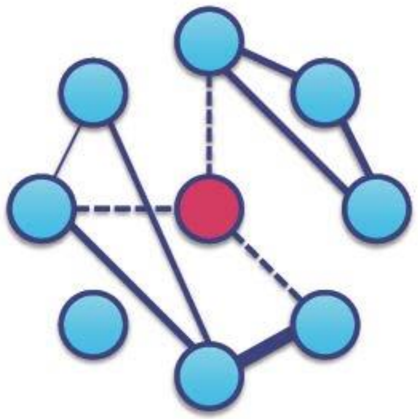
ASCI Blockchain 2024

Blockchain consensus

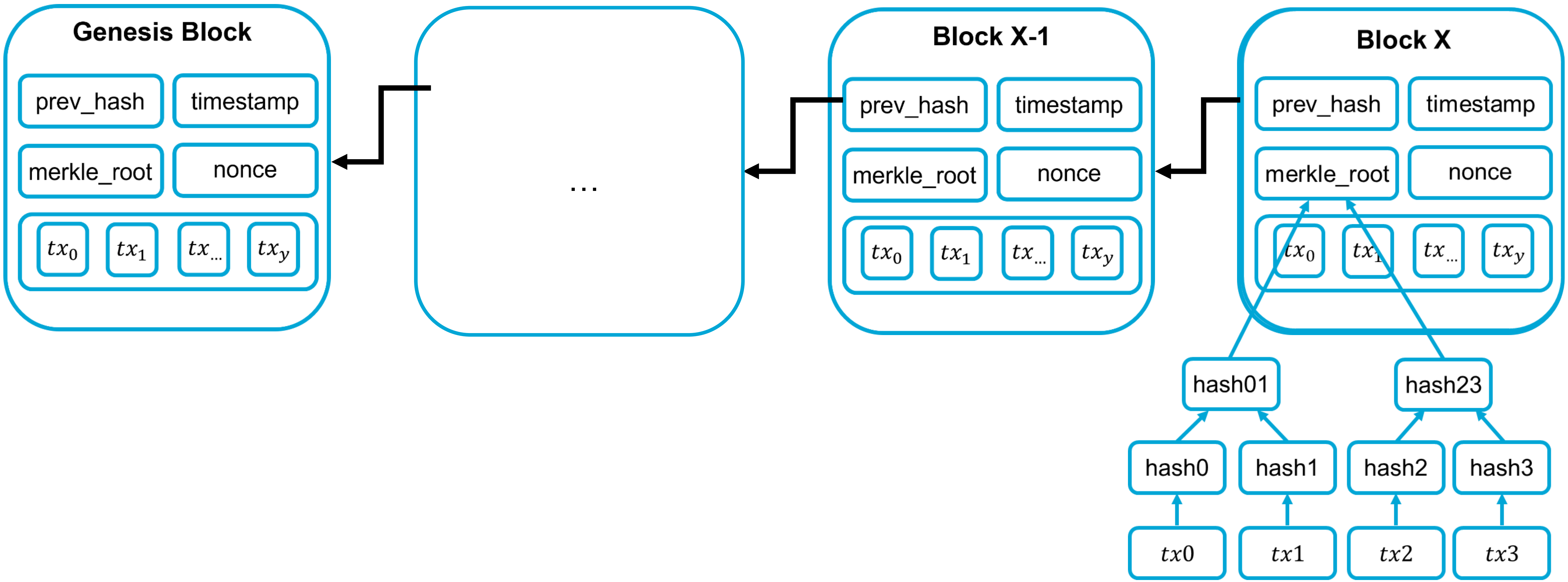
Jérémie Decouchant
j.decouchant@tudelft.nl

