Definition of Blockchain.

A blockchain is an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way without the need for a central authority.

Key Characteristics to be remembered:

- **Open**: Anyone can access blockchain.
- **Distributed or Decentralised:** Not under the control of any single authority.
- **Efficient:** Fast and Scalable.
- **Verifiable:** Everyone can check the validity of information because each node maintains a copy of the transactions.
- **Permanent:** Once a transaction is done, it is persistent and can't be altered.

Contents of a Block.

Blockchain starts with a block called **genesis block.** Each block stores the following information in it:

- **Index:** Position of the block in blockchain. Index of genesis block is 0.
- **Time stamp:** The time when that particular block was created.
- **Hash:** Numeric value that uniquely identifies data just like our fingerprints.
- **Previous hash:** Hash value of the previous block. For genesis block, this value is 0.
- **Data:** Data stored on the node. For example, transactions.
- **Nonce:** It is a number used to find a valid hash. To generate this number, the processing power is used.

Genesis Block	
Previous Hash	0
Timestamp	Thu, 27 Jul 2017 02:30:00 GMT
Data	Welcome to Blockchain CLI!
Hash	0000018035a828da0…
Nonce	56551

Mechanism of Blockchain.

- Blockchain works like a public ledger.
- Any small change in the data value can affect the hash value. Hence, affecting the whole block chain.
- Every peer in a Blockchain network maintains a local copy of the Blockchain.
- All the replicas need to be updated with the last mined block.
- All the replicas need to be consistent — the copies of the Blockchain at different peers need to be exactly similar.

Cryptographic Hash Function.

Map any sized data(x) to a fixed size(H(x)).
e.g. H(x) = x % n,
where x, n = integers
% = modular (remainder after division by n) operations.
x can be of any arbitrary length, but H(x) is within the range [0,n-1].

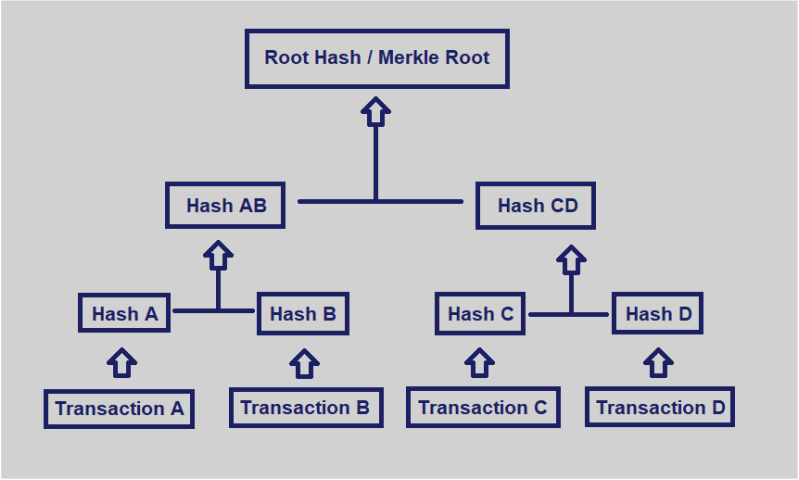
Important Points about Hash Functions:

- You can calculate H(x) from x but the reverse is not possible.
- For even a small change in the value of x, value of H(x) changes. This is called **Avalanche Effect**.

Merkle Tree/Hash Tree.

- Every leaf node is labelled with the hash of a data block.
- Every non-leaf node is labelled with the cryptographic hash of the labels of its child nodes.

- You can traverse this tree just like you traverse a binary tree.

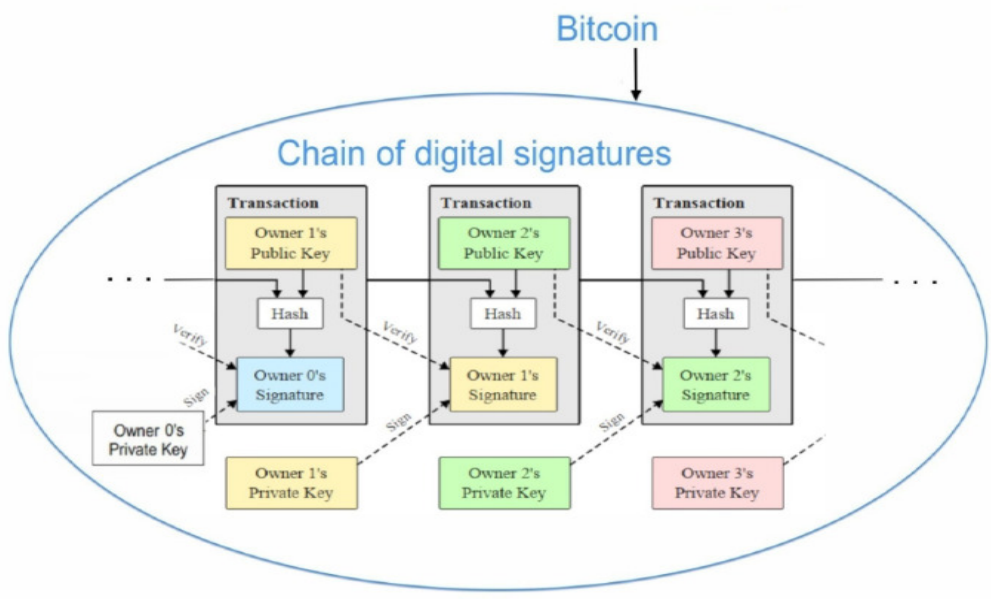


Uses of Merkle Tree:

- Peer to Peer Networks.
- Bitcoin implementation.

Bitcoin.

- Bitcoin is a completely decentralised, peer-to-peer, permissionless cryptocurrency put forth in 2009 by Satoshi Nakamoto.
- Bitcoin is the first blockchain application.
- It is permissionless , i.e. open to anyone.
- Bitcoin blockchain size is growing exponentially.



Smart Contract.

- The term was coined by **Nick Szabo**, a computer scientist and cryptographer, in 1996.
- **Smart contracts** are self-executing **contracts** with the terms of the agreement between buyer and seller being directly written into lines of code.
- It is an automated computerised protocol used for digitally facilitating, verifying or enforcing the negotiation or performance of a legal contract by avoiding intermediates and directly validating the contract over a decentralised platform — faster, cheaper and more secure.
- Crowd funding is an important use of smart contracts.

Advantages of Smart Contracts:

- **Immutable:** No one can change it once it is deployed on blockchain.
- **Distributed:** All the steps of the contract can be validated by every participating party — no one can claim later that the contract was not validated.
- **Saves money:** Since there is no need of third-party to run or maintain the contract, the cost is saved.

Popular Smart Contract Platforms:

- Ethereum
- Hyperledger
- Ripple
- Rootstock

Structure of a Block(Reference: Bitcoin).

A block has two main components:

1. Block Header
2. List of Transactions

1. Block Header:

Contains metadata about a block which includes the following:

1. Version: Block version number
2. Previous Block hash: This is used to compute new block hash. Hence, making the blockchain temper proof.

3. Merkle Tree Root: The root of the Merkle tree is a verification of all the transactions.
4. Timestamp: The time at which block is mined.
5. Bits/Difficulty: Difficulty in Bitcoin is expressed by the hash of a Bitcoin block header being required to be numerically lower than a certain target.
6. Nonce:A 32-bit random number used in blockchain, while calculating the cryptographic hash for a Block.

The last three are collectively called mining statistics.

Hashes	
Hash	00000000000000000000fa111355203562c653e950479419ef437228aecca488
Previous Block	00000000000000000002eea942f19badf8ddc34407331bd61ae3341ffef39ba4d
Next Block(s)	00000000000000000002ab596a54d86581dfecb8df76522659aa48229df6e6e0
Merkle Root	4fccf03c5adbac8bec272fed00d78568d202e7472d92265593e1f1c339b223fc

A block is identified by its hash which is computed using **Double SHA256 algorithm.**

2. List of Transactions

- Transactions are organised as a Merkle Tree. The Merkle Root is used to construct the block hash.
- If you change a transaction, you need to change all the subsequent block hash.
- The difficulty of the mining algorithm determines the toughness of tampering with a block in a blockchain.

Mining:

- It is a mechanism to generate hash of the block.
- Mining involves creating a hash of a block of transactions that cannot be easily forged, protecting the integrity of the entire blockchain without the need for a central system.

Bitcoin Mining:
H(k) = Hash(H(k-1) || T || Nonce)
Here,
H(k-1) = Previous block hash

T = List of transactions
Nonce = Miners find this nonce as per the complexity(number of zeros at the prefix)

Distributed Consensus.

Consensus is the process by which peers agree to the addition of next block in the blockchain.

Distributed Consensus ensures that different nodes in the network see the same data at nearly the same point of time. Hence in case of any failure, the system can still provide a service as the data is decentralised.

To maintain anonymity in this large network, the **permission less protocol** is used where you don't need to record your identity while participating in the consensus.

Challenge Response System

The network poses a challenge and each node tries to solve the challenge. The node which solves the challenge first, gets to dictate what set of data or state elements to be added in the network.

This process continues iteratively and ensures that different nodes win the challenge at different runs. Hence, **no single node will be able to control the network.**

Bitcoin Proof-of-work (PoW) algorithm ensures consensus over a permission less setting based on challenge response.

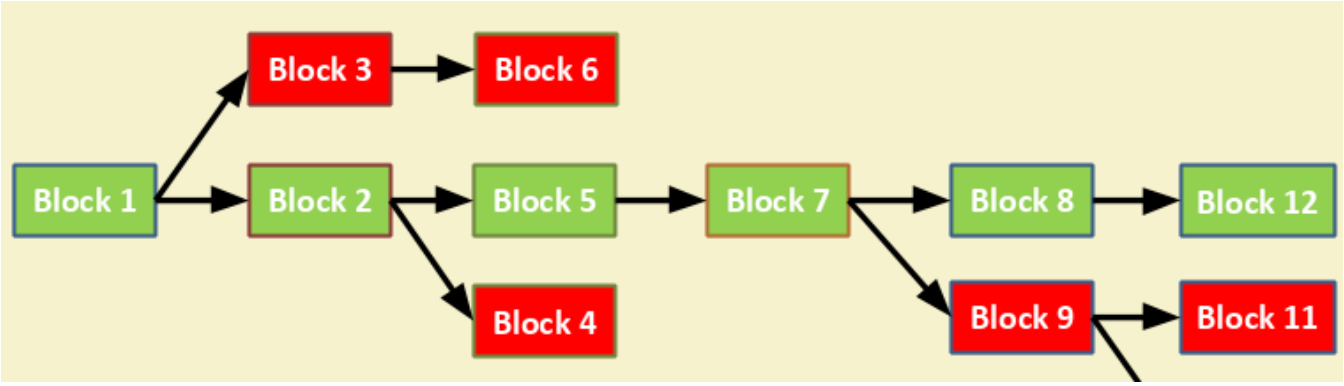
Every node spends a large amount of computational power to solve the mathematical challenge in each iteration of consensus.

The computational effort expended by the nodes in achieving consensus would be **paid for by cryptocurrency** generated and managed by the network.

Blockchain as tree.

Suppose two miners found the nonce at the same time, then they will add the mined blocks (2,3) to the previous block (1).

Then, the next miners mined another three blocks (4,5,6) and added them to the previous blocks they found (2,3). This process of mining continues and the blockchain looks like a tree as following:



The longest chain is the accepted chain which is in green colour in the above figure. Other blocks (red colour) which are not the part of blockchain are called orphaned blocks.

Permissioned Model (Private Blockchain).

- Users/Participants are known already. Security and Privacy are the main factors in this blockchain.
- Only the entities participating in a particular transaction will have knowledge and access to it — other entities will have no access to it.
- The Linux Foundation's Hyperledger Fabric is an example of a permissioned blockchain framework implementation.

Applications .

- Asset Movement and Tracking
- Provenance Tracking: Tracking the origin and movement of high-value items across a supply chain.

Cryptographic Hash Function.

Input: Any string of any length, i.e. the message.

Output: Fixed Size output,i.e. message digest(256 bits for blockchain).

Properties of Hash Function:

1. **Collision free:** For two different messages(inputs), digests(outputs) are always different.
2. **Hiding:** The original message is hidden behind the message digest.
3. **Puzzle Friendly:** The value of k has to be found in Y=H(X||k) using mining puzzle in Bitcoin PoW where X and Y are known.

1. Collision Free

- **One-Way:** Message digest can be computed from message using some algorithm but the vice versa is not possible.
- It is **difficult to find** x and y such that $x \neq y$ but $H(x)=H(y)$.(Not impossible)
- Collision depends on the complexity of hash function. In case of cryptographic hash function, it is difficult to have collision.

Birthday Paradox Problem

Question Statement: What is the probability that in a set of n random people, some of them will have the same birthdays?

Solution: By Pigeon hole Principle, the probability will be 1 if the number of people reaches 366(not leap) or 367(leap) year. This probability decreases with the decrease in number of people. For 23 people, probability will be 0.5 only.

How much time will it take to hack cryptographic hash function?

Suppose a hash function produces N bits of output, then an attacker has to compute 2^(N/2) hashes for a random input to find two matching outputs with probability greater than 0.98.

Thus, for 256-bit hash function, hacker needs to compute 2^{128} hash operations which is time consuming. If every hash takes 1 millisecond, then it would take ~10^{28} years.

Summary:

- If $H(x) = H(y)$, then it is safe to assume $x = y$.
- Need to remember only message digest(only 256-bits) rather than the whole original message.
- Comparison of two large messages become easy with their message digest because you will have to compare only 256-bits of hash.

2. Hiding

- If a message(x) is given, it is easier to find message digest H(x) but it is **computationally difficult** to find message(x) from message digest H(x).
- Hence, the original message is hidden behind the message digest.

3. Puzzle Friendly

In a search puzzle M and Z are known and k is to be computed such that the hash value of M append k is equal to Z. This can be seen by the below formula:

$Z = H(M||k)$

Puzzle friendly property implies that random searching is the best strategy to solve the above puzzle.

Hash Pointer.

Hash pointers are just hashes that are used to reference another piece of known information. With hash pointer, we can retrieve the information and check that the information has not been modified.

For example, Bitcoin uses hashes to reference funds supplied in previous transactions. So, if you want to make a transaction giving some bitcoins to someone, you have to reference (with a hash pointer) the transaction where you were given some funds.

Hash Pointer ensures prevention of data tampering because previous block hash is used to compute the new block hash

Digital Signature.

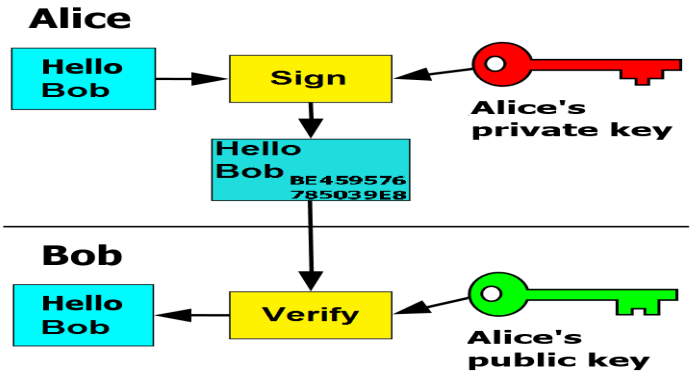
A digital signature is a digital code that is transmitted along with the document to the receiver.

The sender uses a **signing algorithm** to sign the message. The message and the signature are sent to the receiver. The receiver applies **verifying algorithm** to the combination. If the result is true, the message is accepted otherwise rejected.

Types of keys:

1. **Public(Asymmetric key):** This key is available to everyone and is used in verifying the message by receiver.

2. **Private(Symmetric key):** This key is only known to the sender and is used in signing the document by sender.



Properties of Digital Signature:

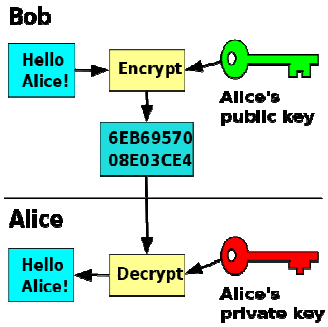
- **Message Authentication:** Ensures the identity of the sender. A message sent by Alice can only be verified using her public key. The message will not be verifiable using Eve's public key.
- **Message Integrity:** Ensures that message has not been tampered with or altered. In case an attacker has access to the data and modifies it, the digital signature verification at receiver end fails. The hash of modified data and the output provided by the verification algorithm will not match. Hence, receiver can safely deny the message assuming that data integrity has been breached.
- **Non-Repudiation:** Sender can't deny about the sending of the data. If Alice has send some message, she can't deny later because the message can be verified using her public key.

