### **Root organization**

This is the organization that manages the root CA, i.e. the self-signed Certificate Authority. All other CAs, orderers, peers, and other nodes can participate in the network because they have certificates that somehow derive from the root CA (via a chain of trust). The root organization might be considered as a founder and administrator of the whole network.

**CA**

Hyperledger Fabric provides an optional [certificate authority service](http://hyperledger-fabric-ca.readthedocs.io/en/latest) that you may choose to use to generate the certificates and key material to configure and manage identity in your blockchain network. However, any CA that can generate ECDSA certificates may be used.

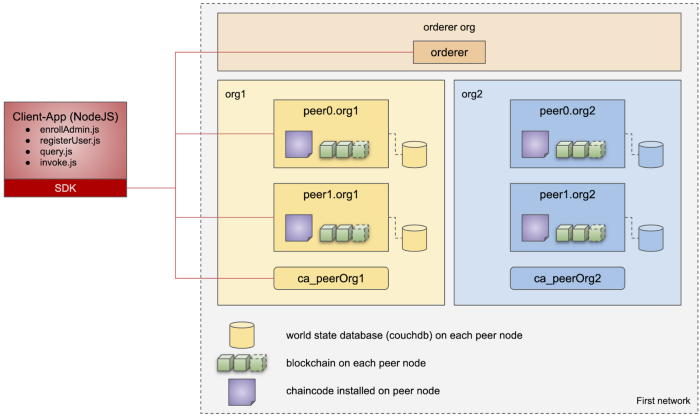
The Fabric CA issues and stores public keys only. Private keys are generated and stored by the client prior to invoking the enroll API (which essentially generates a CSR - certificate signing request - and submits it to the Fabric CA which then issues a signed X509 certificate.

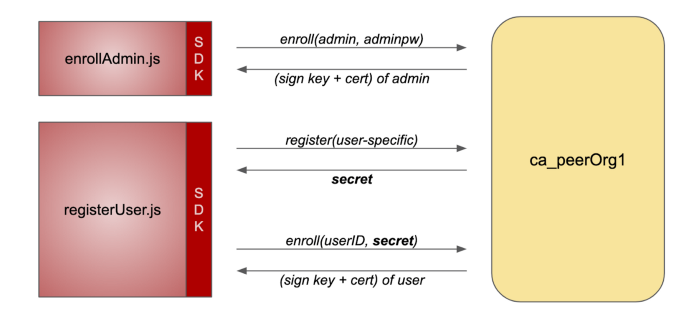
We are using Javascript codes provided in the Fabcar application. In particular we look into enrollAdmin.js and registerUser.js. as both are using SDK to interact with the Fabric-CA and fabric network.

Sqlite db

MSP

* cacerts - hold the public key of the root authority which signs and issues certificates for this MSP (i.e. the root CA which issued the cert in signcerts below)
* intermediatecerts - optionally holds intermediate certificates which sign and issue certificates for the MSP (i.e. the intermediate CA which issued the cert in signcerts below)
* signcerts - the public signing key/certificate for a specific identity
* keystore - the private signing key for a specific identity





**Peers**

[Peers](https://hyperledger-fabric.readthedocs.io/en/release-2.3/peers/peers.html) are the fundamental components of any Fabric network.

Peers store the blockchain ledger and validate transactions before they are committed to the ledger. Peers run the smart contracts that contain the business logic that is used to manage the assets on the blockchain ledger.

A peer may belong to one organization and multiple channels. It is responsible for keeping the **ledgers** and **chaincodes**.

A ledger consists of two parts: a log of transactions (**blockchain**) and a state database (**world state**). All peers in the channel have their own copy of the ledger.

If a peer belongs to multiple channels, it has multiple ledgers, one per channel.

Aside from keeping the state, a peer keeps and manages **chaincodes**. This is a broader concept than the smart contract itself. Smart contracts are packaged as a chaincode. Thus, the chaincode is a bundle that, among others, contains smart contracts.

