Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2022/2023

LECTURE 30,31: DLT AND BLOCKCHAIN

What is a Distributed Ledger?

Distributed Ledger

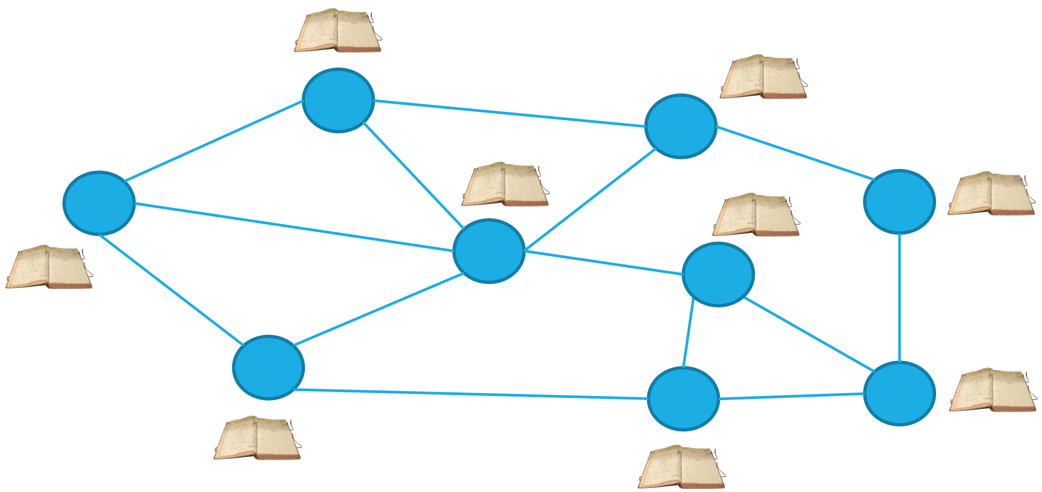
A ledger is a written or computerized record of all the transactions a business has completed

A distributed ledger is a database replicated across several locations or among multiple participants used to store record of transactions

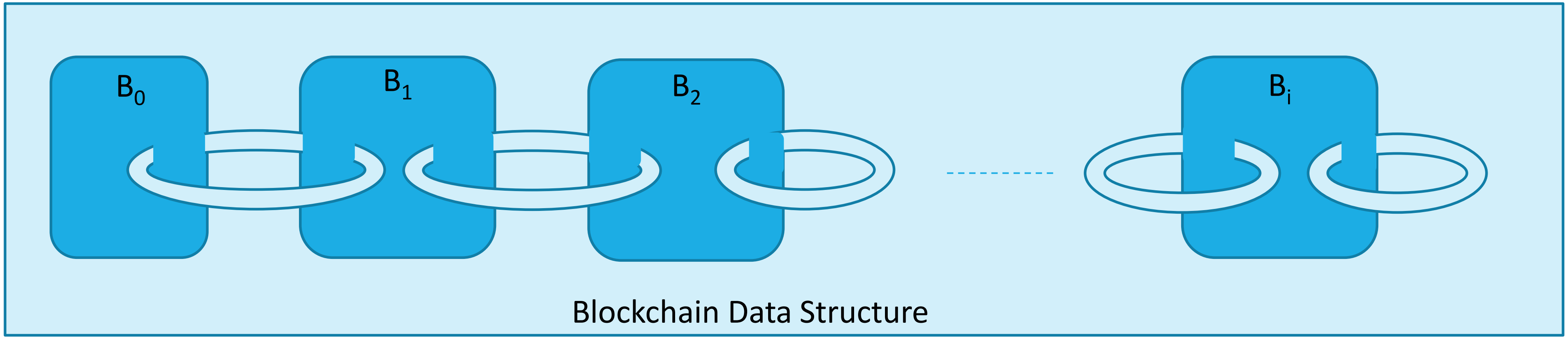
What is a Distributed Ledger?

Key properties

- Consistency
- Integrity
- Availability

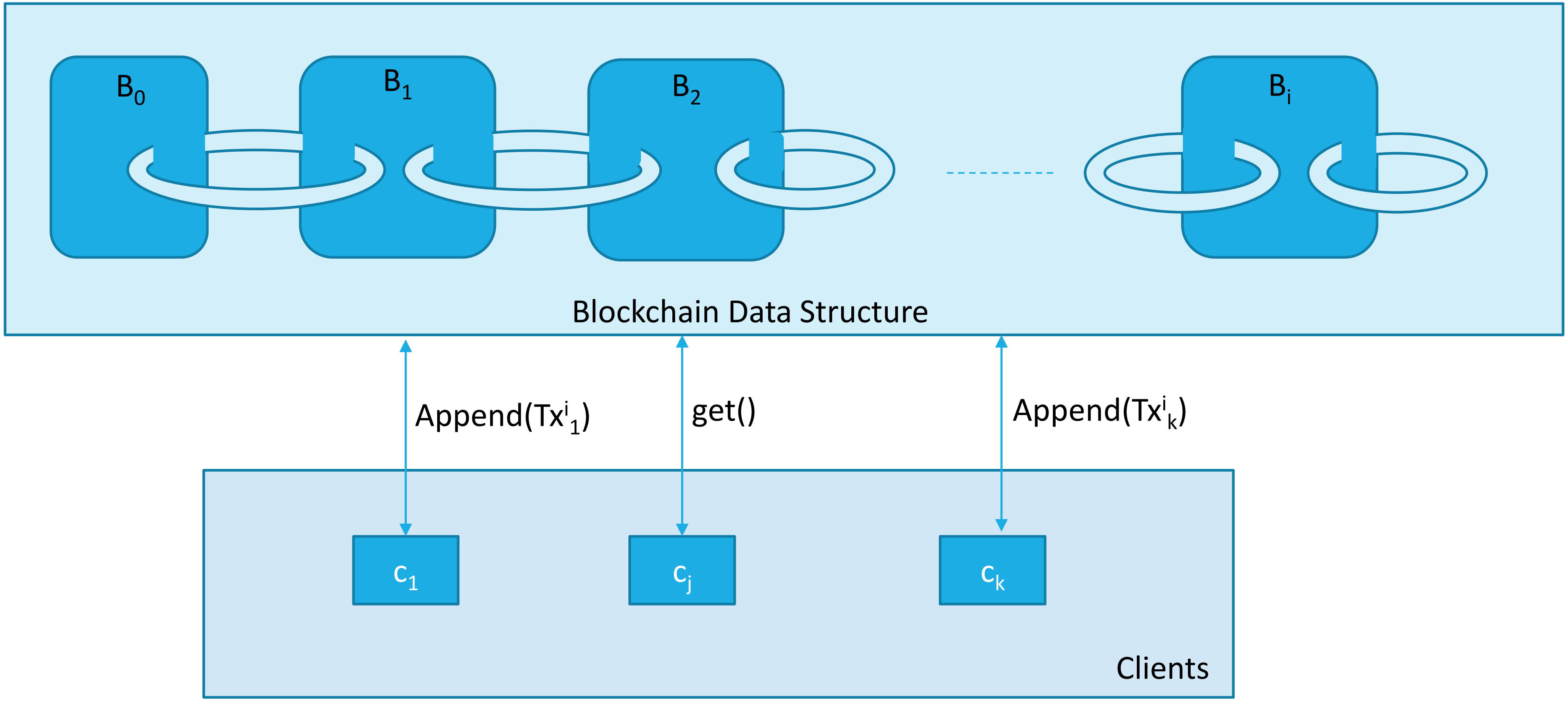


What is a blockchain?



A blockchain is a **decentralized**, **distributed** data structure used to **record transactions** (aggregated in blocks) across many computers.

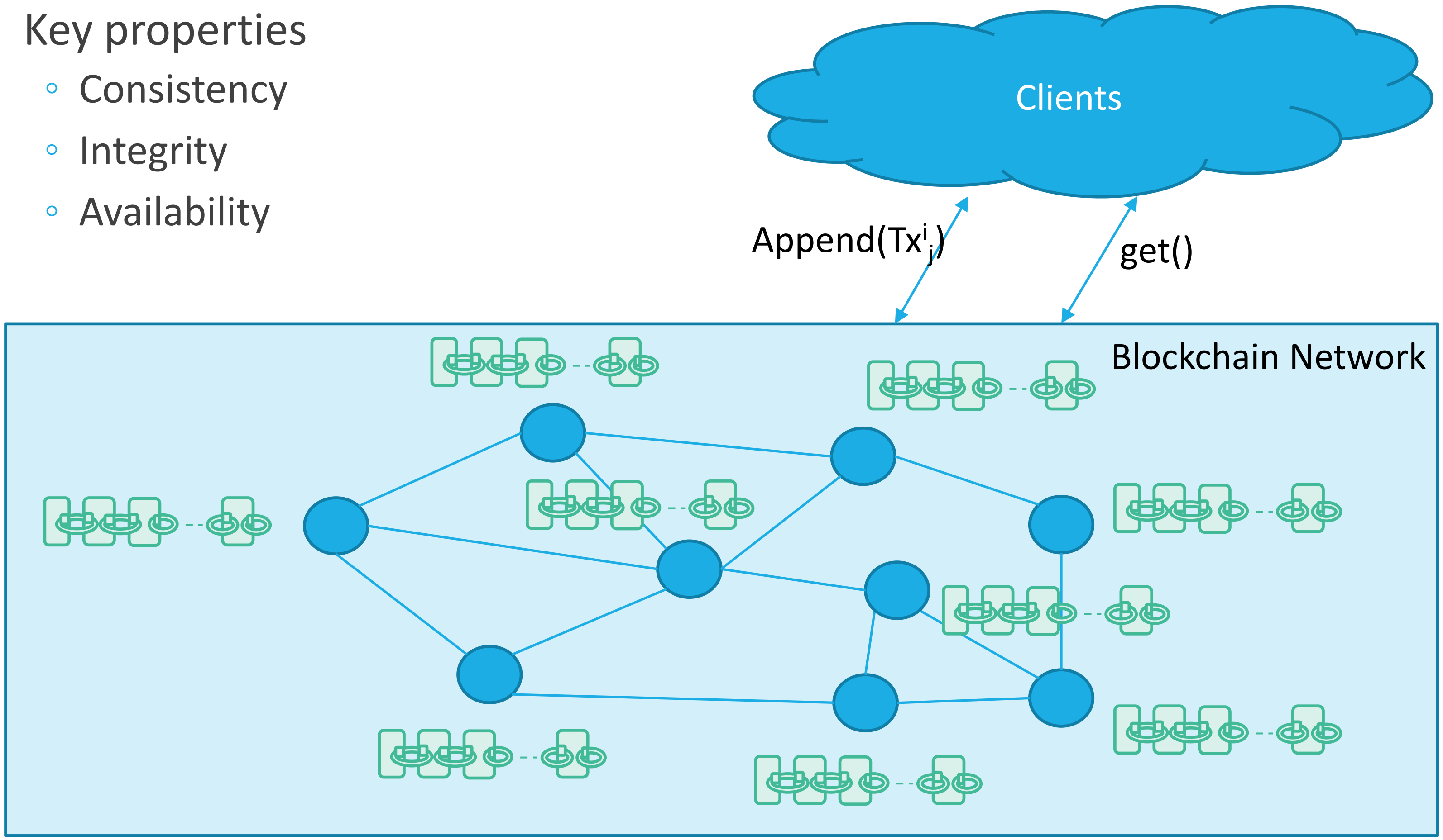
What is a blockchain?