Public Key Cryptography.

In a public key encryption system, any person can encrypt a message using the receiver's public key. That encrypted message can only be decrypted with the receiver's private key.

Terminology of texts:

1. **Plain text:** The original message which is not been encrypted.
2. **Cipher text:** The encrypted text which is obtained by encrypting plain text using some algorithm.



Properties of Public key cryptography:

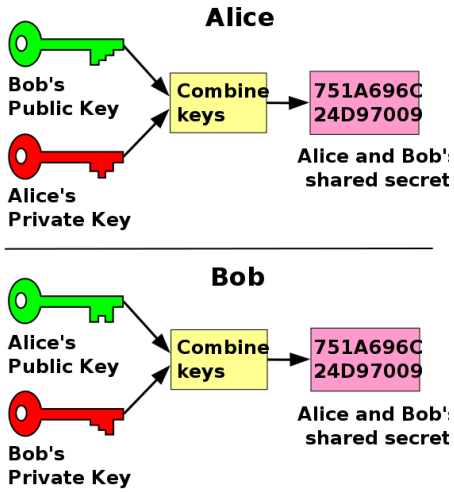
- It is generated randomly that a attacker is unable to guess it in any way.

- Length of the key must be enough for it not to be guessed. Long key will be difficult to guess.
- The key should contain sufficient entropy such that all the bits in the key are equally random.

Digital Signature in Blockchain.

This is used to validate origin of transaction and prevent non-repudiation. The sender can't deny the transaction or anyone else can't claim the transaction by this.

Bitcoin uses **Elliptic Curve Digital Signature Algorithm(ECDSA)** based on elliptic curve cryptography.



Bitcoin Introduction.

Bitcoin is a **decentralized digital currency** that enables instant payments to anyone, anywhere in the world.

Bitcoin uses **peer-to-peer technology** to operate with **no central authority.**

Main Operations:

- **Transaction Management:** Transfer bitcoins from a user to another across the world.
- **Money Issuance:** No central authority regulates monetary base.

Creation of Coins:

- **Controlled Supply:** In this currency is created by the nodes of a peer-to-peer network. The Bitcoin generation algorithm defines, in advance, how currency will be created and at what rate. Any currency that is generated by a malicious user that does not follow the rules will be rejected by the network and thus is worthless.
- New bitcoins are generated during **mining** when a user discovers a new block.
- Roughly **every 4 years**, the number of bitcoins that can be "mined" in a block **reduces by 50%**. Originally the block reward was 50 bitcoins; it halved in November 2012; it then halved again in July 2016.

- In the end, no more than **21 million bitcoins** will ever exist.
- Miners will get less reward for mining bitcoin as the time progresses. Hence, the transaction fees will increase to encourage the miners to complete transaction quickly.

Transactions:

- Bitcoin uses **public key cryptography** to verify the digital signature.
- Every user can have one or more wallet addresses but every address will always have a pair of public and private keys.

Example of transaction: Suppose Alice wants to send some bitcoins to Bob. Then, the following will take place:

- Bob sends his address to Alice.
- Alice adds Bob's address and the amount of bitcoins to transfer to a message: a 'transaction' message.
- Alice signs the transaction with her private key, and announces her public key for signature verification.
- Alice broadcasts the transaction on the Bitcoin network for all to see.

The first are done manually by humans. Last are taken care by Bitcoin client software itself.

Handling Double Spending:

When the same money is spent more than once, it is called Double Spending. For example, Alice has 50 bitcoins and she sent 50 bitcoins to Bob and 50 bitcoins to Eve simultaneously. Then, only one of the transaction will be successful because double spending is not allowed in Bitcoin.

Bitcoin prevents double spending by following ways:

- Details about the transaction are sent and forwarded to all or as many other computers as possible.
- A constantly growing chain of blocks that contains a record of all transactions is collectively maintained by all computers (each has a full copy).
- To be accepted in the chain, transaction blocks must be valid and must include proof of work (one block generated by the network every 10 minutes).
- Blocks are chained in a way so that, if any one is modified, all following blocks will have to be recomputed.

- When multiple valid continuations to this chain appear, only the longest such branch is accepted and it is then extended further.

Anonymity:

- There are no specific usernames, emails, passwords to hold bitcoins.
- Each balance is simply associated with an **address and its public-private key pair.**
- Transacting parties do not need to know each other's identity in the same way that a store owner does not know a cash-paying customer's name.
- A Bitcoin address mathematically corresponds to a public key and looks like this: 1PHYrmdJ22MKbJevpb3MBNpVckjZHt89hz.
- A single person can have multiple addresses making it difficult to know what amount of bitcoins the person holds.

Bitcoin Script.

Bitcoin Script is a programming language to validate bitcoin transactions. A script is essentially a list of instructions recorded with each transaction that describe how the next person wanting to spend the Bitcoins being transferred can gain access to them.

Bitcoin script is FORTH like language which is processed left to right and is based on stack. It is not turing complete, i.e. doesn't support loops.

Implementation of Bitcoin Script.

A transaction is characterised by two components:

- **Output of transaction:** Sender send's bitcoins.
- **Input of transaction:** Receiver receives bitcoins.

Traditionally, Alice sends her public key and signature along with the transaction to Bob. Then, Bob verifies the origin of transaction.

Now, Bitcoin transfers **script (scriptSig, ScriptPubKey)**instead of public key and signature to Bob. Bob can spend coins only if both the scripts return *true* after execution.

What is the benefit of Scripting?

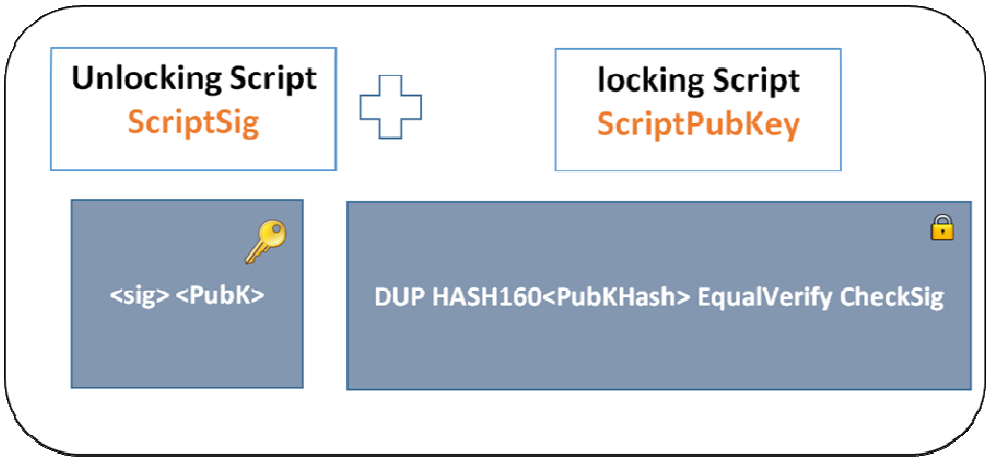
Scripting provides the flexibility to change the parameters of what's needed to spend transferred Bitcoins. For example, the scripting system could be used to require two private keys, or a combination of several keys, or even no keys at all.

A transaction is valid if nothing in the combined script triggers failure and the top stack item is True (non-zero) when the script exits.

Only the intended user will be able to spend bitcoin because only receiver's digital signature will return *true* on solving the script.

Example of Bitcoin Script:

Suppose Alice sends some bitcoins to Bob. She sends the output script **ScriptPubKey** to Bob.



Bitcoin Peer to Peer Network.

- It is an ad-hoc network with random topology, Bitcoin protocol runs on TCP port 8333.
- All nodes (users) in the bitcoin network are treated equally.
- New nodes can join any time, non-responding nodes are removed after 3 hours.

How to join this P2P network?

Suppose you want to join an existing Bitcoin P2P network, then you will follow the following steps:

- You send the request message to join the network. There are certain nodes in the network known as **seed nodes** which provide the initial information to the new node.
- You send a message to seed node to **provide the peer addresses**.
- In response, seed node sends a **set of addresses** to consider as peers.
- Among this set, you select some random addresses and make them as **peers** by making virtual links with them.
- You ask the peers to send the **most recent information** of blockchain. After you receive the information, you compare it and keep the copy which is transferred by most number of peers(>50%).

- Now, you can **start the transaction** in the network.

Transactions in Bitcoin Network.

Representation of transaction:

When Alice sends 10 bitcoins to Bob, it will be represented as:

A->B: BTC 10

Mechanism:

- Alice sends 10 bitcoins(along with scripts) to Bob after getting the most recent information.
- This transaction will be broadcast to all the peers.
- They will validate the transaction using script. If it is valid, they will broadcast this to neighbouring peers.
- This process will continue and each node will receive copy of transaction.
- If a node gets the same transaction information from more than one neighbouring peer, it will keep the copy first received and discard all copies.
- If there is more than one transaction happening in the network, then the transactions are stored in order they are received by the node. So, different nodes may have different transaction pools.

Mining Mechanism:

- Miners are certain nodes in the network that have great computational power. **Not all the nodes in network are miners.**
- Miners collect all the transactions flooded in the network and start mining.
- The miner who solves the puzzle first generates a new block in the network.
- That new block will get flooded in the network.
- It may be possible that multiple miners mine same new block for a transaction or different blocks for different transactions simultaneously.
- As seen earlier, only the longest block chain will be accepted and other nodes will be considered as orphaned. The orphaned block chain is called fork.

Reliable Transactions:

- There should be **no conflict** between two transactions.
- User must not be able to **double spend** the bitcoins.

- The script matches with a pre-given set of whitelist scripts — avoid unusual scripts, avoid infinite loops.

51% Rule:

The copy of the block received from more than 51% of the neighbouring blocks is accepted and is broadcasted further in the network. All the other copies can be discarded.

Consensus.

Consensus is a procedure to reach in a common agreement in a distributed or decentralized multi-agent platform.

For example, there are 4 people who could make two decisions, A or B. Three people are in the favour of A and one person in the favour of B. Then, decision A will be considered as majority of people are in its favour.

Need for Consensus:

Reliability and Fault tolerance in a distributed system. Even if there are certain faulty nodes present in the network, correct operations can take place using consensus.

Uses of Consensus:

- **Commit transaction in a database:** If you wish to transfer money from your bank branch to some other bank branch, then all the bank branches need to reach consensus to decide the validity of transaction and then allow the transaction.
- State Machine Replication.
- Clock Synchronisation.

Types of Fault:

- **Crash Fault:** A node suddenly crashes or becomes unavailable in the middle of a communication due to software or hardware failure.
- **Network or Partitioned Fault:** A network fault occurs (say the link failure) and the network gets partitioned.
- **Byzantine Fault:**A node starts behaving maliciously.

Properties of Distributed Consensus:

- **Termination:** Every correct node(non-faulty) decides some value at the end of the protocol.
- **Validity:** If all the individuals proposes the same value, then all correct individuals decide on that value.

- **Integrity:** Every correct individual decides at most one value, and the decided value must be proposed by some individuals.
- **Agreement:** Every correct individual must agree on the same value.

Message Passing System.

Message passing is a technique for invoking behaviour (i.e., running a program) on a computer. The invoking program sends a message to a process (which may be an actor or object) and relies on the process and the supporting infrastructure to select and invoke the actual code to run

Synchronous Message Passing System:

The message is received within a fixed predefined interval of time. The delay time is also guaranteed. Paxos, Raft, Byzantine fault tolerance (BFT) are some synchronous consensus algorithms explored by distributed system community.

Asynchronous Message Passing System:

The message can be delayed for arbitrary period of time. No fixed message reception time.

Impossibility Result(FLP[85]): No consensus can be guaranteed in a purely asynchronous communication system in the presence of any failures. In such a system, Consensus may occur recognisably, rarely or often. FLP stands for Fischer-Lynch-Patterson who were awarded a Dijkstra Prize for this significant work of impossibility result.

Consensus in Open System.

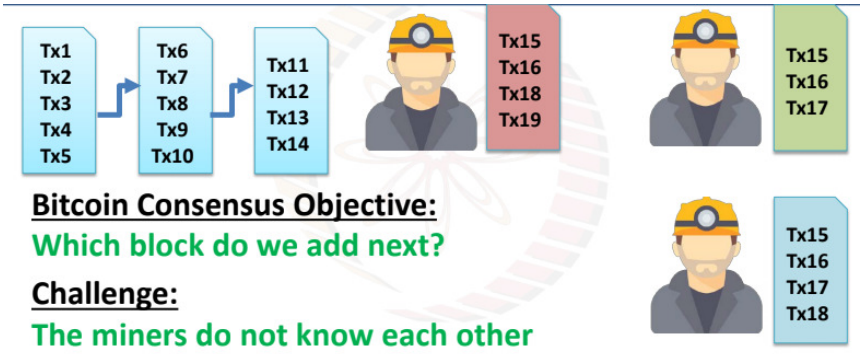
Problems in open system:

- Message passing requires a closed environment. Everyone need to know the identity of others. Since in Bitcoin anyone can join the network anytime, this clearly is not possible in an open environment.
- Shared memory is not suitable for Internet grade computing. Where will we keep the shared memory?

Consensus in Bitcoin:

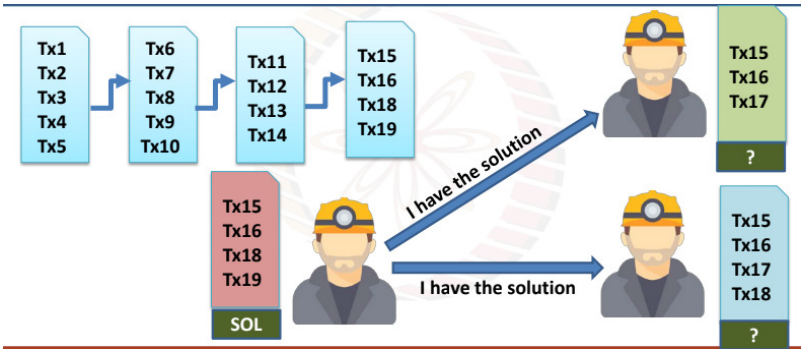
All the nodes in this network need to agree on the correctness of a transaction. Some nodes can also initiate malicious transactions.

Whenever the multiple miners mine new blocks simultaneously, the objective of consensus is to decide which block is to be added next.



Here the miners don't know each other. If we broadcast the transactions (flooding), then the impossibility result might occur because we don't have a global clock. So, it is not feasible to wait for infinite time.

Every miner solves a challenge independently. The miner who completes the challenge first will add the block mined by him to blockchain. This is his Proof of Work(PoW).



Now the miner sends the solution to all the other miners and this block is included in the blockchain. In case any transactions are not logged in the block, they will be included in the next round.

Different Attacks & their Solutions.

Tampering Blockchain:

Each block is mined after doing some work. Thus, the blockchain together contains a large amount of work. If an attacker wants to change any block, he needs to compute all the hashes in the blockchain. This will need a large amount of work which is difficult with current hardware.

Double Spending Problem:

This problem arises when the attacker tries to send the same amount of bitcoins to two different persons.

The solution to the problem in blockchain is that the transactions are irreversible and every transaction is validated against the existing blockchain before accepting it.

A known case of double spending: In November 2013,it was discovered that the GHash.io mining pool appeared to be engaging in repeated payment fraud against BetCoin Dice, a gambling site.

Sybil Attack:

In this attack, the attacker attempts to fill the network with the clients under its control so that he can refuse to relay valid blocks and can perform double spending.

Bitcoin makes these attacks more difficult by only making an outbound connection to one IP address per /16 (x.y.0.0).

Denial of Service(DoS) Attack:

The attacker sends a lot of data in the network to make it busy so that the actual transactions are not able to take place.

Solutions:

- No forwarding of orphaned blocks
- No forwarding of double-spend transactions
- No forwarding of same block or transactions
- Disconnect a peer that sends too many messages
- Restrict the block size to 1 MB
- Limit the size of each script up to 10000 bytes

Proof of Work(Pow).

