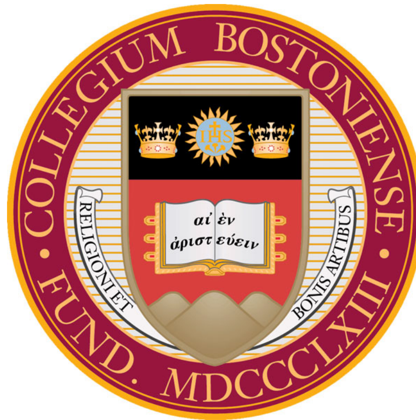
Learning Experiences on Blockchain Related Projects



Xuheng Duan

Computer Science Department

Boston College

Supervisor

Lewis Tseng

In partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science

May 15, 2020

Abstract

Today, blockchain technology becomes a hot topic and attracts people's attention worldwide. Initially, blockchain was introduced in 2008 by Satoshi Nakamoto as a secure system for bitcoin. Today, there are public blockchain projects like Ethereum and Bitcoin that enables anyone to process secure peer-to-peer transactions. Meanwhile, there are also private blockchain projects like Hyperledger Fabric[1], which restricts the participants and realizes secure communications between corporations.

Throughout the past year, I studied the Hyperledger blockchain system and encountered numerous difficulties. Moreover, witnessing so many different sorts of blockchain projects, one would like to understand and compare different blockchain performance at scale. Thus, I also studied three different benchmark tools, namely Blockbench[2], Caliper++[3] and Hyperledger Caliper[4], and focused on the latter two. At last, we used Mininet[5] to emulate a virtual network and tested the performance of Ethereum. Overcoming the struggles, I document my learning outcomes and share my experience on these open-source projects with those who would like to engage the field of blockchain.

Contents

1 Introduction **1**
1.1 Hyperledger Fabric . . . 1
1.2 Hyperledger Caliper . . . 2
1.3 Caliper++ . . . 2
1.4 Mininet and Ethereum . . . 3

2 Hyperledger Fabric **4**
2.1 Structure and concepts . . . 4
2.1.1 Organization and Peers . . . 4
2.1.2 Orderer and Channel . . . 5
2.1.3 Certificate Authority(CA) and Membership Service Provider (MSP) . . . 5
2.2 Related Work and Suggestions . . . 6

3 Hyperledger Caliper and Caliper++ **10**
3.1 Structure and concepts . . . 10
3.2 Related Work and Notes . . . 11

4 Ethereum and Mininet **14**
4.1 Ethereum . . . 14
4.2 Basic structure . . . 14
4.3 Related Work . . . 15
4.4 Data and Results . . . 16
4.4.1 Network Latency . . . 16
4.4.2 Different Topology and Network Failures . . . 18
4.4.3 Heterogeneous Machines . . . 19
4.4.4 Large-scale Test . . . 19

5 Other Errors **20**

6 Summary **21**

References **22**

1. Introduction

The thesis includes four different major parts, corresponding to four different systems: Hyperledger Fabric, Hyperledger Caliper, Caliper++, and Mininet/Ethereum. In the introduction part, I will briefly talk about the basic concepts and structures of the four systems. Then, in the major sections, I included detailed discussion, some related work, and working notes. Lastly, there is a short summary at the end of the thesis.

1.1 Hyperledger Fabric

Admittedly, traditional public blockchain systems have proven their utility. However, many enterprise use cases require a private environment in which the permissionless blockchain project could not provide. For instance, in finance transactions, users need to know each other's identities in order to avoid money laundering or financial fraud. Although many developers adapted earlier public blockchains for business scenarios, the projects still have some inherent problems. As a solution, Hyperledger Fabric[6] was designed for enterprise use, combining novel and unique philosophies. It aims to build a secure network where all participants must be identified. Older blockchain projects like bitcoin have issues of low transaction throughput and high latency of transactions because it relies on miners to package and verify the data. However, Hyperledger Fabric designs orderers to arrange the transactions, which brings the transaction speed to a new level. Using modular architecture and smart contracts, Hyperledger Fabric also makes it easy for participants to customize the network.

1.2 Hyperledger Caliper

Hyperledger Caliper[7] is one of the official benchmark frameworks. Treating the blockchain system as a whole, Caliper can target the SUT(System Under Test) by user-defined chaincodes and integrate the system responses into a meaningful report. Caliper consists of two important configuration files: benchmark files and network files. In a benchmark file, users can define different parameters they want, such as transaction numbers and transactions per second. Furthermore, users are able to customize their own chaincode and plug it in the Caliper, which would be called later during benchmark phase. On the other hand, Caliper uses a network configuration file to contact the SUT. The network file usually defines the network topology, nodes' endpoints, and smart contracts Caliper should deploy or interact with.

1.3 Caliper++

Although Caliper version 0.2 is powerful, it still has some disadvantages. For instance, it could only perform on local, predefined networks. It is also hard to run on large-scale and distributed networks. In addition, Caliper has a benchmark client closely connected to Fabric transaction workflow, so the client would need to wait for endorsing peers before they validate the responses, which would potentially influence the benchmark results. Thus, another development team designed Caliper++ that extended the functions of the original Caliper. The redesigned Caliper++ supports additional functionalities for benchmark Kafka-based[8] Fabric network at scale. To be more specific, it contains scripts that could generate configuration files for Fabric network topology and brings up a large scale of network across any number of cluster nodes. It successfully solved the problem that hindered Caliper and brought new ideas to design a distributed benchmark framework.

1.4 Mininet and Ethereum

Mininet[9] is a battle-tested software that is widely used in prototyping and evaluating Software-Defined Networks(SDNs). It has the capability to illustrate realistic simulation of physical network topology, and it also supports many extensible configurations. Another world-famous Blockchain project, Ethereum, supports customized applications. We deployed Go-Ethereum(Geth) network on the mininet and raised some meaningful results.

