any processes running except those included by the OS. Figure 6.15 (a) shows that without any service enabled we can write off between 1-3 KiB/s to *other* traffic. When we start HLF we can after the initial setup process see a discernible increase in network traffic from running just the peer process. Figure 6.15 (b) shows that this has increased the network traffic approximately by a factor of 10. This is due to the peers communicating trough gossip protocols to verify the consistency of the ledger. Gossip messages from all other nodes compared to only a few being passed on by the local node explains why the received traffic are slightly higher than transferred data, especially on a network of only four nodes.

To see how the network utilization is affected by throughput we measured two



(a) Baseline



(b) Peer Idle

Figure 6.15: Network consumption without transactions.

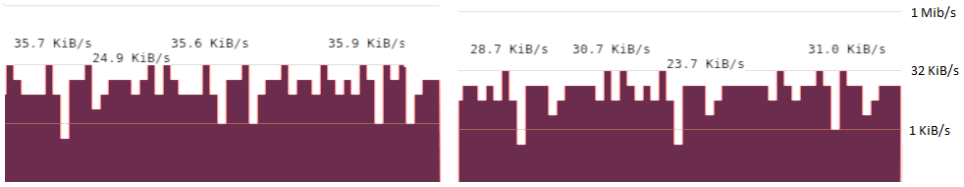
very different levels of throughput with and without external storage enabled. The results displayed in figure 6.16 (a) show that both read and write traffic for sending a transaction every five seconds range mostly within the higher end of the 1-32 KiB/s spectrum. Moving over to figure (b) we see that once we enable data transfer to external storage, we see a noticeable increase on write traffic ranging as high as 80 KiB/s at times. Less expected we see a slight increase also in received traffic, which we assume is from SSHFS issuing confirmation that new files have been stored in the shared folder.

As traffic increase to 25 tx/sec sec in figures 6.16 (c) and (d) we see that granularity decreases in proportion to the increased scale due to constant high throughput. Also more present in (c) than in (a) is the difference between traffic received and sent. This strengthens the claim that gossiping results in received traffic and that it is affected by throughput.

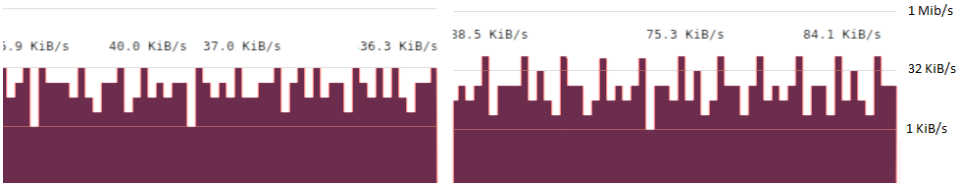
From subfigure 6.16 (d) we can see that transferred data is a product of provenance data and data as

$$610KiB/s + (50KB * 25tx/sec = 1.25MB/s = 1.19MiB/s) =\sim 1.80MiB/s$$

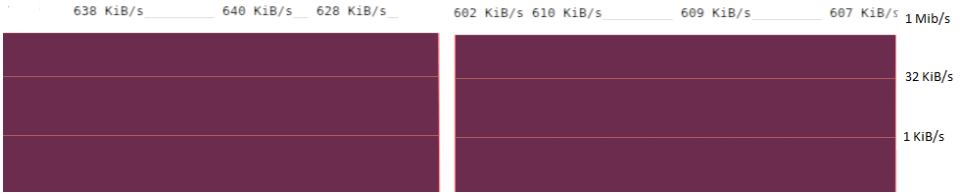
whereas the final 50 to $100KiB/s$ is due to additional traffic related to storing 25 files every second to SSHFS.



(a) 0.2 tx/sec. Only provenance.



(b) 0.2 tx/sec - 50kb. Provenance + SSHFS.



(c) 25 tx/sec. Only provenance.

(d) 25 tx/sec - 50kb. Provenance + SSHFS.

Figure 6.16: Network consumption with and without SSHFS.

6.4 Edge Computing

Many new emerging technologies such as smart city, smart grid, smart healthcare or smart transportation require interconnected sensors and devices that collect and share data. For applications such as these and more, aspects of provenance tracking such as validity, identity and lineage of data can be highly desirable. Devices referred to by the term Internet of Things are often small sensor-based devices with processing power in the couple of MHz range. Based on our current resource measurements we think Hyperprov may be too demanding to run directly on the smallest of sensor devices. However, the terms

IoT Gateway [25] and *Edge Computing* [65] are used to describe a sort of middleware-box between the sensors and their storage or application services. We propose for systems using this *machine-to-gateway* model that running a system like Hyperprov between gateways could provide uniquely tamperproof records with acceptable overhead, especially on relatively energy efficient ARM devices.

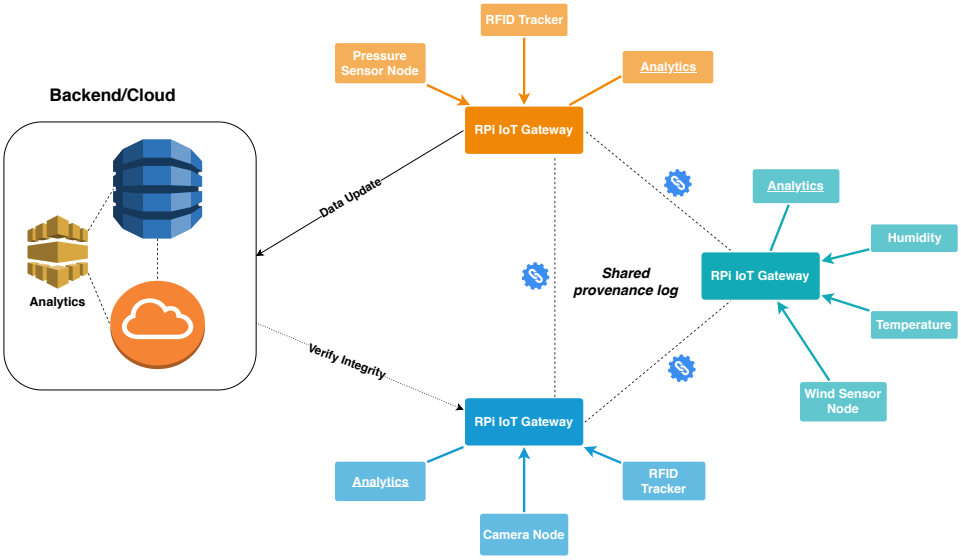
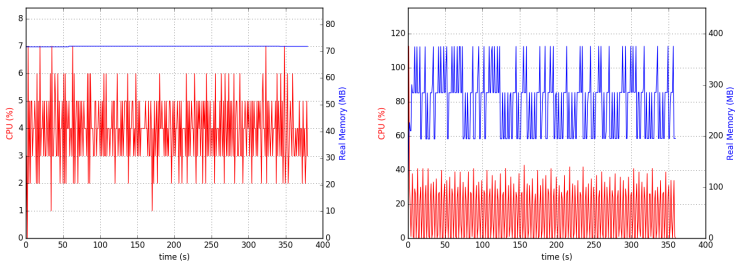


