Performance Modeling & Analysis of Hyperledger Fabric (Permissioned Blockchain Network)

by

Harish Sukhwani

Department of Electrical & Computer Engineering
Duke University

Date: _____

Approved:

_____
Dr. Kishor Trivedi, Supervisor

_____
Dr. John Board

_____
Dr. Krishnendu Chakrabarty

_____
Dr. Daniel J. Sorin

_____
Dr. Andrew Rindos (External)

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Electrical & Computer Engineering
in the Graduate School of Duke University
2018

Abstract

Performance Modeling & Analysis of Hyperledger Fabric (Permissioned Blockchain Network)

by

Harish Sukhwani

Department of Electrical & Computer Engineering
Duke University

Date: _____

Approved:

_____
Dr. Kishor Trivedi, Supervisor

_____
Dr. John Board

_____
Dr. Krishnendu Chakrabarty

_____
Dr. Daniel J. Sorin

_____
Dr. Andrew Rindos (External)

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical & Computer Engineering
in the Graduate School of Duke University
2018

Abstract

A blockchain is an immutable record of transactions (called *ledger*) between a distributed set of mutually untrusting peers. Although blockchain networks provide tremendous benefits, there are concerns about whether their performance would be a hindrance to its adoption. Our research is focused on Hyperledger Fabric (HLF), which is an open-source implementation of a distributed ledger platform for running smart contracts in a modular architecture. This thesis presents our research on performance modeling of Hyperledger Fabric using a Stochastic Petri Nets modeling formalism known as Stochastic Reward Nets (SRN). We capture the key system operations and complex interactions between them. We focus on two different releases of HLF, viz. v0.6 and v1.0+ (V1). HLF v0.6 follows a traditional state-machine replication architecture followed by many other blockchain platforms, whereas HLF V1 follows a novel *execute-order-validate* architecture. We parameterize and validate our models with data collected from a real-world Fabric network setup. Our models provide a quantitative framework that helps compare different deployment configurations of Fabric and make design trade-off decisions. It also enables us to compute performance for a system with proposed architectural improvements before they are implemented. From our analysis, we recommend design improvements along with the estimates of performance improvement. Overall, our models provide a stepping stone to the Hyperledger Fabric community towards achieving optimal performance of Fabric in the real-world deployments.

Contents

List of Tables

List of Figures

Acknowledgements

I am deeply indebted to Prof. Kishor Trivedi for providing me an opportunity to be a part of his research group. His constant feedback on my work helped me stay on course to deliver on my projects. The open-ended questions that he asked me during the group meetings taught me how to think like a researcher and search for meaningful problems on my own. He inspired us to go one step further and collaborate with the funding agency/collaborator to implement the solution in practice. Although such efforts might not result in new papers, he taught me the importance of closing the loop, thus delivering meaningful results beyond just publications. In spite of all the challenges we face as researchers, he taught me by example on how to be kind and generous to fellow researchers and be a good human being.

I would like to thank my dissertation committee members - Prof. John Board, Prof. Krishnendu Chakrabarty, Prof. Dan Sorin - for their feedback and encouragement on this journey. I am especially grateful to Dr. Andrew J. Rindos for facilitating our close collaboration with IBM that helped shape my research direction. His insightful feedback and collaboration of various projects taught me valuable lessons on research delivery.

I would like to thank the following organizations for their generous financial support during the various phases of my graduate school: Duke University Graduate School, Department of Electrical & Computer Engineering, IBM and NASA GSFC.

Thanks to the members of our research group - Javier Alonso, Rafael Fricks,

Xiaodan Li, José M. Martínez, Nan Wang, Ruofan Xia - for helpful discussions, comments, and feedback. Thanks to the visitors of our group - Xiaolin Chang, Zheng Zheng - for their generous feedback about my work. Special thanks to Dr. Andrea Bobbio who mentored me to write my first research paper, gave me opportunity to work on the Green book and for all the good times at IIT Gandhinagar.

I would like to thank Dr. Dorothea Wiesmann, Dr. Jens Jellito and team members at IBM Research - Zurich for providing me an opportunity to deep-dive into Hyperledger Fabric. In addition, thank you Konstantinos Christidis for facilitating meaningful discussion and collaborations in this area.

Thanks to the staff members at ECE, particularly Amy Kostrewa for all her assistance during the various stages of the graduate program, Olena Aleksandrova and Alex Naseree for scheduling group meetings and appointments.

This journey would not have begun without the encouragement and support of my family members, Lata and Manish Assudani, Meena and Manoharlal Sukhwani, who inspired me to leave behind a full-time job with a work visa to pursue my dream career. I could not have finished this journey without the unconditional love and support of my parents Vijaya and Vasudev Sukhwani, and my wife, Palak. In spite of our careers keeping us geographically apart, Palak made sure we spent meaningful time together. Our exciting trips together in Europe kept us sane. I am excited about rebooting to a new life with her in Portland, Oregon.

Thanks to my friends Abhinandan and Aditi for the fun times and cheering me during my gloomy days. Thanks to my friends Abhishek, Anish, Mayuresh, and Nisarg for their great company. Special thanks to my doctor Tom Mitchell for taking care of my health and well-being during the crucial final year of this journey. Finally, I would like to thank my friends at the JKYog NC organization - Ajith, Hina, Jayesh, Laxmi, Neela, Neeta, Parag, Sonali - in whose company I continued my spiritual journey.

1

Introduction

According to National Institute of Standards and Technology (NIST) [1], "Blockchains are immutable digital ledger systems implemented in a distributed fashion (i.e., without a central repository) and usually without a central authority." Each peer maintains a copy of the ledger. For a "block" of proposed transactions in the network, the peers obtain a *consensus* on the validity of the block of transactions. A new block of transactions is appended with the hash of the previous block committed on the ledger, thus forming a hash chain of blocks. This ordered back-linked list of blocks gives it the name blockchain. Therefore, a blockchain network is a distributed database/transaction system where all the peers share information in a decentralized, trusted and secure manner. Thus a cryptocurrency such as Bitcoin [2] is an application running on the blockchain network.

Blockchain network enables trusted parties to send transactions in a peer-to-peer fashion in a verifiable manner, without the need of a trusted intermediary. This peer-to-peer network allows parties to settle transactions quicker, resulting in faster movement of goods and services [3]. Introduction of blockchain networks in businesses and enterprises will bring out ground-breaking changes in the way they

function and operate [3]. It would result in never-seen-before business models as well as a transformation of the existing business models [4]. Blockchains are widely regarded as a promising technology to run trusted exchanges in the digital world.

A significant selling point for organizations to move to a blockchain network is the ability to automate business transactions using smart contracts [5]. A smart contract [6] is a collection of business rules that are shared and validated collectively by a group of stakeholders, which can be deployed on a blockchain [7]. Such smart contracts help execute business processes in an automated and trusted manner. A smart contract is invoked by a transaction referring to it. Thus, a transaction is a request to the blockchain to execute a function on the ledger, which is implemented as a smart contract.

Blockchain networks can be classified based on the following criteria: a) which clients are allowed to submit transactions, b) which peers are allowed to order transactions (including consensus), c) how are new clients/peers authorized to join the network. In a *public* or *permissionless* blockchain network, anyone can participate in the network without a specific identity. Such networks usually involve a native cryptocurrency or other economic incentives. Popular examples are Bitcoin [2] and Ethereum [8]. Such networks use lottery-based consensus protocols such as proof-of-work (PoW). A *permissioned* blockchain network is operated by known entities as in a *consortium* where members or stakeholders in a given business context [9]. All the participants are whitelisted and bounded by strict contractual obligations to behave "correctly" [5]. New peers can be added by permission from the existing peers by approval or pre-defined rules or with permission from a regulator [10]. Such networks use voting-based consensus protocols such as Practical Byzantine Fault Tolerance (PBFT) [11]. There is no inherent need of a cryptocurrency in such networks. A *private* blockchain network is a special case of permissioned blockchain operated by a single entity. In summary, organizations participating in permissioned blockchain

networks can benefit from a distributed ledger technology (DLT) without a need of a cryptocurrency [12].

Hyperledger project is an open source collaborative effort hosted by the Linux Foundation to advance blockchain technologies for business enterprises [13, 14]. The collaborators include leaders from technology, finance, banking, supply chain, IoT and manufacturing companies. In this thesis, we focus on Hyperledger Fabric [15], which is an open-source implementation of a distributed ledger platform for running smart contracts in a modular architecture. It aims at high degrees of confidentiality, resilience, flexibility, and scalability [9]. It is currently deployed in more than 400 proof-of-concept and production distributed ledger systems across different industries and use-cases [16]. Hyperledger Fabric blockchain platform is being used to solve problems in diverse areas such as food safety [17], trade-finance [18] and supply-chain [19].

Although blockchain networks provide tremendous benefits, there are concerns about whether their performance would match up with the mainstream information technology (IT) systems. For public networks such as Bitcoin, the block frequency is ten minutes [20], which means it takes ten minutes or more confirm a transaction. A recent study by [21] shows that Bitcoin achieves a maximum throughput of 7 transactions per second, which is abysmal compared to a mainstream payment system. Fortunately, permissioned blockchain networks can be designed to use more efficient consensus mechanisms such as PBFT, resulting in much higher throughput and lower latency with less computation, bandwidth, and storage requirements [5, 21]. The empirical results for the PBFT protocol are promising, with a block latency of 288ms and throughput of 113k transactions per second for four peers located in the same geographical region [21]. In a similar setup, Hyperledger Fabric has been observed to achieve maximum throughput of 3500 transactions per second with sub-second latency [22]. Another concern is the scalability, whether the performance can keep

up with an increasing number of peers.

Performance evaluation of the system by experiments is necessary; however, it is tedious and time-consuming. For example, the authors in [21] considered a large number of peers (up to 64) spread across eight geographies and multiple block sizes. If such details can be captured in the model, the model can compute performance metrics as a function of various system configurations and parameters. Stochastic models provide a quantitative framework that helps compare different configurations and make design trade-off decisions. It also enables us to compute performance for potential architectural changes that the software engineers are considering for the future releases.

In this dissertation, I present our research on performance modeling of Hyperledger Fabric. We focused on two different releases of Hyperledger Fabric, viz. v0.6 and v1.0+ (V1), each with a different architecture. I also provide an extensive empirical analysis of model parameterizations and other vital observations for both the releases.

Hyperledger Fabric v0.6 has a traditional state-machine replication architecture followed by many other blockchain platforms. Each block of transactions undergoes a consensus process among the participating peers before it is added to the distributed ledger. The consensus is achieved using Byzantine fault tolerant (BFT) protocols such as Practical Byzantine Fault Tolerance (PBFT). There are concerns whether this consensus process could be a performance bottleneck for networks with a large number of peers. To understand this in depth, we modeled it using Stochastic Reward Nets (SRN). We compute the "mean time to complete consensus" for networks up to 100 peers. We created a blockchain network using IBM Bluemix service, running a production-grade IoT application and used the data to parameterize and validate our models. For four peers, we find that the solutions from our model are comparable to the empirical results. Our sensitivity analysis results show that the

mean time to consensus increases by three times the increase in the message transmission time between peers (hence the geographical distance), due to three phases of communication involved. For larger networks, we find that the mean time to consensus increases slowly up to 10 peers and then increases linearly with the number of peers. We also find that the percentage increase in the mean time to consensus with the number of peers is significant only if the message transmission time is of the same order of magnitude as the message processing and queuing time.

To overcome the limitations of v0.6, the Hyperledger Fabric community completely reworked the original architecture, which was released as v1.0 (V1). This reworked system follows a novel *execute-order-validate* architecture, which is a hybrid of passive and active replication. We developed a comprehensive performance model using SRN from which we can compute the throughput, utilization and mean queue length at each peer and critical processing stages within a peer. To validate our model, we setup an HLF network in the Duke datacenter and generated a realistic workload using Hyperledger Caliper [23]. From our analysis results, we find that the endorsing process could be a performance bottleneck, especially when AND() endorsement policy is used. For the committing peer, the transaction validation check (using Validation System Chaincode (VSCC)) is a time-consuming step, but its performance impact can be easily mitigated since it can be parallelized. However, its performance is critical, since it absorbs the shock of bursty block arrivals. The performance bottleneck of the ordering service and ledger write can be mitigated using a larger block size, albeit with an increase in latency. We also analyze various *what-if* scenarios, such as peers processing transactions in a pipeline, and multiple endorsers per organization.

In both the projects mentioned above, we modeled the system using a Stochastic Petri Nets modeling formalism known as Stochastic Reward Nets (SRN) [24]. Such a formalism allows a concise specification and an automated generation/solu-

tion of the underlying (stochastic) process that captures the performance behavior of the blockchain network system. Moreover, using SRNs allows us also to study different scenarios, by easily adding or removing system details. SRNs have been successfully used to model different computer/communication systems from the performance perspective, e.g., [25, 26, 27, 28]. If the firing times for all transitions are exponentially distributed or can be represented by a subnet consisting of exponentially distributed firing time transitions, then the underlying stochastic process is a homogeneous continuous-time Markov chain, which makes SRN a natural choice for a modeling formalism. Even if the firing times of some transitions are not exponentially distributed (as in our analysis in Section 5.3), SRN is a convenient way to define a scalable model for such a complex process, and compute output parameters using discrete event simulation techniques.

1.1 Contributions of the Dissertation

For our work on modeling the PBFT consensus process for HLF v0.6, the research contributions are:

1. Scalable model of the PBFT consensus process
2. Model validated using data collected from an IBM cloud deployment.
3. Evaluation of consensus process for a large number of peers (up to 100)
4. Sensitivity analysis as a function of time to process, queue and transmit a message.

Next, for our work on performance modeling of HLF V1 network. The research contributions are:

1. A comprehensive performance model of the Fabric V1 blockchain network. For Fabric's unique blockchain network architecture, it captures the key steps performed by each subsystem as well as interactions between them.

2. Analysis for critical *what-if* scenarios that system developers and practitioners care about.
3. Model validated using data from a multi-node experimental setup in Duke datacenter.

Since different blockchain networks are designed with varying assumptions of security and use-cases, the system metrics defined in one class of systems are not necessarily applicable to another. In this thesis, I also attempt to present crisp and precise definitions of the performance metrics relevant across all blockchain networks.

1.2 Outline of the Dissertation

This dissertation is organized as follows: In Chapter 2, we provide the background material on the Hyperledger Fabric distributed ledger platform.

In Chapter 3, we discuss the performance metrics used in the literature in the realm of blockchain networks and present crisp and clear definitions of metrics. We also discuss metric definitions applicable specifically to the Hyperledger Fabric.

In Chapters 4 and 5, we discuss performance modeling and empirical analysis for Hyperledger Fabric v0.6. We provide a background of the PBFT consensus process. We also provide details of our experimental setup and model validation. Overall, these two chapters are extended versions of our published paper [29].

In Chapters 6 and 7, we discuss performance modeling and empirical analysis for Hyperledger Fabric V1. First, we present our overall system model, its analysis, and overall model validation. Next, we present models and analysis for each transaction phase, viz. endorsement, ordering, and validation. We also share extensive details of the experimental setup, tools used, Fabric software code changes, and the tools for analysis, such that the experiments can be reproduced and the study can be replicated for future modeling and analysis goals.

In Chapter 8, we discuss our efforts on model verification and validation. We share our experience, the challenges involved and how we overcame them towards our goals. We also share our threats to the validity of the results of this dissertation.

Finally, we conclude our thesis in Chapter 9, and outline future avenues for research in performance aspects of blockchain networks. We briefly describe how the models developed in our study can be integrated as a tool that designers, developers, and operators of Hyperledger Fabric could use to optimize the system performance and other design trade-offs.

2

Overview of Hyperledger Fabric

In this chapter, we provide an overview of the Hyperledger Fabric distributed ledger platform. We start with Hyperledger Fabric's preview release v0.6. Unfortunately, it had significant limitations in its architecture, resulting in poor performance and scalability, among many other things. The community decided to go with a completely different architecture, which was released as v1.0 [30]. Given the vast popularity of this release, its architecture is expected to be continued in the upcoming releases. Although the system architecture of release v0.6 is discontinued by the Fabric community, many other blockchain platforms implement a similar architecture at a high-level and hence is worth reviewing it from a performance and scalability perspective.

2.1 Key Concepts

2.1.1 Smart Contracts (chaincode)

All operations in the HLF are performed using smart contracts (known as *chaincode* in HLF). A smart contract (SC) is a collection of business rules that are shared and validated collectively by a group of stakeholders [7]. Such smart contracts help execute business processes in an automated and trusted manner. Based on the

business logic, several functions can be defined within a smart contract. When a client issues a transaction request, the smart contract is invoked on the peers. Thus, a transaction is a request to the blockchain to execute a function on the ledger, which is implemented as a chaincode. Unlike other blockchain platforms (such as Ethereum) where smart contracts are written in a specialized programming language, Fabric supports chaincode in general-purpose programming languages (e.g., Go, Java, Node.js) running in standard Docker containers [31].

Fabric also supports system chaincodes that are built into peer executable and have the same programming model as application chaincodes. These are:

- Lifecycle system chaincode (LSCC) - to install/instantiate/update chaincode.
- Endorsement system chaincode (ESCC) - to endorse a transaction.
- Validation system chaincode (VSCC) - to validate transaction's endorsement set against its endorsement policy.
- Configuration system chaincode (CSCC) - to manage channel configuration.

Further details are available in the software documentation. During the transaction life-cycle, the peers invokes the system chaincode corresponding to each life-cycle phase. The default system chaincodes work fine out-of-the-box but can be customized by the organizations running the Fabric platform based on their requirements. The most prominent one to be customized is the VSCC.

2.1.2 Consensus

The fundamental building block for a distributed ledger system is the distributed consensus process. The consensus process ensures that all the transactions in the network are agreed upon and executed serially.

It is worth highlighting the difference in the consensus process for a public blockchain network such as Bitcoin versus a permissioned blockchain network such

as Hyperledger Fabric. In a public network, any one can join the network, which induces the risk of Sybil attack [32]. Bitcoin resolves this by making it computationally expensive for a peer to propose a new block of transactions (a process called "mining"). This approach is called proof-of-work (PoW) where each peer needs to find the right random number (nonce) in the block header such that the SHA-256ˆ2 hash value will have a defined high number of leading zeroes [33]. In a permissioned network, all the participants are whitelisted and bounded by strict contractual obligations to behave "correctly", and hence there is no need for a costly consensus process [5].

Hyperledger Fabric v0.6 used a popular consensus protocol called Practical Byzantine Fault Tolerance (PBFT). We discuss relevant details in Section 5.1. For Fabric V1, the latest release of Fabric at the time of writing this thesis (v1.2) only supported a crash-fault tolerant consensus using Kafka. More details in Section 2.3.6. The development of a Byzantine-fault tolerant consensus for Fabric V1 is underway[1].

2.2 Hyperledger Fabric v0.6

2.2.1 System overview

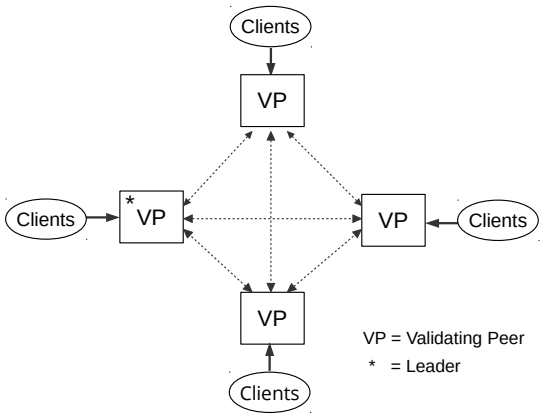


FIGURE 2.1: System model of a Fabric v0.6 network with four validating peers (VPs)

[1] https://lists.hyperledger.org/g/fabric/topic/22676649

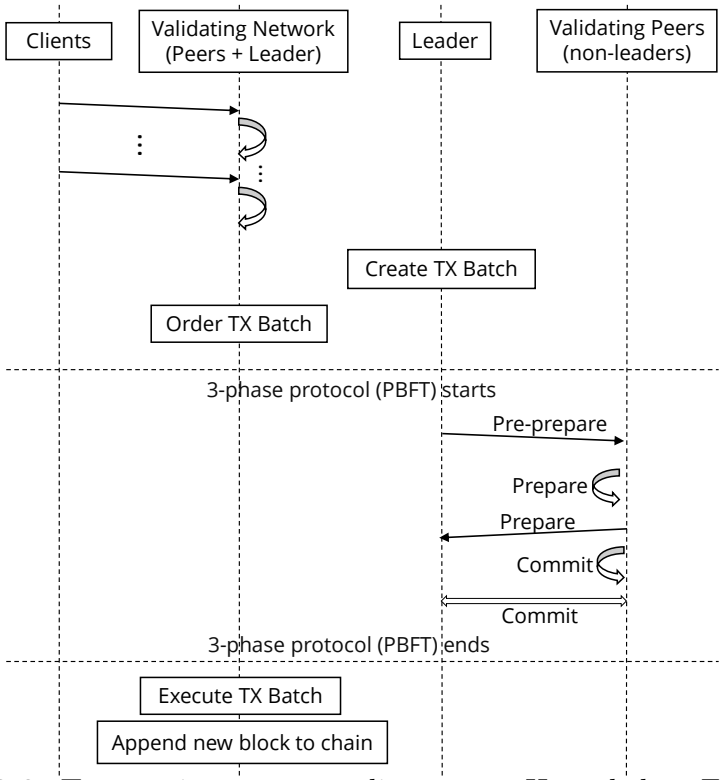


FIGURE 2.2: Transaction sequence diagram on Hyperledger Fabric v0.6

Consider a simple network (Figure 2.1), where four institutions would like to have a distributed ledger of transactions shared between them. In blockchain terminology, each of these institutions is a Validating Peer (VP). One of the validating peers is elected to be the leader (marked with $*$). The leader changes periodically or in certain situations as described later.

The clients make transaction requests to their respective institutions via their VP. The VP validates the transaction and then broadcasts this transaction to other VPs via a secure link, logically indicated by the dashed arrow in the figure. Figure 2.2 provides another view of this process, where the first column represents all the clients on the network, and the second column represents the collection of VPs and leader. The transactions between the leader and VP are shown in the third and fourth column respectively. The clients send transactions to the validating network, which are ordered and batched together as a block by each VP. After a few seconds (defined

as *batch timeout*) or after a set number of pending transactions (defined as *batch size*), the leader creates a block of the pending transactions, maintaining order by timestamp. The leader then broadcasts this candidate block to other VPs to obtain a consensus on the block. This consensus is achieved using a three-phase protocol such as PBFT [11], which we discuss in detail in Section 5.1.

2.2.2 Data structures

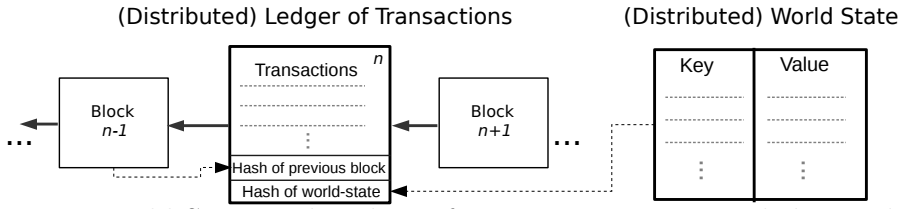


FIGURE 2.3: World State and Ledger of transactions in Hyperledger Fabric v0.6

HLF maintains a global state across all the peers using a versioned key-value store (KVS) (called *world state*) and the ledger, as shown in Figure 2.3. The state of each chaincode can be represented as a collection of key-value pairs. The *world state* of a peer refers to the ensemble collection of states of all chaincodes. Thus the key for the world state is provided as $\{ChaincodeID,\ ckey\}$, where *ChaincodeID* refers to a unique chaincode deployment, and *ckey* is a unique key corresponding to the *chaincodeID*. Before recording block onto the ledger, we need to compute the cryptographic hash (crypto-hash) [34] of the world state observed by a peer. Since this is an expensive operation, world state needs to be organized in a way that enables efficient computation of crypto-hash. Following section describes the implementation details of the world state.

2.2.3 Block Execution process

Block execution process includes executing individual transactions using its corresponding chaincode, update the results in the world state corresponding to the chain-

code, and finally add the block to blockchain and update world state on the ledger.

A block to be added to blockchain contains the list of transactions, the hash of the transactions after execution, hash of the previous block in the chain (*previousBlockHash*), and hash of world state after executing all transactions in the block. Note that only the transaction headers are included in the block for compactness [35]. Following is the prototype of the message Block [35].

```
message Block {
  version = 1;
  google.protobuf.Timestamp timestamp = 2;
  bytes transactionsHash = 3;
  bytes stateHash = 4;
  bytes previousBlockHash = 5;
  bytes consensusMetadata = 6;
  NonHashData nonHashData = 7;
}

message BlockTransactions {
  repeated Transaction transactions = 1;
}
```

To calculate *previousBlockHash*, the block is first serialized using Protocol buffers library [36], and then the hash is calculated using SHA3 SHAKE256 algorithm [35]. The *transactionsHash* and *stateHash* are computed after executing all the transactions in the block. Both are computed as the Merkle root hash (next paragraph).

Bucket tree is one of the implementations for organizing the world state [35]. In this method, the world state is stored in a hash-table consisting of pre-defined no. of buckets (*numBuckets*). A hash function (not a crypto-hash) determines which bucket number contains a given key. These buckets form the leaf nodes of the tree. At the next level, the pre-defined no. of nodes (defined by *maxGroupingAtEachLevel*) are grouped. This process continues for constructing the next higher level until the root node is built. To compute the crypto-hash of the world-state, a Merkle-tree [37] is modeled on top of buckets of the hash table [35]. First, the contents of the

leaf-node buckets are serialized (described in detail in [35]), and then the crypto-hash of leaf-node buckets are computed. For upper-level nodes, the crypto-hash of lower-level nodes are serialized, and crypto-hash is calculated. Finally, the crypto-hash of the root node is considered as the crypto-hash of the world state. Note that the bucket tree's depth (*numBuckets*) and breadth (*maxGroupingAtEachLevel*) can have different implications on the performance cost and resource demanded of various system resources.

Fabric's peer-to-peer communication is built on the Google protocol remote procedure call (gRPC) [38, 35]. It allows bi-directional stream-based messages. It utilizes Protocol Buffers (protobuf) to serialize data structures for transfer between peers. The gRPC is also used in communication between VP and its deployed chaincode container. Chaincode container has a shim layer to handle message protocol between chaincode and VP using protobuf messages.

Based on our understanding from the documentation [35] and reviewing the DEBUG logs messages generated in the VP, we summarize the major steps taken during the block execution in the perfect-world case. First, the VP verifies whether the chaincode corresponding to the transaction is already invoked. Then the chaincode sets its security context, and then triggers the transaction that consists of a sequence of *get_state* and *put_state* messages to the VP via the shim layer. Thus chaincode queries for values of specific assets from the world state, then updates the value and puts them back to the world state. During these interactions, the bucket-tree module keeps track of the leaf-node buckets that have changed. Then, crypto-hash is computed for the transaction block. These steps are run in a sequence for all the transactions on the block. After all transactions are over, the hash of the block is computed. Next, the bucket tree computes the crypto-hash for all the leaf-node buckets in the world state that were modified by the transactions. Then the crypto-hash is computed for the next-level parent buckets corresponding to leaf-node

buckets, and this process is repeated until the root node's crypto-hash is computed. Next, the VP stores the key-value pairs that were changed during the transactions in persistent storage. Finally, the block is committed on the ledger.

2.3 Hyperledger Fabric V1

The architecture of many smart-contract based blockchain platforms resembles closely to the traditional state-machine replication approach [39]. These systems implement active replication: first, the consensus protocol orders the transactions and propagates them to all peers; second, each peer executes all the transactions sequentially. This can be called an *order-execute* architecture [22]. Examples are Ethereum, Tendermint (`http://tendermint.com`), Chain (`https://www.chain.com`), Quorum (`https://www.jpmorgan.com/global/Quorum`) and HLF v0.6. However, this architecture suffers from several limitations that made the real-world business use cases hard to implement [12, 22].

- Consensus is hard-coded within the platform and cannot be changed as per requirements;
- Application needs to use the same trust model as that provided by the underlying consensus mechanism (such as $f = \lfloor (n-1)/3 \rfloor$ in case of PBFT);
- All smart contracts need to be loaded on all peers, which made confidentiality hard;
- All transactions were executed on all peers, which resulted in significant performance decay; Also complex measures required to prevent DoS attack such as Gas in Ethereum;
- Non-deterministic execution in a smart contract can take the entire system down.

The new architecture, termed Fabric V1 here, separates the execution of transac-

tions (via SC) from the ordering of transactions. The transaction flow follows three steps: a) *execute* a transaction on a subset of peers, which *endorse* the transaction; b) *order* the transaction using a consensus protocol; c) *validate* transaction following the application-specific trust assumption, but prevent concurrency-related race conditions. These three steps can be performed on separate peers, thus improving the concurrency significantly.

This fundamental shift from the state-machine replication approach has several advantages, including better scalability, new trust assumptions for transaction validation, support for non-deterministic SC, and using modular consensus implementations [12, 22]. Thus, Fabric V1 follows an *execute-order-validate* architecture [22], where transactions can be executed even before there is a consensus on its ordering. It follows a hybrid replication of active and passive replication. It follows passive replication, whereby transactions are executed by a subset of peers independent of other transactions. It follows active replication, whereby transactions are committed on the ledger only after reaching consensus on the transaction order. This hybrid replication is the heart of the Fabric V1.

Let us introduce some of the key aspects of the system.

2.3.1 Nodes

A blockchain network consists of many nodes communicating with each other to collectively process transactions. A node is a virtual entity, in the sense that it could be running on physical hardware, or a virtual machine or in a container. Multiple nodes can be managed by a single entity running on the same physical hardware. Since HLF is a permissioned network, all nodes that participate have an identity provided by the membership service provider (MSP), associated with an organization. There are three types of nodes in Fabric V1.

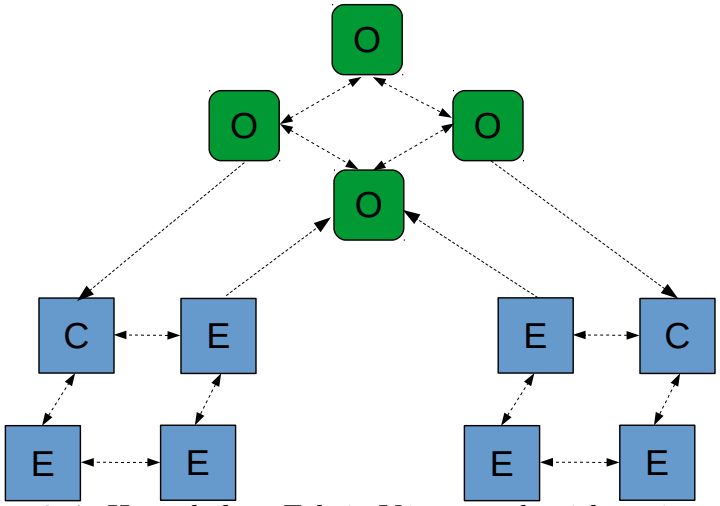


FIGURE 2.4: Hyperledger Fabric V1 network with various nodes

Peers

Peers execute transactions and maintain the distributed ledger. By default, all peers are *committers*, thereby receiving ordered state updates in the form of a block of transactions from the ordering service, and maintaining the ledger. Upon receiving a new block, the peer validates the transactions, commits the changes on the local copy of the ledger and appends onto the block on the blockchain. Peers can take up an additional responsibility of endorsing transactions, thereby called *endorsers*. An endorser simulates the transaction by executing the smart contract (SC) (called *chaincode* in HLF) and appending the results with its cryptographic signature (called endorsement) before sending it back to the client. Note that a single peer node can be both an endorser and committer.

Each organization can have multiple peers deployed, where one of the peers is configured to be an anchor peer. This anchor peer receives the blocks from the ordering service and propagates them to the other blocks via the gossip protocol.

Note that chaincode (along with its endorsement policy) is deployed on a subset of peers, and only those peers are endorsers for relevant transactions.

Orderers

The *orderers* order all the transactions in the network, propose new blocks and seek consensus. The collection of orderers form ordering service. Further details are discussed in Section 2.3.6.

Client

Clients are entities that act on behalf of an end-user. They send *transaction proposals* to peers, coordinate their execution results, verifies if the transaction execution is valid and that it satisfies the endorsement policy, and finally sends the endorsed transaction to the ordering service. A client also connects with the peer belonging to its organization to receive notifications about committed transactions. HLF currently supports client software development kits (SDKs) in Golang, Node.js, Java, and plans to support more programming languages in the future.

2.3.2 Data Structures

HLF maintains a global state across all the peers using a versioned key-value store (KVS) and the ledger, as shown in Figure 2.5.

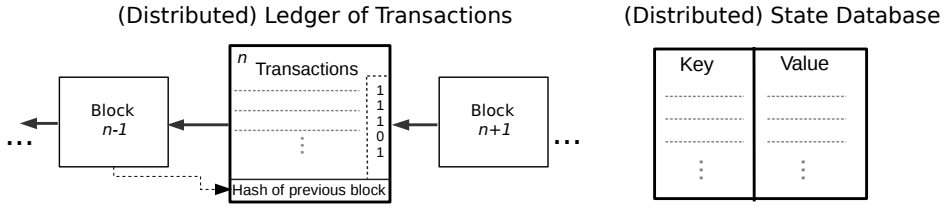


FIGURE 2.5: State database and Ledger of transactions in Hyperledger Fabric V1

Key-value store (KVS)

A versioned KVS (called Peer transaction manager (PTM) in [22]) maintains the latest system state. It stores tuple in the form (*key*, *val*, *ver*) where keys are names and values are arbitrary blobs. Fabric V1 provides two implementations for KVS:

an embedded LevelDB key-value database written in Go[2] and a client-server model database called Apache CouchDB[3]. The key-value store is read by the chaincode using `get()` operation and state updates are proposed by transactions using `put()` operation. The version numbers increase monotonically. The KVS is partitioned by the chaincode. Thus only transactions belonging to a certain chaincode may modify the keys belonging to its chaincode.

Note that the individual peers are responsible for maintaining their state and not the orderers or the chaincode or the client. In case a peer misses a block delivery from the ordering service and goes out of sync, it can request for the missing blocks using the `deliver()` call.

Ledger block store

The ledger provides a verifiable history of all the state changes (whether valid or invalid) that occurred during the operation of the system. The ordering service sends a new block of transactions with a block sequence number and the hash of the previous block. Within each block, the transactions are ordered. Thus the ledger is an ordered hash chain of blocks of (valid or invalid) transactions. The ledger imposes the total order of transactions committed across the system. Unlike other blockchain platforms, a transaction in a block does not guarantee that it is valid. Perhaps a previous transaction (included in the previous or even the same block) updated the state before the current transaction and hence invalidated the current transaction. This validity of transactions is maintained by each peer as a bitmask and recorded in the local copy of ledger as well. Note that this bitmask is not shared with other peers or the ordering service.

[2] https://github.com/syndtr/goleveldb

[3] http://couchdb.apache.org/

2.3.3 Transactions

In both Fabric v0.6 and V1, there are two types of transactions: Invoke and Query. Invoke transaction executes the specified function along with input parameters. It may involve reading and modifying the state database and return output as success or failure. Query transaction executes the specified function, which returns the peer's current state. Thus only invoke transactions modify the state of the distributed ledger.

The transactions need to be "endorsed" by a subset of peers before it can be recorded in the ledger. An endorsement is a process of simulating the transaction with the client provided inputs against the current version of the state database, and record the outcome (success/failure) along with the readset (the version of the keys read by the transaction) and writeset (updated values for the keys). The subset of peers responsible for providing the endorsement is fixed at the time of installing the specific chaincode. The client can obtain this list from its peer. A client itself is responsible for seeking endorsements on the transaction it is proposing, such that it satisfies the endorsement policy.

Endorsement Policy

The endorsement policy is a reflection of the business logic. It is expressed as monotone logical expressions that evaluate to TRUE or FALSE. For example, endorsing set `OR('Org1.member','Org2.member')` means that endorsement from any peer from Org1 or Org2 would suffice. An endorsement policy can be expressed as an arbitrary combination of AND, OR, and k/n expressions, such as `OR(Org1, Org2) AND (2/3 of Org3, Org4, Org5)`. Weights can also be attached to the organizations, and a majority of the total weight can be considered as endorsement approval. The list of endorsing peers and the policy corresponding to a chaincode can be modified only by a system administrator. Thus, endorsement policies reflect the application-level

trust model.

