19CS7PE05T Block Chain & Cryptocurrency (3-0-0) 3 Credits

Course Objective:
1: Understand how blockchain systems (mainly Bitcoin and Ethereum) work,
2: To securely interact with them,
3: Design, build, and deploy smart contracts and distributed applications,
4: Integrate ideas from blockchain technology into their own projects

Module-1: [8 Hrs.]
Basics: Distributed Database, Two General Problem, Byzantine General problem and Fault Tolerance, Hadoop Distributed File System, Distributed Hash Table, ASIC resistance, Turing Complete.Cryptography: Hash function, Digital Signature - ECDSA, Memory Hard Algorithm, ZeroKnowledge Proof.

Module-2: [8 Hrs.]
Introduction, Advantage over conventional distributed database, Blockchain Network, Mining Mechanism, Distributed Consensus, Merkle Patricia Tree, Gas Limit, Transactions and Fee, Anonymity, Reward, Chain Policy, Life of Blockchain application, Soft & Hard Fork, Private and Public blockchain

Module-3: [10 Hrs.]
Nakamoto consensus, Proof of Work, Proof of Stake, Proof of Burn, Difficulty Level, Sybil Attack, Energy utilization and alternate,

Module-4: [10 Hrs.]
Cryptocurrency: History, Distributed Ledger, Bitcoin protocols - Mining strategy and rewards, Ethereum -Construction, DAO, Smart Contract, GHOST, Vulnerability, Attacks, Sidechain, Namecoin Cryptocurrency Regulation: Stakeholders, Roots of Bit coin, Legal Aspects-Crypto currency Exchange, Black Market and Global Economy.
Applications: Internet of Things, Medical Record Management System, Domain Name Service

Module-5: [4 Hrs.]
Smart contract writing using Solidity, Working on Metamask, truffle, Ganache

Course Outcome:
1. Explain design principles of Bitcoin and Ethereum, Explain Nakamoto consensus, Explain the Simplified Payment Verification protocol.
2. List and describe differences between proof-of-work and proof-of-stake consensus.
3. Interact with a blockchain system by sending and reading transactions. Design, build, and deploy a distributed application.
4. Evaluate security, privacy, and efficiency of a given blockchain system

Textbooks:

1. Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder, Bitcoin, and Cryptocurrency Technologies: A Comprehensive Introduction, Princeton University Press (July 19, 2016).

Reference Books

1. Antonopoulos, Mastering Bitcoin: Unlocking Digital Cryptocurrencies
2. Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System
3. DR. Gavin Wood, “ETHEREUM: A Secure Decentralized Transaction Ledger,”Yellow paper.2014.
4. Nicola Atzei, Massimo Bartoletti, and TizianaCimoli, A survey of attacks on Ethereum smart contracts

Distributed Database System

A **distributed database** is a database in which data is stored across different physical locations. It may be stored in multiple computers located in the same physical location (e.g. a data centre); or maybe dispersed over a network of interconnected computers. Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components.

Distributed Database Features

Some general features of distributed databases are:

- **Location independency** - Data is physically stored at multiple sites and managed by an independent DDBMS.
- **Distributed query processing** - Distributed databases answer queries in a distributed environment that manages data at multiple sites.
- **Distributed transaction management** - Provides a consistent distributed database through commit protocols, distributed concurrency control techniques, and distributed recovery methods in case of many transactions and failures.
- **Seamless integration** - Databases in a collection usually represent a single logical database, and they are interconnected.
- **Network linking** - All databases in a collection are linked by a network and communicate with each other.
- **Transaction processing -** Distributed databases incorporate transaction processing, which is a program including a collection of one or more database operations. Transaction processing is an atomic process that is either entirely executed or not at all.

Types:

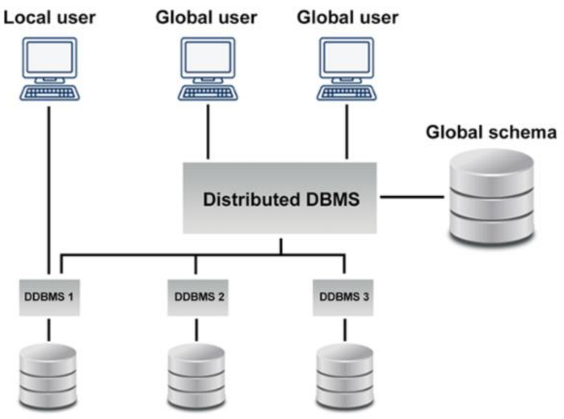
1. Homogeneous Database:

In a homogeneous database, all different sites store database identically. The operating system, database management system, and the data structures used – all are the same at all sites. Hence, they're easy to manage.

2. Heterogeneous Database:

In a heterogeneous distributed database, different sites can use different schema and software that can lead to problems in query processing and transactions. Also, a particular site might be completely unaware of the other sites. Different computers may use a different operating system, different database application. They may even use different data models for the database. Hence, translations are required for different sites to communicate.

Distributed Database Storage (How is Data Stored In Distributed Databases?)



Distributed database storage is managed in two ways:

- **Replication**
- Fragmentation

Replication

In database replication, the systems store **copies of data on different sites**. If an entire database is available on multiple sites, it is a fully redundant database.
The advantage of database replication is that it **increases data availability o**n different sites and allows for parallel query requests to be processed.
However, database replication means that data requires constant updates and synchronization with other sites to maintain an exact database copy. Any changes made on one site must be recorded on other sites, or else inconsistencies occur.
Constant updates cause a lot of server overhead and complicate concurrency control, as a lot of concurrent queries must be checked in all available sites.

Fragmentation

When it comes to fragmentation of distributed database storage, the relations are fragmented, which means they are **split into smaller parts**. Each of the fragments is stored on a different site, where it is required.
The prerequisite for fragmentation is to make sure that the fragments can later be reconstructed into the original relation without losing data.
The advantage of fragmentation is that there are **no data copies**, which prevents data inconsistency.
There are two types of fragmentation:

- **Horizontal fragmentation** - The relation schema is fragmented into groups of rows, and each group (tuple) is assigned to one fragment.
- **Vertical fragmentation** - The relation schema is fragmented into smaller schemas, and each fragment contains a common candidate key to guarantee a lossless join.

Applications of Distributed Database:

- It is used in Corporate Management Information System.
- It is used in multimedia applications.
- Used in Military's control system, Hotel chains etc.
- It is also used in manufacturing control system.

The main advantage of a distributed database system is that it can provide higher availability and reliability than a centralized database system. Because the data is stored across multiple sites, the system can continue to function even if one or more sites fail. In addition, a distributed database system can provide better performance by distributing the data and processing load across multiple sites.

There are several different architectures for distributed database systems, including:

Client-server architecture: In this architecture, clients connect to a central server, which manages the distributed database system. The server is responsible for coordinating transactions, managing data storage, and providing access control.
Peer-to-peer architecture: In this architecture, each site in the distributed database system is connected to all other sites. Each site is responsible for managing its own data and coordinating transactions with other sites.
Federated architecture: In this architecture, each site in the distributed database system maintains its own independent database, but the databases are integrated through a middleware layer that provides a common interface for accessing and querying the data.

Distributed database systems can be used in a variety of applications, including e-commerce, financial services, and telecommunications. However, designing and managing a distributed database system can be complex and requires careful consideration of factors such as data distribution, replication, and consistency.

Advantages of Distributed Database System :

1) There is fast data processing as several sites participate in request processing.
2) Reliability and availability of this system is high.
3) It possess reduced operating cost.
4) It is easier to expand the system by adding more sites.
5) It has improved sharing ability and local autonomy.

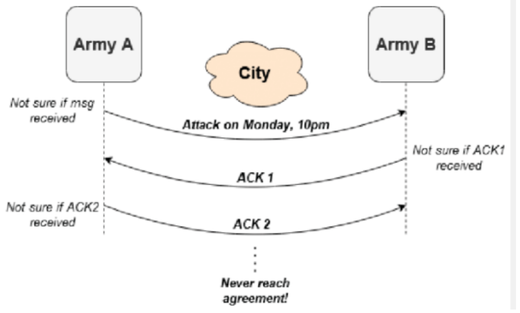
Disadvantages of Distributed Database System :

1) The system becomes complex to manage and control.
2) The security issues must be carefully managed.
3) The system require deadlock handling during the transaction processing otherwise
the entire system may be in inconsistent state.
4) There is need of some standardization for processing of distributed database
system.

The Two General's problem

The Two Generals' Problem, also known as the Two Generals' Paradox or the Two Armies Problem, is a classic computer science and computer communication thought experiment

... I HOPE MY MESSAGE WENT THROUGH...
... WHEN TO ATTACK?
SYNCHRONISED ATTACK IS THE ONLY CHANGE TO WIN

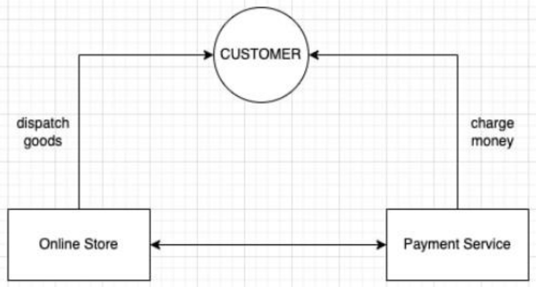


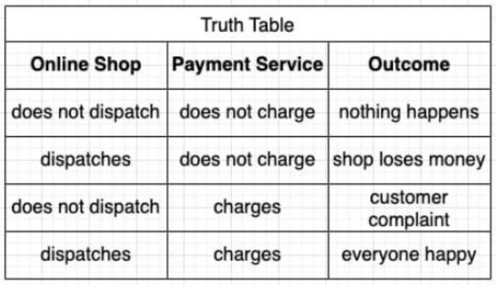
The situation is let's imagine two armies, led by two generals, planning an attack on a common enemy. The enemy's city is in a valley and has a strong defense that can easily fight off a single army. The two generals have to communicate with each other to plan a synchronized attack as this is their only chance to win. The only problem is that to communicate with each other they have to send a messenger across the enemy's territory. If a messenger is captured the message he's carrying is lost. Also, each general wants to know that the other general knows when to attack. Otherwise, a general wouldn't be sure if he's attacking alone and as we know attacking alone is rather pointless.

Now, let's go through a simple scenario. Let's call our generals A and B and let's assume everything goes perfectly fine. General A, who is the leader, sends a message – "Attack tomorrow at dawn". General B receives a message and sends back an acknowledgment – "I confirm, attack tomorrow at dawn". A receives B's confirmation. Is this enough to form a consensus between the generals? Unfortunately not, as General B still doesn't know if his confirmation was received by General A. Ok, so what if General A confirms General's B confirmation? Then, of course, that confirmation has to be also confirmed and we end up with an infinite exchange of confirmations.

In the second scenario, let's also assume that General A sends a message to General B. Some time has passed and General A starts wondering what happened to his message as there is no confirmation coming back from General B. There are two possibilities here. Either the messenger sent by General A has been captured and hasn't delivered a message or maybe B's messenger carrying B's confirmation has been captured. In both scenarios, they cannot come to a consensus again as A is not able to tell if his message was lost or if it was B's confirmation that didn't get through. Again, we ended up in an inconsistent state which would result in either General A or B attacking by himself.

We can quickly realise that no matter how many different scenarios we try and how many messages we send we cannot guarantee that consensus is reached and each general is certain that his ally will attack at the same time. To make it even worse, there is no solution to the Two Generals' Problem, so the problem remains unsolvable.

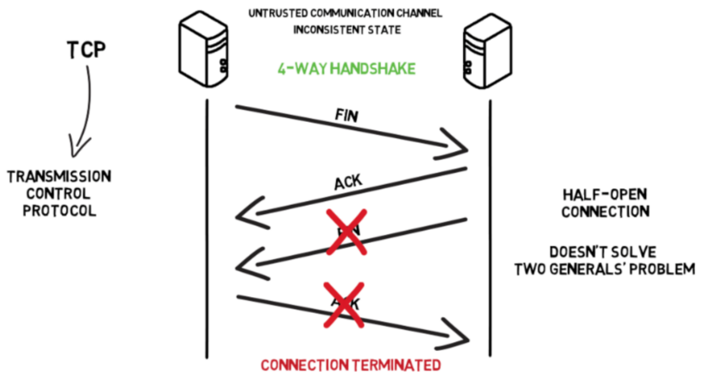




Truth Table		
Online Shop	**Payment Service**	**Outcome**
does not dispatch	does not charge	nothing happens
dispatches	does not charge	shop loses money
does not dispatch	charges	customer complaint
dispatches	charges	everyone happy

How is this related to computer science and TCP?

Instead of two generals, let's imagine two computer systems talking to each other. The main problem here is again the untrusted communication channel and inconsistent state between two machines. A very common example that always comes up when talking about the Two Generals' Problem is the TCP protocol.