Figure 6.17: IOT gateways illustrated

Offloading work to edge devices can save sensor devices for energy, bandwidth and may even improve latency [65]. Doing pre-processing or compression at the edge can also save a lot of energy. [25] show that IoT data is often highly compressible due to its correlation in time and achieved a 9.75 compression ratio on a year of Fitbit data. Based on our early evaluations of Hyperprov and HLF on RPi devices we believe that the performance is sufficient and the resource consumption low enough for this to be useful. Our main concern for this is regarding network traffic as results (Figure 6.15 and 6.16) show constant traffic in the 10-30 KB/s range during idle operation and traffic that can range into the several hundred KB/s range during load for provenance data only. If additional research show that network traffic remains approximately the same for an increased number of orderers and endorsing peers, idle traffic can most likely be reduced by configuring the gossip protocol. We think that Hyperprov could be considered for provenance tracking for large scale IoT networks as a collection of authenticated edge computing nodes on devices like RPi.

6.5 Model Metadata Tracking

Systems that learn from data are increasingly being deployed and tested for a wide range of applications in today's industry. Lately *Federated Machine Learning* [66] have seen a rise in popularity, proposing continuous computation of models across a distributed set of participants, and even across multiple organizations. These machine learning systems often has a complex and often varying combination of models, training and test datasets. Also, because models often are a one-way recipe, transparency of their generation process may be desired for both researchers and users. As a result of this we see the need for a framework to help develop *Model Metadata Tracking* [67] as data scientists and organizations often use their own ad-hoc style and no standardized framework for tracking results, in turn making it difficult to compare results and reproduce successful experiments. Figure 6.18 show the resource consumption of the peer process and client application storing 100 models of ~94 MB. The models are generated by the ImageAI [54] library and has dependency links to lists of files used as training and test datasets. The process of storing the models takes approximately 6 minutes to complete, because everything is done in a sequential manner. Storing a model involves reading it from disk to compute checksum, storing it over network via SSHFS and then pushing a Hyperprov transaction including checksum and location. For a 94MB model on our desktop experimental setup this is completed in 3.6 seconds. From figure 6.18 we see that the peer process is barely affected from its idle 3-4 % CPU consumption, while the client application is under constant load in the 0-40 % CPU range.



(a) Peer process **(b)** Hyperprov client application

Figure 6.18: Resource usage for storing 100 x 94MB ML models.

To further explore the CPU consumption, we did some profiling on the client application. Figure 6.19 show that the client process largely consists of computing checksums and transferring data to storage, while storing data in Hyperprov and garbage collection occupy the last 6 and 7 % respectively. Our client application for model tracking stores models as part of their subsequent models,

requiring sequential operation, but for applications with unassociated models figure 6.18 (b) show that there exists headroom to parallelize checksum calculation.

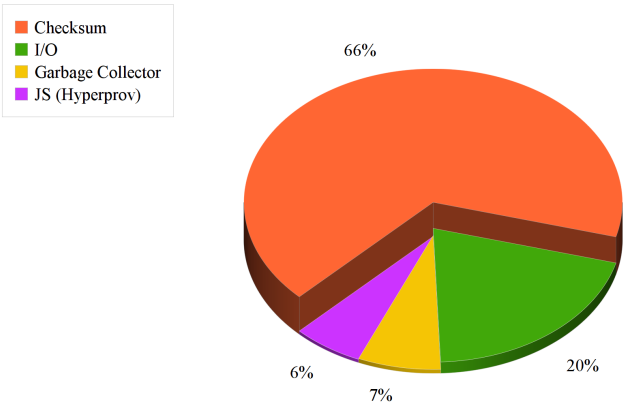
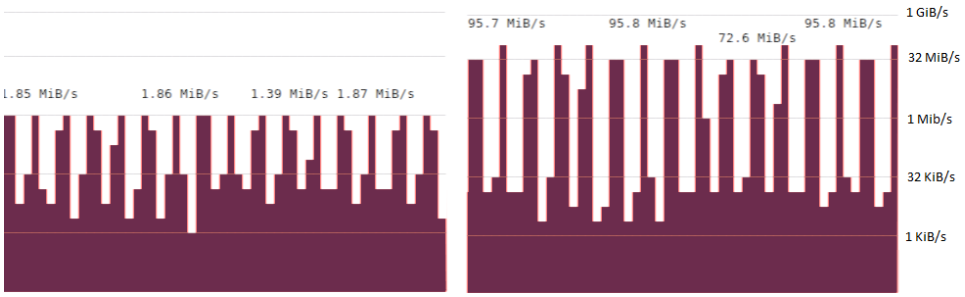


Figure 6.19: V8 Profile for storing 100x 94MB models

In terms of network traffic, the overhead of storing provenance here should be similar to that of Figure 6.16 (a). The high network traffic shown in figure 6.20 is the product of storing 94MB data objects every 3.6 seconds. For most systems this level of traffic is a given if you want to store large files to a shared network file-store. If network traffic is a limitation, we propose that all preceding models could have only its provenance stored and remain stored locally. Then only the final and best performing model could be uploaded to shared storage with appropriate lineage pointers pointing to the preceding models.



(a) 0.27 tx/sec - 94mb. Provenance + SSHFS.

Figure 6.20: Network for storing 94MB models

From our evaluation using Hyperprov to store models generated by ImageAI [54] we found that our system is capable of storing provenance information about the generated models such as checksum, location, who generated it and

optional parameters such as `batch_size` or `num_obects`. We track the current training data and test set, as well as the history of models. We can now trace the lineage of successful experiments and use it to reproduce or verify existing models. From resource measurements and profiling we see that the process of storing 100 94MB models using Hyperprov is limited by the file size in terms of calculating checksum and network transfer. Figures 6.18, 6.19, 6.20 and 6.16 (a) all indicate low overhead for running Hyperprov without off chain storage. Because the system is limited by network transfer and checksums, we instead propose the use of a system like Fast Integrity Verification (FIVER) [68] either along with or integrated into the Hyperprov Client Library. Their experiments show that by concurrently executing transfer and checksum operations and by enabling file I/O share between them they are able to reduce verification overhead from 60 % down to only 10 %.