- This idea was first proposed by Dwork and Naor (1992) to combat junk emails. To discourage the attacker from sending junk emails, the sender had to do some work for the validation of email.
- The work is given by Service provider to the Service requester. This work is moderately hard but feasible for the requester and is easy for the provider to validate.
- The puzzle friendliness property of cryptographic hash functions make them useful to be used as PoW.
- Most implementations of Bitcoin PoW use double SHA256 hash function.
- The probability of getting a PoW is low. It is difficult to say which miner will be able to generate the block. Hence, no single miner will be able to control the network.

Hashcash PoW:

Hashcash is a proof-of-work system used to **limit email spam** and **denial-of-service attacks**, and more recently has become known for its use in bitcoin (and other cryptocurrencies) as part of the mining algorithm. Hashcash was proposed in 1997 by **Adam Back**)

In this a textual encoding of a hashcash stamp is included in an email header to check that the sender has expended a modest amount of CPU time calculating the stamp before sending the email. The header looks like this:
X-Hashcash: 1:20:1303030600:hey@jaindivya.com::McMybZIhxKXu57jd:ckviVersion: number of zero bits in the hashed code: date: resource:
optional extension: string of random characters: counter

The header contains:

- *ver*: Hashcash format version, 1 (which supersedes version 0).
- *bits*: Number of "partial pre-image" (zero) bits in the hashed code.
- *date*: The time that the message was sent, in the format YYMMDD[hhmm[ss]].
- *resource*: Resource data string being transmitted, e.g., an IP address or email address.
- *ext*: Extension (optional; ignored in version 1).
- *rand*: String of random characters, encoded in base-64 format.
- *counter*: Binary counter (up to 220), encoded in base-64 format.

Sender Side Computation:

- Initialise counter to a random number.
- Compute 160-bit SHA-1 hash of the header.
- Accept header if first 20 bits of the header are all 0.
- If not, repeat the above steps with a different counter.

Recipient Side Computation:

- Check if date is within two days.
- Check the email of the sender.
- Check the random string for repetition.
- Compute the 160 bit SHA-1 hash of the entire received string.

- Check if the first 20 bits are zero or not.

On average, the sender will have to try 220 hash values to find a valid header and the receiver required 2 microseconds to validate.

Applications of Hashcash PoW:

- Bitcoin mining.
- Spam filters.
- Comment Spamming prevention.

Monopoly Problem.

During bitcoin's early days, anyone could "mine" it using their home computer. But as the price of digital currency climbed towards $100 in 2013 (it's now over $4,000), professional mining groups with specialised computer chips emerged. Today, these groups, or pools — nearly all based in China — have become concentrated and now dominate the production of new bitcoins.

Why monopoly problem existed?

- Miners are getting less rewards over the time. So, they are discouraged to join as a miner.
- The difficulty of puzzle is increasing which is not possible to be solved by normal hardware.

Solution of Monopoly Problem:

Proof of Stake(PoS) emerged as a solution to this problem. Let's study about it in detail.

Proof of Stake(PoS):

A person can mine or validate block transactions according to how many coins he or she holds. This means that the more Bitcoin or altcoin owned by a miner, the more mining power he or she has

The first cryptocurrency to adopt the PoS method was **Peercoin**.In Peercoin, the coinage is used as a variation of stake. Coinage is calculated by multiplying number of coins by the number of days the coins have been held.

- If an attacker wants to attack, he/she should have more number of bitcoins.
- If the attacker holds majority of bitcoins, then the majority affect will be on attacker only.

Proof of Burn(PoB):

PoB works on the principle of allowing the miners to “burn” or “destroy” the virtual currency tokens, which grants them the right to write blocks in proportion to the coins burnt.

To burn the coins, miners send them to a verifiably un-spendable address. This process does not consume many resources other than the burned coins.

PoB works by burning PoW mined cryptocurrencies. It is power efficient unlike PoW.

Proof of Elapsed Time(PoET):

- PoET is proposed by Intel as a part of hyperledger sawtooth.
- In this each participant in the blockchain network waits a random amount of time. The first participant to finish waiting gets to be leader for the new block.
- To verify that the proposer has really waited for the random amount of time, it relies on a special CPU instruction set called Intel Software Guard Extensions (SGX). SGX allows applications to run *trusted code* in a protected environment.
- **Miners.**

Responsibilities of a miner:

- Validate transactions and construct a block.
- Use hash power to vote on consensus and commit transactions with a new block.
- Store and broadcast the blockchain to the peers.

How a miner mines a bitcoin?

- A miner joins the network and listens for the transactions.
- Listens for new blocks, then validate and re-broadcast a new block when it is proposed.
- Collects transactions for a predefined time and construct a new block.
- Finds a nonce to make the new block valid.
- Broadcasts the new block. Everybody accepts it if it is a part of the main chain.
- Earns the reward for participating in the mining procedure.

Mining Difficulty:

- Mining difficulty is a measure of the amount of difficulty in finding the hash below a given target.

- Bitcoin has a global block difficulty while mining pools have a pool-specific share difficulty.

Difficulty Calculation in Bitcoin:

In Bitcoin, the difficulty changes for every 2016 blocks. The desired rate is that it should take two weeks to mine 2016 blocks provided one block is mined in 10 minutes.

If it takes less than two weeks to mine 2016 blocks, then the difficulty is increased. If it takes more than two weeks to mine 2016 blocks, the difficulty is decreased.
current_difficulty = previous_difficulty *(2 weeks in milliseconds)/(milliseconds to mine last 2016 blocks)

The expected number of hashes we need to calculate to find a block with
difficulty D is :
(D * 2256) / (0xffff * 2208)

Mining Hardware:

- Anyone with access to the internet and suitable hardware can participate in mining.
- In the earliest days of Bitcoin, mining was done with CPUs from normal desktop computers.
- Graphics cards, or graphics processing units (GPUs), are more effective at mining than CPUs and as Bitcoin gained popularity, GPUs became dominant.
- Eventually, hardware known as an ASIC (which stands for Application-Specific Integrated Circuit) was designed specifically for mining Bitcoin. The first ones were released in 2013 and have been improved upon since, with more efficient designs coming to market

---

Mining Pool.

- When the resources are pooled by miners, they create a mining pool.
- The processing power is shared by miners over a network to mine a new block.
- The reward is split proportionally to the amount of work each miner has contributed.
- Slush Pool is the oldest currently active mining pool. AntPool is the largest currently active mining pool.
- Although mining pool allows small miners to participate in the mining process, it also discourages miners for running complete mining procedure.

- Mining centralisation in China is one of Bitcoin's biggest issues at the moment.

Mining Pool Methods.

Terminology:

- *Block Reward:* The new bitcoins distributed by the network to miners for each successfully solved block.
- *Mining Pool Fee:* The fees taken for mining in a pool.

Notations used:

B: Block Reward minus Pool Fee.

p: Probability of finding a block in a share attempt (*p = 1/D*), *D* is the block difficulty.

Pay Per Share(PPS):

- It offers an instant, guaranteed payout to a miner for his contribution to the probability that the pool finds a block.
- Each share costs exactly the expected value of each hash attempt: $R = B*p$.
- This model allows for the least possible variance in payment for miners while also transferring much of the risk to the pool's operator.

Proportional:

- Miners earn shares until the pool finds a block (the end of the mining round).
- After that each user gets reward as per their share:

$R = B * (n/N)$
n = Amount of share of the miner.
N = amount of all shares in the round.**Pay-per-last-N-shares(PPLNS):**

- This is similar to proportional with a difference that the miner's reward is calculated on a basis of N last shares, instead of all shares for the last round.
- If the round was short enough all miners get more profit, and vice versa.

permissioned Blockchain.

- A permissioned blockchain has all the features of the public/permissionless blockchain.
- Blockchain is run among identified and known participants.
- Users know each other but do not trust each other.

- Participants need to obtain an invitation or *permission* to join the network.
- This places restrictions on who is allowed to participate in the network, and only in certain transactions.

Applications of Permissioned Blockchain.

Financial Services:

- **Trade Finance:** Trade finance provides delivery and payment assurance to buyers and sellers. The blockchain can be used by the legal entities to sign all approvals, keeping all parties informed regarding the approval status, when goods are received and when payment is transferred from the importer's to the exporter's bank.
- **Cross-border transactions:** Nostro(ours)/vostro(yours) accounts can become stored account transactions on a blockchain to dramatically improve transparency and efficiency through automated reconciliation of accounts.

Government:

A considerable amount of government involves recording transactions and tracking ownership of assets, all of which can be made more efficient and transparent through the use of blockchain.

Organisations can apply blockchain by issuing digitally authenticated birth certificates that are unforgeable, time-stamped, and accessible to anyone in the world.

Supply Chain Management:

- **Food Safety:** Powered by IBM Blockchain, IBM Food Trust directly connects participants through a permissioned, permanent, and shared record of food origin details, processing data, shipping details, and more.
- **Global Trade:** More than $4 trillion in goods are shipped each year, with 80 percent of those shipments carried by the ocean shipping industry.
  Yet the cost of trade documentation is estimated to reach one-fifth of the actual physical transportation costs. Blockchain's distributed ledger technology to help speed goods on their journey from manufacturer to market, providing one universal view of the truth to unleash new transparency and remove friction.

Health Care:

Blockchain holds the complete medical history for each patient, with multiple granularities of control by the patient, doctors, regulators, hospitals, insurers, and so on, providing a secure mechanism to record and maintain comprehensive medical histories for every patient.

Difference in Permissioned & Permission-less Blockchain:

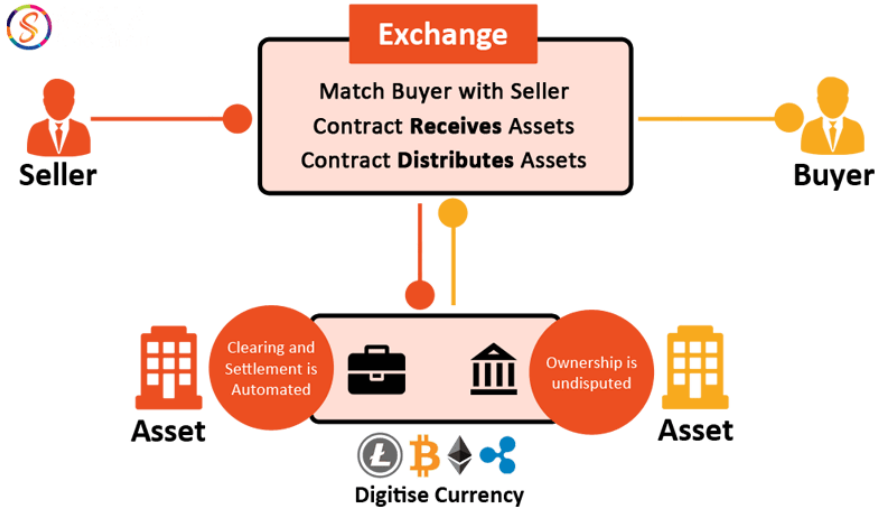
	Permission-less	Permissioned
Access	Open read/write access to database	Permissioned read/write access to database
Scale	Scale to a large number of nodes, but not in transaction throughput	Scale in terms of transaction throughput, but not to a large number of nodes
Consensus	Proof of work/ proof of stake	Closed membership consensus algorithms
Identity	Anonymous/pseudonymous	Identities of nodes are known, but transaction identities can be private/anonymous/pseudonymous
Asset	Native assets	Any asset/data/state

Smart Contracts.

- A smart contract is an **agreement or set of rules** that govern a business transaction.
- It is stored on the blockchain and is executed automatically as part of a transaction.
- Smart contracts allow transactions to be carried out without the need for a governance, legal system, central authority or external enforcement mechanism.
- Smart contracts are created by computer programmers with the help of smart contract development tools available that are digital and compiled using programming code languages such as C++, Go, Python, Java.

Benefits of Smart Contract:

- **Trust:** All documents are encrypted on a shared ledger. Also all the entities or parties could have access to these documents.
- **Autonomous:** All third parties become obsolete in the interactions
- **Security:** All documents are encrypted end to end which makes them near-impenetrable by unethical methods.
- **Redundancy:** Documents are duplicated many times over on the blockchain, and can't ever be "lost".
- **Savings:** Smart contracts save you money by taking out the middleman.
- **Speed:** These contracts automatically self-execute, saving you precious time.
- **Transparency:** For organisations like governments, they could add another level of transparency to dealings.
- **Precision:** Smart contracts execute the exact code provided, ensuring zero errors.



Design Limitations of Smart Contract:

1. **Sequential Execution:**

- Smart Contract executes transactions sequentially based on consensus.
- Requests to the application (smart contract) are ordered by the consensus, and executed in the same order.
- This give a bound on the effective throughput — throughput is inversely proportional.
- There can be a possible attack on the smart contract platform by introducing contract which will take long time to execute.

2. Non-deterministic Execution:

- Smart-contract execution should always needs to be deterministic; otherwise the system may lead to inconsistent states (many fork in the blockchain).
- Iteration over a map may produce a different order in two executions like in Go Lang.

3. Execution on all nodes:

- Execute smart contracts at all nodes, and propagate the state to others to reach consensus.
- Propagate same state to all nodes, verify that the states match.

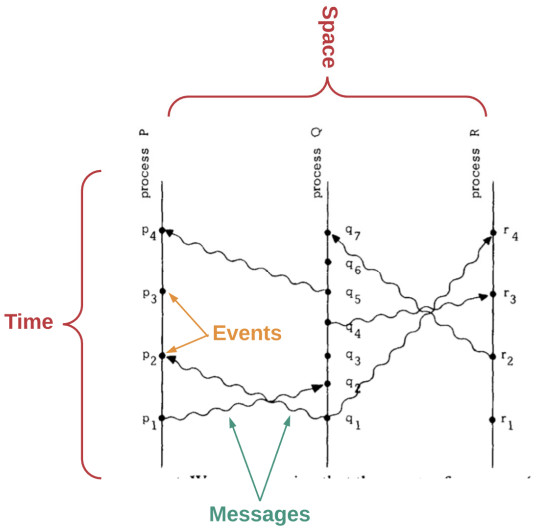
Properties of a Distributed System

Every distributed system has a specific set of characteristics. These include:

A) Concurrency

The processes in the system operate concurrently, meaning multiple events occur simultaneously. In other words, each computer in the network executes events independently at the same time as other computers in the network.

This requires coordination.



Lamport, L (1978). Time, Clocks and Ordering of Events in a Distributed System

B) Lack of a global clock

For a distributed system to work, we need a way to determine the order of events. However, in a set of computers operating concurrently, it is sometimes impossible to say that one of two events occurred first, as computers are spatially separated. In other words, there is no single global clock that determines the sequence of events happening across all computers in the network.

C) Independent failure of components

A critical aspect of understanding distributed systems is acknowledging that components in a distributed system are faulty. This is why it's called "fault-tolerant distributed computing."

It's impossible to have a system free of faults. Real systems are subject to a number of possible flaws or defects, whether that's a process crashing; messages being lost, distorted, or duplicated; a network partition delaying or dropping messages; or even a process going completely haywire and sending messages according to some malevolent plan.

It's impossible to have a system free of faults.

These failures can be broadly classified into three categories:

- **Crash-fail**: The component stops working without warning (e.g., the computer crashes).

- **Omission**: The component sends a message but it is not received by the other nodes (e.g., the message was dropped).
- **Byzantine**: The component behaves arbitrarily. This type of fault is irrelevant in controlled environments (e.g., Google or Amazon data centers) where there is presumably no malicious behavior. Instead, these faults occur in what's known as an "adversarial context." Basically, when a decentralized set of independent actors serve as nodes in the network, these actors may choose to act in a "Byzantine" manner. This means they maliciously choose to alter, block, or not send messages at all.

With this in mind, the aim is to design protocols that allow a system with faulty components to still achieve the common goal and provide a useful service.

Given that every system has faults, a core consideration we must make when building a distributed system is whether it can survive even when its parts deviate from normal behavior, whether that's due to non-malicious behaviors (i.e., crash-fail or omission faults) or malicious behavior (i.e., Byzantine faults).

Broadly speaking, there are two types of models to consider when making a distributed system:

1) Simple fault-tolerance

In a simple fault-tolerant system, we assume that all parts of the system do one of two things: they either follow the protocol exactly or they fail. This type of system should definitely be able to handle nodes going offline or failing. But it doesn't have to worry about nodes exhibiting arbitrary or malicious behavior.

A) Byzantine fault-tolerance

A simple fault-tolerant system is not very useful in an uncontrolled environment. In a decentralized system that has nodes controlled by independent actors communicating on the open, permissionless internet, we also need to design for nodes that choose to be malicious or "Byzantine." Therefore, in a Byzantine fault-tolerant system, we assume nodes can fail or be malicious.