2.3.4 Transaction Flow

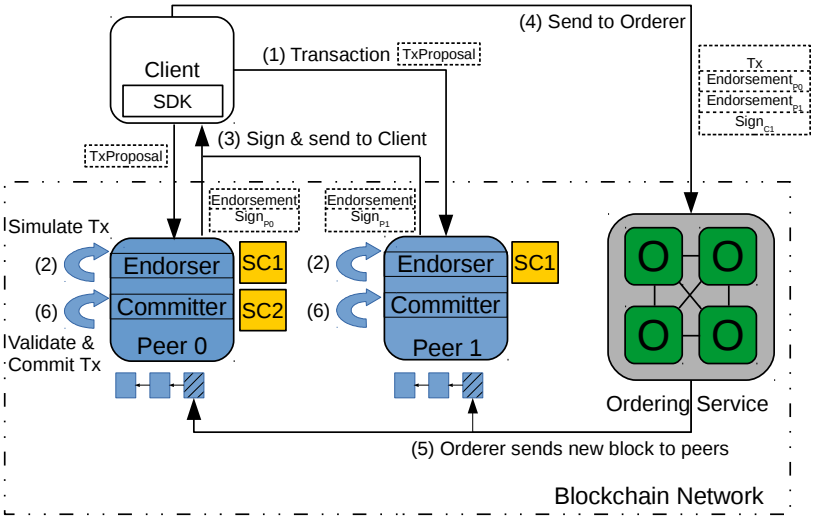


FIGURE 2.6: Transaction flow on Hyperledger Fabric V1 with 2 peers and ordering service running a single channel

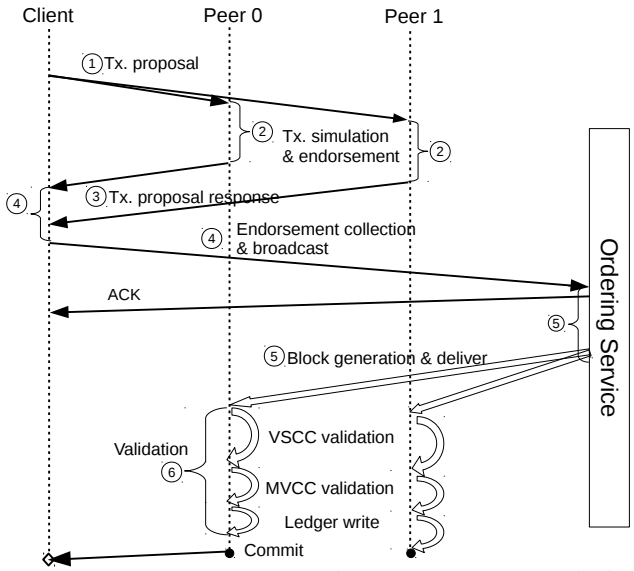


FIGURE 2.7: Transaction sequence diagram on Hyperledger Fabric V1

Let us consider a Fabric network in Figure 2.6 with the corresponding transaction sequence diagram in Figure 2.7. In this example, each peer represents their respective

organizations (say Org0 and Org1). The transaction flow is described as follows.

1. The client sends a *transaction proposal* (*TxProposal*) to the peers defined by the endorsement policy (all running the same smart contract SC1). This proposal consists of clientID, payload, and transactionID, along with a cryptographic signature of the client on the transaction header. The payload contains the chaincodeID, operation and input parameters.

2. Each peer simulates the transaction execution by invoking the corresponding SC (say SC1 in this case) with the user inputs against the local key-value state. The SC runs in a container isolated from the peer. After simulation, the endorser produces the *read-set* which represents the version numbers of keys read by SC, and *write-set* which represents the key-value pairs updated by the SC.

3. Each peer sends to the client the *endorsement* message containing the result, read-set, write-set, and metadata, where the metadata includes transactionID, endorserID, and endorser signature. This message is signed by the peer's cryptographic signature using ESCC.

4. The client waits for endorsements from peers until it satisfies the endorsement policy of the transaction and verify that the results received are consistent. The client then prepares the *transaction* containing payload and the set of endorsements received from the endorsing peers and sends it to the ordering service. This operation is asynchronous, and the client will be notified by its peer when the transaction is successfully committed.

5. After a few seconds (called *block timeout*) or after a set number of pending transactions (called *block size*), the ordering service creates a block of the pending transactions, maintaining order by timestamp. The ordering service does

not inspect the contents of the transaction. The block is appended with a cryptographic signature based on the block header and then broadcasted to all the peers on the same channel.

6. When the peer receives a block of transactions, it evaluates transaction endorsements against its endorsement policy in parallel (using Validation System Chaincode (VSCC)). The ones that fail are marked invalid. Next, for each valid transaction, it performs multi-version concurrency control (MVCC) [40, 41] (called read-write check in [22]), which means it serially verifies if the read-set version matches the current version on KVS (assuming the previous transactions are committed). The validity of transactions is captured as a bit mask and appended to the block before the block is appended on the local ledger. Finally, all the write-sets are written to the local KVS, and the state transition is thus completed. The peer notifies the client about the success or failure of the transaction.

2.3.5 Channels

So far, we described the HLF running a single blockchain network. In a large business network, a subset of peers might want to conduct business transactions privately without letting other peers know about the details of the transactions. Such transactions could be recorded on the blockchain network as well, albeit as a separate ledger. This ledger would have a separate chain of transactions as well as state database. This concept is known as a *channel* in HLF. This subset of peers would request the ordering service to create a separate channel for them and install a different chaincode capturing the business logic. Thus channels partition the state of the blockchain network. Ordering service maintains a separate order of transactions for each channel.

2.3.6 Ordering Service

The ordering service provides a shared communication channel between clients and peers. Once a transaction's endorsement policy is satisfied, the client sends the transaction payload to the ordering service, which orders the transactions into blocks and delivers them to all peers. Each block is delivered with a sequence number and hash of the previous block. Although the service could have dispatched individual transactions, it would incur high overhead due to hashing and transmission. Hence the ordering service groups transactions into blocks.

The ordering service guarantees an atomic delivery of all endorsed transactions, which means the service outputs the same message to all connected peers in the same logical order. The ordering service provides the following guarantees:

1. Safety (consistency guarantees): As long as peers are connected for sufficiently long periods of time (intermittent disconnections are ok as long as the peers restart and reconnect), the ordering service delivers the transactions in the same sequence to all peers and carry identical content for the same sequence number. The delivery also contains the cryptographic hash of the previous delivery.

2. Liveness (delivery guarantee): If the submitting client does not fail, the ordering service guarantees that each connected peers eventually receives each transaction from the ordering service.

The Fabric network uses a built-in *gossip service* to deliver blocks to a large number of peers efficiently. The ordering service delivers blocks to the anchor peer of an organization, which disseminates them further to other peers.

Note that the ordering service neither maintains any state, nor validate or executes transactions. Thus the transaction execution and validation are completely

separated in a Fabric network.

The ordering service in Fabric V1 is designed as a pluggable component. At the time of writing this thesis (latest release was v1.2), two implementations were supported: solo and Kafka-based. Solo consists of a single node with no fault tolerance and intended to be used only during the software development phase. The Kafka service is based on Apache Kafka, which is a distributed, scalable, publish-subscribe messaging system [42]. Apache Kafka works in conjunction with Apache Zookeeper [43], which coordinates between the nodes of the Kafka service, thus enabling crash fault-tolerance. Thus Kafka-based ordering service provides a scalable crash fault-tolerant service for ordering messages in the Fabric that is used in production systems.

Kafka-based Ordering Service

The Kafka-based ordering service consists of a set of Ordering Service Nodes (OSN) and a Kafka cluster with a corresponding Zookeeper ensemble [44]. The OSN nodes act as the interface for the clients. These nodes perform client authentication and relay the client transaction to the Kafka cluster so that it will be added in a future block. These nodes also allow clients to modify the configuration of an existing channel or to setup a new channel. The OSNs are also responsible for delivering blocks of transactions to the peers. In case a client has missed a previous block, it can ask the OSN node to redeliver it.

An ordering service consisting of four OSN nodes and a Kafka cluster is shown in Figure 2.8. Clients send transactions to their respective OSN node, which authenticate the client and forwards the transaction to the Topic corresponding to the channel for which the client sent the transaction. A Kafka cluster can have multiple topics, and each topic can have multiple partitions. The messages are written in a partition in an append-only manner; hence it becomes an ordered immutable sequence of

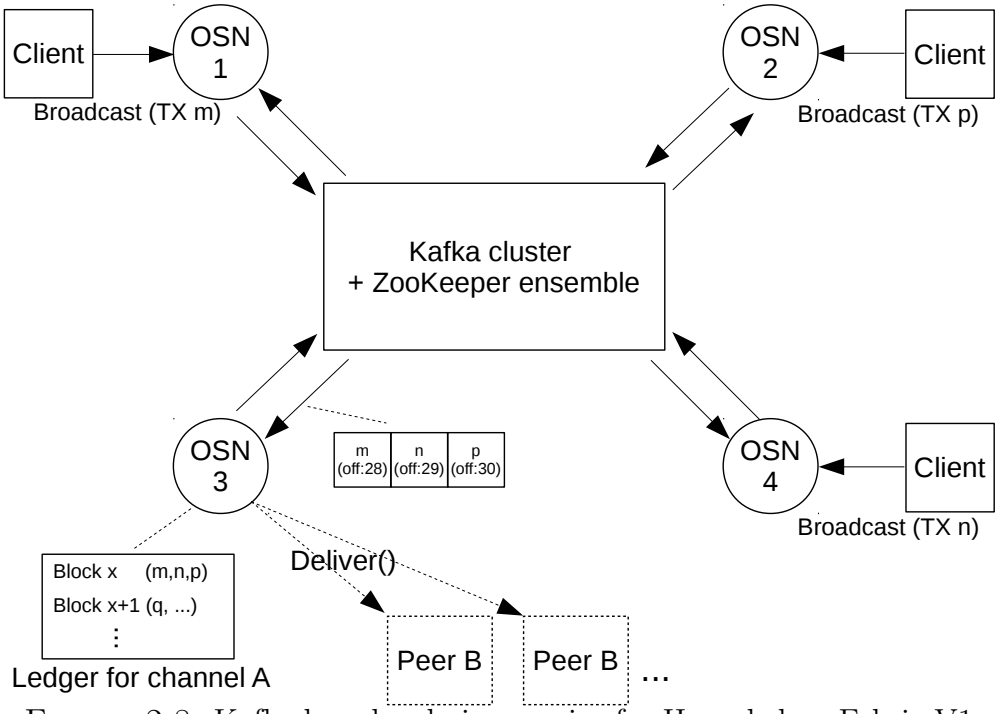


FIGURE 2.8: Kafka-based ordering service for Hyperledger Fabric V1

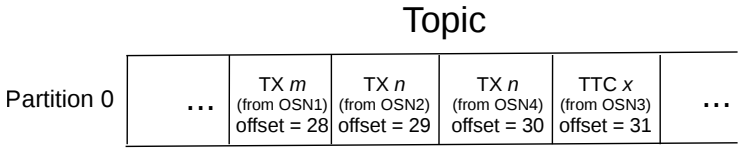


FIGURE 2.9: Transactions corresponding to a channel in a single topic/partition

messages (Figure 2.9). In HLF, each channel has a separate topic consisting of only partition 0.

A block is cut when one of the following conditions are met: a) maximum number of messages for each block (defined by *batch size*); b) passage of time (defined by *batch timeout*); c) block has reached a maximum size (defined by *PreferredMaxBytes*). Since the messages are appended, cutting a block based on count or size is straightforward. However, to cut blocks based on timeout, the OSNs need to synchronize to ensure all of them contain the same set of transactions. This is done by sending a "time to cut block X" (TTC-X) in the same partition. Multiple TTC-X transactions could be sent for the same block X by multiple OSNs. While consuming the messages

from the partition, each OSN considers only the first TTC message for a block as valid. Each OSN then records these group of messages (transactions) on its private ledger.

The block of transactions is delivered to the peers based on the offset number. Thus the requests for the block does not need to go all the way to the Kafka cluster, making it easier for the OSNs to manage it. In case a peer misses out on an arbitrary block number, the OSN can efficiently deliver the block from its private ledger without seeking the messages again from the Kafka cluster. Otherwise, the OSN nodes would have had to save the offset numbers corresponding to each block, which could slow down the delivery. The OSNs thus maintain a ledger for each channel.

3

Performance Metrics for Blockchain Networks

To evaluate the performance of a computer system, we need to define clear and crisp performance metrics. Blockchain brings unique features to the systems, that make it necessary to define the performance metrics applicable to this class of systems. First, blockchain networks are implemented without a central authority or repository. Thus the data (and transaction) is replicated across multiple peers belonging to different organizations, rather than mere replicas belonging to a single organization. It implies that the data (and transaction) is visible to various organizations at different times. Second, as opposed to individual transactions, transactions are usually committed in blocks. Each block of transaction undergoes a consensus process before the transactions are committed and finalized. Complex workflows make it challenging to define when a transaction starts and completes. HLF V1 distinguishes itself one step further, where different nodes perform unique functions to process each transaction.

Since blockchain networks are relatively new, for organizations implementing blockchain networks, their primary goal is the *performance evaluation* of a system, to ensure that it has sufficient capabilities to satisfy the application's expected use-cases. As the technology progresses and the system/application software releases are

more mature, the goal of performance evaluation will be to measure the performance improvement and degradation over releases and in different deployment configurations. Eventually, we can prepare *benchmarks* that define a standard procedure to assess the performance of a blockchain network and compare it with other systems in a standardized way.

The work presented in this chapter was done in collaboration with the Performance & Scalability Working Group (PSWG)[1] hosted by the Hyperledger Project under the Linux Foundation. The participants were researchers in the blockchain domain as well as practitioners working on the development and deployment of blockchain platforms in various Information Technology (IT) companies. At the time of writing this thesis, the first release of this document has been published [45]. In this document, my main contribution was to provide detailed examples for transaction latency and throughput based on finality condition including the figures and plots (Appendix A in [45]). I also contributed to the terminology section, provided relevant citations, and helped proof-read the document.

In this chapter, we discuss performance metrics applicable across the broad class of blockchain networks. We refine the metrics for the specific class of blockchain platforms as appropriate and provide relevant examples. Within each metric, we also share the related metrics discussed in the literature. We focus our performance metrics at a system-level, but we share examples from specific application domains where possible.

3.1 Performance Evaluation Setup

Before we get into the depth of the performance metrics, let us provide a representative setup for performance evaluation of blockchain networks, as shown in Figure 3.1.

The 'Blockchain network under test' is the collection of nodes that run the net-

[1] `https://lists.hyperledger.org/pipermail/hyperledger-perf-and-scale-wg/`

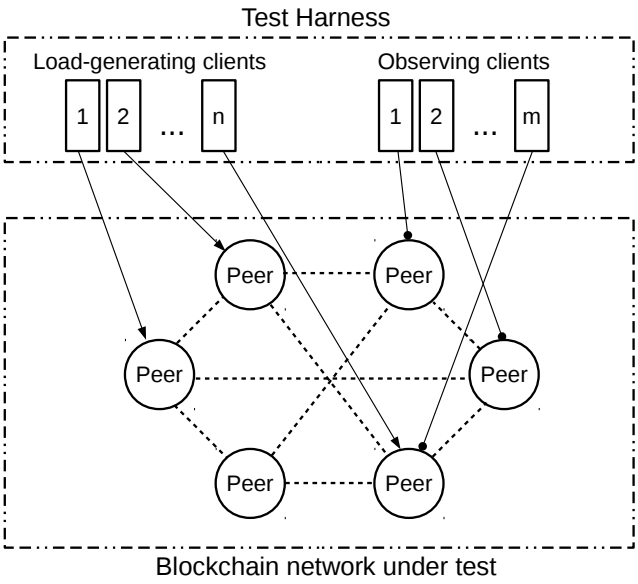


FIGURE 3.1: Representative setup for performance evaluation of a blockchain network

work. It corresponds to the 'system under test' in a performance evaluation. It includes the collection of hardware, software, network and configuration of each required to run and maintain the network [45].

The 'Test Harness' is the collection of nodes that execute the performance evaluation. These nodes are essentially clients, that can play two broad class of roles. *Load-generating clients* submit transactions on behalf of the end user. Thus they generate workload. Usually, an automated test script is used to generate workload. *Observing clients* query the peers or receive notifications regarding the status of transaction completion. The test harness also collects and analyzes the required datasets to estimate performance metrics.

The interface between the client and the blockchain network can range from a simple Representational State Transfer (REST) interface (e.g., Hyperledger Sawtooth Lake) to a comprehensive Software Development Kit (SDK) (e.g., Fabric). Thus a client can be either stateless or stateful.

Let us consider an example of a Fabric V1 network (Figure 3.2). A detailed

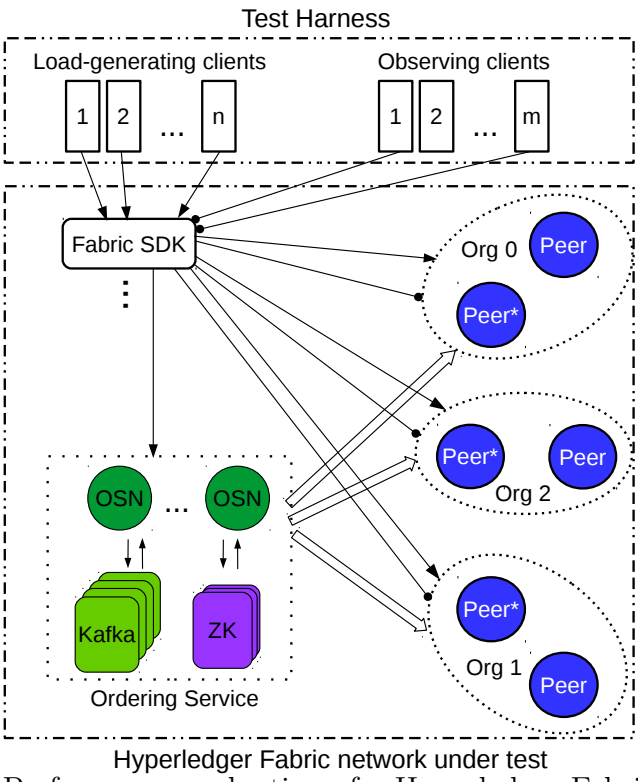


FIGURE 3.2: Performance evaluation of a Hyperledger Fabric V1 network

example is shared in Section 6.1. Due to Fabric V1's unique architecture, the SUT includes a heterogeneous class of nodes, encompassing ordering service and the peer nodes. One unique aspect of Fabric is that the SDK is deeply involved in the complex transaction flow. It raises a question whether to include it as a part of the SUT. We recommend including it in the SUT since it is considered an independent node in the Fabric architecture [22]; also different implementations of client SDK could have different performance implications. Although Caliper currently supports only Node.js SDK, it can support multiple client SDKs in the future.

The individual/organization presenting the performance evaluation results of their blockchain platform and/or application is responsible for sharing all the environment details to ensure that the results are reproducible. Appendix B lists the key aspects to consider while presenting the environment details.

3.2 Transaction Latency

Transaction latency is the time taken between when the transaction is submitted and when the transaction is confirmed committed across the network. This latency includes the propagation time and any settling time due to the consensus algorithm. In short, transaction latency is the amount of time taken for a transaction's effect to be usable across the network [45].

We need to consider two aspects:

1. Number (or %) of peers at which the transaction is observed to be settled
2. Percentile - Percentage of observations equal to or below which the measurement is valid

Our definition is motivated by the definition (ϵ, δ) consensus delay proposed by Eyal et al. in [46], which is the consensus delay at which ϵ fraction of peers agree on the state of the state machine in δ-percentile cases.

Let us consider a few examples. Suppose we evaluate the performance of a four peer blockchain network using Hyperledger Caliper (or similar tool) which observes measurements only at one peer, and reports average transaction latency as 5 sec. Then we would report our transaction latency as 5 sec. @ (1/4, average).

With a more sophisticated performance evaluation tool, let's say we measure the performance at 3 out of 4 peers, so we can consider our transaction confirmation time as the time when a transaction is complete at 3 out of 4 peers. If the tool reports 90th percentile transaction latency as 8 sec, then we would report our transaction latency as 8 sec @ (3/4, 90th percentile).

Transaction latency is most popularly reported as average latency, which is calculated as follows:

Average Transaction Latency = $\sum$ transaction latency / Total committed transactions

For a blockchain platform under test, the presenter of the results must also clearly define "when" the transaction is considered complete. E.g., Caliper considers a Fabric transaction complete when it receives an event notification from the peer. However, if we were to gather this information from logs, we would consider it complete when the write-set changes are written to the key-value database.

It is worth pointing out that committed transactions should only include valid transactions. Note that some platforms such as Fabric record all the transactions that were included in a block by the ordering service, even though they might be rejected later in the validation phase. Section 3.4 elaborates on the reasons for transaction failures in permissioned blockchain networks.

Let us refine the metric further for three classes of blockchain networks based on the finality condition of the consensus protocol and the network topology.

1. Case 1: Immediate finality

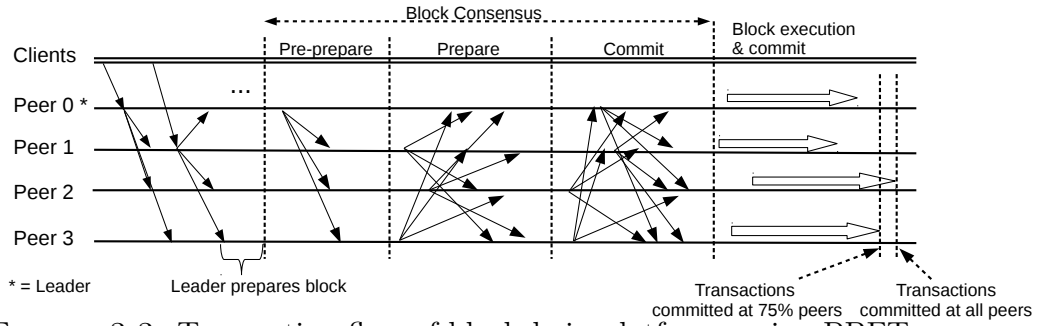


FIGURE 3.3: Transaction flow of blockchain platforms using PBFT consensus

Systems employing voting-based consensus have immediate finality. Once a transaction is committed, the state is guaranteed to be irrevocable. However, the transactions would commit at different peers at different times due to two reasons. First, different peers would receive blocks of transactions at different times due to block propagation time. Second, each peer would execute and

commit transactions at a different rate based on its performance capabilities and how busy it is. We visualize this in Figure 3.3. As described in the main text, the transaction confirmation time can be defined based on the number (or %) of peers at which the transaction confirmation is observed.

2. Case 2: Probabilistic finality and known network topology

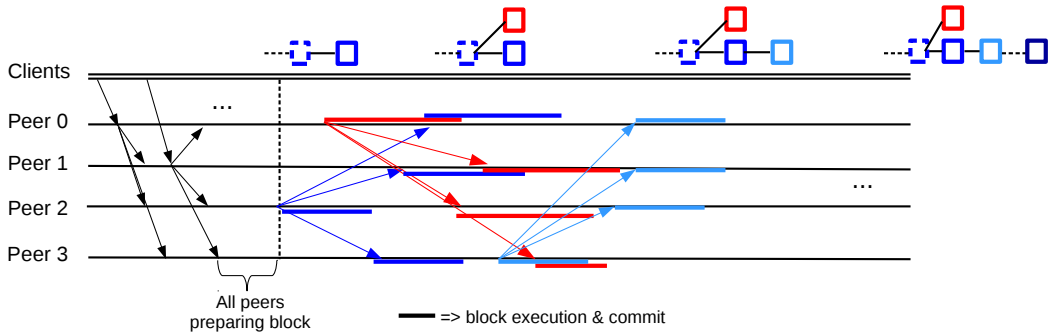


FIGURE 3.4: Transaction flow of blockchain platforms using lottery-based consensus

Systems using lottery-based consensus have probabilistic finality. In most permissioned deployments, the topology will be known. In this case, a transaction is confirmed only when multiple nodes reach the same state, visualized in Figure 3.4. Thus we can use the metric defined above.

3. Case 3: Probabilistic finality and unknown network topology

Some of the prominent examples are public blockchain networks such as Bitcoin and Ethereum that run in an untrusted environment with anonymous miners. A client has access only to limited peers. In this case, a transaction is considered final and irrevocable only when enough blocks have appended the chain. For Bitcoin and Ethereum, it is considered to be 6 and 12 respectively [2, 47]. However, this figure needs to be analyzed for the blockchain network under test. It can be estimated by recording the block in which each transaction shows up and doing a percentile analysis, so we can estimate the percentile of

transactions committed up to a certain block, as shown in Figure 3.5 (Ref. [48]). An organization can use this metric for risk assessment of their transactions.

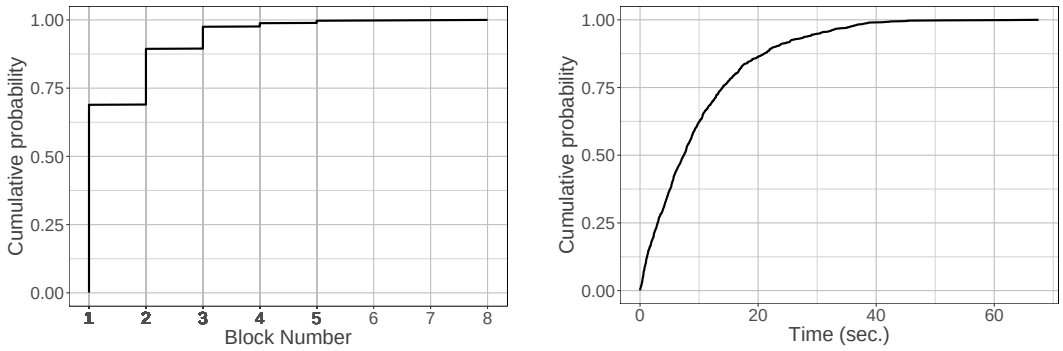


FIGURE 3.5: Transaction confirmation probability for blockchain network with unknown topology

For such networks, the latency can be reported as follows. For example, a transaction submitted at the time when block x is the highest, can be found in block $(x + 2)$ with 50th percentile chance, in block $(x + 5)$ with 90th percentile chance, in block (x + 7) with 99th percentile chance. Thus the transaction latency is 2 blocks @ 50th percentile, 5 blocks @ 90th percentile, and 7 blocks @ 99th percentile respectively. Since the finality is measured in terms of the number of appended blocks, it is more intuitive to present latency in terms of the number of blocks rather than time.

How you choose to report your results depends mainly on your application and its domain. If you are building a blockchain network for latency-sensitive IoT application, you might want to consider reporting latency at a higher percentile (say 90). If your application is in banking/finance domain where the transaction needs to be confirmed at a high percentage of nodes before its considered complete, you might want to take measurements at a large set of peers.

Related definitions

One relevant definition in the literature is the X% block propagation delay [21], which is defined as the time to deliver blocks to X% of the peers (say 50% or 90%). This is a critical metric for large overlay networks such as Bitcoin. To summarize the observation from a collection of transactions, this metric implicitly assumes a summary statistic such as percentile or average. Note that this metric only refers to the latency in block propagation, which is a significant component of the transaction latency. We consider this a useful metric to describe the blockchain network environment (Ref. Appendix B).

In some blockchain platforms such as Fabric, it is useful to measure the latency by each transaction phase. As proposed in [22, 41], we can divide transaction latency into Endorsement, Ordering, and Validation phases. Within validation, we can split the block latency into VSCC validation, MVCC validation, and ledger update. For example, please refer to Section 6.6.

3.3 Transaction Throughput

Transaction throughput is the rate at which the blockchain network commits valid transactions in the defined period of time [45]. Note that this is the rate across the network, which means the discussion regarding the number (or %) of peers across which it is observed to be committed is still applicable. Thus,

Transaction throughput := Total committed transactions / Total time @ (% of committed nodes)

Throughput is commonly expressed in transactions per second (TPS). Unlike latency, we do not need a summary statistic like percentile or average, since throughput intuitively acts as a summary statistic itself.

As discussed earlier, we should consider only valid transactions in our measure-

ments. Thus the throughput is the same as goodput.

Related definitions

For blockchain systems, all the transactions are committed in "blocks", hence throughput can be considered as (block size) / (block interval), where block size could be no. of transactions or in size (kB or MB), and block interval is the frequency of generating blocks ($\tilde{1}$0 mins in case of Bitcoin). Since Bitcoin operates on a vast overlay network, peers receive blocks in a wide range of propagation times. To account for the % of peers receiving block by a particular block interval, Croman et al. [21] propose the metric X% Effective Throughput.

X% Effective Throughput := (block size) / (X% block propagation delay)

where (X% block propagation delay) is as defined earlier.

Let us consider an example. For the Bitcoin network, the average block size is 540KB and the median block propagation time is 8.7 sec. Thus X% effective throughput is 62 kBps or 496 kbps [21].

This metric intuitively provides the blockchain network's capacity as a function of the network topology, which is useful for planning the average block size and frequency to maximize the network throughput. Note that this metric implicitly assumes that the block frequency is fixed, which is a valid assumption for large overlay networks running lottery-based consensus such as Bitcoin and Ethereum. However, for networks running voting-based consensus, the block frequency changes based on the arrival rate, block size, and timeout. For such networks, this metric would be fixed across different arrival rates, and hence it is not as insightful.

3.4 Scalability & Elasticity

Just like any new computer system, stakeholders of blockchain networks are concerned whether the system would be able to handle high transaction throughput,

maintain low transaction latency with increasing workload and increasing number of peers [49]. Such concerns nessessitates the need to understand the "scalability" of blockchain networks.

Scalability connotes the ability of a system to accommodate growing volume of work gracefully and/or to accommodate enlargement [50]. Scalability is a desirable attribute of computer systems and networks. Since there is no formal definition of scalability [51, 50, 52], this term is generally misused, which creates confusion. The presenters of performance analysis results should refrain from calling their system/application as "scalable". Rather, it is useful to present a detailed *performance and scalability analysis study* and let the readers decide if the system is useful for their current and future business use-cases.

In the context of system performance, we define *scalability analysis* as the process of analyzing system performance by scaling key dimensions. In the context of computer systems, dimension denotes key aspects of the system and applications whose scaling affects system behavior. For distributed ledger systems, important dimensions include the number of peers, the arrival rate of transactions, peer hardware and software capabilities. Thus, scalability is related to a system's ability to accommodate the "scaling" of some dimension [52].

Scalability analysis provides useful insights to all the stakeholders of the system. To the researchers and developers of the system, it provides useful insights into the inner workings of their system and encourages them to design and implement solutions that are considered unfeasible today [51]. To the engineers deploying the systems, it presents the opportunities and challenges involved in expanding the system for future use-cases.

To understand the cause-effect relationship between the system dimension and measured output metric, let us consider the scalability framework in Figure 3.6 (adopted from [52]).

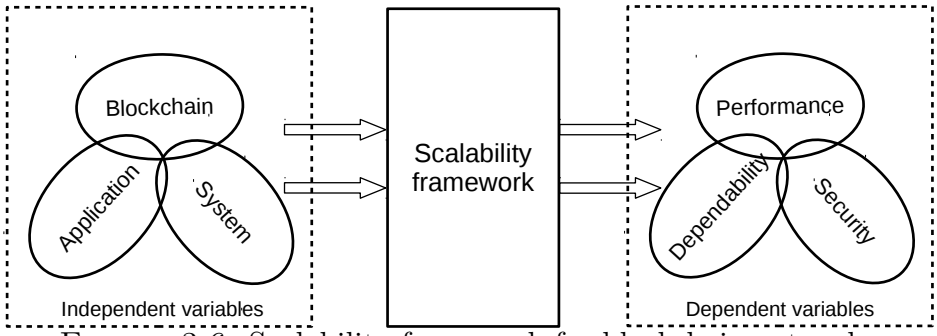


FIGURE 3.6: Scalability framework for blockchain networks

Cause-effect relation can be studied using independent and dependent variables. *Independent variables* represent the subset of variables that affect the system behavior. For blockchain networks, we can classify them into blockchain, system and application domain. For example, blockchain domain includes the number of peers, block size, block frequency, consensus protocol, and placement of peers; system domain includes hardware and software capabilities and configuration of the peers; application domain includes number of smart contracts, size/complexity/memory footprint of smart contracts. These domains are not mutually exclusive.

Dependent variables represent the qualitative and quantitative output metrics that are affected by the independent variables. Depending on the metrics for which the system needs to be optimized, they can also be classified as critical or flexible [52]. However, we prefer to classify them as performance, dependability [53], and security. For example, performance metrics includes throughput and latency; dependability includes availability, reliability, performability [28]; security includes stale block rate [54, 55].

Scalability analysis is conducted by performance evaluation of system by varying one or more independent variables (dimensions) and measuring the dependent variables. Ideally, the dimensions are varied between each performance test run. However, some of the dimensions (like traffic arrival rate) can be varied during a test run. Such analysis is called *elasticity*, which can be defined as the ability of

application to be scaled at runtime [56]. For elasticity analysis, "scaling speed" can be measured, which signifies the time taken to scale at runtime [56]. Note that not all dimensions can be scaled at runtime. In summary, the key differentiator between scalability and elasticity is whether the dimensions change in between the runs or during the runs.

Scalability (and elasticity) analysis results are usually presented as a 2D graph, where the x-axis represents the range of an independent variable, and y-axis represents the value of a dependent variable. If we find a linear relationship between the dependent and independent variable, it is tempting to present its slope as a single numerical result. However, it might not be applicable in all cases.

Related Work

Let us discuss research papers on scalability analysis of HLF. For HLF v0.6, Dinh et al. [57] studied the changes in throughput and latency with increasing number of peers and increasing workload. They found that the system fails to scale beyond 16 peers. On further investigation, they found the consensus messages were getting choked with other messages in the same message queue, resulting in peers getting stuck during the view-change process of PBFT. Thus scalability analysis highlighted the poor protocol implementation.

For HLF V1, Baliga et al. [58] measured the transaction throughput and latency by varying the number of chaincodes, number of channels, and number of organizations per channel (4 to 16). Thakker et al. [41] also measured the transaction throughput and latency by varying the transaction arrival rate, number of channels, number of CPU cores per peer, and the transaction complexity (number of read-writes in chaincode). They propose additional ideas for scalability analysis, viz. varying the number of organizations per channel and number of peers per org. Regarding varying the number of peers/orgs, they observed that endorsement policy

plays a critical role. In addition to some of the above dimensions, Androulaki et al. [22] varied the geographic locations of peers from single data-center to multiple data centers over a WAN. Overall, HLF V1 is found to scale well with the number of peers/organizations, channels, and chaincode at a reasonable workload.

Appendix - Transaction Failure

Since different blockchain platforms handle consensus and transaction validation differently, it is challenging to align error classes across platforms. Following are the broad class of reasons why blockchain transactions get rejected [14]:

- Consensus errors
  - Validation logic (VSCC in the case of Hyperledger Fabric)
  - Policy failure (endorsement policy not satisfied in the case of Hyperledger Fabric)
- Syntax errors
  - Invalid input (smart contract id, unmarshalling errors, and so on)
  - Unverifiable client or endorsement signature
  - Repeated transaction (due to error or replay attack)
- Version errors
  - By version control (readset version mismatch, writeset is unwritable)

4

Empiricial Analysis for Hyperledger Fabric v0.6

In this chapter, we provide extensive details of the empirical analysis presented in our published work [29], where we analyze the block consensus process for HLF v0.6. We extend our work by providing empirical analysis for the block execution process as well.

4.1 Experimental Setup

We setup the blockchain network using the IBM's Bluemix service[1]. This Platform-as-a-Service (PaaS) hosted by IBM provides an ability to deploy blockchain service with four peers in an automated manner. Following are our system configuration details.