**Peer types**

First, there are **anchor peers** - peers that are available outside the organization. Any cross-organization communication requires anchor peers. They play a significant role in the service discovery - they discover the network and they might be discovered from other organizations. In the transaction flow diagram above, anchor peers are **peer 1** and **peer 3**.

Then, at the simulation phase of the transaction, the smart contract is executed in **endorsing peers**. In the transaction flow diagram above, endorsing peers are **peer 2** and **peer 3**.

When the transaction is ordered, the ordering service delivers new blocks to **leading peers** and they distribute it further in the network. In the transaction flow diagram above, leading peers are **peer 2** and **peer 3**.

Finally, **committing peers** append all new transactions to their own copy of the ledger. All peers in the channel are committing peers.

**DB**

Fabric supports two types of peer state databases. LevelDB is the default state database embedded in the peer node.

LevelDB stores chaincode data as simple key-value pairs and only supports key, key range, and composite key queries.

**CouchDB** is efficient and flexible to design the JSON model to all sort of data collection. The indexing technique makes particular query even further accurate and faster, However, indexing is not that necessary to deploy in all data collection but becomes extremely handy against huge data set involving complex queries.

So, this is the general overview of CouchDB as a State Database in Hyperledger Fabric.

CouchDB operates in a docker container, so for installing CouchDB for your network, you need to download CouchDB docker images. In docker-compose.yaml developer needs to define the fabric-couchdb images to download and to create a docker container for CouchDB.

# **What is CouchDB Index?**

CouchDB is efficient in performing a rich query against JSON documents. There is an added advantage by creating indexing JSON files to make CouchDB query even more accurate. If the developer creates a CouchDB index when the collection query doesn’t need to iterate all row and records, that can instantly map with the particular record. Indexing is enormously helpful in case complex and large data set.

## Add the index to your chaincode folder

{

"index":{

"fields":["docType","owner"] **//** Names of the fields to be queried

},

"ddoc":"indexOwnerDoc", **//** (optional) Name of the design document **in** which the index will be created**.**

"name":"indexOwner",

"type":"json"

}

**Orderer**

An ordering service allows peers to focus on validating transactions and committing them to the ledger.

Order the transactions

Cut the block and distribute the block amongst the orgs when the criteria is met ( min txn/size or time).

orderer whose is responsible for receiving endorsed transactions from peers and put them into blocks then these blocks are distributed to all peers. These blocks are Final.

**Raft Consensus for production network**

The problem is that both mechanisms (Kafka and Raft) are only crash fault tolerant. It means that the ordering service should be centralized (at least at the channel level) and governed by a trusted organization. Taking over the orderer nodes and serving malicious content to different peers can lead to corrupted blockchain.

### **CFT and BFT**

CFT (or *crash fault tolerance*) means that a distributed system can operate normally even when a number of nodes **fail**. BFT (or *Byzantine fault tolerance*) means that a distributed system can operate normally even when a number of nodes **lie**.

BFT is crucial when many organizations cooperate. It is in the interest of all organizations to be resistant to malicious actions of the other ones. With proper endorsement policies and the ordering service managed by a trusted organization, the Hyperledger Fabric network might be considered BFT at the organization level. Even when an organization lies, the state won't corrupt.

The sample network uses a single node Raft ordering service that is operated by the orderer organization. You can see the ordering node running on your machine as orderer.example.com. While the test network only uses a single node ordering service, a production network would have multiple ordering nodes, operated by one or multiple orderer organizations. The different ordering nodes would use the Raft consensus algorithm to come to agreement on the order of transactions across the network.

Their design allows different organizations to contribute nodes to a distributed ordering service.

Raft protocol uses a “leader and follower” model.

General orderer count is five.

At least 1 orderer in your HLF network to work.

Not every peer needs to be connected to an orderer - peers can cascade blocks to other peers using the [**gossip**](https://hyperledger-fabric.readthedocs.io/en/release-2.2/gossip.html) protocol.