2023-2024



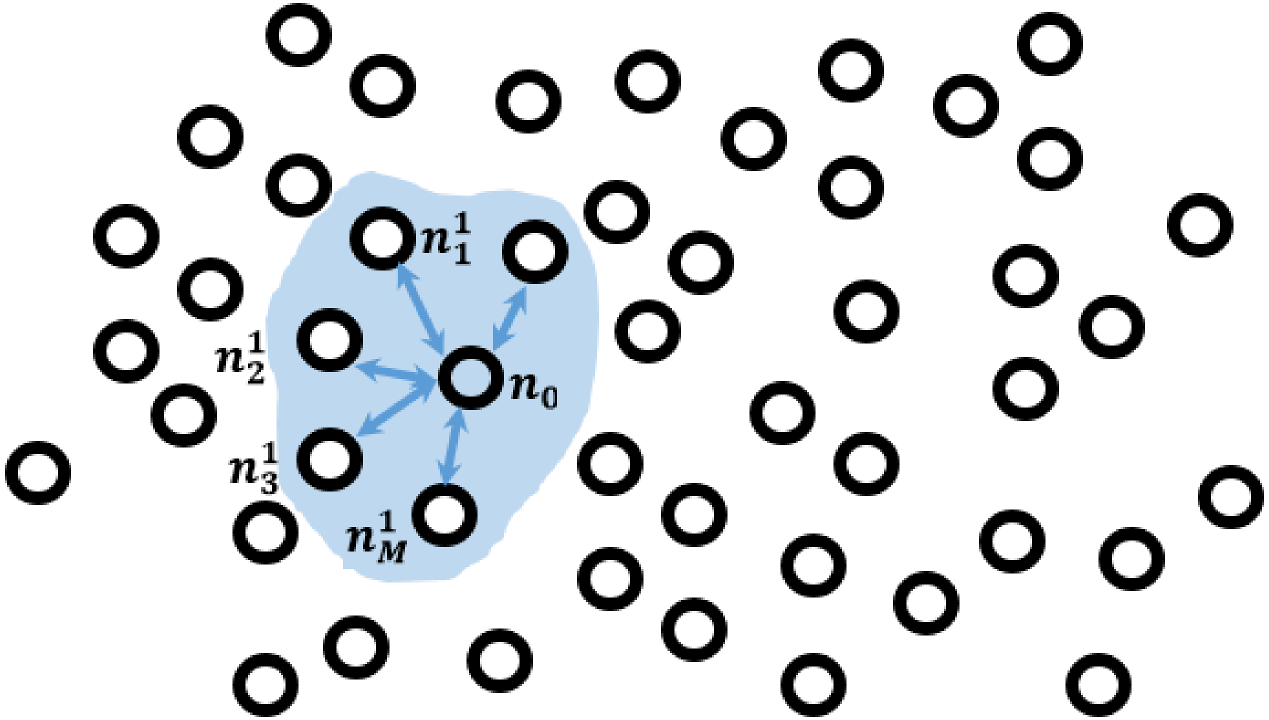
TU Delft

Consensus in blockchains



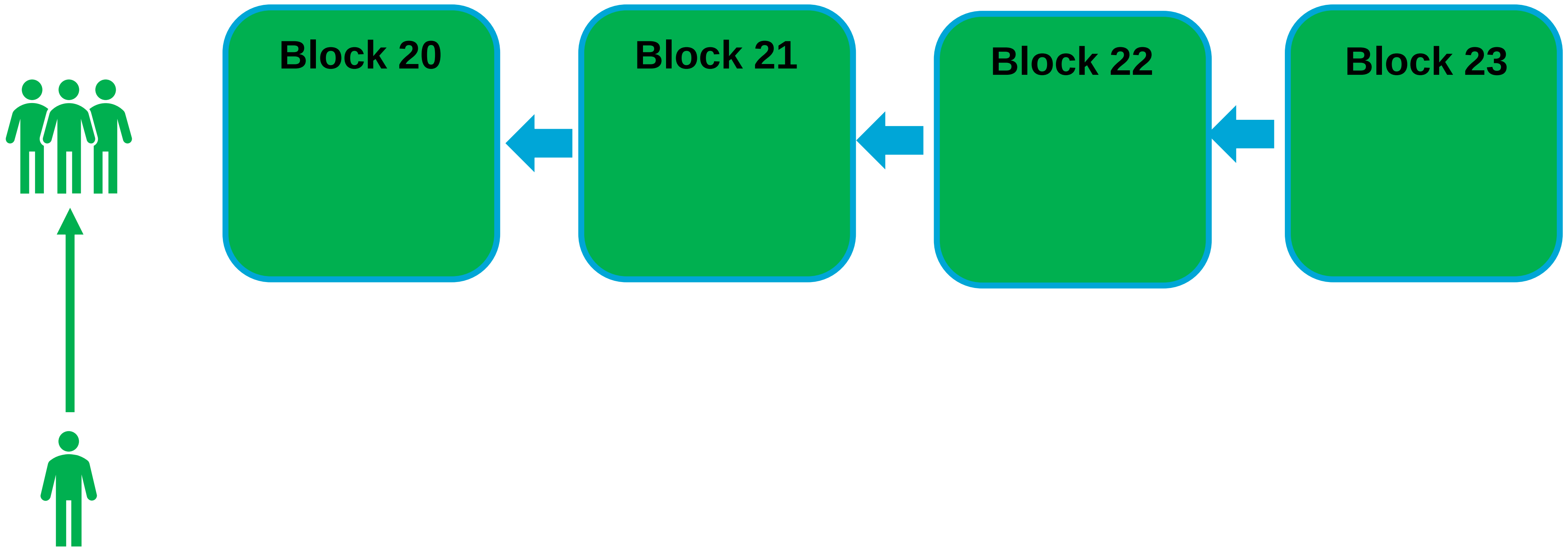
TU Delft

Summary



Liveness

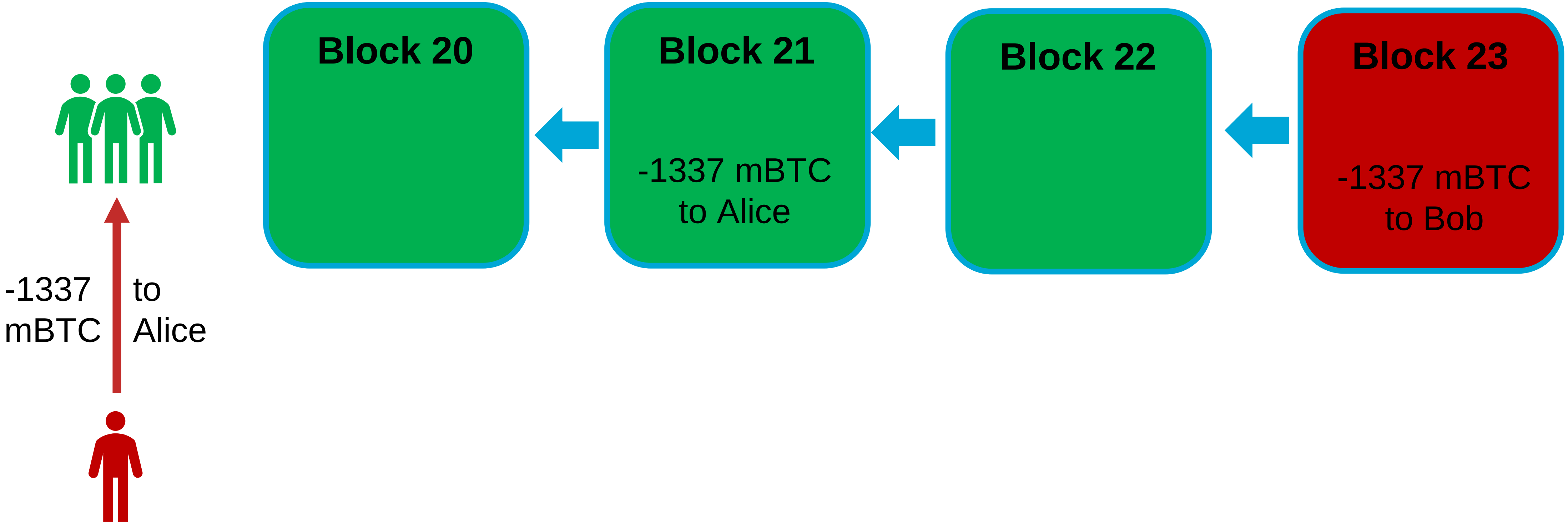
All transactions are eventually processed.



TU Delft

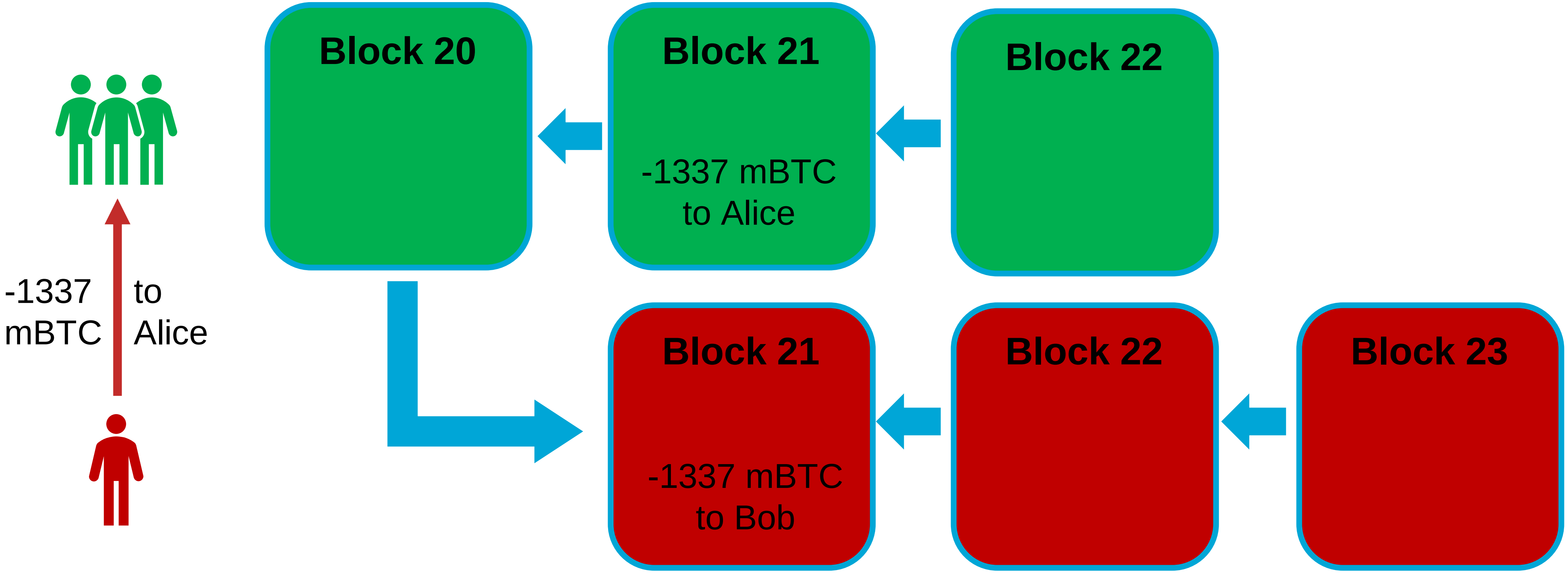
Safety: Case 1

A correct user never executes conflicting transactions.



Safety: Case 2

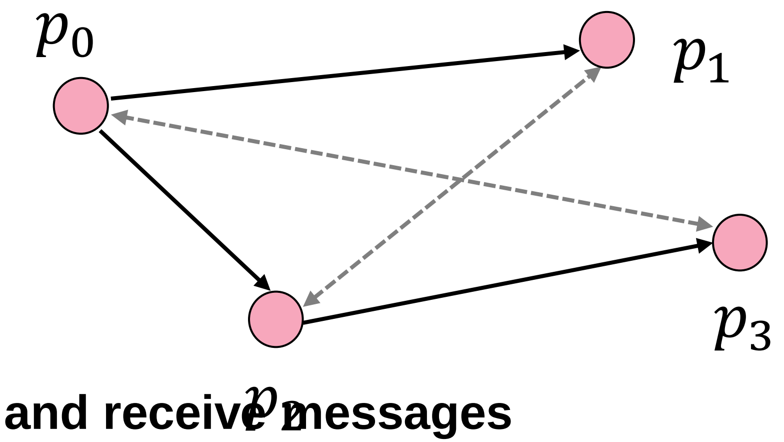
Two correct users never executes conflicting transactions.



From consensus to agreement

TUDelft

Network



- A distributed system runs on top of a graph:
  - A vertex hosts a **process** that can do **local computations**
  - An edge is a **communication channel** where processes can **send and receive messages (generally bidirectional)**

- **Synchrony model**: synchronous, partially-synchronous, or asynchronous (more on that later)
- The network is often assumed to be **connected** sufficiently often
  - Any two processes can eventually communicate
  - Messages can be lost, delayed or tampered with

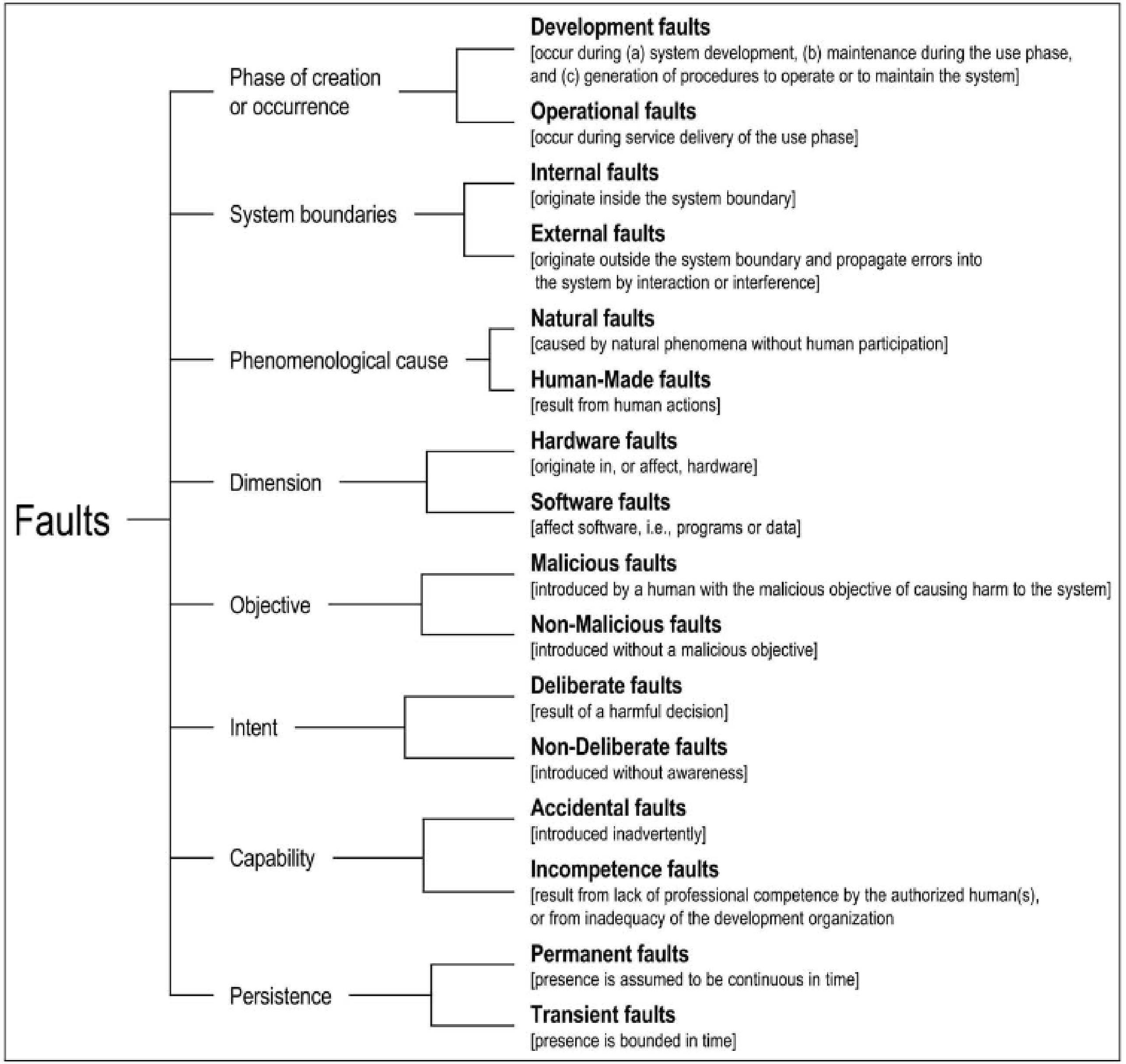
N.B.: we generally use node, process and host indistinctively