1. Number of validating peers (VP) (n) = 4
2. Maximum number of failing peers (f) = 1
3. PBFT view change timeout = 30 seconds
4. PBFT broadcast timeout = 1 second

[1] `https://console.ng.bluemix.net/catalog/services/blockchain/`

5. PBFT automatic view change = disabled
6. PBFT batch size = 1000
7. PBFT batch timeout = 1 second

Test Application

We deployed the 'track and trace' contract [59] on an IoT Contract Platform [60], both developed by the IBM Watson IoT team. This application tracks and traces an IoT device (an asset on the blockchain) deployed in the field. An example is a surgical kit that needs to be tracked from creation through its use in an hospital. Its location is recorded at regular intervals and its possesser is recorded upon each handover. In case the kit is moved outside the perimeter of the hospital campus, an alert is generated for the security team. In our deployment, the most frequently occurring transaction is *updateAssetSurgicalKit* that reads the existing state from the world state, merges the current state to create a new state, and then execute rules to raise or clear alerts. Then it stores the new state (along with past few state updates) to the world state. The transactions are generated by a node-red device that sends bursts of events to the IoT platform, which maps it to the appropriate track and trace contract.

4.2 Analysis for PBFT consensus process

4.2.1 Measurements

To measure the time taken to perform the actions captured in our model, we analyze the log files generated by the peer. After reviewing the log files and the correponding software source code, we identify the log entries correponding to the key events. A set of log file entries corresponding to the consensus process between two validating peers is shown in Figure 4.1, where the validating peer on the left (vp0) is a leader. Timestamp corresponding to each entry is captured with a nanosecond accuracy but

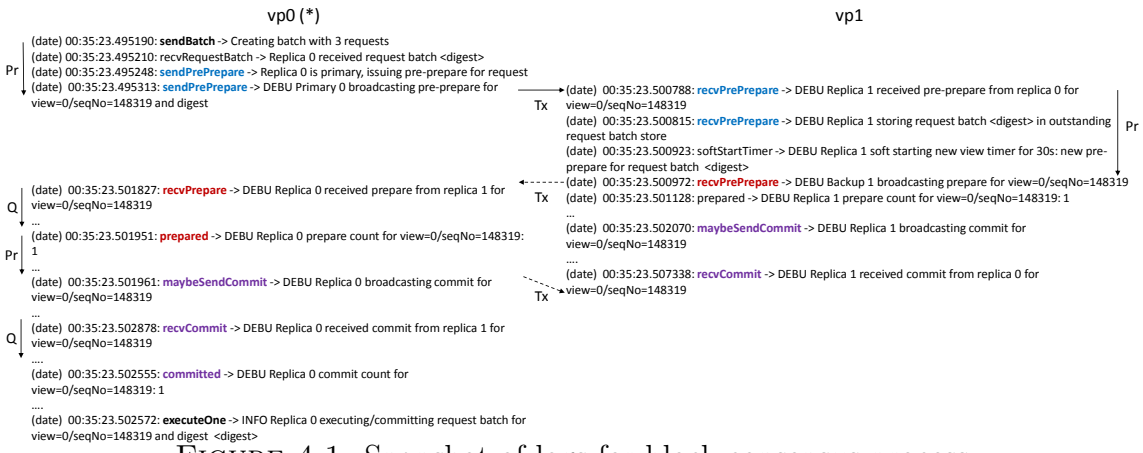


FIGURE 4.1: Snapshot of logs for block consensus process

shown in microsecond accuracy for brevity. Logs are trimmed for compactness. For a system running over few weeks with around 800 blocks committed per hour, we analyze the logs for 50 randomly chosen blocks.

4.2.2 Model parameterization

Our goal is to find the best-fit distribution for the firing time of each SRN transition in our model in Figure 5.2. Each SRN transition corresponds to a key time-consuming operation in the system (discussed in Section 5.2). We collect the datasets corresponding to each operation. We consider the following distributions for our dataset: Exponential, Weibull, Gamma, Hypoexponential (2-stage, 3-stage), LogNormal and Pareto. We estimate the parameter values for each fitted distribution using Maximum Likelihood Estimate (MLE) technique. We evaluate the goodness-of-fit using Kolmogorov-Smirnov (KS) statistic [61] at 5% level for significance. Finally, we select the distribution with the lowest Akaike information criteria (AIC) [62]. For reference, we share our R code for probability distribution fitting in Appendix C.

Let us discuss the measurements for each set of model parameters.

*Time to prepare consensus message for next stage (*Op*)*

We measure this at each stage of the consensus process (as shown in Figure 4.1). For our detailed model, we can estimate parameters for transitions corresponding to each stage separately. However, we perform Analysis of Variance (ANOVA) F-test [63] to see if there is a statistically significant difference in means between the three groups corresponding to the three set of samples. We find that the null hypothesis for equal means can be rejected (p-value = 6.3e-9). Then we perform multiple comparisons procedure using Tukey's Honest Significance Test [63] and find that the time to process a message in the commit phase is much smalller than that for the other two phases. There is no statistically significant difference between the samples corresponding to the pre-prepare and prepare phase. Hence we combine the corresponding datasets and analyze them as a set, finding that Weibull distribution is the best fit, followed by Gamma and 2-stage Hypoexponential. For *Op* in Commit phase, we find 2-stage Hypoexponential is the best fit, followed by 3-stage Hypoexponential and Gamma.

*Time to transmit a message (*Tx*)*

Since the messages could be subjected to queuing delays, measuring the time to transmit a message (*Tx*) is more challenging. Since our IoT application generates a burst of transactions every few seconds, a block gets created with every burst and completes within the same second. Thus, when the leader sends the pre-prepare message for a new block, there are no pending consensus messages at any VP from the previous blocks. For each block analyzed, we measure the time to transmit this pre-prepare message as a sample for *Tx*. For messages during the other phases, we consider them as a valid sample only if they were the first message to reach the peer. We perform the ANOVA F-test followed by Tukey's Honest Significance Test and find no statistically significant difference between the set of samples. We find that Weibull

is the best fit, followed by Gamma and 3-stage Hypoexponential. Given our limited understanding of the size of the messages exchanged, the time to transmit measured corresponds to the end-to-end delay to transmit message between two peers. Time to transmit smaller messages is more sensitive to latency than throughput, and the other way round for larger messages [21]. With sufficient understanding, the transitions with subscript Tx can be used to model the time to transmission in either case.

*Time to process incoming consensus message (*Ip*)*

In our analysis, we consider time to process incoming consensus messages (subscript Ip) in the prepare and commit phase as the time between between receipt of the message and adding them to the prepare/commit count (Figure 4.1). From the log files ordered by timestamp, we see that such messages are processed serially. We find that Weibull distribution is the best fit, followed by Gamma and 2-stage Hypoexponential.

For all the metrics discussed above, we assume the same distribution across all VPs. This assumption holds well in our blockchain setup on IBM Bluemix, since each VP is running in a Docker container with the same configuration and are co-located in the same rack. Table 4.1 provides summary statistics for the datasets discussed above and time to complete consensus. The empirical and fitted cumulative distribution functions (CDF) for Tx and Pr (preprepare and prepare stages only) are shown in Figure 4.2, skipping the distributions that were rejected by the KS test criteria.

4.3 Analysis for Block Execution process

4.3.1 Measurements

A snapshot of the log entries corresponding to the block execution process at one VP is shown in Figure 4.3. Once the consensus process is complete, the block execution starts (*executeOne*). After a few small steps, the VP executes each transaction is

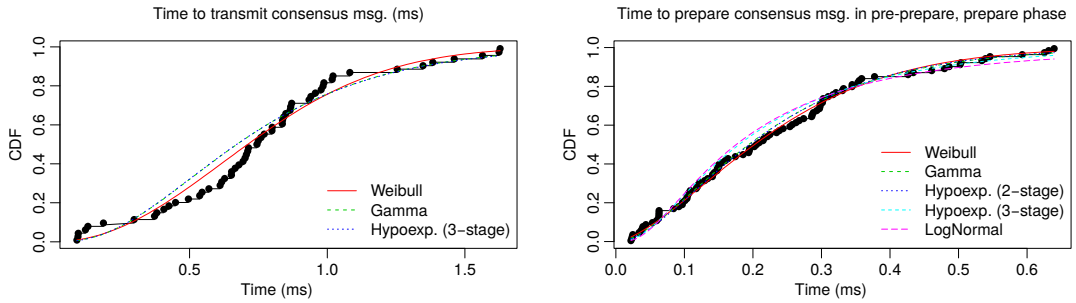


FIGURE 4.2: Empirical, fitted cumulative distribution function (CDF) for time to transmit and process consensus messages

Table 4.1: Summary statistics for datasets corresponding to consensus process

Statistic	Transmit (Tx) (ms)	Prepare Consensus (Op) (pre-prepare, prepare stages) (ms)	Incoming process (Ip) (ms)	Complete (ms)
Minimum	0.0906	0.0219	0.0126	1.706
1st Quartile	0.5453	0.1068	0.0618	2.823
Median	0.7446	0.2050	0.1009	3.314
Mean	0.7541	0.2321	0.1116	3.492
Std. Dev	0.3730	0.1571	0.0748	1.231
3rd Quartile	0.9403	0.3097	0.1357	3.545
Maximum	1.6280	0.6394	0.3366	8.690

series, by invoking the corresponding chaincode in a dedicated docker container. Then, the VP needs to compute the crypto-hash of the world state's root node. The crypto-hash of the changed leaf buckets is computed, followed by the next level of nodes where the leaf nodes were changed, and so on, till the root node's crypto-hash is computed. Since multiple transactions could have updated the same leaf node, the total number of buckets updated depends sublinearly on the no. of transactions executed. From the timestamps, note that computing the crypto-hash of each bucket takes a noticable time. The time to complete block execution is taken as the time between entries *executeOne* and *execDoneSync*.

Before we parameterize our model, we verified that there is no statistically significant difference in the block execution times across the four validating peers. Thus, the results from our model captures the block execution process at any peer.

FIGURE 4.3: Snapshot of logs for block execution process

4.3.2 Model parameterization

We analyze the logs for 50 randomly chosen blocks and estimate the model parameters followingthe same producedure as described in the earlier subsection.

For transaction execution, we find that Gamma is the only distribution that fits. For crypto-hash of the world state, we find Erlang (3-stage) is the best fit, followed by Hypoexponential (3-stage) and Gamma distribution. Summary statistics for the datasets is shown in Table 4.2 and the empirical and fitted cumulative distribution functions (CDF) are shown in Figure 4.4.

As we can see, the time to block execution is an order of magnitude larger than

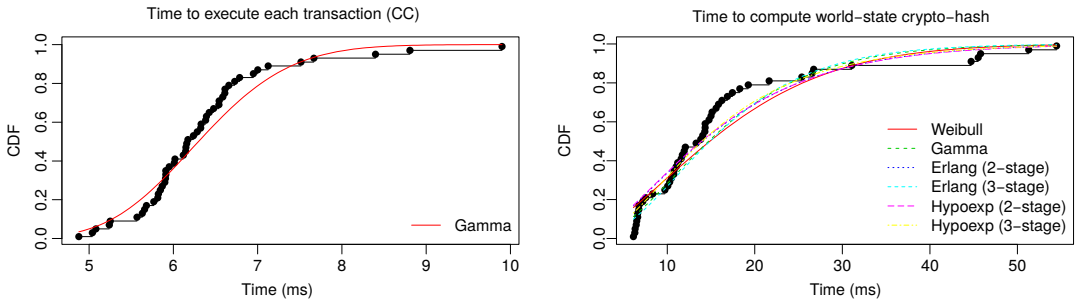


FIGURE 4.4: Empirical, fitted cumulative distribution function (CDF) for time to transaction execution and crypto-hash of world state

Table 4.2: Summary statistics for datasets corresponding to block execution process

Statistic	Transaction execution (ms)	Crypto-hash world state (ms)	Total block execution
Minimum	4.879	6.139	13.23
1st Quartile	5.852	9.810	22.56
Median	6.166	13.557	29.68
Mean	6.336	16.696	34.33
Std. Dev	0.912	12.150	19.711
3rd Quartile	6.604	17.299	42.81
Maximum	9.899	54.464	100.98

the time to consensus. For an average block size of three, it is unexpectedly high, and very likely the performance bottleneck of the system.

4.4 Related Work

In this section, we discuss related papers that provide empirical analysis for v0.6 of HLF. Empirical analysis for Fabric V1 and other blockchain networks is discussed in Section 6.8.

Auh Dinh et al. [57] presents the performance evaluation of HLF v0.6, Ethereum, and Parity, all in a private setting. For performance measurements at the application level, they developed macro benchmarks based on the YCSB and Smallbank benchmarks. Across all benchmarks, they measure the following application-level performance metrics: throughput (same as goodput), latency, scalability (changes in metrics with node failures, similar to performability), fault tolerance (change in

metrics with node failures), and security (stale block rate). They found HLF v0.6 had higher throughput and lower latency compared to other platforms for four peers; however, the performance degraded significantly with increasing peers. Upon further investigation at the consensus layer for a network of 16 peers, they found the consensus messages were getting choked along with the client and gossip messages, and hence peers were not able to complete the consensus in time.

Pongnumkal et al. [64] also evaluated the performance of HLF v0.6 and Etherum (private). Rather than running the workload at a steady rate for a set duration, their workload is a batch of requests of size ranging from 10 to 10k. Also, they considered only one node per network, by disabling the consensus mechanism, thus exercising just the transaction execution mechanism of the platforms. They measured average transaction latency and throughput and total execution time of the batch. Given the limitations of their experimental setup, we refrain from further comments on their experimental results.

5

Performance modeling of Hyperledger Fabric v0.6

In this chapter, we present our performance models for Fabric v0.6. We primarily focus on our model of the Practical Byzantine Fault Tolerance (PBFT) consensus process, which is a critical application-generic performance aspect of the blockchain. We parameterize the model using measurements from a production-grade Internet-of-Things (IoT) application running on the IBM Bluemix service running with four validating peers, and validate our model output with experimental results. Then, we evaluate the mean time to complete consensus for networks up to 100 peers. For the default configuration of four peers, we also perform sensitivity analysis with various system parameters. This chapter is an extended version of our published work [29]. For completeness, we also present a model of the block execution process.

Some of the text, figures, tables in this chapter are reprinted, with permission, from "H. Sukhwani, José M. Martínez, Xiaolin Chang, Kishor S. Trivedi, and Andy Rindos. Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255, Sept. 2017."

The main contributions of this research are as follows:

1. Scalable model of PBFT consensus process for Fabric v0.6, validated using data collected from real-world setup.
2. Evaluation of consensus process for a large number of peers (up to 100) and sensitivity analysis with respect to key parameters.

5.1 PBFT consensus process for Fabric v0.6

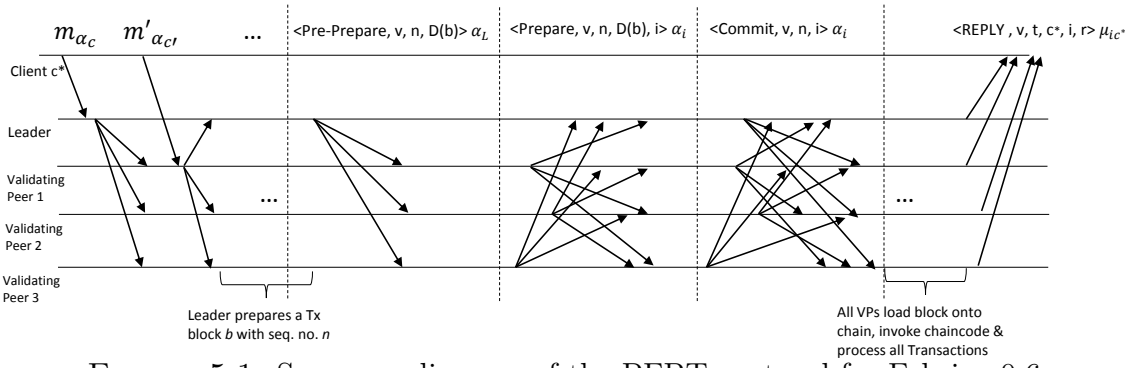


FIGURE 5.1: Sequence diagram of the PFBT protocol for Fabric v0.6

PBFT protocol provides a solution to the Byzantine generals problem to work in asynchronous environments like the internet. PBFT works on the assumption that less than one-third of the peers are faulty (f), which means that the network should consist of at least $n = 3f + 1$ peers to tolerate f faulty peers [11]. Thus $f = \lfloor (n-1)/3 \rfloor$. If $2f + 1$ peers agree on the block of transactions, then each VP executes all the transactions and appends the block as the next block on their private ledger. Thus, due to the consensus, each VP possesses a ledger with the same sequence of blocks.

Let us provide an overview of the consensus process using PBFT protocol. Extensive details of the PBFT protocol are discussed in [11], from where we borrow some of the terminology for this subsection. For explaination in this subsection, we assume symmetric-key encryption such as Message Authentication Code (MAC) for encrypting messages between peers. Note that HLF lets us choose any message

encryption technique, and the choice of the encryption scheme does not affect our analysis.

A sequence diagram of the PBFT consensus process is shown in Figure 5.1. Client c sends *Request* message to its corresponding validating peer at time t with operation o, appended with MAC α_c. After many such request messages, the leader creates a block with sequence number n and starts the *pre-prepare* phase by sending a *Pre-Prepare* message to all the VPs with view id v, and a digest (cryptographic hash) of block message $D(m)$, appended with the MAC for each VP. Each VP receives the pre-prepare message, checks the integrity of the leader by verifying the sequence number and other checks. It also verifies if the message is not a duplicate of a previous message, before adding the tuple (n, v, m) to the database Q. For the *prepare* phase, each non-leader VP sends a *Prepare* message to all other VPs with identity i. Each VP waits for $2f + 1$ agreement messages (including the leader's pre-prepare and its own consensus) to get the majority consensus. Then each VP saves the message tuple to database P. For the *commit* phase, each VP sends a *Commit* message to all other VPs with the same format as *Prepare* message. After receiving *Commit* messages from $2f$ other peers, each VP saves the commit certificate for that block. It waits if there is a missing sequence number between the previously committed message; otherwise, it adds this block to the chain. Then, each VP processes all the transactions inside the block, and sends *Reply* message back to the respective client c with view v, time t, reply output r, and appends it with a MAC code μ_{ic} defined between the client c and VP i. Each client waits for the weak certificate, which means it assumes the transaction is done if it receives $f + 1$ messages, which is one more than the maximum number of peers that can fail.

To prevent a faulty leader from affecting the entire consensus protocol, the VPs are responsible for detecting any misbehavior of the leader, like sending a pre-prepare message with an invalid sequence number or invalid MAC [11]. In such cases, the

VP can issue a view-change protocol, which initiates a consensus between the working VPs to elect a new leader. Later when the ex-leader is back on the network, it can sync up with another VP to come up to speed with the blockchain, using synchronization messages in case of Fabric [35].

In summary, PBFT is a state machine replication technique that supports two properties, viz. safety and liveness, assuming that at most $\lfloor (n-1)/3 \rfloor$ out of n VPs are faulty within a small window of vulnerability [11]. Safety means all VPs would execute the transactions in the same order. Liveness means that clients eventually receive replies to their requests. Further details of the PBFT protocol can be found in [11]. In contrast with the protocol described in [11], where the message under consensus is a message sent by a client itself, in Hyperledger, the message under consensus is a block of transactions sent by the leader. A major concern here is the large number of messages exchanged between the VPs to obtain this consensus.

5.2 Performance model of PBFT consensus process

In this section, we model the PBFT consensus process using SRNs. We are interested in computing the "mean time to complete consensus" for a block using the three-phase protocol. In our model (Figure 5.2), we capture the important steps in the protocol that consume noticeable time, viz. transmission time of consensus messages between peers (transitions with subscript Tx), time to process incoming consensus message (transitions with subscript Ip), and time to prepare consensus message for next stage (transitions with subscript Op).

We start with the following set of assumptions (many of these assumptions can be relaxed if needed):

1. A leader peer is already chosen before the block transaction starts, and it does not change during the execution of the three-phase protocol for a single block.

2. VPs do not fail at any time during the execution of the three-phase protocol for a single block.

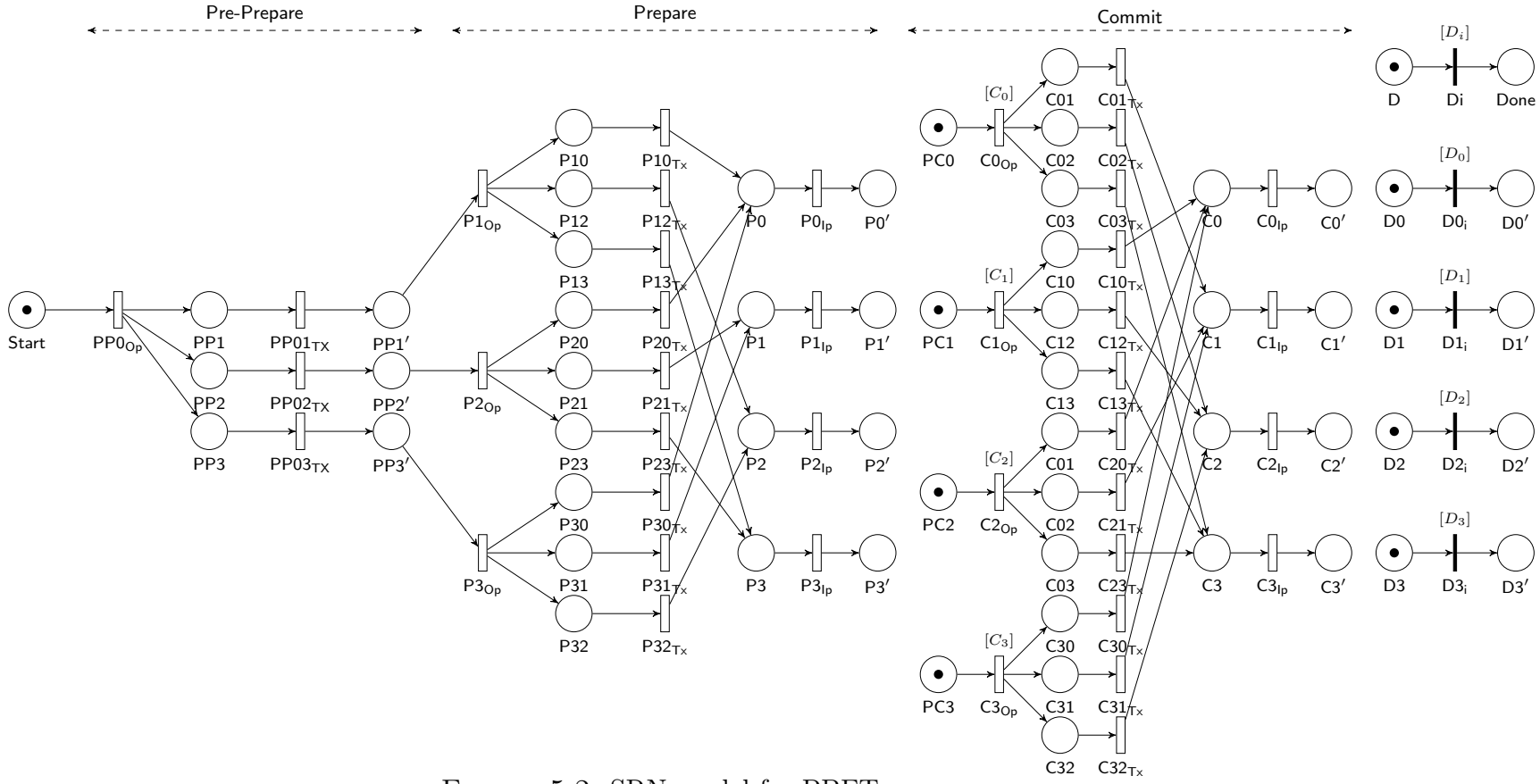


FIGURE 5.2: SRN model for PBFT consensus process

The SRN model in Figure 5.2 shows the details of the block consensus for a system with four VPs. It starts from a token in place *Start*, signifying that the leader is ready with the new proposed block, and ends with a token in place *Done*, signifying that the consensus process is complete. Hereafter, we identify the leader with the number 0 and the other VPs with the numbers 1, 2, and 3 respectively. In the *pre-prepare* phase, the leader prepares a block, indicated by firing of transition $PP0_{\mathrm{Op}}$ and starts to transmit it to the three VPs, indicated by a token each in places *PP1*, *PP2*, *PP3*. After a transmission delay, denoted by transitions $PP01_{\mathrm{Tx}}$, $PP02_{\mathrm{Tx}}$, $PP03_{\mathrm{Tx}}$ respectively, the message reaches the three VPs, denoted by a token each in places *PP1′*, *PP2′*, *PP3′*. This completes the pre-prepare phase of the protocol.

In the *prepare* phase, each VP processes and prepares the "prepare" message, denoted by transitions $P1_{\mathrm{Op}}$, $P2_{\mathrm{Op}}$, and $P3_{\mathrm{Op}}$ respectively. When those transitions fire, say $P1_{\mathrm{Op}}$, it deposits tokens in three places, say *P10*, *P12*, and *P13*, which are followed by a transmission delay, denoted by transitions $P10_{\mathrm{Tx}}$, $P12_{\mathrm{Tx}}$, and $P13_{\mathrm{Tx}}$ before the messages reach three other VPs. The messages are received by the respective VP and processed serially, denoted by transitions $\mathrm{P0_{Ip}}$ to $\mathrm{P3_{Ip}}$. When a leader receives $2f$ messages from other VPs or non-leader VPs receives $2f-1$ messages from other VPs (since it has its own and leader's consensus), indicated by tokens in places P0′ to P3′, the VP enters the *commit* phase by enabling the guard function, say $[\mathrm{C_0}]$ for transition $\mathrm{C0_{Pr}}$, to start preparing a "commit" message. Rest of the process is similar to that in the prepare phase. Eventually, multiple tokens are deposited in each place, say C0′, corresponding to the commit messages received by for VP0. The commit phase is complete when each VP receives at least $2f$ commit messages from other VPs, which means $2f$ tokens in place, say C0′. At this point, the transition, say $\mathrm{D0_i}$ is enabled by the guard function $[\mathrm{D_0}]$, thus completing the commit phase for VP0. Since each peer completes consensus independently, in the post-commit phase we assume that all the peers $(3f + 1)$ must complete consensus before we consider

the consensus process complete. This is done by incorporating guard function [D_i]. Table 5.1 summarizes the guard functions for this model.

Table 5.1: Guard functions for SRN model in Figure 5.2

Guard Name	Guard Function
[C_0]	If $\#P0' \geqslant 2f$, return 1, else return 0
[C_x], $x \in (1,2,3)$	If $\#Px' \geqslant 2f-1$, return 1, else return 0
[D_x], $x \in (0,1,2,3)$	If $\#Cx' \geqslant 2f$, return 1, else return 0
[D_i]	If $\sum_{y\in(0,1,2,3)} \#Dy' = 3f+1$, return 1, else return 0

Thus we have a detailed model, where we model the transmission time between all pairs of VPs in both prepare and commit phases, and model the processing and queuing delays in each VP separately. The SRN in Figure 5.2 is a template for extending the model to consider a network with more than four peers. We share our Python script in Appendix C, which we use to generate SRN model for larger number of peers.

Assuming the firing times for all transitions are exponentially distributed, the underlying process (i.e., Markov chain) suffers from the state space explosion problem. Such a detailed model is solvable when we have four VPs, but difficult to solve for a larger number of peers. This issue imposes some restrictions to the applicability of the analytic-numeric analysis of the model to compute the necessary output metrics. Hence we compute the output metrics using the "simulation approach", using the Stochastic Petri Net Package (SPNP) [65]. The SPNP code for SRN model with $n = 4$ is shared in Appendix C.

5.3 Model Validation

We discuss details regarding the experimental setup, data collection, and model parameterization in Sections 4.1, 4.2. To summarize it here, we parameterize our model as follows: Weibull (shape = 2.092, scale = 0.8468) for transitions with subscript Tx, Weibull (shape = 1.509, scale = 0.2575) for transitions with subscript Pr in

pre-prepare and prepare phase, 2-stage Hypoexponential ($\lambda_1 = 22.050$, $\lambda_2 = 267.97$) for transitions with subscript Pr in commit phase, Weibull (shape = 1.561, scale = 0.124) for transitions with subscript Q.

Since the firing time corresponding to transitions in our model have non-exponential distribution, we use SPNP tool in simulator mode only (as opposed to computing analytical-numerical solution). In the simulation approach, the SPNP package "simulates" the behavior of the SRN. For our model, we compute the total time (TT) up to a token is deposited in *Done* place, which corresponds to the "time to complete consensus" for a block. By averaging the TT over several simulation runs we obtain the "mean time to complete consensus". We conduct 5000 runs and consider the average TT value (along with confidence interval) as our result. For the empirical results, we consider the mean time observed between the statements *sendBatch* and *executeOne*, as seen in Figure 4.1. Due to the presence of outliers in the empirical results, we compare the computed mean time to consensus from our model (3.0815 ms) with the median empirical results (3.314 ms). With a relative error of about 7%, we find the results comparable and our model validated.

5.4 Model Analysis

5.4.1 Sensitivity Analysis

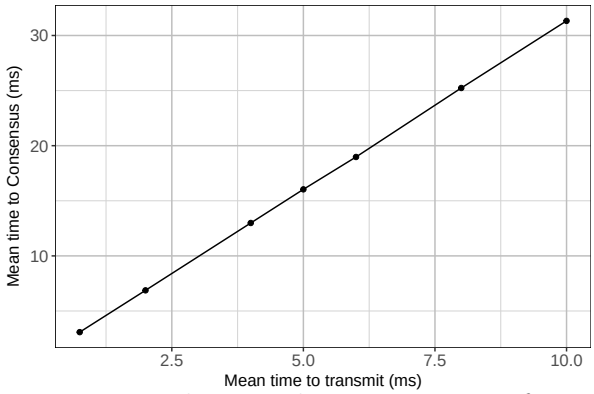


FIGURE 5.3: Sensitivity analysis with increasing Tx for equidistant peers

Let us analyze how the mean time to consensus is affected by various parameter values. Assuming all the peers are equidistant from each other, we increase the time to transmit (Tx) the message between all pairs of peers. From our analysis in Section 4.2, we assume all transitions for Tx are Weibull distributed. We vary the mean time to transmit the message from 0.75 ms to 10 ms and assume standard deviation as 20% of the mean. We plot our results in Figure 5.3. We find that an increase in the mean transmission time between all pairs of peers by 1 ms corresponds to a 3.0539 ms increase in the mean time to consensus, which makes sense since messages are transmitted in three phases of the consensus process. For reference, the round-trip latency between two nodes hosted on Amazon Web Services (AWS) ranges from 5 ms [66] for nodes within the same AWS region to around 200 ms [67] between two AWS regions across the Pacific. The average round-trip latency across the continental US is around 45 ms [68]. Note that due to the asymmetric nature of the internet routes, half of the mean round-trip time is only a rough estimate for mean time to transmit a message [69].

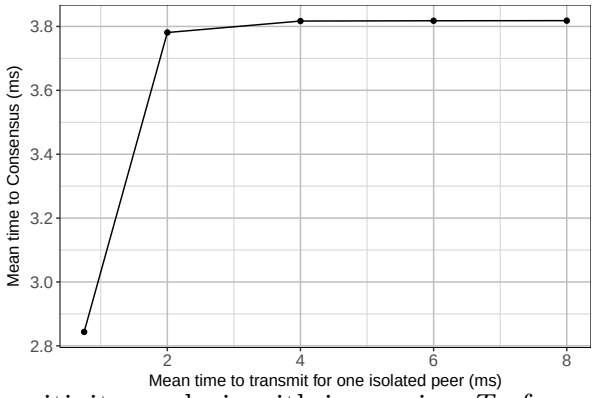


FIGURE 5.4: Sensitivity analysis with increasing Tx for one isolated peer

Next, let us consider a scenario where one peer is located far from the rest of the peers. Let us increase the mean time to transmit messages between the isolated peer to the rest of peers, keeping the mean time to transmit messages between rest of peers as 0.75 ms. Since the isolated peer will take significantly longer to complete

the consensus, let us consider that the consensus process is complete when $3f$ peers achieve consensus (changing guard function [D_i]). The results in Figure 5.4 satisfy our intuition, where the mean time to consensus would not increase significantly, since the dislocated peer would not be able to participate as actively in the consensus process.

For Pr, we vary the mean time to process a message in preprepare and prepare phase from 0.02 ms to 2 ms, and find that the mean time to consensus increases with a slope of 1.89. For Q, we vary the mean time to queue a message from 0.01 ms to 1 ms and find that the mean time to consensus increases with a slope of 3.309. Thus, a slowdown in handling incoming prepare and commit messages can have a greater impact on the mean time to consensus than the slowdown in processing a message for the new phase.

5.4.2 Large number of peers

We can easily scale our model to a large number of peers. Since $n = 3f + 1$, we consider values of n in increments of f, i.e., $n = 4, 7, 10$ for $f = 1, 2, 3$, respectively, up to $n = 100$. We evaluate the model using the parameters from our experimental validation. Let us first focus on analysis up to 10 peers, in the inset of Figure 5.5. As expected, the time to consensus increases with n, however, the slope decreases at $n = 7$. Since $2f + 1$ consensus messages are required by each peer in the prepare and the commit phases, the proportion of peers required for consensus decreases from three out of four peers (75%) for $n = 4$ to five out of seven peers (71.42%) for $n = 7$, and so on, asymptotically reaching $\frac{2}{3}$ (66.667%). However, the slope starts increasing again after $n = 10$. This happens due to increasing queuing delays for messages in the prepare and commit phases. The slope continues to increase slightly as n increases. Eventually, the mean time to consensus for $n = 100$ is 5.34 times that for $n = 4$.

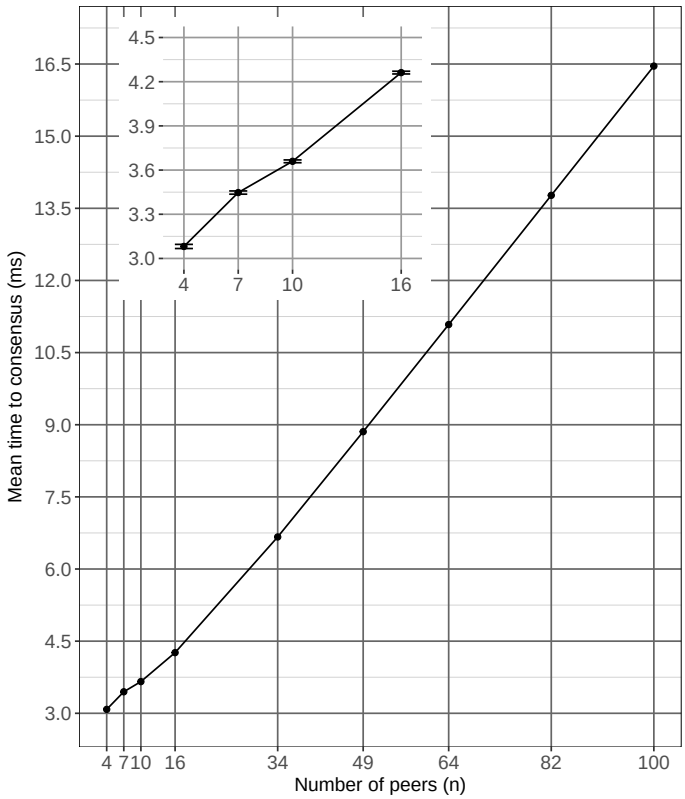


FIGURE 5.5: Mean time to consensus for large number of peers (mean Tx = 0.75ms)

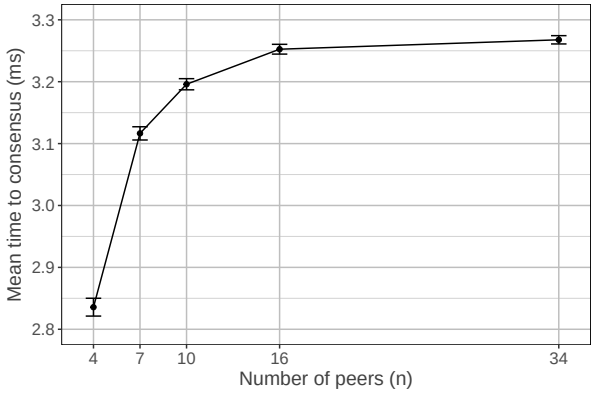


FIGURE 5.6: Mean time to consensus for large number of peers from the model that considers no queuing

To further elaborate our observation, let us consider a model without any queueing for prepare and commit phase, essentially removing transitions PO_Q to $P3_Q$ and CO_Q to $C3_Q$ and related changes in our model in Figure 5.2. Thus, when $2f + 1$ prepare messages are ready, the peer starts processing in the commit phase. We use the same parameterization as above and plot our observation in Figure 5.6. You can see that the rate of increase in mean time to consensus decreases as n increases, asymptotically approaching a value. This plot exaggerates the fact that the proportion of peers required for consensus decreases as n increases.