D) Message passing

As noted earlier, computers in a distributed system communicate and coordinate by "message passing" between one or more other computers. Messages can be passed using any messaging protocol, whether that's HTTP, RPC, or a custom protocol built for the specific implementation. There are two types of message-passing environments:

1) Synchronous

In a synchronous system, it is assumed that messages will be delivered within some fixed, known amount of time.

Synchronous message passing is conceptually less complex because users have a guarantee: when they send a message, the receiving component will get it within a certain time frame. This allows users to model their protocol with a fixed upper bound of how long the message will take to reach its destination.

However, this type of environment is not very practical in a real-world distributed system where computers can crash or go offline and messages can be dropped, duplicated, delayed, or received out of order.

2) Asynchronous

In an asynchronous message-passing system, it is assumed that a network may delay messages infinitely, duplicate them, or deliver them out of order. In other words, there is no fixed upper bound on how long a message will take to be received.

What It Means to Have Consensus in a Distributed System

So far, we've learned about the following properties of a distributed system:

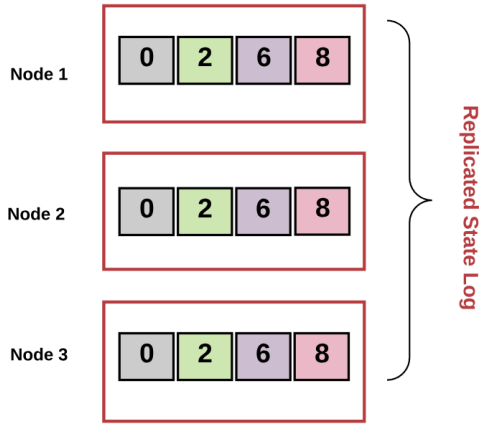
- Concurrency of processes
- Lack of a global clock
- Faulty processes
- Message passing

Next, we'll focus on understanding what it means to achieve "consensus" in a distributed system. But first, it's important to reiterate what we alluded to earlier: there are hundreds of hardware and software architectures used for distributed computing.

The most common form is called a replicated state machine.

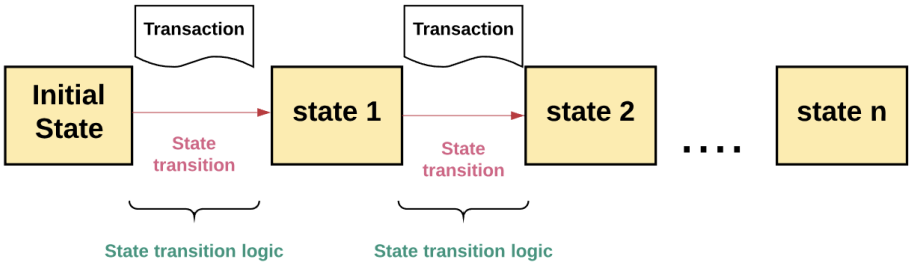
Replicated State Machine

A replicated state machine is a deterministic state machine that is replicated across many computers but functions as a single state machine. Any of these computers may be faulty, but the state machine will still function.



In a replicated state machine, if a transaction is valid, a set of inputs will cause the state of the system to transition to the next state. A transaction is an atomic operation on a database. This means the operations either complete in full or never complete at all. The set of transactions maintained in a replicated state machine is known as a “transaction log.”

The logic for transitioning from one valid state to the next is called the “state transition logic.”



Diagram

In other words, a replicated state machine is a set of distributed computers that all start with the same initial value. For each state transition, each of the processes decides on the next value. Reaching “consensus” means that all the computers must collectively agree on the output of this value.

In turn, this maintains a consistent transaction log across *every* computer in the system (i.e., they “achieve a common goal”). The replicated state machine must continually accept new transactions into this log (i.e., “provide a useful service”). It must do so despite the fact that:

1. Some of the computers are faulty.
2. The network is not reliable and messages may fail to deliver, be delayed, or be out of order.
3. There is no global clock to help determine the order of events.

And this, my friends, is the fundamental goal of any consensus algorithm.

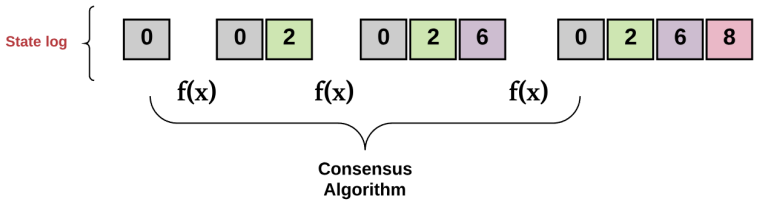


Diagram.

The Consensus Problem, Defined

An algorithm achieves consensus if it satisfies the following conditions:

- **Agreement**: All non-faulty nodes decide on the same output value.
- **Termination**: All non-faulty nodes eventually decide on some output value.

*Note: Different algorithms have different variations of the conditions above. For example, some divide the **Agreement** property into **Consistency** and **Totality**. Some have a concept of **Validity** or **Integrity** or **Efficiency**. However, such nuances are beyond the scope of this post.*

Broadly speaking, consensus algorithms typically assume three types of actors in a system:

1. **Proposers**, often called leaders or coordinators.
2. **Acceptors**, processes that listen to requests from proposers and respond with values.
3. **Learners**, other processes in the system which learn the final values that are decided upon.

Generally, we can define a consensus algorithm by three steps:

Step 1: Elect

- Processes elect a single process (i.e., a leader) to make decisions.
- The leader proposes the next valid output value.

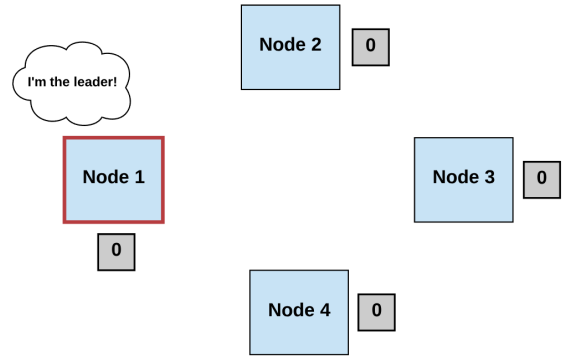
Step 2: Vote

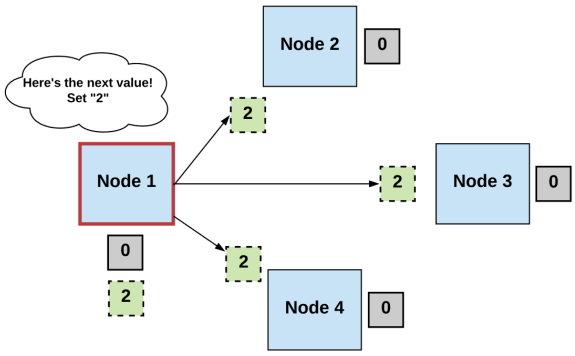
- The non-faulty processes listen to the value being proposed by the leader, validate it, and propose it as the next valid value.

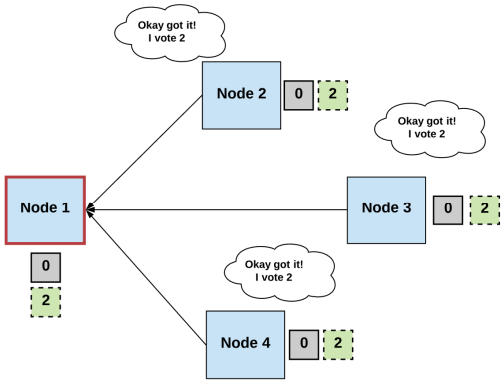
Step 3: Decide

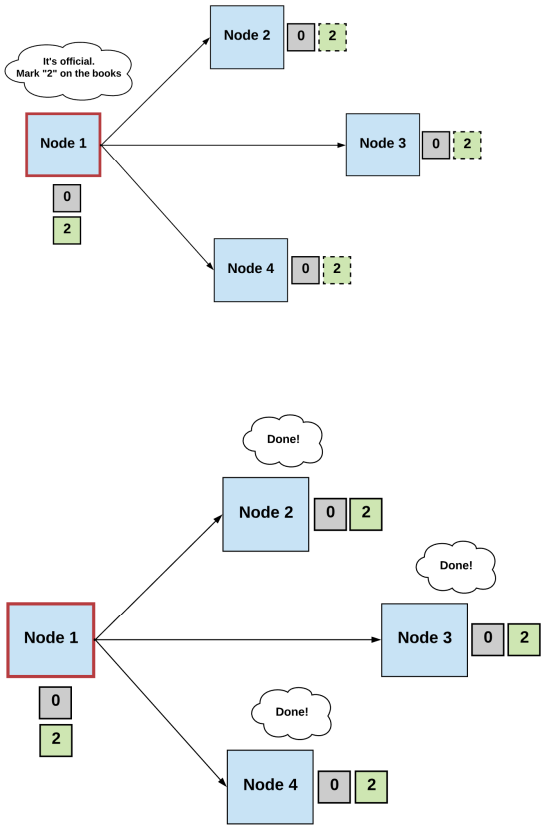
- The non-faulty processes must come to a consensus on a single correct output value. If it receives a threshold number of identical votes which satisfy some criteria, then the processes will decide on that value.

- Otherwise, the steps start over.









Diagrams 1 to 5

It's important to note that every consensus algorithm has different:

- Terminology (e.g., rounds, phases),
- Procedures for how votes are handled, and
- Criteria for how a final value is decided (e.g., validity conditions).

Nonetheless, if we can use this generic process to build an algorithm that guarantees the general conditions defined above, then we have a distributed system which is able to achieve consensus.

FLP impossibility

… Not really. But you probably saw that coming!

Recall how we described the difference between a synchronous system and asynchronous system:

- In synchronous environments, messages are delivered within a fixed time frame
- In asynchronous environments, there's no guarantee of a message being delivered.

This distinction is important.

Reaching consensus in a synchronous environment is possible because we can make assumptions about the maximum time it takes for messages to get delivered. Thus, in this type of system, we can allow the different nodes in the system to take turns proposing new transactions, poll for a majority vote, and skip any node if it doesn't offer a proposal within the maximum time limit.

But, as noted earlier, assuming we are operating in synchronous environments is not practical outside of controlled environments where message latency is predictable, such as data centers which have synchronized atomic clocks.

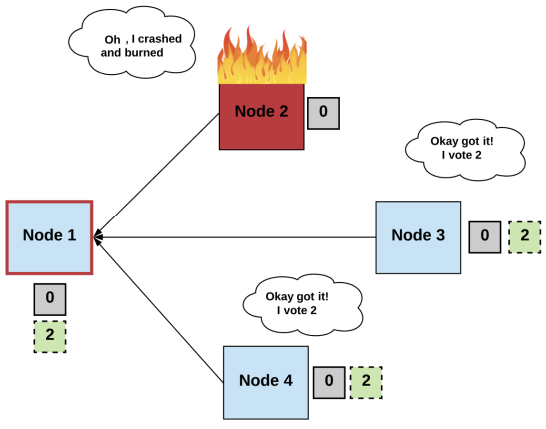
In reality, most environments don't allow us to make the synchronous assumption. So we must design for asynchronous environments.

If we cannot assume a maximum message delivery time in an asynchronous environment, then achieving termination is much harder, if not impossible. Remember, one of the conditions that must be met to achieve consensus is "termination," which means every non-faulty node *must decide* on some output value.

This is formally known as the "FLP impossibility result." How did it get this name? Well, I'm glad you asked!

Even a single faulty process makes it impossible to reach consensus among deterministic asynchronous processes.

In their 1985 paper "Impossibility of Distributed Consensus with One Faulty Process," researchers Fischer, Lynch, and Paterson (aka FLP) show how even a single faulty process makes it impossible to reach consensus among deterministic asynchronous processes. Basically, because processes can fail at unpredictable times, it's also possible for them to fail at the exact opportune time that prevents consensus from occurring.



Diagram

This result was a huge bummer for the distributed computing space. Nonetheless, scientists continued to push forward to find ways to circumvent FLP impossibility.

At a high level, there are two ways to circumvent FLP impossibility:

1. Use synchrony assumptions.
2. Use non-determinism.

Let's take a deep dive into each one right now.

Approach 1: Use Synchrony Assumptions

Let's revisit our impossibility result. Here's another way to think about it: the FLP impossibility result essentially shows that, if we cannot make progress in a system, then we cannot reach consensus. In other words, if messages are asynchronously delivered, termination cannot be guaranteed. Recall that termination is a required condition that means every non-faulty node must eventually decide on some output value.

But how can we guarantee every non-faulty process will decide on a value if we don't know when a message will be delivered due to asynchronous networks?

To be clear, the finding does not state that consensus is unreachable. Rather, due to asynchrony, consensus cannot be reached in a fixed time. Saying that consensus is "impossible" simply means that consensus is "not always possible." It's a subtle but crucial detail.

One way to circumvent this is to use timeouts. If no progress is being made on deciding the next value, we wait until a timeout, then start the steps all over again. As we're about to see, this is what consensus algorithms like Paxos and Raft essentially did.

Paxos

Introduced in the 1990s, Paxos was the first real-world, practical, fault-tolerant consensus algorithm. It's one of the first widely adopted consensus algorithms to be proven correct by Leslie Lamport and has been used by global internet companies like Google and Amazon to build distributed services.

Paxos works like this:

Phase 1: Prepare request

1. The proposer chooses a new proposal version number (n) and sends a "prepare request" to the acceptors.
2. If acceptors receive a prepare request ("prepare," n) with n greater than that of any prepare request they had already responded to, the acceptors send out ("ack," n, n', v') or ("ack," n, ^ , ^).
3. Acceptors respond with a promise not to accept any more proposals numbered less than n.

4. Acceptors suggest the value (v) of the highest-number proposal that they have accepted, if any. Or else, they respond with ^.

Phase 2: Accept request

1. If the proposer receives responses from a majority of the acceptors, then it can issue an accept request ("accept," n, v) with number n and value v.

2. n is the number that appeared in the prepare request.

3. v is the value of the highest-numbered proposal among the responses.

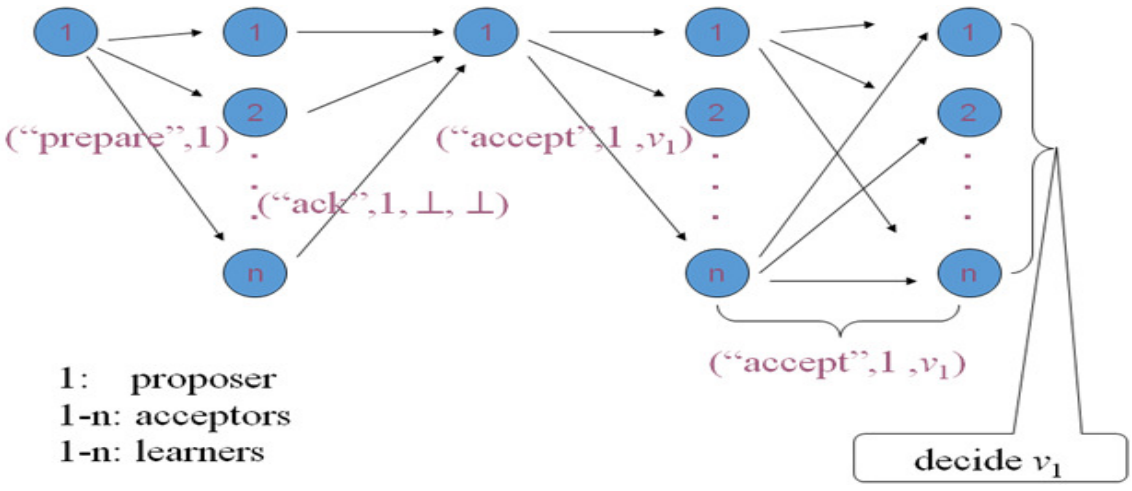
4. If the acceptor receives an accept request ("accept," n, v), it accepts the proposal unless it has already responded to a prepare request with a number greater than n.

Phase 3: Learning phase

1. Whenever an acceptor accepts a proposal, it responds to all learners ("accept," n, v).

2. Learners receive ("accept," n, v) from a majority of acceptors, decide v, and send ("decide," v) to all other learners.