TCP
TRANSMISSION CONTROL PROTOCOL
UNTRUSTED COMMUNICATION CHANNEL
INCONSISTENT STATE
4-WAY HANDSHAKE
FIN
ACK
FIN
ACK
HALF-OPEN CONNECTION
DOESN'T SOLVE TWO GENERALS' PROBLEM
CONNECTION TERMINATED

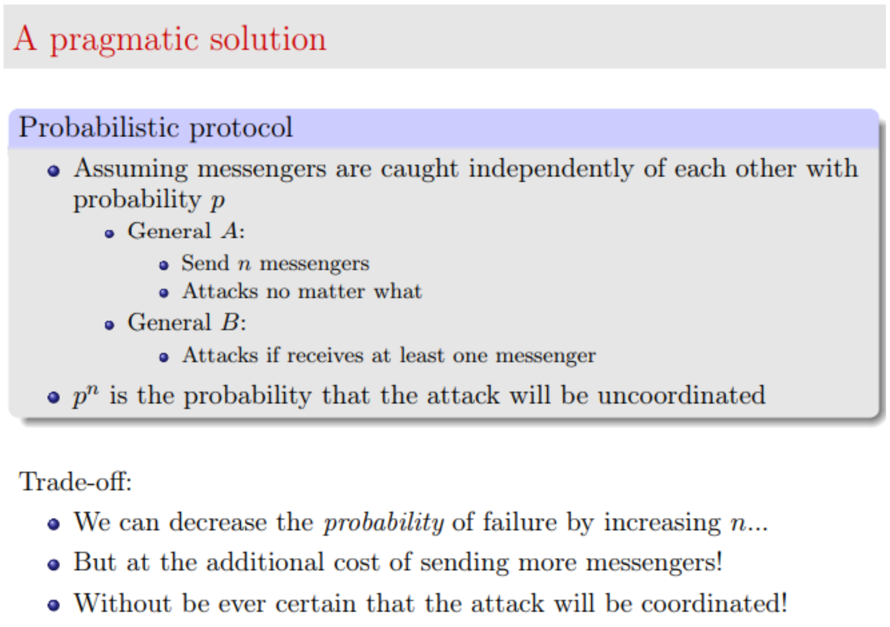
As we probably know, TCP uses a mechanism called 4-way handshake to terminate the connection. In this mechanism, a system that wants to terminate a connection sends a FIN message. The system on the other side of the communication channel replies with an ACK and sends its own FIN message which is followed by another ACK from the system which initialised termination. When all of those messages are received correctly, both sides know that the connection is terminated. So far it looks ok, but the problem here is again the shared knowledge between the two systems. When, for example, the second FIN is lost we end up with a half-open connection where one side is not aware that the connection has been closed. That's why even though TCP is very reliable protocol it doesn't solve the Two Generals' Problem

A Prgmatic way of solving the 2 general problem. The main assumption here is to accept the uncertainty of the communication channel and mitigate it to a sufficient degree.

Let's go back to our generals. What if General A instead of sending only 1 messenger sends 100 of them assuming that General B will receive at least 1 message. How about marking each message with a serial number starting from 1 up to 100. General B, based on the missing numbers in the sequence, would be able to gauge how reliable the communication channel is and reply with an appropriate number of confirmations. These approaches, even though, quite expensive are helping the generals to build up their confidence and come to a consensus.

If sacrificing messengers is a problem, we can come up with yet another approach where the absence of the messengers would build up generals' confidence. Let's assume that it takes 20 minutes to cross the valley, deliver a message and come back. General A starts sending messengers every 20 minutes until he gets a confirmation from General B. Whenever confirmation arrives General A stops sending messengers. In the meantime, General B after sending his messenger with his confirmation awaits for the other messengers coming from General A, but this time an absence of a messenger builds up General's B confidence as this is what the Generals agreed on.

In this case, we have a clear speed vs cost tradeoff and it's up to us which approach is more suitable to our problem



A pragmatic solution

Probabilistic protocol

- Assuming messengers are caught independently of each other with probability p
  - General A:
    - Send n messengers
    - Attacks no matter what
  - General B:
    - Attacks if receives at least one messenger
- p^n is the probability that the attack will be uncoordinated

Trade-off:

- We can decrease the *probability* of failure by increasing n...
- But at the additional cost of sending more messengers!
- Without be ever certain that the attack will be coordinated!

What is the Byzantine Generals' problem?

The Byzantine generals problem is a well-known concept in distributed computing and computer science that describes the difficulty of coordinating the actions of several independent parties in a distributed system.
The problem is particularly relevant in distributed systems such as blockchain, where multiple parties must reach a consensus on the system's state to maintain its integrity.

The Byzantine generals problem has several key parts that are important to understand:

- there are multiple independent parties (generals in the military metaphor) that must coordinate their actions
- the parties cannot rely on a central authority and must coordinate their actions in a decentralized environment
- we can only communicate with one another by sending messages
- we cannot ensure that the messages are authentic and, therefore, cannot fully trust one another
- the parties must reach a consensus despite the possibility of deception and betrayal from other parties

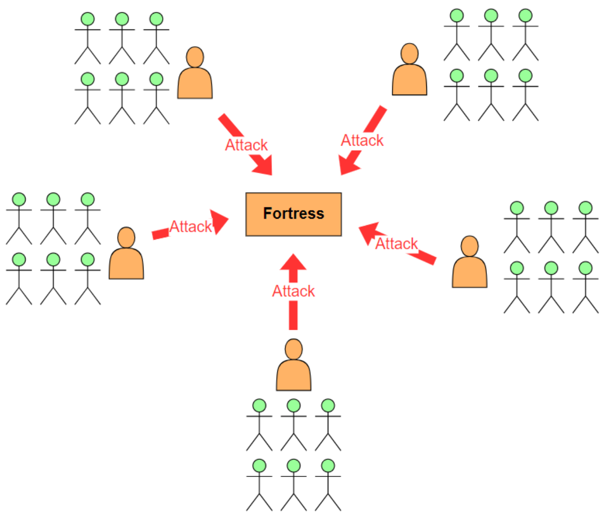
To summarize, the problem illustrates the difficulty of achieving consensus in a decentralized environment where parties cannot trust one another and must rely on communication to coordinate their actions.

A group of generals attacks a fortress; every general has an army and surrounds a fort from one side. Every general has a preference about whether to attack or retreat. It has to be a coordinated attack or retreat to incur minimum losses. Thus, a consensus is held, and the majority decision is implemented. This consensus is formed after following the following steps:

1. Every general sends their own choice to all other generals.
2. After receiving the choice of all generals, every general calculates the votes in favor of attacking and retreating.
3. If the majority is in favor of retreat, then they retreat; otherwise, they attack.

Suppose there is a traitor general who sends a retreat message to half generals and an attack message to the other half generals. Then half of the generals may end up attacking while the other half will retreat, causing the army to lose.

The following slides show all the scenarios of a successful attack, a successful retreat, and an unsuccessful attack.

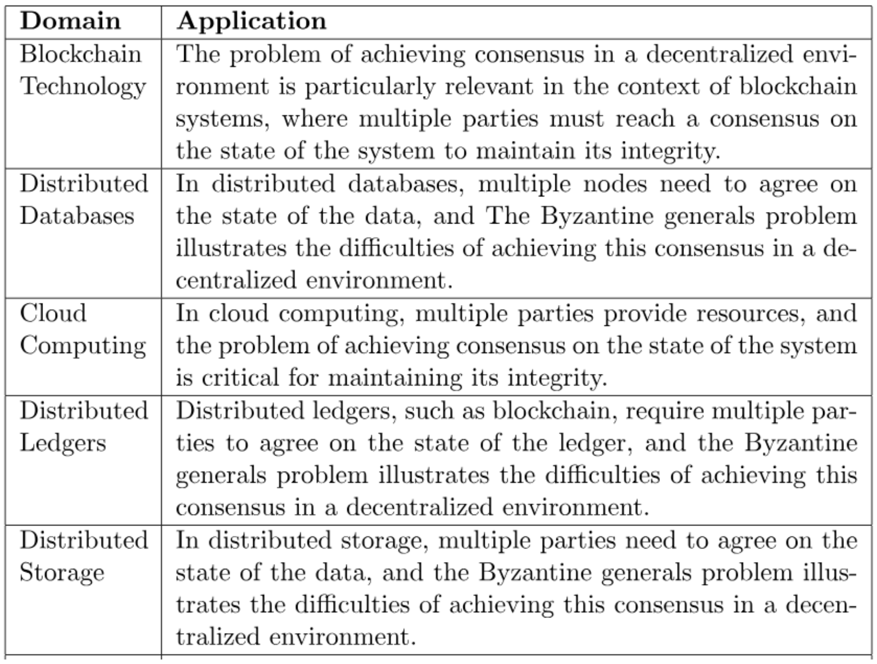


A successful attack

Real-Life Situations Where the Byzantine Generals Problem Is Applicable

One of the earliest solutions proposed to the Byzantine generals problem is the Byzantine fault tolerance (BFT) algorithm, which is based on the concept of a consensus protocol. BFT is a mechanism that allows a group of parties to reach a consensus despite faulty actors. A supermajority of parties must agree on a decision before implementing it.

Another solution is the practical Byzantine fault tolerance (PBFT) algorithm, a variation of BFT. PBFT is more efficient and scalable than BFT and widely used in blockchain systems such as Ethereum.



Domain	Application
Blockchain Technology	The problem of achieving consensus in a decentralized environment is particularly relevant in the context of blockchain systems, where multiple parties must reach a consensus on the state of the system to maintain its integrity.
Distributed Databases	In distributed databases, multiple nodes need to agree on the state of the data, and The Byzantine generals problem illustrates the difficulties of achieving this consensus in a decentralized environment.
Cloud Computing	In cloud computing, multiple parties provide resources, and the problem of achieving consensus on the state of the system is critical for maintaining its integrity.
Distributed Ledgers	Distributed ledgers, such as blockchain, require multiple parties to agree on the state of the ledger, and the Byzantine generals problem illustrates the difficulties of achieving this consensus in a decentralized environment.
Distributed Storage	In distributed storage, multiple parties need to agree on the state of the data, and the Byzantine generals problem illustrates the difficulties of achieving this consensus in a decentralized environment.

Byzantine Fault Tolerance (BFT)

Byzantine fault tolerance refers to the ability of a network or system to continue functioning even when some components are faulty or have failed.

Byzantine fault tolerance (BFT) is a decentralized permissionless system's ability to identify and reject false information. A decentralized, permissionless system is said to be Byzantine fault tolerant if it has solved the Byzantine Generals' Problem — a problem that Bitcoin finally solved in a distributed manner.

Byzantine fault tolerance refers to the ability of a network or system to continue functioning even when some components are faulty or have failed.

With a BFT system, blockchain networks keep functioning or implementing planned actions as long as most network participants are reliable and genuine. This means that over half or two-thirds of the nodes on the blockchain network must agree to validate a transaction and add it to the block.

For compromised nodes to cause malice on a Byzantine fault-tolerant blockchain, they must be in the majority. This malice can be in the form of double spending, a 51% attack, a Sybil attack, and so on.

Byzantine fault tolerance in blockchain technology originates from the Byzantine general problem pioneered by Leslie Lamport, Marshall Pease, and Robert Shostak.

Decentralized networks implement Byzantine fault tolerance via consensus rules or protocols. All the nodes in the network must adhere to these protocols or algorithms if they want to participate in validating and processing transactions.

For a transaction to be validated, processed, and added to a growing block, most nodes must agree that the transaction is authentic through the network's consensus algorithm. Bitcoin, Ethereum, and other proof of work (PoW) and proof of stake (PoS) blockchains employ BFT algorithms.

The generals would need an algorithm that could guarantee the following conditions.

1. All the loyal generals would act and agree on the same plan of action.
2. The loyal generals of the Byzantine army would not follow a bad plan under the influence of traitor generals.
3. The loyal generals would follow all the rules specified in the algorithm
4. All the loyal generals of the Byzantine army must reach a consensus irrespective of the actions of traitors.
5. Most important of all, the loyal generals should also reach an agreement on a specific and reasonable plan.

If this system is not Byzantine fault tolerant, a network member can feed false information to the system and compromise the network's reliability.

With a BFT system, blockchain networks keep functioning or implementing planned actions as long as most network participants are reliable and genuine.