Nodes

- The system consists of honest nodes and of a limited proportion of faulty nodes.
- **Correct nodes** always follow a specified protocol
- **Byzantine nodes** can deviate arbitrarily from a protocol
  - due to hardware or software faults
  - or because of a malicious adversary
- Consensus algorithms sometimes assume that nodes might **crash**
- In consensus algorithms, we often focus on:
  - Omission faults: not sending a message
  - Equivocation: sending conflicting messages to different nodes

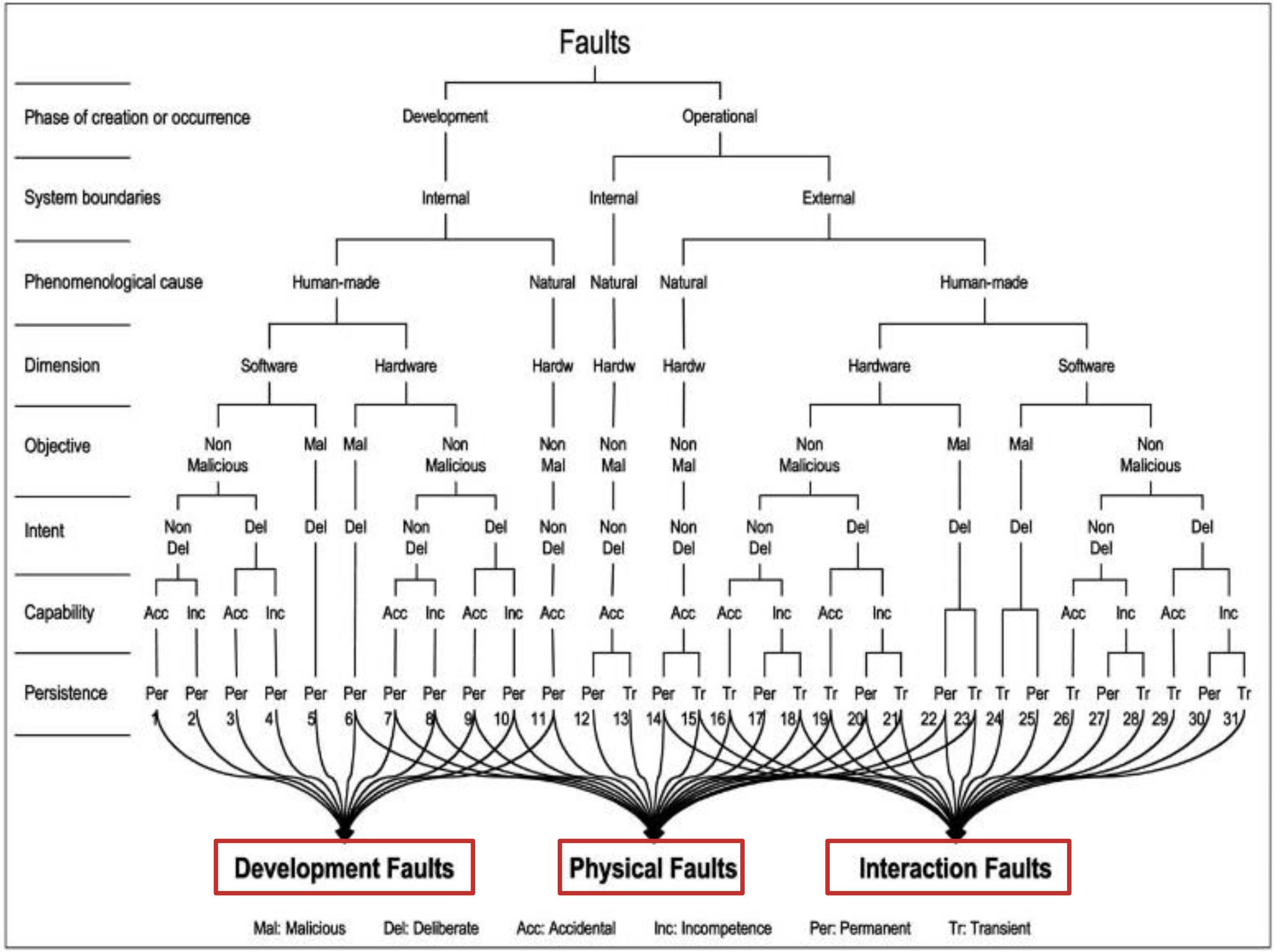
Elementary fault classes

Basic Concepts and Taxonomy of Dependable and Secure Computing. Avizienis, Laprie, Randell and Landwehr, IEEE TDSC, 2004



Tree representation of fault classes

Basic Concepts and Taxonomy of Dependable and Secure Computing.
Avizienis, Laprie, Randell and Landwehr, IEEE TDSC, 2004



TU Delft

Malicious faults

Basic Concepts and Taxonomy of Dependable and Secure Computing. Avizienis, Laprie, Randell and Landwehr, IEEE TDSC, 2004

logic bomb: *malicious logic* that remains dormant in the host system till a certain time or an event occurs, or certain conditions are met, and then deletes files, slows down or crashes the host system, etc.

Trojan horse: *malicious logic* performing, or able to perform, an illegitimate action while giving the impression of being legitimate; the illegitimate action can be the disclosure or modification of information (attack against confidentiality or integrity) or a *logic bomb*;

trapdoor: *malicious logic* that provides a means of circumventing access control mechanisms;

virus: *malicious logic* that replicates itself and joins another program when it is executed, thereby turning into a *Trojan horse*; a virus can carry a *logic bomb*;

worm: *malicious logic* that replicates itself and propagates without the users being aware of it; a worm can also carry a *logic bomb*;

zombie: *malicious logic* that can be triggered by an attacker in order to mount a coordinated attack.

Fig. 7. Malicious logic faults.

TU Delft

Cryptographic assumptions

- Consensus algorithms have first been designed assuming authenticated links
  - i.e., a message received on a link has been sent by its announced sender
  - does not make assumption on the computational power of an adversary
  - Hardest settings: more complicated and less efficient solutions

- We consider that processes have access to:
  - An asymmetric encryption scheme
  - A signature scheme
  - A hash function

From permissioned to permissionless, and back

The first consensus algorithms were permissioned: a fix group of nodes run a protocol.

Permissioned

- Closed membership
- Deterministic finality
- Requires attacking 33%
- **High performance, but low scalability**

Permissionless

- Open membership
- High transparency
- Requires attacking 51%
- Probabilistic finality
- **Low performance, but high scalability**

Hyperledger

- Lead by IBM, supported by > 300 organizations
- Five major projects
  - Fabric – PBFT
  - Burrow
  - Sawtooth
  - Indy
  - Iroha - BChain

Formal definition of consensus

- A distributed computing abstraction with two functions: propose(v) and decide()
  - Each process has an initial value that it proposes from some set V.
  - All correct processes must decide a single value.

- **Termination:** every correct process eventually decides some value
- **Validity:** If a process decides v, then v was proposed by some process.
- **Integrity:** No process decides twice.
- **Agreement:** No two correct processes decide differently.

Termination and Agreement are the difficult ones

TU Delft

The FLP Impossibility

- Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *Journal of the ACM (JACM)* 32.2 (1985): 374-382.

- **Fundamental result**: there is no deterministic algorithm for solving consensus in asynchronous networks with at least one process that might crash.

- Algorithms have to circumvent this impossibility. How?
  1. Assume that the network will be synchronous at some point
  2. Use randomized algorithms

Understanding FLP

- Solving consensus becomes difficult when the network has periods of asynchrony, or when processes are Byzantine A

- Blockchains have to deal with both!