3. Learners receive ("decide," v) and the decided v.



Source: https://www.myassignmenthelp.net/paxos-algorithm-assignment-help

As we now know, every distributed system has faults. In this algorithm, if a proposer failed (e.g., because there was an omission fault), then decisions could be delayed. Paxos dealt with this by starting with a new version number in Phase 1, even if previous attempts never ended.

In favor of offering flexibility in implementation, several specifications in key areas are left open-ended. Things like leader election, failure detection, and log management are vaguely or completely undefined.

This design choice ended up becoming one of the biggest downsides of Paxos. It's not only incredibly difficult to understand but difficult to implement as well. In turn, this made the field of distributed systems incredibly hard to navigate.

In Paxos, although timeouts are not explicit in the algorithm, when it comes to the actual implementation, electing a new proposer after some timeout period is necessary to achieve termination. Otherwise, we couldn't guarantee that acceptors would output the next value, and the system could come to a halt.

Raft

In 2013, Ongaro and Ousterhout published a new consensus algorithm for a replicated state machine called Raft, where the core goal was understandability (unlike Paxos).

One important new thing we learned from Raft is the concept of using a shared timeout to deal with termination. In Raft, if you crash and restart, you wait at least one timeout period before trying to get yourself declared a leader, and you are guaranteed to make progress.

But Wait… What about 'Byzantine' Environments?

While traditional consensus algorithms (such as Paxos and Raft) are able to thrive in asynchronous environments using some level of synchrony assumptions (i.e. timeouts), they are not Byzantine fault-tolerant. They are only crash fault-tolerant.

Crash-faults are easier to handle because we can model the process as either working or crashed — 0 or 1. The processes can't act maliciously and lie. Therefore, in a crash fault-tolerant system, a distributed system can be built where a simple majority is enough to reach a consensus.

In an open and decentralized system (such as public blockchains), users have no control over the nodes in the network. Instead, each node makes decisions toward its individual goals, which may conflict with those of other nodes.

In a Byzantine system where nodes have different incentives and can lie, coordinate, or act arbitrarily, you cannot assume a simple majority is enough to reach consensus. Half or more of the supposedly honest nodes can coordinate with each other to lie.

For example, if an elected leader is Byzantine and maintains strong network connections to other nodes, it can compromise the system. Recall how we said we must model our system to either tolerate simple faults or Byzantine faults. Raft and Paxos are simple fault-tolerant but not Byzantine fault-tolerant. They are not designed to tolerate malicious behavior.

The 'Byzantine General's Problem'

Trying to build a reliable computer system that can handle processes that provide conflicting information is formally known as the "Byzantine General's Problem." A Byzantine fault-tolerant protocol should be able to achieve its common goal even against malicious behavior from nodes.

The paper *"Byzantine General's Problem"* by Leslie Lamport, Robert Shostak, and Marshall Pease provided the first proof to solve the Byzantine General's problem: it showed that a system with **x** Byzantine nodes must have at least **3x + 1 total nodes** in order to reach consensus.

Here's why:

If **x** nodes are faulty, then the system needs to operate correctly after coordinating with **n minus x** nodes (since x nodes might be faulty/Byzantine and not responding). However, we must prepare for the possibility that the **x** that doesn't respond may not be faulty; it could be the **x** that *does* respond. If we want the number of non-faulty nodes to outnumber the number of faulty nodes, we need at least **n minus x minus x > x**. Hence, **n > 3x + 1** is optimal.

the concerns for reaching consensus in a Byzantine and asynchronous setting into two buckets: safety and liveness.

Safety

This is another term for the "agreement" property we discussed earlier, where all non-faulty processes agree on the same output. If we can guarantee safety, we can guarantee that the system as a whole will stay in sync. We want all nodes to agree on the total order of the transaction log, despite failures and malicious actors. A violation of safety means that we end up with two or more valid transaction logs.

Liveness

This is another term for the "termination" property we discussed earlier, where every non-faulty node eventually decides on some output value. In a blockchain setting, "liveness" means the blockchain keeps growing by adding valid new blocks. Liveness is important because it's the only way that the network can continue to be useful — otherwise, it will stall.

As we know from the FLP impossibility, consensus can't be achieved in a completely asynchronous system. The DLS paper argued that making a partial synchrony assumption for achieving the liveness condition is enough to overcome FLP impossibility.

Remember that if nodes aren't deciding on some output value, the system just halts. So, if we make some synchrony assumptions (i.e., timeouts) to guarantee termination and one of those fails, it makes sense that this would also bring the system to a stop.

The PBFT Algorithm

Another Byzantine fault-tolerant algorithm, published in 1999 by Miguel Castro and Barbara Liskov, was called "Practical Byzantine Fault-Tolerance" (PBFT). It was deemed to be a more "practical" algorithm for systems that exhibit Byzantine behavior.

"Practical" in this sense meant it worked in asynchronous environments like the internet and had some optimizations that made it faster than previous consensus algorithms. The paper argued that previous

algorithms, while shown to be "theoretically possible," were either too slow to be used or assumed synchrony for safety.

And as we've explained, that can be quite dangerous in an asynchronous setting.

In a nutshell, the PBFT algorithm showed that it could provide safety and liveness assuming **(n-1)/3** nodes were faulty. As we previously discussed, that's the minimum number of nodes we need to tolerate Byzantine faults. Therefore, the resiliency of the algorithm was optimal.

The algorithm provided safety regardless of how many nodes were faulty. In other words, it didn't assume synchrony for safety. The algorithm did, however, rely on synchrony for liveness. At most, **(n-1)/3** nodes could be faulty and the message delay did not grow faster than a certain time limit. Hence, PBFT circumvented FLP impossibility by using a synchrony assumption to guarantee liveness.

The algorithm moved through a succession of "views," where each view had one "primary" node (i.e., a leader) and the rest were "backups." Here's a step-by-step walkthrough of how it worked:

1. A new transaction happened on a client and was broadcast to the primary.
2. The primary multicasted it to all the backups.
3. The backups executed the transaction and sent a reply to the client.
4. The client wanted **x + 1** replies from backups with the same result. This was the final result, and the state transition happened.

If the leader was non-faulty, the protocol worked just fine. However, the process for detecting a bad primary and reelecting a new primary (known as a "view change") was grossly inefficient. For instance, in order to reach consensus, PBFT required a quadratic number of message exchanges, meaning every computer had to communicate with every other computer in the network.

Note: Explaining the PBFT algorithm in full is a blog post all on its own! We'll save that for another day ;).

While PBFT was an improvement over previous algorithms, it wasn't practical enough to scale for real-world use cases (such as public blockchains) where there are large numbers of participants. But hey, at least it was much more specific when it came to things like failure detection and leader election (unlike Paxos).

It's important to acknowledge PBFT for its contributions. It incorporated important revolutionary ideas that newer consensus protocols (especially in a post-blockchain world) would learn from and use.

For example, Tendermint is a new consensus algorithm that is heavily influenced by PBFT. In their "validation" phase, Tendermint uses two voting steps (like PBFT) to decide on the final value. The key difference with Tendermint's algorithm is that it's designed to be more practical.

For instance, Tendermint rotates a new leader every round. If the current round's leader doesn't respond within a set period of time, the leader is skipped and the algorithm simply moves to the next round with a new leader. This actually makes a lot more sense than using point-to-point connections every time there needs to be a view-change and a new leader elected.

Main consensus protocols

In distributed systems, there is no perfect consensus protocol. The consensus protocol needs to make a trade-off among consistency, availability and partition fault tolerance (CAP) [3]. Besides, the consensus protocol also needs to address Byzantine Generals Problem that there will be some malicious nodes deliberately undermining the consensus process [4]. In this section, we make a detailed description of some popular blockchain consensus protocols that can effectively address Byzantine Generals Problem.

PoW (Proof of Work): PoW is adopted by Bitcoin,Ethereum, etc [1], [5]. PoW selects one node to create a new block in each round of consensus by computational power competition. In the competition, the participating nodes need to solve a cryptographic puzzle. The node who first addresses the puzzle can have a right to create a new block. The flow of the block creation in PoW is presented in Fig. 1. It is very difficult to solve a PoW puzzle. Nodes need to keep adjusting the value of nonce to get the correct answer, which requires much computational power. It is feasible for a malicious attacker to overthrow one block in a chain, but as the valid blocks in the chain increase, the workload is also accumulated, therefore overthrowing a long chain requires a huge amount of computational power. PoW belongs to the probabilistic-finality consensus protocols since it guarantees eventual consistency.

PoS (Proof of Stake): In PoS, selecting each round of node who creates a new block depends on the held stake rather than the computational power. Although nodes still need to solve a SHA256 puzzle:SHA256(timestamp,previoushash…)<target×coin.The different from PoW is that nodes do not need to adjust nonce for many times, instead, the key to solve this puzzle is the amount of stake (coins). Hence, PoS is an energy-saving consensus protocol, which leverages a way of the internal currency incentive instead of consuming lots of computational power to reach a consensus. The flow of PoS is shown in Fig. 2. Like PoW, PoS is also a probabilistic-finality consensus protocol. PPcoin was the first cryptocurrency to apply PoS to the blockchain. In PPcoin, in addition to the size of the stake, the coin age is also introduced in solving a PoS puzzle [6]. For instance, if you hold 10 coins for a total of 20 days, then your coin age is 200. Once a node creates a new block, his coin age will be cleared to 0. In addition to PPcoin, many cryptocurrencies adopt PoS, e.g., Nxt, Ouroboros [7], [8]. Note that Ethereum plans to transition from PoW to PoS.

DPoS (Delegated Proof of Stake): The principle of DPoS is to let nodes who hold stake vote to elect block verifiers (i.e., block creators) [9]. This way of voting makes the stakeholders give the right of creating blocks to the delegates they support instead of creating blocks themselves, thus reducing their computational power consumption to 0. We can clearly see the flow of DPoS in Fig. 3. DPoS is like a parliamentary system, as shown in Fig. 3, if the delegates are unable to generate blocks in their turns, they will be dismissed and the stakeholders will select new nodes to replace them. DPoS makes the most use of the shareholders' votes to reach a consensus in a fair and democratic way. Compared to PoW and PoS, DPoS is a low-cost and high-efficiency consensus protocol. There are also some cryptocurrencies adopting DPoS such as BitShares, EOS, etc [9], [10]. The new version of EOS has turned DPoS to BFT-DPoS (Byzantine Fault Tolerance-DPoS) [10].

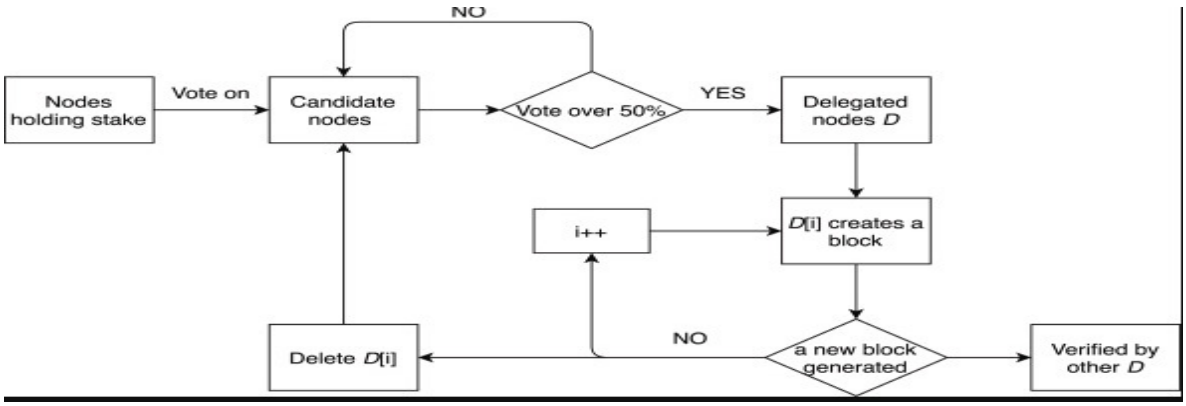


Fig. 3. Flow of DPoS.

PBFT (Practical Byzantine Fault Tolerance): PBFT is a Byzantine Fault Tolerance protocol with low algorithm complexity and high practicality in distributed systems [11]. PBFT contains five phases: *request*, *pre-prepare*, *prepare*, *commit* and *reply*. Fig. 4 describes how PBFT works. The primary node forwards the message sent by the client to the other three nodes. In the case that node 3 is crashed, one message goes through five phases to reach a consensus among these nodes. Finally, these nodes reply to the client to complete a round of consensus. PBFT guarantees nodes maintain a common state and take a consistent action in each round of consensus. PBFT achieves the goal of strong consistency, thus it is an absolute-finality consensus protocol. As mentioned before, EOS takes a combined consensus protocol. EOS leverages PBFT to simultaneously work with the block validation and creation in DPoS, greatly reducing the time required for a round of consensus [10]. A new protocol called Stellar is an improvement of PBFT. Stellar adopts FBA (Federated Byzantine Agreement) protocol, in which nodes can choose the federation they trust to conduct the consensus process [12].

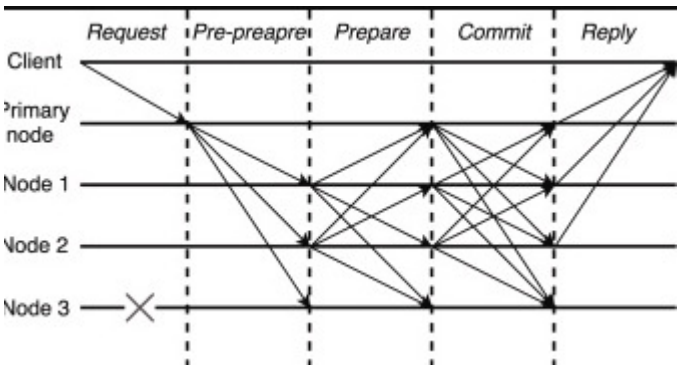


Fig. 4. Process of PBFT.

Ripple: Ripple is an open source payment protocol [13]. In Ripple, transactions are initiated by clients and broadcast throughout the network via tracking nodes or validating nodes. However, the consensus process in Ripple is performed by validating nodes, each of which owns a list of trusted nodes called UNL (Unique Node List). Nodes in UNL can vote on the transactions they support. The process of consensus in Ripple is presented in Fig. 5. Each validating node sends its own transactions set as a proposal to other validating nodes. Once receiving the transaction proposals sent by nodes in UNL, the validating node will check each transaction in the proposal. The transaction in the proposal will get one vote if there is the same transaction in its local transactions set. When the transaction gets more than 50% of the votes, this transaction will enter the next round. The screening threshold will be increased for each round, and transactions with more than 80% of the votes will be finally recorded in the distributed ledger. Hence, Ripple is an absolute-finality consensus protocol.

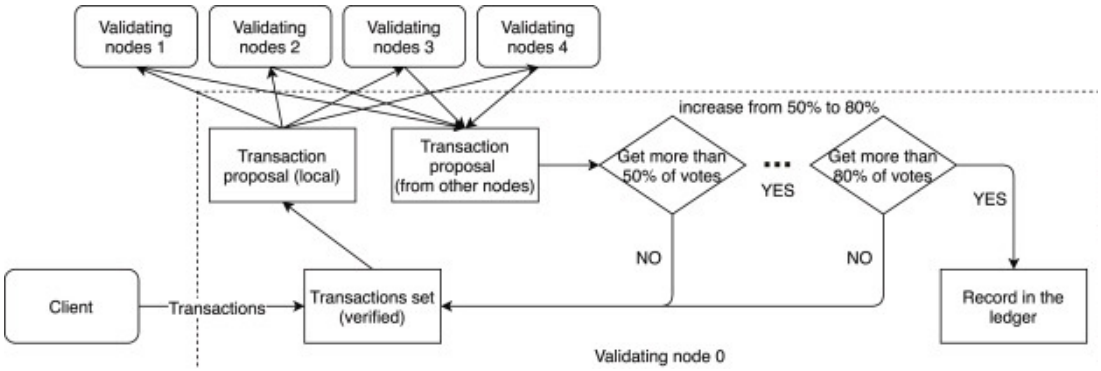


Fig. 5. Process of Ripple.

3. Analysis and comparison

In this section, we analyze the main consensus protocols mentioned in Section 2 in terms of fault tolerance, limitation, scalability, and application scenarios. The analysis and comparison results are summarized in Table 1.

Table 1. Main consensus protocols comparison.