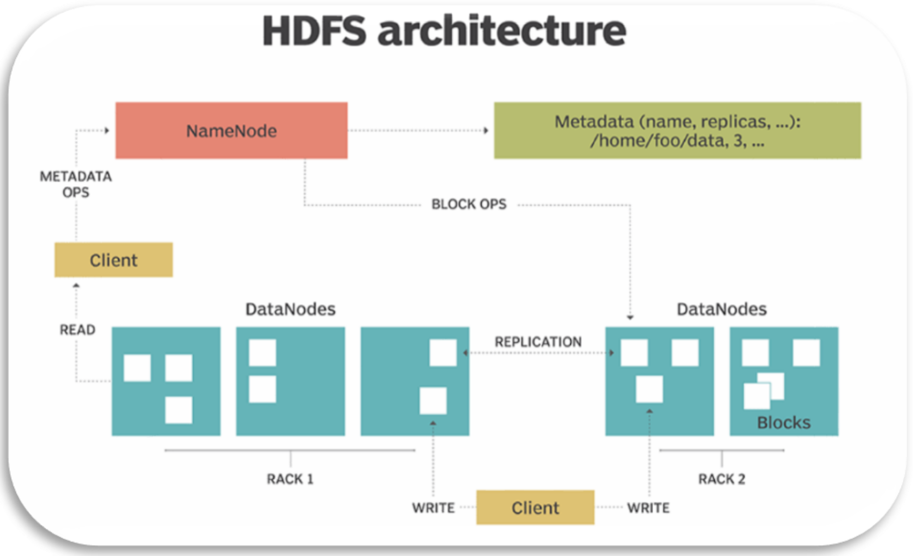
Importance of Byzantine fault tolerance

Byzantine fault tolerance is important because blockchain networks can operate normally even if some nodes broadcast false information or have stopped working. Crypto users should understand the Byzantine fault tolerance of the blockchains whose coins they have invested in because it tells them how secure their crypto transactions will be.

Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a distributed file system optimized to store large files and provides high throughput access to data.

The Hadoop Distributed File System (HDFS) is the primary data storage system used by Hadoop applications. HDFS employs a NameNode and DataNode architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters.



How does HDFS work?

HDFS enables the rapid transfer of data between compute nodes. Coupled with MapReduce, a framework for data processing that filters and divides up work among the nodes in a cluster, and it organizes and condenses the results into a cohesive answer to a query.

When HDFS takes in data, it breaks the information down into separate blocks and distributes them to different nodes in a cluster.

With HDFS, data is written on the server once, and read and reused numerous times after that. HDFS has a primary NameNode, which keeps track of where file data is kept in the cluster.

HDFS architecture, NameNodes and DataNodes

HDFS uses a primary/secondary architecture. The HDFS cluster's NameNode is the primary server that manages the file system namespace and controls client access to files.

NameNode(MasterNode):

The NameNode is the central point of contact for the HDFS. It manages the file system's metadata. The metadata, at a high level, is a list of all the files in the file system, the mapping from each file to the list of blocks the file has. This metadata is persisted on disk. As in other file systems, one of the significant attributes of the metadata that is built at runtime is the mapping from the file blocks to the physical locations of the blocks. The NameNode also controls the read/write accesses to the files from clients. The NameNode keeps track of the nodes in the cluster, the disk space the nodes have, and whether any node is dead. This information is used to schedule block replications for newly created files, and also to maintain a sufficient number of replicas of existing files.

DataNode(SlaveNode):

The DataNodes are the slaves in the HDFS cluster. When a client makes a request to create a file and write data to it, the NameNode assigns certain DataNodes to write the data to.

If the replica of the file under construction is, for example, 3, a write pipeline would be set up between the three DataNodes. The blocks would be written to the first DataNode in the pipeline, and that DataNode would write the blocks to the next DataNode in the pipeline, and so on until the last DataNode. A write is considered successful when all the replicas have been successfully written. This guarantees data consistency. The DataNodes also serve up blocks when clients request them to do so. They remain in touch with the NameNode, periodically send disk utilization reports, and periodically send block reports. The block reports are used by the NameNode to map the blocks of a file to its locations.

The HDFS Client: The client talks to the NameNode first for any file access. For file creations, the NameNode updates its metadata for the newly created file. It also chooses DataNodes that the client should write the file blocks to. For file open requests, the NameNode responds back with the set of locations that the client could read the data blocks from. If there are multiple DataNodes from where a given block could be read, the client chooses the one that is closest to it. This reduces the amount of data sent over the network.

HDFS utilizes rack-awareness and replication for high availability.

HDFS daemons: Daemons are the processes running in background.

Moreover, the HDFS is designed to be highly fault-tolerant. The file system replicates -- or copies -- each piece of data multiple times and distributes the copies to individual nodes, placing at least one copy on a different server rack than the other copies.

HDFS exposes a file system namespace and enables user data to be stored in files. A file is split into one or more of the blocks that are stored in a set of DataNodes. The NameNode performs file system namespace operations, including opening, closing and renaming files and directories. The NameNode also governs the mapping of blocks to the DataNodes. The DataNodes serve read and write requests from the clients of the file system, block creation, deletion and replication when the NameNode instructs them to do so.

HDFS supports a traditional hierarchical file organization. An application or user can create directories and then store files inside these directories. The file system namespace hierarchy is like most other file systems -- a user can create, remove, rename or move files from one directory to another.

The NameNode records any change to the file system namespace or its properties. An application can stipulate the number of replicas of a file that the HDFS should maintain. The NameNode stores the number of copies of a file, called the replication factor of that file.

- Dividing the file into multiple blocks done because easy to perform various read and write operations on it compared to doing it on a whole file at once. Hence faster data access i.e. reduce seek time.
- Replicate the blocks in data nodes while storing because to achieve fault-tolerance.

HDFS Read Image:

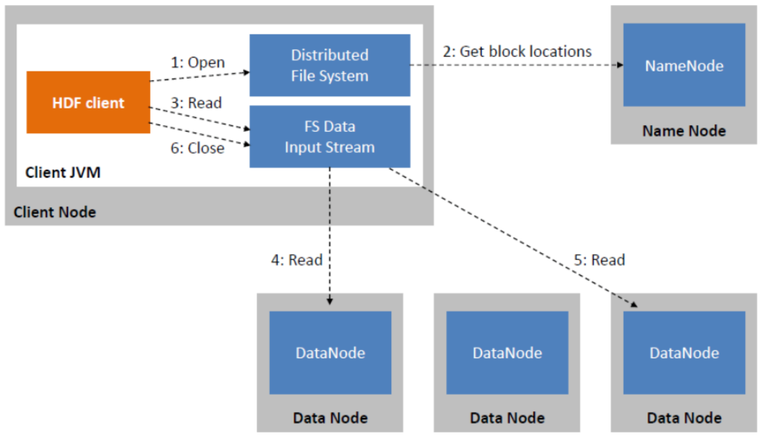


Figure: HDFS Read

HDFS Write Image:

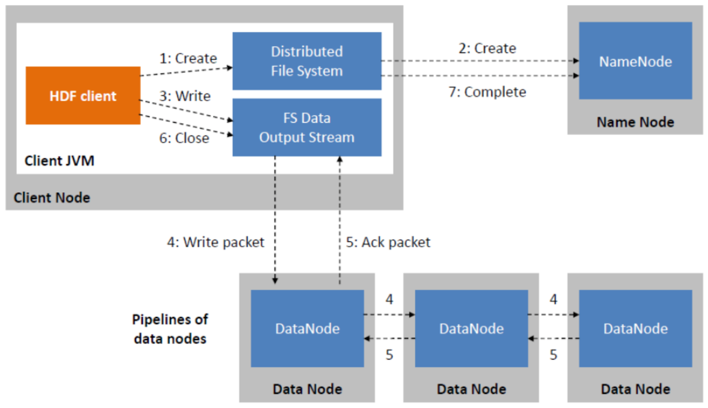


Figure: HDFS Write

Features of HDFS

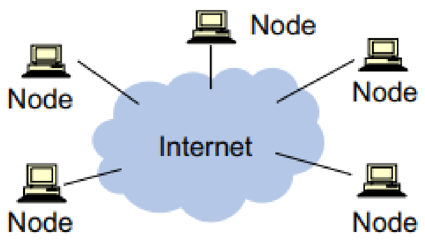
- Data replication.
- Fault tolerance and reliability. HDFS' ability to replicate file blocks and store them across nodes in a large cluster ensures fault tolerance and reliability.
- High availability.
- Scalability. Because HDFS stores data on various nodes in the cluster, as requirements increase, a cluster can scale to hundreds of nodes.
- High throughput. Because HDFS stores data in a distributed manner, the data can be processed in parallel on a cluster of nodes.
- Data locality. With HDFS, computation happens on the DataNodes where the data resides, rather than having the data move to where the computational unit is.

What are the benefits of using HDFS?

There are five main advantages to using HDFS, including:

- Cost effectiveness.
- Large data set storage.
- Fast recovery from hardware failure.
- Portability.
- Streaming data access.

What is a Peer-to-Peer (P2P) system?

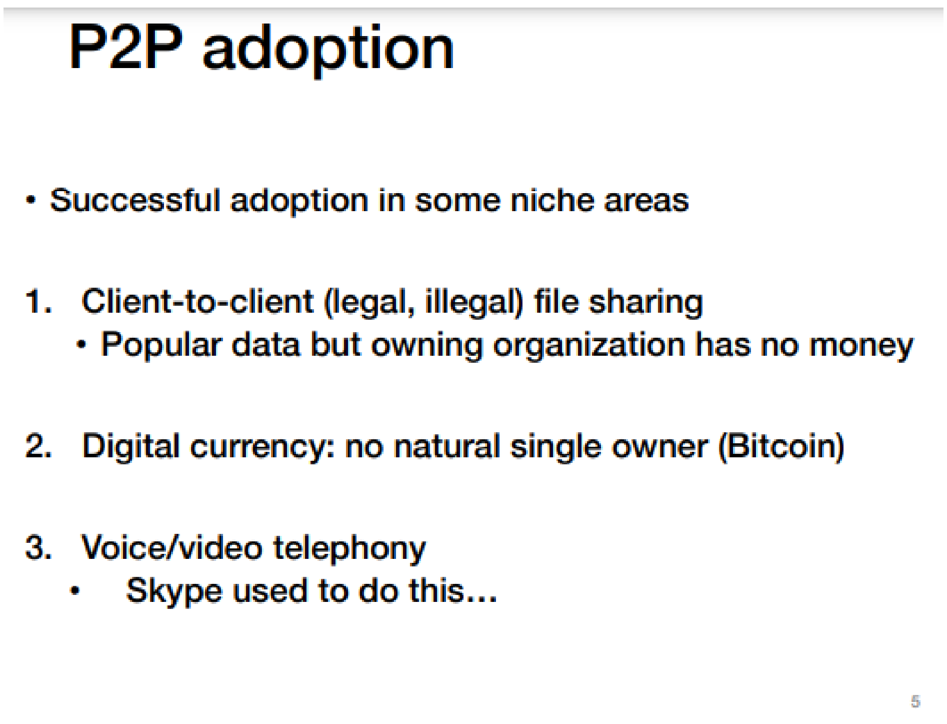


- A distributed system architecture:
  - No centralized control
  - Nodes are roughly symmetric in function
- Large number of unreliable nodes

3

Advantages of P2P systems

- High capacity for services through parallelism:
  - Many disks
  - Many network connections
  - Many CPUs
- No centralized server or servers may mean:
  - Less chance of service overload as load increases
  - A single failure won't wreck the whole system
  - System as a whole is harder to attack

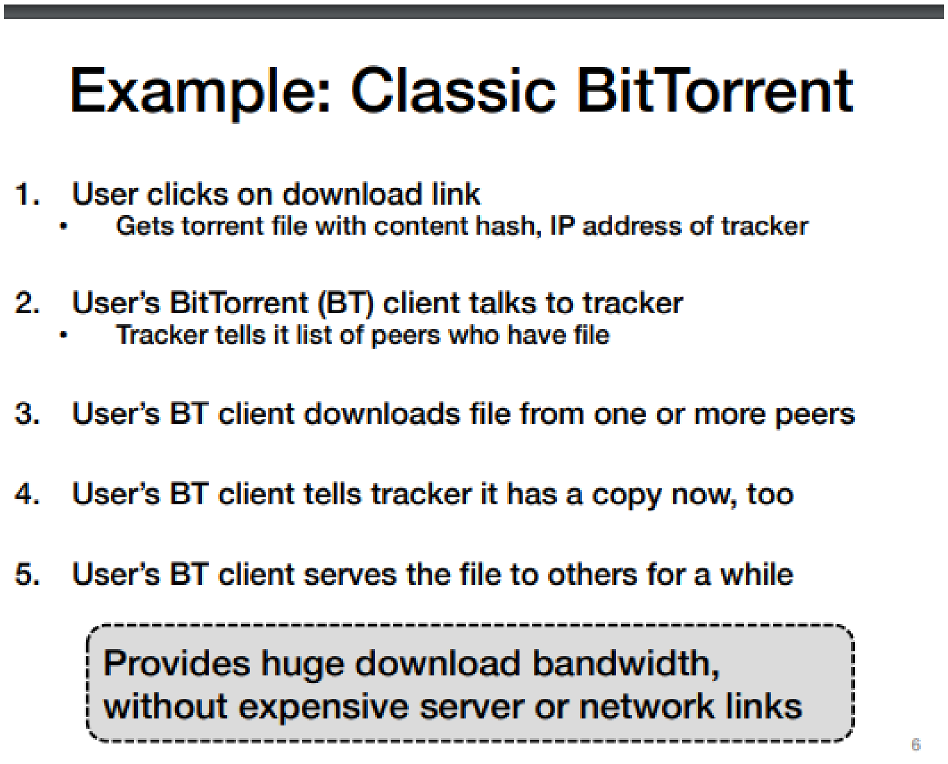


P2P adoption

- Successful adoption in some niche areas

1. Client-to-client (legal, illegal) file sharing
   - Popular data but owning organization has no money
2. Digital currency: no natural single owner (Bitcoin)
3. Voice/video telephony
   - Skype used to do this…

5



Example: Classic BitTorrent

1. User clicks on download link
   - Gets torrent file with content hash, IP address of tracker
2. User's BitTorrent (BT) client talks to tracker
   - Tracker tells it list of peers who have file
3. User's BT client downloads file from one or more peers
4. User's BT client tells tracker it has a copy now, too
5. User's BT client serves the file to others for a while

Provides huge download bandwidth, without expensive server or network links

6

Distributed Hash Tables (DHTs)

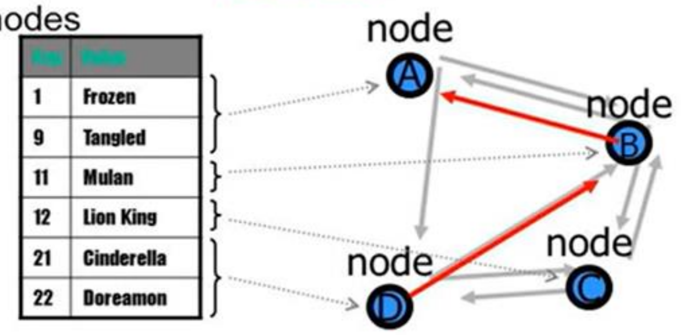
A distributed hash table (DHT) is a decentralized storage system that provides lookup and storage schemes similar to a hash table, storing key-value pairs.