What is a Blockchain?

Key properties

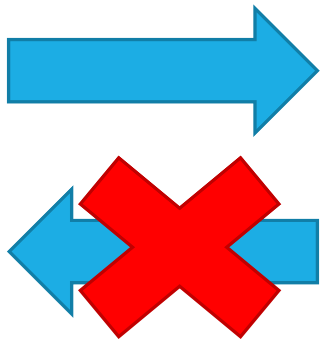
- Consistency
- Integrity
- Availability



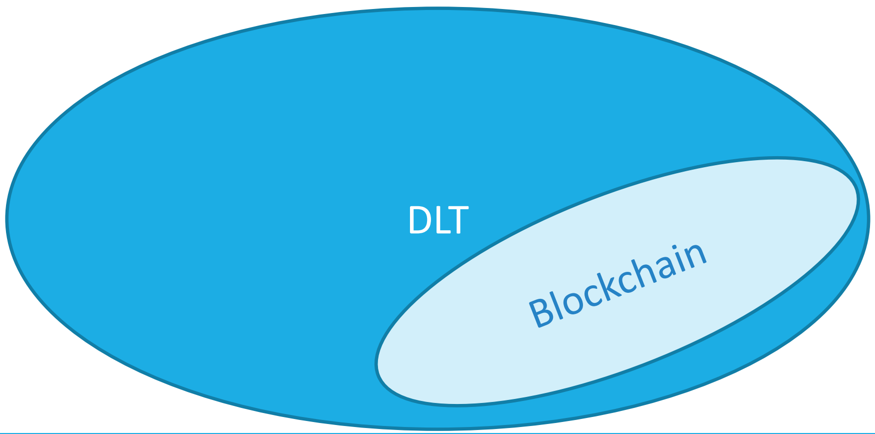
Blockchain vs DL

Every blockchain represents a distributed ledger but the vice versa is not true

Blockchain



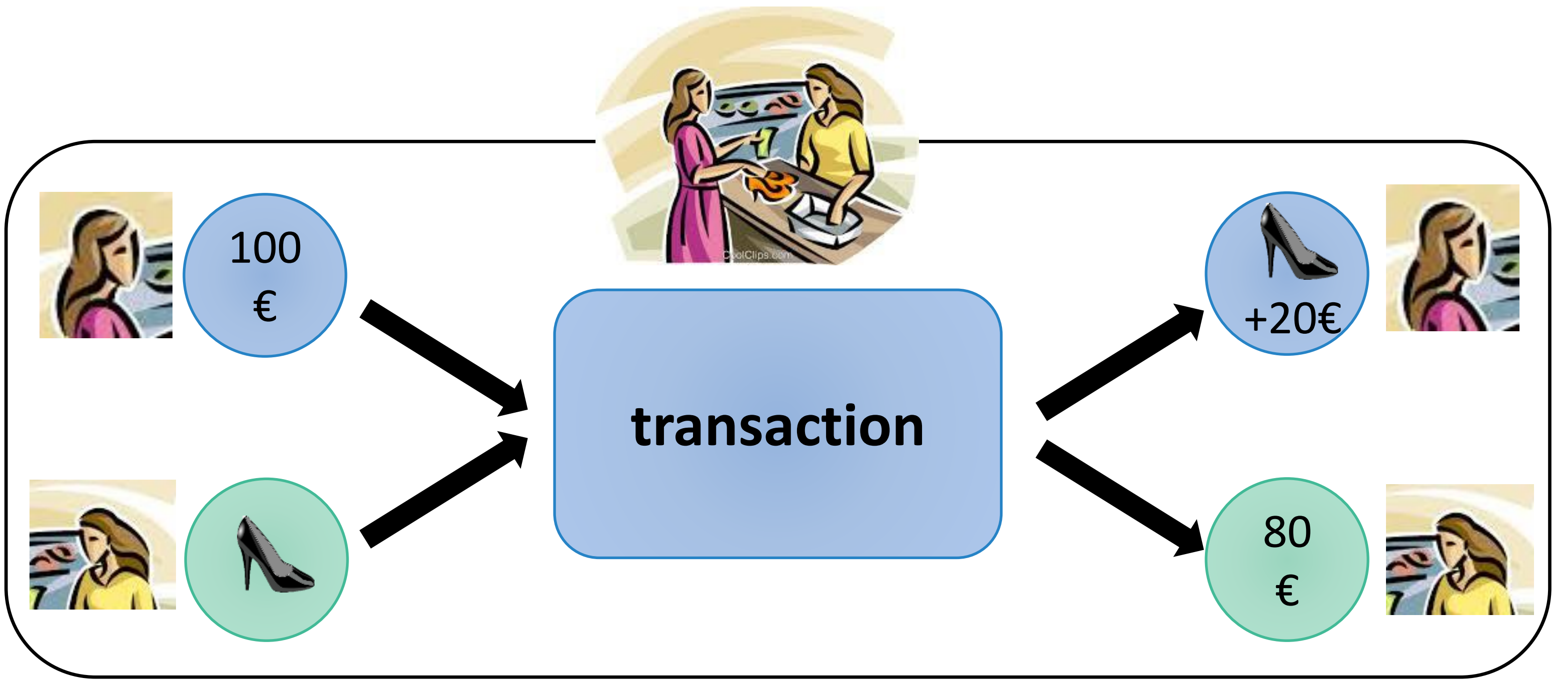
Distributed Ledger



What is a transaction?

A transaction is

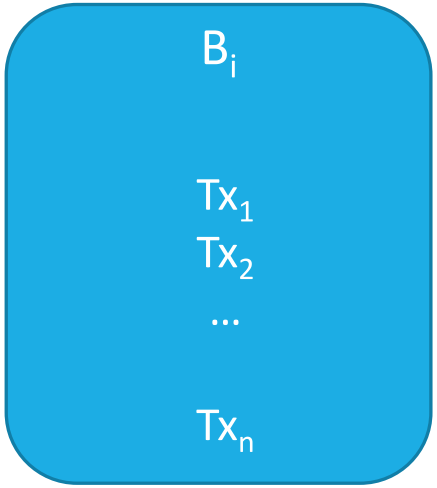
- an instance of buying or selling something
- an exchange or interaction between people



What is a Block?

Every block in a blockchain contains a set of transaction

- Clients generate transactions and submit them to nodes implementing the blockchain



How to create a Block?

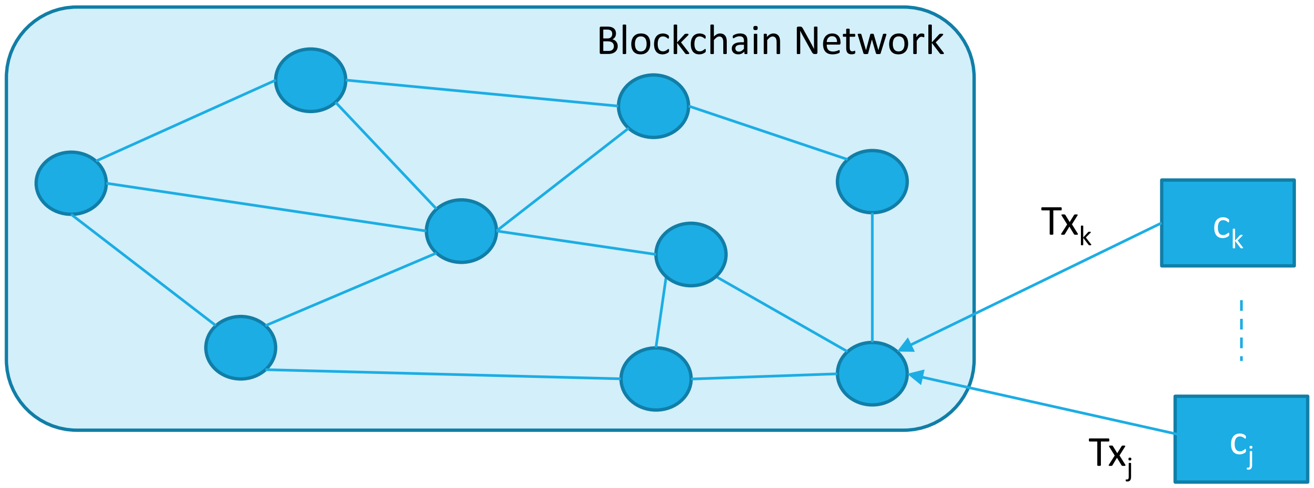
Transactions are collected by processes

Transactions need to be validated i.e., they need to be valid with respect to the ledger specification