7

Discussion and Concluding Remarks

In this final chapter, we list and discuss our design choices and our findings. We also outline possibilities for future work and conclude the thesis.

7.1 Discussion

We have implemented a prototype of Hyperprov, a general purpose-provenance solution using blockchain to enable transparent and tamperproof metadata records. As a measure to appeal to a large audience of potential use cases, design choices were made in favor of customization. One of these choices was to not store any storage specific data in the ledger but have generic pointers that can instead be implemented and handled client-side. This allows for greater variety in terms of storage services. To access the ledger, a client library was implemented in NodeJS which is made available through NPM. This was developed through the use of the HLF Node SDK, but the client could also be ported to other SDKs at a later point in time like Java or Python. The Hyperprov chaincode is used to store file pointers in a generic manner. Checksums are tracked to ensure the validity of data stored. We optionally store lists of dependencies when adding data to be able to track the lineage of combined objects. A custom field is allocated to optionally track additional

meta data to describe the data derivation process or for sorting. For increased efficiency a set of lightweight retrieval functions are built into the chaincode, this allows us to quickly retrieve collections of records such as the history of a specified item, items within a key range or the lineage of a specified item. This functionality has shown only about 100 ms degradation in latency for every 70 records requested in a single query.

7.1.1 Findings

Because the HLF-framework requires 64-bit we need to run an unofficially supported OS for our RPi setup. This led us to uncover that due to limitations of the Linux kernel on 64-bit ARM devices, the CPU does not scale above the minimum rated threshold of 600MHz to the maximum rated 1.4GHz for our devices, essentially rendering our devices stuck in *powersave* mode. Because of this our two experimental systems run on nodes with 366 % difference in CPU performance at 600 MHz and 2.8 GHz respectively. Experiments shown in figures 6.2 & 6.4 comparatively show an increase in throughput of approximately 650 % indicating a less than optimal scale with increased hardware. Other research [30] using HLF similarly show an only 14.12 % increase in throughput for twice as powerful hardware, but then at a significantly higher throughput 255 to 291 tx/sec against our 6.6 to 50 tx/sec. We would need to run the system on additional and more powerful hardware to verify if our system is prone to the same amount of diminishing returns. However, we can not directly compare either as they used the v0.6 HLF-framework and provide different functionality.

Peer and client processes seem to be the main resource consumers from our measurements with an average 60 & 70 % CPU consumption respectively at full load. The orderer process and chaincode comparatively use an average of 12 & 10 % CPU. We see from additional measurements at varying load that resource consumption scales mostly relative to throughput, but files in the megabytes-range are affected additionally on the client side due to checksum computation. For realistic loads of transactions of 50KB every five seconds we see as low as 3 & 4 % CPU on desktops and 15 & 18 % on RPi. As an argument to the feasibility of Hyperprov for use in IoT, this level of load only amounts to about 0.08W or about 2.95 % increased power consumption over our idle measurements.

If we compare our system to Smartprovenance [8] which state no total time for the full procedure of recording a document change, their operations *Initiate Change*(858ms), *Vote*(829ms) and *Record Change*(950ms) stacked already surpass our time of 2146ms both for changes and new documents. This being running on our own network, then without the per-operation gas-tax on

Ethereum [12] smart contracts. ProvChain [7] claim sub-second storage speeds, but also rely on Chainpoint [14] that can require up to 10 seconds for initial proofs and 60-90 minutes [69] to be anchored to Bitcoin [16]. Smartprovenance does not state retrieval times, but ProvChain [7] claim to retrieve 10 records using an average time of 221 ms. Comparatively we retrieve a single record in ~100 ms, while using our built-in chaincode queries we were able to retrieve up to 10 dependent IoT records in ~102 ms and 1000 in 1552 ms. Provchain [7] also claim a peak of 30 transactions per second in their figures, on our desktop setup with 2000+ transaction batches we can achieve about 54 tx/sec (see Figure 6.3). However, on a more comparable load of 500 transaction batches of 64KB we achieve about 36 tx/sec (See Figure 6.5). For energy measurements we have no other provenance system to compare against, but we can refer to the overhead of 2.95% and 10.7% increase in power consumption over idle for load levels we consider realistic in terms of high and low throughput.

7.2 Future Work

For future work, we would like to explore integrating Vegvisir [18] into HyperProv to better cope with lack of network connectivity for IOT edge devices, and handle network partitions.

We want to test HLF's built-in support for CouchDB [53] to enable more advanced queries to allow provenance for fields which require retrieval of provenance based on application-specific data.

An automated system for enrolling new devices and revoking certificates could similarly be explored.

Due to the high overhead of checksum calculation and network transfer for large file sizes we would like to also explore integrating a more efficient system like Fiver [68] to reduce this overhead.

We also want to evaluate whether the proposed extensions to HLF [5] by the FastFabric [70] project, can help in improving performance further for HyperProv.

Last but not least, the CPU-scaling issues in the Linux kernel for 64-bit RPi [61] currently require a fix to fully utilize the devices' potential.

7.3 Concluding Remarks

In this thesis we have described the design and implementation of Hyperprov, a general-purpose provenance system built using HLF with a Node.js client library. We have also made a contribution in terms of building HLF Docker images for ARM, making the process easier for new projects to develop systems using HLF on ARM-devices. By building a provenance framework using blockchain we can provide tamperproof and transparent records distributed across a collection of peers while maintaining similar functionality to already established systems [2].

Our evaluation show that Hyperprov can be deployed to track provenance metadata with competitive throughput and response times on commodity desktop hardware, while also maintaining low overhead on ARM-based devices like RPi. We achieve similar or better performance to other comparable provenance solutions anchoring in blockchain technology without relying on POW and currency reliant blockchains [7] [8].

Bibliography

[1] A. Dragland and SINTEF, "Big data - for better or worse," may 2013. [Online]. Available: https://www.sintef.no/en/latest-news/big-data-for-better-or-worse/

[2] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1084805.1084812

[3] M. Herschel, R. Diestelkämper, and H. Ben Lahmar, "A survey on provenance: What for? what form? what from?" *The VLDB Journal*, vol. 26, no. 6, pp. 881–906, Dec. 2017. [Online]. Available: https://doi.org/10.1007/s00778-017-0486-1

[4] M. Herlihy, "Blockchains and the future of distributed computing," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017, pp. 155–155. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087873

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[6] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: ACM, 2017, pp. 1085–1100. [Online]. Available: http://doi.acm.org/10.1145/3035918.3064033

[7] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla,

"Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 468–477.