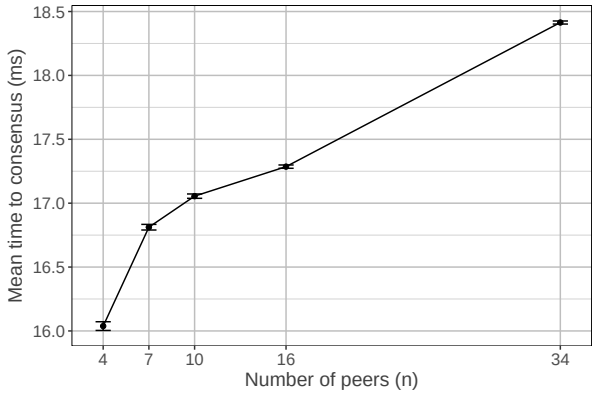


FIGURE 5.7: Mean time to consensus for large number of peers (mean Tx = 5ms)

In the analysis discussed earlier, we assumed that the mean time to transmit messages is the same as that observed in our Bluemix setup. However, in a realistic scenario, the peers might be located in separate regions of the same data centers or different data centers, and hence the mean time to transmit messages will be much larger. Let us assume that the mean time to transmit messages between all pairs of peers is 5 ms. As shown in Figure 5.7, we see a similar pattern for the mean time to consensus; however, the percentage increase in the mean time to consensus for $n = 34$ compared to $n = 4$ is 15%, compared to 116% increase when mean time to transit is 0.75 ms (Figure 5.5). This percentage increase continues to decrease for networks with even larger mean transmission delays. Thus if the transmission delays are an order of magnitude or two greater than the time to process or queue the

message, the mean time to consensus does not increase significantly as n increases.

5.5 Performance model of Block Execution process

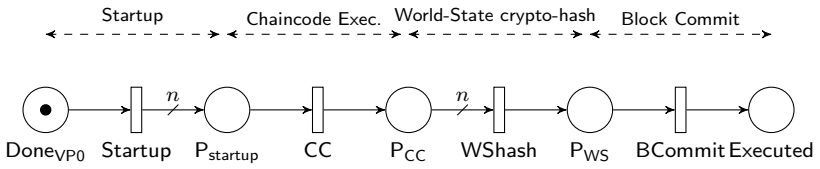


FIGURE 5.8: SRN model for Block Execution process at each VP

In this section, we present the SRN model of the block execution process. This model can be used in conjunction with the previous SRN model to compute the end-to-end latency from block creation to transaction completion.

For our modeling and analysis, we focus on the two steps that consume the most significant time: a) Transaction execution and computing the block crypto-hash, b) computing the crypto-hash for the world-state. As highlighted in Figure 4.3, we consider "Startup" as the stage before the first transaction execution begins, and we consider "Block Commit" as the stage after the crypto-hash is computed. Since all these steps are performed in a sequence, we can model this as a simple SRN model as shown in Figure 5.8.

After the consensus process is complete, the block is executed at each VP independently. Thus place Done_{VP0} indicates completion of consensus process at VP0. After a few steps to startup the block execution process (represented by transition *Startup*), each transaction is executed by the corresponding chaincode. The chaincode needs to be launched when a transaction corresponding to it is encountered for the first time, which we assume is already done in a steady-state. Assuming our block has n transactions, each transaction is executed and its crypto-hash is computed serially (represented by transition *CC*. This process is repeated n times before all transactions are complete. After a few smaller steps (which we ignore in

our analysis), the next major step is to compute the crypto-hash of the world-state as described earlier. For simplicity, we represent this as a single transition $\mathrm{WS}_{\mathrm{hash}}$. Finally the process of storing key-value pairs in persistent storage, and other closing steps are represented by transition *BCommit*.

One of the challenges here is to estimate the value of n. Since each transaction execution (via chaincode) takes 6.336 ms, this contributes significantly to the block execution time. Assuming that the transaction arrival process is Poisson with rate λ, since our PBFT batch timeout (T) is one sec., then the average number of transactions per block $n = \lambda T = \lambda$. Keeping this observation in mind, we decided to use Poisson arrival process while testing our HLF V1 network (ref. Section 6).

5.6 Discussion

5.6.1 Threats to validity

The results of our analysis are subjected to the validity of the parameters estimated in our experiments. The time to transmit a message (Tx) that we considered in our model corresponds to the end-to-end transmission delay of consensus messages between two peers. Given the granularity of our logs, this is the best we can capture. Future work should consider measuring the transmission latency in two parts: between the two peers and latency due to the software stack. Same way, time to process a message (Pr) consists of various steps, including message authentication, verifying for duplication or incorrect sequence number, where time consumed in each step is hard to measure. In our research of HLF V1, we have paid more attention to such details (ref. Section 6.4). Also, larger networks might result in blocks of larger size, which could require more time to process and queue. Given the closeness of results from our model ($n = 4$) with the empirical results, we feel confident about our results and analysis.

5.7 Related Work

5.7.1 Performance evaluation of BFT consensus protocol

Croman et al. [21] study the latency (time to confirm a transaction) and throughput (number of transactions per second) of a consensus system using PBFT protocol up to 64 peers spread across eight AWS data centers. They find that latency increases and throughput decreases as the number of peers increases. Due to $O(n^2)$ messages exchanged, they express concern that the performance will be much worse for higher number of peers, which we attempt to answer in our research. The authors in [70] model the performance of PBFT protocol using a combination of model checking and simulation techniques. The authors consider two time-consuming operations: network operations (i.e. message transmissions) and cryptography (using MAC). Based on our measurement abilities, we consider "time to process message", which includes cryptography and other verification steps together. The authors developed micro benchmarks to obtain sample execution time for each path in the target protocol, focusing their analysis on the paths with longest execution time. In our work, we do performance modeling using SRN in a general case, without focusing only on the longest execution path. Clement et al. [71] study the fault-tolerance of popular BFT protocols such as PBFT [11] and Zyzzyva [72] in the presence of Byzantine faults. They perform experimental evaluation with four co-located peers and observe a latency of under 15ms for 99.99% of requests, where latency consists mainly of the time to complete consensus. From our SRN model, we evaluate mean time to consensus for much larger networks. Similar modeling and analysis can be performed for other popular BFT protocols such as Zyzzyva.

5.8 Conclusions

In this chapter, we presented a detailed and scalable model of the PBFT consensus process for Hyperledger Fabric v0.6. We created a Fabric network using IBM Bluemix service, running a production-grade IoT application and use the data to parameterize and validate our model. We estimate the "mean time to complete consensus" using simulation techniques using the SPNP package. For four peers, we find the solutions from the SRN model are comparable to the empirical results, with a relative error of about 7%. Using the validated SRN models, we analyze PBFT consensus process up to 100 peers and find that the mean time to consensus increases by 5.34 times for 100 peers compared to that for four peers if the transmission delays are of the same order of magnitude as the processing and queuing delays. However, in a real-world scenario where peers are geographically dispersed, and transmission delays are large, the percentage increase in the mean time to consensus would not be as significant for a large number of peers.

Future research work should focus on validating the model for a larger number of peers. Also validate the model for different deployment configuration and PBFT configuration parameters. We could not continue this research work with Fabric v0.6 since the community had move on with development of v1.0, which had a completely different architecture. Also there are technical issues in the implementation of Fabric v0.6 (discussed in detail in [57]), due to which the performance was expected to be limited. Future research work could consider other order-execute style blockchain platforms that are free and open-source, such as Hyperledger Sawtooth Lake. Similar research could be conducted for other popular consensus protocols, such as BFT-SMaRt [73], Raft [74].

6

Empiricial Analysis for Hyperledger Fabric V1

We setup a Fabric V1 network in the Duke datacenter and use the data collected to parameterize and validate our model. Unlike the research summarized in Section 6.8.1, the goal of this work is not to benchmark Fabric's performance by stress loading the system; instead we take measurements from a Fabric network subjected to a realistic traffic pattern. In this chapter, we provide details of the tools used, network setup, and our methodology for data collection and analysis and interesting empirical observations. This chapter provides extensive empirical details of our work presented in our published work [75].

6.1 Experimental Setup

The deployed network is shown in Figure 6.1. Each node is launched as a Docker container and then connected in a network using the Docker Swarm[1]. Although peers and orderers can run natively on the physical/virtual machine, a network of Docker containers is a recommended approach for deploying Fabric networks [76]. This ap-

[1] `https://docs.docker.com/engine/swarm/`

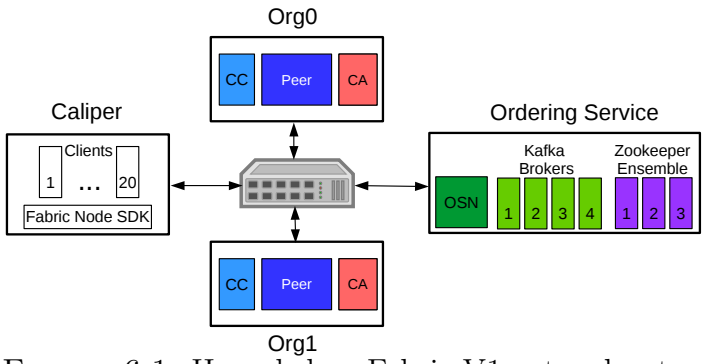


FIGURE 6.1: Hyperledger Fabric V1 network setup

proach is taken by the Hyperledger Cello[2] project as well. Containers corresponding to each organization (peer and CA) are run on an independent physical node (Org0, Org1). Peers executes chaincode in a separate container (called CC). All containers corresponding to the ordering service run in a single physical node ('Ordering Service'). It includes one ordering service nodes (OSN), four Kafka brokers, three ZooKeeper nodes. Hyperledger Caliper is deployed on a separate physical node, with multiple client threads that interact with the locally installed Fabric Node.js SDK[3]. We provide the detailed steps for setting up the Fabric network in Appendix A. Note that we consider a network with one peer per organization, and hence we ignore the implication of the gossip protocol on the network performance.

Physical machines corresponding to Org0, Org1, and Caliper have 4 CPUs (1 socket, 4 core) (Intel Xeon 2.2 GHz) with 12GB RAM, and that running ordering service has 16 CPUs (2 sockets, 4 cores, 2 hyper-threads) Intel Xeon 2.4 GHz with 32GB RAM. Each machine is running Ubuntu 16.04 LTS on a 7200 rpm Hard Disk Drive with Fabric release v1.1[4] installed. All physical machines are connected with a 1 Gbps switch. All nodes are synchronized using Network Time Protocol (NTP) service so that transaction latency can be measured across nodes. Communication

[2] `https://www.hyperledger.org/projects/cello`

[3] `https://github.com/hyperledger/fabric-sdk-node`

[4] commit id 523f644

between all nodes is configured to use Transport Layer Security (TLS).

6.2 Load generation using Hyperledger Caliper

To execute workload, we use the Hyperledger Caliper [23], recently approved as a project by the Hyperledger community. It is a benchmark execution platform that enables the user to measure the performance of different DLT platforms consistently. The end goal of this tool is to help users evaluate the blockchain platform that best suits their application. At the time of writing this thesis, it supported Hyperledger Fabric, Hyperledger Sawtooth Lake, Hyperledger Iroha and Hyperledger Composer projects.

An advantage of using this tool is that it takes care of the complex workflow performed by the client (using Fabric SDK) including handling of event notifications from the peer. To generate traffic following a Poisson arrival process, we implemented a new rate-control function in Caliper. A current limitation of Caliper is that it supported interaction with only one OSN node.

6.3 Test application

For our performance testing, we leverage the *simple* chaincode provided by the Caliper tool and extend it for our needs. This application maintains account balances for users. It can perform two functions. Function 'open' checks if an account exists, and if not, create a new account and assigns it an account balance. Thus it performs one read, one write operation to the key-value store. Function 'transfer' allows transfer of money from one account to another. Thus it performs two read, two write operations. Before running 'transfer' transactions, we run 'open' transactions for a few mins. to populate the key-value store. Authors in [41] also followed a similar approach of differentiating workloads by the number of read-write operations. For both the functions, the input account number(s) are selected randomly; hence

there is almost no dependency between consecutive transactions; thus the transactions never seem to fail the MVCC validation. This way, we can easily generate a workload of all valid transactions at a high rate[5].

6.4 Measurements

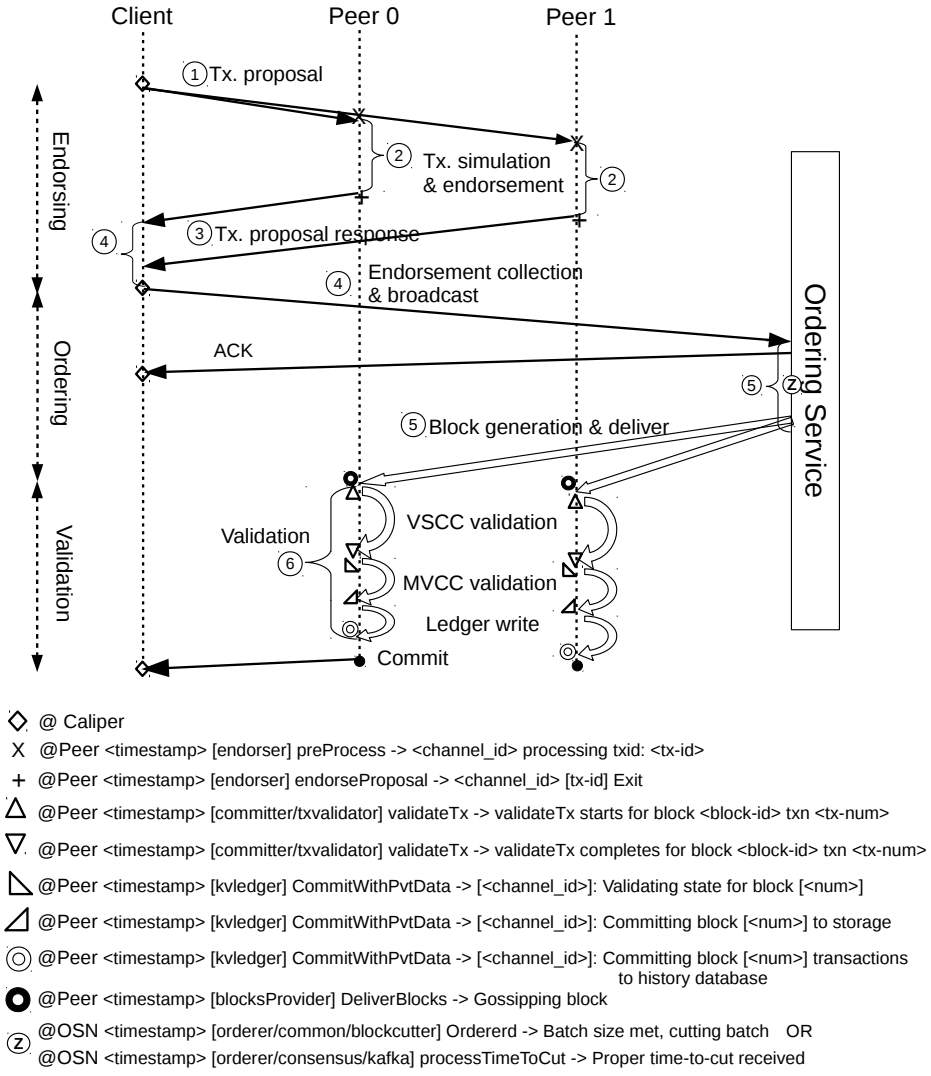


FIGURE 6.2: Transaction life-cycle on Hyperledger Fabric V1 with measurement details

We measure the time taken to perform critical steps in the transaction life-cycle

[5] https://lists.hyperledger.org/g/perf-and-scale-wg/topic/17550808

by analyzing the caliper output files, peer logs, and orderer logs. Figure 6.2 shows the transaction sequence diagram along with the venue where the timestamp is captured and a snapshot of the log entry in peer/orderer. We converted these vital log entries from DEBUG mode to INFO mode. For some parameters such as time to endorsement, there were multiple candidate log entries for start and/or end of the operation. We identified the best choice by performing distributional analysis (ref. Section 6.5) for all combination of start and/or end operations and verify that the measurement does not inadvertently include a queuing delay.

To measure the mean queue length at each transaction life-cycle phase, we add additional log entries to capture the time when a new transaction enters and leaves that phase. For each life-cycle phase, let us consider its state as the number of jobs waiting for service or in-service. By analyzing these log entries, we can measure the time spent in each state. The average time spent in each state weighted by the number of jobs gives the average queue length in that phase [77].

We also increased the timestamp resolution to microseconds. Finally, we measured the overhead of additional logging for a setup with block size 500 and $\lambda_C = 100$, and observe only a 0.64% increase in the average transaction latency, and 0.48% increase in 75%ile latency. Our Fabric source code changes can be found here[6].

We run Caliper with 20 client threads and test duration of 240 sec., ensuring that the transaction arrivals follow a Poisson arrival process. We trim the first and last 20 sec. of a test run as a ramp-up and ramp-down phase. To validate our model for different configuration settings, we vary three parameters: a) client tx. arrival rate (λ_C), b) block size, c) Transaction type ('open' and 'transfer'). Due to our hardware limitations, we could not vary the number of CPUs for validating peer. We plan to pursue it in our future work. For a given block size, we keep λ_C sufficiently high such that most blocks are created due to block size rather than

[6] `https://bitbucket.org/hvs2/fabric-perf-model/src/master/diff_files`

timeout (ref. Section 7.5.2). All docker instances were restarted between each test run.

6.5 Model Parameterization

We start this section by describing the general procedure followed in analyzing the datasets to estimate the model parameters. Further discussion on the model parameters used in our analysis in Chapter 7 is described in Section 7.2.

The parameter values for our model are measured either at transaction-level or a block-level. The ones at transaction-level are time to endorse, time to transmit to orderer, and time to validate (VSCC). The ones at block-level are time to consensus (+ block transmission), time to MVCC check, and time to ledger write. We run our experiments for each combination of block size and client arrival rate and collect datasets for each combination separately. Note that the experiments are run for the same duration for each combination; hence the dataset size varies by the arrival rate.

For transaction-level parameters, ideally, there should be no statistical variation in the measurements across the combinations. However, using the ANOVA and Tukey's honest significance test, we find that it is not the case. It makes model parameterization challenging since it is cumbersome to manage a set of parameter values for each combination of the experimental setting. Also, we lose the purpose of preparing a stochastic model. From the above-described collection of datasets, our goal here is to derive a single dataset for each model parameter which we can use to parameterize our model. We describe our procedure as follows.

For the transaction-level parameters, for each block size, we compare the datasets across client arrival dates using Tukey's Honest significance test. In our case, we had three datasets corresponding to three different arrival rates. We found that at most one out of three datasets was statistically different than the others. We discarded such datasets and merged the remaining datasets. Thus we have one dataset for each

block size. Then we perform a similar comparison again for datasets across block sizes and discarded the ones that were statistically different than the others. Thus we derive one dataset for each parameter value. In all the above comparisons, we equalize the length of datasets by sampling the datasets by the length of the smallest dataset.

For block-level parameters, the measurements are expected to be statistically different for each block size. We validate this by comparing the datasets using Tukey's honest significance test. We follow the similar procedure as described earlier; however we derive a dataset for each block-level.

Let us provide the empirical details of datasets corresponding to each model parameter. Then, we attempt to fit well-known statistical distributions, following the same procedure as described in Chapter 4. We also describe our challenges in fitting distributions to datasets of transaction-level parameters.

6.5.1 Transaction-level parameters

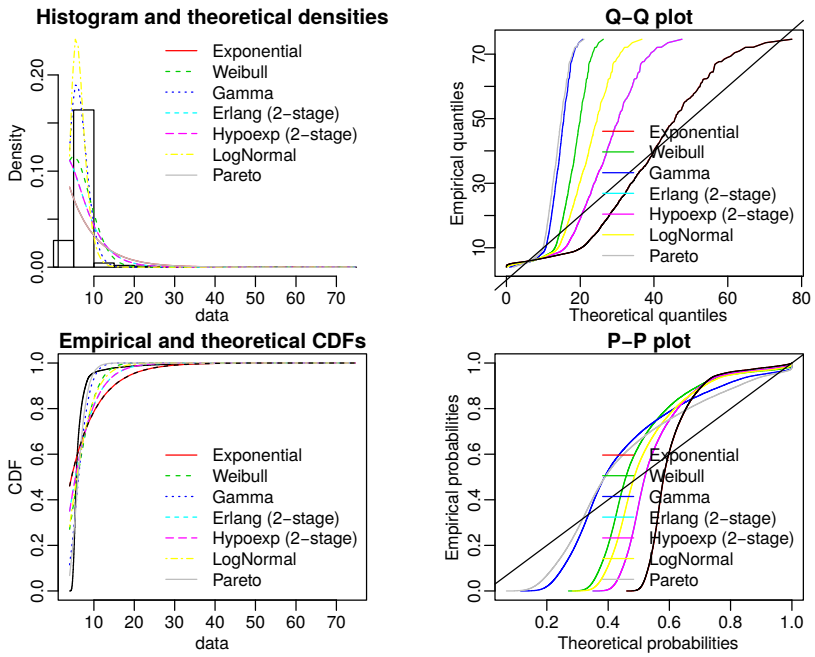


FIGURE 6.3: Empirical analysis for 'time to client processing' (T_{Pr})

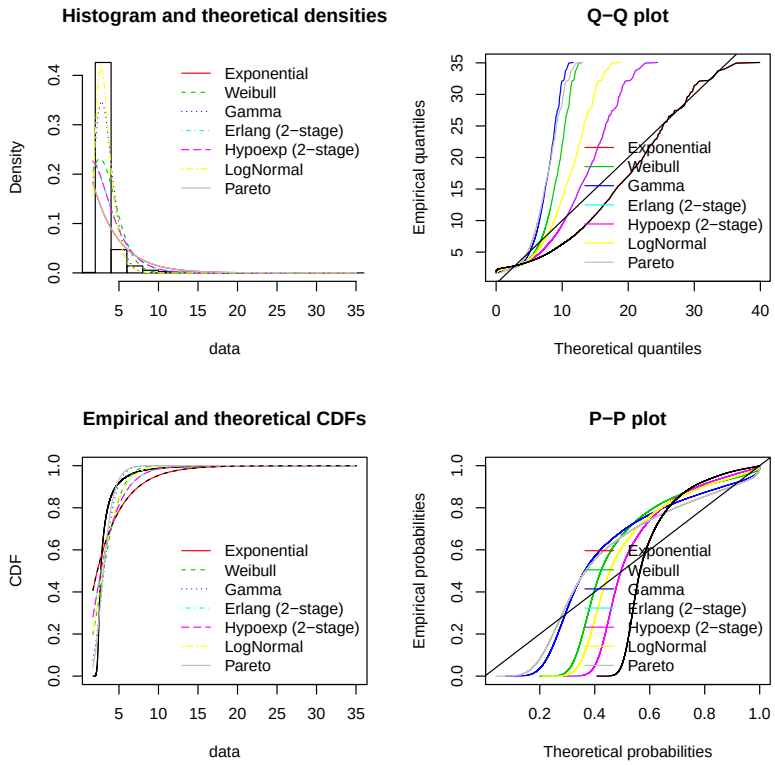


FIGURE 6.4: Empirical analysis for 'time to endorsement' (T_{En})

Figures 6.3,6.4,6.5,6.6 shows the empirical density, cumulative distribution function, Q-Q, and P-P plots along with distribution fitting for the datasets corresponding to four model parameters, viz., client processing, endorsement, transmit to ordering service, and VSCC validation respectively. Table 6.1 presents the summary statistics of the plotted datasets.

Table 6.1: Summary statistics for datasets of transaction-level parameters

Statistic	Client pr. (ms)	Endorsement (ms)	Tx. to OS (ms)	VSCC validation (ms)
Minimum	4.008	1.703	2.719	1.217
25 %ile	5.227	2.460	3.778	1.830
Median	5.640	2.703	4.187	2.090
Mean	6.470	3.249	5.221	2.525
Std. Dev.	3.513	1.73	5.125	1.433
75 %ile	6.545	3.323	4.878	2.741
95 %ile	9.426	6.059	8.770	4.445
Maximum	74.400	35.060	121.472	29.426

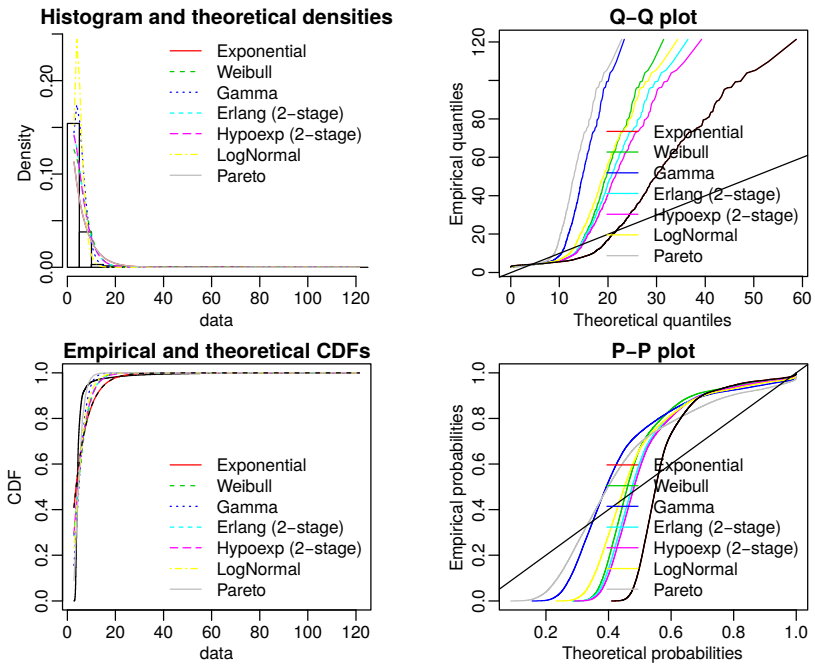


FIGURE 6.5: Empirical analysis for 'time to transmit to ordering service' (T_{Tx})

From the density plots, we can see that all the datasets have a heavy-tail. This makes distribution fitting challenging, even when we try to fit popular heavy-tail distributions such as Pareto or LogNormal. Using the goodness of fit criteria such as AIC and BIC, we find Gamma distribution to be the best fit among all the distributions. However, looking at the Q-Q plot, it seems the dataset has an unusually large number of samples in the 75 percentile to the top percentile range. It is hard to say if it is a measurement error or it is how the system behaves.

In summary, we experienced challenges in fitting distributions to the above datasets. We tried fitting distributions on a subset of 1000 samples from the original dataset, but the results are similar.

6.5.2 Block-level parameters

Figures 6.7,6.8, and 6.9 shows the empirical density, cumulative distribution function, Q-Q, and P-P plots along with distribution fitting for the datasets corresponding to block-level model parameters, viz., block creation and delivery, MVCC validation

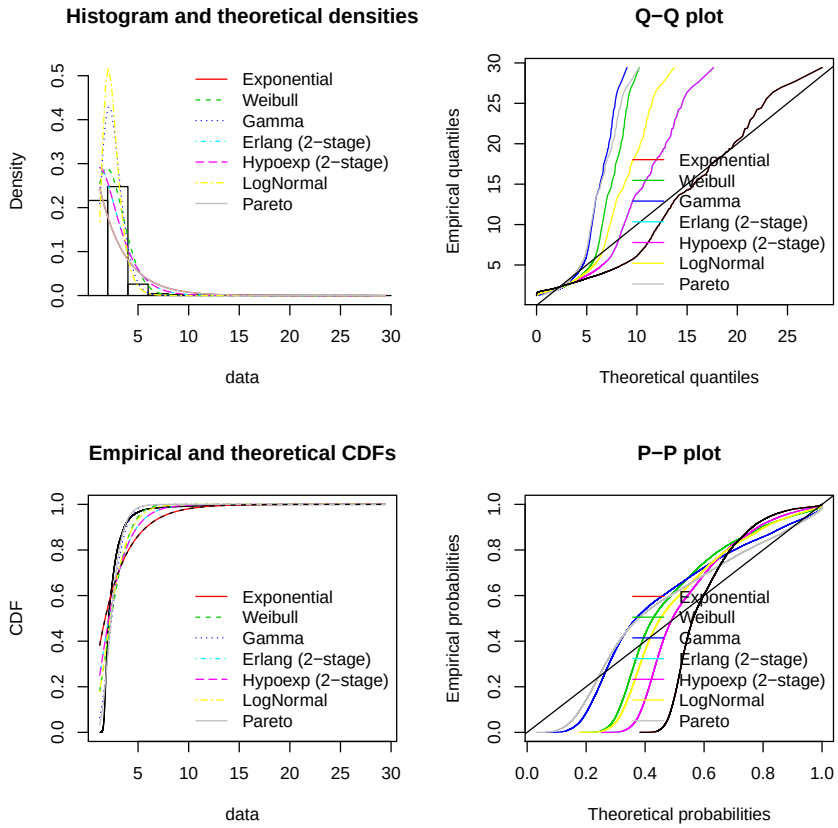


FIGURE 6.6: Empirical analysis for 'time to validate transaction' (T_{VSCC})

Table 6.2: Summary statistics for datasets of block-level parameters

Statistic	Block creation + delivery			MVCC validation(ms)			Ledger write (ms)		
Block Size	40	80	120	40	80	120	40	80	120
Minimum	56.31	64.76	75.80	1.948	4.223	6.228	181.8	178.9	163.4
25 %ile	68.93	76.27	87.64	2.322	4.641	6.802	199.0	203.1	179.4
Median	73.94	80.00	91.28	2.442	4.841	7.007	205.8	207.9	185.6
Mean	75.74	81.60	93.56	2.562	5.103	7.200	207.8	208.3	188.4
Std. Dev	11.82	9.58	10.36	0.527	1.055	0.795	13.6	10.4	18.4
75 %ile	79.67	85.04	96.53	2.624	5.091	7.253	212.1	213.5	193.0
95 %ile	97.64	97.63	113.61	3.277	7.014	8.319	231.4	222.7	204.9
Maximum	134.46	143.05	154.21	6.013	12.314	12.163	283.3	274.4	309.6

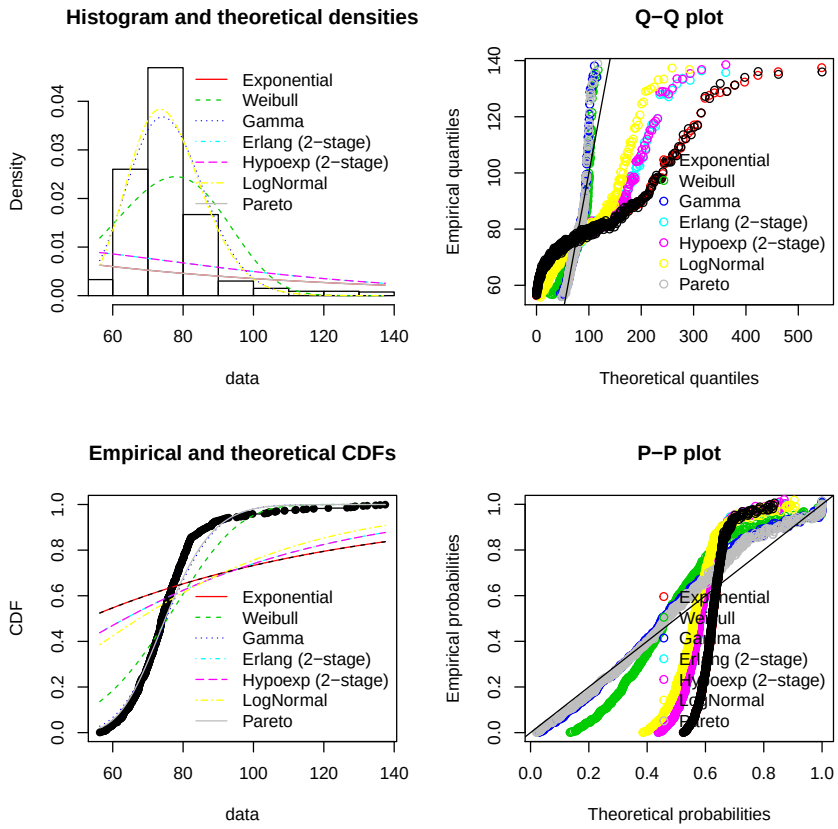


FIGURE 6.7: Empirical analysis for 'time to block creation and delivery' (T_{OS})

and ledger write respectively for block size 40 only. Table 6.2 presents the summary statistics of the plotted datasets.

For the block creation and delivery, LogNormal is the best-fit distribution, followed by Gamma. The measurement for this parameter can be split into two: block creation and block transmission. Across the block sizes, we find that only the block transmission time changes significantly. Thus, the total time to block creation and transmit varies across block sizes mainly due to the block transmission. Thus Kafka-based ordering service does not take significantly longer to create blocks of larger size. Given the size of the block is large enough (in 100s of kBs), the transmission is throughput bound rather than latency bound. Hence transmission time is proportional to the size of the block (ref. Section 6.6.2). For reference, the size of each

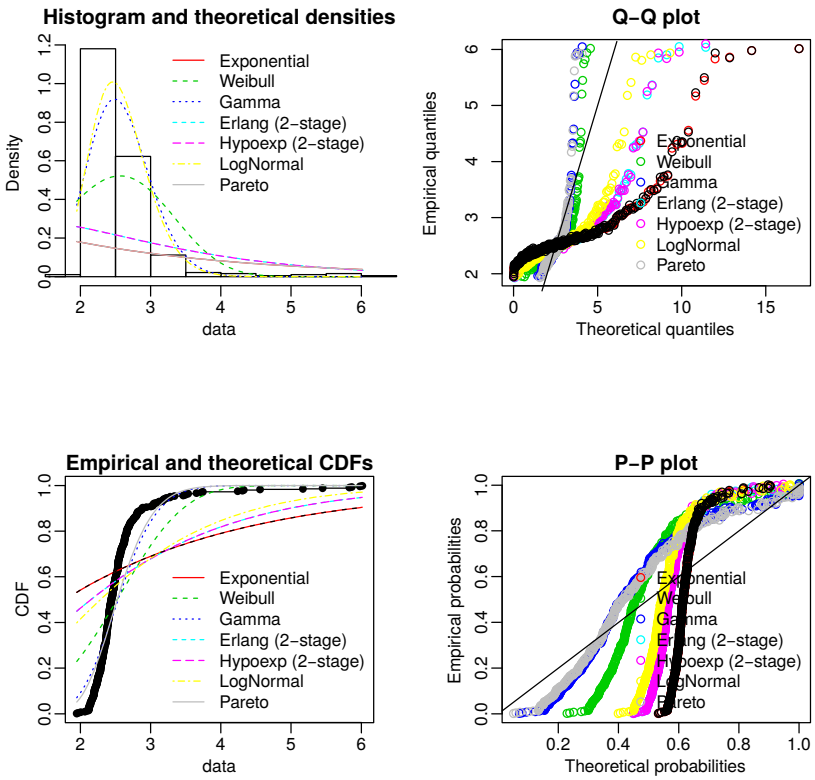


FIGURE 6.8: Empirical analysis for 'time to MVCC validation' (T_{MVCC})

'open' transaction is around 3.7KB, the size of blocks with batch size 40, 80, and 120 are 148KB, 296KB, and 440KB respectively.

For the other two parameters as well, LogNormal followed closely by Gamma are the best-fit distributions. However, from the Q-Q plot, it looks like even the Weibull and Pareto distributions are a reasonable fit. Across the block sizes, the time to MVCC validation seems to increase in proportion to the block size, which is expected. However, for the ledger write, we are not sure why ledger write for block size 120 takes a shorter time than the other two. We rerun the experiment to collect new datasets, but the observations were consistent.