Property	PoW	PoS	DPoS	PBFT	Ripple
Type	Probabilistic-finality	Probabilistic-finality	Probabilistic-finality	Absolute-finality	Absolute-finality
Fault tolerance	50%	50%	50%	33%	20%
Power consumption	Large	Less	Less	Negligible	Negligible
Scalability	Good	Good	Good	Bad	Good

Property	PoW	PoS	DPoS	PBFT	Ripple
Application	Public	Public	Public	Permissioned	Permissioned

3.1. Fault tolerance

PoW, PoS and DPoS are probabilistic-finality protocols, and attackers need to accumulate a large amount of computational power or coins (stake) to create a long private chain to replace a valid chain. For instance, in Bitcoin, a 50% fraction of the computational power is sufficient for an attacker to create a longer private chain to successfully complete a double-spend attack [1]. Hence, if attacker's fraction of the computational power is more than or equal to 50%, the blockchain network will be undermined. Like PoW, PoS and DPoS can only allow the existence of the stakeholder with less than 50% of the held stake. In PBFT, if there are a total of 3f+1 nodes in the network, the number of normal nodes must exceed 2f+1, which means that the number of malicious or crashed nodes must be less than f. Therefore, the fault tolerance of PBFT is 1/3 [11]. The fault tolerance of Ripple is only 20%, i.e., Ripple can tolerate Byzantine Problem in 20% of nodes in the entire network without affecting the correct result of consensus

Proof-of-Work vs. BFT Scalibility

Proof-of-Work (PoW) aspect of the hashchain [22]: a Bitcoin block contains nonces that a Bitcon miner (i.e., a node attempting to add a block to the chain) must set in such a way that the hash of the entire block is smaller than a known target, which is typically a very small number. In fact, in Bitcoin, the difficulty of mining, inversely proportional to the target, is adjusted dynamically throughout the lifetime of the system. The adjustment is made with respect to the block-mining rate and, indirectly, with respect to the computational power of nodes participating in the system, to maintain the expected block-mining rate at roughly one block every 10 min [48].

This latency of 10 minutes (per block) is often referred to as the block frequency (see e.g., [23]) and is one of the two critical "magic numbers" in Bitcoin, the other being the block size, which is set in Bitcoin to 1 MB. In the early days of Bitcoin, the performance scalability of its probabilistic PoW-based blockchain was not a major issue.

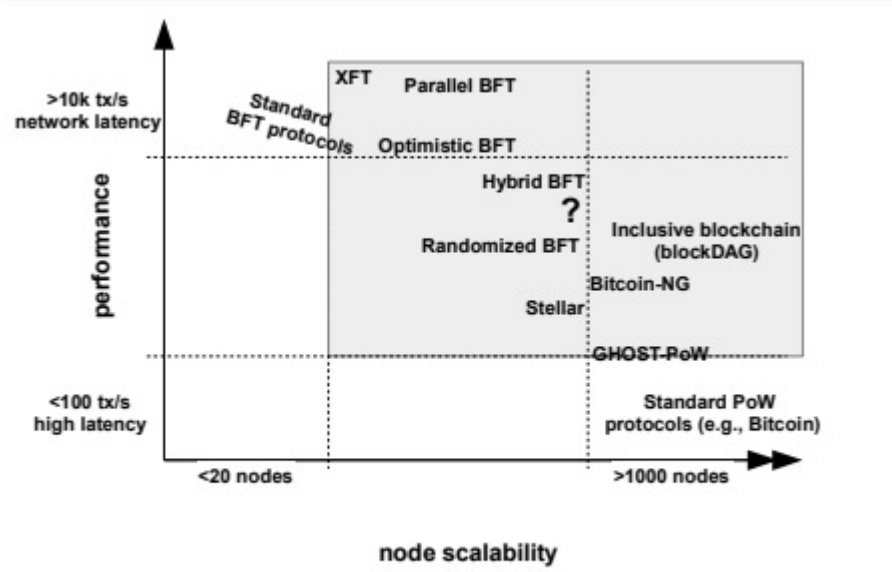
On top of this, the Bitcoin network uses a lot of power, which, in 2014, was roughly estimated to be in the ballpark of 0.1-10 GW [49]. However, blockchain requirements change rapidly, with high latency and low throughput of Bitcoin-like blockchain becoming a major challenge [6]. As a comparison, leading global credit-card payment companies serve roughly 2000 transactions per second on average [59], with a peak capacity designed to sustain more than 10000 transactions per second. Moreover, the trend of modern cryptocurrency platforms, such as Ethereum [58], is to support execution of Turing-complete code on blockchain fabric in the form of smart contracts, which are, roughly speaking, custom, self-executing programs (distributed applications) that automatically enforce properties of a digital contract.

In fact, smart-contract blockchain is seen as a candidate technology for distributed ledgers in many industries. Clearly, in many of the intended smart-contract use cases, distributed applications require much better performance than that offered by Bitcoin. The banking industry is one prominent example, where potential

blockchain use cases go well beyond digital payments [46] to, e.g., securities trade settlements and trade finance.

In more than three decades of research, BFT protocol prototypes have been shown to be practical [10], reaching practically minimal latencies allowed by the network, and supporting tens of thousands transactions per second (see e.g., [35, 3]). However, BFT and state-machine replication protocols in general are often challenged for their scalability in terms of number of nodes (replicas) [8], and have not been throughly tested in this aspect critical to blockchain.

In summary, blockchain consensus technologies of today, PoW and BFT, sit at the two opposite ends of the scalability spectrum. Roughly speaking, PoWbased blockchain offers good node scalability with poor performance, whereas BFT-based blockchain offers good performance for small numbers of replicas, with not-well explored and intuitively very limited scalability. This current state of blockchain scalability is sketched in Figure 1. Given seemingly inherent tradeoffs between the number of replicas and performance, it is not clear today what the optimal blockchain solution is for the sweet spot relevant for many use cases in which the number of nodes n ranges from a few tens to 1000 (or perhaps few thousands).



1. Illustration of performance and scalability of different families of PoW and BFT

Table 1. High-level comparison between PoW and BFT blockchain consensus families for a set of important blockchain properties. Entries in bold suggest desirable features and highlight advantages of one consensus family over the other

	PoW consensus	BFT consensus
Node identity management	**open, entirely decentralized**	permissioned, nodes need to know IDs of all other nodes
Consensus finality	no	**yes**
Scalability (no. of nodes)	**excellent (thousands of nodes)**	limited, not well explored (tested only up to $n \leq 20$ nodes)
Scalability (no. of clients)	**excellent (thousands of clients)**	**excellent (thousands of clients)**
Performance (throughput)	limited (due to possible of chain forks)	**excellent (tens of thousands tx/sec)**
Performance (latency)	high latency (due to multi-block confirmations)	**excellent (matches network latency)**
Power consumption	very poor (PoW wastes energy)	**good**
Tolerated power of an adversary	$\leq 25\%$ computing power	$\leq 33\%$ voting power
Network synchrony assumptions	physical clock timestamps (e.g., for block validity)	**none for consensus safety** (synchrony needed for liveness)
Correctness proofs	no	**yes**

Node identity management. How node identities are managed in PoW and BFT protocols is possibly their most fundamental difference. PoW blockchains feature an entirely decentralized identity management — for example, anybody can download the code for Bitcoin miner, and start participating in the protocol, knowing basically only a single peer to start with.

In contrast, the BFT approach to consensus typically requires every node to know the entire set of its peer nodes participating in consensus. This in turn calls for a (logically) centralized identity management in which a trusted party issues identities and cryptographic certificates to nodes

Consensus Finality If a correct node p appends block b to its copy of the blockchain before appending block b 0 , then no correct node q appends block b 0 before b to its copy of the blockchain.

Performance. Beyond the very limited performance of Bitcoin of up to 7 transactions per second (with the current block size) and 1-hour latency with 6-block confirmation, PoW-based blockchains face inherent performance challenges

In contrast, modern BFT protocols have been confirmed to sustain tens of thousands of transactions with practically network-speed latencies, not only as prototypes (e.g., [35, 12, 3]) but also as practical systems.

Adversary. PoW and BFT consider different adversaries. In PoW blockchains, what matters is the total computational (hashing) power controlled by the adversary.

In contrast, BFT voting schemes are known to tolerate at most n/3 corrupted nodes [20]. This bound holds only when the network is allowed to be (from time to time) fully asynchronous — strengthening synchrony assumptions makes it possible to raise this threshold.

Network synchrony. Bitcoin relies on the local time of a node to timestamp a block. Roughly speaking, a block is accepted as valid if its timestamp is greater than the median of the last 11 blocks

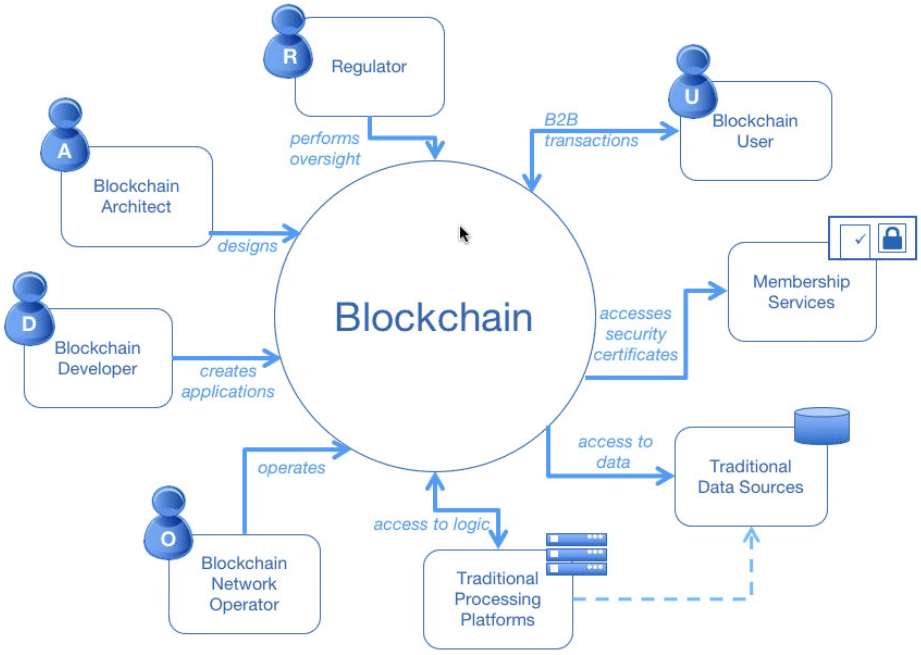
BFT protocols typically do not rely on any physical clock.4 However, eventually synchronous communication is needed to ensure liveness, owing to the FLP consensus impossibility result, which states that consensus is impossible to achieve deterministically with potentially faulty nodes in a purely asynchronous system

Correctness proofs. Historically, state-machine replication protocols, and in particular their BFT variants, have been recognized as very challenging to design and implement). Even if it may be understandable why Bitcoin was originally deployed without having been subjected to similar scrutiny, it is rather surprising that novel PoW blockchains are rarely accompanied by a detailed security and distributed protocol and security analysis

---

Actors in Blockchain.

- **Blockchain Architect:** Responsible for the architecture and design of the blockchain solution.
- **Blockchain Developer:** The developer of applications and smart contracts that interact with the Blockchain and are used by Blockchain users.
- **Blockchain User:** The business user, operating in a business network. This role interacts with the Blockchain using an application. They are not aware of the Blockchain.
- **Blockchain Regulator:** The overall authority in a business network. Specifically, regulators may require broad access to the ledger's contents.
- **Blockchain Operator:** Manages and monitors the Blockchain network. Each business in the network has a Blockchain Network operator.
- **Membership Services:** Manages the different types of certificates required to run a permissioned Blockchain.
- **Traditional Processing Platform:** An existing computer system which may be used by the Blockchain to augment processing. This system may also need to initiate requests into the Blockchain.
- **Traditional Data Sources:** An existing data system which may provide data to influence the behavior of smart contracts.



Components in Blockchain Solution.

Ledger:

A ledger is a channel's chain and current state data which is maintained by each peer on the channel.

Smart Contract:

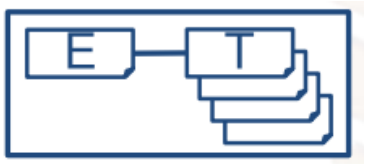
Software running on a ledger, to encode assets and the transaction instructions (business logic) for modifying the assets.

Peer Network:

A broader term overarching the entire transactional flow, which serves to generate an agreement on the order and to confirm the correctness of the set of transactions constituting a block.

Membership:

Membership Services authenticates, authorizes, and manages identities on a permissioned blockchain network.



Events:

Creates notifications of significant operations on the blockchain (e.g. a new block), as well as notifications related to smart contracts.

System Management:

Provides the ability to create, change and monitor blockchain components.

Wallet:

Securely manages a user's security credentials.

System Integration:

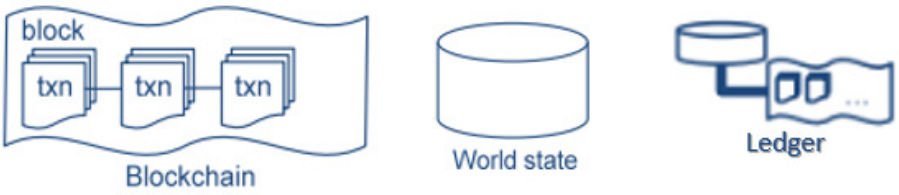
Responsible for integrating Blockchain bi-directionally with external systems. Not part of blockchain, but used with it.

More About Ledger.

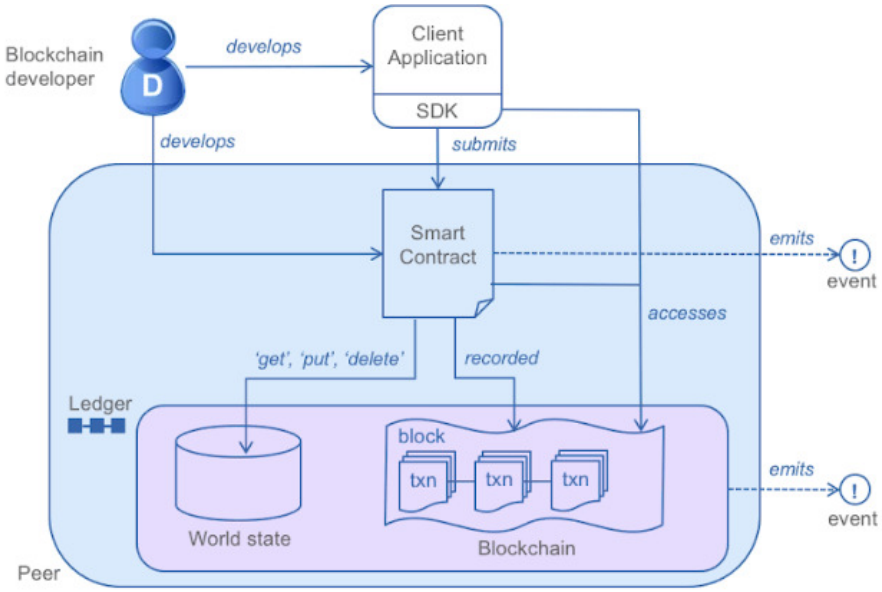
A ledger consists of two data structures:

- Blockchain:
  - A linked list of blocks or hash chain.
  - Each block describes a set of transactions.
  - Immutable, i.e., blocks can not be tampered.

- World State:
  - Stores the most recent state of smart contracts/output of transactions.
  - Stored in a traditional database.
  - Data elements can be deleted, added, modified.



How Applications interact with ledger?



A developer like you and me develops client application and smart contract. The client application will do the following:

- Invoke the functions written in the smart contract.
- Use SDK to submit transactions on the blockchain.
- Access blockchain to see the committed transactions and history of transactions.

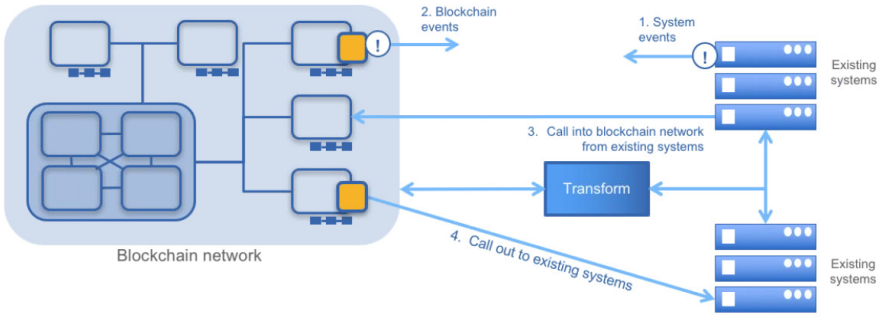
Smart Contract will do the following:

- Record all the transactions on the blockchain.
- Record the modified data elements on world state for every ‘get’, ‘put’ or ‘delete’.