2. Hyperledger Fabric

2.1 Structure and concepts

A typical Hyperledger Fabric transaction will undergo the following steps: the peer initializes a transaction and uses the user's cryptographic credentials to sign the transaction proposal. Then, the transaction will be sent to the orderer, and the orderer would verify and arrange the transaction, sending them back to peers as well as commiting the verified transaction to the world state.

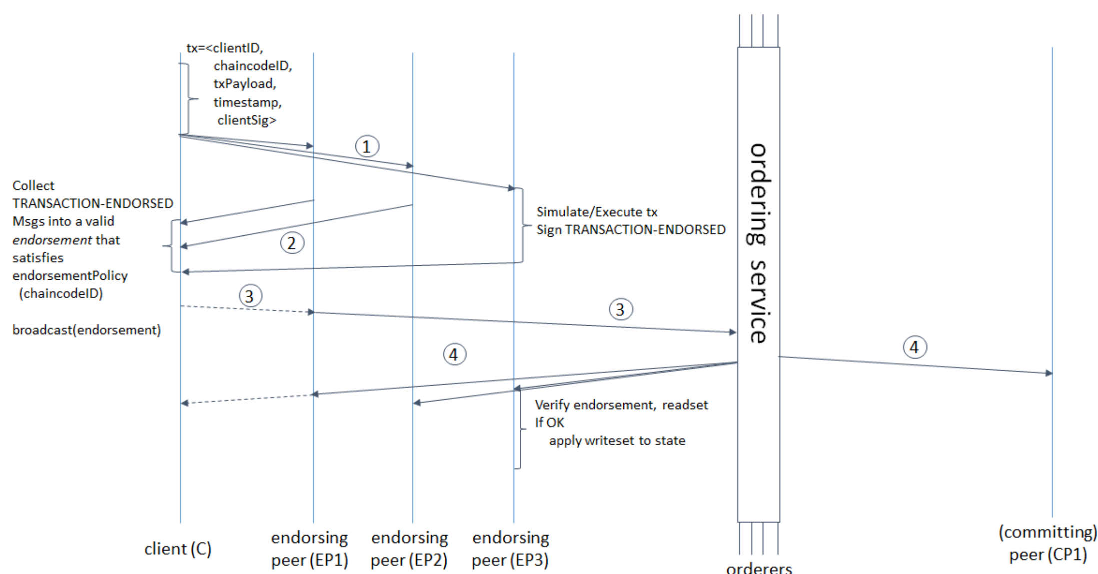


Figure 2.1 Hyperledger Fabric Workflow

In this section, I will discuss in detail the various components of Hyperledger Fabric: organization, peers, orderer, channel, CA and MSP.

2.1.1 Organization and Peers

Fabric network contains organizations, which is also known as the consortium. Under organizations, the network deploys peers, which are the fundamental units

of Hyperledger Fabric. The peers usually are assigned names in the format of peerX.orgX.example.com, where X is a number. Similar to miners in public blockchain systems, Hyperledger Fabric uses peer units to accomplish validating transactions and to commit data to the ledger. Meanwhile, the peers also run chaincodes(smart contracts), which contain user-defined codes, to manage the assets.

2.1.2 Orderer and Channel

Different from public blockchain systems, Fabric uses orderers to complete consensus work. After peer units have completed validating the transactions, orderers would receive the endorsed blocks from peers, and then start to reach a consensus. Eventually, orderer nodes would add the blocks to the ledger, finishing the entire process. Peers could update their own data from the world state at other times. On top of orderers, a network needs multiple channels to operate. Channels give Hyperledger Fabric the ability to build private communications between desired organizations while not interfering with the other organizations. As a result, a channel creates a black box and private ledger for consortiums that are invited to the channel.

2.1.3 Certificate Authority(CA) and Membership Service Provider (MSP)

CA and MSP are used to verify one's identity in Hyperledger Fabric. The CA would issue cryptographic materials for the network components, like orderers, organizations, and peers. These materials, usually a pair of public and private keys, would be later used to enroll components' admin, as well as peers. After the network generates cryptographic materials, it also needs MSPs to identify which certificates are needed and to issue valid identities for their members. Using the metaphor I learned before, certificates are similar to the credit cards that can tell whether or not an individual is able to pay(and this is created by the bank, which is parallel to CA in Hyperledger Fabric), and MSPs are the list of accepted credit cards that individual can use in a store.

2.2 Related Work and Suggestions

To successfully launch a functional Hyperledger Fabric network, one needs to address several components and set up the running environment correctly: prerequisites, crypto materials, channel configurations, and chaincode. After we successfully install the chaincode on the Fabric network, we could query against our ledgers and retrieve the data.

Some necessary packages are:

- Git: used to download binaries and repositories from Github
- Docker and Docker-Compose: used to establish Fabric network
- Go Language/Java/Javascript/Node/Python: Hyperledger Fabric supports different languages.
- Npm: for package management

Check each package's version to make sure they are installed correctly.

```
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ git version
git version 2.7.4
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ docker version
Client: Docker Engine - Community
 Version:           19.03.9
 API version:       1.40
 Go version:        go1.13.10
 Git commit:        9d988398e7
 Built:             Fri May 15 00:25:34 2020
 OS/Arch:           linux/amd64
 Experimental:      false

Server: Docker Engine - Community
 Engine:
  Version:          19.03.9
  API version:      1.40 (minimum version 1.12)
  Go version:       go1.13.10
  Git commit:       9d988398e7
  Built:            Fri May 15 00:24:07 2020
  OS/Arch:          linux/amd64
  Experimental:     false
 containerd:
  Version:          1.2.13
  GitCommit:        7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:          1.0.0-rc10
  GitCommit:        dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:          0.18.0
  GitCommit:        fec3683
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ docker-compose version
docker-compose version 1.25.5, build 8a1c60f6
docker-py version: 4.1.0
CPython version: 3.7.5
OpenSSL version: OpenSSL 1.1.0l  10 Sep 2019
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ go version
go version go1.13.4 linux/amd64
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ npm version
{ npm: '3.5.2',
  ares: '1.10.1-DEV',
  http_parser: '2.5.0',
  icu: '55.1',
  modules: '46',
  node: '4.2.6',
  openssl: '1.0.2g',
  uv: '1.8.0',
  v8: '4.5.103.35',
  zlib: '1.2.8' }
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ python --version
Python 2.7.12
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$
```