Seminal consensus algorithms

- **Synchronous network and crash faults:**
  - Trivial solution

- **Synchronous network and Byzantine faults:**
  - Lamport's OM and SM protocols: N > f, $O(N^{f+1})$ messages, f+1 latency

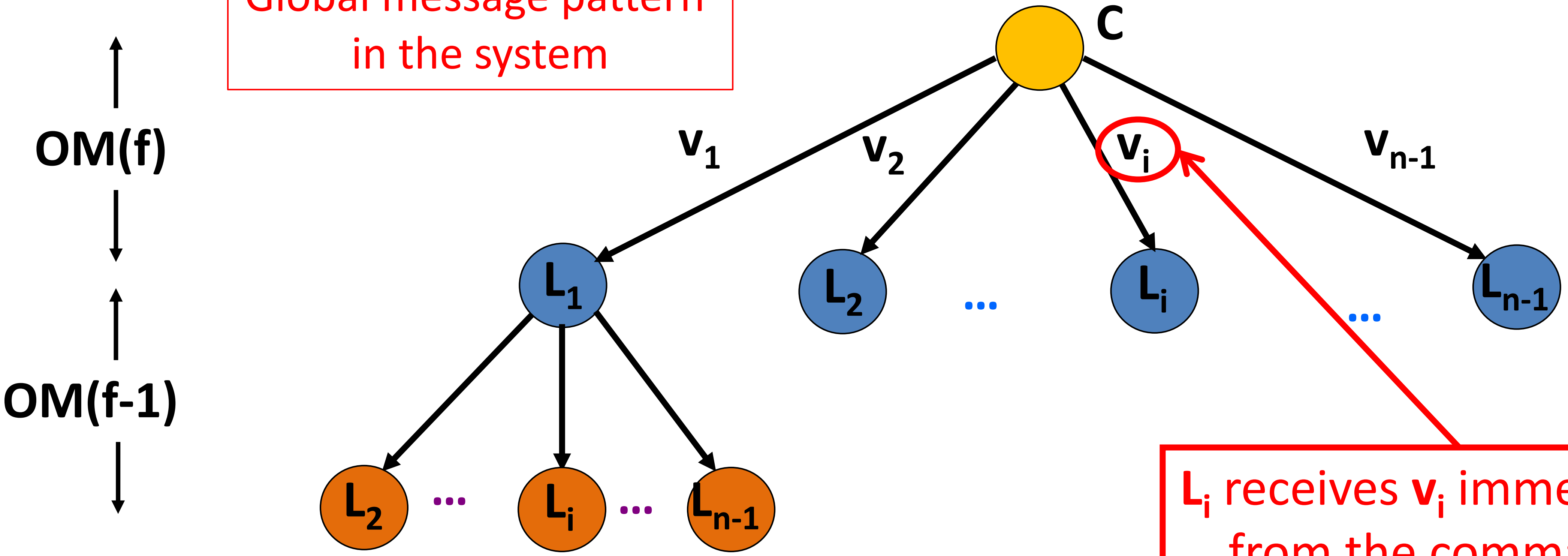
- **Asynchronous network and Byzantine faults:**
  - Ben-Or's randomized protocol: N > 3f+1, $O(n^2 . 2^N)$ messages, $O(2^N)$ latency

- Those protocols are very heavy. In practice, permissioned blockchains assume a partially synchronous model:
  - Maintain safety during asynchrony: N > 3f+1
  - Ensure liveness during synchrony

TU Delft

OM: Byz. Agreement in the Unauthenticated and Sync. Model

Global message pattern in the system



L_i receives v_i immediately from the commander

here L_i decides on **its own v'_1** as a lieutenant of L_1

degree in this tree is reduced by 1 in every next level

TU Delft

Randomized Byzantine agreement

notification phase

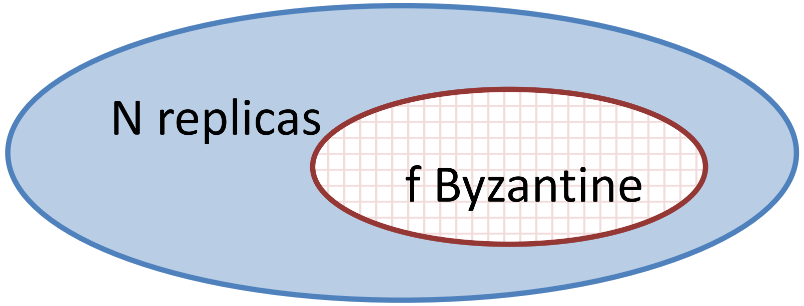
proposal phase

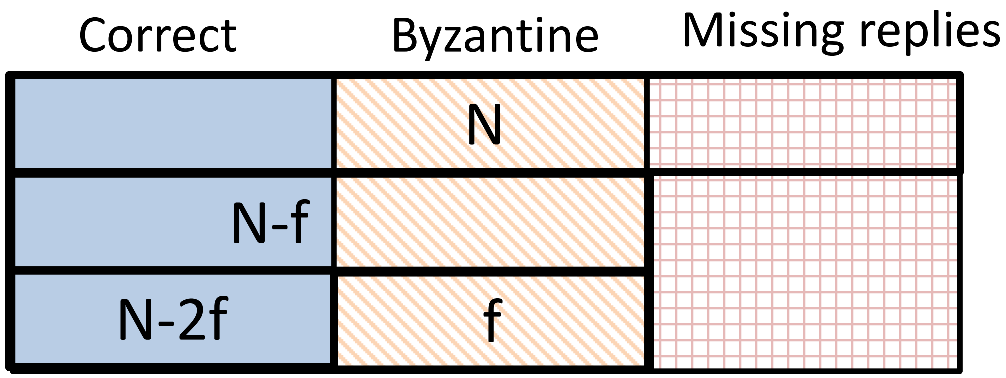
decision phase

```
r=1; decided:=false
do forever
    broadcast(N,r,v)
    await (n-f) messages of the form (N,r,*)
    if (>(n+f)/2 messages (N,r,w), w=0,1) then   /* enough support for a */
        broadcast(P,r,w)                         /* specific proposal 0 or 1 */
    else broadcast(P,r,?)            /* otherwise no proposal (don't know) */
    if decided then STOP
    else await (n-f) messages of the form (P,r,*)
    if (>f  messages (P,r,w), w=0,1) then
        v:=w
        if (>3f messages (P,r,w)) then
            decide(w)
            decided:=true
    else v:=random(0,1)
    r:=r+1
```

TU Delft

Number of replicas in the asynchronous model





Not all replies might arrive in a bounded amount of time

- Worst case: (N-f) values

Among those replies, f might be incorrect (Byzantine)

- Worst case: (N-f) – f equal answers

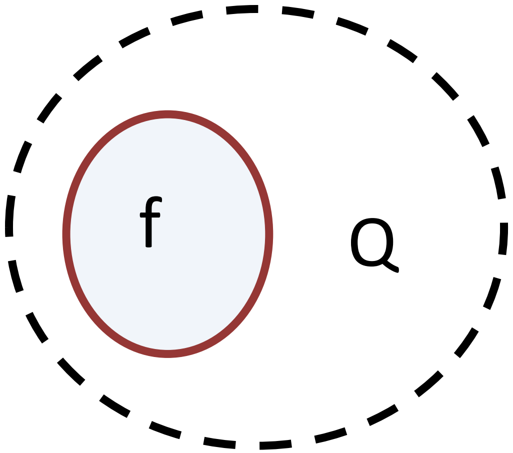
To be convinced that those answers are the right ones, we need

(N-f)-f > f.

$$N \geq 3f + 1$$

Q: Byzantine Quorum size

Decide that an object can only have value V upon receiving Q equal answers.
What value is possible for Q?

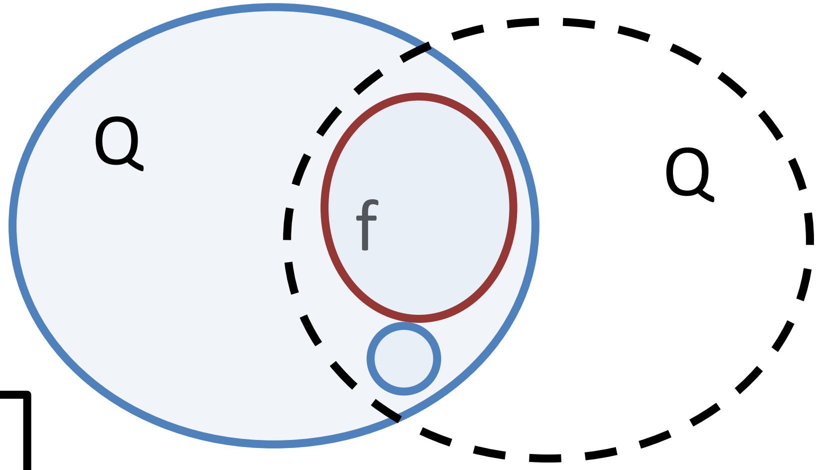


There must be at least Q correct replicas (liveness):

$$Q \leq N - f$$

Any two sets of Q+ replicas must intersect in at least 1 correct replica (safety):

$$2Q - (f + 1) \geq N$$



$$Q \geq \frac{N + f + 1}{2}$$

TUDelft

Agreement

The consensus abstraction assumes that all processes propose a value.
In practice, blockchains implement agreement, a variant of consensus.