Query is not a transaction which writes into the ledger. So it doesn't need the orderer. For query, it will pick up the data from the peer's local database.

### **Orderer organization**

The orderer organization might be separated from the root organization. This is the organization that manages the orderer nodes. It is responsible for keeping the data consistent and generating blocks of ordered transactions that form a blockchain.

In some cases the orderer nodes might be distributed among other organizations. It might be required in your deployment, however it is also risky in terms of the performance and maintaining trust in the network. The orderer organization, as well as the root organization, should be governed by an authority with the highest trust in the network.

**Orderer Count:**

While it's possible to use the console to build a configuration of any number of ordering nodes (no configuration is explicitly restricted), some numbers provide a better balance between cost and performance than others. The reason for this lies in satisfying the needs of high availability (HA) and in understanding the Raft concept of the "quorum", the minimum number of nodes that must be available (out of the total number) for the ordering service to process transactions.

In Raft, a majority of the total number of nodes is needed to form a quorum. In other words, if you have one node, you need that node available to have a quorum, because the majority of one is one. Similarly, if you have two nodes, you will need both available, since the majority of two is two (for this reason, a configuration of two nodes is discouraged; there is no advantage to a two node configuration). In a similar vein, the majority of three is two, the majority of four is three, the majority of five is three, and so on.

While satisfying the quorum will make sure the ordering service is functioning, production networks also have to think about deployment configurations that are highly available (in other words, configurations in which the loss of a certain number of nodes can be tolerated by the system). Typically, this means tolerating two nodes failing: one node going down during a normal maintenance cycle, and another going down for any other reason (such as a power outage or error).

This is why five nodes is a good number for a production network. Recall that the majority of five is three. This means that in a five node configuration, the loss of two nodes can be tolerated. If your configuration features four nodes, only one node can be down for any reason before another node going down means a quorum has been lost and the ordering service will stop processing transactions.

**Ex:**

Three orderer peer

I have solved this in such a way that in my case I have three orderers that run independently on different environments. If I crash all these orderers, the peer containers will continue to run on the other participants of the network. As you said, they cannot make any transactions. If one of my orderers crashes it is not so bad after the raft consensus, the containers keep running. If another one fails, no transactions can be made. In this case I let the peers continue and check if the orderers are available again.

**PEER**

We can see that it's the peer that hosts both the ledger and chaincode

Peers have an identity assigned to them via a digital certificate from a particular certificate authority. You can read a lot more about how X.509 digital certificates work elsewhere in this guide, but for now, think of a digital certificate as like an ID card that provides verifiable information about a peer

*policy in the channel configuration uses the peer's identity to determine its rights.* The mapping of identity to organization is provided by a component called a *Membership Service Provider* (MSP) --- which determines how a peer gets assigned to a specific role in a particular organization and accordingly, gains authorized access to resources. Moreover, a peer can be owned only by a single organization, and is therefore associated with a single MSP. Think of an MSP as linking an individual identity with a particular organizational role in a Fabric network.

**Channel**

create a Fabric channel for transactions between Org1 and Org2. Channels are a private layer of communication between specific network members.

Each channel has a separate blockchain ledger.

After you have created a channel, you can start using [smart contracts](https://hyperledger-fabric.readthedocs.io/en/release-2.3/smartcontract/smartcontract.html) to interact with the channel ledger. Smart contracts contain the business logic that governs assets on the blockchain ledger.

**Endorsement Policy:**

* Endorsement policy - each transaction has a target to one or more chaincodes (namespaces in the DB). Each such namespace has an chaincode has its own endorsement policy. If the transaction is signed by peers such that the endorsement policy is satisifed, then the transaction is deemed as one that passed the endorsement policy.
* MVCC(Multi version concurrency control) - A transaction that read keys during its execution, records their versions, and then upon commit the peer checks that the versions didn't change, so the causality of the data still remains intact.