[8] A. Ramachandran and M. Kantarcioglu, "Smartprovenance: A distributed, blockchain based dataprovenance system," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: ACM, 2018, pp. 35–42. [Online]. Available: http://doi.acm.org/10.1145/3176258.3176333

[9] Hlf arm images dockerhub. Accessed: 08.05.2019. [Online]. Available: https://hub.docker.com/u/ptunstad

[10] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Report of the acm task force on the core of computer science," New York, NY, USA, Tech. Rep., 1988, aCM Order No.: 201880.

[11] P. Tunstad, "Towards energy efficient blockchain-based systems for secure data sharing on edge devices," UIT The Arctic University of Tromso, dec 2018.

[12] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, accessed: 2019-04-01. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[13] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche, "The open provenance model core specification (v1.1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743 – 756, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X10001275

[14] Chainpoint, "Chainpoint: A scalable protocol for anchoring data in the blockchain and generating blockchain receipts," 2019, accessed: 2019-04-01. [Online]. Available: https://www.chainpoint.org

[15] R. C. Merkle, "Protocols for public key cryptosystems," in *1980 IEEE Symposium on Security and Privacy*, April 1980, pp. 122–122.

[16] S. Nakamoto. (2009) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[17] A. Demichev, A. Kryukov, and N. Prikhodko, "The approach to managing

provenance metadata and data access rights in distributed storage using the hyperledger blockchain platform," *CoRR*, vol. abs/1811.12706, 2018. [Online]. Available: http://arxiv.org/abs/1811.12706

[18] K. Karlsson, W. Jiang, S. Wicker, D. Adams, E. Ma, R. van Renesse, and H. Weatherspoon, "Vegvisir: A partition-tolerant blockchain for the internet-of-things," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 1150–1158.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[20] I. Foster, J. Vockler, M. Wilde, and Yong Zhao, "Chimera: a virtual data system for representing, querying, and automating data derivation," in *Proceedings 14th International Conference on Scientific and Statistical Database Management*, July 2002, pp. 37–46.

[21] J. Zhao, C. Goble, R. Stevens, and S. Bechhofer, "Semantically linking and browsing provenance logs for e-science," in *Semantics of a Networked World. Semantics for Grid Databases*, M. Bouzeghoub, C. Goble, V. Kashyap, and S. Spaccapietra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 158–176.

[22] C. Pancerella, J. Hewson, W. Koegler, D. Leahy, M. Lee, L. Rahn, C. Yang, J. D. Myers, B. Didier, R. McCoy, K. Schuchardt, E. Stephan, T. Windus, K. Amin, S. Bittner, C. Lansing, M. Minkoff, S. Nijsure, G. von Laszewski, R. Pinzon, B. Ruscic, A. Wagner, B. Wang, W. Pitz, Y.-L. Ho, D. Montoya, L. Xu, T. C. Allison, W. H. Green, Jr., and M. Frenklach, "Metadata in the collaboratory for multi-scale chemical science," in *Proceedings of the 2003 International Conference on Dublin Core and Metadata Applications: Supporting Communities of Discourse and Practice—metadata Research & Applications*, ser. DCMI '03. Dublin Core Metadata Initiative, 2003, pp. 13:1–13:9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1383296.1383313

[23] J. Frew and R. Bose, "Earth system science workbench: a data management infrastructure for earth science products," in *Proceedings Thirteenth International Conference on Scientific and Statistical Database Management. SSDBM 2001*, July 2001, pp. 180–189.

[24] Y. Cui and J. Widom, "Practical lineage tracing in data warehouses," in

Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073), Feb 2000, pp. 367–378.

[25] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy, "Towards blockchain-based auditable storage and sharing of iot data," in *Proceedings of the 2017 on Cloud Computing Security Workshop*, ser. CCSW '17. New York, NY, USA: ACM, 2017, pp. 45–50. [Online]. Available: http://doi.acm.org/10.1145/3140649.3140656

[26] J. C. Nelson, M. Ali, R. Shea, and M. J. Freedman, "Extending existing blockchains with virtualchain," 2016.

[27] R. Nygaard, L. Jehl, and H. Meling, "Distributed Storage with Strong Data Integrity based on Blockchain Mechanisms," Master's thesis, University of Stavanger, Norway, 2018.

[28] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, The University of Guelph, Guelph, Ontario, Canada, jun 2016.

[29] J. Benet, "IPFS - content addressed, versioned, P2P file system," *CoRR*, vol. abs/1407.3561, 2014. [Online]. Available: http://arxiv.org/abs/1407.3561

[30] A. Stanciu, "Blockchain based distributed control system for edge computing," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, May 2017, pp. 667–671.

[31] Docker - enterprise container platform for high-velocity innovation. Accessed: 11.05.2019. [Online]. Available: https://docker.com

[32] C. Mcluckie. (2014, Aug) Containers vms kubernetes and vmware. [Online]. Available: http://googlecloudplatform.blogspot.com/2014/08/containers-vms-kubernetes-and-vmware.html.

[33] M. Selimi, A. R. Kabbinale, A. Ali, L. Navarro, and A. Sathiaseelan, "Towards blockchain-enabled wireless mesh networks," *CoRR*, vol. abs/1804.00561, 2018. [Online]. Available: http://arxiv.org/abs/1804.00561

[34] Hyperledger, "Five hyperledger blockchain projects now in production," Hyperledger Blog, nov 2018.

[35] Bitcoin energy consumption index. Accessed: 11.05.2019. [Online]. Available: https://digiconomist.net/bitcoin-energy-consumption

[36] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 51–68. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132757

[37] S. Micali, S. Vadhan, and M. Rabin, "Verifiable random functions," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 120–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795665.796482

[38] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[39] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643666

[40] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 173–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296824

[41] J. R. e. a. J. Kreps, N. Narkhede, "Kafka: A distributed messaging system for log processing," in *2011 Proceedings of the NetDB*. Athens, Greece: ACM, jun 2011, pp. 1–7.