In summary, distribution fitting for block-level parameters looks rather straightforward compared to that for the transaction-level parameters. We have shared the

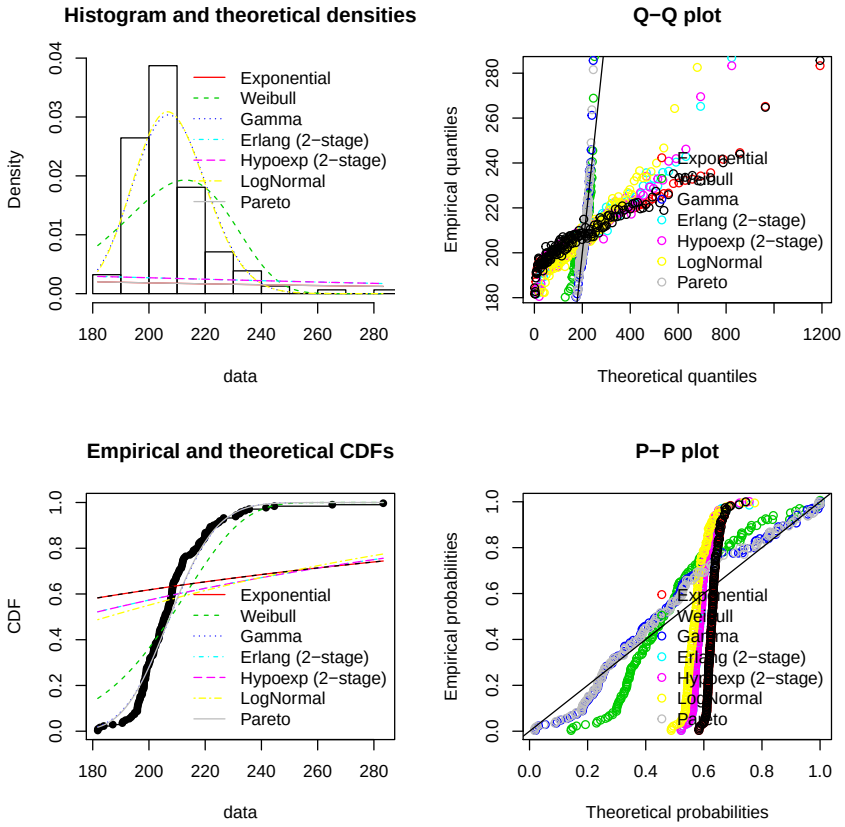


FIGURE 6.9: Empirical analysis for 'time to Ledger write' (T_{Ledger})

raw datasets in our repository[7] for further research efforts on distribution fitting.

6.6 Analysis by transaction phase

Motivated by the empirical analysis presented in [22, 41], let us analyze each transaction phase, viz. endorse, ordering, and validation. These phases are shown in Figure 6.2. Unless otherwise mentioned, all analysis in this section is for block size 40 with $\lambda_C = 80$.

[7] `https://bitbucket.org/hvs2/fabric-perf-model/src/master/data_analysis/datasetsV1`

Table 6.3: Summary statistics for time to complete each transaction phase for block size = 40, $\lambda_C = 80$

Statistic	Endorsing (ms)	Ordering (ms)	Validation (ms)
Minimum	7.00	59.36	183.7
25 %ile	8.00	195.61	218.0
Median	10.00	328.09	227.1
Mean	13.42	337.25	228.4
Std. Dev.	10.18	167.34	15.6
75 %ile	13.00	464.40	236.8
95 %ile	35.00	609.60	253.3
Maximum	146.00	1063.09	328.7

6.6.1 Endorsing

Time to complete endorsement

For transactions with endorsement policy `AND(Org1, Org2)`, let us analyze the time to complete endorsement, which is the time difference between the transaction creation (at Caliper) and the time endorsement process is completed (at Caliper as well). Figure 6.10 shows the empirical density and CDF plots. The summary statistics are presented in Table 6.3.

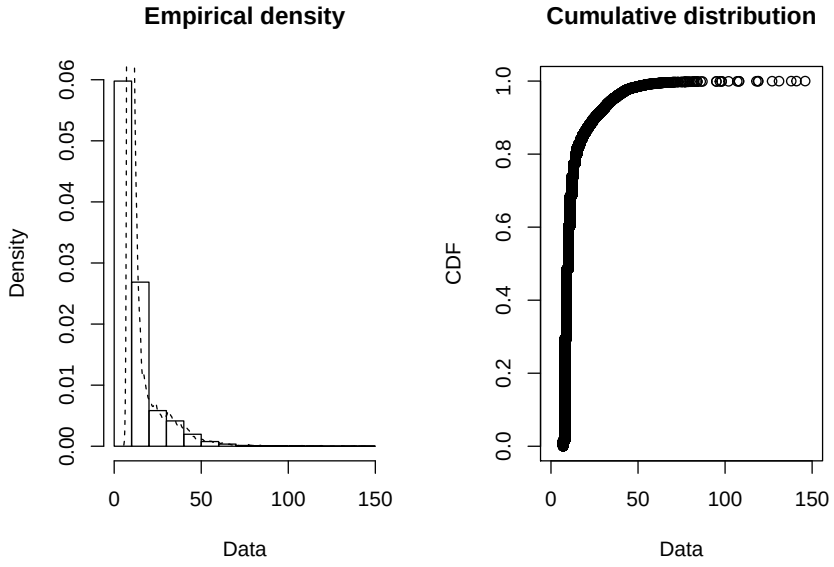


FIGURE 6.10: Time to complete endorsement for AND() policy

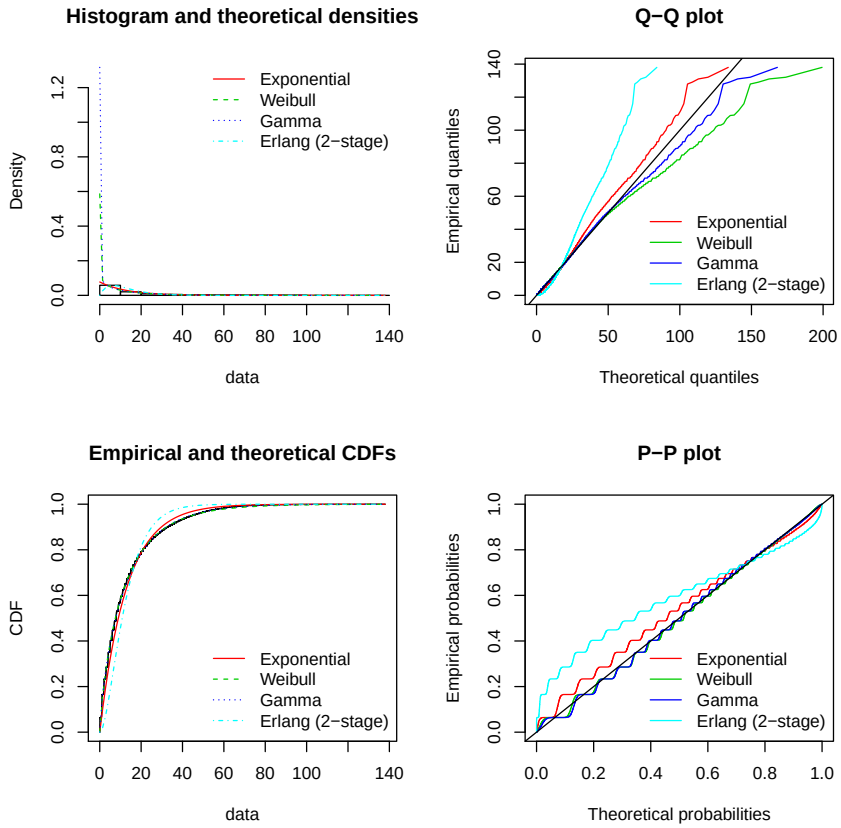


FIGURE 6.11: Empirical analysis for 'Time to complete endorsement' for AND() policy

Let us analyze the arrival process of transactions at the ordering service, by analyzing the inter-arrival time between endorsed transactions. The results are shown in Figure 6.11. The mean inter-arrival time is 12.94 which is close to the mean inter-arrival time of 12.927 ms for transactions at the client.

From the distribution fitting analysis, the best-fit distribution was Weibull followed by Gamma and Exponential. Since Exponential distribution is a reasonably good fit, we can assume that the endorsed transaction arrivals follow a Poisson process (as done in Section 7.5.2). It also encourages us to consider a simpler model for the full network (Section 7.4), by abstracting out the endorsement process for

straightforward endorsement policies.

6.6.2 Ordering

Time to complete ordering

The ordering latency of a transaction is the time difference between an orderer receiving an endorsed transaction and the transaction included in the block and received by the peer. A significant component of this measurement is the waiting time of the transaction before it gets included in the block, which would vary by the block size and arrival rate of endorsed transactions. From the summary statistics (Table 6.3), ordering latency is the most significant contributor to the transaction latency.

Block transmission time

The summary statistics for block transmission time for various block sizes is presented in Table 6.4. We observe that the block transmission time increases linearly with the block size. We attempt to fit distributions (shown for block size 40 in Figure 6.12) on datasets of each block size. In each case, we find that LogNormal is the best-fit distribution, followed closely by Gamma and then Weibull.

Table 6.4: Summary statistics for block transmission time with $\lambda_C = 80$

Statistic	Block size 40 (ms)	Block size 80 (ms)	Block size 120 (ms)
Minimum	5.619	8.798	14.35
25 %ile	6.673	11.073	17.31
Median	6.863	11.696	17.97
Mean	7.202	11.959	18.31
Std. Dev.	0.861	1.407	1.75
75 %ile	7.506	12.666	19.32
95 %ile	8.919	14.454	21.53
Maximum	12.503	18.989	26.39

Block arrival process λ_B

We analyze the inter-arrival times between blocks at the peer (Figure 6.13). The mean inter-arrival time is 521.2 ms with std. deviation of 79.44 ms.

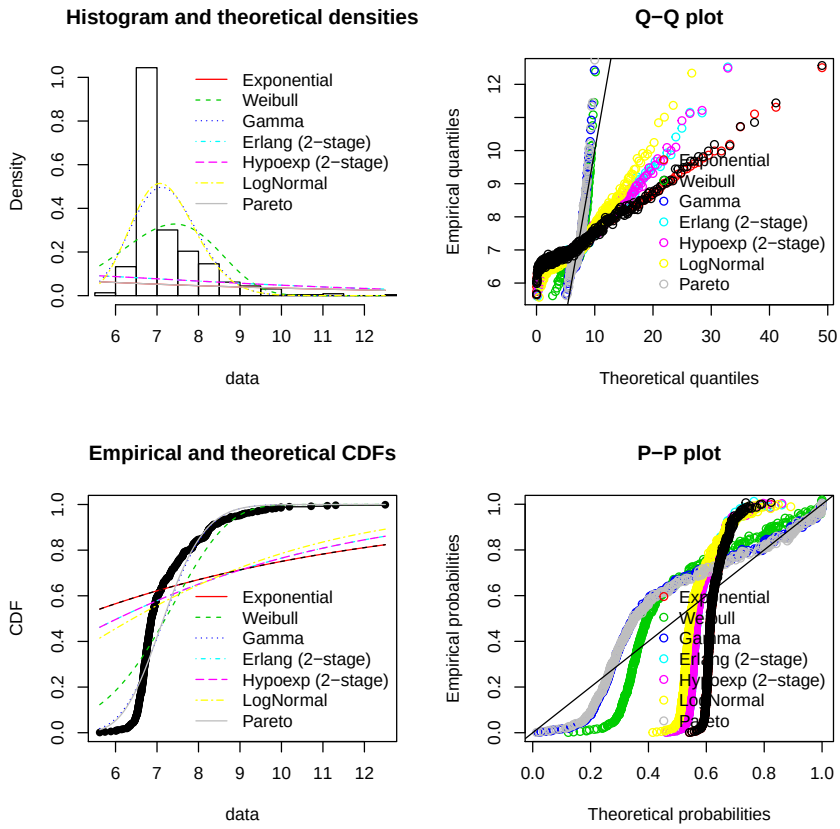


FIGURE 6.12: Empirical analysis for 'Block transmission time' for block size 40, $\lambda_C = 80$

6.6.3 Validation

Time to complete validation

We measure the validation latency of a transaction as the time difference between a peer starting a transaction validation, and the peer receives the transaction completion notification. It consists of VSCC validation, MVCC validation and ledger write. Note that the MVCC validation and ledger write are measured at a block level. From the summary statistics (Table 6.3), we can see that the validation latency is in the range of 100s of ms.

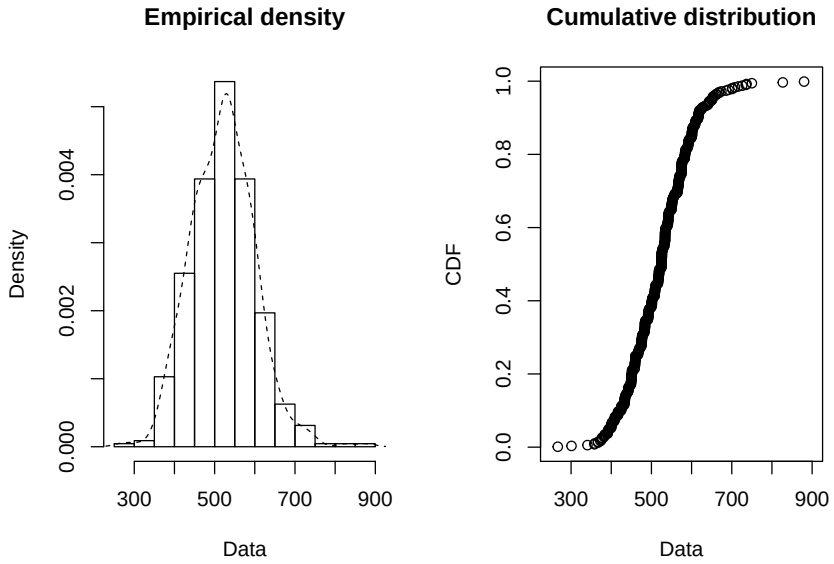


FIGURE 6.13: Inter-arrival time between blocks (block size 40, $\lambda_C = 80$)

6.7 Implications of block size on transaction throughput and latency

Irrespective of the block size, generating blocks is computationally expensive since they need to be cryptographically signed [78]. This is true whether the ordering service is designed using a crash-fault tolerant messaging service like Kafka or using a BFT consensus mechanism. Additional performance implication of using BFT style consensus mechanism is that the latency increases significantly as the number of peers increases [79]. Since block creation and ledger write take the longest time, both of which are found to be independent of the block size, from our analysis (ref. Figure 7.5), we show that the maximum throughput increases significantly as the block size increases, albeit with an increase in latency. Similar observations are made empirically by authors in [22] as well. However, authors in [58] observed that the maximum throughput starts tapering off after a particular block size. Their experiments were done with HLF v1.0 where the VSCC check was not parallelized, and hence the committing peer was the bottleneck [41]. We need to repeat our

experiments with block sizes larger than 120 to ensure that this problem cannot be reproduced for HLF v1.1 onwards.

Another implication of block size is the block transmission time, which is a big problem in large overlay networks such as Bitcoin [54, 21]. It is critical to ensure that the high percentage (if not 100%) of peers should receive the block before the next block is dispatched. This prevents an unfair advantage to peers that receive blocks much before the other peers [21]. In a permissioned blockchain network, the location of the peers is known, and hence the latency between the committers and the ordering service is well known; hence the block size (or frequency based on expected requested arrival rate) should be decided to prevent any unfair advantage. Fortunately larger block sizes are more likely to be throughput bound than latency bound [54, 21], hence preventing unfair advantage to peers located disproportionately closer to the ordering service than others. Also for larger block sizes, the system throughput would not get affected by the block transmission time [78].

Regarding batch timeout, although it has no performance implication in high throughput systems (as we show in Section 7.5.2), it is an essential parameter to tune for low throughput but latency sensitive services such as IoT to bound the expected latency.

6.8 Related Work

In this section, we discuss papers that provide empirical analysis for Fabric V1 and other blockchain platforms. Note that the related research work covering Fabric v0.6 is discussed in Section 4.4.

6.8.1 Hyperledger Fabric V1

Performance of Hyperledger Fabric v1.0 was studied extensively by Thakkar et al. in [41], where they vary five tunable parameters: block size, endorsement policy, number

of channels, number of vCPUs for peers, and key-value database (GoLevelDB vs. CouchDB). They observed that VSCC validation is a performance bottleneck, but can be parallelized easily. Validation of endorsement policies was particularly time-consuming for policies with a large number of organizations (each with a unique signature) and deeply nested policies. The system scaled well with an increasing number of channels if the number of vCPUs allocated to the channel were greater than or equal to the number of channels. They also observed that CouchDB was much slower than GoLevelDB since it runs in a separate container from the peer and peer reads/writes using https REST API calls. In summary, their research resulted in three optimizations for Fabric: parallelization of VSCC validation, cache for Membership Service Provider (MSP), and bulk read/write for CouchDB, all of which were incorporated in release v1.1, that was studied in [22] and our work.

Androulaki et al. [22] presented extensive details of Fabric V1. For performance evaluation, they developed an application called *Fabcoin* using the data model is similar to the Bitcoin-style UTXO[8], where previous incoming transactions need to be spent to issue new transactions, thus consecutive transactions are independent. From the latency breakup across the three transaction phases, they found that ordering phase takes the longest, followed by validation. Within the validation phase, the VSCC validation takes the longest, followed by ledger write and MVCC validation. These observations are consistent with ours.

In contrast to [22, 41] where they run peers on virtual machines in a cloud datacenter, we run our nodes on physical machines. Also, [22] used Solid-state Drive (SSD) as opposed to HDD in our case, perhaps why their ledger write latencies are much lower.

Baliga et al. [58] also evaluated the performance of Fabric v1.0. In their workload, they vary the percentage of read/write transactions, including null workload (no read

[8] https://bitcoin.org/en/glossary/unspent-transaction-output

or write). As the transaction arrival rate increases, the read workload throughput increases, however, the write and even the null workload throughput tapers off. Thus the performance of a Fabric network is limited by the three-phase transaction. Their unique contribution is the microbenchmarks, where they vary the a) read-write set size, b) size of key-value store database, c) chaincode payload. Note that they used 'solo' ordering service in their experimental setup, which provides no crash-fault tolerance and is not expected to be used in a production deployment.

Sousa et al. [78] integrated BFT-SMaRt [73] as an ordering service for Fabric V1, and present performance analysis. They hypothesize three potential bottlenecks for throughput: a) rate of ordering blocks by BFT-SMaRt consensus process, b) no. of blocks signed per second, c) size of generated blocks. To evaluate the bottleneck due to signature generation, they develop a benchmark and run it on a 16-core system, and find it reaching 84k signatures per second. They evaluate the HLF system in a LAN environment by varying the no. of peers for ordering service (4, 7, 10), the no. of receivers (1 to 32), and the size of the transaction proposal (40 bytes to 4 kB). They find a) throughput decreases as no. of receivers increase, but the rate of decrease is not much for larger transaction size (1 to 4kB). This is because, for larger block sizes, the overhead of consensus protocol dominates the overhead of block transmission. For a larger number of receivers, the throughput saturates in spite of the block size or the number of orderers. The authors also deployed the HLF in a geo-distributed setting over Amazon Web Services data centers. They found that latency is in the range of 500 ms, and does not vary by the block size. Their performance analysis is focused only on the ordering service.

6.8.2 Public blockchain networks

The two most popular public blockchain networks are Bitcoin [2] and Ethereum [8]. They operate over a large overlay network, which means peers receive blocks at in a

wide range of propagation times. Croman et al. [21] study the performance issues in Bitcoin networks and measure throughput and latency in Bitcoin as a function of block size and block interval. Since transactions are committed in blocks, the highest tx throughput is effectively capped at maximum block size divided by block interval. Their goal was to find the optimal block size and interval such that it maximizes the transaction throughput and/or minimize latency. Following up on the Bitcoin's performance study conducted by Decker and Wattenhofer [54] in 2012, they repeat the measurement studies in 2014-15. They observed the 10%, median and 90% block propagation times are 0.8 sec, 8.7 sec., and 79 sec. respectively. The average block size is 540 KB. They also observe that above 80kB block size, the propagation times are throughput bound (as opposed to latency bound), and it grows linearly with the size of the block. Given the current overlay network and 10 min. block interval, for blocks to propagate to 90% of peers within ten min., they estimate that the block size should not be more than 4 MB in size, which corresponds to at most 27 transactions/sec. Given the high bandwidth per node, it is surprising that the overall system bandwidth is so low. They suspect two reasons for this bottleneck. One, there are repeated messages for the same transaction, by gossiping about a transaction and then mining it. Second, there is no pipelining of messages for a transaction, which means the net latency is the sum of latencies of all links.

Weber et al. [80] studied the time to commit characteristics for Bitcoin and Ethereum. Across both platforms, the time to commit is significantly affected by the out-of-order arrival of transactions (child transactions before parents in case of Bitcoin and higher nonce before lower in case of Ethereum) and transaction fee included by the issuer (called gas price in Ethereum). It is assumed that the transactions are committed in the main chain with a high probability with six successive blocks in case of Bitcoin and 12 in case of Ethereum. Overall, the time to commit has a high variance ranging from minutes to days, which is unreasonable to be useful in

consortium blockchain networks that are targetted by Hyperledger Fabric.

6.8.3 Performance evaluation framework

Given the complexity of conducting end-to-end testing of such systems, both the research and product-development communities have recognized the need for a benchmarking tool that would interface with different blockchain platforms and benchmark workloads. Auh Dinh et al. [57] presented the *blockbench* framework for quantitative analysis of permissioned blockchain networks as a data processing platform. To gain insights into the inner workings of the blockchain platforms, they propose dividing the architecture into the following four layers: consensus layer for the consensus protocol, data model layer for the ledger and related data structures, execution engine layer for the runtime execution of smart contracts and application. They developed separate micro-benchmarks to gain insights into the performance characteristics of each layer.

Different blockchain platforms are developed with varying assumptions of trust in mind, which results in performance metrics that are defined inconsistently. Thus, it is hard to compare performance metrics across them. Hyperledger Caliper [23] is a benchmark execution platform that has been developed to address this challenge. Their goal is to present the performance evaluation measurements consistently across all platforms. It interfaces with Hyperledger blockchain projects such as Fabric, Sawtooth Lake, Iroha, and Composer.

Both frameworks mentioned above are available in open-source.

6.9 Future Work

Empirical analysis of blockchain networks is an exciting area of research. Although blockchain brings enormous promises, given the poor performance of mainstream blockchain networks such a Bitcoin and Ethereum, some of the potential adopters of

this technology remain skeptical. However, given the promising performance numbers of permissioned blockchain networks deployed in the field [81, 49], there is enormous interest and excitement in the community. After all, practitioners are always excited to see real metrics. From the perspective of performance modeling, we need to parameterize our models using datasets from setups that are the most representative of real-world usage. Also, we need to validate the model for a wide range of use-cases and configurations. We outline the following directions of work

1. Model parameterization for transaction-level parameters

   We find that the popular distributions do not seem to fit very well. One alternative to consider is phase-type distributions [82]. This could further improve our model validation results.

2. Model validation using cross-validation techniques

   Currently, model parameterization and validation (Section 7.4) is done using entire dataset. One could consider k-fold cross validation.

3. Experimental Setup in a Wide area network (WAN)

   Most of the research work on empirical analysis on Hyperledger Fabric V1 is done in a setup where all peers are deployed with a lab (our case, [58]) or within the same datacenter ([22, 41]). Such a setup would provide an environment comparable to deploying the Fabric network using IBM Blockchain service[9]. However, for security and data confidentiality reasons, many organizations would prefer to deploy their peer in their datacenter (or their preferred public cloud service provider). Thus it is critical to measure and analyze the performance in a WAN setting, where peers and orderers are located in different datacenters. It would significantly improves the security and fault-tolerance at the cost of performance. To utilize our model, measurements for model pa-

[9] https://www.ibm.com/blockchain

rameters need to be collected in some of the popular deployment settings and validated. Then our model can provide estimates for other similar deployment settings and use-cases.

7

Performance Modeling of Hyperledger Fabric V1

As the HLF project is evolving and maturing, it is imperative to model the complex interactions between peers performing different functions. Such models provide a quantitative framework that helps compare different configurations and make design trade-off decisions. In this chapter, we present a performance model of Fabric V1 using Stochastic Reward Nets (SRN) to compute the throughput, utilization and mean queue length at various peers as a function of various system parameters. Just like multiple subsystems of the HLF are "pluggable", we ensure the corresponding sub-models are pluggable as well. With this model, we can ask various *what*-if questions, such as

1. How does the throughput, utilization and mean queue length vary at each peer with an increasing transaction arrival rate?
2. If the peer validates transactions in a pipeline as opposed to batches, will the overall system throughput increase?
3. If there are multiple endorsing peers per organization, how much performance speed-up do we get?

The contributions of this research are as follows:

1. A comprehensive performance model of the Fabric V1 blockchain network. For Fabric's unique blockchain network architecture, it captures the key steps performed by each subsystem as well as interactions between them.
2. Analysis for critical *what-if* scenarios that system developers and practitioners care about.
3. Validate the model with a multi-node experimental setup.

This chapter is an extended version of our published work [75]. Some of the text, figures, tables in this chapter are reprinted, with permission, from "H. Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy Rindos. Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network). In *IEEE International Symposium on Network Computing and Applications (NCA)*, 2018."

7.1 SRN model of the system

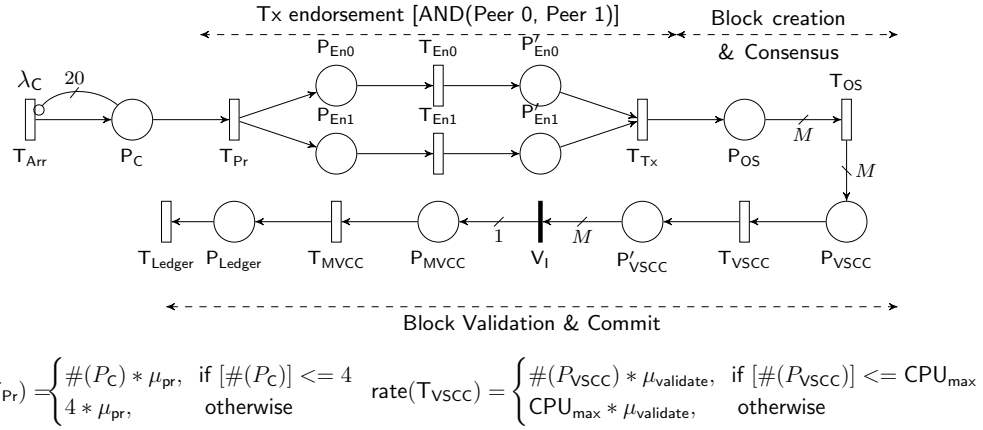


FIGURE 7.1: SRN model of Hyperledger Fabric V1 network

The SRN model for a single-channel Fabric network with one client, two endorsing peers (`AND()`) and one peer running the validation logic is shown in Figure 7.1.

Transaction requests follow a Poisson arrival process with rate λ_C. Maximum pending requests are capped to the no. of threads in workload generator (20 in our case). Client prepares the endorsement request and sends it to the endorsing peers (say peer 0 and peer 1) (transition T_{Pr}, which includes transmission time). Peers endorse the transaction (transitions T_{En0} and T_{En1} respectively). When the client receives a response from both peers, the client sends the endorsed transaction to the ordering service (transition T_{Tx}), indicated by a token deposited in place P_{OS}. After *block size* pending transactions (denoted by M), a block of transactions is created and delivered to the committing peers (transition T_{OS}). The committing peer first performs a VSCC validation all the transactions in a block in parallel, limited by the number of logical processors (cores/vCPUs) in the peer (CPU_{max}). Then it performs MVCC validation for all transactions serially (transition T_{MVCC}). Finally, all transactions in the block are written to the local copy of the ledger (transition T_{Ledger}). Both transitions T_{MVCC} and T_{Ledger} are captured at a block level. For reasons discussed in Section 7.5, we do not consider *block timeout* in this model. Note that we implicitly assume that all transactions are of the same complexity and independent of each other. In summary, the three phases of a Fabric transaction can be seen in the SRN model.

We obtain the metrics from our model as follows. The throughput of a transaction phase corresponds to the rate of the corresponding transition, using function `rate()` in SPNP [83]. E.g., the rate of transition T_{Ledger} signifies the block throughput of the system (multiply by M to obtain transaction throughput). The utilization of a transaction phase is computed by the probability that the corresponding transition in SRN is enabled, using function `enabled()`. For transitions with function-dependent marking rate (such as T_{VSCC}), the average utilization across all logical processors is computed using reward functions. The mean queue length of a transaction phase can be obtained by the number of tokens in the corresponding phase, using function

`mark()`. E.g., the mean number of tokens in place P_{OS} signifies the mean queue length at the ordering service.

7.2 Model Parameterization

We perform test runs for each set of configuration parameter values described in Section 6.4 and collect the log files, from which we derive the required output metrics. Depending on λ_C, each test run consists of 7k to 30k transactions, resulting in 7k to 30k samples for transaction level parameter values and 150 to 900 samples for block-level parameter values. To ensure that the parameter values are consistent across various groups, we perform Analysis of Variance (ANOVA) F-test [63] to see if there is a statistically significant difference in means, followed by multiple comparisons procedure using Tukey's Honest Significance Test [63, 84]. Thus, we derive our parameter values from a large dataset.

In our current work, we assume the firing time for all transitions is exponentially distributed. In our future work, we plan to do a distributional analysis and choose the best-fit distribution for each transition (preliminary results shown in Section 6.5). Since our validation results are good, we feel confident about our choice. With this choice, the underlying stochastic process is a continuous-time Markov chain (CTMC), and hence we can analyze our models using analytic-numeric solutions (Section 7.5). If the firing time for transitions were hypoexponential or Erlang or hyperexponentially distributed, then the the underlying stochastic process would still be a CTMC. Otherwise, the model solution could be obtained only using simulation techniques.

The parameter values for 'open' transactions are summarized in Table 7.1. Transitions T_{MVCC} and T_{Ledger} are measured at a block level since the measurements are too small for each transaction. Time to process messages at client (T_{Pr}), and time to prepare a block (T_{OS}) also includes the transmission time to the peer. Regarding

T_{OS}, although the current Kafka based ordering service does not have a notion of consensus, in future when Byzantine fault tolerance (BFT) consensus protocols are implemented, this transition will also cover time to consensus.

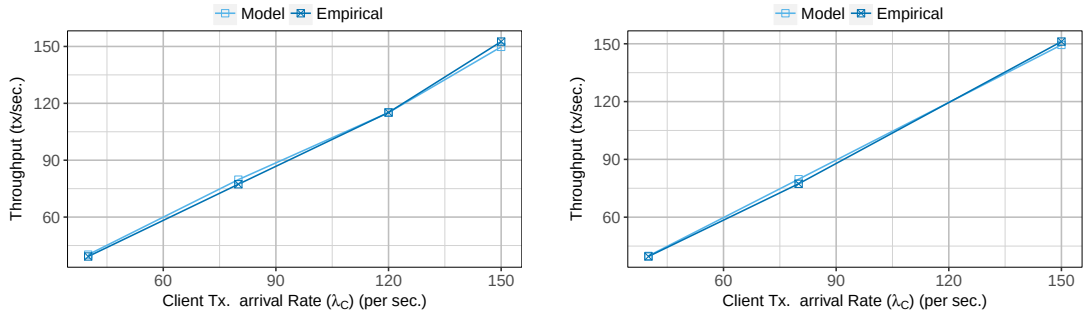
Table 7.1: Parameter values for SRN model transitions for 'open' transaction

Parameter (SRN transition)	Block Size	Mean time (ms)	Rate (ms^{-1})
Client processing (T_{Pr})	–	6.47	0.155
Endorsement (T_{En0}, T_{En1})	–	3.25	0.308
Transmit to Ordering service (T_{Tx})	–	5.22	0.192
Block creation and delivery (T_{OS})	40	75.74	0.013
	80	81.60	0.012
	120	93.56	0.011
VSCC validation (T_{VSCC})	–	2.52	0.397
MVCC validation (T_{MVCC})	40	2.56	0.391
	80	5.10	0.196
	120	7.20	0.139
Ledger write (T_{Ledger})	40	207.80	0.0048
	80	208.30	0.0048
	120	188.40	0.0053

The system-level operations that correspond to each SRN transaction are as follows. *Endorsement* includes transaction simulation (using application chaincode) and proposal endorsement (using ESCC system chaincode). Due to difficulties in measuring, it does not include the pre-processing stage that checks the transaction proposal header, the uniqueness of the transaction and the access control list (ACL). *VSCC validation* includes check whether transactions is properly formed and signed, validation of transaction with VSCC and endorsement policy. *MVCC check* includes validating the key-value pairs, range query and hashed key (see `state_based_validator.go` file in Fabric source code). *Ledger write* includes committing block to storage, state database and history database (blockchain).

7.3 Model Validation

Let us analyze the overall system model (Figure 7.1) for a network using `AND()` endorsement policy with $CPU_{max} = 4$ for VSCC validation and various block sizes.



(a) block-size = 40 (b) block-size = 80

FIGURE 7.2: Model validation comparing overall system throughput

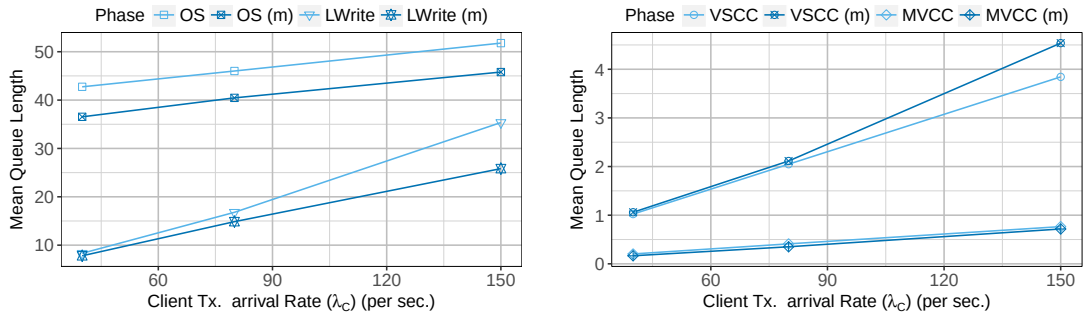


FIGURE 7.3: Model validation comparing mean queue length at various transaction phases with empirical measurements (m) (block-size = 80)

Unfortunately, the underlying Markov model has a huge space-space, and we could not solve the full system model using analytical-numeric solution. Hence we take the simulation approach using SPNP.

We first validate our model by comparing the overall throughput computed from the model with the empirical results. In Figure 7.2, we plot the results for block sizes 40 and 80. Thus, we validate our model for the output metric that is the most intuitive to the user.

Next, we validate our model by comparing a critical system-internal metric which is 'mean queue length'. We chose 'mean queue length' since it was reliable to measure and provides intuitive insights into the system performance. we compute the mean queue length at the ordering service (OS) and critical processing stage within a peer

(VSCC validation, MVCC validation and Ledger Write (LWrite)) and compare it with the empirical results. We validated our results for different client arrival rates and different block sizes.

Our results are shown in Figure 7.3. We observe that all measurement results (marked as (m), in darker color) are comparable to the model results (lighter color) at different client arrival rates. The results are similar for other block sizes as well. Hence we consider our model validated. Unfortunately, we were not able to reliably measure the queue length at the endorsing peer from the peer logs. However, since the results match at four other measurement points, we leverage its results from our model.

7.4 Overall system analysis

From our model results, let us visualize the utilization and mean queue length (Figure 7.4) at each transaction phase with increasing λ_C. We compute the utilization for client processing and transmission to ordering service as well. We find that the transmission from client to the ordering service is a performance bottleneck with a sharply rising utilization, followed by the endorsing peer. The queue length at the ordering service/ledger write is expected to be large since it is waiting for *block size* transactions before generating/committing a block.