Each node in a DHT is responsible for keys along with the mapped values. Any node can efficiently retrieve the value associated with a given key.

Just like in hash tables, values mapped against keys in a DHT can be any arbitrary form of data.

- DHTs are P2P algorithms (not protocol or application)– Chord, Pastry and Tapestry are specific DHT algorithms

A dynamic distribution of a *hash table* onto a set of cooperating nodes



[illegible]	[illegible]
1	Frozen
9	Tangled
11	Mulan
12	Lion King
21	Cinderella
22	Doreamon

DHTs have the following properties:

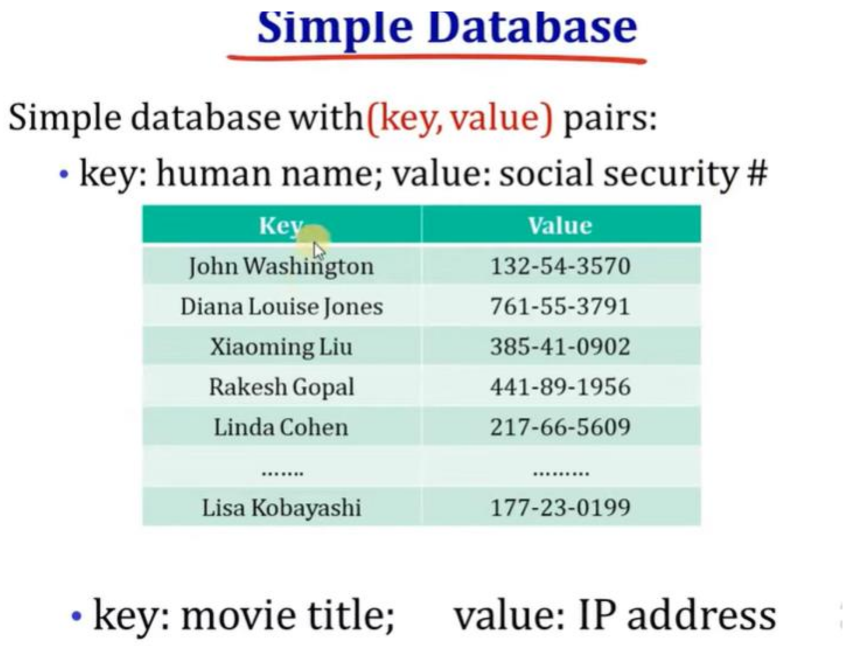
Decentralised & Autonomous: Nodes collectively form the system without any central authority.
Fault Tolerant: System is reliable with lots of nodes joining, leaving, and failing at all times.
Scalable: System should function efficiently with even thousands or millions of nodes.
Just like hash tables, DHTs support the following 2 functions:

Basic operations
- Local hash table:

key = Hash(name)
– Store(key, val)
– val = Retrieve(key)

Distributed Hash Table (DHT): Do (roughly) this across millions of hosts on the Internet!

The nodes in a DHT are connected together through an overlay network in which neighboring nodes are connected. This network allows the nodes to find any given key in the key-space.



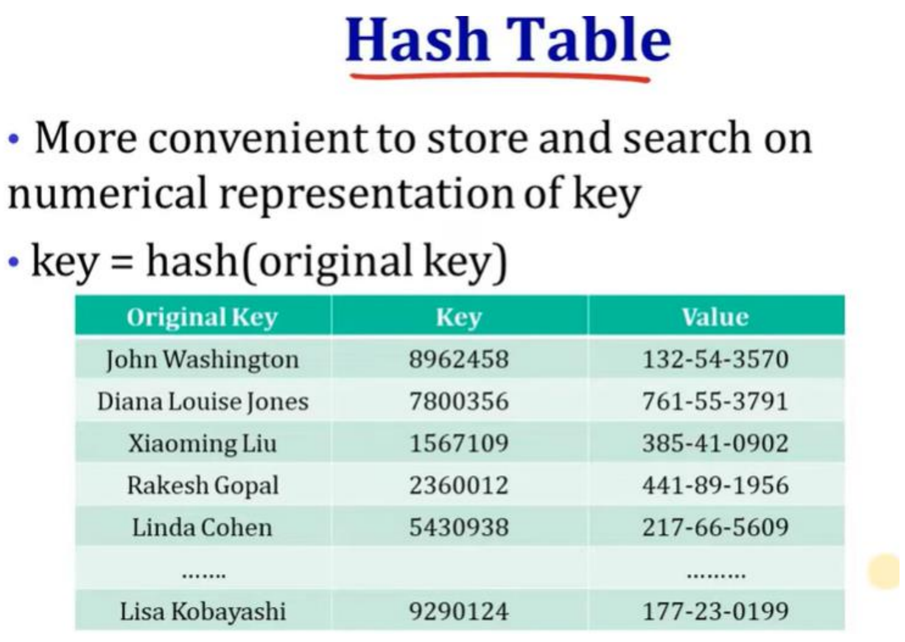
Simple Database

Simple database with (key, value) pairs:

- key: human name; value: social security #

Key	Value
John Washington	132-54-3570
Diana Louise Jones	761-55-3791
Xiaoming Liu	385-41-0902
Rakesh Gopal	441-89-1956
Linda Cohen	217-66-5609
…….	………
Lisa Kobayashi	177-23-0199

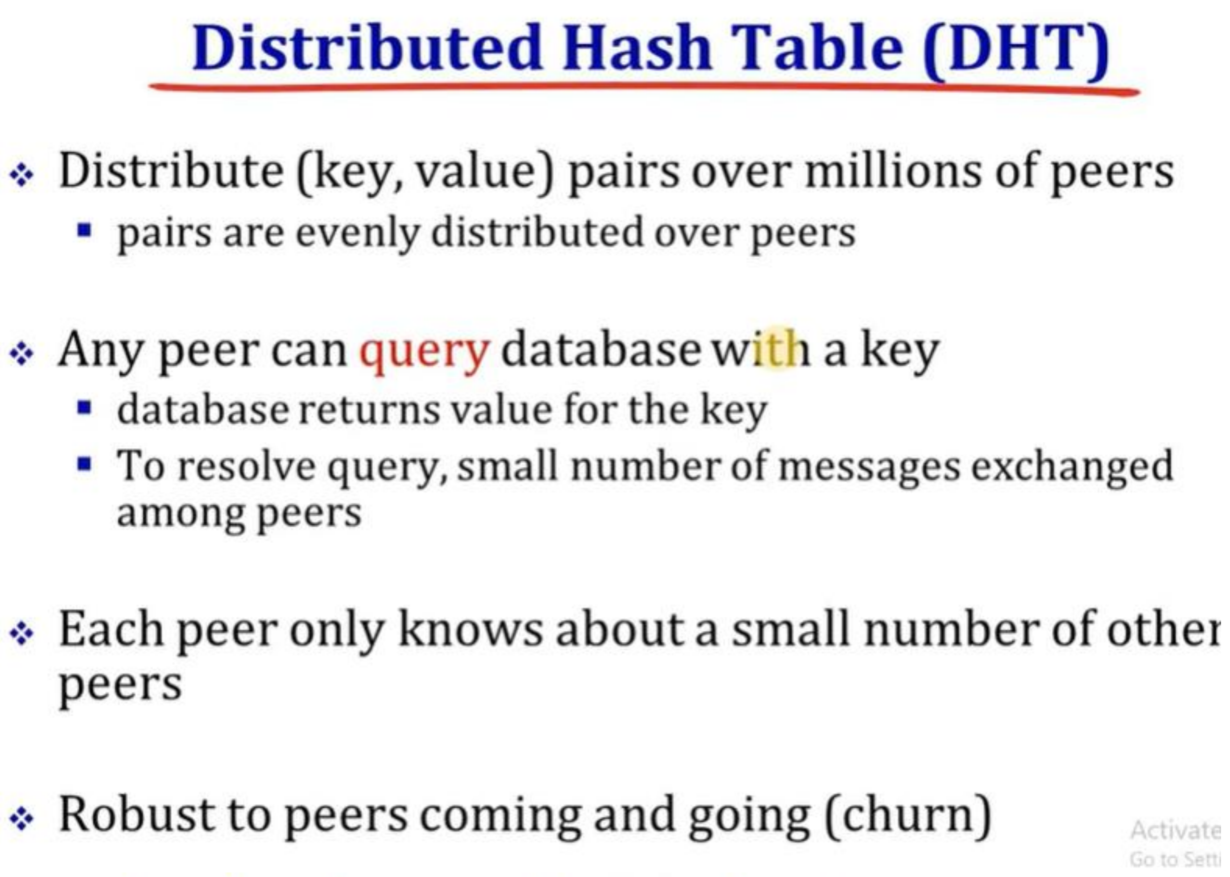
- key: movie title; value: IP address



Hash Table

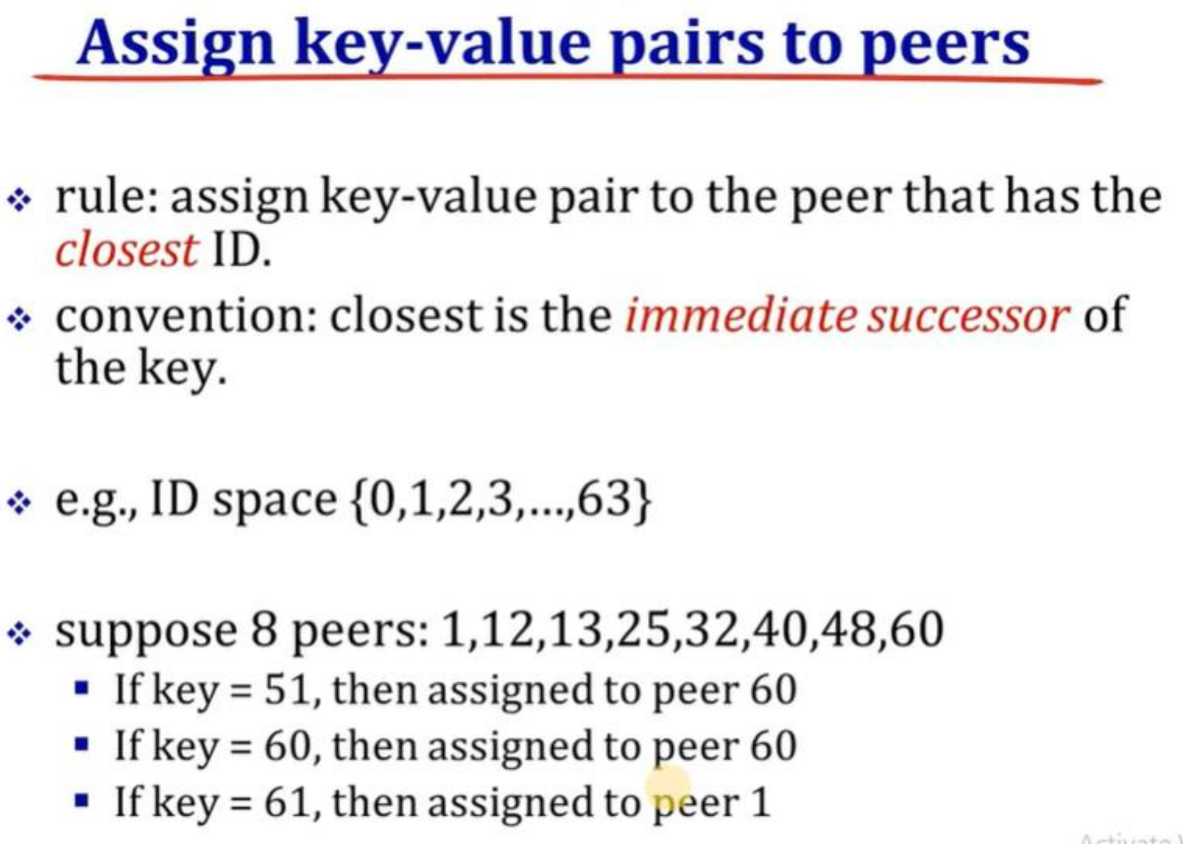
- More convenient to store and search on numerical representation of key
- key = hash(original key)