[42] Hyperledger fabric docs - peers - transaction flow. Accessed: 11.05.2019. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/peers/peers.html

[43] Fabric ca user's guide. Accessed: 11.05.2019. [Online]. Available: https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/users-guide.html#generating-a-crl-certificate-revocation-list

[44] Hyperledger fabric roadmap. Accessed: 11.05.2019. [Online]. Available: https://wiki.hyperledger.org/display/fabric/Hyperledger+Fabric+Roadmap

[45] G. Greenspan. https://coincenter.org/entry/do-you-really-need-a-blockchain-for-that. Accessed: 19.05.2019. [Online]. Available: https://coincenter.org/entry/do-you-really-need-a-blockchain-for-that

[46] U. Javaid, M. N. Aman, and B. Sikdar, "Blockpro: Blockchain based data provenance and integrity for secure iot environments," in *Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems*, ser. BlockSys'18. New York, NY, USA: ACM, 2018, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/3282278.3282281

[47] Client identity chaincode library. Accessed: 21.05.2019. [Online]. Available: https://github.com/hyperledger/fabric/blob/master/core/chaincode/shim/ext/cid/README.md

[48] Hyperledger fabric client sdk for node.js. Accessed: 24.04.2019. [Online]. Available: https://github.com/hyperledger/fabric-sdk-node

[49] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, Feb. 2012. [Online]. Available: https://doi.org/10.1109/MC.2011.389

[50] Sshfs - sftp remote file access. Accessed: 13.05.2019. [Online]. Available: https://github.com/libfuse/sshfs

[51] Z. I. Berlin. Xtreemfs - fault-tolerant distributed file system for all storage needs. Accessed: 21.05.2019. [Online]. Available: http://www.xtreemfs.org

[52] N. Oceanic and A. Administration. Global surface summary of the day - gsod dataset. Accessed: 27.04.2019. [Online]. Available: https://data.noaa.gov/dataset/dataset/global-surface-summary-of-the-day-gsod

[53] Using couchdb as the state database. Accessed: 25.05.2019. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html

[54] Imageai - easy to use computer vision library for state-of-the-art artificial intelligence. Accessed: 07.05.2019. [Online]. Available: https://github.com/OlafenwaMoses/ImageAI

[55] Idenprof dataset containing images of identifiable professionals. Accessed: 07.05.2019. [Online]. Available: https://github.com/OlafenwaMoses/IdenProf

[56] J. Layton. Sshfs – installation and performance. Accessed: 21.05.2019. [Online]. Available: http://www.admin-magazine.com/HPC/Articles/Sharing-Data-with-SSHFS

[57] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon, "Innovations in internetworking," C. Partridge, Ed. Norwood, MA, USA: Artech House, Inc., 1988, ch. Design and Implementation of the Sun Network Filesystem, pp. 379–390. [Online]. Available: http://dl.acm.org/citation.cfm?id=59309.59338

[58] Openafs - distributed filesystem. Accessed: 21.05.2019. [Online]. Available: https://www.openafs.org/

[59] Amazon. Aws elastic file storage. Accessed: 21.05.2019. [Online]. Available: https://docs.aws.amazon.com/efs/latest/ug/efs-ug.pdf#how-it-works

[60] AbelPelser. Steps to build fabric-javaenv:arm64-1.4.1. Accessed: 08.05.2019. [Online]. Available: https://github.com/Tunstad/Hyperprov/issues/14

[61] Temperature sensor and cpu freq scaling broken in 4.9.x and 4.10.x kernels for rpi3/aarch64. Accessed: 06.05.2019. [Online]. Available: https://github.com/raspberrypi/linux/issues/1918

[62] Psrecord: Record the cpu and memory activity of a process. Accessed: 17.04.2019. [Online]. Available: https://github.com/astrofrog/psrecord

[63] I. Ward. Speedometer: Network & file system speed monitor for ubuntu linux. Accessed: 17.04.2019. [Online]. Available: http://excess.org/speedometer/

[64] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1809049.1809065

[65] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[66] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine

learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 12:1–12:19, Jan. 2019. [Online]. Available: http://doi.acm.org/10.1145/3298981

[67] S. Schelter, F. Biessmann, T. Januschowski, D. Salinas, S. Seufert, and e. a. G. Szarvas, "On challenges in machine learning model management," in *IEEE Data Engineering Bulletin*, 2018.

[68] E. Arslan and A. Alhussen, "Fast integrity verification for high-speed file transfers," *CoRR*, vol. abs/1811.01161, 2018. [Online]. Available: http://arxiv.org/abs/1811.01161

[69] Chainpoint. (2019) Chainpoint Client Node.js. Accessed: 2019-05-16. [Online]. Available: https://github.com/chainpoint/chainpoint-client-js

[70] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC '19)*. IEEE, may 2019. [Online]. Available: http://arxiv.org/abs/1901.00910

Appendix A: Hyperprov system specifications and configuration

Hyperprov is a general purpose provenance framework for data provenance based on the permissioned blockchain Hyperledger Fabric. The system have been run for experiments and evaluation on both commodity desktop hardware and Raspberry Pi devices. This document should give an idea of how the system is built and configured beyond the more general descriptions in the thesis.

Getting started

Before getting started make sure you have the required depdencies. We recommend Go version 1.11.1, Docker 18.09.1, Docker-compose 1.22.0, and NodeJS v8.15.0. For installation help see the install section below.

Set up Docker Compose

To start off we need to configure the docker-compose file `docker-compose-cli.yaml` to orchestrate the devices and paths. The most important are to change paths to this directory, as all devices need to have the `Hyperprov` directory cloned and available. By default it is `/data/Hyperprov` but if you have it elsewere, simply do a find and replace for the path in `docker-compose-cli.yaml`.

The devices used need to be specified in the docker-compose file to match the containers running on them. There are six containers for this here four node setup, whereas your head-node should run both peer, orderer and fabric-tools. To specify device hostnames change the six `node.hostname ==` entries to match your device(s).

Lastly the docker image pointers of each service must be set to match the devices architecture eg. `ptunstad/fabric-ccenv:arm64-1.4.1` or `hyperledger/fabric-ccenv:amd64-1.4.0`.

Docker Swarm

The prototype uses Docker Swarm to manage multiple nodes from a single device for easy management. To start a swarm run `docker swarm init` on your "main" node. This will be the same node you use to start and shut down the network. This will return a command along the lines of `docker swarm join --token SWMTKN-1-xxxxxxxx 192.168.1.xxx:2377` which you need to call on all your other nodes to join the network. You can verify that all nodes have been joined by running `docker node ls` on the initial node. The initial node should also

initialize an overlay network by running `docker network create -d overlay --attachable hyperledger-fabric`.

Starting the network

Before you actually run a network you should regenerate your certificates and genesis block, but for a quick up and running test you can use the block and certificates provided in this repository. See the section below on how to regenerate this.

Assuming all prerequisites are installed and docker images are downloaded, run: `docker node ls` to check that all nodes in swarm is up and running.

To start the network run the command `docker stack deploy --compose-file docker-compose-cli.yaml Hyperprov && docker ps` on your "main" swarm node.