In our lab setup, at high client arrival rate, transactions tend to timeout (presumably due to queuing delays) and fail, although peers or the ordering service is not particularly busy. It also explains that the transmission time of endorsed transactions to the ordering service is a performance bottleneck. From our measurements, it is hard to say whether the delay is due to network transmission time or due to internal queuing at the ordering service. Assuming that this delay is not a problem in another setup (replace T_{Tx} with an immediate transition), let us compute the maximum throughput possible in this network. In our model, we keep increas-

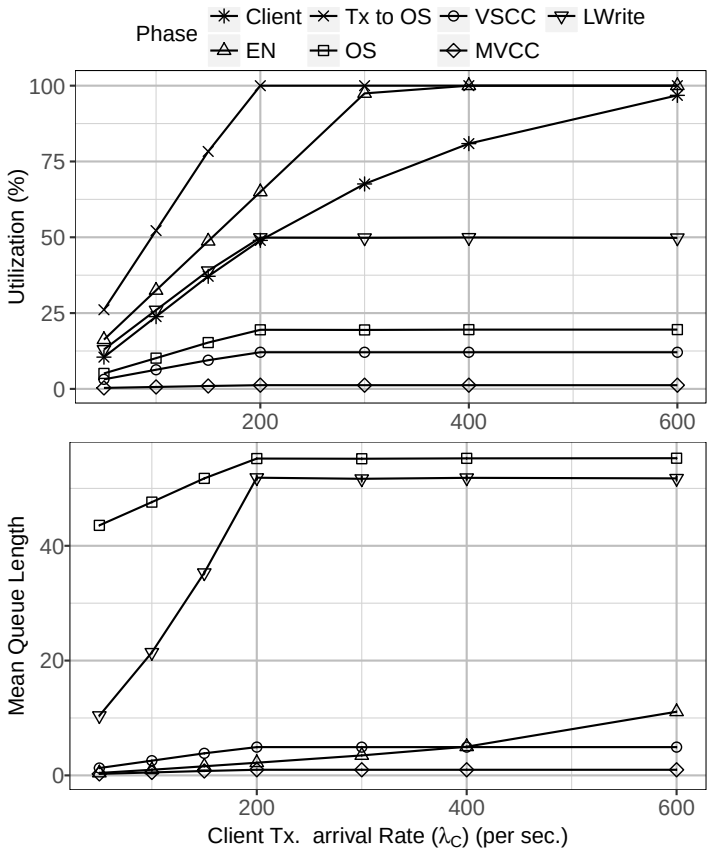


FIGURE 7.4: Utilization, mean queue length at various transaction phases (block-size = 80)

ing the client transaction arrival rate until the utilization at any transaction phase reaches 90%. We also consider a scenario where there are multiple endorsers per org. ($\mathrm{En}_{\max}$), using marking-dependent firing rate for transitions $\mathrm{T}_{\mathrm{En0}}, \mathrm{T}_{\mathrm{En0}}$. From our results in Figure 7.5, we see that the maximum throughput increases significantly as the block size increases, albeit with an increase in mean latency (not shown). When $\mathrm{En}_{\max} = 1$, the bottleneck is the endorsing peer. When $\mathrm{En}_{\max} = 4$, the bottleneck is the ledger write. Thus the maximum system throughput can tremendously increase (especially for larger block sizes like 120) if there are multiple endorsers per org. This is feasible in systems where concurrent transactions are touching independent set of key-value pairs and thus can be processed in parallel.

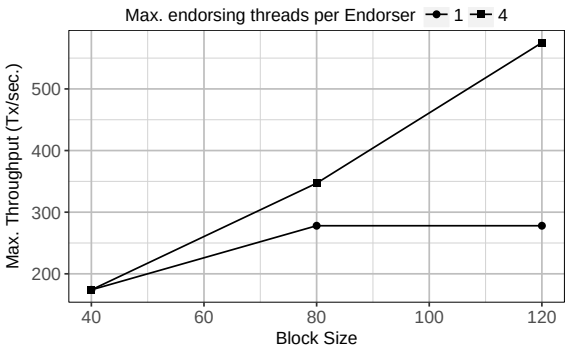
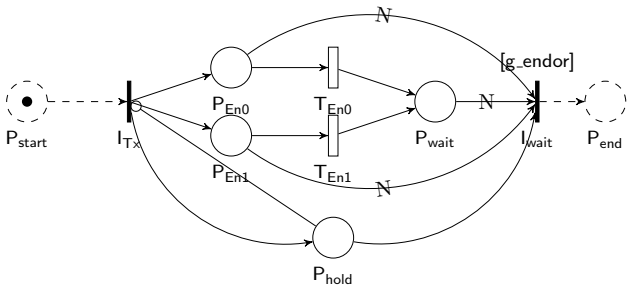


FIGURE 7.5: Impact of block size, multiple endorsers on max. throughput

7.5 Model Analysis

In the following three subsections, we analyze the subsystem corresponding to each transaction phase. All analysis in this Section is done using analytic-numeric solution in SPNP.

7.5.1 Endorsement Process



$$[\text{g\_endor}]_{\text{AND}} = \begin{cases} 1, & \text{if } [\#(P_{\text{wait}})] > 1 \\ 0, & \text{otherwise} \end{cases}$$

$$[\text{g\_endor}]_{\text{OR}} = \begin{cases} 1, & \text{if } [\#(P_{\text{wait}})] > 0 \\ 0, & \text{otherwise} \end{cases}$$

FIGURE 7.6: Generalized SRN model to capture AND/OR endorsement policy between two peers

The client node is responsible for seeking endorsements on the transaction it is proposing such that it satisfies the endorsement policy. The endorsement policies are monotone logical expressions that evaluate to TRUE or FALSE. E.g., endorsement policy `OR(Org1, Org2)` means that endorsement from any peer from Org1 or Org2 would suffice. An endorsement policy can be expressed as an arbitrary combination

of AND, OR, and k/n expressions, such as `OR(Org1, Org2) AND (2/3 of Org3, Org4, Org5)`.

The endorsement part of the model in Figure 7.1 represents two peers in an `AND()` policy; thus the client waits for a response from both before forwarding the transaction to ordering service. This model can be easily extended to represent `AND()` policy for more peers. To model the `OR()` and `k/n()` policies, we flush tokens from places that did not fire, using variable-cardinality arcs (called `viarc()`) in SRNs, shown by arcs with Z sign in Figure 7.6. Also, a guard function is used for the immediate transition $\mathrm{I_{wait}}$, which is written similar to the endorsement logical expression. Thus, complex endorsement policies can be captured easily by extending the net as shown in Figure 7.6 (without the dotted part) and plugged in the overall system net in Figure 7.1.

The endorsement process adds significant latency to a transaction. First, the chaincode executes in a separate Docker container at each peer, adding a reasonable performance overhead. Second, the client needs to wait for endorsement response from multiple peers to satisfy the endorsement policy. Let us analyze the mean time to complete endorsement for different endorsement policies in Table 7.2. We assume the time to endorsement at each peer is exponentially distributed with rate 0.308 per ms.

Table 7.2: Mean time to endorsement (MTTE) for different policies

Endorsement policy	MTTE (ms)
OR (2 peers)	1.623
OR (3 peers)	1.082
AND (2 peers)	4.870
AND (3 peers)	5.952
AND (4 peers)	6.764
AND (10 peers)	9.510
AND (20 peers)	11.681
2/3 peers	2.706
6/9 peers	3.233
12/18 peers	3.393
OR (2 peers) AND (2/3 of 3 peers)	3.193

Unfortunately, the underlying state-space for this model increases exponentially with the number of peers. Assuming time to endorsement at each peer follows the same distribution, we can consider an alternate model where a single place represents all the peers currently working on the endorsement, which is an input place to a transition with marking-dependent firing rate. For such a model, the state-space would increase linearly with the number of peers. We leave the analysis for future work.

In the current release of Fabric Node SDK, the client waits for replies (or timeout) from all endorsing peers before the client prepares the endorsement message for the ordering service. Thus we cannot validate our full-system model with different endorsing policies. A new feature[1] will address this limitation.

7.5.2 Ordering Service

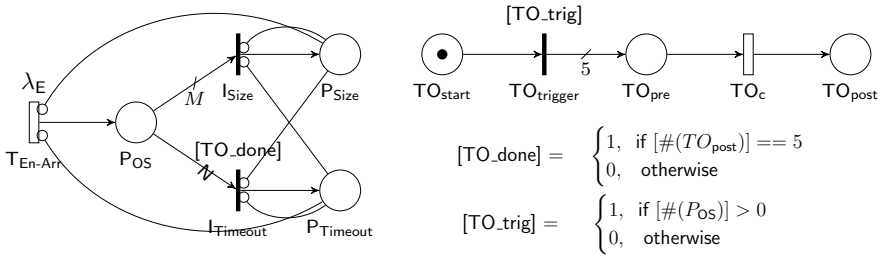


FIGURE 7.7: SRN model of the ordering service considering block timeout and block size constraints

The ordering service receives endorsed transactions from the client, orders them and creates a block of transactions based on block timeout or block size. We would like to assess the probability that a block was generated due to block size or block timeout, given an arrival rate of endorsed transactions (λ_E), block size, and block timeout.

[1] `https://jira.hyperledger.org/browse/FAB-10672`

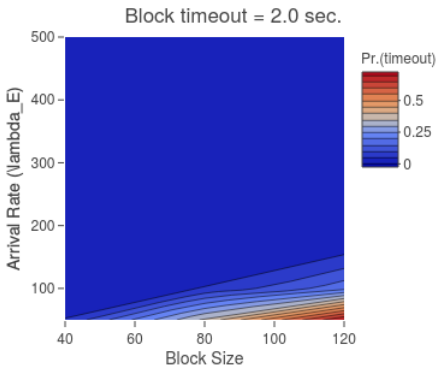


FIGURE 7.8: Probability of block generated due to timeout as a function of endorsed transaction arrival rate and block size

Let us consider the SRN model in Figure 7.7. The number of tokens in place P_{OS} represents the pending transactions. A token is deposited in place P_{Size} or $P_{Timeout}$ depending on whether the block is created due to size limit or timeout. Since block timeout is deterministic, we approximate this using Erlang distribution [85, 86], which is shown as a separate net. The immediate transition $TO_{trigger}$ is enabled when there is at least one token in place P_{OS}. We consider a 5-stage Erlang distribution where each stage fires with rate 5/(block timeout).

Figure 7.8 presents the contour plot of the probability that a block created due to timeout condition for timeout = 1.0 sec. For smaller block size, the blocks are generated due to size limits even at low arrival rates (λ_E). As block size increases, a higher endorsed transaction arrival rate (λ_E) is required to skip the timeout condition from our model. Thus, we can skip the additional logic to account for the timeout condition, thereby significantly reducing the size and complexity of the full system model (Figure 7.1). We repeated the analysis by considering a 25-stage Erlang distribution for a more accurate approximation of the determistic timeout, but the results were similar.

In Appendix D, we have shared the SRN code for model (Figure 7.7) and the R code to generate the contour plot (Figure 7.8)

7.5.3 Block Validation & Commit

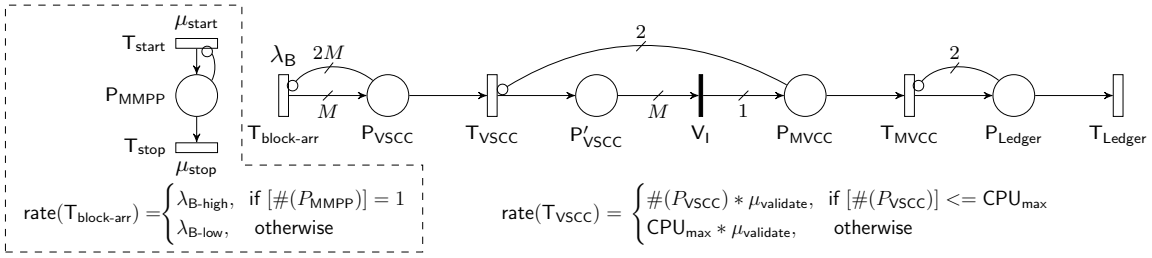


FIGURE 7.9: SRN model of a *committer* peer, validating and committing blocks of transations

Let us analyze the performance of the committing peer using the SRN model in Figure 7.9. Let us assume that blocks arrivals (from the ordering service) follow a Poisson arrival process with rate λ_B, each of size M. To limit the size of the underlying Markov model, we consider a maximum queue length of two blocks in each phase.

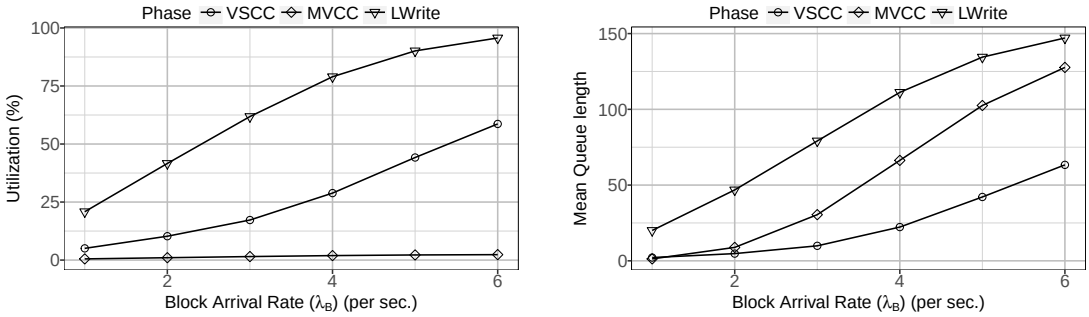


FIGURE 7.10: Utilization, mean queue length at various block validation stages for model with block-size = 80, VSCC validation $\text{CPU}_{\max} = 4$

Let us analyze the model with block size = 80, VSCC validation $\text{CPU}_{\max} = 4$. As the block arrival rate increases, the net throughput will increase, however, the utilization at each stage will increase as well; also the rejection probability of an incoming block (due to two pending blocks in the queue) will increase too. From the results shown in Figure 7.10, we find that the utilization and mean queue length of

Ledger write increases significantly with increasing block arrival rates; thus it is the performance hotspot within the committing peer. The mean queue length of ledger write and MVCC validation is on the higher side since it waits for a block worth of transactions.

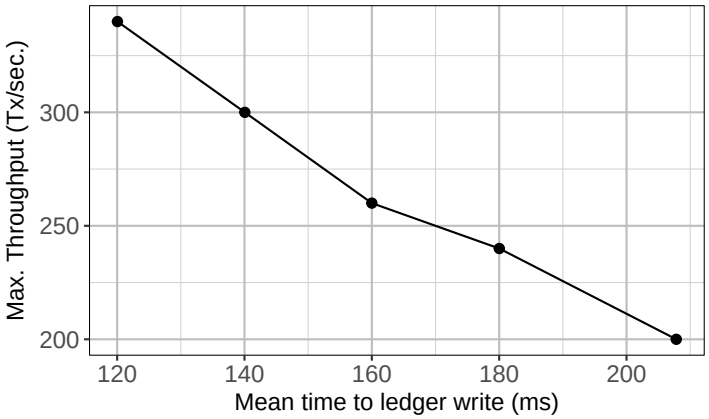


FIGURE 7.11: Sensitivity of the *committer* peer with mean time to ledger write

From the three parameters in the committing peer model, which parameter has the most influence on the output metrics? To find out, let us perform *sensitivity analysis* [28, 87]. If we had an analytical expression to capture the output behavior of the system as a function of the input parameters, we would have computed partial derivatives of the output parameter with respect to input parameters and derived sensitivity functions for each input parameter. In our case, the committing peer model has a large state space (1449 states when the block size is 40) and hence it is challenging to derive a closed-form expression. Rather, we vary each input parameter, compute its influence on the output metric, and rank the parameters [87]. For our output metric, we consider the maximum throughput of the committing peer, which we discussed earlier in Section 7.4. We find that the ledger write has the maximum influence on the system performance and is our performance bottleneck. Figure 7.11 shows how the maximum throughput of committing peer varies with changing the

mean time to ledger write.

Let us consider two scenarios of interest to the system architect.

Bursty arrival of blocks

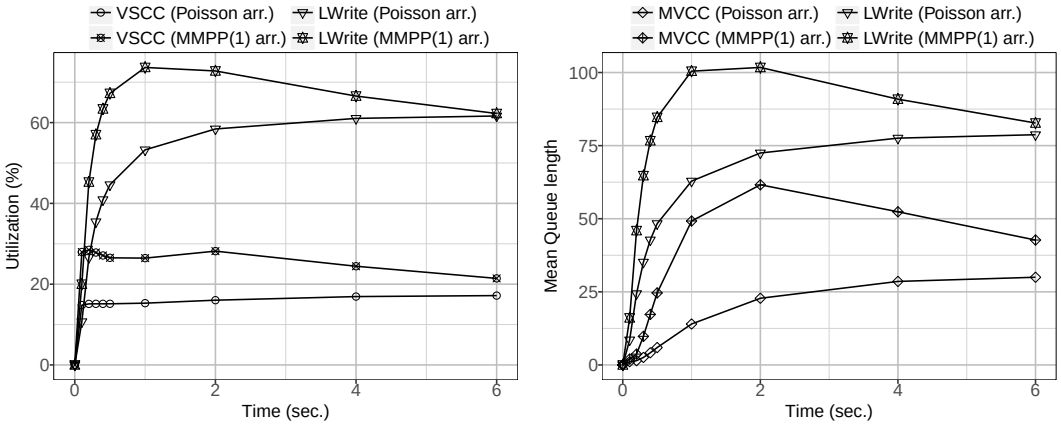


FIGURE 7.12: Transient analysis for utilization and queue length of various stages in a committer with block-size = 80, VSCC validation $\text{CPU}_{\max} = 4$

Let us consider a scenario in which the committing peer faces a bursty stream of blocks, consisting of periods of high and low mean block arrival rates. To capture this, we model the input arrival process using Markov modulated Poisson process (MMPP) [85, 86]. Let us consider the extended SRN model (shown in dashed box) in Figure 7.9 with block size M. When there is a token in place P_{MMPP}, blocks arrive at a high rate (say 6 blocks per sec.) for the average fraction of time $\frac{\mu_{\text{start}}}{\mu_{\text{start}}+\mu_{\text{stop}}}$ and low rate otherwise (say 2.25 blocks per sec.). The mean time to start of burst mode is eight sec. ($\frac{1}{\mu_{\text{start}}}$) and it lasts for a mean time of 2 sec. ($\frac{1}{\mu_{\text{stop}}}$). To compare the results from the model with non-bursty arrival rates, we consider the same average arrival rate. Thus,

$$\lambda_{\text{B}} = \lambda_{\text{B-high}} * \frac{\mu_{\text{start}}}{\mu_{\text{start}} + \mu_{\text{stop}}} + \lambda_{\text{B-low}} * \frac{\mu_{\text{stop}}}{\mu_{\text{start}} + \mu_{\text{stop}}}$$

We compare the results for the MMPP arrival starting in burst mode (MMPP(1)) with that of a model with Poisson arrival rate λ_{B}= 3 blocks per sec. From Figure 7.12,

we find that the utilization of Ledger write and VSCC validation stages jumps for MMPP(1) arrival in the first few seconds. There is a significant jump in the mean queue length of both MVCC check and Ledger write stages. Overall, the VSCC validation stage seems to absorb the shock of bursty arrivals quite well. However, the utilization and mean queue length of ledger write stage is affected the most, and hence it is critical from a performance perspective. Keep in mind that the blocks are delivered from the ordering service in a sequence. Thus we need to ensure that the input queue is sufficiently large and that the committing peer can process the blocks fast enough. Requesting for a block again fom the ordering service can significantly slow down the peer.

Pipeline model

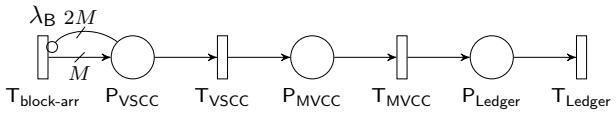


FIGURE 7.13: SRN model of a committing peer in pipeline order

To improve the performance of a committing peer, the authors in [22, 41] proposed

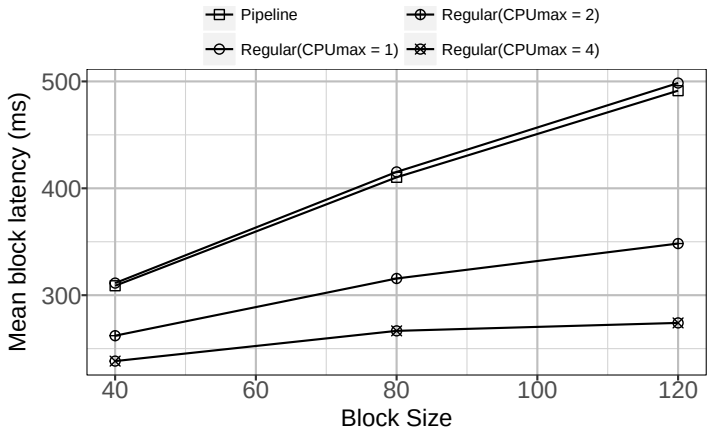


FIGURE 7.14: Mean latency to complete block validation & commit for pipeline model vs regular model

a pipeline architecture, where each transaction passes through various stages in a pipeline, as opposed to the current architecture where transactions pass through each stage in blocks. We assume such a system would have only one logical processor for the VSCC validation. We assume the same parameter value for VSCC validation and parameter value divided by block size for MVCC, LWrite. We compute the maximum throughput for this pipeline model (Figure 7.13) for various block sizes. Surprisingly, the maximum throughput is comparable to that from the regular model with $\text{CPU}_{\text{max}} = 1$. However, mean queue length at MVCC validation is slightly larger, and that at ledger write is smaller (not shown). The rest of the metrics are comparable.

Let us compute the latency for a block of transactions using a modified version of models in Figures 7.9 and 7.13, by removing the transition $\text{T}_{\text{block-arr}}$ and considering M initial tokens in place P_{VSCC} and an output place Done from transition T_{Ledger}. When M tokens are deposited in place Done, it signifies completion of a block. We compute the mean block latency for the regular model and pipeline model for different block sizes (Figure 7.14). As expected, the mean block latency improves slightly in the pipeline model compared to regular model with $\text{CPU}_{\text{max}} = 1$. As we add more CPUs in the regular model, the latency reduces further.

We perform sensitivity analysis for the pipeline model for mean block latency to the VSCC validation rate. We find that the maximum throughput results are comparable with that of the regular model with VSCC validation $\text{CPU}_{\text{max}} = 2$, 4 and so on. Overall, pipeline architecture would slightly improve the block latency, but would not improve the maximum throughput and other performance metrics compared to the conventional architecture.

7.6 Discussion

7.6.1 Largeness of stochastic model

We used a high-level formalism, i.e., SRN to model the complex interactions of a Fabric network. A drawback of this approach is that the underlying stochastic model generated is huge. We are unable to generate the underlying state-space for the full-system model since it is infinite. Hence we took the simulation approach (ref. Section 7.4). For the committing peer model (ref. Section 7.5), we considered a maximum queue length of two blocks at each stage. This truncates the state-space of the underlying model. We solved this model using analytic-numeric solution. Interestingly, we find that the model state-space size increases linearly with the block size.

The largeness problem limits our ability to answer questions regarding the scalability of Fabric networks. In our future work, we would consider a hierarchical approach and fixed-point iterative solution techniques to mitigate this issue [28, 88].

7.6.2 Limitations of our model

A limitation of our model is that we cannot compute the latency at a specific transaction throughput. From our model, we can estimate the mean latency of a block of transactions (ref. Section 7.5.3). However, this does not account for any queuing delays at various nodes when the system is "loaded." Along those lines, this model also cannot capture transaction failures due to timeout (mainly due to queuing delays). In our future work, we plan to compute the response time distribution of a transaction using a state-space modeling approach, in lines of work presented in [89].

7.6.3 Threats to validity

One threat to the validity of our results is that we did not use custom VSCC. Custom VSCC lets the user define additional (business) logic to validate the transactions.

Since this is in addition to the endorsement policy signature validation done during VSCC validation, we feel a higher mean parameter would capture this easily. Note that authors in [22] used custom VSCC but not the authors in [41]. Another threat is that our nodes were connected in a LAN setting, as opposed to a wide-area network (WAN) that would be expected in the real world. In our future work, we plan to replicate our results by running the Fabric on a cloud service like Amazon Web Services (AWS).

7.7 Related Work

Decker and Wattenhofer [54] presented a simple model to compute the stale block generation rate in the Bitcoin network, which takes into account the block generation and the block propagation process of bitcoin. From their empirical analysis of 10,000 blocks conducted in 2013, they estimated the block generation rate is exponentially distributed with a mean of around 633 sec (10.5 minutes). For an ensemble of block sizes and (source, destination) pairs, time to propagate a block is also found to be exponentially distributed with a mean of 11.37 sec (albeit after correcting some clock skew). For this single dataset estimated, the empirical stale block rate is comparable to that from the model (around 1.78%).

Gervais et al. [55] model and analyze the security and performance aspects of Proof-of-Work (PoW) based blockchain networks. For performance modeling, they develop a simulator that mimics the block mining and propagation process as a function of block interval, block size and block propagation mechanism. However, its output metric is stale block rate, which is not beneficial for our analysis. In summary, in Proof-of-Work (PoW) based blockchains, the goal is to find the optimal block interval time that maximizes the throughput and minimizes the stale block rate (and hence reduce security risk). In case of BFT-based blockchains, the goal is to ensure that the block consensus and block execution process take much lesser time

than the block generation interval (default batch timeout is 1 second, and batch size is 500).

Papadis et al. [90] developed stochastic models for PoW based blockchain networks to compute the block growth rate and stale block rate as a function of the propagation delay and the hashing power of the compute nodes. Using the assumption that the block generation at each peer is a Poisson process [91] and the block propagation delays follow an exponential distribution [54], the stochastic process tracking newly mined or received blocks at ledger in each peer is Markovian. They truncate the model state space by assuming the block generation rate is much slower than block propagation delay and provide the stationary distribution and asymptotic expansions for the long-term growth rate for this model. For systems with 2 nodes and 5 nodes, the empirical results are comparable to that from the model.

Kocsis et al. [92] motivated the need for performance modeling of blockchain systems. They presented an outline for a measurement based approach to charcterize performance of blockchain systems, which can be used to develop quantitative models for analysis.

In our work described in Section 5, we modeled and analyzed the practical Byzantine fault tolerance (PBFT) consensus process of Fabric v0.6 [29]. Since the architecture of Fabric V1 has significantly evolved compared to release v0.6 [12], the models discussed in this chapter are not applicable to release v0.6.

7.8 Conclusions

In this chapter, we developed stochastic models for a popular distributed ledger platform called Hyperledger Fabric. We analyze the full-system as well as the subsystems corresponding to each transaction phase in details. We collect data from a Fabric setup running realistic workload to parameterize and validate our models. Our models provide a quantitative framework that helps a system architect esti-

mate performance as a function of different system configurations and make design trade-offs decisions.

For an incoming block, since the committing peer validates transactions (VSCC) in parallel, there is a significant performance improvement if the committing peer is deployed on a system with a large number of CPUs. It can significantly reduce the queue length at the VSCC validation as well as let system absorb the shock of a burst arrival of blocks. However, it is not the performance bottleneck for the committing peer, since ledger write has the highest utilization.

We also analyzed two hypothetical scenarios: peers endorsing multiple transactions in parallel (same as multiple endorsing peers per org.) and pipeline architecture for the validator. Transaction endorsement parallelization can significantly reduce the endorser queue length and increase the maximum throughput of the system if the endorsement process is the performance bottleneck. The pipeline architecture provides around 1% improvement in mean block validation latency but offers no other performance benefit.

8

Model Verification & Validation

In this chapter, we summarize our approach of verification and validation of the models presented in this thesis. Under the desired system and model use-case, model verification is the process of ensuring that the model conforms with the system understanding, and model validation is the process of ensuring that the model results conform with the empirical results [93]. Model verification and validation is an iterative process which is performed until there is no further improvement. This process can be summarized in Figure 8.1.

All the models discussed in this thesis are stochastic models. These models are defined using a high-level paradigm like Stochastic Reward Net (SRN). We use the analytical package Stochastic Petri Net Package (SPNP) [65] to automatically generate the underlying stochastic models. These models are solved using analytical-numeric solutions (Sections 7.5) or using simulation techniques (Section 7.4, Chapter 5) provided by SPNP.

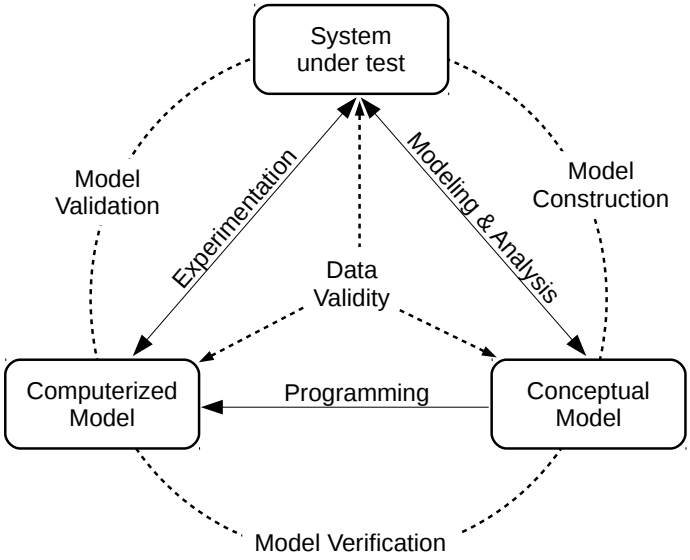


FIGURE 8.1: Model verification & validation process

8.1 Steps for Model Verification & Validation

Following the methodology proposed by Naylor and Finger [94] and summarized in [28], we outline the following three steps to perform model verification and validation.

8.1.1 Face validation

Face validation is a process of discussing the models with the field experts such that all parties agree about the model. This discussion starts with the conceptual model and then translated into the operational model during the discussion itself.

Since Hyperledger Fabric is developed in the open-source and is well documented, in both systems (v0.6 and V1), we reviewed the system specifications extensively before we met the field experts. For v0.6, first, we brainstormed the system extensively within our team (co-authors in [29]) to find a problem of interest (which was PBFT consensus process) and then prepared the SRN model in a few iterations. Then we met two experts at IBM RTP face-to-face to discuss our conceptual understanding of the system followed by a walk-through of the SRN model. Thus we verified our model. However, this process still missed out one subtle but important aspect of the

system, which is time to process incoming consensus. This aspect was found to be the performance bottleneck in [57]. Without this transition, our output validation failed.

For V1 which is a more complex system, I acquired a good hand-on understanding of the system over several months of my internship at IBM Research - Zurich, before I started modeling it. I prepared the first version of the model and discussed with my team (co-authors in [75]). Then I reached out to two field experts at IBM Research - India. Since they were familiar with Petri Nets modeling paradigm, we were able to discuss the models over emails and phone. I discussed SRN models directly with them, and we found some mistakes in the Client SDK logic and fixed them.

8.1.2 Input-output validation

Input validation ensures that the input data used is accurate, unbiased, complete, and appropriate to the physical system [28]. In our models, the inputs refer to the firing time of all SRN model transitions, which includes the firing time distribution and its parameter values.

For v0.6, we collected data from a blockchain network deployed in the IBM cloud. This is an expected use-case for running blockchain networks even in V1 and in the future. The data collection and analysis process was tedious, and hence we analyze only 50 randomly chosen blocks across a week of data collection. We perform distribution analysis for all parameters and solve the model using simulation techniques.

Since we had difficulty changing PBFT and Fabric parameters in v0.6, hence for V1 we decided to setup our blockchain network in our lab. Since we had full access to the datasets, we collected large datasets and wrote our parsers and scripts to analyze the data. We performed the distribution analysis for all parameters but found it was hard to fit distributions for transition-level parameters. For simplicity, we assumed an exponential distribution for all parameters. Note that for block-level parameters,

we found exponential distribution to be a good-fit distribution.

Output validation is ensured by comparing the model results with the empirical results. For v0.6, we have one output metric (mean time to consensus) along with confidence interval, which we compare with the observed results and report the percentage error. For V1, we have multiple output metrics, out of which we chose four output metrics (mean queue length at four stations) for validation (explained in Section 7.4). To validate this for a different combination of policy conditions [94], we chose two system parameters, client arrival rates, and block size. Due to a large number of data points involved, we present the results graphically.

8.1.3 Validation of model assumptions

We need to ensure that the model assumptions are clearly identified and documented. For each assumption, we should either be able to theoretically prove that it is correct or perform statistical hypothesis testing to validate them. In our stochastic models, our assumptions are either about the model structure or about the firing time of the transitions.

Regarding model structure, any assumption we made was verified using code reviews or discussing with field experts. One example in HLF is whether a single peer endorses transactions in a sequence or parallel. From code review, it was clear that it endorses in a sequence. However, from the log files, it was not as obvious. We were finally able to verify with the experts that each peer endorses in a sequence. Given the way the log entries were captured for the endorsement process, it explained why we had challenges measuring mean queue length at the endorsing peer.

Regarding firing time of transitions, for v0.6 we assume the best-fit distributions for each SRN transitions. For V1, we assumed an exponential distribution for all transitions since we encountered challenges in fitting standard distribution functions for transition-level parameters (Ref. Chapter 6). Future work should focus on vali-

dating the model using better-fitting distribution functions.

8.2 Threats to validity

8.2.1 Model Logic and Code

The first threat is the validity of model logic and associated SPNP code. In addition to the discussion in the above section, for both HLF v0.6 and V1, the SPNP code was written independently by two authors (me and another co-author in each project) and we verified the output results, thus verifying the code. We also extensively brainstormed our model and analysis results. During one such session, for HLF V1, we figured out that transition T_{Pr} (Ref. Figure 7.1) could have marking-dependent firing rate.

8.2.2 Model Parameters

The next threat is the measurements of the model parameter values. In our models, the parameter value corresponded to the time difference between the events of when a task was started and completed. These events are captured in the peer/orderer log files. Although most of these events were intuitive to identify, some of them had multiple possible options. To understand the event messages in depth, I extensively reviewed the software code along with the log files from test runs and eliminated a few obvious ones. For others, we collected and analyzed datasets for all possible combination of time-between-events. From the summary statistics and box-plots, we could identify time-between-events that might inherently include queuing delays and eliminated them. Thus, we ensured that we collected the best-possible datasets for our parameter values. It is worth noting that the measurements involving transmission time across different nodes (like time to transmit endorsement message) had high variance and were inconsistent across different system configuration settings. This is pronounced only in transmission of individual transactions rather than in a

block of transactions. We feel this happens because transaction message size was small and hence its latency-bound rather than throughput-bound. Further research work is required to ensure the accuracy of such measurements.

Another important aspect is the number of samples collected for our analysis. Each test run lasted for 240 sec., ensuring the system achieves a steady-state. We also clip the first 20 and last 20 sec. of our dataset as ramp-up and ramp-down phase. We ensured the samples were collected across different configuration settings. For parameterization, we merged the datasets that showed no statistically significant difference in their means (Ref. Section 6.5).

8.2.3 System configuration settings

Another threat to validity is the configuration of our blockchain network setup. For HLF V1, we deployed the network using the community-recommended approach [76]. One limitation of our setup is that the nodes were connected in a LAN setting, as opposed to a WAN. In an ideal world, each organization would setup a peer in their own datacenter. Given the complexity of setting up and running a Fabric network, it is likely that many organizations would prefer a cloud service such as IBM Blockchain to deploy their blockchain network, and hence our assumption is not as far from reality. Another threat is that we did not use custom VSCC. As discussed earlier, we feel the implications of custom VSCC can be captured easily in the corresponding transition parameter and would not require any model rework.

9

Conclusions

9.1 Conclusions

In this dissertation, we study Hyperledger Fabric from a performance perspective. Hyperledger Fabric is an open-source implementation of the distributed ledger platform for running smart contracts in a modular architecture. It is gaining popularity with 400+ proof-of-concept and production implementations across different industries and use-cases. Our main contribution is a stochastic model that capture critical steps performed by each subsystem and the interactions between them. We also provide a detailed empirical analysis of model parameterization and validation. Blockchains have spawned a new era of systems development, and this thesis sets the foundation for future research in modeling and analysis in this area.

The models and analysis in this thesis are useful for both the system developers and the architects deploying Fabric in the field. System developers gain insights into the inner-workings of the system that provides guidance on which subsystems and functions to improve and its implications on the system performance. Architects can use the models to estimate performance as a function of different system

configurations and make design trade-off decisions.