**Interaction Network:**

The peer CLI allows you to invoke deployed smart contracts, update channels, or install and deploy new smart contracts from the CLI.

**Endorsing peer**

All endorsing peer must have instantiated chaincode so that they can simulate transactions and create Read/Write(RW) set. Installing and instantiating chaincode are different transaction than regular invoke.

Invoking some transaction meansa it can be one of

1. Query
2. Adding Asset, Updating Asset, Deleting Asset

HF maintains versioning of each unique key and it increases chronologically when we update same asset. This versioning (MVCC - Multiversion concurrency control) avoid double-spending problem as well.

HF dont care about whatever value we puts for key(Value could be anything).

Lets assume we have to add car asset with key car1 and value is {"name":"Audi", "owner":"ABC"}. When we send the transaction to all endorsing peer, they create RW set as below

* Read Set: NA
* Write set: Key-car1, version-1, value-{"name":"Audi", "owner":"ABC"}

When the tx gets committed to blockchain, one of the block will have this transaction and the current state database (Couch db or level db) will heve this latest value for that key : Key-car1, version-1, value-{ "name": "Audi", "owner": "ABC"}

Let's assume if we updating this same asset(car1)

car1 - { "name": "Audi", "owner": "PQR"}

Here we are changing owner from ABC to PQR

This time Endorsing peer create the following RW set

* Read Set: Key-car1, version-1
* Write set: Key-car1, version-2, value-{"name":"Audi", "owner":"PQR"}

Once the tx gets committed to the blockchain. again this transaction gets added into one block and most importantly, the current state database will get updated with this latest value and old value will be updated and the version also get changed to 2.

### **Smart contracts**

Smart contracts in Hyperledger Fabric are small programs written typically in Go, JavaScript/TypeScript or Java that contain the business logic to be executed as transactions on the blockchain.

### **Endorsement policy**

Endorsement policy defines the rules that organizations in the network must execute and agree on the smart contract execution result to consider the transaction as valid. For instance, it might be enough to execute the transaction by one organization, all organizations or with the majority of organizations, or even by another sub-group of organizations defined by the endorsement policy.

### **Private data collection**

Hyperledger Fabric has a mechanism to store the data that is kept off-chain and might be available only to a subset of organizations in the channel. Only the hash of the data is saved in the blockchain.

This mechanism is called private data. Each chaincode might contain many private data collections, available only to organizations specified during the chaincode installation process.

### Private data

The private data mechanism keeps your data private and still guarantees its integrity by the blockchain. It is possible because when you save private data in a smart contract, a SHA-256 control sum of the data is saved on the blockchain.

You can configure the private data collection in a chaincode in a way that only selected organizations will have read or write access to the data. Regardless of permissions, all organizations may verify the data by checking if the hash saved on blockchain is the same as the hash of the data they want to check.

Additionally, each chaincode has an implicit private data collection. It can be used to store an individual organization’s private data, and does not need to be defined explicitly (see the [documentation](https://hyperledger-fabric.readthedocs.io/en/latest/private-data-arch.html#implicit-private-data-collections)).

Note that the private data mechanism in Hyperledger Fabric is not GDPR-compliant, since there is no way to remove the data from temporary databases on peers (see [FAB-12038](https://jira.hyperledger.org/browse/FAB-12038) and related issues). Besides, private data should not consist of simple values. The SHA-256 control sum is deterministic and might be vulnerable to brute force attack.

## **CHAINCODE INSTALLATION**

You can think of chaincodes and smart contracts as a set of automated business rules that concern multiple organizations in the network. We need a formal process of deploying them. The network participants need to agree on a way they cooperate.

It is not required to install the chaincode on all peers in the channel. You can choose only the subset of peers.