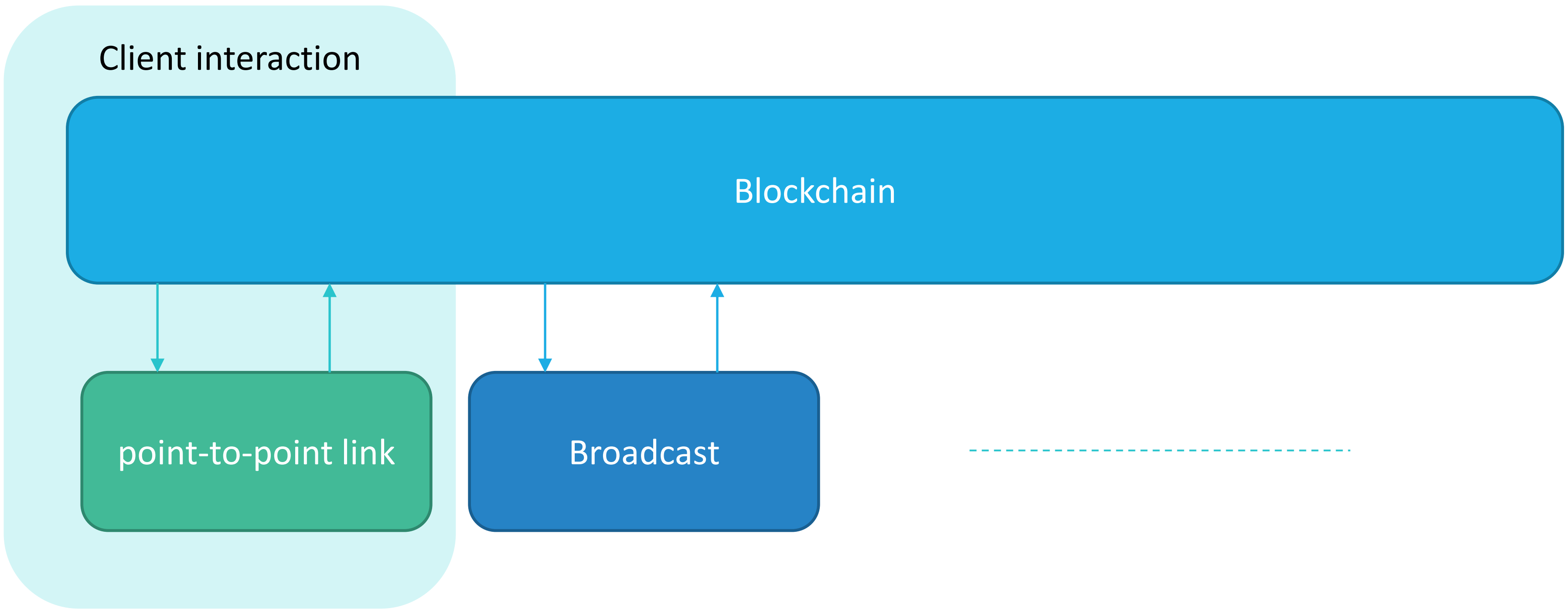
- E.g., in a financial ledger a transaction cannot spend money that are not in the account

Transactions remain *unconfirmed* until they are inserted in a block that is successfully attached to the chain

When ”enough” transactions have been collected a block can be created and attached



Key ingredients



How to attach a block to the chain?

Validate transaction

- In order to validate a transaction, nodes need to read the “last” state of the leader and check that the current transaction is correct with respect to the semantics of the ledger
  - E.g. if in the last block we found that Alice’s account has 300$, then any transaction spending at most 300$ will be valid

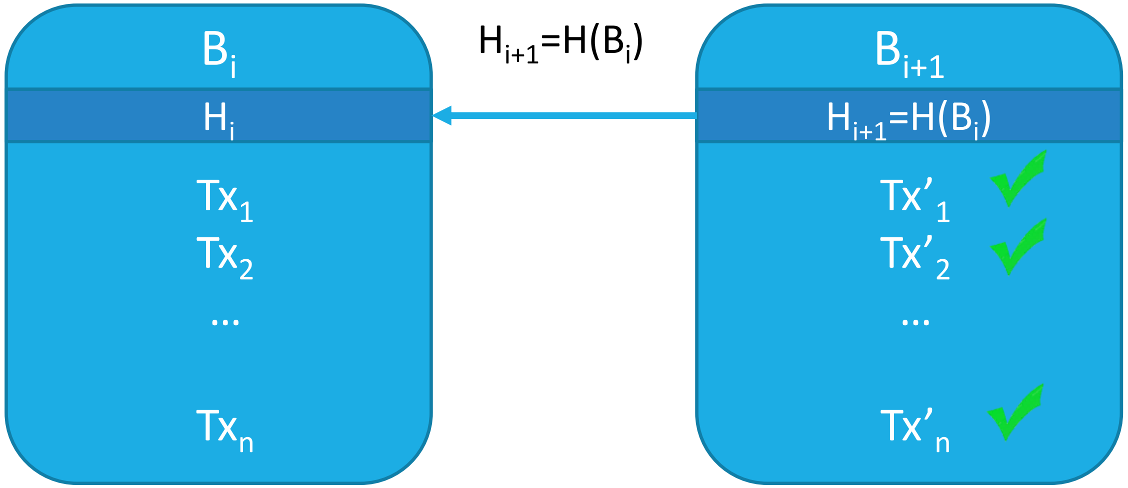
OBSERVATION

- Ordering of transactions in the blocks is fundamental to check and guarantee the validity of the ledger

How to attach a block to the chain?

Chaining blocks

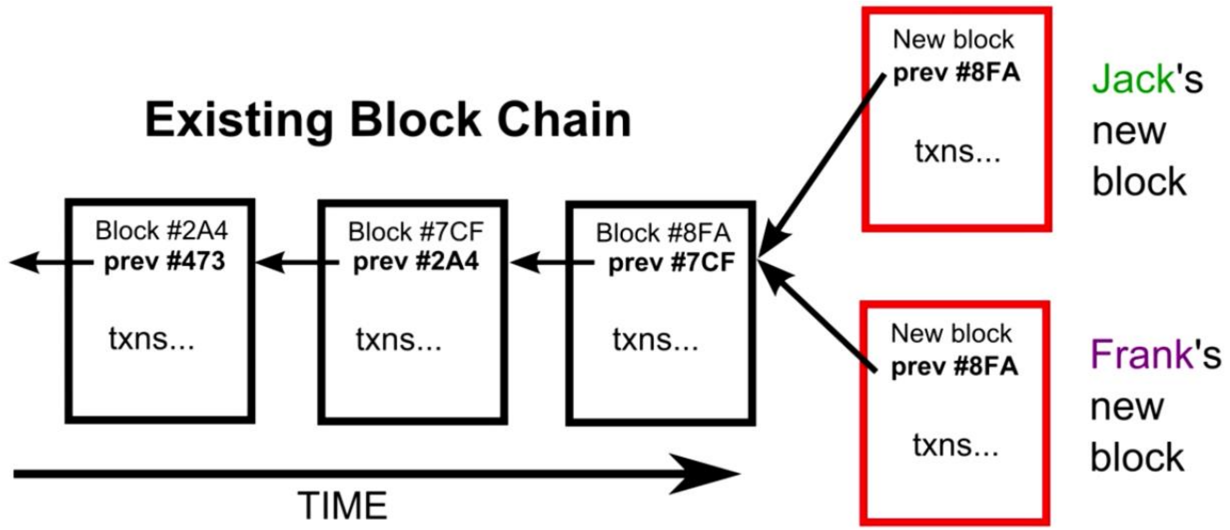
1. Check that all the transactions in the block are valid
2. Compute the hash H() of the last block attached to the chain
3. Include this hash in the current block to generate a pointer and attach the current block



How to attach a block to the chain?

ISSUES

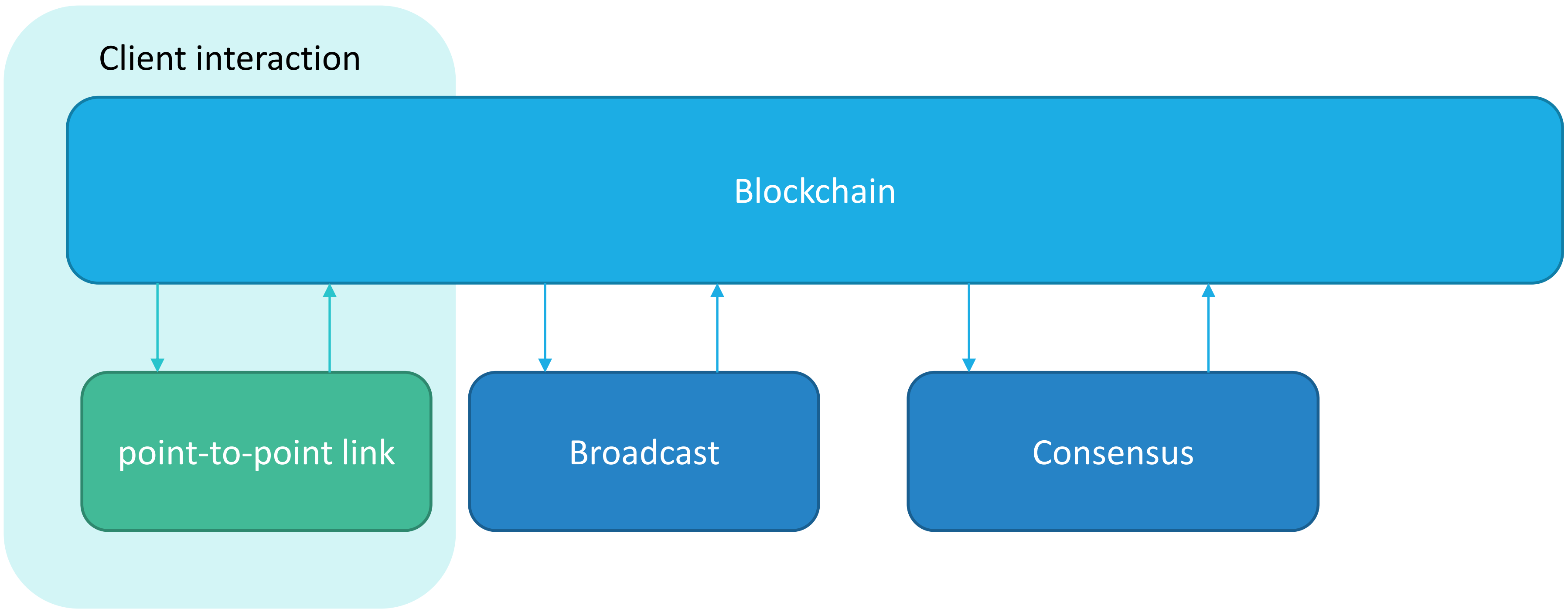
- Concurrency in the block creation process
- Asynchronous systems