We developed models for two different releases of Fabric, viz. v0.6 and V1. Both releases support a different architecture. HLF v0.6 follows a traditional state-machine replication architecture similar to many other blockchain platforms. We developed a detailed and scalable model for the PBFT consensus process, from which we estimate the "mean time to complete consensus." To parameterize and validate our model, we created the Fabric network using the IBM Bluemix service. We run a production-grade IoT application to generate a significant workload. We used the data collected in the log files to parameterize and validate our models. For four peers, we find the solutions from the SRN model are comparable to the empirical results, with a relative error of 7%. Using the validated SRN model, we analyze the PBFT consensus process up to 100 peers and find that the mean time to consensus increases by 5.34 times for 100 peers compared to that for four peers if the transmission delays are of the same order of magnitude as the processing and queuing delays. However, in a real-world scenario where peers are geographically dispersed, and transmission delays are significant, the percentage increase in the meantime to consensus would not be as significant for a large number of peers.

HLF V1 follows a novel *execute-order-validate* architecture where each transaction undergoes three phases: endorsing, ordering, and validation. We develop a detailed model using Stochastic Reward Nets (SRN) from which we can compute the throughput, utilization, and mean queue length at peers corresponding to each phase. To parameterize and validate our model, we setup a Fabric network in our lab and run a realistic workload using Hyperledger Caliper. We summarize our findings as follows:

- *Endorsing* - Endorsing policies that require endorsements from a large number of organizations can add significant latency. E.g., `AND()` policy across many

peers. However, its impact on the throughput can be mitigated if there are multiple peers within an organization for transaction endorsement, effectively parallelizing endorsements within an organization.

- *Ordering* - At the ordering service, waiting to prepare and prepare the block are the performance bottlenecks of Fabric V1 network. Its impact on throughput can be mitigated by using a larger block size, albeit with an increase in latency.

- *Validation* - Transaction validation check using VSCC is a time-consuming step, but it is embarrassingly parallel; Hence its performance impact can be easily mitigated by using peers with a large number of cores/CPUs. It also helps the validating peer absorb the shock of a burst arrival of blocks. We also analyzed a pipeline architecture for committing peers but found no significant performance improvement.

9.2 Fabric Performance Management Infrastructure

Stochastic models developed in this dissertation can be utilized in a Fabric network management infrastructure as shown in Figure 9.1. A Fabric network is expected to be deployed in the cloud, where peers for different organizations are deployed within the same data-center or different data-centers. Monitoring infrastructure collects required data from all the peers, ordering service, and SDK nodes. This data is used to parameterize the model. The results from the models (such as expected throughput, latency, mean queue length at each peer) can be validated with the monitoring data. The results of model validation are useful to tune the parameterization, like considering alternate parameter models, so that the model estimates improve. The result from the model can be used for analysis for short-term (like next several minutes or hours) or can be used for long-term (like weeks or months). The results are shared with the administrators as well as with an automated network manager. The man-

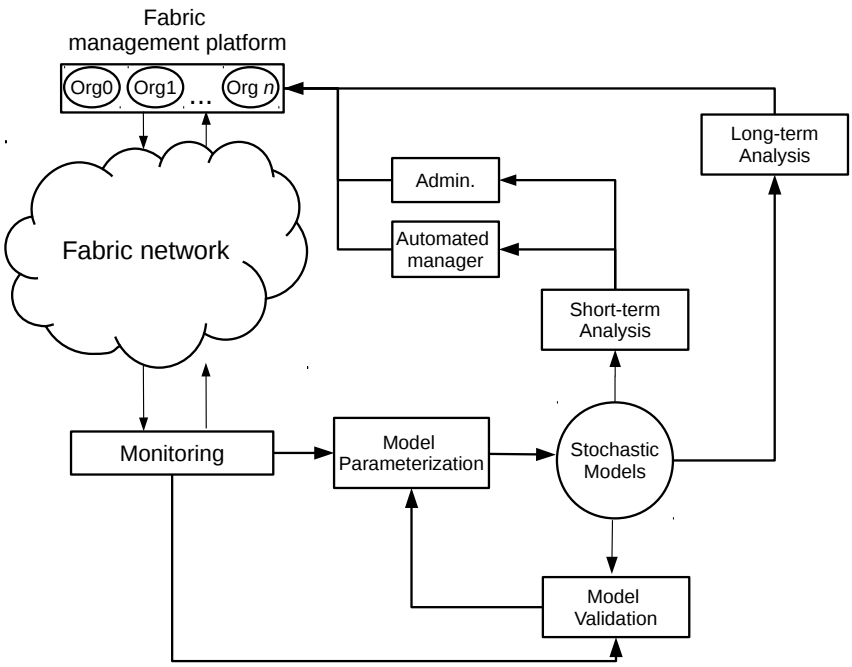


FIGURE 9.1: Fabric Network management infrastructure

agement platform is co-owned by different organizations, ensuring that the network control is decentralized.

Compared to a traditional cloud infrastructure, there are additional challenges involved in managing permissioned blockchain infrastructure. First, the monitoring infrastructure needs to be co-owned by multiple organizations, since a different organization owns each peer. One needs to ensure that the monitoring infra maintains privacy and confidentiality of transactions. Second, the decisions that affect the entire network (such as block size) need to be agreed upon by all parties using some consensus approach.

An example use-case is as follows. Suppose a consortium Fabric network needs to guarantee a specific service-level agreement (SLA) for transaction latency. If the Fabric network is experiencing higher than expected latency, it could be because any of the three transaction phases are taking longer than expected. If the latency increase is in the endorsing phase, the network manager can consider deploying more

endorsers in each org. or consider changing the endorsement policy (after necessary agreement from all orgs.). If the ordering phase is taking longer than expected, the manager can consider increasing the number of Kafka brokers, or decreasing the block size (yet ensuring the throughput SLAs are met). If the validation phase is taking longer than expected, the manager could recommend peers to live-migrate to a system with better hardware, such as more CPU cores.

9.3 Future Research Directions

We summarize our research as follows

9.3.1 Systems & Performance

As an extension to our research on modeling PBFT consensus process, it will be useful to validate the model for a large number of peers and a wide range of PBFT parameters and system configurations. Also, combine the consensus and block execution models to compute the throughput and latency for order-execute style blockchains as a function of the number of peers, time to execute a transaction, and time to update data store. Since *order-execute* style design is popular across blockchain platforms, it will worth extending the research to other variants of PBFT (such as SBFT, BFT-SMaRt) and other popular consensus protocols such as Raft [74].

Blockchain networks are being deployed across a wide range of domains such as Banking/Finance [49, 81], IoT, and supply chain [19]. It will be useful to study the performance and dependability of blockchain platforms from a domain perspective. Each domain has a varied set of requirements, transaction arrival, and execution process, which will impact the system performance. Although generic blockchain platforms such as Fabric are designed to address a variety of use-cases, it is worth doing performance evaluation with specific industry use-cases in mind. Also, define crisp and clear performance metrics that make sense for that domain.

Just like public blockchains, permissioned blockchains are also expected to run across a wide area network (WAN). Although the number of peers will be a magnitude or two smaller, peers are expected to be deployed across different data centers spread across the world. Hence network latency would still have a significant performance impact. Since block verification at each relay node was a significant contributor to high propagation delay in Bitcoin networks [54], we need to ensure such mistakes are not repeated in the permissioned blockchain space.

Although blockchains provide a tamper-proof record of transactions, the system executing the transactions might be compromised. One way to mitigate this is to run the application inside a trusted execution environment (TEE) such as Intel SGX [95]. However, it would have some performance implications, resulting in a trade-off between performance and privacy.

Energy usage is a popular topic of discussion in public blockchain space [96, 97]. Such research is needed from the context of permissioned blockchain space as well. Also, study the trade-off between performance and energy usage.

9.3.2 Adoption & Usability

As computer engineers and scientists, it is easy to get obsessed with performance metrics such as transaction throughput and latency in elaborate details. However, it is useful to remind ourselves how much time such a transaction would have taken without a blockchain in place. In an informal conversation with a friend working in one of the top financial services companies in the US, where they are moving the company-internal settlement process to a private blockchain, they can resolve transactions in minutes what would previously take 2-3 days. Given the significant improvement in their efficiency, a latency speed-up of a few ms would not matter as much. A similar sentiment is echoed in the report [81] published by the Royal Bank of Scotland, where for a cross-border clearing and settlement system they found

throughput of 100 tps among six banks good enough. Thus, we should continue to focus on improving the usability and adoption of permissioned blockchain networks. Some of the challenges for broad adoption of blockchain-based systems include integrating blockchain systems with existing IT infrastructure [49] and training the technical personnel in writing good bug-free smart contacts [98].

Standardization efforts of blockchain technologies have started picking up as well. Some of the prominent organizations working on it are NIST [1], IEEE [99], International Organization for Standardization (ISO) [100], International Telecommunication Union (ITU) [101], and various working groups under Hyperledger.

Appendix A

Hyperledger Fabric V1 network setup

A.1 Hyperledger Fabric software installation & network setup

The instructions are also available here[1].

Installation of Hyperledger Fabric

On each physical node, run the following setps

1. Basic steps

```
sudo apt install libtool libltdl-dev
```

2. Install docker-ce

```
# Refer:
# https://docs.docker.com/install/linux/docker-ce/ubuntu/\#install-docker-ce-1
sudo apt-get update
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

[1] https://bitbucket.org/hvs2/fabric-perf-model/src/master/network_setup/

```
sudo apt-get update
# sudo apt-get install docker-ce
# sudo apt-cache policy docker-ce to remove stale versions of docker-ce
# sudo apt autoremove to cleanup apt cache
sudo apt-get install docker-ce=18.03.1~ce-0~ubuntu
sudo docker run hello-world
sudo usermod -aG docker $USER
```

At this point, logout and login.

3. Install docker-compose

```
# Refer: https://docs.docker.com/compose/install/\#install-compose
sudo curl -L https://github.com/docker/compose/releases/download/1.20.1/docker
    -compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
docker-compose --version
```

4. Install Go 1.9.1

```
sudo curl -O https://storage.googleapis.com/golang/go1.9.1.linux-amd64.tar.gz
sudo tar -xvf go1.9.1.linux-amd64.tar.gz
sudo mv go /usr/local
mkdir ~/go-workspace
echo 'export GOROOT=/usr/local/go' >> ~/.bashrc
echo 'export GOPATH=$HOME/go-workspace' >> ~/.bashrc
echo 'export PATH=$PATH:$GOROOT/bin:$GOPATH/bin' >> ~/.bashrc
source ~/.bashrc
```

5. Install Python 2.7

```
sudo apt-get install python2.7
sudo apt-get install python-pip
pip install pyyaml
```

6. Install Fabric

```
mkdir -p $GOPATH/src/github.com/hyperledger
cd $GOPATH/src/github.com/hyperledger
git clone http://gerrit.hyperledger.org/r/fabric
cd fabric
git checkout v1.1.0
make release
make gotools #Adding this for now
make docker
make peer
make orderer

#While doing `make docker', if you see the error `Cannot connect to the Docker
     daemon', you probably forgot to do `sudo usermod -aG docker \$USER'.
    After this, logout and login.

sudo mkdir -p /var/hyperledger/product
sudo chown -R $(whoami): /var/hyperledger
```

Add the following to .bashrc and `source /.bashrc`

```
export PATH=$PATH:$GOPATH/src/github.com/hyperledger/fabric/build/bin/
```

Create Docker Swarm network between peers

```
# On one of the peers
docker swarm init --advertise-addr "ip_addr"
docker network create --attachable --driver overlay my-net
# On the rest of peers
docker swarm join --token "token_id" ip_addr:2377
# Once all done, to stop the network
docker swarm leave
```

Making changes to Fabric source code

On your computer, do the following

1. Patch the changes (from diff_file) to

   ```
   $GOPATH/src/github.com/hyperledger/fabric\
   ```

2. Create new Docker images with name, say 1.1.x

   ```
   cd build/image/peer/
   docker build -t hyperledger/fabric-peer:x86_64-1.1.x .
   cd build/image/orderer/
   docker build -t hyperledger/fabric-orderer:x86_64-1.1.x .
   ```

3. Export the docker images

   ```
   docker save -o <path for generated tar file> <image name>
   # scp the .tar file to all nodes. There, do the following
   docker load -i <path to image tar file>
   ```

Other details

The diff files for Fabric source code, and for a new rate-control function for Poisson arrival process in Caliper are shared here[2].

[2] https://bitbucket.org/hvs2/fabric-perf-model/src/master/diff_files

Appendix B

Environment details of a blockchain network

To conduct a performance evaluation of a blockchain network, here are some of the environment aspects to consider (alphabetic order)

- **Blockchain-specific configuration** - Block size and/or frequency
- **Consensus protocol** - In case the platform offers a pluggable choice
- **Data store** - In case the platform offers a pluggable choice
- **Geographical distribution of nodes** - Are the nodes co-located or geographically distributed? If distributed, where is each node located? Also present the block propagation time analysis (refer Figure B.1 and [48]).
- **Hardware**:
  - CPU - model, speed, number of cores, number of vCPUs (for Virtual Machines)
  - Disk drive - speed, type (hard disk vs. solid state drive)
  - Memory - size, speed, type
- **Number of Nodes** of each type (peers, validators)
- **Software** - the software configuration of each node

- **Workload** - the smart contract used, the transaction (+ parameters) mix invoked by the client. If the smart contract is propriety, at least describe the complexity, including the no. of reads/writes performed for each transaction type.

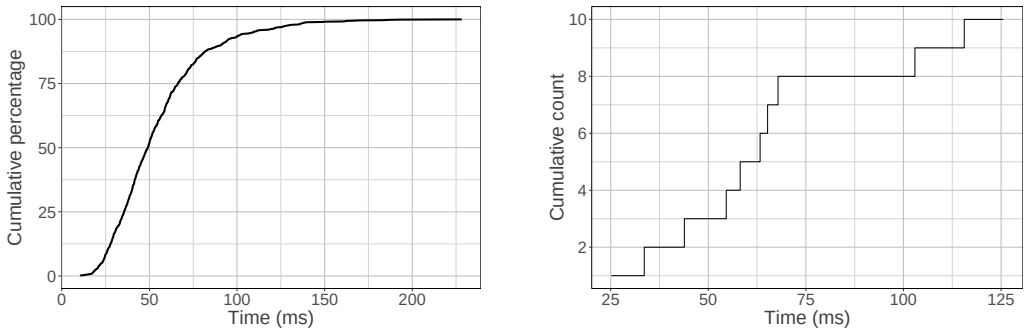


FIGURE B.1: Block propagation time across the network of peers

We summarize the above list for Hyperledger Fabric as follows:

- **Data Store** - LevelDB or CouchDB
- **Fabric configuration** - Batch size (in no. of transactions or MB) and batch timeout.
- **Geographical distribution of nodes**:
  - Ordering service - OSNs, Kafka brokers, zookeeper nodes
  - Peers - Anchor peer and other peers within each organization
  - Fabric Client SDKs
  - Clients (workload generator)
- **Hardware** configuration of each node
- **Number of Nodes**:
  - Ordering service - number of OSNs, Kafka brokers, zookeeper nodes
  - Peers - no. of peers per organization
  - Fabric Client SDKs

- **Ordering service** - Solo or Kafka-based. Future releases plan to support Byzantine-fault tolerant protocols like Raft.
- **Workload**
  - Application - Chaincode (if non-propriety), type of transactions, size of each transaction (in KBs), complexity of transaction (in terms of no. of read/write performed), dependencies between transactions.
  - Transaction mix, arrival process of transactions (e.g., periodic or Poisson), no. of client threads, no. of listener threads.

Appendix C

Model and analysis code for Hyperledger Fabric v0.6

C.1 SRN code for model with $n = 4$

Please refer to Figure 5.2. Also available here[1].

```
#include <stdio.h>
#include "user.h"
#include <stdlib.h>
#include <time.h>

/* global variables */
#define PR12Val1 7.7448
#define PR12Val2 1.50895
#define PR3Val1 34.7047
#define PR3Val2 1.202207
#define TxVal1 1.41605
#define TxVal2 2.09217
#define PR3HypoVal1 2
#define PR3HypoVal2 267.9746
#define PR3HypoVal3 22.04979
#define PR3HypoVal4 0
#define QVal1 25.864
#define QVal2 1.56081
#define QueueRate 8.959

int f = 1;
double timeVal;

/* Prototype for the function(s) */
int gPS0 () {
    if ((mark("PP_Op")>= 2*f))
        return (1);
```

[1] https://bitbucket.org/hvs2/fabric-perf-model/src/master/srn_code_v0.6/srn_model_n4.c

```
28      else
29          return (0);
30  }
31  int gPS1 () {
32      if ((mark("PP_1p")>= (2*f-1)))
33          return (1);
34      else
35          return (0);
36  }
37  int gPS2 () {
38      if ((mark("PP_2p")>= (2*f-1)))
39          return (1);
40      else
41          return (0);
42  }
43  int gPS3 () {
44      if ((mark("PP_3p")>= (2*f-1)))
45          return (1);
46      else
47          return (0);
48  }
49  int gM0 () {
50      if ((mark("P_0p")>= 2*f))
51          return (1);
52      else
53          return (0);
54  }
55  int gM1 () {
56      if ((mark("P_1p")>= 2*f))
57          return (1);
58      else
59          return (0);
60  }
61  int gM2 () {
62      if ((mark("P_2p")>= 2*f))
63          return (1);
64      else
65          return (0);
66  }
67  int gM3 () {
68      if ((mark("P_3p")>= 2*f))
69          return (1);
70      else
71          return (0);
72  }
73  int gCReply () {
74      if (mark("PM0")+mark("PM1")+mark("PM2")+mark("PM3") >= (3*f+1))
75          return (1);
76      else
77          return (0);
78  }
79  int myhalt () {
80      if (mark("Cend") !=0)
81          return (0);
82      else
83          return (1);
84  }
85
86  /* ================= OPTIONS ================ */
87  void options() {
88      iopt(IOP_SIMULATION,VAL_YES);
89      iopt(IOP_SIM_STD_REPORT,VAL_YES);
90      iopt(IOP_SIM_SEED,35983453) ;
91      iopt(IOP_SIM_RUNS, 5000);
92      fopt(FOP_SIM_ERROR, 0.1 ) ;
```

```
93      fopt(FOP_SIM_LENGTH, 20);
94      iopt(IOP_SIM_CUMULATIVE,VAL_NO);
95      fopt(FOP_SIM_CONFIDENCE,.9);
96  }
97
98  /* ========= DEFINITION OF THE NET ========== */
99  void net() {
100     /*  ======  PLACE  ======  */
101     place("PP_0"); place("PP_0p"); place("P_0"); place("P_0p");
102     place("PP_1"); place("PP_1p"); place("P_1"); place("P_1p");
103     place("PP_2"); place("PP_2p"); place("P_2"); place("P_2p");
104     place("PP_3"); place("PP_3p"); place("P_3"); place("P_3p");
105     place("M0"); place("M1"); place("M2"); place("M3");
106     init("M0",1); init("M1",1); init("M2",1); init("M3",1);
107     place("PPS1_0"); place("PPS1_2"); place("PPS1_3");
108     place("PPS2_0"); place("PPS2_1"); place("PPS2_3");
109     place("PPS3_0"); place("PPS3_1"); place("PPS3_2");
110     place("PS0_1"); place("PS0_2"); place("PS0_3");
111     place("PS1_0"); place("PS1_2"); place("PS1_3");
112     place("PS2_0"); place("PS2_1"); place("PS2_3");
113     place("PS3_0"); place("PS3_1"); place("PS3_2");
114     place("PM0"); place("PM1"); place("PM2"); place("PM3");
115     place("PS0"); place("PS1"); place("PS2"); place("PS3");
116     init("PS0",1); init("PS1",1); init("PS2",1); init("PS3",1);
117     place("Cend");
118     place("PPS1"); place("PPS2"); place("PPS3");
119     place("N01"); place("N02"); place("N03");
120     place("Pcl"); init("Pcl",1);
121     place("CLeader"); init("CLeader",1);
122
123     /*  ======  TRANSITION  ======  */
124     imm("tcend");guard("tcend",gCReply);priority("tcend",1);probval("tcend",1);
125     weibval("TC0",PR12Val1, PR12Val2); weibval("TN1",TxVal1, TxVal2);
126     weibval("TN2",TxVal1, TxVal2); weibval("TN3",TxVal1, TxVal2);
127     weibval("TThink1_1",PR12Val1, PR12Val2);
128     weibval("TThink1_2",PR12Val1, PR12Val2);
129     weibval("TThink1_3",PR12Val1, PR12Val2);
130     weibval("TPPS1_0",TxVal1, TxVal2); weibval("TPPS1_2",TxVal1, TxVal2);
131     weibval("TPPS1_3",TxVal1, TxVal2);
132     weibval("TPPS2_0",TxVal1, TxVal2); weibval("TPPS2_1",TxVal1, TxVal2);
133     weibval("TPPS2_3",TxVal1, TxVal2);
134     weibval("TPPS3_0",TxVal1, TxVal2);
135     weibval("TPPS3_1",TxVal1, TxVal2); weibval("TPPS3_2",TxVal1, TxVal2);
136     hypoval("TThink2_0",PR3HypoVal1,PR3HypoVal2,PR3HypoVal3,PR3HypoVal4);
137     hypoval("TThink2_1",PR3HypoVal1,PR3HypoVal2,PR3HypoVal3,PR3HypoVal4);
138     hypoval("TThink2_2",PR3HypoVal1,PR3HypoVal2,PR3HypoVal3,PR3HypoVal4);
139     hypoval("TThink2_3",PR3HypoVal1,PR3HypoVal2,PR3HypoVal3,PR3HypoVal4);
140     guard("TThink2_0",gPS0); guard("TThink2_1",gPS1);
141     guard("TThink2_2",gPS2); guard("TThink2_3",gPS3);
142     weibval("TPS0_1",TxVal1,TxVal2); weibval("TPS0_2",TxVal1,TxVal2);
143     weibval("TPS0_3",TxVal1,TxVal2);
144     weibval("TPS1_0",TxVal1,TxVal2); weibval("TPS1_2",TxVal1,TxVal2);
145     weibval("TPS1_3",TxVal1,TxVal2);
146     weibval("TPS2_0",TxVal1,TxVal2); weibval("TPS2_1",TxVal1,TxVal2);
147     weibval("TPS2_3",TxVal1,TxVal2);
148     weibval("TPS3_0",TxVal1,TxVal2); weibval("TPS3_1",TxVal1,TxVal2);
149     weibval("TPS3_2",TxVal1,TxVal2);
150     weibval("TP0",QVal1,QVal2); weibval("T0",QVal1,QVal2);
151     weibval("TP1",QVal1,QVal2); weibval("T1",QVal1,QVal2);
152     weibval("TP2",QVal1,QVal2); weibval("T2",QVal1,QVal2);
153     weibval("TP3",QVal1,QVal2); weibval("T3",QVal1,QVal2);
154     imm("TM0"); guard("TM0",gM0); priority("TM0",1); probval("TM0",1);
155     imm("TM1"); guard("TM1",gM1); priority("TM1",1); probval("TM1",1);
156     imm("TM2"); guard("TM2",gM2); priority("TM2",1); probval("TM2",1);
157     imm("TM3"); guard("TM3",gM3); priority("TM3",1); probval("TM3",1);
```

```
158
159     halting_condition(myhalt);
160     /*  ======  ARC ====== */
161     iarc("TM0","M0"); oarc("TM0","PM0");
162     iarc("TM1","M1"); oarc("TM1","PM1");
163     iarc("TM2","M2"); oarc("TM2","PM2");
164     iarc("TM3","M3"); oarc("TM3","PM3");
165     iarc("TN1","N01"); oarc("TN1","PPS1");
166     iarc("TN2","N02"); oarc("TN2","PPS2");
167     iarc("TN3","N03"); oarc("TN3","PPS3");
168
169     iarc("TC0","CLeader"); oarc("TC0","N03"); oarc("TC0","N02"); oarc("TC0","N01
            ");
170     iarc("TPPS1_0","PPS1_0"); oarc("TPPS1_0","PP_0");
171     iarc("TPPS1_2","PPS1_2"); oarc("TPPS1_2","PP_2");
172     iarc("TPPS1_3","PPS1_3"); oarc("TPPS1_3","PP_3");
173     iarc("TPPS2_0","PPS2_0"); oarc("TPPS2_0","PP_0");
174     iarc("TPPS2_1","PPS2_1"); oarc("TPPS2_1","PP_1");
175     iarc("TPPS2_3","PPS2_3"); oarc("TPPS2_3","PP_3");
176     iarc("TPPS3_0","PPS3_0"); oarc("TPPS3_0","PP_0");
177     iarc("TPPS3_1","PPS3_1"); oarc("TPPS3_1","PP_1");
178     iarc("TPPS3_2","PPS3_2"); oarc("TPPS3_2","PP_2");
179
180     iarc("TPS0_1","PS0_1"); oarc("TPS0_1","P_1");
181     iarc("TPS0_2","PS0_2"); oarc("TPS0_2","P_2");
182     iarc("TPS0_3","PS0_3"); oarc("TPS0_3","P_3");
183     iarc("TPS1_0","PS1_0"); oarc("TPS1_0","P_0");
184     iarc("TPS1_2","PS1_2"); oarc("TPS1_2","P_2");
185     iarc("TPS1_3","PS1_3"); oarc("TPS1_3","P_3");
186     iarc("TPS2_0","PS2_0"); oarc("TPS2_0","P_0");
187     iarc("TPS2_1","PS2_1"); oarc("TPS2_1","P_1");
188     iarc("TPS2_3","PS2_3"); oarc("TPS2_3","P_3");
189     iarc("TPS3_0","PS3_0"); oarc("TPS3_0","P_0");
190     iarc("TPS3_1","PS3_1"); oarc("TPS3_1","P_1");
191     iarc("TPS3_2","PS3_2"); oarc("TPS3_2","P_2");
192
193     iarc("tcend","Pcl");
194     iarc("TThink1_1","PPS1"); oarc("TThink1_1","PPS1_0");
195     oarc("TThink1_1","PPS1_2"); oarc("TThink1_1","PPS1_3");
196     iarc("TThink1_2","PPS2"); oarc("TThink1_2","PPS2_0");
197     oarc("TThink1_2","PPS2_1"); oarc("TThink1_2","PPS2_3");
198     iarc("TThink1_3","PPS3"); oarc("TThink1_3","PPS3_0");
199     oarc("TThink1_3","PPS3_1"); oarc("TThink1_3","PPS3_2");
200
201     iarc("TThink2_0","PS0"); oarc("TThink2_0","PS0_1");
202     oarc("TThink2_0","PS0_2"); oarc("TThink2_0","PS0_3");
203     iarc("TThink2_1","PS1"); oarc("TThink2_1","PS1_0");
204     oarc("TThink2_1","PS1_2"); oarc("TThink2_1","PS1_3");
205     iarc("TThink2_2","PS2"); oarc("TThink2_2","PS2_0");
206     oarc("TThink2_2","PS2_1"); oarc("TThink2_2","PS2_3");
207     iarc("TThink2_3","PS3"); oarc("TThink2_3","PS3_0");
208     oarc("TThink2_3","PS3_1"); oarc("TThink2_3","PS3_2");
209
210     iarc("TP0","PP_0"); oarc("TP0","PP_0p");
211     iarc("TP1","PP_1"); oarc("TP1","PP_1p");
212     iarc("TP2","PP_2"); oarc("TP2","PP_2p");
213     iarc("TP3","PP_3"); oarc("TP3","PP_3p");
214
215     iarc("T0","P_0"); oarc("T0","P_0p");
216     iarc("T1","P_1"); oarc("T1","P_1p");
217     iarc("T2","P_2"); oarc("T2","P_2p");
218     iarc("T3","P_3"); oarc("T3","P_3p");
219     oarc("tcend","Cend");
220 }
221
```

```
int QlenP0 () {
    return (mark("PP_0"));
}
int QlenC0 () {
    return (mark("P_0"));
}
int UtilP0 () {
    return (enabled("TP0"));
}
int UtilC0 () {
    return (enabled("T0"));
}
int assert() {
}
void ac_init() {
    pr_net_info();
}
void ac_reach() {
    pr_rg_info();
}
double holdingTime() {
    if(mark("Cend") == 0)
        return(1.0);
    else
        return(0.0);
}
void ac_final() {
    pr_cum_expected("time in non-absorbing markings", holdingTime);
}
```

C.2 Python script to generate SRN code for larger networks

Also available here[2].

```
def create_SRN(f,N):
    file = open("exact_f%s_N%s_new_param_0421_script.c" % (f,N),'w+')
    file.write('#include <stdio.h>\n#include "user.h"\n')
    file.write('#include <stdlib.h>\n#include <time.h>\n')
    file.write("#define PR12Val1 7.7448 \n")
    file.write("#define PR12Val2 1.50895 \n")
    file.write("#define PR3Val1 34.7047 \n")
    file.write("#define PR3Val2 1.202207 \n")
    file.write("#define TxVal1 1.41605 \n")
    file.write("#define TxVal2 2.09217 \n")
    file.write("#define PR3HypoVal1 2 \n")
    file.write("#define PR3HypoVal2 267.9746 \n")
    file.write("#define PR3HypoVal3 22.04979 \n")
    file.write("#define PR3HypoVal4 0 \n")
    file.write("#define QVal1 25.864 \n")
    file.write("#define QVal2 1.56081 \n")
    file.write("#define QueueRate 8.959 \n")
    file.write("int f = %s; \n" % (f))
    file.write("int n = %s; \n" % (N))
    file.write("double timeVal; \n")

    # Functions
    file.write("int gPS0 () {\n")
```

[2] https://bitbucket.org/hvs2/fabric-perf-model/src/master/srn_code_v0.6/generate_srn.py

```
24      file.write("\tif ((mark(\"PP_0p\")>= 2*f)) return (1);\n");
25      file.write("\t\telse\n");
26      file.write("\treturn (0);}\n");
27
28      for a in range(1,N):
29          file.write("int gPS%s () {  \n" % (a))
30          file.write("\tif ((mark(\"PP_%sp\")>= (2*f-1)))" % (a))
31          file.write( "\treturn (1);\n    else\n    return (0);\n    }\n")
32
33      for a in range(0,N):
34          file.write("int gM%s () {  \n" % (a))
35          file.write("\tif ((mark(\"P_%sp\")>= 2*f))" % (a))
36          file.write("\treturn (1);\n    else\n    return (0);\n    }\n")
37
38      file.write('int gCReply () { \n')
39      file.write('\tif ((')
40
41      for a in range(0,N):
42          file.write("mark(\"PM%s\")" % (a))
43          if a != N-1:
44              file.write("+")
45          a+=1
46      file.write(">= (3*f+1)))\n")
47      file.write("\treturn (1);\n    else\n    return (0);\n    }\n")
48
49      file.write("int myhalt () {\n\tif (mark(\"Cend\") !=0)\n\t\treturn (0);\n\
            telse\n\t\treturn (1);\n}\n")
50
51      #Options
52      file.write('void options() { \n')
53      file.write('\tiopt(IOP_SIMULATION,VAL_YES); \n')
54      file.write('\tiopt(IOP_SIM_STD_REPORT,VAL_YES); \n')
55      file.write('\tiopt(IOP_SIM_SEED,35983453) ; \n')
56      file.write('\tiopt(IOP_SIM_RUNS, 5000); \n')
57      file.write('\tfopt(FOP_SIM_ERROR, 0.1 ) ; \n')
58      file.write('\tfopt(FOP_SIM_LENGTH, 20); \n')
59      file.write('\tiopt(IOP_SIM_CUMULATIVE,VAL_NO); \n')
60      file.write('\tfopt(FOP_SIM_CONFIDENCE,.9); \n')
61      file.write('}\n')
62
63      file.write('void net() {\n')
64      for a in range(0,N):
65          file.write("\tplace(\"PP_%s\");\n" % (a))
66          file.write("\tplace(\"PP_%sp\");\n" % (a))
67          file.write("\tplace(\"P_%s\");\n" % (a))
68          file.write("\tplace(\"P_%sp\");\n" % (a))
69          file.write("\tplace(\"M%s\");\n" % (a))
70          file.write("\tinit(\"M%s\",1);\n" % (a))
71          file.write("\tplace(\"PM%s\");\n" % (a))
72          file.write("\tplace(\"PS%s\");\n" % (a))
73          file.write("\tinit(\"PS%s\",1);\n" % (a))
74
75      for a in range(1,N):
76          for b in range(0,N):
77              if b != a:
78                  file.write("\tplace(\"PPS%s_%s\");\n" % (a,b))
79      file.write('\n')
80
81      for a in range(0,N):
82          for b in range(0,N):
83              if b != a:
84                  file.write("\tplace(\"PS%s_%s\");\n" % (a,b))
85      file.write('\n')
86
87      file.write('\tplace("Cend");')
```

```
88      file.write('\n')
89
90      for a in range(1,N):
91          file.write("\tplace(\"PPS%s\");\n" % (a))
92          file.write("\tplace(\"N0%s\");\n" % (a))
93
94      file.write('\tplace("Pcl");\n ')
95      file.write('\tinit("Pcl",1);\n ')
96      file.write('\tplace("CLeader");\n ')
97      file.write('\tinit("CLeader",1);\n ')
98
99      file.write('\timm("tcend"); guard("tcend",gCReply);')
100     file.write('\tpriority("tcend",1); probval("tcend",1);\n')
101
102     file.write('\tweibval("TC0",PR12Val1, PR12Val2);\n')
103
104     for a in range(1,N):
105         file.write("\tweibval(\"TN%s\",TxVal1, TxVal2);\n" % (a))
106         file.write("\tweibval(\"TThink1_%s\",PR12Val1, PR12Val2);\n" % (a))
107     file.write('\n')
108
109     for a in range(1,N):
110         for b in range(0,N):
111             if b != a:
112                 file.write("\tweibval(\"TPPS%s_%s\",TxVal1, TxVal2);\n" % (a,b))
113     file.write('\n')
114
115     for a in range(0,N):
116         file.write("\thypoval(\"TThink2_%s\",PR3HypoVal1, PR3HypoVal2,
            PR3HypoVal3, PR3HypoVal4);\n" % (a))
117         file.write("\tguard(\"TThink2_%s\",gPS%s);\n" % (a,a))
118     file.write('\n')
119
120     for a in range(0,N):
121         for b in range(0,N):
122             if b != a:
123                 file.write("\tweibval(\"TPS%s_%s\",TxVal1,TxVal2);\n" % (a,b))
124     file.write('\n')
125
126     for a in range(0,N):
127         file.write("\tweibval(\"TP%s\",QVal1,QVal2);\n" % (a))
128         file.write("\tweibval(\"T%s\",QVal1,QVal2);\n" % (a))
129         file.write("\timm(\"TM%s\");\n" % (a))
130         file.write("\tguard(\"TM%s\",gM%s);\n" % (a,a))
131         file.write("\tpriority(\"TM%s\",1);\n" % (a))
132         file.write("\tprobval(\"TM%s\",1);\n" % (a))
133     file.write('\thalting_condition(myhalt);\n')
134     file.write('  /*  ======  ARC ====== */\n')
135
136     for a in range(0,N):
137         file.write("\tiarc(\"TM%s\",\"M%s\");\n" % (a,a))
138         file.write("\toarc(\"TM%s\",\"PM%s\");\n" % (a,a))
139
140     for a in range(1,N):
141         file.write("\toarc(\"TN%s\",\"PPS%s\");\n" % (a,a))
142         file.write("\toarc(\"TC0\",\"N0%s\");\n" % (a))
143         file.write("\tiarc(\"TN%s\",\"N0%s\");\n" % (a,a))
144
145     file.write('\tiarc("TC0","CLeader");\n')
146
147     for a in range(1,N):
148         for b in range(0,N):
149             if b != a:
150                 file.write("\tiarc(\"TPPS%s_%s\",\"PPS%s_%s\");\n" % (a,b,a,b))
151     file.write('\n')
```

```
152
153     for a in range(0,N):
154         for b in range(0,N):
155             if b != a:
156                 file.write("\tiarc(\"TPS%s_%s\",\"PS%s_%s\");\n" % (a,b,a,b))
157     file.write('\tiarc("tcend","Pcl");\n')
158
159     for a in range(1,N):
160         file.write("\tiarc(\"TThink1_%s\",\"PPS%s\");\n" % (a,a))
161     file.write('\n')
162
163     for a in range(0,N):
164         file.write("\tiarc(\"TThink2_%s\",\"PS%s\");\n" % (a,a))
165     file.write('\n')
166
167     for a in range(1,N):
168         for b in range(0,N):
169             if b != a:
170                 file.write("\toarc(\"TPPS%s_%s\",\"PP_%s\");\n" % (a,b,b))
171                 file.write("\toarc(\"TThink1_%s\",\"PPS%s_%s\");\n" % (a,a,b))
172     file.write('\n')
173
174     for a in range(0,N):
175         file.write("\tiarc(\"TP%s\",\"PP_%s\");\n" % (a,a))
176         file.write("\toarc(\"TP%s\",\"PP_%sp\");\n" % (a,a))
177     file.write('\n')
178
179     for a in range(0,N):
180         for b in range(0,N):
181             if b != a:
182                 file.write("\toarc(\"TThink2_%s\",\"PS%s_%s\");\n" % (a,a,b))
183                 file.write("\toarc(\"TPS%s_%s\",\"P_%s\");\n" % (a,b,b))
184     file.write('\n')
185
186     for a in range(0,N):
187         file.write("\tiarc(\"T%s\",\"P_%s\");\n" % (a,a))
188         file.write("\toarc(\"T%s\",\"P_%sp\");\n" % (a,a))
189     file.write('\n')
190
191     file.write('\toarc("tcend","Cend");\n')
192     file.write('  /* Inhibtor Arcs */ \n')
193     file.write('}')
194     file.write('\n')
195
196     file.write('  /* GUARD Functions */  \n')
197
198     file.write('int assert() {}\n')
199     file.write('void ac_init() {\n\tpr_net_info();\n}\n')
200     file.write('void ac_reach() {\n\tpr_rg_info();\n}\n')
201     file.write('double holdingTime() {\n if(mark("Cend") == 0) \n return(1.0); \
        n else \n return(0.0); } \n')
202     file.write('void ac_final() {\n')
203     file.write('\tpr_cum_expected("time in non-absorbing markings", holdingTime)
        ; \n}\n')
204
205 create_SRN(1,4)
```

C.3 R code for Probability distribution fitting

R code and datasets are available here[3].