- One node starts with a binary value. Each of the remaining nodes decide a binary value.
  - **Termination:** every correct process eventually decides a value
  - **Validity:** If the source is correct, then all correct processes agree on the value it proposed.
  - **Agreement:** All correct processes agree on the same value
  - **Integrity:** No correct process decides twice.

N.B.:

- If the source is faulty, the correct processes can **agree on any value.**
  - It is irrelevant on what value **a faulty process** decides.
- This problem is also called Terminating Reliable Broadcast.

Equivalence between consensus and agreement

- **Assume that we can solve agreement.**
  - For consensus, each node proposes a value
  - We run an agreement protocol for each node to agree on the value it proposed
  - We can chose the majority outcome to all agree on a value (consensus)

- **Assume that we can solve consensus:**
  - For agreement, one node N broadcasts a value.
  - Nodes can wait a limited amount of time, and propose the value they have received from N to each other (or a default value otherwise)
  - Using consensus, we can all agree on the same final value (agreement).

From agreement to State Machine Replication

2
9

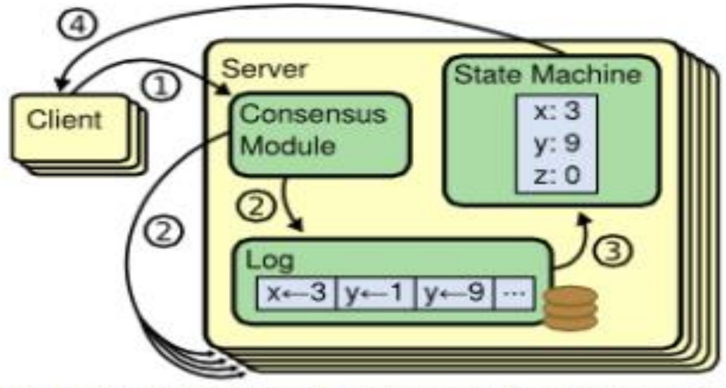
From agreement to State Machine Replication

- With agreement, nodes can agree on a single (binary) value

- We need more to build a distributed ledger:
  - Interaction with clients
  - Need to agree on a sequence of values and on their order

- State Machine Replication is the abstraction that provides this functionality

State Machine Replication (1/2)



- **Fault-free** centralized operation
  - a single server maintains a **state machine** (e.g., a data store)
  - clients issue **requests** to the server (e.g., reading and writing)
  - the server **serializes** and executes the requests
- In the **face of faults** or **poor performance**
  - replicate the server: **State Machine Replication** (SMR)
  - have the replicas execute **the same client requests in the same order**
  - so servers have to **achieve consensus** on the log of client requests

State Machine Replication (2/2)

- Potential types of failures:
  - stopping / pausing processors
  - malicious (due to explicit attacks or hardware/software errors)
- Models are usually assumed to be **asynchronous**
  - sometimes weaker timing assumptions
  - may lead to livelock
- Four seminal algorithms:
  - **Paxos** (crash-recover faults)
  - **Raft** (crash-recover faults)
  - **PBFT** (Byzantine faults)
  - **Zyzzyva** (Byzantine faults)

TUDelft

From Consistent to Reliable Broadcast

Reliable

Consistent

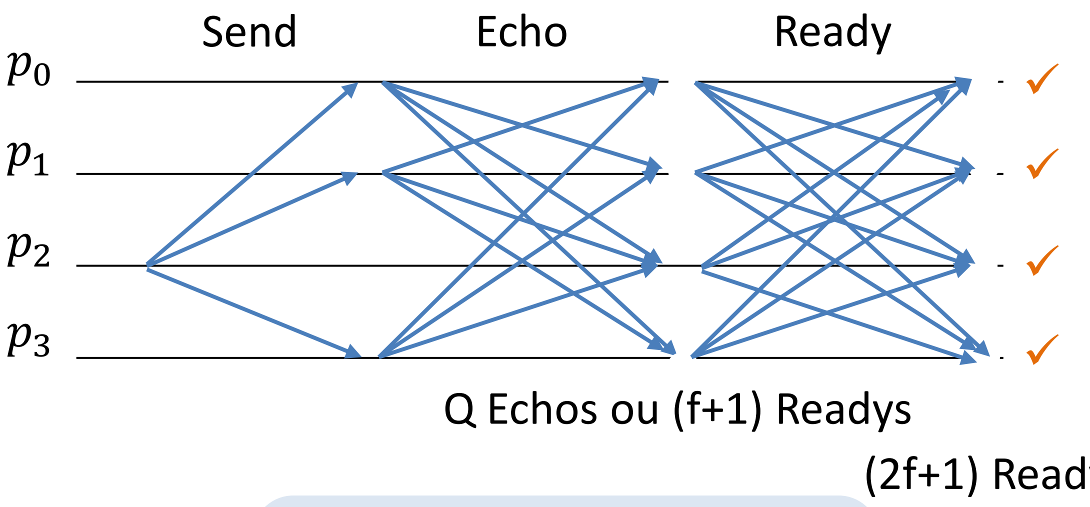
Validity: If a **correct** process p broadcasts m then all correct processes eventually deliver m.

No *duplication*: Every correct process delivers a message at most once.

Integrity: If a correct process delivers m with sender p, then m was broadcast by p.

Consistency: If a correct process delivers m and another correct process delivers m' then m=m'.

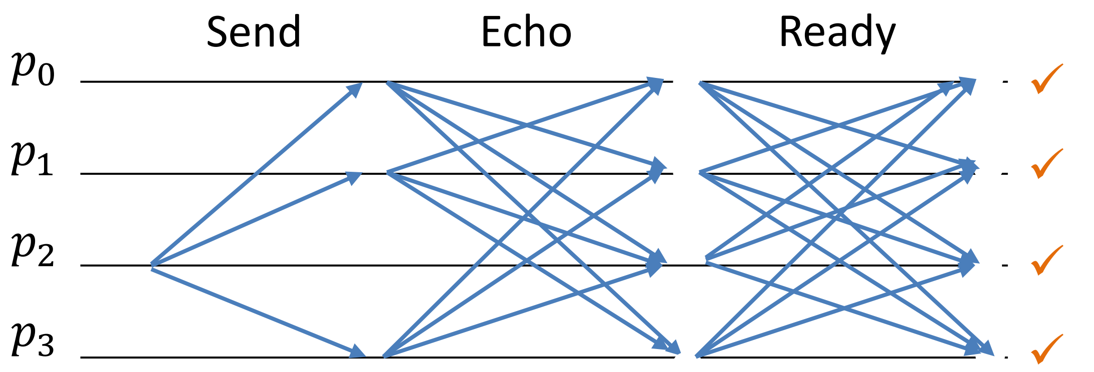
Totality*: If m is delivered by a correct process, then all correct processes eventually deliver m.*



With consistent broadcast, a Byzantine sender might cause only a subset of correct processes to deliver.

TU Delft

Reliable broadcast ($N \geq 3f + 1$)



Algorithm 7 (Bracha broadcast [Bra87]).

upon *r-broadcast*(m): // only P_s
  send message (SEND, m) to all

upon *receiving a message* (SEND, m) *from* P_s:
  send message (ECHO, m) to all

upon *receiving* $\lceil \frac{n+t+1}{2} \rceil$ *messages* (ECHO, m) *and not having sent a* READY *message:*
  send message (READY, m) to all

Ready amplication

upon *receiving* $t+1$ *messages* (READY, m) *and not having sent a* READY *message:*
  send message (READY, m) to all

upon *receiving* $2t + 1$ *messages* (READY, m):
  r-deliver(m)

TU Delft

Proof of totality

- If a correct party has r-delivered m, it has received a READY message with m from 2t+1 distinct parties.
- Therefore, at least t + 1 correct parties have sent a READY message with m, which will be received by all correct parties and cause them to send a READY message as well.
- Because $n - t \geq 2t + 1$, all correct parties eventually receive enough READY messages to terminate.

Total order broadcast: reliable broadcast + total order

Validity: If a correct process p broadcasts m then all correct processes eventually deliver m.

N*o duplication*: Every correct process delivers a message at most once.

Integrity: If a correct process delivers m with sender p, then m was broadcast by p.

Agreement: If a message m is delivered by some correct process, then m is eventually delivered by all correct process.

Total order: Suppose that p and q are two correct processes that deliver m1 and m2. If p delivers m1 before m2, then q delivers m1 before m2.

TU Delft

TOB Broadcast is equivalent to Consensus

- Total-order Byzantine broadcast is also equivalent to Byzantine consensus.