Figure 2.2 Prerequisites for Hyperledger Fabric ver2.0

Then, we need to generate crypto materials using cryptogen and configtxgen tools. Cryptogen tool consumes a YAML file, named crypto-config.yaml, which contains the topology of the network, and output a folder that contains the certificates and keys for orderers. On the other hand, configtxgen tool consumes configtx.yaml, which contains the definition for the network, and output the genesis block of the fabric network. The complicated part is generating configurations for channels. Basically, for one single channel, we need to do the followings: generate channel transaction artifact for the channel, and update anchor peers for the channel members. In our case, we need to update the anchor peers for organization 1 and organization 2. Later, we will use these materials to create channels and interact with them.

```
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ export CHANNEL_NAME=mychannel  && ../bin/configtxgen -profile TwoOrgsChannel -o
utputCreateChannelTx ./channel-artifacts/channel.tx -channelID $CHANNEL_NAME
2020-05-22 08:25:21.768 UTC [common.tools.configtxgen] main -> INFO 001 Loading configuration
2020-05-22 08:25:21.801 UTC [common.tools.configtxgen.localconfig] Load -> INFO 002 Loaded configuration: /home/blockbenchsummergroup/fabric-sam
ples/first-network/configtx.yaml
2020-05-22 08:25:21.801 UTC [common.tools.configtxgen] doOutputChannelCreateTx -> INFO 003 Generating new channel configtx
2020-05-22 08:25:21.804 UTC [common.tools.configtxgen] doOutputChannelCreateTx -> INFO 004 Writing new channel tx
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ ../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-a
rtifacts/Org1MSPanchors.tx -channelID $CHANNEL_NAME -asOrg Org1MSP
2020-05-22 08:25:45.443 UTC [common.tools.configtxgen] main -> INFO 001 Loading configuration
2020-05-22 08:25:45.476 UTC [common.tools.configtxgen.localconfig] Load -> INFO 002 Loaded configuration: /home/blockbenchsummergroup/fabric-sam
ples/first-network/configtx.yaml
2020-05-22 08:25:45.476 UTC [common.tools.configtxgen] doOutputAnchorPeersUpdate -> INFO 003 Generating anchor peer update
2020-05-22 08:25:45.477 UTC [common.tools.configtxgen] doOutputAnchorPeersUpdate -> INFO 004 Writing anchor peer update
```

Figure 2.3 Channel Configurations for Hyperledger Fabric ver2.0

Hyperledger Fabric relies on Docker to spin up the network. Fabric will initialize a number of Docker containers to simulate the peers in the network. At this point, we have already started a Hyperledger Network. However, the network will not be functional unless we also set up the channels and install the chaincodes.

Every time we want to create a new channel, we have to first define the channel in configtx.yaml, then use configtxgen tool to generate the channel configuration files based on the materials we defined in configtx.yaml, and ultimately create the channel with the specific channel configuration files.

Finally, the Fabric network interacts with the ledger through chaincodes, and we need to install chaincodes on all the participants of the channel before we can use them. Hyperledger Fabric supports different languages for the chaincodes(Java, Go and Node.js). Notice that all the peers on the same channel should agree with the same chaincode package. Figure 2.4 and 2.5 show the installation of chaincodes on the network. In my case, we can query the chaincodes to show the quantity of our account and move the desired amount of value from one account to the other one. Initially, both account a and b have value of 100.

```
blockbenchsummergroup@instance-2:~/fabric-samples/first-network$ docker exec -it cli bash
bash-5.0#
bash-5.0# CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.
example.com/msp
bash-5.0# CORE_PEER_ADDRESS=peer0.org1.example.com:7051
bash-5.0# CORE_PEER_LOCALMSPID="Org1MSP"
bash-5.0# CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.o
rg1.example.com/tls/ca.crt
bash-5.0# echo hi
hi
bash-5.0# echo $CHANNEL_NAME

bash-5.0# export CHANNEL_NAME=mychannel
bash-5.0# echo $CHANNEL_NAME
mychannel
bash-5.0# peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/channel.tx --tls --cafile /opt/gopath/src/gith
ub.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
2020-05-22 08:30:48.508 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
2020-05-22 08:30:48.531 UTC [cli.common] readBlock -> INFO 002 Expect block, but got status: &{NOT_FOUND}
2020-05-22 08:30:48.537 UTC [channelCmd] InitCmdFactory -> INFO 003 Endorser and orderer connections initialized
2020-05-22 08:30:48.739 UTC [cli.common] readBlock -> INFO 004 Expect block, but got status: &{SERVICE_UNAVAILABLE}
2020-05-22 08:30:48.743 UTC [channelCmd] InitCmdFactory -> INFO 005 Endorser and orderer connections initialized
2020-05-22 08:30:48.944 UTC [cli.common] readBlock -> INFO 006 Expect block, but got status: &{SERVICE_UNAVAILABLE}
2020-05-22 08:30:48.948 UTC [channelCmd] InitCmdFactory -> INFO 007 Endorser and orderer connections initialized
2020-05-22 08:30:49.149 UTC [cli.common] readBlock -> INFO 008 Expect block, but got status: &{SERVICE_UNAVAILABLE}
2020-05-22 08:30:49.153 UTC [channelCmd] InitCmdFactory -> INFO 009 Endorser and orderer connections initialized
2020-05-22 08:30:49.354 UTC [cli.common] readBlock -> INFO 00a Expect block, but got status: &{SERVICE_UNAVAILABLE}
2020-05-22 08:30:49.358 UTC [channelCmd] InitCmdFactory -> INFO 00b Endorser and orderer connections initialized
2020-05-22 08:30:49.559 UTC [cli.common] readBlock -> INFO 00c Expect block, but got status: &{SERVICE_UNAVAILABLE}
2020-05-22 08:30:49.564 UTC [channelCmd] InitCmdFactory -> INFO 00d Endorser and orderer connections initialized
2020-05-22 08:30:49.768 UTC [cli.common] readBlock -> INFO 00e Received block: 0
bash-5.0#  peer channel join -b mychannel.block
2020-05-22 08:31:19.027 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
2020-05-22 08:31:19.069 UTC [channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
bash-5.0# CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.
example.com/msp CORE_PEER_ADDRESS=peer0.org2.example.com:9051 CORE_PEER_LOCALMSPID="Org2MSP" CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.
com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt peer channel join -b mychannel.blo
ck
2020-05-22 08:31:38.352 UTC [channelCmd] InitCmdFactory -> INFO 001 Endorser and orderer connections initialized
2020-05-22 08:31:38.396 UTC [channelCmd] executeJoin -> INFO 002 Successfully submitted proposal to join channel
```