**Transaction Flow**

### Phase 1 - Transaction Proposal and Endorsement

Phase 1 of a ledger update (write) consists of transaction proposal submission, execution and endorsement:

a) Transaction proposal --- The client application (A1) submits a signed transaction proposal by connecting to the gateway service on P1. A1 must either delegate the selection of endorsing organizations to the gateway service or explicitly identify the organizations required for endorsement.

b) Transaction execution --- The gateway service selects P1, or another peer in its organization, to execute the transaction. The selected peer executes the chaincode (S1) specified in the proposal, generates a proposal response (containing the read-write set). The selected peer signs the proposal response and returns it to the gateway.

c) Transaction endorsement --- The gateway repeats transaction execution (b) for each organization required by the chaincode (smart contract) endorsement policies. The gateway service collects the signed proposal responses and creates a transaction envelope --- which it returns to the client (SDK) for signing.

### Phase 2 - Transaction Submission and Ordering

Phase 2 of a ledger update consists of transaction submission and ordering into blocks:

a) Transaction submission --- The client (SDK) sends the signed transaction envelope to the gateway service. The gateway forwards the envelope to an ordering node and returns a success message to the client.

b) Transaction ordering --- The ordering node (O1) verifies the signature, and the ordering service orders the transaction, and packages it with other ordered transactions into blocks. The ordering service then distributes the block to all peers in the channel for validation and commitment to the ledger.

### Phase 3 - Transaction Validation and Commitment

Phase 3 of a ledger update consists of transaction validation, ledger commitment and a commit event:

a) Transaction validation --- Each peer checks that the client signature on the transaction envelope matches the signature on the original transaction proposal. Each peer also checks that all read-write sets and status responses are equivalent (i.e. the endorsements from all peers match) and that the endorsements satisfy the endorsement policies. Each peer then marks each transaction as valid or invalid for commitment to the ledger.

b) Transaction commitment --- Each peer commits the ordered block of transactions to the channel ledger (L1). The commit is an immutable ledger update (write) to the channel ledger. The world state (essentially, the sum of all valid transactions) of the channel is updated with results of valid transactions only.

c) Commit event --- Each peer that commits to the ledger sends the client a commit status event with proof of the ledger update.

### **World state**

This is a key value store that contains data created by transactions from the blockchain. Unlike the blockchain, the world state is mutable. It represents the latest values for all keys updated in the transactions.

1. At first, the chaincode needs to be **built** by a tool specific for the language used and **packaged** by the Hyperledger Fabric tools to prepare a package to be disseminated among the peers.
2. All peers from organizations in the channel that want to have the chaincode installed, **install** the chaincode package.
3. Organizations need to **approve** the chaincode definition. By default, chaincode needs to be approved by single peers from the majority of organizations.
4. A peer **commits** the chaincode definition. Now the chaincode (or new version of the chaincode) can execute transactions.
5. Some chaincodes may require manual initialization, i.e. calling the Init method before any other smart contracts. In this case, if you want your chaincode operational, you should first invoke the Init transaction on the chaincode.

The same process leads to both chaincode installation and upgrade. If there is an upgrade, peer manages to terminate the container with the old version of the chaincode.

The state in Hyperledger Fabric consists of two elements:

1. The blockchain, i.e. an immutable log of transactions.
2. The world state, i.e. a database with business objects created and updated by those transactions.

The world state is derived from the blockchain and has the form of a key-value store. Chaincodes/smart contracts (i.e. applications executed on the blockchain) do interact with the world state, not the blockchain itself.

**Client SDKs** are used to interact with the network, for example obtain certificates from the CA or call smart contracts. **Contract APIs** are frameworks for writing smart contracts.

**FQA**

* If you upgrade a chaincode, you will continue accessing to the data of the previous chaincode. It is done automatically by Hyperledger Fabric.
* That's why it is always recommended to have more and more of endorsing peers within the network and organizations. The bigger the network - harder it would be beat with malicious intent.
* That's the beauty of a decentralized distributed system. Even if you or someone else changes the state of your database/ledger it will not match with the state of others in the network, neither will it match the transaction block hash rendering any transactions invalid by the endorsers unless you can restore the actual agreed upon state of the ledger from the network participants or the orderer. To take advantage of the immutability of ledger you must query the ledger. Querying the database does not utilize the power of blockchain and hence must be protected in fashion similar to how access to any other database is protected.