TUDelft

PBFT (1/5): assumptions

- Handle **Byzantine node failures** of replicas
- **Adversary** cannot break collision-resistant hashes, encryption, signatures
- Clients may also be faulty
- Use message **digests** and **signatures**
- Provide **safety**: linearizability (does not depend on synchrony)
- Provide **liveness**: assume weak synchrony:
  - message delays grow at most linearly with time
  - system is synchronous for periods of time

TU Delft

PBFT (2/5): views and data

- At every moment, there is a **view**
  - one replica is the **primary**
  - the other replicas are **backups**
  - view number **v** has primary **p = v mod n** (predetermined)
  - when the primary supposedly fails, change view

- Replica **data structures**
  - state machine
  - view number
  - message log
  - checkpoints

PBFT (3/5): similarities

- **Algorithm structure**
  - agreement protocol
  - checkpoint protocol
  - view-change protocol
- **Checkpoints**
  - maintain history
  - stable checkpoints: truncate history

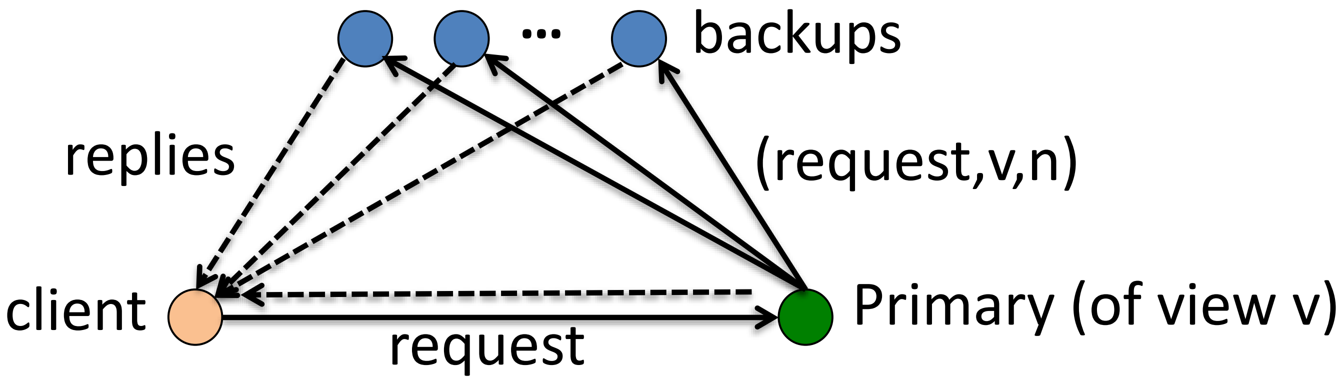
PBFT (4/5): differences

- **PBFT:**
  - achieves consensus on request order with a 3-phase protocol among replicas
  - “a correct server only emits replies that are stable”
- **Speculative protocols (Zyzzyva, and others):**
  - faster speculative execution with larger burden on the clients
  - “a correct client only acts on replies that are stable”

TU Delft

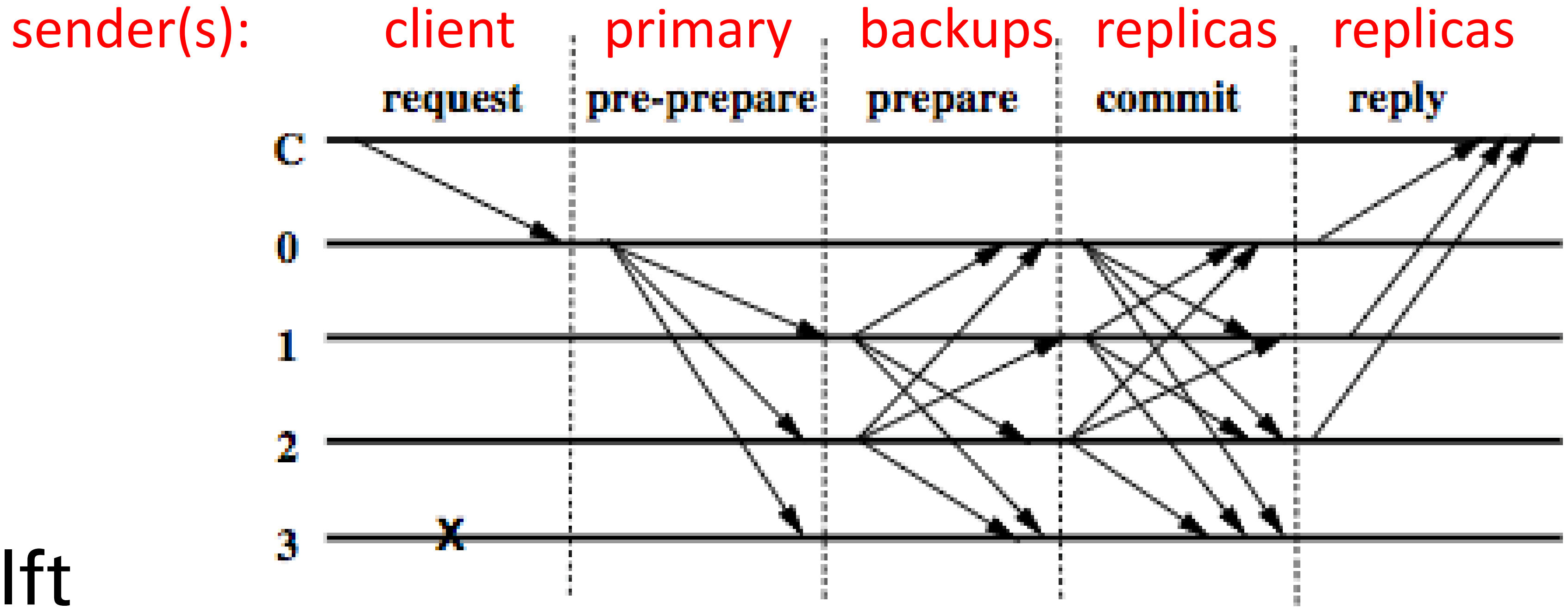
PBFT (1/8): outline

1. **Client sends request** to the primary (with logical time stamp)
2. **Primary assigns sequence number and broadcasts request** to backups
3. **Replicas** execute the request and **reply to the client**
4. **Client waits for f+1 replies** with the same result



PBFT (2/8): normal operation

- Normal operation = primary does not fail
- **Three-phase** protocol (three types of messages):
  - **pre-prepare + prepare phases**: totally order requests in the **same view**
  - **prepare + commit phases**: totally order requests **across views**
- All three types of messages contain a view number and a request number



PBFT (3/8): accepting a pre-prepare

- A backup **accepts** a pre-prepare message if:
  - it is in the same view
  - it has not accepted a pre-prepare with the same view and sequence number
- It then enters the prepare phase and **broadcasts a prepare message**
- The **predicate** prepared(m,v,n,i) is true if **replica i** has entered into its message log:
  - the request
  - the corresponding pre-prepare message
  - **2f** corresponding prepare message from other backups (Byz quorum)
- **Assertion**: if prepared(m,v,n,i) is true for a correct replica **i**, then prepared(m',v,n,j) is false for any m≠m' and any correct **j**

TUDelft *unique request in same view with same sequence number across replicas*

PBFT (4/8): commit

- When prepared(m,v,n,i) is true, replica **i** broadcasts a **commit message**
- Predicate committed(m,v,n) is true if prepared(m,v,n,i) is true in at least **f+1** correct replicas
- Predicate committed-local(m,v,n,i) is true if prepared(m,v,n,i) is true and replica **i** has accepted **2f+1** commit messages (then it **executes** the request)
- **Assertion:** if committed-local(m,v,n,i) is true in some correct replica **i**, then committed(m,v,n) is true
- **Consequences:**
  - correct replicas agree on the sequence numbers of requests even if they commit locally in different views
  - a request that commits locally at a correct replica, does so in at least **f+1** correct replicas (any Byz. quorum intersects with this set)

TUDelft

PBFT (5/8): checkpoints

- **Checkpoint:**
  - state after the execution of a fixed multiple of K requests
- **Stable checkpoint:**
  - a checkpoint with a “proof”
- Replicas broadcast **checkpoint messages** with the **sequence number** of the last request represented in the checkpoint plus the digest of the state
- **Proof of correctness of a checkpoint:**
  - **2f+1** matching checkpoint messages
- Upon a checkpoint becoming stable, **discard history**:
  - discard previous checkpoints and checkpoint messages
  - discard all messages related to earlier requests

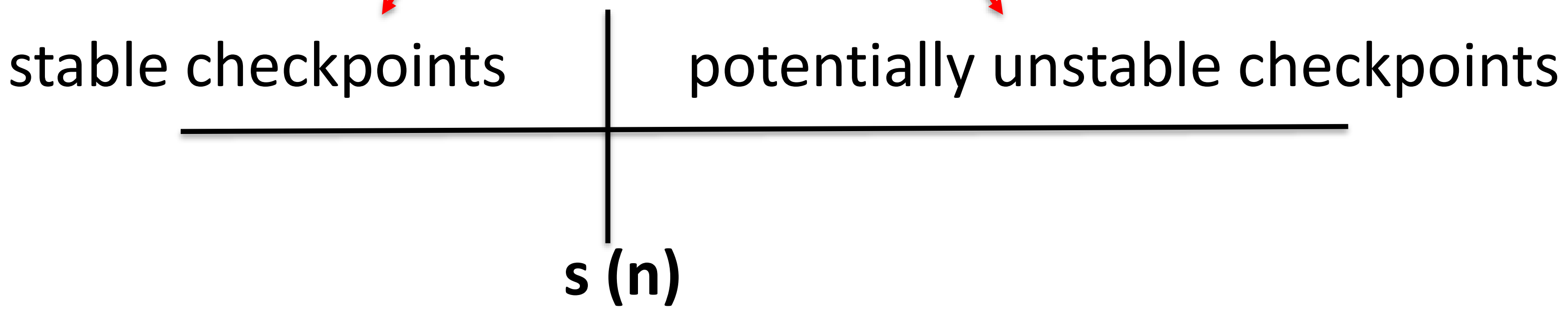
PBFT (6/8): overview of view change

- If a client **does not receive f+1** identical replies soon enough, it broadcasts its request to all **replicas**
- A replica then
  - re-sends its reply to the client, if it has already processed the request
  - otherwise it sends the request to the primary
- If the primary then does not broadcast the request to the backups, it is **suspected of failure** by the replicas
- The backups then **initiate a view change**
- **The new view is announced** by the new primary