Figure 2.4 Create and Join Channel for Hyperledger Fabric ver2.0

In figure 2.6, we move 10 values from the account a to b, and then we asked the ledger to see if the final result is correct.

```
bash-5.0# peer lifecycle chaincode install mycc.tar.gz
2020-05-22 08:37:02.441 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 001 Installed remotely: response:<status:200 payload:"\nGmyc
c_1:edabb6e44241ea17f4c699df8f9ac55fc216360a581c52041291df2f00a438f9\022\006mycc_1" >
2020-05-22 08:37:02.441 UTC [cli.lifecycle.chaincode] submitInstallProposal -> INFO 002 Chaincode code package identifier: mycc_1:edabb6e44241ea
17f4c699df8f9ac55fc216360a581c52041291df2f00a438f9
bash-5.0#
```

Figure 2.5 Install Chaincode for Hyperledger Fabric ver2.0

```
bash-5.0# peer lifecycle chaincode checkcommitreadiness --channelID $CHANNEL_NAME --name mycc --version 1.0 --init-required --sequence 1 --tls t
rue --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscace
rts/tlsca.example.com-cert.pem --output json
{
        "approvals": {
                "Org1MSP": true,
                "Org2MSP": true
        }
}
bash-5.0# peer lifecycle chaincode commit -o orderer.example.com:7050 --channelID $CHANNEL_NAME --name mycc --version 1.0 --sequence 1 --init-re
quired --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledg
er/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --t
lsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca
.crt
2020-05-22 08:49:43.097 UTC [chaincodeCmd] ClientWait -> INFO 001 txid [0ee2c2cfc4821102451c3e00bec7ebdfe4694d9db75f46d90f5e9a6d748e1212] commit
ted with status (VALID) at peer0.org2.example.com:9051
2020-05-22 08:49:43.117 UTC [chaincodeCmd] ClientWait -> INFO 002 txid [0ee2c2cfc4821102451c3e00bec7ebdfe4694d9db75f46d90f5e9a6d748e1212] commit
ted with status (VALID) at peer0.org1.example.com:7051
```

Figure 2.6 Install Chaincode for Hyperledger Fabric ver2.0

```
bash-5.0# peer chaincode invoke -o orderer.example.com:7050 --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordere
rOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc --peerAddresses peer0
.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/pee
r0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer
/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"Args":["invoke","a","b","10"]}' --waitForEvent

2020-05-22 08:56:40.815 UTC [chaincodeCmd] ClientWait -> INFO 001 txid [01896db8b4ed7a9e93898fe414754a617d7ed9537420f53040a8b704d0043c8f] commit
ted with status (VALID) at peer0.org2.example.com:9051
2020-05-22 08:56:40.815 UTC [chaincodeCmd] ClientWait -> INFO 002 txid [01896db8b4ed7a9e93898fe414754a617d7ed9537420f53040a8b704d0043c8f] commit
ted with status (VALID) at peer0.org1.example.com:7051
2020-05-22 08:56:40.816 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 003 Chaincode invoke successful. result: status:200
bash-5.0#
bash-5.0# peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
90
```

Figure 2.7 Invoke Chaincode for Hyperledger Fabric ver2.0

Following the official guidance is the most straightforward way to start with Hyperledger Fabric. The official BYFN[10] tutorial explains most of the concepts

one needs to know about setting up a functional Hyperledger Fabric network. However, completing the tutorial does not provide one enough knowledge nor experience to construct a customized network. Hereby, I will include some difficulties I confronted, and I will also shine some lights to way to learn Hyperledger Fabric efficiently.

First, Hyperledger Fabric relies heavily on Docker containers, so it will be helpful to have some understandings of Docker. Specifically, the Docker configuration file plays an important role in defining the Fabric network. Although BYFN provided predefined Docker YAML files, it is still critical for one to understand and adjust the files for specific circumstances. On the other hand, while initialing the network, it is usually helpful to check the log of each container to examine the problems. The Docker documentation[11] consists of everything one needs to know.