### Integration with Hashicorp Vault / Amazon SecretsManager

Sometimes sensitive data such as private keys shouldn’t be kept on disk. Polygon Edge offers the ability to store and manage private keys on Hashicorp Vault or Amazon SecretsManager.

### What peers should endorse the transaction?

It is determined by the endorsement policy that is assigned to the chaincode. If the endorsement policy is not fulfilled, i.e. not enough organizations receive the transaction proposal, the transaction fails.

Hyperledger Fabric client SDKs have various options to determine the peers to send the transaction proposals.

1. Peer addresses can be provided manually or in a configuration file.
2. If there is a discovery service configured, it will be responsible for determining the endorsing peers.
3. Target organizations can be provided manually. Then the proposals are sent to endorsing peers from the organizations.
4. If none of the previous conditions is met, the proposal is sent to endorsing peers from the channel.

Still, regardless of the peer selection process, the only requirement is to satisfy the endorsement policy.

### What peer should be queried?

Hyperledger Fabric client SDKs have several built-in strategies for querying peers. By default, you can query a single peer in your organization and change the peer if the query fails (MSPID\_SCOPE\_SINGLE) or you can make each call to a different peer (MSPID\_SCOPE\_ROUND\_ROBIN).

It may happen that your organization has no peers and you want to call other organizations' peers. In this case, there are similar predefined strategies that start with PREFER\_MSPID\_ instead of MSPID\_. Alternatively, if you want something custom, you can implement your own strategy.

### Which user should call a smart contract?

There are basically two main approaches. First, you can use the Hyperledger Fabric network in a database-like manner. In this case, there is a dedicated user per organization to call smart contracts. Each organization should create this user in CA and use its credentials in enrollment (step 1 in the transaction flow).

On the other hand, you may want application users to call smart contracts. In this case, each application user should be registered in CA and use its own credentials in the enrollment. This approach is more complex, since you have to create and maintain user accounts in the CA. It allows you, however, to track who actually called a smart contract, which might be required in many business cases.

### When can a transaction fail?

A transaction can fail on many levels. At the endorsement phase (i.e. step 3 and 4), validation is performed by the peers and client SDK. There might be an error in smart contract execution, the endorsement policy might not be fulfilled or the responses from peers might differ. In this case, the transaction is cancelled and not appended to the chain.

Once the SDK assembles and sends the transaction to the ordering service, it will be a part of the block. It might, however, be valid or invalid. In this case, the transaction may fail, for example, in case of not fulfilling endorsement policy or in case of key collisions. A new block containing the transaction might be also rejected by the peer (see step 7 on the transaction flow).

### How many transactions are in the block?

It depends. There are several configuration options regarding the block size.

There is a BatchSize group of parameters: MaxMessageCount, AbsoluteMaxBytes and PreferredMaxBytes). The first one specifies the maximum number of transactions in the block, the latter ones set maximum and preferred block size.

Then, there is a BatchTimeout config parameter to specify how long the orderer should wait to form a block. Even if there are fewer transactions than specified by MaxMessageCount or other parameters, the block will be created after the specified amount of time. This value should be typically set to 1-5 seconds to keep the responsiveness of the blockchain in case of low load.

Note that in some cases it is useful to set MaxMessageCount to 1. This is a quick and easy way to handle the key collisions problem in the networks that do not need high performance.

**Block size scaling**

The configuration of the Fabric can be done in many areas of the network. Enterprises

can decide for themselves the composition of peers by deciding how many endorsers,

committers and orderers that suits their use case. Furthermore, there is one parameter,

MaxMessageCount, that can be optimized to maximize throughput. This parameter tells

the orderer how many transaction that can be included inside a batch which is then sent to

committers to form the next block.