Every peer in the network will perform the above functions. Whenever client application submits data to smart contract, the contract is executed on all the peers and all the nodes update their world state simultaneously. Then, they will agree that the output is valid and consistent. After that the block gets added in the blockchain.

Unlike bitcoin, there will not be any folks or orphaned blocks in permissioned blockchain. An event can be emitted whenever a block is committed/rejected, a function is invoked, disk is full, etc. The event emission depends on the internal processing of the organisation for asynchronous processing systems.

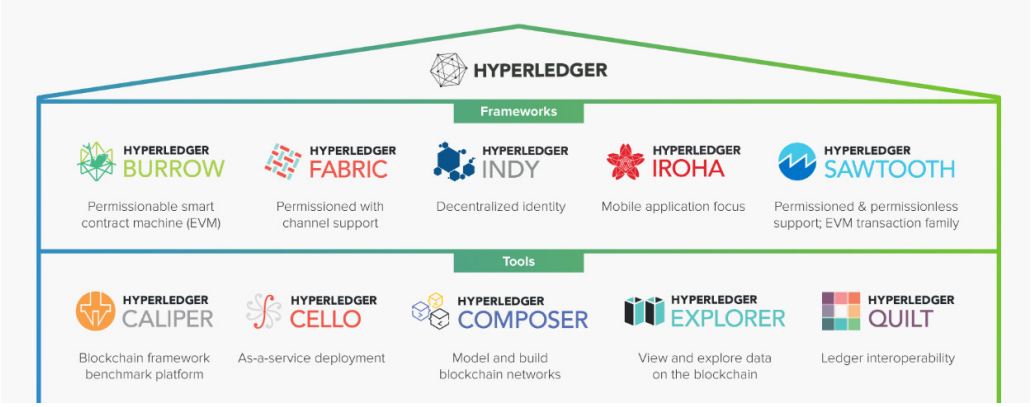
Integrating Blockchain with Existing Systems.



In an organisation, the blockchain can be integrated with the exiting systems. The information can be sent to blockchain and can also be get from the blockchain. This is a two way exchange.

hyperledger.

- Hyperledger is the umbrella open source project that The Linux Foundation has created and hosted since 2015.
- It aims at advancing and promoting cross-industry blockchain technologies to ensure accountability, transparency, and trust among business partners.
- Hyperledger makes business network and transactions more efficient.



- Hyperledger blockchains are generally permissioned blockchains, which means that the parties that join the network are authenticated and authorized to participate on the network.
- Hyperledger's main goal is to create enterprise-grade, open source, distributed ledger frameworks and code bases to support business use cases.

Hyperledger vs Ethereum vs Bitcoin.

	Bitcoin	Ethereum	Hyperledger Frameworks
Cryptocurrency based	Yes	Yes	No
Permissioned	No	No	Yes (in general)*
Pseudo-anonymous	Yes	No	No
Auditable	Yes	Yes	Yes
Immutable ledger	Yes	Yes	Yes
Modularity	No	No	Yes
Smart contracts	No	Yes	Yes
Consensus protocol	PoW	PoW	Various**

Hyperledger Fabric.

Hyperledger Fabric was **the first proposal for a codebase**, combining previous work done by Digital Asset Holdings, Blockstream's libconsensus, and IBM's OpenBlockchain.

Hyperledger Fabric provides a **modular architecture**, which allows components such as consensus and membership services to be **plug-and-play**.

Hyperledger Fabric is revolutionary in allowing entities to conduct **confidential transactions** without passing information through a central authority. This is accomplished through different channels that run within the network, as well as the division of labor that characterizes the different nodes within the network.

Hyperledger Fabric supports **permissioned deployments**.

Roles within Hyperledger Fabric Network:

- **Clients**
  Clients are applications that act on behalf of a person to propose transactions on the network.

- **Peers**
  Peers maintain the state of the network and a copy of the ledger. There are two different types of peers: **endorsing** and **committing** peers. However, there is an overlap between endorsing and committing peers, in that endorsing peers are a special kind of committing peers. All peers commit blocks to the distributed ledger.
  - *Endorsers* simulate and endorse transactions
  - *Committers* verify endorsements and validate transaction results, prior to committing transactions to the blockchain.

- **Ordering Service**
  The ordering service accepts endorsed transactions, orders them into a block, and delivers the blocks to the committing peers.

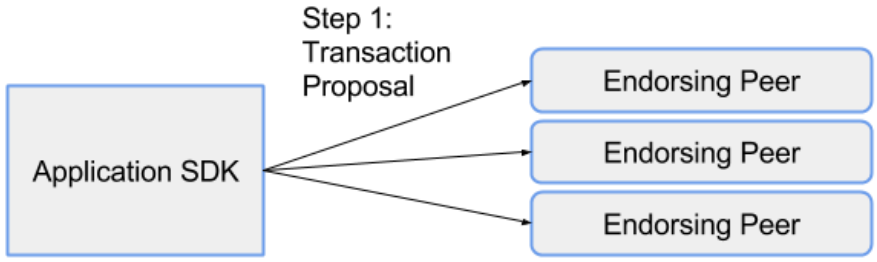
Transaction Flow in Hyperledger Fabric.

In Hyperledger Fabric, consensus is made up of three distinct steps:

- Transaction endorsement
- Ordering
- Validation and commitment

Step I: Propose Transation

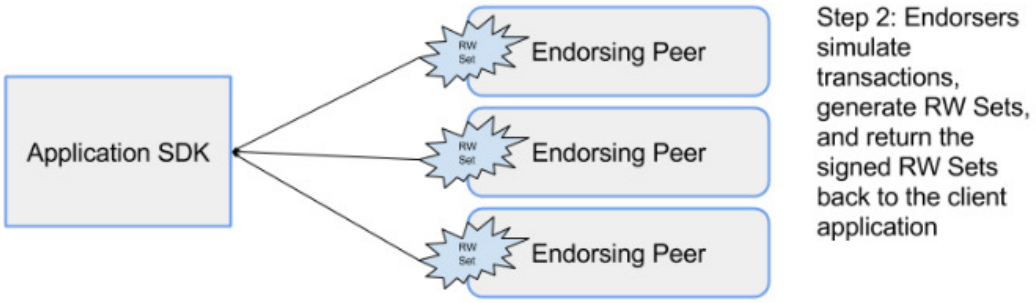
Within a Hyperledger Fabric network, transactions start out with client applications sending transaction proposals, or, in other words, proposing a transaction to endorsing peers.



Client applications are commonly referred to as **applications** or **clients**, and allow people to communicate with the blockchain network. Application developers can leverage the Hyperledger Fabric network through the application SDK.

Step II: Execute Proposed Transaction

Each endorsing peer simulates the proposed transaction, without updating the ledger. The endorsing peers will capture the set of **R**ead and **W**ritten data, called **RW Sets**. These RW sets capture what was read from the current world state while simulating the transaction, as well as what would have been written to the world state had the transaction been executed. These RW sets are then signed by the endorsing peer, and returned to the client application to be used in future steps of the transaction flow.



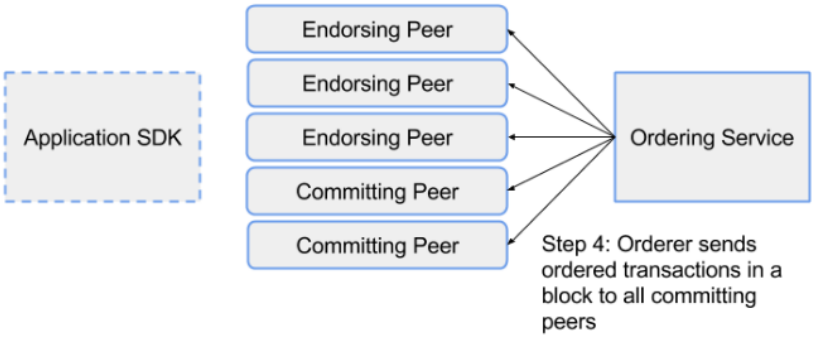
Endorsing peers must hold smart contracts in order to simulate the transaction proposals.

Steps III: Proposal Response

The application then submits the endorsed transaction and the RW sets to the ordering service. Ordering happens across the network, in parallel with endorsed transactions and RW sets submitted by other applications.

Step IV: Order & Deliver Transaction

The ordering service takes the endorsed transactions and RW sets, orders this information into a block, and delivers the block to all committing peers.

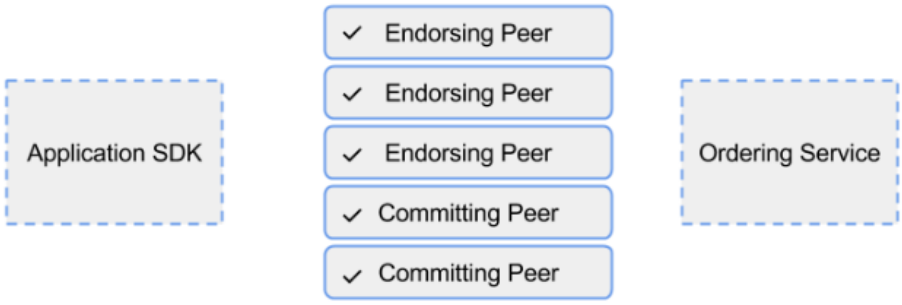


The **ordering service**, which is made up of a cluster of orderers, does not process transactions, smart contracts, or maintains the shared ledger. The ordering service accepts the endorsed transactions and specifies the order in which those transactions will be committed to the ledger.

The Fabric v1.0 architecture has been designed such that the specific implementation of 'ordering' (Solo, Kafka, BFT) becomes a pluggable component. The default ordering service for Hyperledger Fabric is **Kafka**. Therefore, the ordering service is a modular component of Hyperledger Fabric.

Step V: Validate Transaction

The committing peer validates the transaction by checking to make sure that the RW sets still match the current world state. Specifically, that the Read data that existed when the endorsers simulated the transaction is identical to the current world state. When the committing peer validates the transaction, the transaction is written to the ledger, and the world state is updated with the Write data from the RW Set.



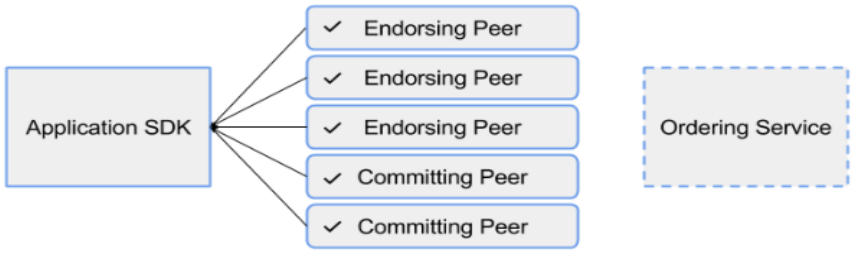
Step 5: Committing peers validate each transaction in the block

If the transaction fails, that is, if the committing peer finds that the RW set does not match the current world state, the transaction ordered into a block will still be included in that block, but it will be marked as invalid, and the world state will not be updated.

Committing peers are responsible for adding blocks of transactions to the shared ledger and updating the world state. They may hold smart contracts, but it is not a requirement.

Step VI: Notify Transaction

Lastly, the committing peers asynchronously notify the client application of the success or failure of the transaction. Applications will be notified by each committing peer.



Step 6: Committing peers asynchronously notify the Application of the results of the transaction.

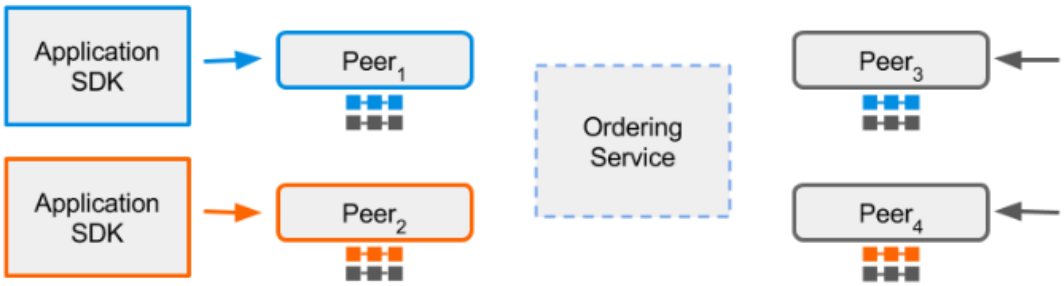
Important Points To Remember:

- The state of the network is maintained by peers, and not by the ordering service or the client.
- Machines that are part of the ordering service should not be set up to also endorse or commit transactions, and vice versa.
- Endorsing peers do commit blocks, but committing peers do not endorse transactions.

Hyperledger Fabric Details.

Channels:

- Channels allow organizations to utilize the same network, while maintaining separation between multiple blockchains.
- Channels can either be shared across an entire network of peers or can be permissioned for a specific set of participants.
- Peers can participate in multiple channels.
- Channels partition the network in order to allow transaction visibility for stakeholders only.



In the above diagram, blue, grey and orange are the channels with each channel having its own application, ledger, and peers.

Ordering Service:

Transactions within a timeframe are sorted into a block and are committed in sequential order by the ordering service.

Unlike the Bitcoin blockchain, where ordering occurs through the solving of a cryptographic puzzle, or *mining*, Hyperledger Fabric allows the organizations running the network to choose the ordering mechanism that best suits that network.

Hyperledger Fabric provides three ordering mechanisms: SOLO, Kafka, and Simplified Byzantine Fault Tolerance (SBFT), the latter of which has not yet been implemented in Fabric v1.0.

- **SOLO** is the Hyperledger Fabric ordering mechanism most typically used by developers experimenting with Hyperledger Fabric networks. SOLO involves a **single ordering node**.
- **Kafka** is the Hyperledger Fabric ordering mechanism that is **recommended for production use**. This ordering mechanism utilizes Apache Kafka, an open source stream processing platform that provides a unified, high-throughput, low-latency platform for handling real-time data feeds. In this case, the data consists of endorsed transactions and RW sets. The Kafka mechanism provides a crash fault-tolerant solution to ordering.
- **SBFT** stands for Simplified Byzantine Fault Tolerance. This ordering mechanism is both **crash fault-tolerant and byzantine fault-tolerant**, meaning that it can reach agreement even in the

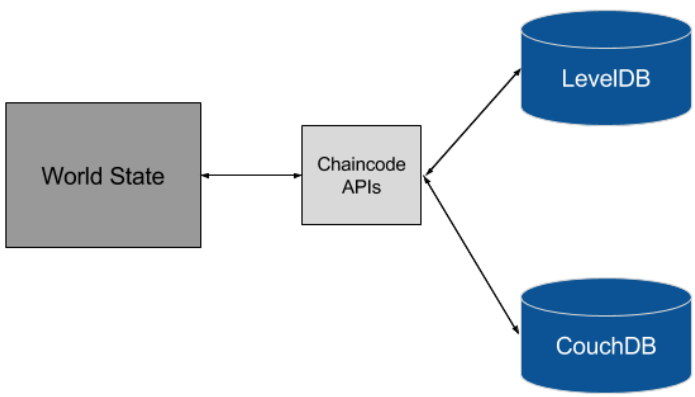
presence of malicious or faulty nodes. The Hyperledger Fabric community has not yet implemented this mechanism, but it is on their road map.

Chain code:

- Hyperledger Fabric smart contracts are called **chaincode** and are written in Go.
- The chaincode serves as the business logic for a Hyperledger Fabric network, in that the chaincode directs how you manipulate assets within the network.
- Applications interact with the blockchain ledger through the chaincode. Therefore, the chaincode needs to be installed on every peer that will endorse a transaction and instantiated on the channel.