Furthermore, if one simply follows the official tutorial, there will be a natural tendency to omit some essential parts of Hyperledger Fabric. When I first tried to set up a customized network, I experienced many problems regarding CA, MSPs, and cryptographic materials for organizations. To name a few, in official BYFN, the example used pre-built binaries(crytogen) to assign certificates. However, one needs to manually generate extra certificates and move them to correct paths under the new hosts when adding new organizations to the network. Without predicting such onerous work, I was stuck by CA and MSP multiple times during the deployment phase, even later in Caliper work. Other than that, I experienced multiple deprecated binaries. Since the version of Hyperledger Fabric documentation does not synchronize with the version of Hyperledger Fabric, it's natural to have the wrong version in some packages.

3. Hyperledger Caliper and Caliper++

3.1 Structure and concepts

Hyperledger Caliper has three major parts, Workload Module, Benchmark configuration file, and network configuration file. Caliper itself does not have any tangible benchmark implementation. Alternatively, workload modules are responsible for generating transactions and submits them against the System Under Test (SUT). These Workload Modules are Node.js codes that utilize outside APIs. Thus, be careful with the Node.js version when one uses Caliper. The current stable version should be either Node.js 8 or Node.js 10. Benchmark configuration file consists of detailed information about the benchmarks, like how it should be executed and what's the desired transaction rate. By default, Hyperledger Caliper contains two types of benchmark scenarios. One of which mimics the bank environment and the other one contains various simple and straightforward tests again SUT, like plain query and open functions. Lastly, network configuration files define the SUT network topology. Since Caliper itself is a benchmark workframe, which does not generate the SUT itself, it is users' responsibility to construct a functional SUT with accessible endpoint addresses. Similarly, Caliper itself utilized files from BYFN.

When I touched Hyperledger Caliper, it was still on version 0.2 and has little documentation on distributed benchmarks. Thus, another development team designed Caliper++[3]. Their contributions include the followings: they extended Hyperledger Caliper by adding support for distributed benchmarking. Furthermore, the development team designed scripts that can start Fabric with varying

sizes and configurations. With Caliper++ the team successfully showed that endorsing peers in the Hyperledger Fabric network with Kafka ordering service is the bottleneck.

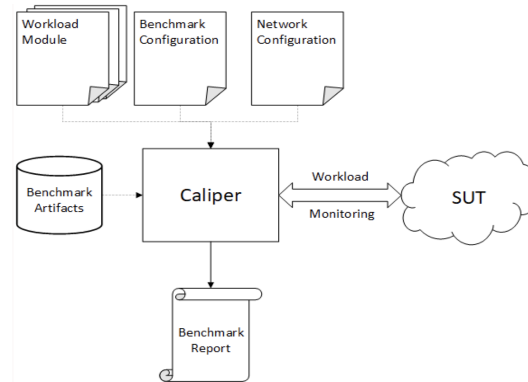


Figure 3.1 The architecture of Caliper

3.2 Related Work and Notes

While reproducing the results in *Understanding the scalability of Hyperledger Fabric*[3], I confronted numerous problems. I successfully resolved some of these difficulties by contacting the author and querying on the technical forums. Hereby, I would like to share these problems here and help successors to avoid them.

Caliper++ is built upon Hyperledger Caliper, and thus it shares the same structure of Hyperledger Caliper. The first problem appeared when I tried to generate certificates. In Caliper++ instructions, it simply said "generate two config files" with yarn genConfig command and bring up the network. This was misleading to me because, in the paper, the author described that their script could automatically generate the configuration file and bring up the Hyperledger Fabric network. Thus, unsurprisingly, I jumped into errors when I simply executed the command. After I conducted with one of the authors, Dumitrel Loghin, it turns out that they established a Hyperledger Fabric network on the cluster, which was not included in the paper. The configuration file merely serves as a connection point between one's network and Caliper++. Therefore, my task shifts to building up a functional Hyperledger Fabric network in a cluster.

Caliper++ used Docker Swarm to link the virtual machines into a united cluster. Unluckily, personally I don't have previous experience on Docker Swarm,

and thus Docker Swarm was not a viable option to me. But, there could be multiple ways to build up a cluster for the Hyperledger Fabric network. To establish a cluster, I chose between Google Cloud Platform and virtual machines as nodes. After setting up the machines, I need to connect these nodes together, making sure they could communicate with each other. In my case, I set up two virtual machines in VirtualBox, with Ubuntu 16.04 LTS. [12] and [13] guided me complete this task. Meanwhile, users need to emulate the network so that virtual machines could connect to the internet and receive an external IP address. [14] showed how to set up network cards. After ensuring that both machines could talk with each other by Secure Socket Shell(ssh), I used example configuration files from Hyperledger Fabric official site to construct the network. Everything was smooth and neat until when I tried to adjust the YAML files into a more complex network topology. Although articles like [15] and [16] explained how to configure one's own Hyperledger Fabric network, unfortunately, I could not resolve the problems I encountered later.

Hyperledger Fabric network needs certificates on every host. Therefore, before using Caliper[4] or Caliper++, users need to assign the Hyperledger Fabric artifacts, including certificates, crypto information, and other binaries to every host and under correct paths. Manually allocating these materials is unrealistic, and I did not figure out an easy way that could help me to do that. On the other hand, making sure hosts in the distributed system could contact each other properly is challenging. Most of the time, connection requests from one host will be denied by another one due to diverse problems. There is no easy way to solve them, except by carefully examining the bug and getting help from others.

Figure 3.2 illustrates a typical report generated by Hyperledger Caliper. The report would express the SUT on the right side of the page, under "System Under Test" section. In the figure, I targeted Caliper against Hyperledger Caliper version 1.4.1 and set up the network topology with 2 organizations, one peer each. The orderer type was solo, and the network was on a single host. Finally, the world state was maintained by GoLevel Database. On the other hand, the report section shows us what kind of benchmark was performed by the chaincode. In my case, I simply queried the ledger, fetched the responses, and calculated the latency.