This will start all containers and run a setup script `script_ds.sh` and `utils.sh` to create channel, join peers, install chaincode and do some tests on query and invoke.To follow the output of this script, get the id of CLI-container from `docker ps` that uses fabric-tools and run `docker logs -f 6e4c43c974e7` where `6e4c43c974e7` is the id of CLI-container.

If you encounter any problems run `docker stack ps Hyperprov --no-trunc` on main swarm node to see useful error messages.

Shutting down can be done with `docker stack rm Hyperprov` on main swarm node, this will shut down all nodes in the network and cause it to lose its state.

The Hyperprov Client Library

The Hyperprov client library is used to interact with the Hyperledger Fabric instances running your provenance ledger. It is published to npm and can be downloaded with `npm i hyperprov-client`. The library is also locally configured in this respository `client/hyperprov-client` to easily experiment with modifications. The client library should be used to build an application that needs to interact with the ledger/chaincode. The table below show the functionality currently present in the client library, whereas for simply interacting with the chaincode you could get by with only ccInit, ccPost and ccGet.

Function	Required Input	Expected output
registerAdmin	Keystore, CA-url, CA-name, Adminuname, Adminpw, MSPID	eCert in Keystore
registerUser	Keystore, Username, Affiliation, Role, CA-url, CA-name, MSPID	eCert in Keystore
ccInit	Certificate, Channel+ChaincodeID, Peer+Orderer URL	Success
ccPost	Key, Checksum, Path, Dependency List, Custom Provenance Data	txID
ccGet	Getfunction, Key/txID/Startkey-Endkey	Query Result
storeFile	File, Key, Dependency List, Custom Provenance Data	
retrieveFile	Key, File-path	
InitFileStore	FileStorePath	Success
StoreData	File, Key, Dependency List, Custom Provenance Data	txID
StoreDataFS	File, Key, Dependency List, Custom Provenance Data	Input for StoreDataHL
StoreDataHL	Key, Checksum, Path, Dependency List, Custom Provenance Data	txID
GetDataFS	Key	File, txID

Chaincode and Certificates

Hyperprov Chaincode

The chaincode supports multiple operations related to data provenance. The operations are: storing provenance data of an item, retrieving the last provenance information on an item, requesting the checksum of an item, getting an item with its corresponding transaction ID, getting a specific version of an item from transaction ID, recursively getting all items listed as lineage of a certain item, getting the history of a single item and retrieving a list of items with a key-range query. The current chaincode can be found in `/chaincode/chaincode_hyperprov/chaincode_hyperprov.go`. For the chaincode to implement the desired data provenance functionality it has the following type of data on a stored data item:

Field	Description
txID	Unique transaction ID for each operation
Hash	Checksum of the stored data
Location	First part of data path
Pointer	Second part of data path
Certificate	CA issued unique ID linked to certificate (CID-CC)
Type	Type of operation
Description	Additional metadata, eg. on the process
Dependencies	All txID's that form the lineage of this item

The operations used to implement proveneance for storage and retrieval of data items is the following:

CC Operation	Input	Expected output
get	item data	txID
set	key	JSON with item data of current version of key
checkhash	key	Only checksum of current version of key
getfromid	txID	JSON with specific item data for txID
getdependencies	txID, depth	JSON of txID lineage of item, specified by depth
getkeyhistory	key	JSON with item data for all updates on key
getbyrange	start, end	JSON with item data for all keys in range start-end

Issue chaincode changes

Updates to the chaincode is not issued if it detects already running chaincode with the same version number. To change the version number you need to specify it when it is instantiated in `utils.sh`. Instead you can delete and overwrite the current chaincode. To delete current chaincode this need to be performed on all nodes after a shutdown: `docker stop $(docker ps -aq) && docker rm -f $(docker ps -aq)` Then do `docker images` to find the chaincode container ID, usually the image is named something like `dev-peer0.org1.ptunstad.no-myccds-1.2-97ed6ab7c0e9eda2b3d967ab471-b3691e7eb90fd2b84c0fc33f5c2588b170e4f`. Then do `docker rmi xxxxxxxxxx` replacing the x's with the container ID of running chaincode images to delete them. Now you can start the network with your updated chaincode, just make sure to have it updated on all nodes.

Genesis block and certificates regeneration

Before running a new network you should always regenerate the genesis block and following certificates. Aslo if you need to make any changes to either `crypto-config.yaml` or `configtx.yaml` you may need to regenerate the certificates for your network anyway. To do this first delete the folder `/crypto-config` and `/channel-artifacts`. Then run `export PATH=<replace this with your path>/bin:$PATH` with the full path to your bin folder.
To generate network entities such as peers, organizations and genesisblock run the following:

```
bin/cryptogen generate --config=./crypto-config.yaml
export FABRIC_CFG_PATH=$PWD
bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block
```

Then to generate a channel for our peers to interact on run:

```
export CHANNEL_NAME=mychannel  && bin/configtxgen -profile TwoOr-
gsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -
-channelID $CHANNEL_NAME

bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate
./c-hannel-artifacts/Org1MSPanchors.tx -channelID $CHANNEL_NAME -
-asOrg Org1MSP
```

This should have generated new crypto material and your new network should be able to deploy. The previous eCerts will no longer work for accessing the network so you will also have to generate new certificates for the client.

eCerts and the CA server

To access the network you need eCerts issued by a certificate authority. For this we use Hyperledger Farbrics own fabric CA image which can be retrieved with `docker pull hyperledger/fabric-ca`.

The CA server requires to have Golang installed and to have `sudo apt install libtool libltdl-dev` installed. Then to start it move to `/fabric-ca` and run `docker-compose up -d`. This will start the CA server by default on port 7054 and allow it to respond to requests. You create certificates by first generating an admin certificate using the Hyperprov-client function registerAdmin with the username and password set in the `fabric-ca/docker-compose.yml` file. Once this admin certificate is generated, you can use it to register multiple certificates using the registerUser functionality of the Hyperprov-client. After all certificates are generates the CA server docker container can be shut down again with `docker-compose down`. If any data need to be read, like who owns or registered a certificate, you can access the fabric-ca database in `fabric-ca/fabric-ca-server/fabric-ca-server.db` with sqlite.

Example applications

To show how the system is used and to evaluate its usability we have created a series of applications built on top of the Hyperprov client library.

Benchmark

The benchmark application found in `client\benchmark.js` is used to benchmark the peformance of the system. This supports three differnet types of benchmarks. These are:

- Single benchmark - Runs a single round on benchmark of a specified number of transactions and data size.