State Database:

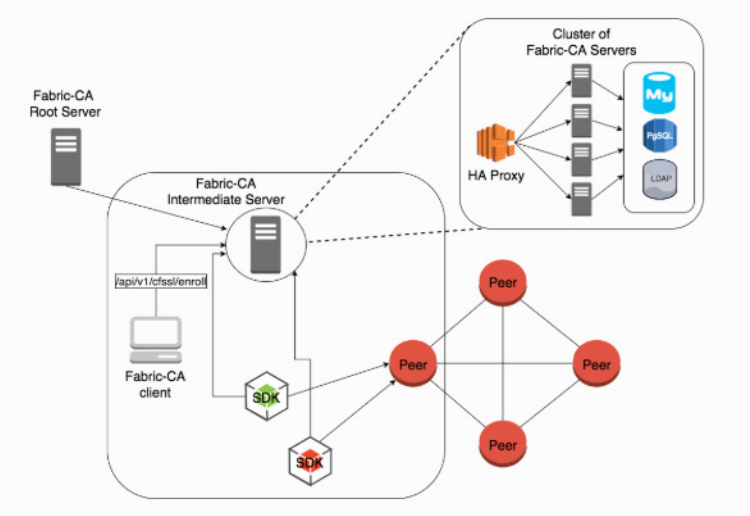
- The current state data represents the latest values for all assets in the ledger. Since the current state represents all the committed transactions on the channel, it is sometimes referred to as **world state**.
- Chaincode invocations execute transactions against the current state data. To make these chaincode interactions extremely efficient, the latest key/value pairs for each asset are stored in a state database.
- The state database is simply an indexed view into the chain's committed transactions. It can therefore be regenerated from the chain at any time.
- The state database will automatically get recovered (or generated, if needed) upon peer startup, before new transactions are accepted.
- The default state database, **LevelDB**, can be replaced with **CouchDB**.
- **LevelDB** is the default key/value state database for Hyperledger Fabric, and simply stores key/value pairs.
- **CouchDB** is an alternative to LevelDB. Unlike LevelDB, CouchDB stores JSON objects. CouchDB is unique in that it supports keyed, composite, key range, and full data-rich queries.



- Hyperledger Fabric's LevelDB and CouchDB are very similar in their structure and function.
- Both LevelDB and CouchDB support core chaincode operations, such as getting and setting key assets, and querying based on these keys.
- With both, keys can be queried by range, and composite keys can be modeled to enable equivalence queries against multiple parameters.
- But, as a JSON document store, CouchDB additionally enables rich query against the chaincode data, when chaincode values (e.g. assets) are modeled as JSON data.

Fabric Certificate Authority:

- *Certificate Authorities* manage enrollment certificates for a permissioned blockchain.
- **Fabric-CA** is the default certificate authority for Hyperledger Fabric, and handles the registration of user identities.
- The Fabric-CA certificate authority is in charge of issuing and revoking Enrollment Certificates (E-Certs).
- The current implementation of Fabric-CA only issues E-Certs, which supply long term identity certificates.
- E-Certs, which are issued by the Enrollment Certificate Authority (E-CA), assign peers their identity and give them permission to join the network and submit transactions.



Consensus in Hyperledger Fabric

The consensus in Hyperledger Fabric network is a process where the nodes in the network provide a guaranteed ordering of the transaction and validating those block of transactions that need to be committed to the ledger. Consensus must ensure the following in the network:

- Confirms the correctness of all transactions in a proposed block, according to endorsement and consensus policies.

- Agrees on order and correctness and hence on results of execution (implies agreement on global state).
- Interfaces and depends on smart-contract layer to verify correctness of an ordered set of transactions in a block.

Consensus Properties

Consensus must satisfy two properties to guarantee agreement among nodes: safety and liveness.

Safety means that each node is guaranteed the same sequence of inputs and results in the same output on each node. When the nodes receive an identical series of transactions, the same state changes will occur on each node. The algorithm must behave identical to a single node system that executes each transaction atomically one at a time.

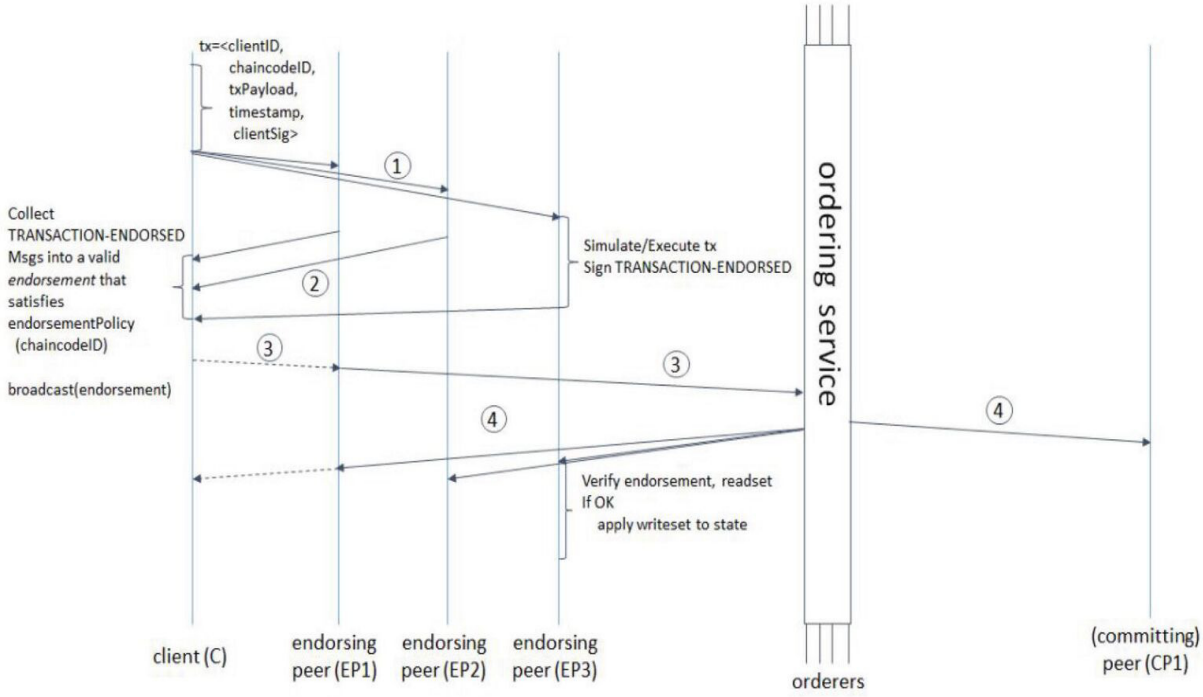
Liveness means that each non-faulty node will eventually receive every submitted transaction, assuming that communication does not fail.

The voting-based Consensus

There are two main types of Consensus. Hyperledger makes use of the permissioned voting-based consensus from the pool of other consensus named the lottery-based consensus. The operating assumption for Hyperledger developers is that business blockchain networks will operate in an environment of partial trust. Thus, the voting-based algorithms are advantageous in that they provide low-latency finality. When a majority of nodes validates a transaction or block, consensus exists and finality occurs. Because voting-based algorithms typically require nodes to transfer messages to each of the other nodes on the network, the more nodes that exist on the network, the more time it takes to reach consensus. This results in a trade-off between scalability and speed.

Consensus in Hyperledger Fabric is broken out into 3 phases: Endorsement, Ordering, and Validation.

- Endorsement is driven by policy (m out of n signatures) upon which participants endorse a transaction.
- Ordering phase will get the endorsed transaction and agrees to the orgder to be committed to the ledger.
- Validation takes a block of ordered transactions and validates the correctness of the result.



TRANSACTION FLOW IN HYPERLEDGER FABRIC

Multiple ordering plugins are being developed currently, including BFT Smart, Simplified Byzantine Fault Tolerance (SBFT), Honey Badger of BFT, etc. For Fabric v1, Apache Kafka is provided out-of-the-box as a reference implementation. The application use-cases and its fault tolerance model should determine which plugin to use.

Byzantine Fault Tolerance

In a blockchain network, every non-malicious entity has the same blockchain state. The implication for Hyperledger Fabric is that the orderer service should be jointly controlled by the network's members using a BFT algorithm that resists malicious activities by bad actors. It's insufficient for one organization to control the orderer, because that organization itself may not be trustworthy. After all, one of the motivations to use blockchain is so that organizations can cooperate while only partially trusting one another.

Kafka in Hyperledger Fabric Ordering Service

In Kafka, only the leader does the ordering and only the in-sync replicas can be voted as leader. This provides crash fault-tolerance and finality happens in a matter of seconds. While Kafka is crash fault tolerant, it is not Byzantine fault tolerant, which prevents the system from reaching agreement in the case of malicious or faulty nodes.

In addition to the multitude of endorsement, validity and versioning checks that take place, there are also ongoing identity verifications happening in all directions of the transaction flow. Access control lists are implemented on hierarchal layers of the network (ordering service down to channels), and payloads are repeatedly signed, verified and authenticated as a transaction proposal passes through the different architectural components. To conclude, consensus is not merely limited to the agreed upon order of a batch of

transactions, but rather, it is an overarching characterization that is achieved as a byproduct of the ongoing verifications that take place during a transaction's journey from proposal to commitment.

What is Chaincode?

Chaincode is a program, written in Go, node.js, or Java that implements a prescribed interface. Chaincode runs in a secured Docker container isolated from the endorsing peer process. Chaincode initializes and manages the ledger state through transactions submitted by applications.

A chaincode typically handles business logic agreed to by members of the network, so it similar to a "smart contract". A chaincode can be invoked to update or query the ledger in a proposal transaction. Given the appropriate permission, a chaincode may invoke another chaincode, either in the same channel or in different channels, to access its state. Note that, if the called chaincode is on a different channel from the calling chaincode, only read query is allowed. That is, the called chaincode on a different channel is only a Query, which does not participate in state validation checks in subsequent commit phase.

In the following sections, we will explore chaincode through the eyes of an application developer. We'll present a simple chaincode sample application and walk through the purpose of each method in the Chaincode Shim API.

Chaincode API

Every chaincode program must implement the Chaincode interface:

- Go
- node.js
- Java

whose methods are called in response to received transactions. In particular the Init method is called when a chaincode receives an instantiate or upgrade transaction so that the chaincode may perform any necessary initialization, including initialization of application state. The Invoke method is called in response to receiving an invoke transaction to process transaction proposals.

The other interface in the chaincode "shim" APIs is the ChaincodeStubInterface:

- Go
- node.js
- Java

which is used to access and modify the ledger, and to make invocations between chaincodes.

Chaincode access control

Chaincode can utilize the client (submitter) certificate for access control decisions by calling the GetCreator() function. Additionally the Go shim provides extension APIs that extract client identity from the submitter's certificate that can be used for access control decisions, whether that is based on client identity itself, or the org identity, or on a client identity attribute.

For example an asset that is represented as a key/value may include the client's identity as part of the value (for example as a JSON attribute indicating that asset owner), and only this client may be authorized to make updates to the key/value in the future. The client identity library extension APIs can be used within chaincode to retrieve this submitter information to make such access control decisions.

To add the client identity shim extension to your chaincode as a dependency, see Managing external dependencies for chaincode written in Go.

Chaincode encryption

In certain scenarios, it may be useful to encrypt values associated with a key in their entirety or simply in part. For example, if a person's social security number or address was being written to the ledger, then you likely would not want this data to appear in plaintext. Chaincode encryption is achieved by leveraging the entities extension which is a BCCSP wrapper with commodity factories and functions to perform cryptographic operations such as encryption and elliptic curve digital signatures. For example, to encrypt, the invoker of a chaincode passes in a cryptographic key via the transient field. The same key may then be used for subsequent query operations, allowing for proper decryption of the encrypted state values.

Hyperledger Fabric Node SDK and Client Application

Hyperledger Fabric is a container-based blockchain framework used for developing decentralized applications using plug-and-play components aimed at making it modular. It doesn't mean that we can't create network components (peers and orderers) natively. But this isn't how the software is distributed. However, chaincode currently supports containerized (sandbox) environments only.

Fabric Node SDK provides access to interact with chaincode apart from the containerized environment.

Connection-Profile

A connection profile allows SDK to connect with the network. connection profile describes entire network topology of peers, orderers and certificate Authorities and which peers are part of the channel and where

they're located. So, Whenever client submits transaction, it will then populate over all the peers using configuration defined in connection profile for the particular channel. You can only connect to the network with proper connection profile. A connection profile is normally created by an administrator who understands the network topology. Connection profiles play a key role in connecting clients to the network. Here is the structural explanation about connection profiles.

Main Modules in Hyperledger Fabric Node SDK

SDK consists of three major modules

1.fabric-network:

This module provides a high-level API for interacting with the deployed chaincode. Interaction includes submission of transactions and queries.

2.fabric-ca-client:

This module provides various certification operations like user registrations and enrollments, re-enrollments on the network.

3.fabric-client:

This module provides network-level operations like creating channels, joining peers to the channel, installing and Instantiation of chaincode, submitting transactions and querying chaincode, querying transaction details, block height, installed chaincodes and many more.

even though fabric-client is capable of doing many operations, fabric-network is recommended API for client applications to interact with smart contracts deployed to Fabric network. because the majority of key classes for client applications are only inheritable from fabric-network.

Key Classes and methods in fabric-network

fabric-network has three main classes

1. Gateway

Gateway Class in fabric-network module is used to connect and interact with running fabric networks. this class includes various methods. those are,

a. connect:

This method connects running fabric-network based on the peers and their IP addresses defined in connection profile using existed user or Admin identity.

b. disconnect:

This method disconnects from running fabric-network and cleans up the cache.

c. getClient:

This method returns current registered client details as an object.

d. getNetwork:

This method communicates with a specified channel in the network.

e. getContract method will have access to a particular chaincode deployed to channel on top of the network defined in the connection profile.

f. submitTransaction method will submit a specified chaincode method and args to the peers(endorsers).

g. **evaluateTransaction** is similar to GET method in HTTP requests, it can only read the ledger state and used for query methods in chaincode.

2. FileSystemWallet

This class defines the implementation of an Identity wallet that persists to the fabric file system. this class includes some common methods exists, import, delete.

a. exists:

This method checks whether provided identity exists in the file system or not.

b. import:

This method imports generated PKI and x509 certificates and keys into the filesystem wallet under the identity of participant.

c. delete:

This method deletes the identity of a particular user from the filesystem wallet.

3. X509WalletMixin

Basically, CA provides enrollment certificates in PKI format(public key infrastructure). This class is used for creating identity in X.509 credential model using PKI user Certificates. This class includes createIdentity() method for creating Identity. This X.509 certificate is used for signing transactions in the network.

Key methods in fabric-ca-client

fabric-ca-client has few common methods used for CA operations. these are register, enroll, re-enroll.

a. **register**:

This method is used for registering new Participants. when registration is successful, it returns user secret. This secret needs to be provided while enrolling.

b. enroll:

This method is used for enrolling registered Participants in the network. in order to enroll, the user must be registered first. if enrollment succeeded, this method will return PKI based Certificate and Private key of the user.

c. reenroll:

There could be cases when a certificate expires or gets compromised (so it has to be revoked). So this is when re-enrollment comes into the picture and you enroll the same identity again with the CA to get new certificates using this method.

Hyperledger tools are very popular for building blockchain and decentralized applications. In particular, Hyperledger Fabric and Hyperledger Composer are the most widely used tools. Hyperledger Fabric Architecture and Components for Blockchain Developers and Installing Hyperledger Fabric on AWS articles are great resources for learning about Hyperledger Fabric. Once you learn about Hyperledger Fabric, you can move on to explore Hyperledger Composer.

Hyperledger Composer is a set of collaboration tools for business owners and developers that make it easy to write chaincode for Hyperledger Fabric and **decentralized applications** (**DApps**). With Composer, you can quickly build POC and deploy chaincode to the blockchain in a short amount of time. Hyperledger Composer consists of the following toolsets:

- **A modeling language called CTO**: A domain modeling language that defines a business model, concept, and function for a business network definition

- **Playground**: Rapid configuration, deployment, and testing of a business network

- **Command-line interface (CLI) tools**: The client command-line tool is used to integrate business network with Hyperledger Fabric

Composer-CLI is the most important tool for Composer deployment; it contains all the essential command-line operations. Other very useful tools include **Composer REST server**, **generator Hyperledger Composer**, **Yeoman**, and **Playground**. Composer CLI provides many useful tools for developers.

Composer CLI can be used to perform multiple administrative, operational, and development tasks. Here is a summary of the CLI commands:

Command	**Description**	**Examples**
composer archive <subcommand>	Composer archive command.	Composer archive list.
composer card <subcommand>	Command for managing business network cards.	Composer card list.
composer generator <subcommand>	Composer generator command to convert a business network definition into code.	Composer generator docs.
composer identity <subcommand>	Composer identity command.	Composer identity issue.