Caliper Report

Basic information

DLT: fabric
Benchmark: simple
Description: This is an example benchmark for caliper, to test the backend DLT's performance with simple account opening & querying transactions
Test Rounds: NaN
Details

Benchmark results

Summary
open

System Under Test

Version: 1.4.1
Size: 2 Orgs with 1 Peer
Orderer: Solo,
Distribution: Single Host
StateDB: GoLevelDB
Details

Summary

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
open	100	0	1.0	2.37	0.31	0.39	1.0

open

Test description for the opening of an account through the deployed chaincode

round 0

Performance metrics

Name	Succ	Fail	Send Rate (TPS)	Max Latency (s)	Min Latency (s)	Avg Latency (s)	Throughput (TPS)
open	100	0	1.0	2.37	0.31	0.39	1.0

Resource consumption

Type	Name	Memory(max)	Memory(avg)	CPU% (max)	CPU% (avg)	Traffic In	Traffic Out	Disc Read
Docker	dev-peer0.org1.example.com-smallbank-v0	6.4MB	6.4MB	0.01	0.00	882B	672B	0B
Docker	dev-peer0.org2.example.com-smallbank-v0	6.4MB	6.4MB	0.01	0.00	840B	630B	0B

Figure 3.2 Hyperledger Caliper report

4. Ethereum and Mininet

Finally, we did some work on Ethereum and Mininet[9]. Mininet is a software that creates realistic virtual networks and enables users to adjust related parameters inside of it. We measured how some common factors, like bandwidth, delay, jitter and network loss influence the ethereum network.

4.1 Ethereum

Ethereum[17], an open-source, blockchain-based project, features its function of smart contract that allows users to deploy decentralized customized applications on the Ethereum network. Similar to its brother, Bitcoin, Ethereum also supports its corresponding cryptocurrency, Ether. In our work, we used one of the Ethereum implementation, Go-Ethereum[18], also known as Geth, to build up our projects. We want to detect how the Ethereum network will be influenced under different network conditions.

4.2 Basic structure

We choose Mininet[9] to simulate the network, and further adapt the network with Ethereum. Thus, the entire structure looks like this: running Mininet on a virtual machine (Google Cloud Platform), the single machine provides the basic environment. Then, we built an Ethereum environment on the top of it, which includes hosts and miners. In addition, users could configure configurations of mininet in a single YAML file. We predefined several topologies, including single switch, linear and fat-tree topology. Furthermore, mininet provides many potentials to control the network environment, and we chose several most significant ones from them,

which are bandwidth, delay, jitter, and network loss.

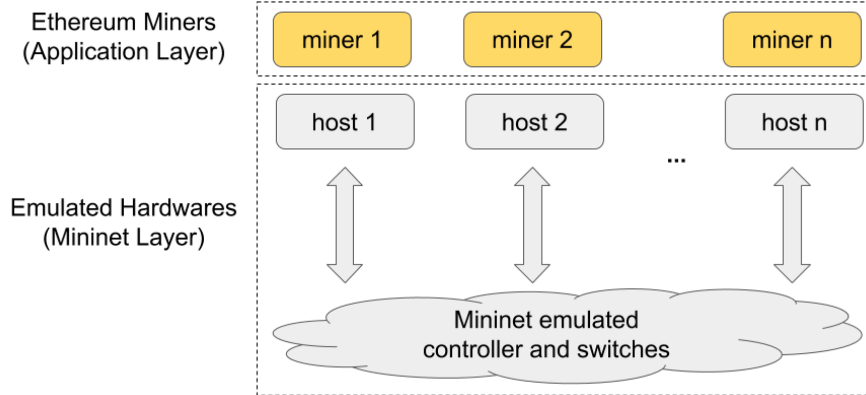


Figure 4.1 The architecture of the project

4.3 Related Work

We created two virtual instances on Google Cloud Platforms to perform our evaluations. The source code could be found on github[19]. Both instances equipped Ubuntu 18.04.4 LTS as their operating system. One has 16 vCPU and the other one gets 96 vCPU, aiming for large-scale network tests. First of all, benchmarking work requires root permission, and the related repositories should be put under the root directory. Check figure 4.2 for a visual example. Then, we created several scripts to help the benchmark process. Under nw3/mngeth/, there is a script named setup.sh, which will install the prerequisites and prepare the environment for the user. Mainly, the script will install important packages like Golang, Go-Ethereum, Python, and Mininet.

Users could change the network factors and running times in configuration files. To be more specific, v2_config.yaml file controls mininet topology and configurations, and v2_start.py contains the code to get the topology and instantiate the benchmark. Users could easily modify the topology under "class" tag and adjust the network factors of links in v2_config.yaml file. Right now, the script supports bandwidth, delay, jitter, and loss. The run.sh script will fire up the framework. The run.sh script will further call the prerun2.sh and analysis.py. In prerun2.sh, the script will initialize the Geth network based on the genesis block we defined in

the json file, and miners. Finally, analysis.py will fetch the log data and calculate the result. Check figure 4.3 for a sample result we get.

```
root@instance-gc20
ssh.cloud.google.com/projects/composed-region-242018/zones/us-e
blockbenchsummergroup@instance-gc20-525:~$ sudo su
root@instance-gc20-525:/home/blockbenchsummergroup# cd
root@instance-gc20-525:~# ls
go  go-ethereum-1.9.12  mininet  nw3  oldNw3
root@instance-gc20-525:~#
```

Figure 4.2 Enter Root Directory

```
*****report*****
[overall]        runtime         808.68 (sec)
[overall]        total_valid     549 (blocks)
[overall]        throughput      0.68 (blocks/sec)
[overall]        latency average 12.02 (sec)
[overall]        latency median  11.66 (sec)
[overall]        latency 95th    19.13 (sec)
[10.0.0.1]       throughput      0 (blocks/sec)
[10.0.0.1]       latency average 0 (sec)
[10.0.0.1]       latency median  0 (sec)
[10.0.0.1]       latency 95th    0 (sec)
[10.0.0.2]       throughput      0.0 (blocks/sec)
[10.0.0.2]       latency average 10.24 (sec)
[10.0.0.2]       latency median  10.24 (sec)
[10.0.0.2]       latency 95th    10.24 (sec)
[10.0.0.3]       throughput      0.0 (blocks/sec)
[10.0.0.3]       latency average 12.35 (sec)
[10.0.0.3]       latency median  12.38 (sec)
```

Figure 4.3 Sample Result from Analysis.py

4.4 Data and Results

4.4.1 Network Latency

We first test the network latency on the impact of throughput. Each experiment is conducted in a Geth network with 4 miners and 3 threads each, for 150 rounds.