- Multi benchmark - Runs a collection of single benchmarks for varying load/data size, calulating average response time, total time and transactions/min from multiple samples.
- Load test - runs a specific level of load over a set time. This means that over 10 minutes we can send 3000 transactions of size 4KB and this benchmark will manage the amount of sleep neccecary between each transaction for an even load.

IoT

The IoT application in `client/iot.js` shows how the system can be used to send a collection of small data and track lineage. It uses the GSOD dataset [1] downloaded using NOAA-GSOD-GET [2]. This dataset is then parsed into records for a few select stations, and stored onto the ledger in batches of 30 records. Every 90 records a data item and the previous "analysed" record is concatinated to mimic analysing data, where the goal is to store the concatinated items as a dependency recorded for the newly created item. This allows us to use the `getdependencies` chaincode functionality to get the full lineage of all "analysed" records.

ML

The machine learning model management example in `client/ml.js` uses models from the IdenProf dataset [3] and models generated by ImageAI [4] to track its provenance metadata. The simple code used to train and recognize based on models can also be found in the `client/ML` folder. Our example application tracks models, testdata and training data. We track the full model files and store in off chain storage, while for testdata and training data we only track file lists. This is done by checking what files are present in specified folders and comparing against records stored in the ledger for differences. If differences are found, the records are updated. When we store models, we list the training dataset and testdata set-lists as dependencies so the metadata includes what test and training data was used to produce the model. We can queue the full history of a single model using `getkeyhistory` or the datasets used to produce it with `getdependencies` chaincode.

[1] https://data.noaa.gov/dataset/dataset/global-surface-summary-of-the-day-gsod
[2] https://github.com/BStudent/NOAA-GSOD-GET
[3] https://github.com/OlafenwaMoses/IdenProf
[4] https://github.com/OlafenwaMoses/ImageAI

REST

The REST-client in `client/API.js` is a simple restful api client built on the express web framework to allow external access to interact with the blockchain trough http-requests. The application supports direct access to chaincode functionality such as `/set`, `/get` , `/getfromid`, `/getkeyhistory`, `/getdependencies`, `/getbyrange`, `/sendfile` and `/getfile`. The client can also only listen to localhost to limit to internal access from other applications on the same device.

Datastore

The application in `client/DataStore.js` is the most baseline appliaction, showing the minimal required code for storing a file in off chain storage and on the ledger, then retrieving it again.

Latency

The latency application in `client/latency.js` is a simple example of sending 100 transactions with five seconds of sleep in between. This is used to measure latency of transactions. The application measures individual times of all 100 transactions and then exports it as .csv format.

Measurements

For measurements on how the system performs we often couple the load test benchmark mode with resource monitoring tools, some of which are listed below.

CPU/Memory

To measure CPU and memory consumption we used the psrecord [5] utility built on psutil. Make sure the measuring node has python, psutil and matplotlib installed. To measure we start off with installing dependencies. These are installed by running `./client/cpu_mem_setup.sh` and possibly also `export DISPLAY=:8` again to fully set up Xvfb which is needed for plotting on headless devices. Then to run measurements edit `client/cpu_mem.sh` with your desired process names, interval, duration and output figures. You can add and remove to record only one, or maybe four processes simultaneously.

[5] https://github.com/astrofrog/psrecord

Network

To capture network usage we used `iftop` and the monitoring tool speedometer 2.8 [6]. To measure with speedometer you can run `speedometer -r eno1 -t eno1` to capture all recieved and transferred network traffic on the network interface `eno1`.

NodeJS Profiler

To profile the client you can use the built in NodeJS profiler by appending the `--prof` flag when running an application. This will output a file named something like `isolate-xxxxxxxx-v8.log`. Then run `node --prof-process isolate-xxxxxxxx-v8.log` to process and output results about the profiling. Here you can explore the number of ticks and percentages occupied the runtime of the profiled process.

Dependencies

Operating System

The experiments were run on Raspberry Pi 3 B+ using the Debian Buster 64-bit Raspberry Pi OS which you can download from here [7]. 64-bit OS will only run on Raspberry Pi 3 and is currently required as HLF only supports 64 bit. This guide targets Raspberry Pi because they are most difficult to set up, but most of this applies to Ubuntu 16.04 by replacing `arm64` with `amd64` and pulling docker images directly from the `hyperledger/` repo.

Start by making sure your system is up to date and have some important dependencies used in the following steps: `apt-get update && apt-get install curl wget sudo`.

Docker and Docker Compose

Start off by installing Docker and Docker compose. We have tested only with Docker 18.09.1 and Docker compose 1.22.0, so if problems arise revert to these versions.

```
curl -sSL https://get.docker.com | sh
curl -s https://packagecloud.io/install/repositories/Hypriot/Schatz-
kiste/script.deb.sh | bash
apt-get install docker-compose
```

[6] http://excess.org/speedometer/
[7] https://wiki.debian.org/RaspberryPi3

To check if you installed them correctly run `docker --version && docker-compose --version` if it does not work, a reboot will usually solve this problem.

Docker images

Because no HLF docker images are officially available from Hyperledger Fabric, i have compiled my own images for HLF v1.4 on ARM64 [8]. If you want to compile your own images see Appendix B or Github[9].

```
docker pull ptunstad/fabric-baseos:arm64-0.4.15 &&
docker pull ptunstad/fabric-basejvm:arm64-0.4.15 &&
docker pull ptunstad/fabric-baseimage:arm64-0.4.15 &&
docker pull ptunstad/fabric-ccenv:arm64-1.4.1 &&
docker pull ptunstad/fabric-peer:arm64-1.4.1 &&
docker pull ptunstad/fabric-orderer:arm64-1.4.1 &&
docker pull ptunstad/fabric-zookeeper:arm64-1.4.1 &&
docker pull ptunstad/fabric-kafka:arm64-1.4.1 &&
docker pull ptunstad/fabric-couchdb:arm64-1.4.1 &&
docker pull ptunstad/fabric-tools:arm64-1.4.1 &&
docker pull apelser/fabric-javaenv:arm64-1.4.1
```

Golang

The version of Golang used for Hyperledger Fabric v1.4.1 is Go 1.11.1, installing it on RPI can be done by:

```
wget https://dl.google.com/go/go1.11.1.linux-arm64.tar.gz
tar -C /usr/local -xzf go1.11.1.linux-arm64.tar.gz
echo 'export PATH=$PATH:/usr/local/go/bin' >> ~/.profile
echo 'export GOPATH=$HOME/go' >> ~/.profile
```

To verify run `go version` and `echo $GOPATH` to verify its set to `/home/pi/go`.