How to create and chain blocks

Attaching a new block requires agreement between nodes

Key ingredients

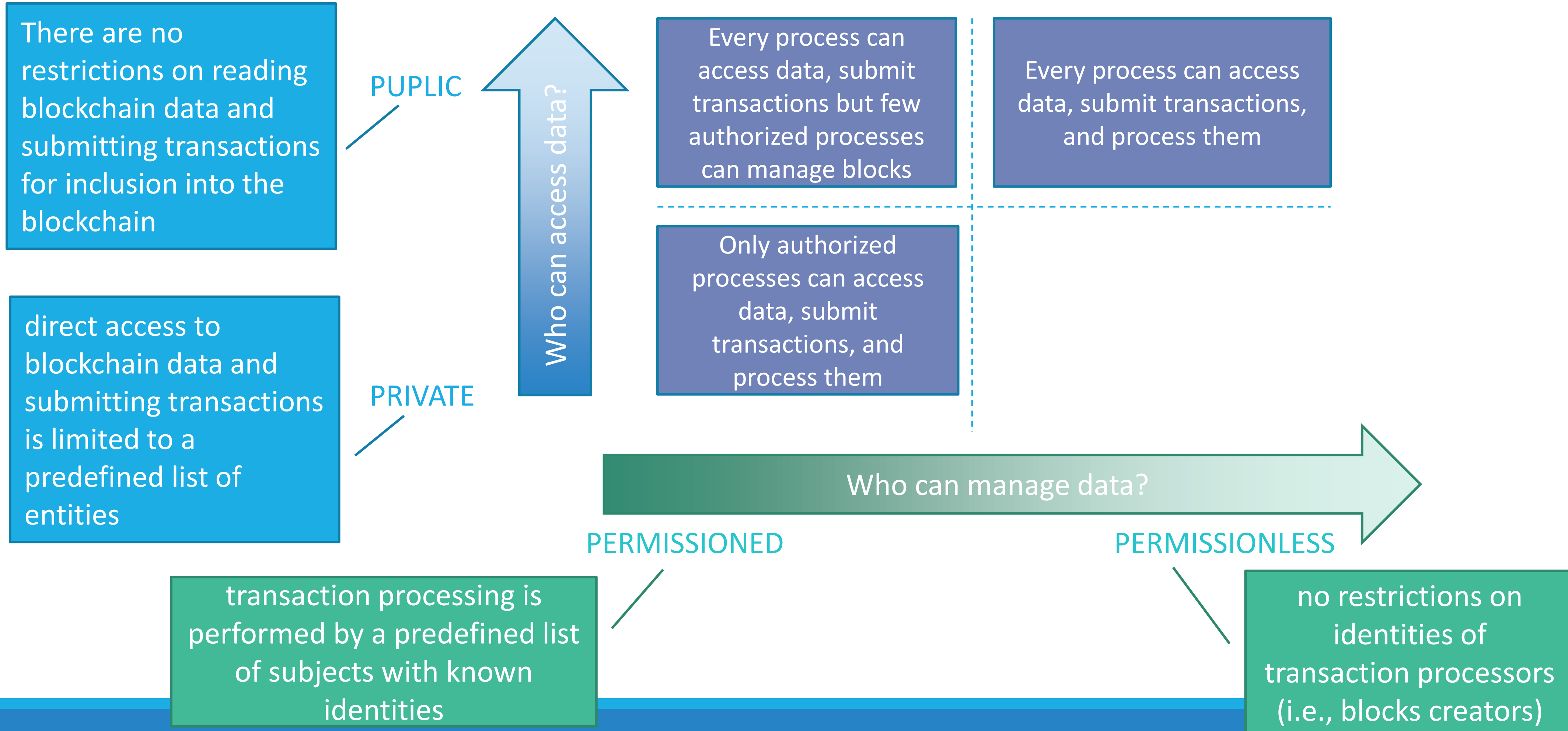


How to create and chain blocks

Attaching a new block requires agreement between nodes

Which Consensus protocol should I use?

Blockchain Classification



How to create and chain blocks

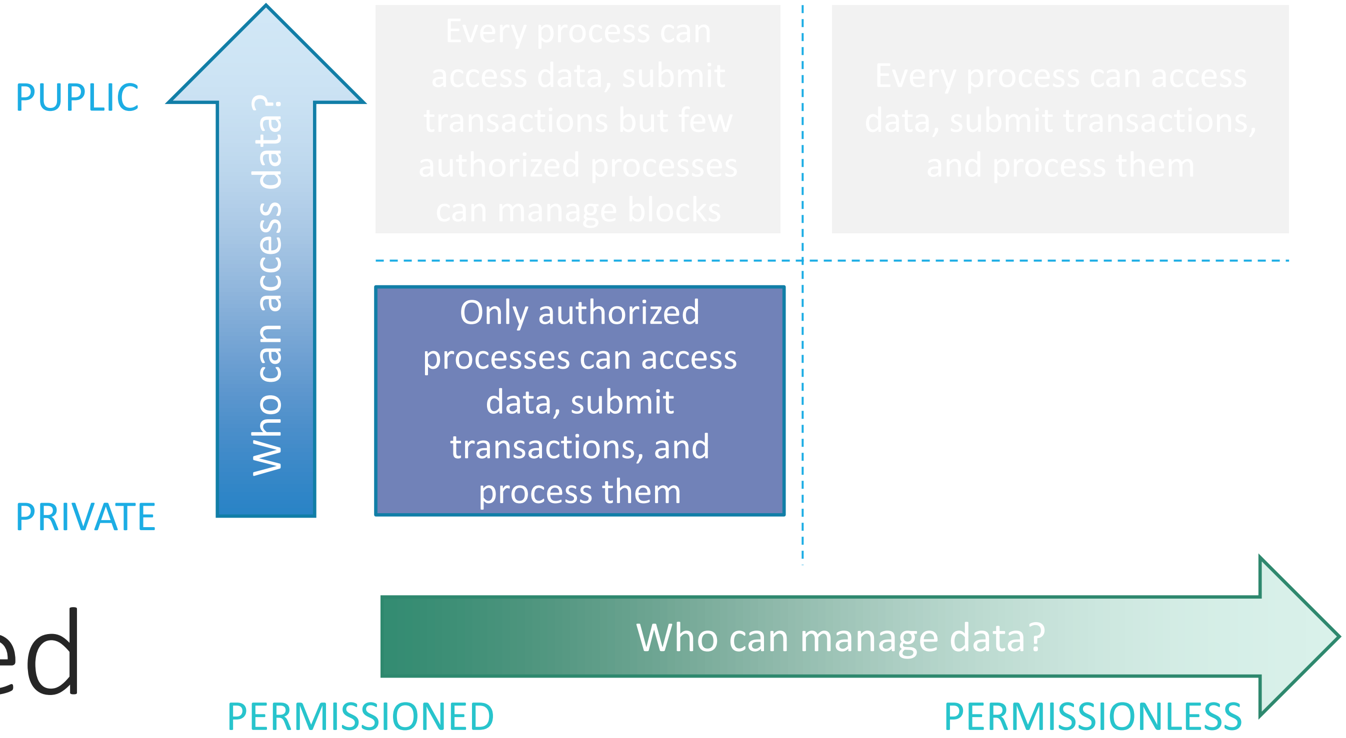
Attaching a new block requires Consensus

Which Consensus protocol should I use?

Public Permissionless Blockchain → **Proof of X** (i.e., leader based consensus)

OR

Private Permissioned Blockchain → **BFT Consensus** (fully distributed consensus)



Private Permissioned Blockchain

THE PBFT PROTOCOL

Practical Byzantine Fault Tolerance

In private permissioned blockchain we have that

- Transactions are submitted only by known clients and only known clients can access them
- Blocks are created and attached by known and authorized processes

IDEA

- Processes maintain a copy of the Blockchain locally and run a BFT consensus protocol to agree on the next block
- PBFT is currently the adopted protocol
  - It has been designed to support replication
  - It merges the primary-backup approach with the Byzantine General Consensus approach

Practical Byzantine Fault Tolerance

System model:

- Asynchronous distributed system
  - nodes connected by network
  - messages can be lost, delayed, duplicated, no order
- Independent node failures (byzantine)

Basically the same assumed to solve the Byzantine Generals Problem

Practical Byzantine Fault Tolerance

System model:

- Cryptographic techniques (public key signatures, MAC, message digest produced by collision resistant hash functions)
  - needed to avoid spoofing, replay attacks and corruption
- All replicas know each other’s public keys to verify signatures

Practical Byzantine Fault Tolerance

System model:

- Strong adversary that can
  - coordinate the action of faulty nodes
  - delay communication
  - delay correct nodes (not indefinitely)
- The adversary cannot subvert the cryptographic techniques cited above

Practical Byzantine Fault Tolerance

Properties:

- Safety
  - satisfies linearizability – like a centralized system that executes operations atomically one at a time
  - regardless of faulty clients
  - but faulty clients can do strange (legal) operations (e.g., write garbage in a file system)
  - limit damage by access control mechanisms