Original Key	Key	Value
John Washington	8962458	132-54-3570
Diana Louise Jones	7800356	761-55-3791
Xiaoming Liu	1567109	385-41-0902
Rakesh Gopal	2360012	441-89-1956
Linda Cohen	5430938	217-66-5609
…….		………
Lisa Kobayashi	9290124	177-23-0199



Distributed Hash Table (DHT)

- Distribute (key, value) pairs over millions of peers
  - pairs are evenly distributed over peers
- Any peer can query database with a key
  - database returns value for the key
  - To resolve query, small number of messages exchanged among peers
- Each peer only knows about a small number of other peers
- Robust to peers coming and going (churn)

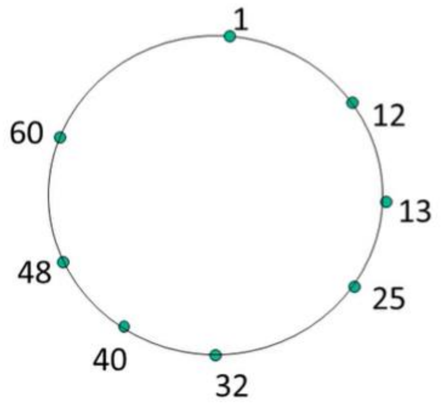


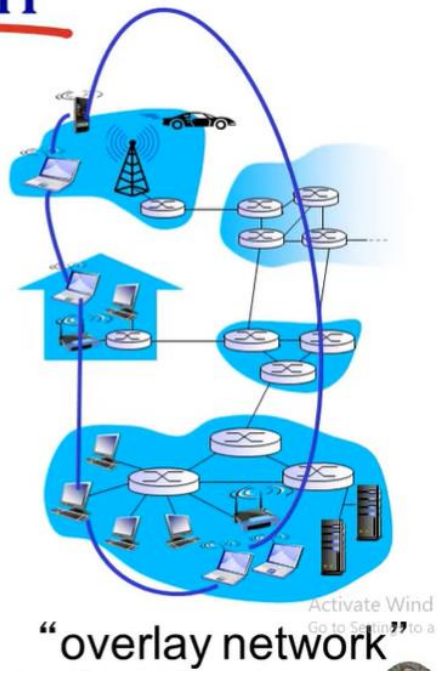
Assign key-value pairs to peers

- rule: assign key-value pair to the peer that has the *closest* ID.
- convention: closest is the *immediate successor* of the key.
- e.g., ID space {0,1,2,3,…,63}
- suppose 8 peers: 1,12,13,25,32,40,48,60
  - If key = 51, then assigned to peer 60
  - If key = 60, then assigned to peer 60
  - If key = 61, then assigned to peer 1

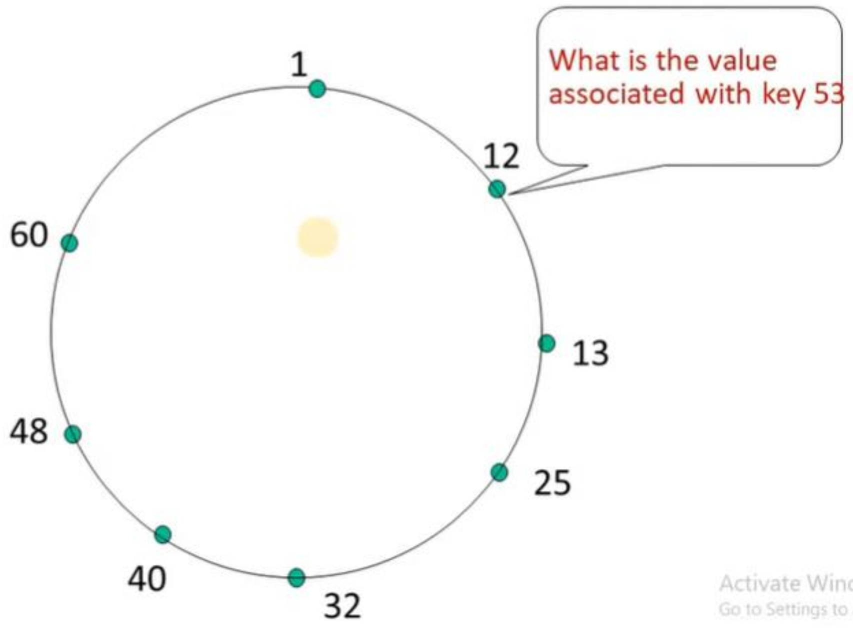
Circular DHT

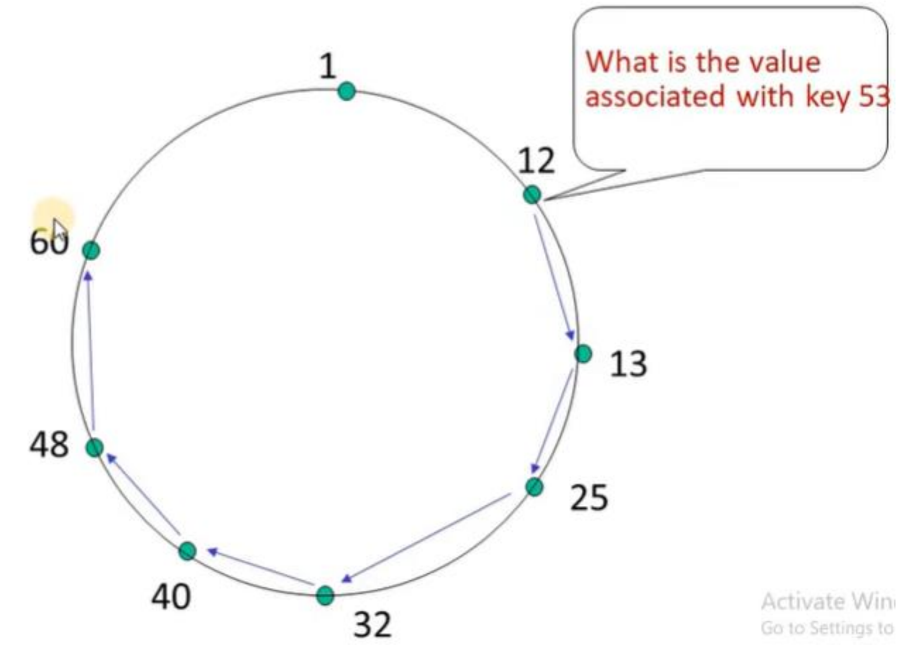
- each peer *only* aware of immediate successor and predecessor.

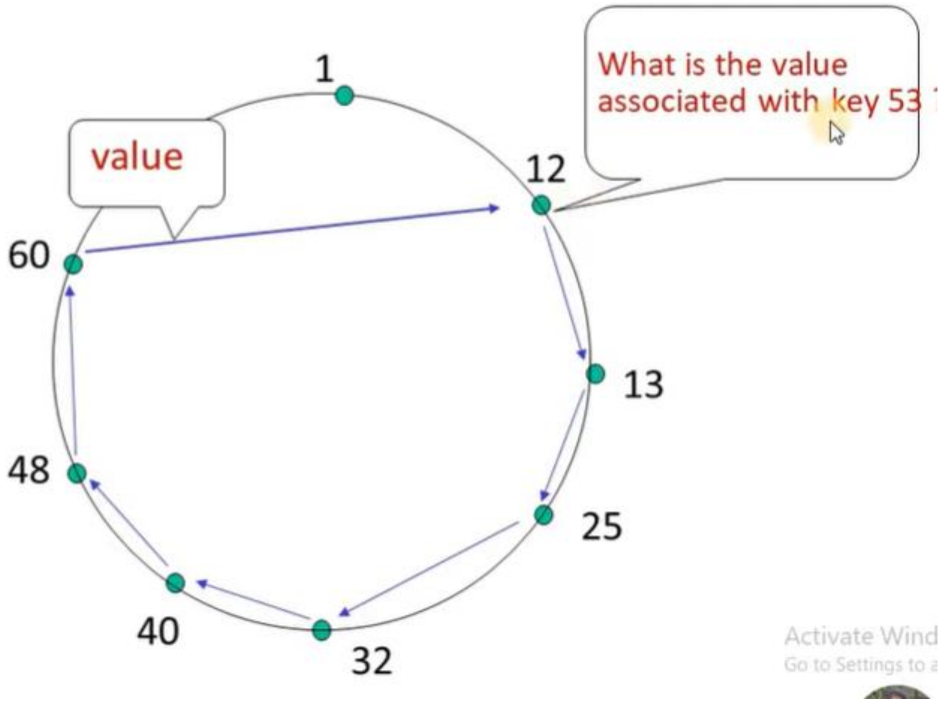




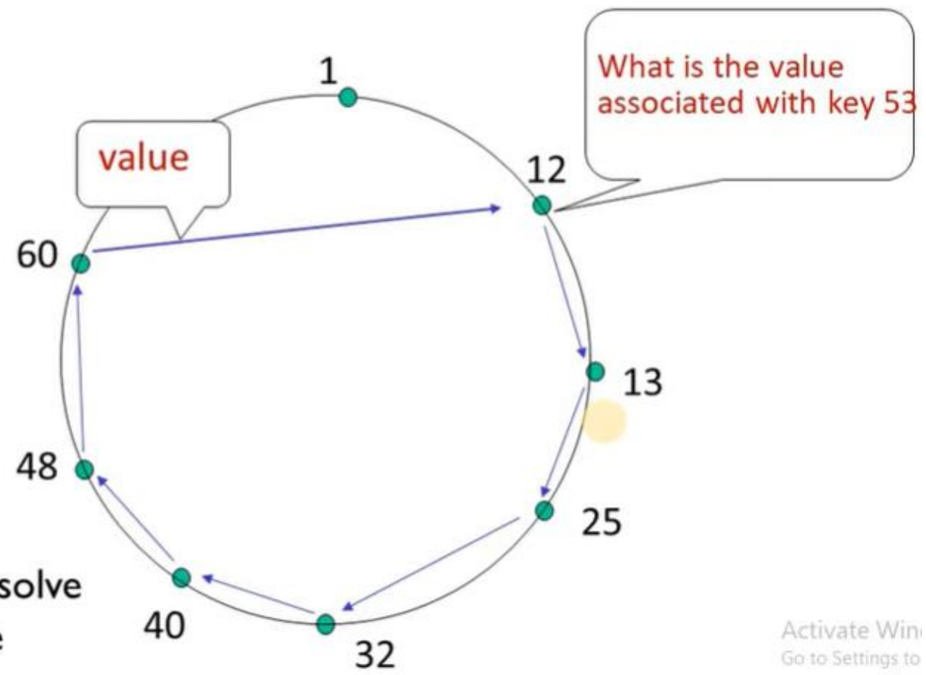
Resolving Query




1
12
13
25
32
40
48
60
What is the value associated with key 53
Activate Win
Go to Settings to

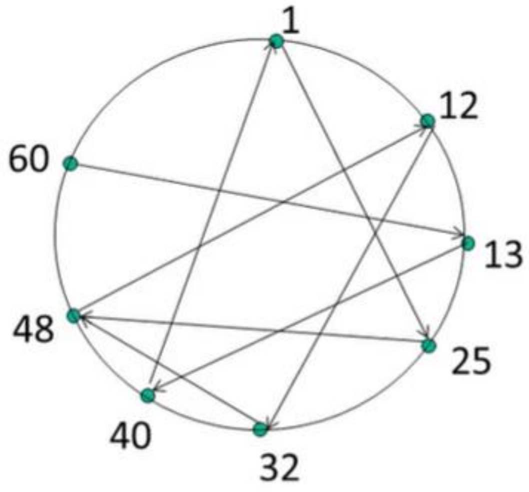

1
12
13
25
32
40
48
60
value
What is the value associated with key 53
Activate Wind
Go to Settings to