**Endorser scaling** Here we mainly focus on endorser scaling on the same channel. we concluded that adding endorsers on different channels would not have any impact on the performance of a channel. Therefore, we limit the test to a single channel.

The test is configured with a single client that uses 1 thread to send 1000 transactions to varied amount of endorsers and record the time taken for all transaction to be included in a block The most significant drop in throughput; from one endorser to two endorsers; is the most interesting part.

**Finality**

In Hyperledger Fabric, on the other hand, once a transaction is committed to the chain, it becomes final (absolute finality). There are no forks because the service responsible for the consensus itself - the ordering service - is centralized and deterministic. Blockchain cannot be changed.

**Best practice**

If you can, increase the pool of available endorsing peers and load balance across that set to achieve greater TPS, improve latency or both. Note that transaction latency is also a function of the time your chaincode takes to execute for an endorsement, so as noted above, YMMV.

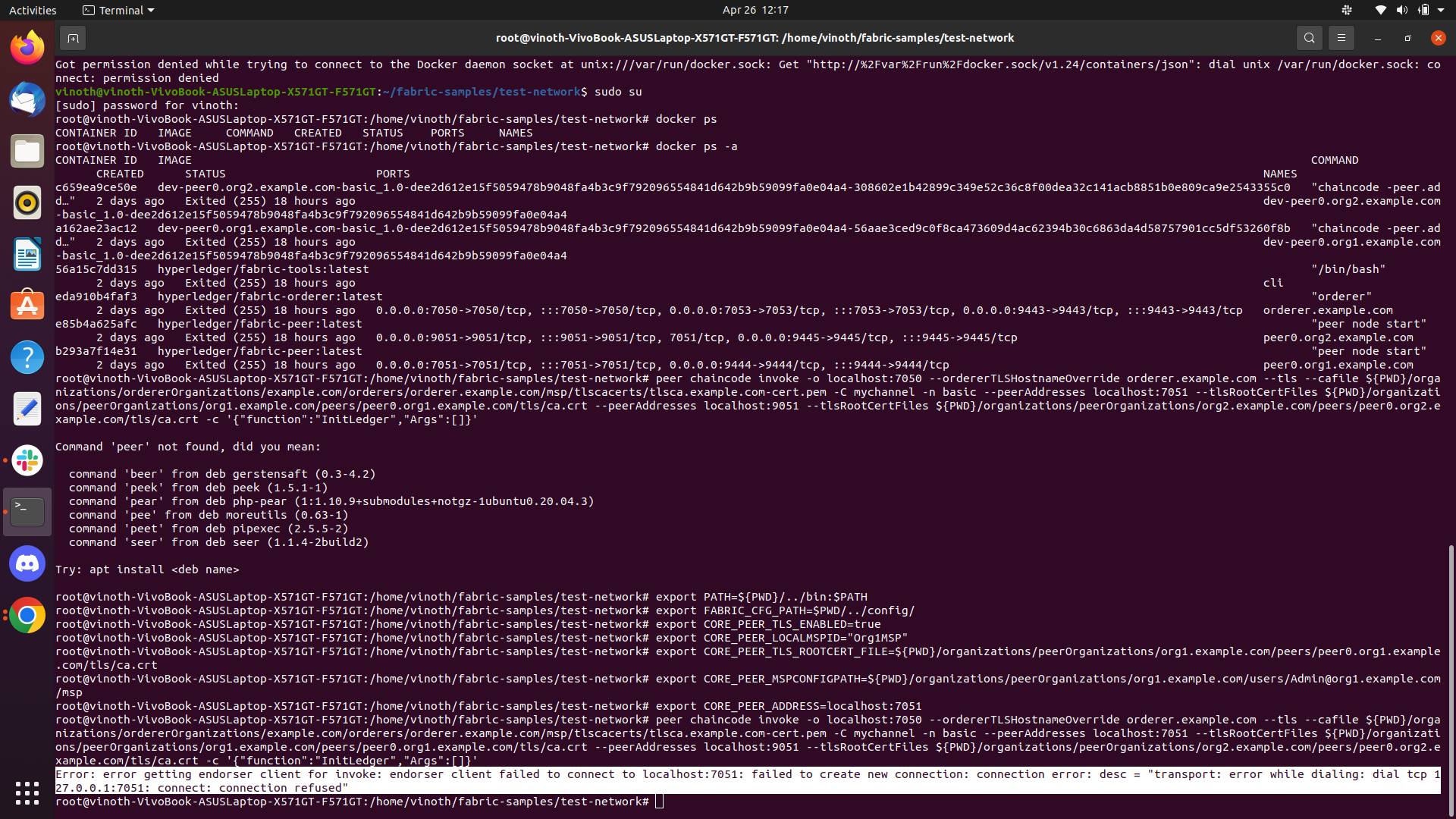
**Dealing with latency**

Note that I haven’t mentioned latency until now. The reality is that as you scale the rate of transactions sent on a channel, you will likely reach a point where [transaction latency](https://www.hyperledger.org/resources/publications/blockchain-performance-metrics#transaction-latency) becomes untenable because the peers become saturated consuming all the available CPU and/or disk i/o allocated to the container. For purposes of this set of experiments, I endeavored to keep the average latency under one second. We can actually scale beyond the TPS cited above if we are willing to tolerate additional latency, though with diminishing returns because while you may distribute the processing of endorsement across a set of available endorsers, every peer that participates in a channel acts as a committer and must validate the transactions received from the ordering service.