Practical Byzantine Fault Tolerance

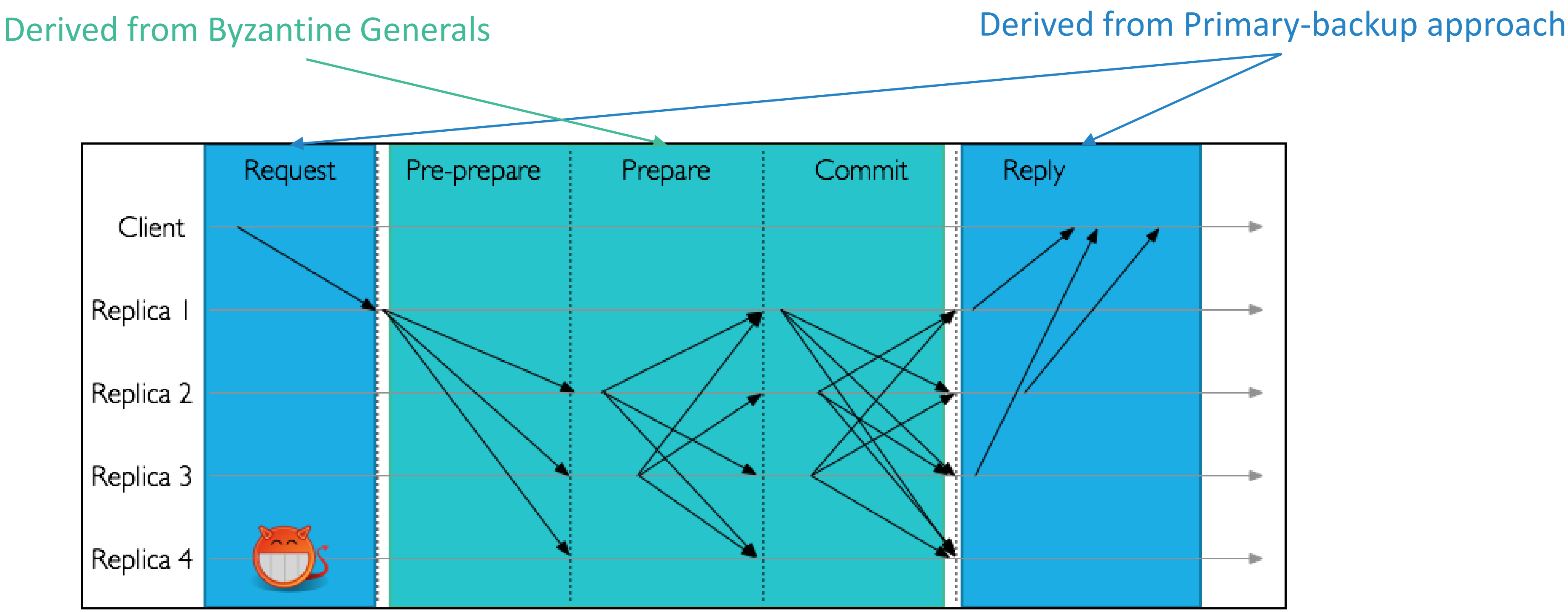
Properties:

- Liveness
  - clients receive responses to their requests if at most (n-1)/3 replicas are faulty and delay(t) does not grow faster than t indefinitely

Practical Byzantine Fault Tolerance

The protocol is divided in five phases:

- Request, pre-prepare, prepare, commit and reply



Practical Byzantine Fault Tolerance

Basic idea:

- A client sends a request to invoke an operation to the primary
- The primary multicasts the request to the backups
- Replicas execute the request and send a reply to the client
- The client waits for f+1 replies from different replicas with the same result; this is the result of the operation

Practical Byzantine Fault Tolerance

If the primary is not correct it is substituted with a new one as soon as a misbehavior is detected

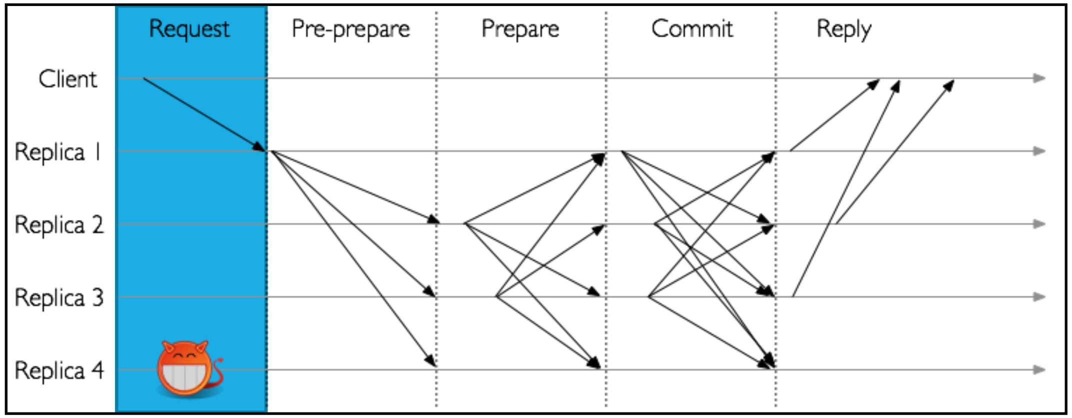
The system lifetime is characterized by a series of *views* each with a different primary

Views are numbered

Practical Byzantine Fault Tolerance

Request phase:

- Client *c* requests the execution of operation *o* issuing the request <REQUEST, *o*, *t*, *c*>$_{\sigma c}$ to the primary
- The request is sent to the last primary known by the client
  - If a response is not received by a timeout the request is multicast to all replicas
- *t* is a timestamp used to guarantee *exactly-once* semantic



Practical Byzantine Fault Tolerance

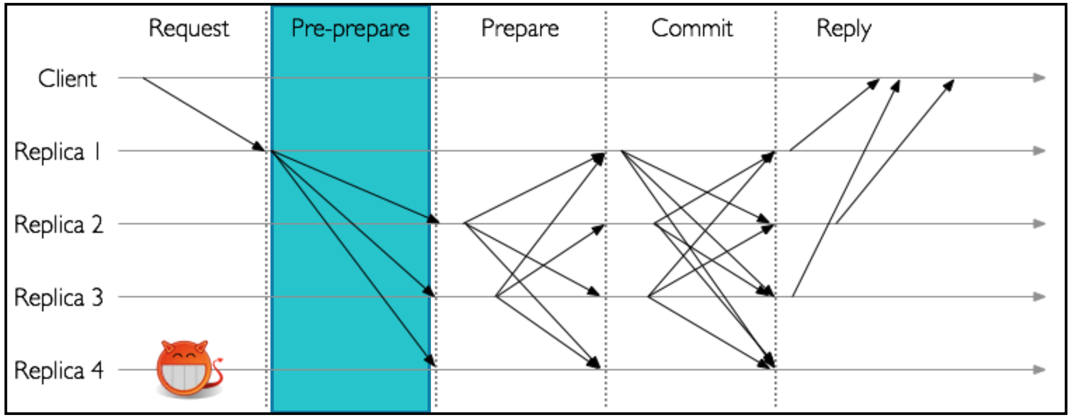
Replica internal state:

- replica id *i* (between 0 and N-1)
- service state
- view number *v* (initially 0)
- log of accepted messages

Practical Byzantine Fault Tolerance

Pre-prepare phase:

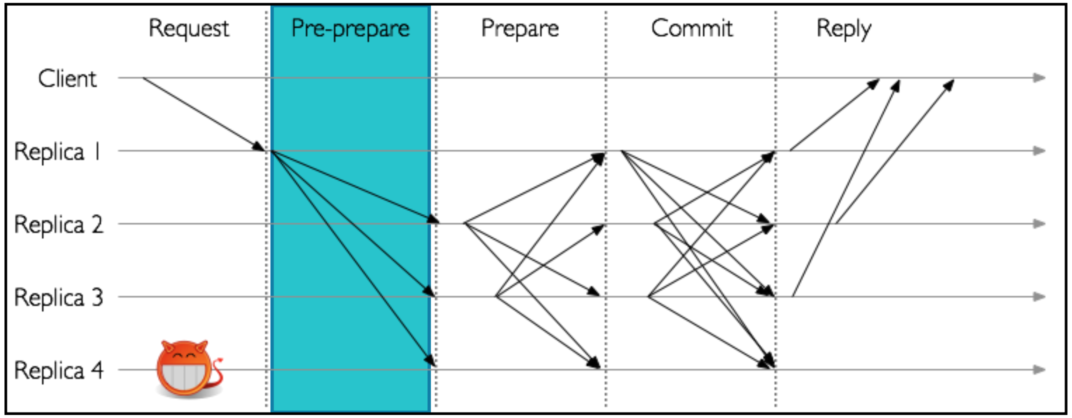
- the primary *p* sends to all backups the message
  <<PRE_PREPARE, *c*, *n*, *d*>$_{\sigma p}$, *m*>
  - *c* is the client id
  - *n* is a sequence number for the current view assigned by the primary
  - *d* is a digest of the client request
  - *m* is the client request



Practical Byzantine Fault Tolerance

Pre-prepare phase:

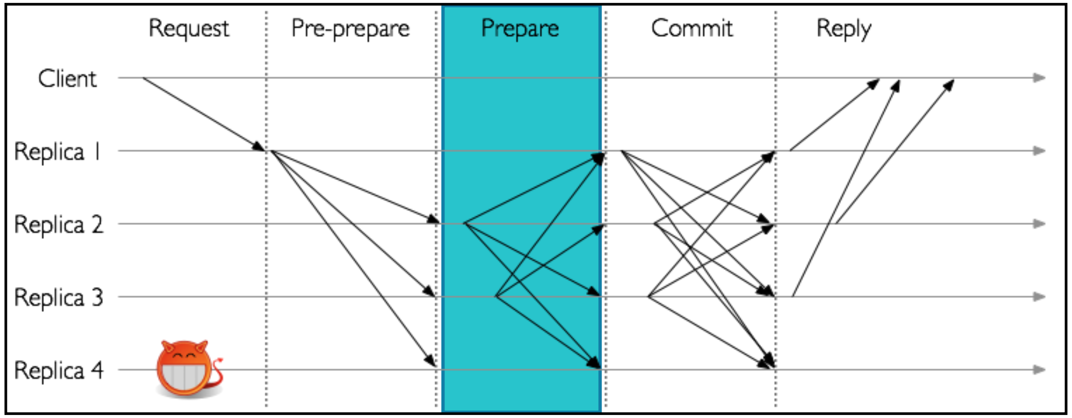
- a backup accepts (puts in the log) the pre-prepare msg iff:
    1. the signature in the pre-prepare and request messages are verified, and *d* is *m*’s digest
    2. it is currently in view *v*
    3. it has not accepted a pre-prepare message for view *v* and sequence number *n* containing a different digest
    4. the sequence number *n* is between a low watermark *h* and a high watermark *H*