Resolving Query



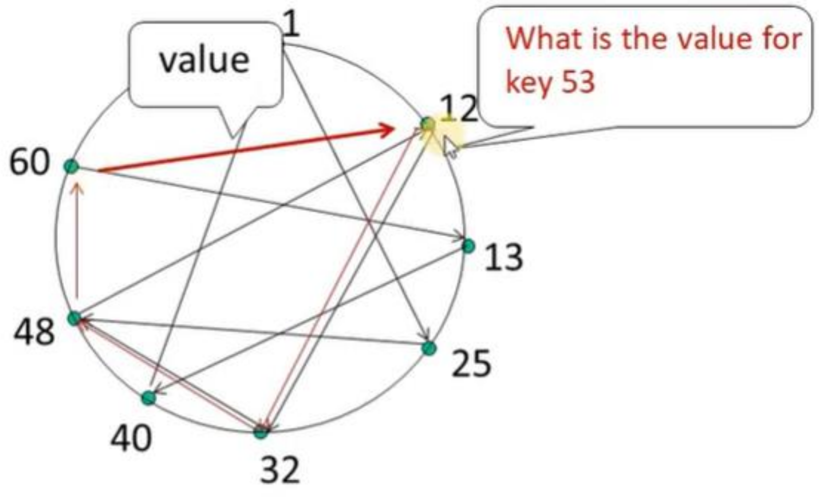
O(N) messages on avgerage to resolve query, when there are *N* peers

Circular DHT with shortcuts



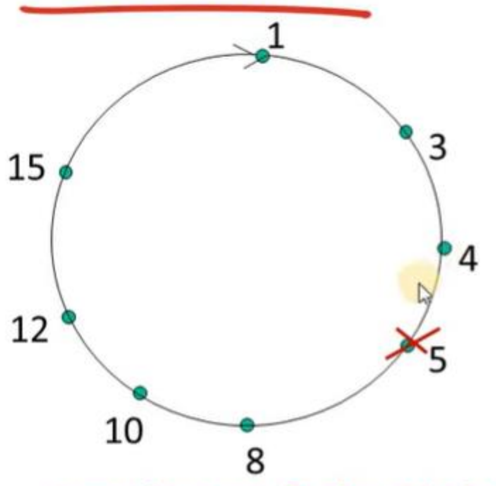
- each peer keeps track of IP addresses of predecessor, successor, short cuts.
- reduced from 6 to 3 messages.
- possible to design shortcuts with *O(log N)* neighbors, *O(log N)* messages in query

Circular DHT with shortcuts



- each peer keeps track of IP addresses of predecessor, successor, short cuts.
- reduced from 6 to 3 messages.
- possible to design shortcuts with *O(log N)* neighbors, *O(log N)* messages in query

Peer churn



example: peer 5 abruptly leaves

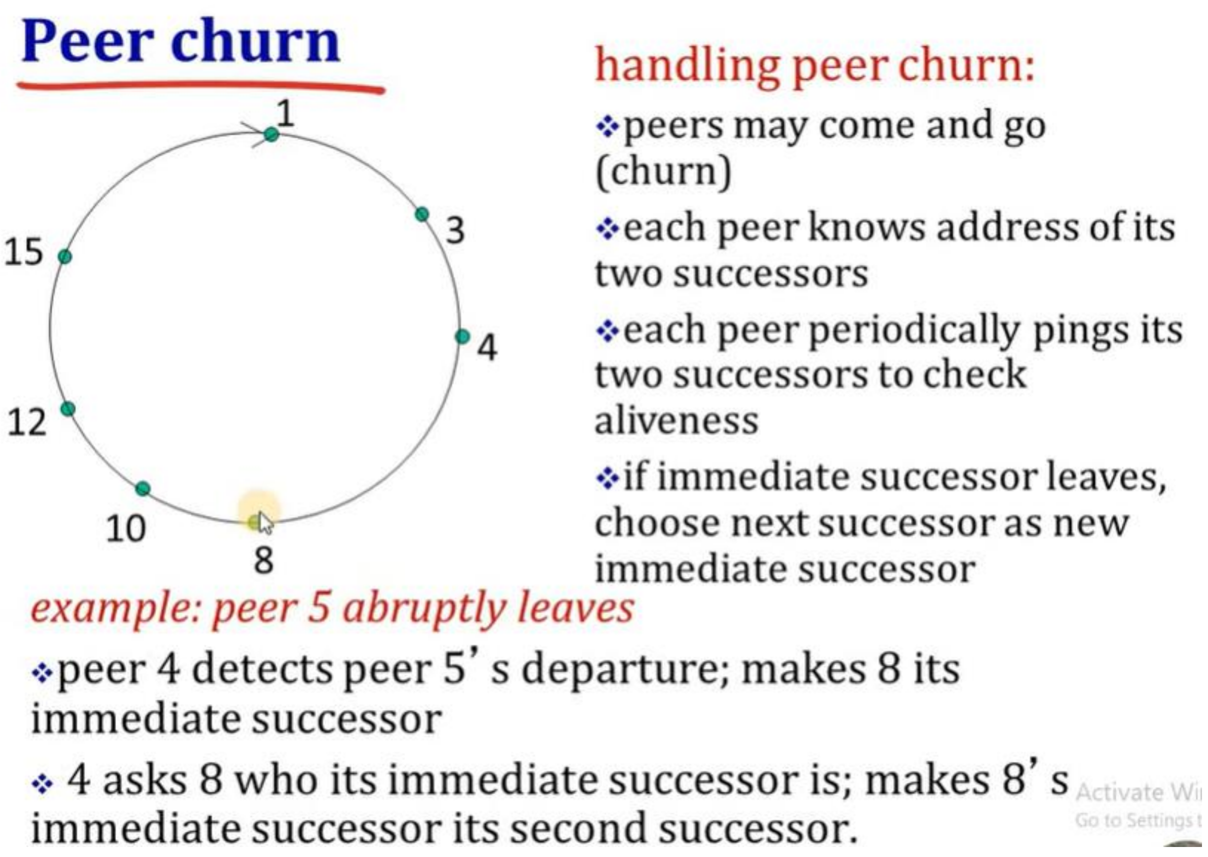
handling peer churn:

❖peers may come and go (churn)

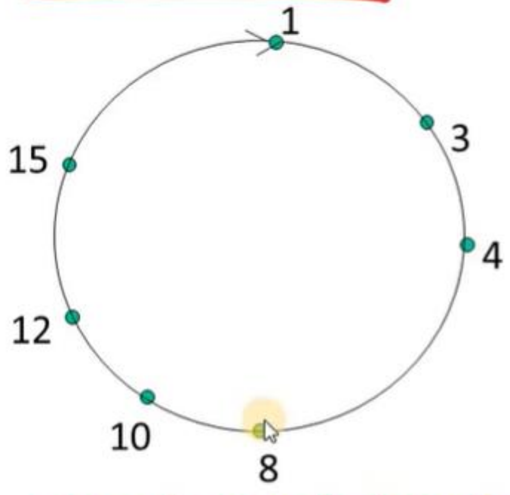
❖each peer knows address of its two successors

❖each peer periodically pings its two successors to check aliveness

❖if immediate successor leaves, choose next successor as new immediate successor



Peer churn



handling peer churn:

- peers may come and go (churn)
- each peer knows address of its two successors
- each peer periodically pings its two successors to check aliveness
- if immediate successor leaves, choose next successor as new immediate successor

example: peer 5 abruptly leaves

- peer 4 detects peer 5’s departure; makes 8 its immediate successor
- 4 asks 8 who its immediate successor is; makes 8’s immediate successor its second successor.

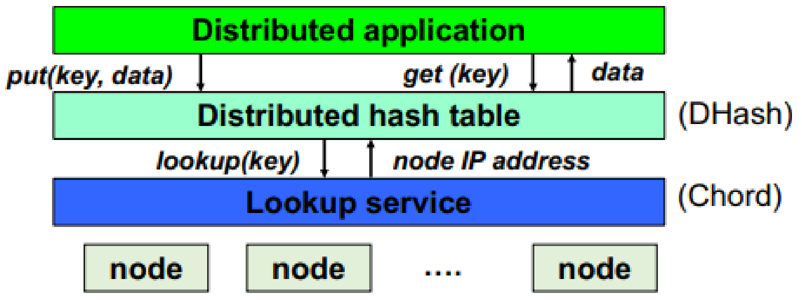
What is a DHT (and why)?

- Distributed Hash Table:

```
key = hash(data)
lookup(key) → IP addr (Chord lookup service)
send-RPC(IP address, put, key, data)
send-RPC(IP address, get, key) → data
```

- Partitions data in a large-scale distributed system
  - Tuples in a global database engine
  - Data blocks in a global file system
  - Files in a P2P file-sharing system

Cooperative storage with a DHT



- App may be distributed over many nodes
- DHT distributes data storage over many nodes

Chord lookup algorithm

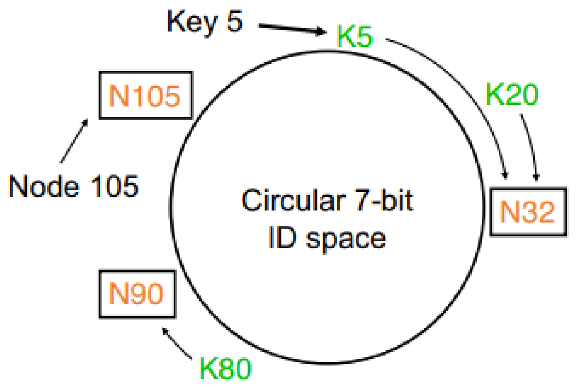
Interface: lookup(key) → IP address

- **Efficient:** O(log *N*) messages per lookup
  - *N* is the total number of servers
- **Scalable:** O(log *N*) state per node
- **Robust:** survives massive failures

Chord Lookup: Identifiers

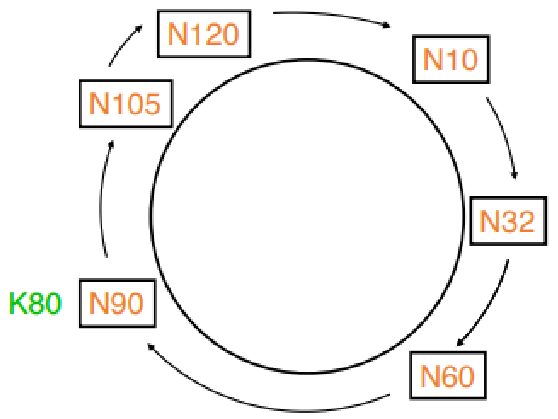
- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- SHA-1 distributes both uniformly
- How does Chord partition data?
  - i.e., map key IDs to node IDs

Consistent hashing

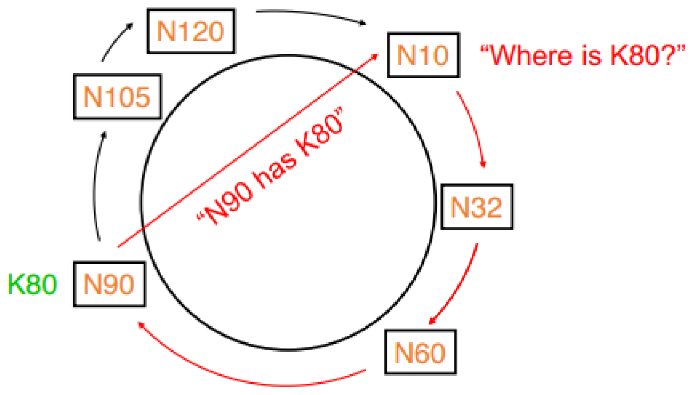


Key is stored at its successor: node with next-higher ID

Chord: Successor pointers



Basic lookup



24

Simple lookup algorithm

```
Lookup(key-id)
  succ ← my successor
  if my-id < succ < key-id   // next hop
      call Lookup(key-id) on succ
  else                       // done
    return succ
```

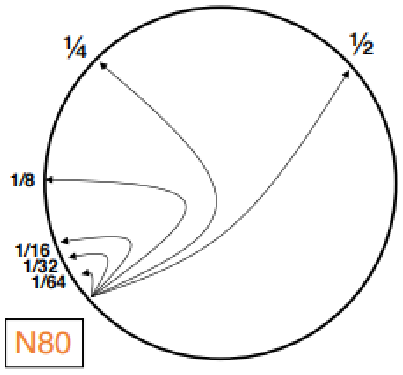
- Correctness depends only on successors

Improving performance

- **Problem:** Forwarding through successor is slow
- Data structure is a linked list: O(n)
- Idea: Can we make it more like a binary search?
  - Need to be able to halve distance at each step

26

"Finger table" for O(log *N*)-time lookups



ASIC-resistant

What is ASIC Resistant?
A coin is considered ASIC-resistant if it can be mined without using ASIC miners (Application-specific Integrated Circuits).

ASICs are integrated circuits designed to execute a specific computing task. Mining farms use ASIC-based computers to optimize the Proof-of-Work (PoW) required to mine a cryptocurrency (such as Bitcoin) in the cryptocurrency space.

The process of mining a cryptocurrency require computers on the network to confirm transactions grouped as a block. For this block to be added to the chain, computers have to repeatedly solve complex puzzles to derive a one-way mathematical solution (this process is called hashing). ASICs are specially optimized to carry out multiple hashing functions per second at impressive energy efficiency levels, improving the hashing rate on the network and, consequently, better security and faster transactions.

ASICs challenge the fundamental basis of a cryptocurrency – decentralization.