Node

To run the applications you need NodeJS and npm installed. We encountered some errors with newer versions of node and therefore encourage you to downgrade to version 8 to avoid problems for now.

```
curl -sL https://deb.nodesource.com/setup_8.x | -E bash -
sudo apt-get install -y nodejs
```

[8] https://hub.docker.com/r/ptunstad/
[9] https://github.com/Tunstad/Hyperprov/blob/master/compiling.md

Other

If you get a problem due to missing dependencies other than the ones listed above, installing some of these additional packages may help:

```
apt-get install git python-pip python-dev docker-compose build-essen-
tial libtool libltdl-dev libssl-dev libevent-dev libffi-dev
pip install --upgrade pip
pip install --upgrade setuptools
pip install behave nose docker-compose
pip install -I flask==0.10.1 python-dateutil==2.2 pytz==2014.3 pyyam-
l==3.10 couchdb==1.0 flask-cors==2.0.1 requests==2.4.3 pyOpenSSL==16-
.2.0 pysha3==1.0b1 grpcio==1.0.4
pip install --trusted-host pypi.org docker-compose
```

Clone repository to /data folder

The docker compose file currenty relies on the code being placed in /data/Hyperprov. To clone this repository there do the following:

```
sudo mkdir /data && sudo chmod -R ugo+rw /data
git clone -b "master" https://github.com/Tunstad/Hyperprov.git
```

Otherwise you can edit the path in `docker-compose-cli.yaml`.

Swap Partition

The Raspberry Pi devices may run out of memory during execution or start up, which will cause a crash. You especially need to do this if you want to build your own docker images. To avoid running out of memory i suggest setting up a swap partition if not already present. You can check for existing swap either with `top` or `swapon --show`. 1GB of swap should be more than enough, to create perform the following actions:

```
sudo fallocate -l 1G /swapfile
sudo dd if=/dev/zero of=/swapfile bs=1024 count=1048576
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
sudo echo '/swapfile swap swap defaults 0 0' >> /etc/fstab
```

Appendix B: Building Hyperledger Fabric v.1.4 to Docker Images for Raspberry Pi on ARM64

Because there seemingly is no guide for building HLF images from source, i decided to write one, specifically targeted towards Raspberry Pi devices. The base operating system used to compile the images is Debian buster aarch64 64-bit operating system, with unofficial support on Raspberry3. The source image used can be downloaded from the Debian page on RPI3 [1]. The kernel used during the successful builds were *4.18.0-3-arm64*.

Dependencies

The dependenices for building are similar to those required to run the system and guides for installing on Rasbperry Pi can be found in the Install section of Appendix A. Your will need *docker*, *docker-compose* and *golang*(specifically version 1.11.1 for Fabric v1.4). To compile the javaenv repository you will also need to install *gradle*.

```
sudo apt-get install git curl gcc libc6-dev libltdl3-dev python-setuptools -y
```

Set gopath with `export GOPATH=~/golang` and make symbolic link as some of the files use another path `ln -s ~/golang ~/go`.

Then you can start downloading repositories to ~/golang/src/github.com/hyperledger/

```
git clone https://github.com/hyperledger/fabric-baseimage
```

```
git clone https://github.com/hyperledger/fabric
```

```
git clone https://github.com/hyperledger/fabric-chaincode-java
```

Building Baseimages

Before you can build baseimages you need to add the line `DOCKER_BASE_aarch64=aarch64/ubuntu:xenial` to the Makefile. Then you can run `make -f Makefile` or `make docker` to build baseimage, basejvm and baseos images. To build third party images like kafka, zookeeper and couchdb you can run `make dependent-images`. After building you should see the images in `docker images`.

CouchDB error

CouchDB may return an error when built, the solution is to edit the file `fabric-baseimage/images/couchdb/Dockerfile.in` to find the line

[1] https://wiki.debian.org/RaspberryPi3

`&& ./configure --disable-docs \` and add the following lines after it

```
&& chmod +w bin/rebar \
&& mv bin/rebar bin/rebar-orig \
&& cd bin \
&& curl -fSL https://github.com/rebar/rebar/wiki/rebar --output rebar \
&& chmod +x rebar \
&& cd .. \
```

Thanks to Sasha Pesic and YR Sang on the Hyperledger JIRA for solving this. [2]

Building Fabric Images

For the resulting images such as peer, orderer, tools and ccenv we need to go to the fabric folder cloned with `git clone https://github.com/hyperledger/fabric`. Here we need to make a few modifications as pointed out in this thread [3]. The main changes are in the core.yaml files, in HLFv1.4 there exist two versions of this, you can find them using `find . -name core.yaml`. I chose to edit both, although it might not be necessary. The patch code is

```
     # Gossip related configuration
     gossip:
-        bootstrap: 127.0.0.1:7051
         # Use automatically chosen peer (high avalibility) to distr-
         ibute blocks in channel or static one
         # Setting this true and orgLeader true cause panic exit
         useLeaderElection: false
@@ -280,7 +279,7 @@ vm:
                 Config:
                     max-size: "50m"
                     max-file: "5"
-            Memory: 2147483648
+            Memory: 16777216
```

The other modification mentioned in dockerutil.go is not present in HLFv1.4 so it might not be required any more. Further if the variable BASEIMAGE_RELEASE in Makefile does not match the baseimage built in the above step, change it accordingly. Now to build you can run `make docker` to compile all images, or build them individually with `make peer peer-docker make orderer orderer-docker make tools-docker make buildenv make ccenv`.

[2] https://jira.hyperledger.org/browse/FAB-11912
[3] https://stackoverflow.com/questions/45800167/hyperledger-fabric-on-raspberry-pi-3

Bulding binary executables

Now we will build the binary executables like configtxgen, cryptogen and more. Move to the fabric folder and run `make native`. The images will appear in `.build/bin` and can then be copied to the /bin folder of your Hyperledger Fabric project repository.

Building Javaenv

The image 'fabric-javaenv' is required to run java chaincode in Hyperledger Fabric. It was in version 1.1 separated from the other fabric build projects into a separate project which can be cloned with `git clone https://github.com/hyperledger/fabric-chaincode-java`. You may have to install Gradle locally aswell with `apt-get install gradle`. I have not yet had success with building this image on the arm64 architecture due to issues with the packages `com.google.protobuf` and `protoc-gen-grpc-java` not beeing available for arm64/aarch64.

Thanks to AbelPelser [4] for writing steps to build javaenv. The prebuilt image can be downloaded with `docker pull apelser/fabric-javaenv:arm64-1.4.1`. Steps to build it can be found here [5].

[4] https://github.com/AbelPelser
[5] https://github.com/Tunstad/Hyperprov/issues/14