TUDelft

PBFT (7/8): view change

- When in view **v** the timer of a backup expires, it **broadcasts** a **view-change message with parameters**:
  - the new view number **v+1**
  - the **sequence number n** of the **last stable checkpoint s** it knows
  - a set of **2f+1** checkpoint messages proving the correctness of **s**
  - for every request prepared at the backup with request number higher than **n**, the corresponding pre-prepare message and **2f** prepare messages ("the message log after the last stable checkpoint")



PBFT (8/8): new view

- When the primary of view **v+1** receives **2f** view-change messages, it broadcasts a **new-view message with parameters**:
  - the **new view** number **v+1**
  - the **set of view-change messages** it has received
  - a **set of pre-prepare messages** derived from the view-change messages received to cause requests that may be missing at some replicas to be executed
- The primary then enters view **v+1**
- When a backup **receives a new-view message**, it catches up:
  - it derives from the pre-prepare messages in it and from its own message log on which of these messages it still has to act
  - it may have to retrieve requests or checkpoints from other replicas

TUDelft

Optimizing PBFT

- Use MAC instead of signatures
- Batch requests
- Use weighted voting (PoS?)
- Etc.

- But the message pattern is what is really limiting performance.

Wheat [Sousa and Bessani, SRDS 2015]

Some nodes have a better network than others: let them accelerate the decision process.

- $N = 3f + 1 + \Delta$: number of nodes
- $N_v = \sum V_i = 3F_v + 1$: sum of all the votes, F_v votes can be discarded
- $Q_v = 2F_v + 1$: quorum weight

- Binary weight distribution: either V_{max} (for u fast nodes) or V_{min}
- $N_v = \mathrm{uV_{max}} + (\mathrm{N} - \mathrm{u})\mathrm{V_{min}}$
- $F_v = (\Delta + f)V_{min} = fV_{max}$
- $V_{max} = \frac{\Delta+f}{f}V_{min}$

- With $V_{min} = 1$, $F_v = (\Delta + f)$, $V_{max} = \frac{\Delta+f}{f} = \frac{\Delta}{f} + 1$, and $u = 2f$
- A minimal quorum needs $2f + 1$ votes and more than Q_v weight.

TU Delft

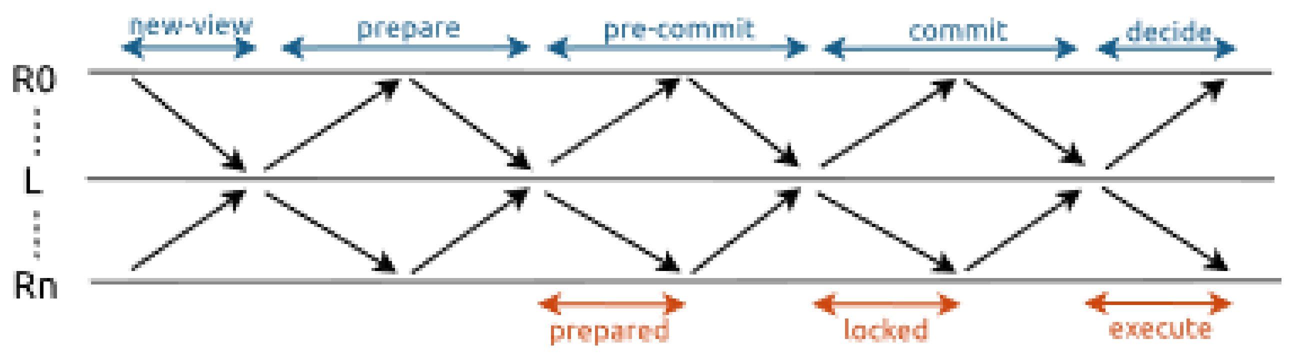
Performance of PBFT

- $N \geq 3f + 1$
- 3 network latencies to commit a message
- $O(N^2)$ message complexity
- View-change is expensive: $O(N^2)$ messages

- Limited scalability with the number of nodes
- Large number of messages = limited throughput

HotStuff: Pipelining

Figure 1 Communication phases in HotStuff.



- Linear communication pattern
- Rotating leader: no view change required
- Network latency: from 3 to 8
- Higher throughput
- Pipelining

TU Delft

Mir-BFT: Multi-leader

- Requests are affected to buckets

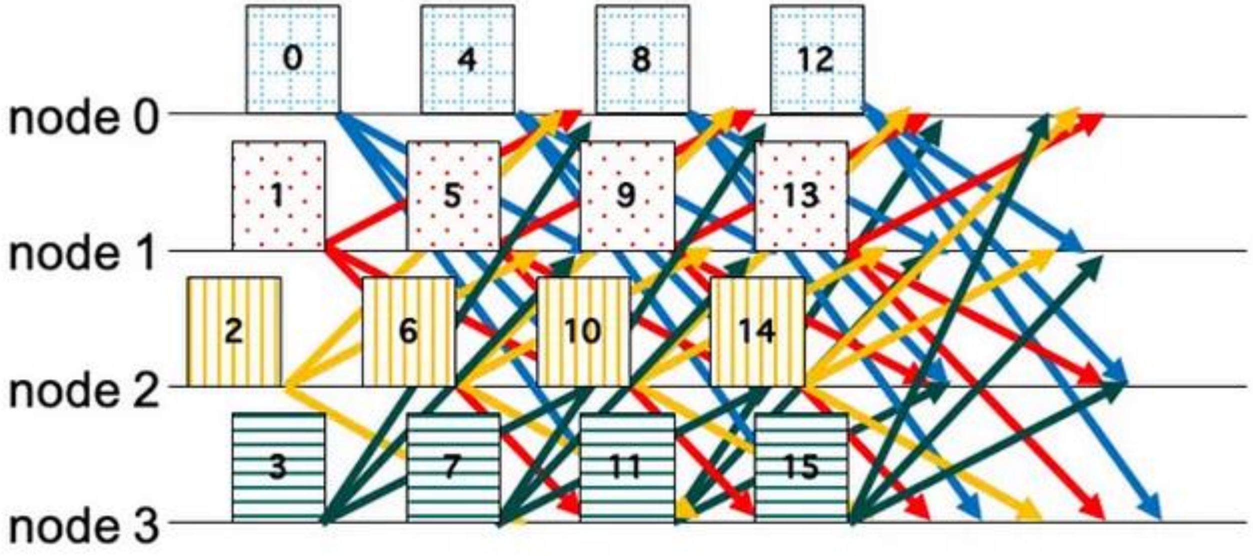
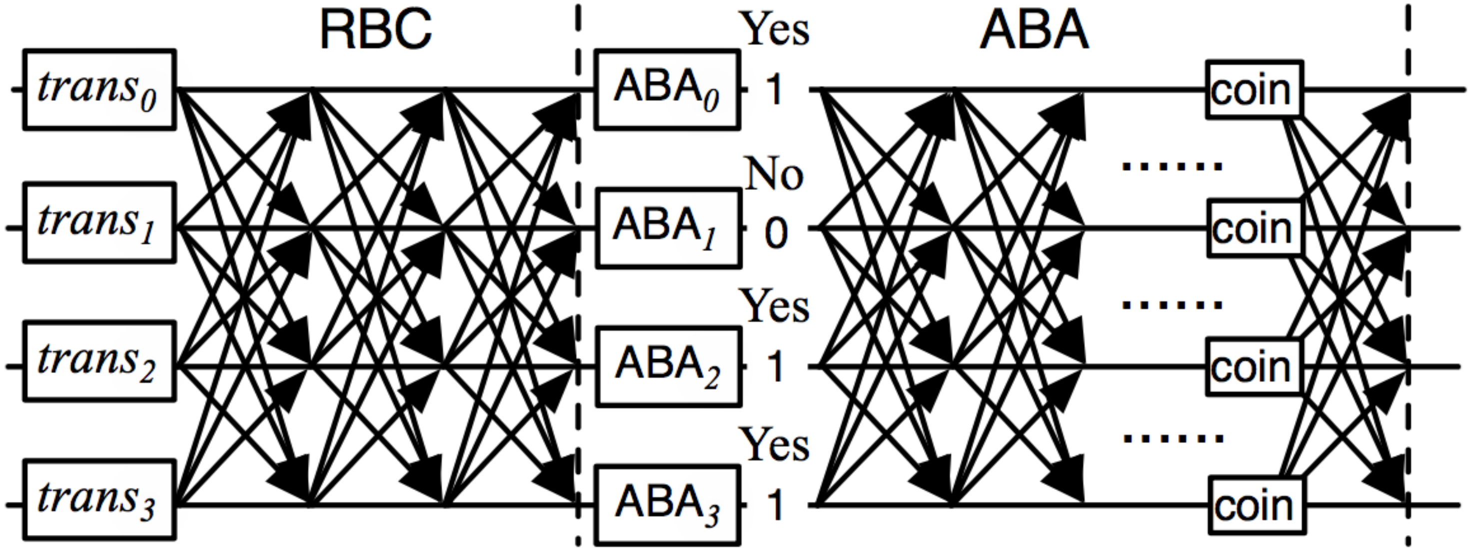


Figure 3: PRE-PREPARE messages in an epoch where all 4 nodes are leaders balancing the proposal load. Mir partitions batch sequence numbers among epoch leaders.