ASIC resistance – What does it mean for a coin to be ASIC resistant?
Given ASICs (Application Specific Integrated Circuit) miners are specialized machines, several massive mining farms constitute a major portion of the entire network's hashing power. This is because average cryptocurrency miners cannot generate the computing power required to match hashing rate and hashing power of these major mining farms using consumer-grade hardware like personal computers.

This centralizes the hash power of the Bitcoin blockchain network to a few locations in the world, which was the primary cause for the hash power of Bitcoin's network dropping by almost 50% in early 2021 due to a power outage at a massive mining farm in Northern China.

ASIC-resistant cryptocurrencies are a way to challenge this centralization of the hashing power of a blockchain network. The protocol and mining algorithm of ASIC-resistant cryptocurrencies are such that it is almost not profitable mining them with ASIC machines. In some cases, it is even worse than using generalized CPU and GPU hardware.

However, the hashing algorithm of ASIC-resistant cryptos is constantly modified and improved as ASIC manufacturers constantly improve on the ASIC miners. Some newer models have been seen to bypass the hashing algorithm of some ASIC-resistant coins.

What cryptocurrencies are ASIC resistant?

Ethereum is the most popular and largest cryptocurrency with the most Application Specific Integrated Circuit (ASIC) resistant hashing algorithm (called Ethash). The Ethereum mining algorithm was designed to be ASIC-resistant in a bid to lower the entry barrier posed to lone miners and encourage mining pool communities.

Other notable coins with ASIC-resistant hash functions include Monero (MXR), Vertcoin (VTC), and Horizon (ZEN).

What is a Turing machine?

A Turing machine is a hypothetical machine thought of by Alan Turing in 1936 that is capable of simulating the logic of any computer algorithm, no matter how complex. It is a very simple machine that consists of a tape of infinite length on which symbols can be written, a read/write head that can move back and forth along the tape and read or write symbols, and a finite state machine that controls the head and can change its state based on the symbols it reads or writes. The Turing machine is capable of solving any problem that can be solved by a computer algorithm, making it the theoretical basis for modern computing.

Turing Complete

Turing Complete refers to a machine that, given enough time and memory along with the necessary instructions, can solve any computational problem, no matter how complex. The term is normally used to describe modern programming languages as most of them are Turing Complete (C++, Python, JavaScript, etc.).

A device or programming language is considered to be **Turing Complete** when it can replicate a Turing Machine by running any program or solving any problem the Turing Machine could run or solve. On the other hand, if a device or programming language is not able to do it, then it is said to be **Turing Incomplete**.

Blockchain and Turing Completeness
While some applications of blockchain technology are Turing Complete, others are Turing Incomplete. This varies according to the scripting technology implemented. For example, the **scripting language used in Bitcoin is intentionally designed as Turing Incomplete** because it serves its purpose and increased complexity would potentially introduce problems. By keeping it simple, the developers can predict with high accuracy how it is going to react in the finite number of situations in which it is used.

Ethereum as Turing Completeness:
Ethereum, on the other hand, is built as a Turing Complete blockchain. This is important because it needs to understand the agreements which make up smart contracts. By being Turing Complete, Ethereum has the capability to understand and implement any future agreement, even those that have not been thought of yet. In other words, Ethereum’s Turing Completeness means that it is able to use its code base to perform virtually any task, as long as it has the correct instructions, enough time and processing power.

There are multiple features of the Ethereum platform that make it different from other crypto platforms. The platform is functionally different and unique from other platforms due to the programming language and setup via the Turing completeness process. The purpose of using such language is to protect data and to avoid the possible threat of information theft. Moreover, it enhances the system’s security against hack attacks and provides a secure electronic transaction system.

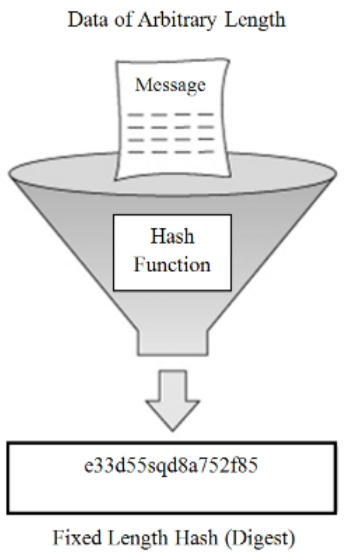
The Turing completeness offers a more logical solution that creates a protected and advanced system for safer investment and transaction monitoring. Therefore, Turing completeness can theoretically make cryptocurrency transactions more safe and secure.

Drawbacks of Turing completeness in blockchain
In public blockchains where code is visible to all, code may be vulnerable to disruptions (such as bugs in the smart contracts), or unintended uses, that impede the intended functioning of the protocol. Being able to program any kind of computation allows for a vast possibility of outcomes, and it’s not possible to anticipate all of them.

What Is Hash Function?

The hash function applies to the message to generate a message digest, a fixed-size hexadecimal output.

It takes an arbitrary size input as a message. INPUT M: The Message
Uses a Cryptographic hash function H to encrypt the message M.
Generates the output called Digest.
H(M) = Digest



Cryptographic hash functions are irreversible. That means it's a 1-way function, and one can't generate the message back using the digest.

A hash function is a mathematical function that takes input data of arbitrary size and outputs a fixed-size string of data, typically a sequence of numbers and letters that represent the original input. The main aim of a hash function is to provide a way to map data of arbitrary size to data of fixed size.

These functions are used in many applications, including cryptography, data integrity checking, data indexing, and data fingerprinting. For instance, in cryptography, hash functions are used to generate digital signatures, which can be used to verify the authenticity of a message or document.

Properties

Determinism: A hash function is deterministic, meaning a given input will always produce the same output.
Uniformity: A good hash function should produce uniformly distributed outputs.
Non-reversibility: A hash function is non-reversible, meaning it is impossible to determine the input that produced a given output. This property is essential because it helps to ensure data security and confidentiality.
Fixed-size output
Sensitivity to input changes: A slight change in the input to a hash function should produce a significant difference in the output
Collision resistance: A good hash function should be resistant to collisions, which occur when different inputs produce the same output. Collision resistance is significant because it helps ensure data accuracy and reliability.
Speed: A hash function should be fast and efficient, as it is for real-time applications where speed is critical.

Application

Cryptography: These are used in cryptography to ensure the confidentiality and integrity of data. They generate digital signatures, which verify the authenticity of a message or document. Hash functions also create message digests, ensuring data integrity during transmission.
Data integrity checking: These verify that data has been unaltered during transmission. This is done by generating a hash value for the data before it is transmitted and another hash value for the data after it has been received.
Data indexing: These create indexes for large data sets. This allows for quick retrieval of data, even from extensive databases.
Data fingerprinting: These uniquely identify data, such as file-sharing networks. Generating a hash value for a piece of data makes it possible to identify and ensure it is uniquely safe.
Password storage
Digital forensics: These are popular in digital forensics to ensure the authenticity of evidence.
Blockchain: These are extensively popular in blockchain technology. Each block in a blockchain contains a hash value of the previous block, ensuring the entire blockchain's integrity. Additionally, they also mine new partnerships in a blockchain.

Types of Cryptographic Hash Functions

1. RACE Integrity Primitives Evaluation Message Digest (RIPEMD): This set includes RIPEMD, RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320.
2. Message-Digest Algorithm: This family comprises hash functions MD2, MD4, MD5, and MD6.
MD5 is the most widely used cryptographic hash function.
3. BLAKE2:
4. BLAKE3:
5. Whirlpool
6. Secure Hashing Algorithm: 4 SHA algorithms: SHA-0, SHA-1, SHA-2, and SHA-3.

> SHA-0 is a 160-bit hash function that was published by the National Institute of Standards and Technology in 1993.
> SHA-1 was designed in 1995 to correct the weaknesses of SHA-0. In 2005, a method was found to uncover collisions in the SHA-1 algorithm due to which long-term employability became doubtful.
> SHA-2 has the following SHA variants, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, and SHA-512/256. It is a stronger hash function and it still follows the design of SHA-1.
> In 2012, the Keccak algorithm was chosen as the new SHA-3 standard.
> SHA-256 is the most famous of all cryptographic hash functions because it's used extensively in blockchain technology. The SHA-256 Hashing algorithm was developed by the National Security Agency (NSA) in 2001.

Uses of Hash Functions in Blockchain

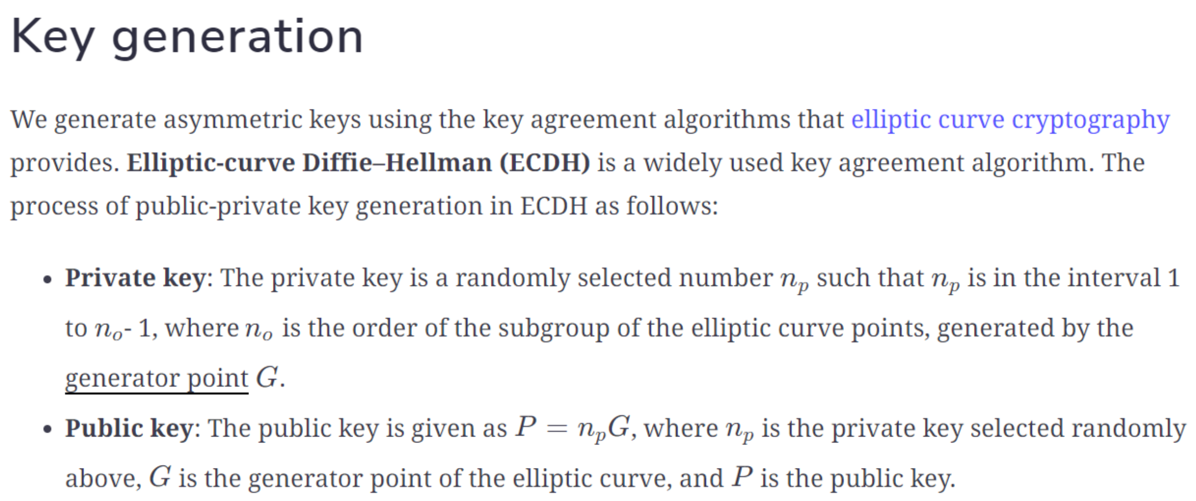
The blockchain has a number of different uses for hash functions. Some of the most common uses of the hash function in blockchain are:

Merkle Tree: This uses hash functions to make sure that it is infeasible to find two Merkle trees with the same root hash. This helps to protect the integrity of the block header by storing the root hash within the block header and thus protecting the integrity of the transactions.
Proof of Work Consensus: This algorithm defines a valid block as the one whose block header has a hash value less than the threshold value.
Digital signatures: Hash functions are the vital part of digital signatures that ensures data integrity and are used for authentication for blockchain transactions.

The chain of blocks: Each block header in a block in the blockchain contains the hash of the previous block header. This ensures that it is not possible to change even a single block in a blockchain without being detected. As modifying one block requires generating new versions of every following block, thus increasing the difficulty

Elliptic Curve Digital Signature Algorithm (ECDSA)?

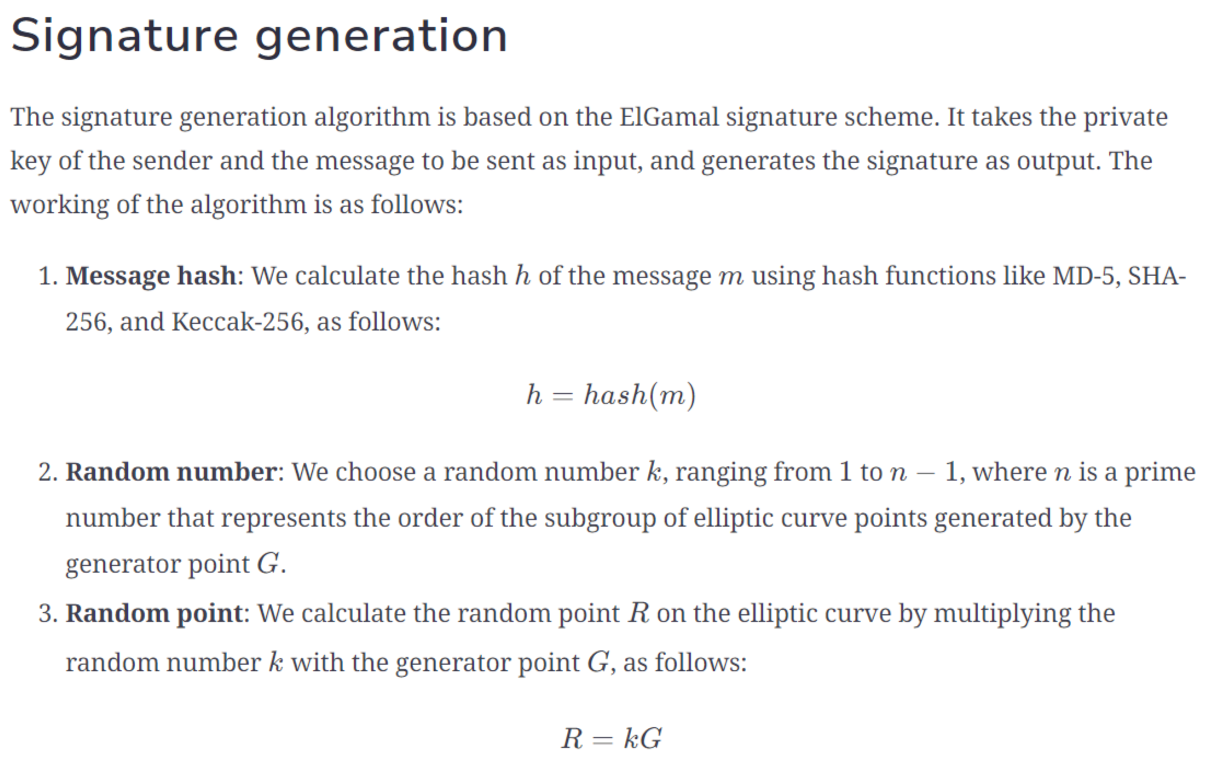
The Elliptic Curve Digital Signature Algorithm (ECDSA) is a digital signature algorithm (DSA). ECDSA relies on elliptic curves defined over a finite field to generate and verify signatures. The underlying elliptic curves make the signing process more efficient and secure, as the process relies on the complexity of the elliptic-curve discrete logarithm problem (ECDLP).