```
library(fitdistrplus)
source('distributions.R')
dataset <- read.table('datasetsV06/tx_all.csv')
dataset <- dataset$V1
summary(dataset)
sd(dataset)

#dataset fitting
fexp <- fitdist(dataset, "exp")
fw <- fitdist(dataset, "weibull", method = "mle", lower = c(0, 0))
fg <- fitdist(dataset, "gamma", lower=c(0,0))
fE2 <- fitdist(dataset, "erlang_2", start=c(0.01))
fE3 <- fitdist(dataset, "erlang_3", start=c(0.01))
fHypo_2 <- fitdist(dataset, "hypoexp_2", start=c(0.0001, 0.01)) #NOTE: the two starting parameter values should be DIFFERENT
fHypo_3 <- fitdist(dataset, "hypoexp_3", start=c(0.0001, 0.01, 0.001)) #NOTE: the three starting parameter values should be DIFFERENT
fPareto <- fitdist(dataset, "pareto", start=list(shape = 1, scale = 500))
fLogN <- fitdist(dataset, "lnorm")

par(mfrow = c(2, 2))
plot.legend <- c("Exponential", "Weibull", "Gamma", "Hypoexp (2-stage)", "Hypoexp (3-stage)", "LogNormal", "Pareto")
denscomp(list(fexp, fw, fg,fHypo_2, fHypo_3, fLogN, fPareto), legendtext = plot.legend)
qqcomp(list(fexp, fw, fg, fHypo_2, fHypo_3, fLogN, fPareto), legendtext = plot.legend)
cdfcomp(list(fexp, fw, fg, fHypo_2, fHypo_3, fLogN, fPareto), legendtext = plot.legend)
ppcomp(list(fexp, fw, fg,  fHypo_2, fHypo_3, fLogN, fPareto), legendtext = plot.legend)
quantile(fexp, probs = 0.95)
quantile(dataset, probs = 0.95)

gofstat(list(fexp, fw, fg, fE2, fE3, fHypo_2, fHypo_3, fLogN, fPareto),fitnames = c("exp", "weibull", "gamma", "Erlang2", "Erlang3", "Hypoexp_2", "Hypoexp_3", "LogNormal", "Pareto"))
ks.test(dataset, "pexp", fexp$estimate)
ks.test(dataset, "pweibull", fw$estimate[1], fw$estimate[2])
ks.test(dataset, "pgamma", fg$estimate[1], fg$estimate[2])
ks.test(dataset, "perlang_2", fE2$estimate[1])
ks.test(dataset, "perlang_3", fE3$estimate[1])
ks.test(dataset, "phypoexp_2", fHypo_2$estimate[1], fHypo_2$estimate[2])
ks.test(dataset, "phypoexp_3", fHypo_3$estimate[1], fHypo_3$estimate[2], fHypo_3$estimate[3])
ks.test(dataset, "plnorm", fLogN$estimate[1], fLogN$estimate[2])
ks.test(dataset, "ppareto", fPareto$estimate[1], fPareto$estimate[2])

#NOTE: Different packages use different distn. fn. for distributions like Weibull, Gamma etc.
#Weibull distn fn for SPNP is different than that in R
#The following code converts the parameter values for SPNP
shape <- fw$estimate[1]
scale <- 1/(fw$estimate[2]^fw$estimate[1])
```

[3] https://bitbucket.org/hvs2/fabric-perf-model/src/master/data_analysis/

Appendix D

Model and analysis code for Hyperledger Fabric V1

D.1 Endorsing process

SRN code for the model in Figure 7.6, also available here[1].

```
1  #include <stdio.h>
2  #include "user.h"
3
4  /* global variables */
5  double RATE_ENDOR = 0.308;
6
7  int guard_end();
8  int PendingP0();
9  int PendingP1();
10 int PendingWait();
11
12 void options() {
13     iopt(IOP_PR_RSET, VAL_YES);
14     iopt(IOP_PR_RGRAPH, VAL_YES);
15     iopt(IOP_PR_FULL_MARK, VAL_YES);
16 }
17
18 int guard_end() {
19     if (mark("P_wait") > 0) {
20         return 1;
21     } else {
22         return 0;
23     }
24 }
25
26 int PendingP0() {
27     return (mark("P_en0"));
28 }
29
```

[1] https://bitbucket.org/hvs2/fabric-perf-model/src/master/srn_code_V1/srn_endorsing.c

```
30 int PendingP1() {
31     return (mark("P_en1"));
32 }
33
34 int PendingWait() {
35     return (mark("P_wait"));
36 }
37
38 void net() {
39     /* Places and Transitions */
40     place("Pend");
41     place("P_en0"); init("P_en0",1);
42     place("P_en1"); init("P_en1",1);
43     place("P_wait");
44
45     imm("I_wait");
46     guard("I_wait",guard_end);
47     priority("I_wait",1);
48     probval("I_wait",1);
49
50     rateval("T_en1",RATE_ENDOR);
51     rateval("T_en0",RATE_ENDOR);
52
53     /* Arcs */
54     iarc("T_en0","P_en0");
55     iarc("T_en1","P_en1");
56     viarc("I_wait","P_wait",PendingWait);
57     viarc("I_wait","P_en0",PendingP0);
58     viarc("I_wait","P_en1",PendingP1);
59     oarc("T_en0","P_wait");
60     oarc("T_en1","P_wait");
61     oarc("I_wait","Pend");
62 }
63
64 int assert() {
65
66 }
67
68 void ac_init() {
69     pr_net_info();
70 }
71
72 void ac_reach() {
73     pr_rg_info();
74 }
75
76
77 void ac_final() {
78     int loop;
79     solve(INFINITY);
80     pr_mtta("time to absorption");
81 }
```

D.2 Ordering Service

SRN code for the model in Figure 7.7, also available here[2]

[2] https://bitbucket.org/hvs2/fabric-perf-model/src/master/srn_code_V1/srn_ordering.c

```c
#include <stdio.h>
#include "user.h"
#include <stdlib.h>
#include <time.h>

/* global variables */
#define BLOCK_SIZE 120
#define TX_ARRIVAL 500

#define QUEUE_SIZE BLOCK_SIZE*2
#define TIMEOUT 2
#define ERLANG_STAGES 5
#define ERL_STAGE_RATE 2.5 // ERLANG_STAGES/TIMEOUT

void options() {
    iopt(IOP_SSMETHOD, VAL_POWER);
    iopt(IOP_SIMULATION,VAL_NO);
}

double holdingTime_Erlang () {
    if (mark("postfire") < ERLANG_STAGES) {
        return (1.0);
    } else
        return (0.0);
}

double holdingTime1 () {
    if (mark("end_timeout") == 0) {
        return (1.0);
    } else
        return (0.0);
}

double holdingTime2 () {
    if (mark("end_size") == 0) {
        return (1.0);
    } else
        return (0.0);
}

int guard_batch_size() {
    if (mark("end_timeout") == 1 || mark("end_size") == 1) {
        return 0;
    } else {
        return 1;
    }
}

int guard_timeout() {
    if (mark("postfire") == ERLANG_STAGES && mark("end_timeout") == 0 && mark("
        end_size") == 0) {
        return 1;
    } else {
        return 0;
    }
}

int timeout_trigger_fn() {
    if (mark("ordering_service") > 0) {
        return 1;
    } else {
        return 0;
    }
}

```

```
65  void net() {
66      /* Main NET */
67      rateval("tx_arrival", TX_ARRIVAL);
68      mharc("tx_arrival", "end_timeout", 1);
69      mharc("tx_arrival", "end_size", 1);
70
71      place("ordering_service");
72      /* Timeout case */
73      imm("check_timeout");
74      guard("check_timeout", guard_timeout);
75      priority("check_timeout", 1);
76      place("end_timeout");
77      oarc("tx_arrival", "ordering_service");
78      iarc("check_timeout", "ordering_service");
79      oarc("check_timeout", "end_timeout");
80      /* Size case */
81      imm("batch_size");
82      guard("batch_size", guard_batch_size);
83      priority("batch_size", 1);
84      place("end_size");
85      miarc("batch_size", "ordering_service", BLOCK_SIZE);
86      oarc("batch_size", "end_size");
87
88      /* Net for Deterministic timeout */
89      place("TO_start");
90      init("TO_start", 1);
91      imm("timeout_trigger");
92      priority("timeout_trigger", 1);
93      guard("timeout_trigger", timeout_trigger_fn);
94
95      place("prefire");
96      place("postfire");
97      rateval("timeout_stage", ERL_STAGE_RATE);
98      iarc("timeout_trigger", "TO_start");
99      moarc("timeout_trigger", "prefire", ERLANG_STAGES);
100     iarc("timeout_stage", "prefire");
101     oarc("timeout_stage", "postfire");
102     mharc("timeout_trigger", "prefire", 1);
103     mharc("timeout_trigger", "postfire", 1);
104 }
105
106 int assert() {
107 }
108
109 void ac_init() {
110     pr_net_info();
111 }
112
113 void ac_reach() {
114     pr_rg_info();
115 }
116
117 double option_timeout() {
118     return mark("end_timeout");
119 }
120
121 double option_size() {
122     return mark("end_size");
123 }
124
125 void ac_final() {
126     solve(INFINITY);
127     pr_expected("Prob. of completing timeout", option_timeout);
128     pr_expected("Prob. of completing Rateval", option_size);
129 }
```

R code for generating the plot in Figure 7.8, also available here[3]

```
library("plotly")
library("reshape2")
ordering <- read.table('datasets/ordering.csv', header = TRUE, sep = ',')
ordering_2.0 <- subset(ordering, timeout==2)
ordering_2.0 <- subset(ordering_2.0, block_size!=20)
ordering_2.0_melt <- melt(ordering_2.0, id.vars = c('block_size', 'arrival_rate'), measure.vars = 'prob_timeout')
names(ordering_2.0_melt)[4] <- "prob_timeout"

xax <- list(
  title = "Block Size"
)
yax <- list(
  title = "Arrival Rate (\\lambda_E)"
)

plot_ly(ordering_2.0_melt, x = ~block_size, y = ~arrival_rate, z = ~prob_timeout,
  type = "contour",width = 400, height = 300, colorscale="Blues") %>%
  layout(xaxis = xax, yaxis = yax, title="Block timeout = 2.0 sec.") %>%
  colorbar(title = "Pr.(timeout)")
```

D.3 Committing peer

SRN code for the model in Figure 7.9, also available here[4].

```
#include <stdio.h>
#include "user.h"
#include <stdlib.h>
#include <time.h>

/* global variables */
#define CORES 4
#define RATE_BLOCK_ARR 1.0
#define RATE_VSCC 396.197
#define RATE_MVCC 390.777 // Size = 40
//#define RATE_MVCC 196.08 // Size = 80
//#define RATE_MVCC 138.89 // Size = 120

#define BLOCK_SIZE 40
#define QUEUE_SIZE BLOCK_SIZE*2
#define RATE_LWRITE 4.8123 // Size = 40
//#define RATE_LWRITE 4.801 // Size = 80
//#define RATE_LWRITE 5.308 // Size = 120

void options() {
    //iopt(IOP_SSMETHOD, VAL_POWER);
    iopt(IOP_SIMULATION,VAL_NO);
}

double rate_vscc() {
    if (mark("vscc_check") <= CORES) {
        return (RATE_VSCC*mark("vscc_check"));
```

[3] https://bitbucket.org/hvs2/fabric-perf-model/src/master/data_analysis/ordering_ContourPlot.R

[4] https://bitbucket.org/hvs2/fabric-perf-model/src/master/srn_code_V1/srn_committer.c

```
28      } else {
29          return (RATE_VSCC*CORES);
30      }
31  }
32
33  double tput_arr() {
34      return (rate("block_arrival"));
35  }
36
37  double util_arr() {
38      return (enabled("block_arrival"));
39  }
40
41  double tput_vscc() {
42      return (rate("vscc"));
43  }
44
45  double util_vscc() {
46      return (enabled("vscc"));
47  }
48
49  double qlen_vscc() {
50      return mark("vscc_check");
51  }
52
53  double tput_mvcc() {
54      return (rate("mvcc"));
55  }
56
57  double util_mvcc() {
58      return (enabled("mvcc"));
59  }
60
61  double qlen_mvcc() {
62      return mark("mvcc_check");
63  }
64
65  double tput_lwrite() {
66      return (rate("lwrite"));
67  }
68
69  double util_lwrite() {
70      return (enabled("lwrite"));
71  }
72
73  double qlen_lwrite() {
74      return mark("ledger_write");
75  }
76
77  double incoming_queue_full() {
78      if (mark("vscc_check") == QUEUE_SIZE) {
79          return 1;
80      } else {
81          return 0;
82      }
83  }
84
85  void net() {
86      /* Places and Transitions */
87      rateval("block_arrival", RATE_BLOCK_ARR);
88      place("vscc_check");
89      ratefun("vscc", rate_vscc);
90      place("vscc_check_done");
91      imm("vscc_collect");
92      priority("vscc_collect", 1);
```

```
93
94      place("mvcc_check");
95      rateval("mvcc", RATE_MVCC);
96      place("ledger_write");
97      rateval("lwrite", RATE_LWRITE);
98
99      /* Arcs */
100     mharc("block_arrival", "vscc_check", QUEUE_SIZE);
101     moarc("block_arrival", "vscc_check", BLOCK_SIZE);
102     iarc("vscc", "vscc_check");
103     oarc("vscc", "vscc_check_done");
104     mharc("vscc", "vscc_check_done", QUEUE_SIZE);
105
106     miarc("vscc_collect", "vscc_check_done", BLOCK_SIZE);
107     oarc("vscc_collect", "mvcc_check");
108     mharc("vscc_collect", "mvcc_check", 2);
109
110     iarc("mvcc", "mvcc_check");
111     oarc("mvcc", "ledger_write");
112     mharc("mvcc", "ledger_write", 2);
113
114     iarc("lwrite", "ledger_write");
115 }
116
117 /* metrics */
118
119 int assert() {
120
121 }
122
123 void ac_init() {
124 /* Information on the net structure */
125  pr_net_info();
126 }
127
128 void ac_reach() {
129 /* Information on the reachability graph */
130  pr_rg_info();
131 }
132
133 void ac_final() {
134     solve(INFINITY);
135     pr_expected("Throughput at Block Arri", tput_arr);
136     pr_expected("Utilizati. at Block Arri", util_arr);
137     pr_expected("Incoming Queue Full     ", incoming_queue_full);
138     pr_expected("Throughput at VSCC check", tput_vscc);
139     pr_expected("Utilizati. at VSCC check", util_vscc);
140     pr_expected("Queue Len. at VSCC check", qlen_vscc);
141     pr_expected("Throughput at RW   check", tput_mvcc);
142     pr_expected("Utilizati. at RW   check", util_mvcc);
143     pr_expected("Queue Len. at RW   check", qlen_mvcc);
144     pr_expected("Throughput at Ledger    ", tput_lwrite);
145     pr_expected("Utilizati. at Ledger    ", util_lwrite);
146     pr_expected("Queue Len. at Ledger    ", qlen_lwrite);
147 }
```

Appendix E

Mathematical expression for Probability distributions

In this section, we provide the probability density functions ($f_X(t)$) for various probability distributions used in our thesis. Probability distributions are used in two tools, R for data analysis, and SPNP for our models. The functions described below are common to both the tools, unless otherwise stated.

- Exponential distribution

$$f_X(t) = \lambda e^{-\lambda t}, \quad t \geqslant 0$$

- Weibull distribution

$$\text{R:} \quad f_X(t) = \frac{a}{b}\left(\frac{t}{b}\right)^{a-1} e^{-\left(\frac{t}{b}\right)^a}, \quad t \geqslant 0, a > 0, b > 0$$

$$\text{SPNP:} \quad f_X(t) = \lambda\alpha t^{\alpha-1} e^{-\lambda t^\alpha}, \quad t \geqslant 0, \alpha > 0, \lambda > 0$$

  Thus, to convert parameter value from R to SPNP, $\alpha = a$ and $\lambda = \frac{1}{b^a}$

- Gamma distribution

$$f_X(t) = \frac{1}{\Gamma(\alpha)}\lambda^\alpha t^{\alpha-1} e^{-\lambda t}, \quad \alpha > 0, \lambda > 0, t \geqslant 0$$

- Erlang distribution

$$f_X(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!}, \quad \lambda > 0, t \geqslant 0$$

  where k is the number of stages.

- Hypoexponential distribution (2-stage)

$$f_X(t) = \frac{\lambda_1 \lambda_2}{(\lambda_2 - \lambda_1)} \left(e^{-\lambda_1 t} - e^{-\lambda_2 t} \right), \quad \lambda_1 \neq \lambda_2, \lambda_1 > 0, \lambda_2 > 0, t \geqslant 0$$

- LogNormal distribution

$$f_X(t) = \frac{1}{t\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln t - \mu)^2}{2\sigma^2}}, \quad \sigma > 0, t \geqslant 0$$

- Pareto distribution

$$f_X(t) = \frac{\alpha \theta^\alpha}{(t+\theta)^{(\alpha+1)}}, \quad \alpha > 0, \theta > 0, t \geqslant 0$$

Bibliography

[1] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain Technology Overview (Draft NISTIR 82022)," https://csrc.nist.gov/CSRC/media/Publications/nistir/8202/draft/documents/nistir8202-draft.pdf.

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," https://bitcoin.org/bitcoin.pdf.

[3] D. Tapscott and A. Tapscott, *Blockchain Revolution : How the Technology behind Bitcoin is Changing Money, Business, and the World*. New York: Portfolio Penguin, 2016.

[4] J. Mendling, I. Weber, W. V. D. Aalst, J. V. Brocke, C. Cabanillas, F. Daniel, S. Debois, C. D. Ciccio, M. Dumas, S. Dustdar, A. Gal, L. García-Bañuelos, G. Governatori, R. Hull, M. L. Rosa, H. Leopold, F. Leymann, J. Recker, M. Reichert, H. A. Reijers, S. Rinderle-Ma, A. Solti, M. Rosemann, S. Schulte, M. P. Singh, T. Slaats, M. Staples, B. Weber, M. Weidlich, M. Weske, X. Xu, and L. Zhu, "Blockchains for Business Process Management - Challenges and Opportunities," *ACM Trans. Manage. Inf. Syst.*, vol. 9, no. 1, pp. 4:1–4:16, Feb. 2018.

[5] T. Swanson, "Consensus-as-a-Service: a brief report on the emergence of permissioned, distributed ledger systems," http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf, 2015.

[6] N. Szabo, "Smart contracts: Building blocks for digital markets," 1996. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[7] Hyperledger, "Hyperledger Architecture, Volume 2 - Smart Contracts," https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf.

[8] "Ethereum - White Paper (wiki)," https://github.com/ethereum/wiki/wiki/White-Paper, accessed: 2017-01-15.

[9] C. Cachin and M. Vukolić, "Blockchain Consensus Protocols in the Wild," in *International Symposium on Distributed Computing (DISC)*, A. W. Richa, Ed., 2017, pp. 1:1–1:16.

[10] IBM, "The difference between public and private blockchain," https://www.ibm.com/blogs/blockchain/2017/05/the-difference-between-public-and-private-blockchain/.

[11] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[12] M. Vukolić, "Rethinking Permissioned Blockchains," in *ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC)*, 2017, pp. 3–7.

[13] "Hyperledger Business Blockchain Technologies," https://www.hyperledger.org/projects.

[14] Hyperledger, "Hyperledger Architecture, Volume 1 - Consensus," https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

[15] "Hyperledger Fabric," https://www.hyperledger.org/projects/fabric.

[16] IBM, "IBM Blockchain Platform," https://www.ibm.com/blogs/blockchain/2017/08/your-guide-to-the-ibm-blockchain-platform-announcement/.

[17] ——, "One nations move to increase food safety with blockchain," https://www.ibm.com/blogs/blockchain/2018/02/one-nations-move-to-increase-food-safety-with-blockchain/, accessed: 2018-05-14.

[18] ——, "New collaboration on trade finance platform built on blockchain," https://www.ibm.com/blogs/blockchain/2017/10/new-collaboration-on-trade-finance-platform-built-on-blockchain/, accessed: 2018-05-14.

[19] ——, "Digitizing Global Trade with Maersk and IBM," https://www.ibm.com/blogs/blockchain/2018/01/digitizing-global-trade-maersk-ibm/, accessed: 2018-05-14.

[20] "Mining - Bitcoin Wiki," https://en.bitcoin.it/wiki/Mining.

[21] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer, *On Scaling Decentralized Blockchains*. Springer Berlin Heidelberg, 2016, pp. 106–125.

[22] 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 *EuroSys*, ser. EuroSys, 2018, pp. 30:1–30:15.

[23] "Hyperledger Caliper," https://www.hyperledger.org/projects/caliper.

[24] J. K. Muppala, G. Ciardo, and K. S. Trivedi, "Stochastic Reward Nets for Reliability Prediction," in *Communications in Reliability, Maintainability and Serviceability*, 1994, pp. 9–20.

[25] R. Mitchell and I. R. Chen, "Effect of Intrusion Detection and Response on Reliability of Cyber Physical Systems," *IEEE Transactions on Reliability*, vol. 62, no. 1, pp. 199–210, March 2013.

[26] D. Bruneo, "A Stochastic Model to Investigate Data Center Performance and QoS in IaaS cloud computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 560–569, 2014.

[27] D. S. Kim, J. B. Hong, T. A. Nguyen, F. Machida, J. S. Park, and K. S. Trivedi, "Availability Modeling and Analysis of a Virtualized System using Stochastic Reward Nets," in *IEEE CIT*, Dec 2016, pp. 210–218.

[28] K. Trivedi and A. Bobbio, *Reliability and Availability Engineering: Modeling, Analysis, and Applications*. Cambridge University Press, 2017.

[29] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric)," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Sept. 2017, pp. 253–255, ©2017 IEEE.

[30] "Hyperledger Fabric v1.0 announcement," https://www.hyperledger.org/announcements/2017/07/11/hyperledger-announces-production-ready-hyperledger-fabric-1-0, accessed: 2017-12-11.

[31] Docker, "What is a Container," https://www.docker.com/what-container.

[32] J. R. Douceur, "The Sybil Attack," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. Springer-Verlag, 2002, pp. 251–260.

[33] "Proof of work - Bitcoin Wiki," https://en.bitcoin.it/wiki/Proof_of_work.

[34] J. Katz and Y. Lindell, *Introduction to modern cryptography*. Boca Raton: Chapman & Hall/CRC, 2008.

[35] "Hyperledger Fabric v0.6," http://web.archive.org/web/20160924231627/http://hyperledger-fabric.readthedocs.io/en/latest/protocol-spec.

[36] "Protocol buffers - Google," https://developers.google.com/protocol-buffers/docs/overview, accessed: 2017-01-15.

[37] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology*, C. Pomerance, Ed. Springer Berlin Heidelberg, 1988, pp. 369–378.

[38] "gRPC guide," http://www.grpc.io/docs/guides/index.html, accessed: 2017-01-15.

[39] F. B. Schneider, "Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[40] C. H. Papadimitriou and P. C. Kanellakis, "On Concurrency Control by Multiple Versions," *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 89–99, Mar. 1984.

[41] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform," in *IEEE MASCOTS*, 2018.

[42] J. Kreps, N. Narkhede, and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[43] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX Annual Technical Conference*, 2010.

[44] K. Christidis, "A Kafka-based Ordering Service for Fabric," https://docs.google.com/document/d/1vNMaM7XhOlu9tB_10dKnlrhy5d7b1u8lSY8a-kVjCO4, accessed: 2017-12-11.

[45] *Hyperledger Blockchain Performance Metrics White Paper*, v1.0 ed. The Linux Foundation, Oct. 2018. [Online]. Available: https://www.hyperledger.org/resources/publications/blockchain-performance-metrics

[46] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A Scalable Blockchain Protocol," in *Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 45–59.

[47] "How should i handle blockchain forks in my (ethereum) dapp?" https://ethereum.stackexchange.com/questions/183/how-should-i-handle-blockchain-forks-in-my-dapp/.

[48] H. Sukhwani, "Plots for PSWG's metrics documents," https://github.com/tallharish/PerfMetricsPSWG.

[49] SWIFT, "gpi real-time Nostro Proof of Concept," https://www.swift.com/resource/gpi-real-time-nostro-proof-concept, 2018.

[50] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *International Workshop on Software and Performance*, ser. WOSP '00, 2000, pp. 195–203.

[51] M. D. Hill, "What is scalability?" *SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 18–21, Dec. 1990.

[52] L. Duboc, D. S. Rosenblum, and T. Wicks, "A framework for modelling and analysis of software systems scalability," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 949–952.

[53] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[54] C. Decker and R. Wattenhofer, "Information propagation in the Bitcoin network," in *IEEE P2P 2013*, 2013, pp. 1–10.

[55] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the Security and Performance of Proof of Work Blockchains," in *ACM SIGSAC CCS*, 2016, pp. 3–16.

[56] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking Scalability and Elasticity of Distributed Database Systems," *VLDB*, vol. 7, no. 12, pp. 1219–1230, Aug. 2014.

[57] 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 *ACM International Conference on Management of Data*, ser. SIGMOD, 2017, pp. 1085–1100.

[58] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance Characterization of Hyperledger Fabric," in *Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018.

[59] "IBM Watson IoT Track and Trace contract - GitHub," https://github.com/ibm-watson-iot/blockchain-samples/tree/master/contracts/industry/trackandtrace, accessed: 2018-08-17.

[60] "IBM Watson IoT Contract Platform - GitHub," https://github.com/ibm-watson-iot/blockchain-samples/tree/master/contracts/platform/iotcontractplatform, accessed: 2017-04-15.

[61] NIST, "Kolmogorov-Smirnov Goodness-of-fit test," http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm, accessed: 2017-04-15.

[62] H. Akaike, "A New Look at the Statistical Model Identification," *IEEE Trans. on Auto. Control*, vol. 19, no. 6, pp. 716–723, Dec 1974.

[63] T. Hothorn and B. S. Everitt, *A Handbook of Statistical Analyses using R*. Boca Raton, FL: CRC Press/Taylor & Francis Group, 2014.

[64] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance Analysis of Private Blockchain Platforms in Varying Workloads," in *IEEE International Conference on Computer Communication and Networks (ICCCN)*, July 2017, pp. 1–6.

[65] G. Ciardo, J. K. Muppala, and K. S. Trivedi, "SPNP: Stochastic Petri Net Package," in *Petri Nets and Performance Models*, 1989, pp. 142–151.

[66] "AWS Inter-Region EC2 Latency," https://www.concurrencylabs.com/blog/choose-your-aws-region-wisely/, accessed: 2017-04-23.

[67] "Latency between AWS global regions," https://web.archive.org/web/20161107091957/http://zhiguang.me/2016/05/10/latency-between-aws-global-regions/, accessed: 2017-04-10.

[68] "Verizon Enterprise Solutions Latency Statistics," http://www.verizonenterprise.com/about/network/latency/#pip.

[69] A. Abdou, A. Matrawy, and P. C. van Oorschot, "Accurate one-way delay estimation with reduced client trustworthiness," *IEEE Communications Letters*, vol. 19, no. 5, pp. 735–738, May 2015.

[70] R. Halalai, T. A. Henzinger, and V. Singh, "Quantitative Evaluation of BFT Protocols," in *International Conference on Quantitative Evaluation of Systems (QEST)*, Sept 2011, pp. 255–264.

[71] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine Fault Tolerant Systems tolerate Byzantine Faults," in *USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09, 2009, pp. 153–168.

[72] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *ACM SIGOPS SOSP*, 2007, pp. 45–58.

[73] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State Machine Replication for the Masses with BFT-SMaRt," in *IEEE/IFIP DSN*, 2014, pp. 355–362.

[74] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX*. USENIX Association, 2014, pp. 305–319.

[75] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network)," in *IEEE International Symposium on Network Computing and Applications (NCA)*, 2018, ©2018 IEEE.

[76] "Does Hyperledger Fabric need Docker?" https://stackoverflow.com/questions/48070380/does-hyperledger-fabric-need-docker.

[77] J. P. Buzen and P. J. Denning, "Measuring and Calculating Queue Length Distributions," *IEEE Computer*, vol. 13, no. 4, pp. 33–44, April 1980.

[78] J. Sousa, A. Bessani, and M. Vukolic, "A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform," in *IEEE/IFIP DSN*, 2018, pp. 51–58.

[79] M. Vukolić, *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*. Cham: Springer International Publishing, 2016, pp. 112–125.

[80] I. Weber, V. Gramoli, A. Ponomarev, M. Staples, R. Holz, A. B. Tran, and P. Rimba, "On Availability for Blockchain-Based Systems," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2017, pp. 64–73.

[81] "Emerald - proving Ethereum for the Clearing Use Case," https://emerald-platform.gitlab.io/static/emeraldTechnicalPaper.pdf.

[82] M. F. Neuts, *Phase-type Probability Distributions*. Boston, MA: Springer US, 2013, pp. 1132–1134.

[83] K. S. Trivedi, "SPNP User's Manual – Version 6.0," 1999.

[84] Y. Hochberg and A. C. Tamhane, *Multiple Comparison Procedures*. John Wiley & Sons, Inc., 1987.

[85] K. Trivedi, *Probability & Statistics with Reliability, Queueing & Computer Science applications*, 2nd ed. Wiley, 2001.

[86] S. Ramani, K. S. Trivedi, and B. Dasarathy, "Performance Analysis of the CORBA Event Service using Stochastic Reward Nets," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2000, pp. 238–247.

[87] E. Bauer, X. Zhang, and D. A. Kimber, *Practical System Reliability*. Wiley-IEEE Press, 2009.

[88] H. Sukhwani, A. Bobbio, and K. S. Trivedi, "Largeness Avoidance in Availability Modeling using Hierarchical and Fixed-point Iterative Techniques," *International Journal of Performability Engineering*, vol. 11, no. 4, pp. 305–319, July 2015.

[89] S. Ramani, K. S. Trivedi, and B. Dasarathy, "Performance Analysis of the CORBA Notification Service," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2001, pp. 227–236.

[90] N. Papadis, S. Borst, A. Walid, M. Grissa, and L. Tassiulas, "Stochastic Models and Wide-Area Network Measurements for Blockchain Design and Analysis," in *IEEE INFOCOMM*, 2018.

[91] J. Göbel, H. Keeler, A. Krzesinski, and P. Taylor, "Bitcoin blockchain dynamics: The selfish-mine strategy in the presence of propagation delay," *Performance Evaluation*, vol. 104, pp. 23 – 41, 2016.

[92] I. Kocsis, A. Pataricza, M. Telek, A. Klenik, F. Deé, and D. Cseh, "Towards Performance Modeling of Hyperledger Fabric," in *International IBM Cloud Academy Conference (ICACON)*, 2017.

[93] R. G. Sargent, "Verification and Validation of Simulation Models," in *Winter Simulation Conference (WSC)*, Dec 2005, pp. 14 pp.–.

[94] T. H. Naylor and J. M. Finger, "Verification of computer simulation models," *Management Science*, vol. 14, no. 2, pp. B–92–B–101, 1967.

[95] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric," *CoRR*, vol. abs/1805.08541, 2018.

[96] K. O'Dwyer and D. Malone, "Bitcoin Mining and its Energy Footprint." The Institution of Engineering & Technology, 2014.

[97] S. Sankaran, S. Sanju, and K. Achuthan, "Towards realistic energy profiling of blockchains for securing internet of things," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, July 2018, pp. 1454–1459.

[98] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by Step towards creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab," in *Financial Cryptography and Data Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 02 2016, vol. 9604, pp. 79–94.

[99] "IEEE Blockchain Standards," https://blockchain.ieee.org/standards, accessed: 2018-09-20.

[100] *Blockchain and electronic distributed ledger technologies.* International Organization for Standardization (ISO), 2016, no. ISO/TC 307.

[101] "Focus Group on Application of Distributed Ledger Technology," https://www.itu.int/en/ITU-T/focusgroups/dlt/Pages/default.aspx, accessed: 2018-09-20.

Biography

Harish Sukhwani received his B.E. degree in Electronics & Telecommunication Engineering from Thadomal Shahani Engineering College, University of Mumbai in 2007, and his M.S. degree in Electrical Engineering (Computer Networks) from the University of Southern California in 2010. Prior to joining the Ph.D. program, he worked in the industry for $2\frac{1}{2}$ years as a Software Engineer for Cisco Systems in RTP, NC. During the Ph.D. program, he interned with NetApp Inc., IBM Corporation and IBM Research - Zurich. Harish received IBM Ph.D. Fellowship for the year 2016-17 and 2017-18. He received the First Patent Application Invention Achievement Award at IBM in 2017. He defended his Ph.D. dissertation on November 13, 2018.

His research interests are stochastic modeling, software performance & reliability, blockchain networks, Internet of Things (IoT). He is interested in pursuing a career in the industry, where he is interested in putting cutting-edge analytical research in practice to solve practical problems. His scientific publications are listed below:

1. H. Sukhwani, Nan Wang, Kishor S. Trivedi, and Andy Rindos. Performance Modeling of Hyperledger Fabric (Permissioned Blockchain Network). In *IEEE International Symposium on Network Computing and Applications (NCA)*, 2018.
2. H. Sukhwani, José M. Martínez, Xiaolin Chang, Kishor S. Trivedi, and Andy Rindos. Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255, Hong Kong, Sept.

2017.

3. H. Sukhwani, Rivalino Matias, Kishor S. Trivedi, and Andy Rindos. Monitoring and Mitigating Software Aging on IBM Cloud Controller System. In *IEEE International Workshop on Software Aging and Rejuvenation (WoSAR)*, pages 266–272, Toulouse, France, Oct. 2017
4. H. Sukhwani, Javier Alonso, Kishor S. Trivedi, and Isaac McGinnis. Software Reliability Analysis of NASA Space Flight Software: A Practical Experience. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 386–397, Vienna, Austria, Aug 2016.
5. H. Sukhwani, Andrea Bobbio, and Kishor S. Trivedi. Largeness Avoidance in Availability Modeling using Hierarchical and Fixed-point Iterative Techniques. *International Journal of Performability Engineering*, 11(4):305–319, July 2015.
6. Rekha Singhal, Manoj Nambiar, H. Sukhwani, and Kishor S. Trivedi. Performability Comparison of Lustre and HDFS for MR Applications. In *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 51–51, Nov 2014.

He participates in the Hyperledger Performance & Scalability Working Group (PSWG), where he collaborated extensively to develop the following whitepaper

7. *Hyperledger Blockchain Performance Metrics White Paper*, v1.0 ed. The Linux Foundation, Oct. 2018. [Online]. Available: `https://www.hyperledger.org/resources/publications/blockchain-performance-metrics`