Here the throughput is measured as the number of blocks that are successfully appended to the main (canonical) chain per second. Latency is measured as the duration between (i) the time a miner reports it has mined a potential block; and (ii) the time the miner reports its block has reached the canonical chain. In this set of experiments, we use a single switch topology.

Figure 4.4 describes the impact of added latency(link latency) on network latency.

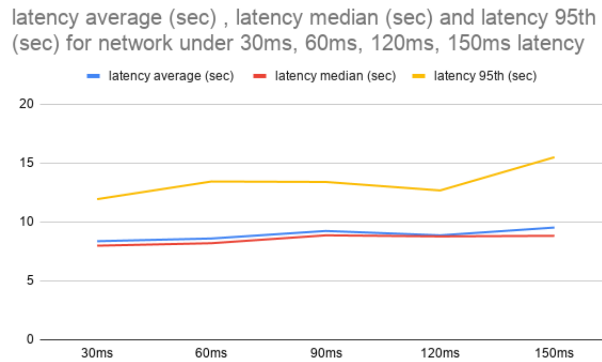


Figure 4.4 Geth Latency vs. Link Latency

Meanwhile, figure 4.5 presents the throughput under added latency (inside Mininet) from 30ms to 150ms. The latency between the host and the switch is called the "added latency" that we use Mininet to simulate. We can see that the throughput is quite stable with moderate latency. Then we enlarged the added latency to 500ms and 1000ms. Check figure 4.6 for more details.

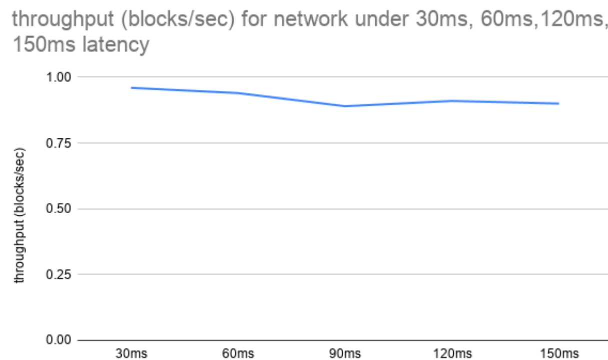


Figure 4.5 Geth Throughput vs. Link Latency

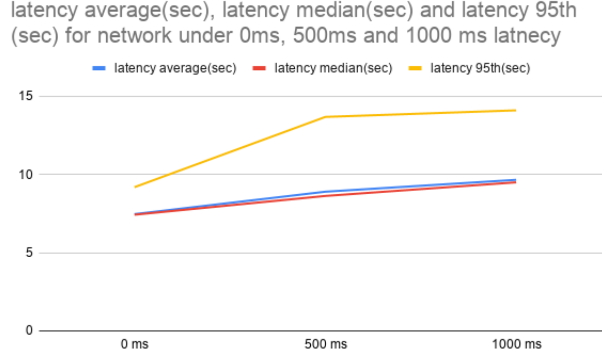


Figure 4.6 Geth Latency vs. Large Link Latency

4.4.2 Different Topology and Network Failures

Other than normal network delay, we also tested the system with various hardware-level topologies and different failure patterns.

Table 4.1 presents Geth's latency and throughput numbers in different topology with 2% and 4% packet loss rate. Each topology has 5 miners with each link having 500ms delay and 10ms jitter. The FatTree topology has three layers (core, aggregate, and edge) and one can adjust the number of switches in each layer and the number of hosts under each edge switch. For links, one can tune the network delay, jitter, loss, and bandwidth constraint, and for hosts, one can configure the CPU constraint and the maximum number of physical cores to use. Our result indicate that fat-tree, while suffering slightly higher latency, is more robust to network loss. We have not observed similar study and analysis before.

Table 4.1: Geth performance vs. Network Failures

	Fat-Tree 2%loss	Fat-Tree 4%loss	Linear 2%loss	Linear 4%loss
throughput (blocks/sec)	0.82	0.8	0.82	0.76
avg. latency (sec)	9.83	9.69	9.53	9.67
med. latency (sec)	9.29	9.17	8.86	9.17
95th latency (sec)	15.14	14.93	15.24	14.48

4.4.3 Heterogeneous Machines

Table 4.2 presents Geth's latency and throughput numbers with heterogeneous hosts, i.e., each host has a different computation power. In this particular evaluation, we have a Fat-tree topology with 500ms added latency per link, no jitter, and no package loss. The relative computation power of each host is listed in the table. The evaluation runs for 3153.27s, and the whole system generates 2023 valid blocks.

Table 4.2: Geth's performance with heterogeneous machines

	Overall	host1	host2	host3	host4	host5
relative comp. power	1	0.1	0.1	0.27	0.27	0.26
throughput (blocks/sec)	0.64	0.02	0.01	0.27	0.01	0.33
avg. latency (sec)	24.23	268.31	330.15	13.35	17.27	10.27
med. latency (sec)	11.16	270.8	332	12.46	18.36	9.66
95th latency (sec)	28.78	424.79	511.6	22.71	24.03	16.41

4.4.4 Large-scale Test

Finally, we extend our framework on a virtual instance with 96 vCPUs and 360 GB memory. The network adopts a Fat-Tree topology with 30 hosts, 500 ms added latency per link, no jitter, and no package loss. Table below presents the result. As expected, the throughput is similar regardless of the number of hosts due to the design of PoW protocol with a fixed difficulty. Table 4.3 shows the result.