Key generation

We generate asymmetric keys using the key agreement algorithms that elliptic curve cryptography provides. **Elliptic-curve Diffie–Hellman (ECDH)** is a widely used key agreement algorithm. The process of public-private key generation in ECDH as follows:

- **Private key**: The private key is a randomly selected number n_p such that n_p is in the interval 1 to n_o- 1, where n_o is the order of the subgroup of the elliptic curve points, generated by the generator point G.
- **Public key**: The public key is given as $P = n_pG$, where n_p is the private key selected randomly above, G is the generator point of the elliptic curve, and P is the public key.



Signature generation

The signature generation algorithm is based on the ElGamal signature scheme. It takes the private key of the sender and the message to be sent as input, and generates the signature as output. The working of the algorithm is as follows:

1. **Message hash**: We calculate the hash h of the message m using hash functions like MD-5, SHA-256, and Keccak-256, as follows:

$$h = hash(m)$$

2. **Random number**: We choose a random number k, ranging from 1 to $n - 1$, where n is a prime number that represents the order of the subgroup of elliptic curve points generated by the generator point G.
3. **Random point**: We calculate the random point R on the elliptic curve by multiplying the random number k with the generator point G, as follows:

$$R = kG$$

4. x**-coordinate**: We select the x-coordinate of the random point generated above, as follows:

$$r = R.x$$

5. **Signature proof**: We apply the following equation to calculate the signature proof s, as follows:

$$s = k^{-1} \times (h + r \times n_p)\,(mod\,n)$$

The signature consists of two integer values calculated above r and s.

Signature Verification

The signature verification algorithm takes the message and the signature $r,\ s$ as input, and returns a boolean value representing whether the signature is verified. The signature verification algorithm works as follows:

1. **Message hash**: We calculate the hash h of the message m using the same hash function that was we used during the signature generation, as follows:

$$h = hash(m)$$

2. **Modular inverse**: We calculate the modular inverse of the signature, as follows:

$$s_{inverse} = s^{-1}(mod\,n)$$

3. **Random point**: We recalculate the random point R' as in the signature generation process, where P is the public key of the sender, as follows:

$$R\text{'} = (h \times s_{inverse}) \times G + (r \times s_{inverse}) \times P$$

4. x**-coordinate:** We get the x-coordinate of the recalculated random point, as follows:

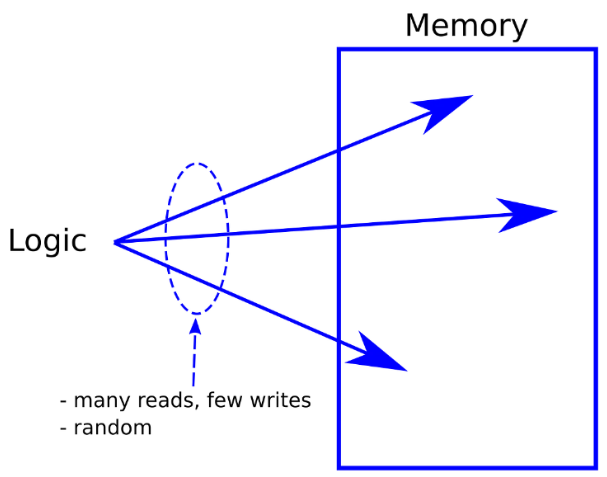
$$r\text{'} = R\text{'}.x$$

5. **Verify**: We verify the result by matching the recently calculated r' with the r that came as part of the signature, as follows:

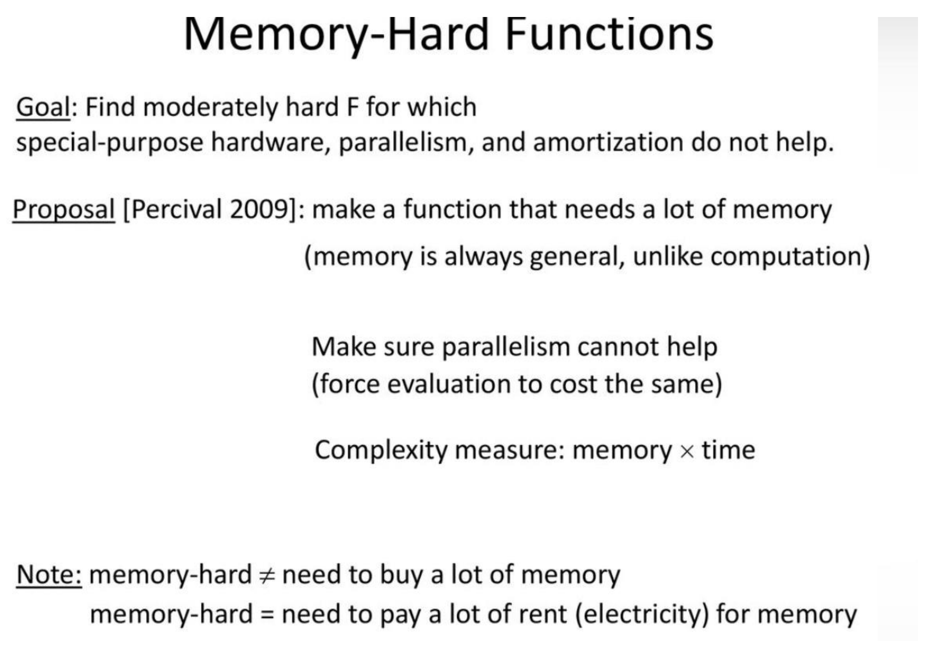
$$r\text{'} == r$$

Memory Hard Algorithm

Memory-hard algorithms were introduced to prevent the centralization of mining power on the blockchain network, as ASICs can be expensive to purchase and maintain. Examples of memory-hard algorithms used in blockchain include Ethash, Equihash, and CryptoNight



A memory-hard algorithm in the blockchain is a type of computational puzzle that requires a large amount of memory to solve, making it resistant to ASIC-based mining.
Memory-hard algorithms were introduced to prevent the centralization of mining power on the blockchain network, as ASICs can be expensive to purchase and maintain.
Examples of memory-hard algorithms used in blockchain include Ethash, Equihash, and CryptoNight.
These algorithms require miners to access data from memory continuously, which slows down the mining process, ensuring that mining remains decentralized and accessible to all.
This makes it more difficult for attackers to create a parallel blockchain, as they would need to have a large amount of memory available to support their fork.



Memory-Hard Functions

Goal: Find moderately hard F for which special-purpose hardware, parallelism, and amortization do not help.

Proposal [Percival 2009]: make a function that needs a lot of memory
(memory is always general, unlike computation)

Make sure parallelism cannot help
(force evaluation to cost the same)

Complexity measure: memory $\times$ time

Note: memory-hard $\neq$ need to buy a lot of memory
memory-hard = need to pay a lot of rent (electricity) for memory

What Are Zero-Knowledge Proofs?

A zero-knowledge (ZK) proof is a cryptographic protocol that enables one person (the prover) to convince another (the verifier) that a particular claim is true without disclosing any details about the claim itself.

A good zero-knowledge proof should fulfil the following three criteria:

- Completeness: The verifier will accept the proof with a high probability if the proposition is true, and both the prover and the verifier adhere to the protocol.
- Soundness: If the claim is untrue, no prover should be able to persuade the verifier of the opposite, save under extremely unlikely circumstances.
- Zero-knowledge: Even after engaging with the prover, the verifier only understands the truth of the statement and nothing else about the secret.

There are different types of ZK-proofs:

- Interactive ZK-proofs: Interactive zero-knowledge proofs require back-and-forth communication between the prover and verifier.
- Non-interactive ZK-proofs: Non-interactive zero-knowledge proofs provide a compact proof that can be verified in a single step.
- Statistical ZK-proofs: Statistical zero-knowledge proofs offer computational soundness with a small probability of error.
- Proof-of-knowledge (PoK): PoK is a subclass of ZK-proofs that shows that the prover possesses specific knowledge related to the statement.
- Proofs of shuffle and range: These ZK-proofs are used in electronic voting and privacy-preserving transactions.
- Sigma protocols: Sigma protocols are a class of ZK-proofs that involve three steps: commitment, challenge and response.
- Bulletproofs: Bulletproofs are designed to provide efficient range proofs for large sets of values

How do zero-knowledge proofs work?

Zero-knowledge proofs (ZKPs) work by allowing a prover to demonstrate the validity of a statement without revealing any additional information about the statement. This is done by providing proof, or a small amount of information, that can be verified by a verifier to ensure that the statement is true.

A real-life example of this involves the banking industry, where a customer may want to prove their identity to a bank without revealing any personal information, such as their social security number. A ZKP called a zero-knowledge proof of identity (ZKPI) can be used. In this scenario, the customer is the prover, and the bank is the verifier. The customer provides a proof, such as a government-issued ID, which the bank can use to verify their identity. The bank can then confirm that the customer is who they claim to be without revealing any sensitive information.

Another example can be found in the field of electronic voting systems, where a voter can prove that their vote was cast and counted without revealing who they voted for. This is done by providing a proof, such as a digital signature, which can be verified by the voting system without revealing the actual vote.

In both examples above, the prover is providing a proof that can be verified by the verifier without revealing any additional information about the statement, which is, in this case, the personal information of the customer or the vote of the voter.

Why do we need zero-knowledge proofs?

There are several reasons why zero-knowledge proofs (ZKPs) are important.

1. Privacy And Security: ZKPs allow for the verification of information without revealing the underlying data, providing a high level of security and privacy. This is particularly useful in situations where sensitive information needs to be kept confidential, such as in financial transactions or personal identification.

2. Compliance: ZKPs can help organizations comply with various regulations, such as data privacy laws. They can also be used to provide secure and private identity verification for compliance with know-your-customer (KYC) and anti-money laundering (AML) regulations.

3. Scalability: ZKPs can help improve the scalability of blockchain networks by allowing for the verification of transactions without the need to reveal the underlying data. This can decrease the amount of data that must be retained on the blockchain, thereby augmenting the efficiency of the network.

4. Interoperability: ZKPs can help to facilitate the interoperability of different blockchain networks by providing a secure and private way to share information across different networks.

5. Identity Verification: ZKPs can also be used to provide secure and private identity verification. This can be useful in scenarios where users want to prove their identity without revealing sensitive personal information.

Zero-knowledge proofs are a powerful tool that can be used to enhance privacy and security, comply with regulations, improve scalability and interoperability and provide secure identity verification. They allow for the verification of information without revealing the underlying data. This provides a high level of security and privacy, which is particularly important in fields such as finance and personal identification.

The use of ZKPs in blockchain technology is particularly valuable, as it allows for the verification of transactions without revealing the underlying data, which can improve the scalability and privacy of blockchain networks.

As the need for protection and privacy in the digital era increases, zero-knowledge proofs (ZKPs) will assume a more significant function in diverse domains like blockchain technology.

What are the applications of zero-knowledge proofs?

ZK-proofs are crucial in the world of cryptocurrencies for improving transaction privacy and scalability.

ZK-proofs can be used in the authentication and access control fields to demonstrate an understanding of a password or a cryptographic key without revealing the password or key itself.

ZK-proofs are also used in electronic voting systems, where they allow voters to demonstrate the legitimacy of their vote without disclosing the actual vote, protecting both voter privacy and the integrity of the electoral process.

ZK-proofs also have implications for secure data transfer and verification, giving one side the ability to demonstrate the accuracy of computations on private data without disclosing the data itself.

Zero-knowledge proofs can improve transaction privacy in central bank digital currencies (CBDCs) by facilitating private transactions and upholding user anonymity. By balancing privacy and transparency in CBDC transactions, ZK-proofs enable auditability without disclosing transaction specifics.