**Scaling Hyperledger Fabric**

Another myth (or maybe it is FUD?) that I have heard recently is that Hyperledger Fabric limits the number of peer nodes in a network. I’m not sure of the source of this myth, but nothing could be further from the truth! As a function of our system testing for a release, we kick off a scale test to ensure that we have not introduced any changes that might affect performance and scale. Our testing creates a network comprised of 32 organizations each with four peers for resilience (a total of 128 peers) and establishes a channel between each pair of organizations for a total of 325 channels — six of the organizations are configured as auditor or regulator nodes that observe each channel but do not perform endorsement.

**Error:**

****

**Invoking cmd in test network:**

**peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride orderer.example.com --tls --cafile ${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses localhost:7051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles ${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt -c '{"function":"InitLedger","Args":[]}'**

**Error: error getting endorser client for invoke: endorser client failed to connect to localhost:7051: failed to create new connection: connection error: desc = "transport: error while dialing: dial tcp 127.0.0.1:7051: connect: connection refused"**

**Sol**

**Kill that port then restart network**

**Fuser -k 7050/tcp**

**./network down**

**./network up**

**Reference:**

[**https://github.com/hyperledger/fabric/blob/main/docs/source/peers/peers.md**](https://github.com/hyperledger/fabric/blob/main/docs/source/peers/peers.md)

[**https://softwaremill.com/hyperledger-fabric-cheat-sheet/**](https://softwaremill.com/hyperledger-fabric-cheat-sheet/)

[**https://www.diva-portal.org/smash/get/diva2:1111497/FULLTEXT01.pdf.10**](https://www.diva-portal.org/smash/get/diva2:1111497/FULLTEXT01.pdf.10)

[**https://deeptiman.medium.com/couchdb-as-a-state-database-in-hyperledger-fabric-adb5d820c82e**](https://deeptiman.medium.com/couchdb-as-a-state-database-in-hyperledger-fabric-adb5d820c82e)

[**https://kctheservant.medium.com/exploring-fabric-ca-registration-and-enrollment-1b9f4a1b3ace**](https://kctheservant.medium.com/exploring-fabric-ca-registration-and-enrollment-1b9f4a1b3ace)