Practical Byzantine Fault Tolerance

Prepare phase:

- a backup i sends to all nodes the message
  <PREPARE, *v*, *n*, *d*, *i*>$_{\sigma i}$
- another node accepts a prepare message (and puts it in the log) iff:
  1. signatures are correct
  2. it is currently in view *v*
  3. *n* is between *h* and *H*



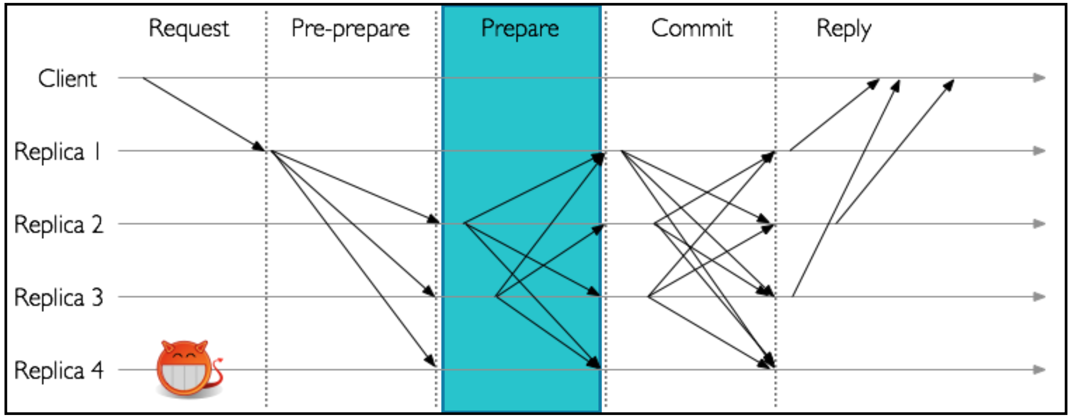
Practical Byzantine Fault Tolerance

Replicas remain blocked until the predicate *prepared*(*m*,*v*,*n*,*i*) becomes true

This happen iff replica *i* has in its log:

1. the message m
2. a pre-prepare for m
3. More than *2f* prepare messages from different backups that match the pre-prepare

matching among messages is done checking v, n and d



Practical Byzantine Fault Tolerance

The pre-prepare and prepare phase guarantee that non-faulty replicas agree on a total order of execution for requests in a same view

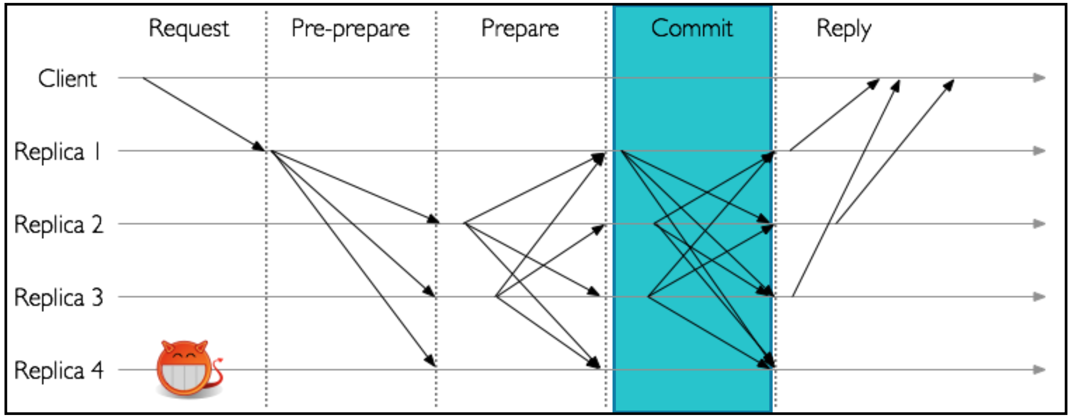
More precisely, they ensure that if *prepared*(m,v,n,i) is true then *prepared*(m',v,n,j) is false for any non faulty replica j and any m' such that the digest of m is different from the digest of m'

This is due to the intersection among quorums of nodes that accept the prepare messages

Practical Byzantine Fault Tolerance

Commit phase:

- as soon as *prepared*(*m*,*v*,*n*,*i*) becomes true replica i sends to all the message <COMMIT, *v*, *n*, *d*, *i*>$_{\sigma i}$
- another node accepts a commit message (and puts it in the log) iff:
  1. signatures are correct
  2. it is in view *v*
  3. *n* is between *h* and *H*



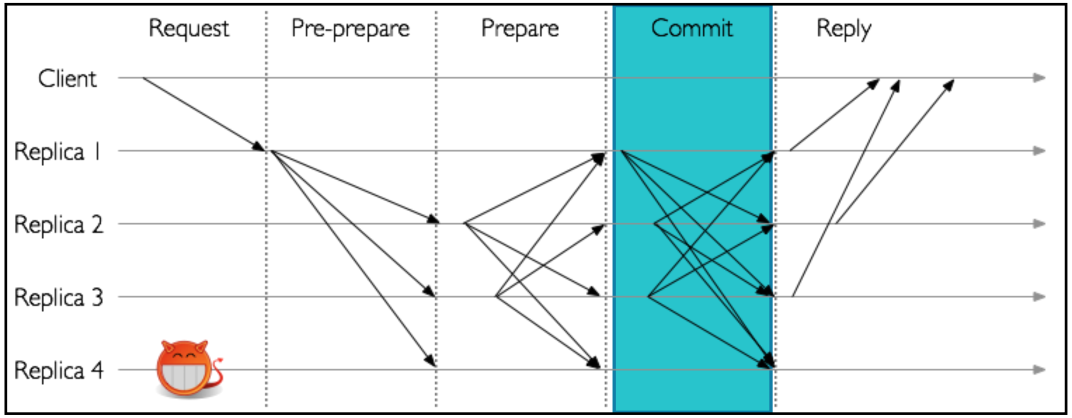
Practical Byzantine Fault Tolerance

The predicate *committed*(*m*,*v*,*n*) is true iff *prepared*(*m*,*v*,*n*,*i*) is true for all *i* in a set of *f+1* non faulty replicas.

The predicate *committed-local*(*m*,*v*,*n,i*) is true iff *prepared*(*m*,*v*,*n*,*i*) is true and *i* has accepted *2f+1* commits from different replicas that match the pre-prepare for m

The commit phase guarantees that if *committed-local*(*m*,*v*,*n,i*) is true for some non faulty *i*, then *committed*(*m*,*v*,*n*) is true.

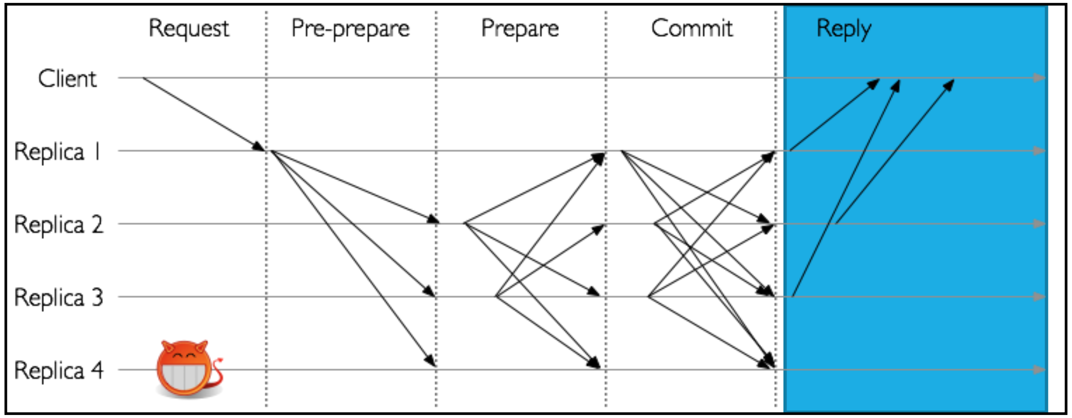
- because if *i* received *2f+1* commits, *prepared*(*m*,*v*,*n*,*i*) is true for at least *f+1* non-faulty nodes.



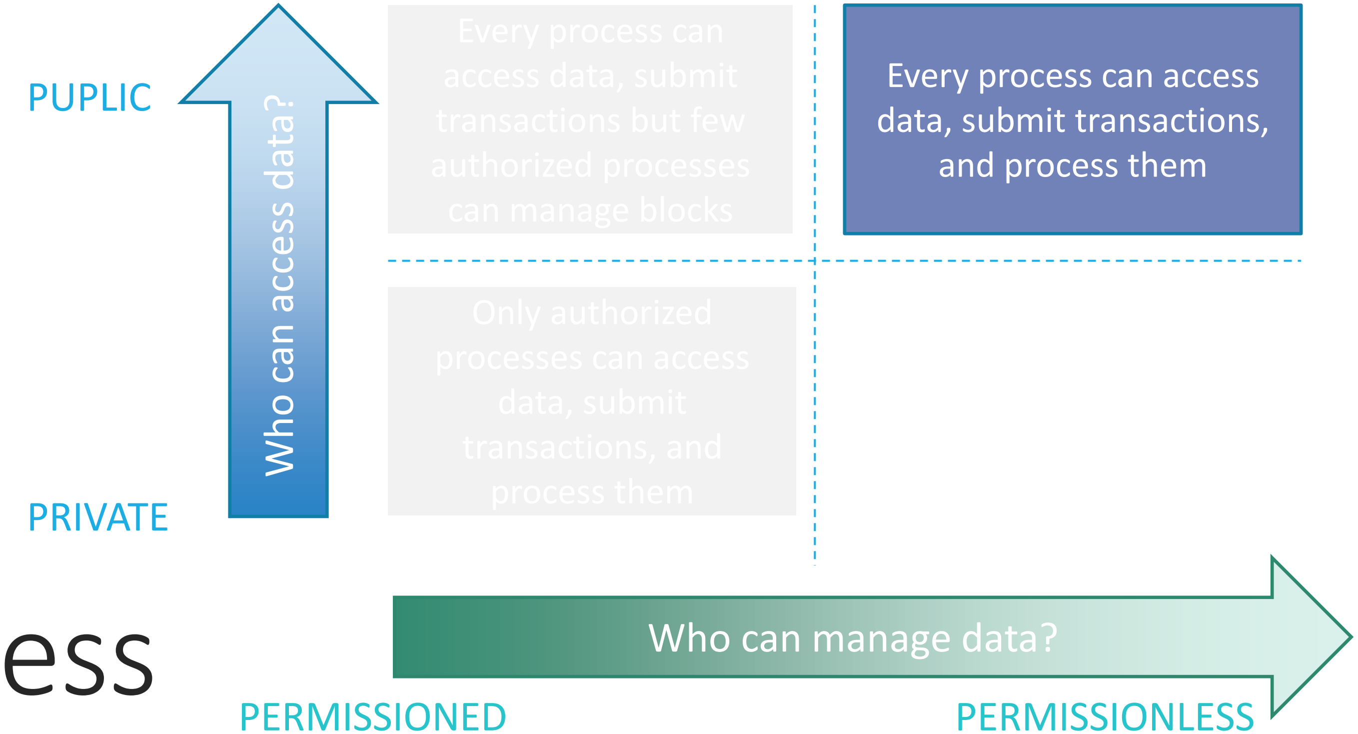
Practical Byzantine Fault Tolerance

Reply phase:

- A replica *i* executes the request as soon as
  - *committed-local*(*m*,*v*,*n*,*i*) is true
  - *i*'s state reflects the sequential execution of all requests
- The response is then sent to the client <REPLY, *v*, *t*, *c*, *i*, *r*>$_{\sigma i}$
- The client waits for *f+1* replies with valid signatures and the same values for *t* and *r*



Public Permissionless Blockchain



PROOF OF X

Public Permissionless Blockchain

In public permissionless blockchain we have that

- Transactions can be submitted by everyone and every process can read the chain
- Blocks can be created and attached by every process in the system

OBESERVATION

- Public permissioned blockchains are characterized by
  - Lack of trust in other processes
  - Possibly large scale

CONSEQUENCE

- PBFT-like protocols do not work

Public Permissionless Blockchain

IDEA

- Processes start a competition and only the winner can attach its block to the blockchain (i.e., they are trying to elect a leader)
- They implement a “randomized leader election”

ISSUES

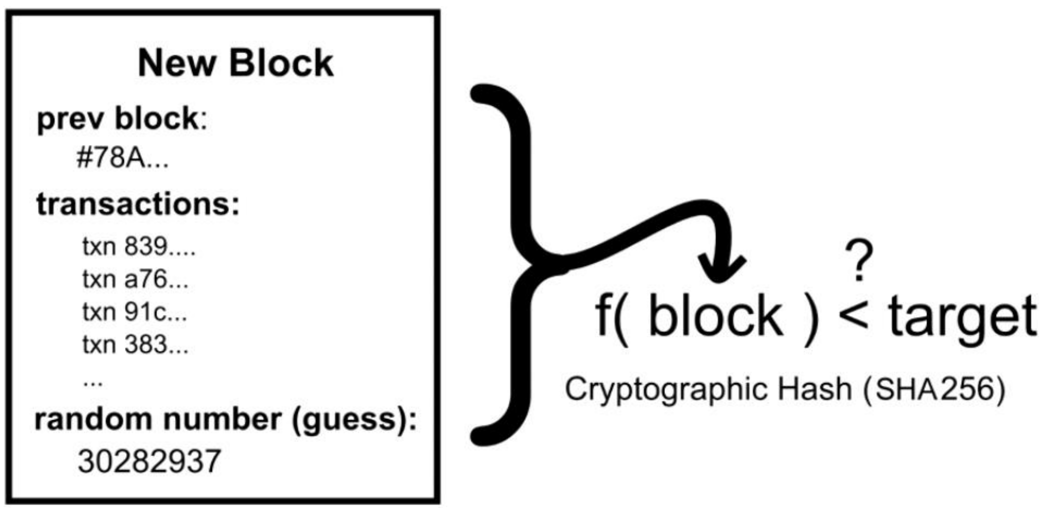
- Two processes may win the competition

⇩

- Two blocks can be attacked to the chain

Proof-of-work (PoW) & Mining

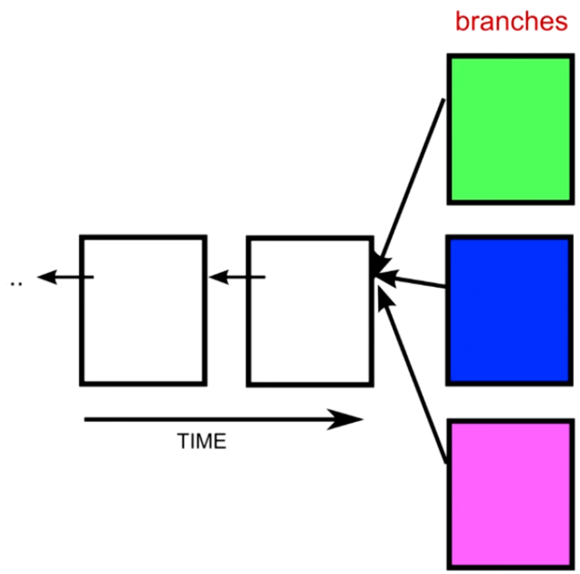
- **Proof-of-work:** mathematical challenge to "solve" a block
- **Mining:** find a number s.t. hash(block) < target
- The first node who "solves" a **block** can propose it as next in the blockchain
- Other nodes which receive the block re-compute the hash to check the validity



Proof-of-work & Mining: Branches Management

- Occasionally two or more blocks may arrive together → **temporal disagree**
  - The probability that two or more nodes mine a block at the same time is very low

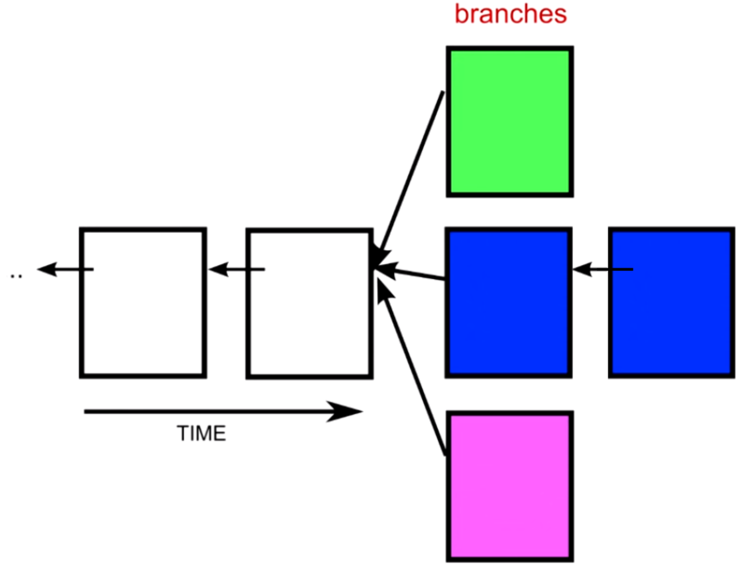
- **RULE**: in case of branches the network has to converge to the longest branch.



Proof-of-work & Mining: Branches Management

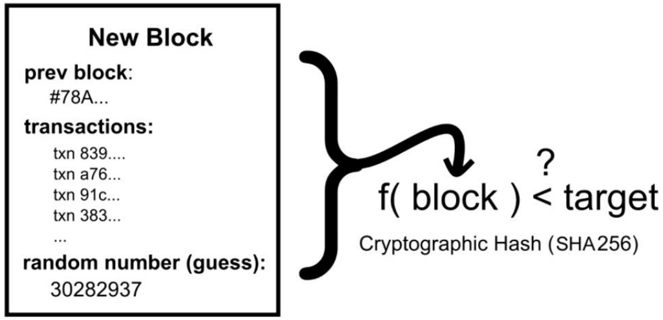
- Occasionally two or more blocks may arrive together → **temporal disagree**
  - The probability that two or more nodes mine a block at the same time is very low

- **RULE**: in case of branches the network has to converge to the longest branch.
  - **Problem solved with next blocks:** *eventually one branch will become the longest (usually after one block) bringing convergence*



Scalability Issues

- Transaction rate depends on two parameters
  - **Block size**: how many transactions to include in a block?
  - **Block interval**: how long to wait for a block to propagate to all the nodes?
    - ➢ Change run-time the **difficult of the target** to solve a block according to the computational power of the network

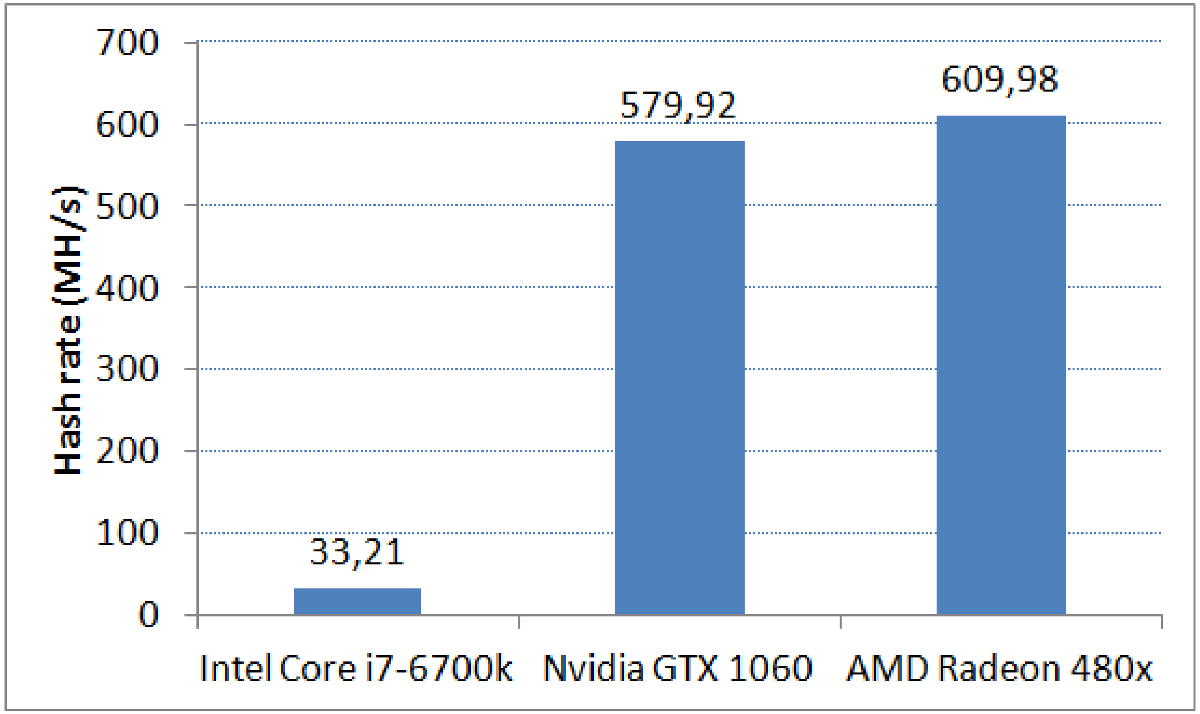


- Metrics
  - **Throughput**: how many transactions per second?
  - **Latency**: how long to wait for a transaction to complete?