HoneyBadgerBFT
[Miller et al., CCS 2016]

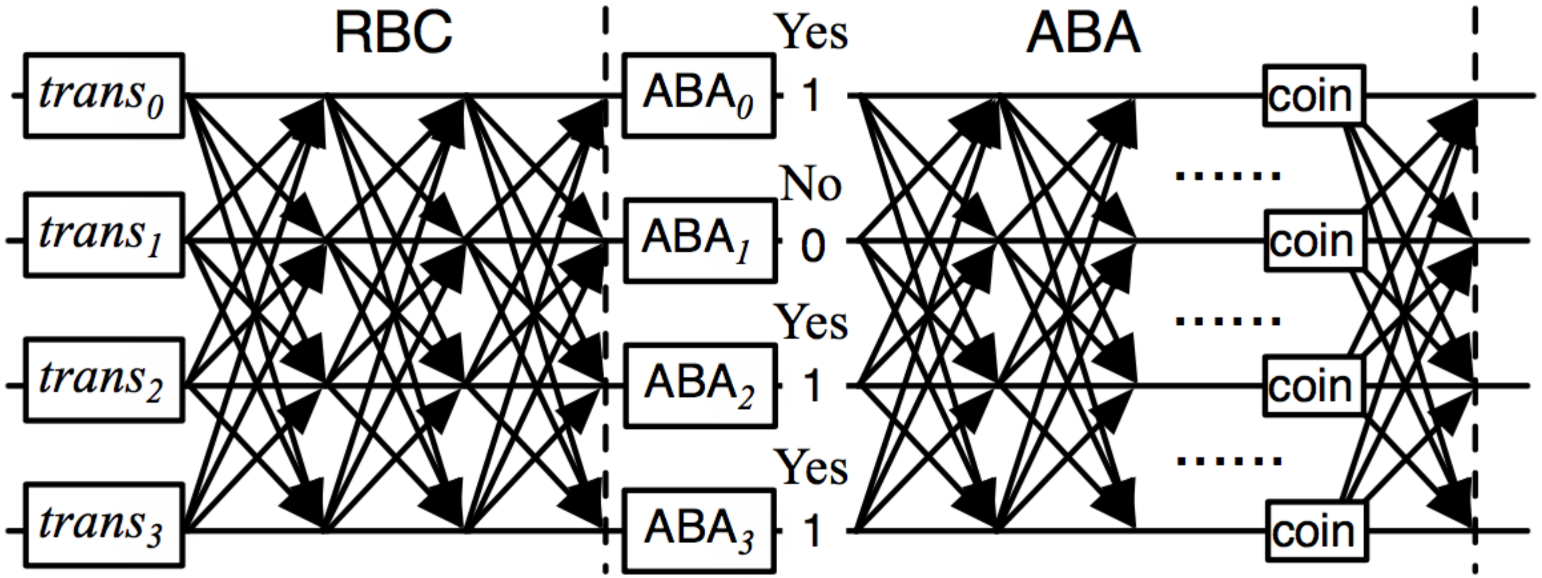
- Implements total order using **Asynchronous Common Subset (ACS)** [Ben-Or et al., PODC 1994; Cachin et al., CRYPTO 2001]

- Implements ACS, in turn, using **Reliable broadcast (RBC)** and **asynchronous binary Byzantine agreement (ABA)**



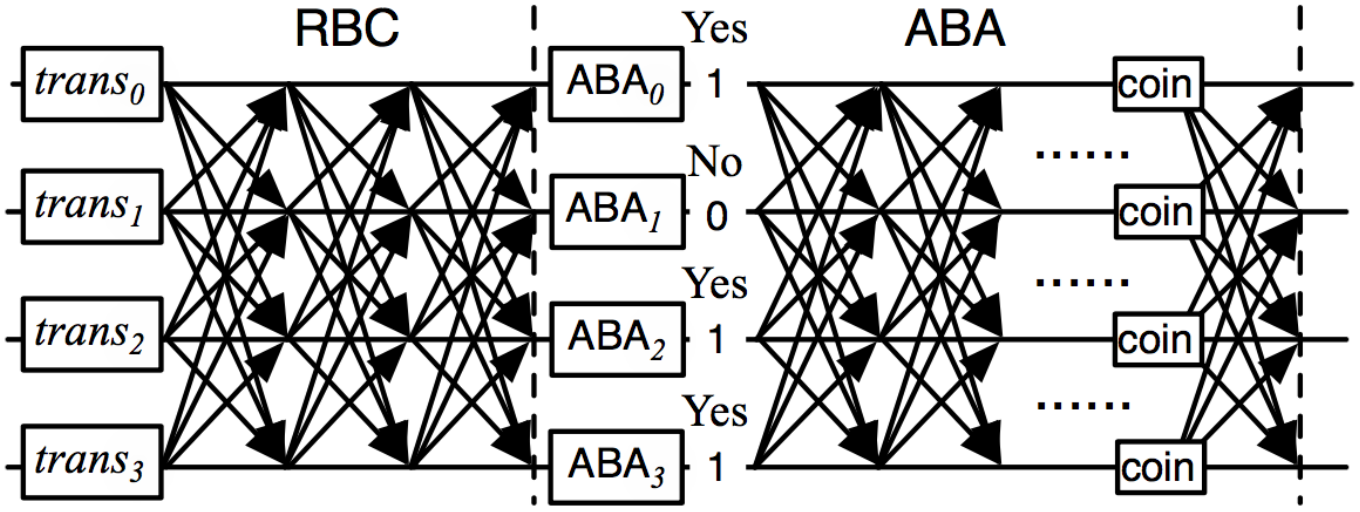
Asynchronous Common Subset (ACS)

- The goal
  - Every node proposes some transactions
  - Agree on the superset of all the proposed transactions



Asynchronous Common Subset (ACS)

- RBC: Reliable broadcast
  - Every node proposes some transactions
  - Randomly from the transaction pool
- ABA
  - Agreement on the proposed transactions by each node
  - N parallel ABAs



Other scalability techniques

- Hierarchical consensus
  - Steward, by Amir, Yair, et al. "Scaling byzantine fault-tolerant replication towide area networks." *DSN.* IEEE, 2006.
  - My Infocom 2024 paper
- Partitions/Sharding
  - Eyrie/Volery
  - Bezerra, Carlos Eduardo, Fernando Pedone, and Robbert Van Renesse. "Scalable state-machine replication." *DSN*. IEEE, 2014.
- Trusted components
  - Require 2f+1 instead of 3f+1 replicas, and less communication phases
  - Damysus, Eurosys 2022.

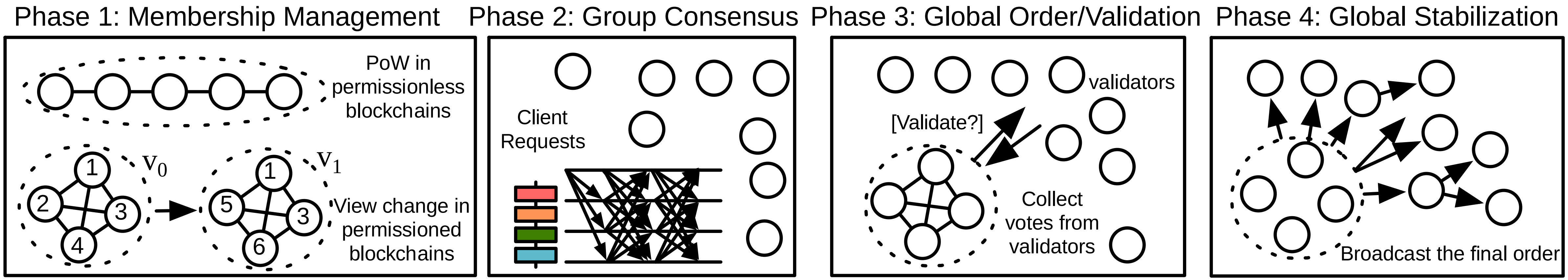
Why hybrid blockchains?

- Permissionless
  - Open network (anyone can join)
  - Server scalability (large number of servers)
  - Bad performance (poor client scalability, long latency)
- Permissioned
  - Relatively closed network (need to know the identities of all the nodes)
  - Good performance (large number of concurrent clients, low latency)
  - Poor server scalability
- Hybrid blockchains
  - Combine both and enjoy the benefits of both
  - But it is challenging!

Hierarchy vs partition-based SMR

- Number of nodes that are involved
  - Hierarchy: all the nodes still need to learn the results
  - Partition: only those nodes that are involved in the relevant partitions
- Total order of requests
  - Hierarchy: yes and straightforward
  - Partition: only order those requests that might create conflicts…
- Bottleneck
  - Hierarchy: group communication
  - Partition: operations that involve multiple partitions

An overview


Phase 1: Membership Management
PoW in permissionless blockchains
1
2
3
4
v_0
1
5
3
6
v_1
View change in permissioned blockchains
Phase 2: Group Consensus
Client Requests
Phase 3: Global Order/Validation
validators
[Validate?]
Collect votes from validators
Phase 4: Global Stabilization
Broadcast the final order