Table 4.3: Large-scale Test

runtime(sec)	808.68
total valid blocks	549
throughput (blocks/sec)	0.68
avg. latency (sec)	12.02
med. latency (sec)	11.66
95th latency (sec)	19.13

5. Other Errors

Here I include some common errors, as well as some general tips for successors:

- Docker Socket: Sometimes one will get permission denied from Docker daemon socket. This might be caused by various reasons, but highly likely because of the low level of permission.[20] explained some potential solutions and sudo chmod 666 /var/run/docker.sock solved my issue.
- Node.js Version: Hyperledger Fabric and Hyperledger Caliper have a restricted requirement for Node.js version. They do not support the latest version of Node (currently 14.0.0), but only version 8.x.x or 10.x.x. Thus, check the Node.js version before you start running them.
- Cluster: As mentioned before, Caliper and Caliper++ used Docker Swarm to set up the cluster. Thus, I would recommend to learn and use swarm in the first place. Otherwise, Kubernetes is also a viable option.
- Bash Script: Hyperledger Fabric includes a heavy amount of scripts. Thus, learning how to read script files is also critical. [21] is the bash cheat sheet I used.
- Research and choose the right tools. I did not do enough study before deciding which tool to use, and thus wasted a good amount of time switching from plan to plan. Thus, make a good plan of action is the most important part of a project.
- Don't hesitate to ask. It would save me much time if I asked the author of the paper about the cluster and network earlier. Furthermore, the tech forum and community provided me a lot of assistance as well.

6. Summary

In general, this paper concentrates on the Hyperledger Fabric, Hyperledger Caliper, Caliper++, as well as Ethereum and mininet. Each part consists of the basic concepts of the system, and the related work. Summarizing the errors I made and giving out some advices, I hope my thesis provides enough information to those who wish to enter the field of Blockchain, specially Hyperledger Fabric. Meanwhile, realizing my inadequacies in decision making, I also write some general suggestions to help others to improve.

Foremost, I would like to express my sincere gratitude to my advisor, Prof. Lewis Tseng, for the support and guide, for his patience, motivation, and encouragement. Nevertheless, I would like to thank my fellow, Haochen Pan, and Yingjian Wu, for their consistent help and assistance.

References

[1] T. L. Foundation, *Hyperledger fabric*, 2020. [Online]. Available: https://www.hyperledger.org/projects/fabric.

[2] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing privateblockchains", 2017. DOI: http://dx.doi.org/10.1145/3035918.3064033.

[3] N. Q. Minh, D. Loghin, and T. T. A. Dinh, "Understanding the scalability of hyperledger fabric", 2019. [Online]. Available: https://bcdl.comp.nus.edu.sg/papers/understanding_the_scalability_of_hyperledger_fabric.pdf.

[4] Hyperledger, *Measuring blockchain performance with hyperledger caliper*, Mar. 2018. [Online]. Available: https://www.hyperledger.org/blog/2018/03/19/measuring-blockchain-performance-with-hyperledger-caliper.

[5] *Mininet, http://mininet.org/*. [Online]. Available: http://mininet.org/.

[6] *Hyperledger fabric: A blockchain platform for the enterprise*, Available at https://hyperledger-fabric.readthedocs.io/en/latest/index.html, Lastest Version: 2.1.

[7] *Caliper*, Mar. 2020. [Online]. Available: https://github.com/hyperledger/caliper.

[8] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing", 2011, NetDB, Vol. 11, pp. 1–7.

[9] M. Team. (). Mininet, an instant virtual network on your laptop, [Online]. Available: http://mininet.org/.

[10] *Building your first network*. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/build_network.html?highlight=byfn.

[11] D. Inc, *Docker documentation*, Available at https://docs.docker.com/.

[12] G. Szabo, *Virtualbox host-only network - ssh to remote machine*, 2018. [Online]. Available: https://code-maven.com/virtualbox-host-only-network-ssh-to-remote-machine.

[13] *Setup 2 ubuntu boxes in virtualbox to communicate with each other*, 2018. [Online]. Available: https://code-maven.com/setup-2-ubuntu-boxes-in-virtualbox-to-communicate-with-each-othere.

[14] B. Linkletter, *How to emulate a network using virtualbox*, Jan. 2017. [Online]. Available: https://www.brianlinkletter.com/how-to-use-virtualbox-to-emulate-a-network/.

[15] N. Afraz. (). Hyperledger caliper on multiple hosts, [Online]. Available: https://medium.com/@nima.afraz/hyperledger-caliper-on-multiple-hosts-6bcd07492e07.

[16] N. D. J. (). Adapting hyperledger caliper to custom hyperledger fabric networks, [Online]. Available: https://medium.com/tallyx/adapting-hyperledger-caliper-to-custom-hyperledger-fabric-networks-3ffa650215a0.

[17] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform",

[18] *Go-ethereum*, 2020. [Online]. Available: https://github.com/ethereum/go-ethereum.

[19] HaochengPan, *Nw3*, Mar. 2020. [Online]. Available: https://github.com/haochenpan/nw3.

[20] devdojo. (Aug. 2019). How to fix docker: Got permission denied, [Online]. Available: https://www.digitalocean.com/community/questions/how-to-fix-docker-got-permission-denied-while-trying-to-connect-to-the-docker-daemon-socket.

[21] cs.washington.edu, *Bash shell reference*, Available at https://courses.cs.washington.edu/courses/cse391/17sp/bash.html.