- How parameters impact on metrics
  - Increasing block size improves throughput, but the resulting bigger blocks take longer to propagate in the network
  - Reducing the block interval reduces latency, but leads to instability where the system is in disagreement and the blockchain is subject to reorganization

Miner Requirements

- A miner needs a **high computational power** to compute a thousands of hash per seconds (KH/s or MH/s)
- CPUs are ineffective (few cores) better using GPUs



Miner Requirements

- Possible to improve performances by employing a cluster of GPU

KADA 6.1 GPU Mining Rig Open Air Frame Case Chassis with 6 USB Risers - Ethereum

99.6 % Positive feedback (★ ★ ★ ★ ★)

235 $

Listing Status: Completed

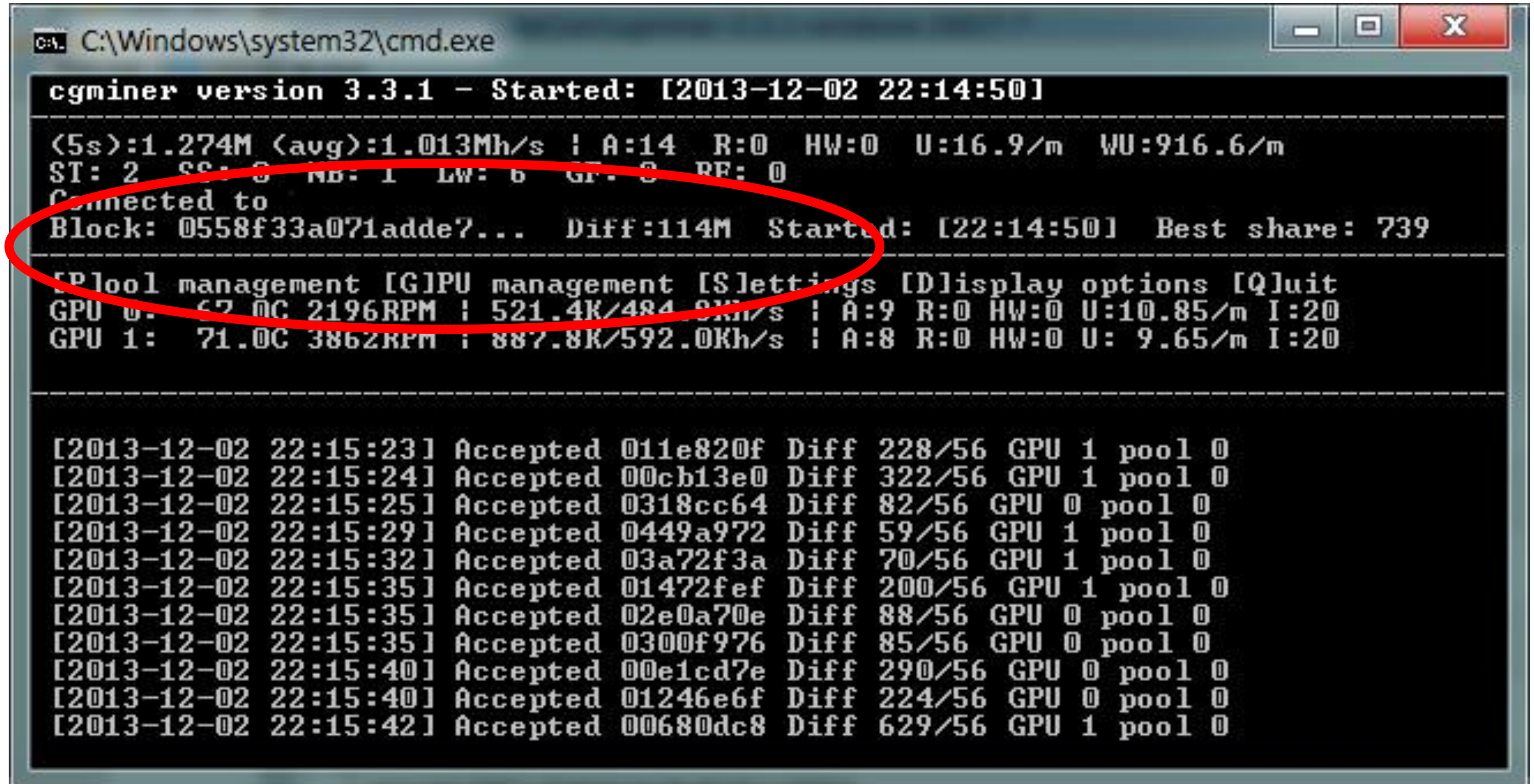
Country: US

Item condition: New

Buy Now!

Software for Mining

- CGminer, BFGminer, BitMiner, BTCMiner, DiabloMiner, …



ASIC

- ASIC: alternative specific hardware for mining

- Example: AntMiner S5
  - Hashrate: 1.15GH/s
  - Price: $370
  - Power consumption: 590W
    - Mining for 24 hours/day is expensive!

Question:

Why should a user mine?

Miner Incentive

- **Reward**: Solving a block gives coins to node that found the proof-of-work
  - Some txn may bring **additional fee** to node who mines the block containing that txns
  - Currently the txn fees are quasi-zero
  - Miners will be motivated to include in a block txn with a fee

- Rewards are an incentive for nodes to keep them supporting the blockchain and keep nodes honest
  - Invalid txns won’t give to miner the reward!

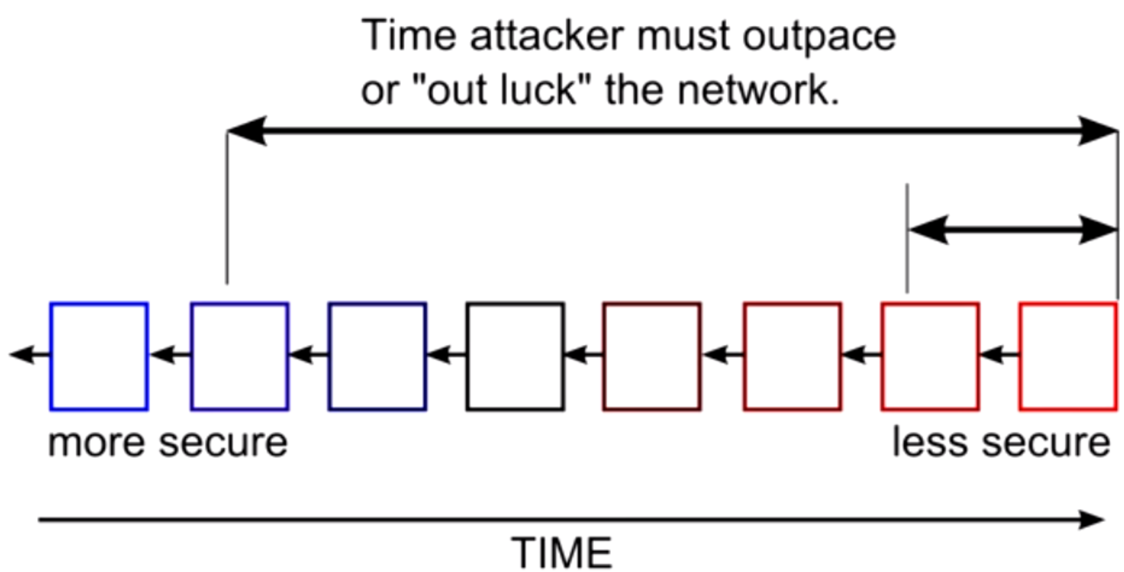
- Furthermore, it is a way to distribute coins into circulation
  - There is no central authority issuing new coins
  - Each crypto-currency platfrom will not erogate new coins to miners forever → currency deflation
    - Is effective to mine whilst you earn more money than those spent for electric power

How the PoW Ensures Integrity: Computational Power

- **Impossible to change a txn in a block *b* in the blockchain:** an attacker should be quicker than the rest of the whole network to mine a block.
- In that case he could be able to re-mine *n+1* blocks (i.e. all blocks next to *b* +1) quicker than the rest of the network to mine a new block.
- If so, the attacker could obtain the longest (modified) blockchain and all network would converge to it.
- But for doing that, he would have the 50% of the computational power of the network to have a 50% probability to solve a block before another node.
- And he would have a higher percentage of computational power to solve more blocks sequently.

How Blockchain Ensures Integrity: Computational Power

- Last blocks are so less secure
- Wait for 5/6 blocks makes a success probability too low for an attacker
- This solution protects from both **Integrity** and **Double Spending Fraud**



Alternative to PoW?

- **PoW pro:** very secure
- **PoW cons:** waste of electric power

- **Proof-of-Stake (PoS)** is the PoW alternative
  - **Secure without mining → no energy wasted**

Proof-of-Stake

- Instead of mine a block, the creator of the next block is chosen in a deterministic way according to its wealth (i.e. stake)
- The reward are not related to the created block but according to your wallet
- The longer you keep the coin in the wallet, the more the reward is high
- The probability to ***mint*** (instead of mine) a block is proportional to your wallet
  - **Minting** process require a lot of coin to attack the network
  - If you have a the *p*% of coins of the network you will mint the *p*% of the blocks
  - Very difficult to mint two consecutive blocks!