Practical Byzantine Fault Tolerance

Castro, Miguel, and Barbara Liskov. "Practical Byzantine fault tolerance." In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 173-186. 1999.

Presented in: CS 3551 Advanced Topics in Distributed Information Systems

Presented by: Amy Babay

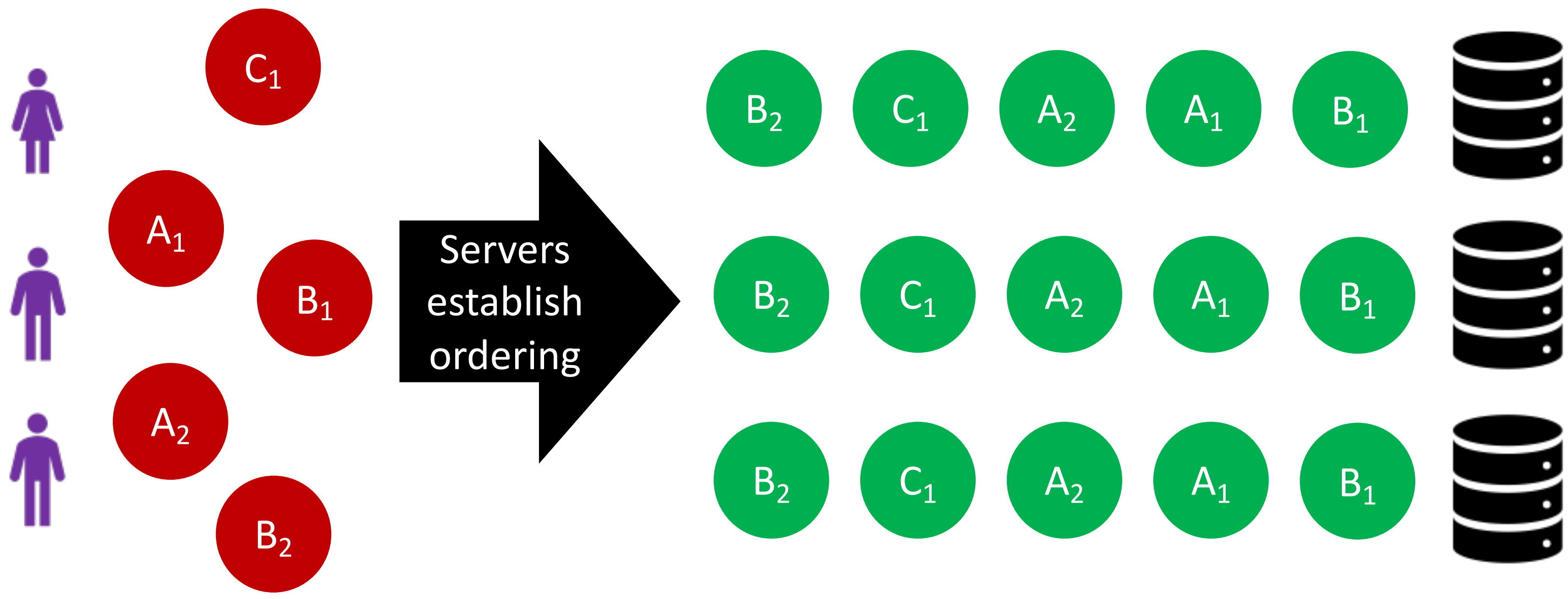
University of Pittsburgh

Recall: State Machine Replication

- **State Machine Replication (SMR)**: technique for implementing strongly consistent fault-tolerant services
  - Servers start in the same state
  - Servers apply deterministic updates in the same order
  - => Servers progress through exactly the same sequence of states

Recall: State Machine Replication

- **State Machine Replication (SMR)**: technique for implementing strongly consistent fault-tolerant services



Clients generate updates

Servers apply updates in order

System Model

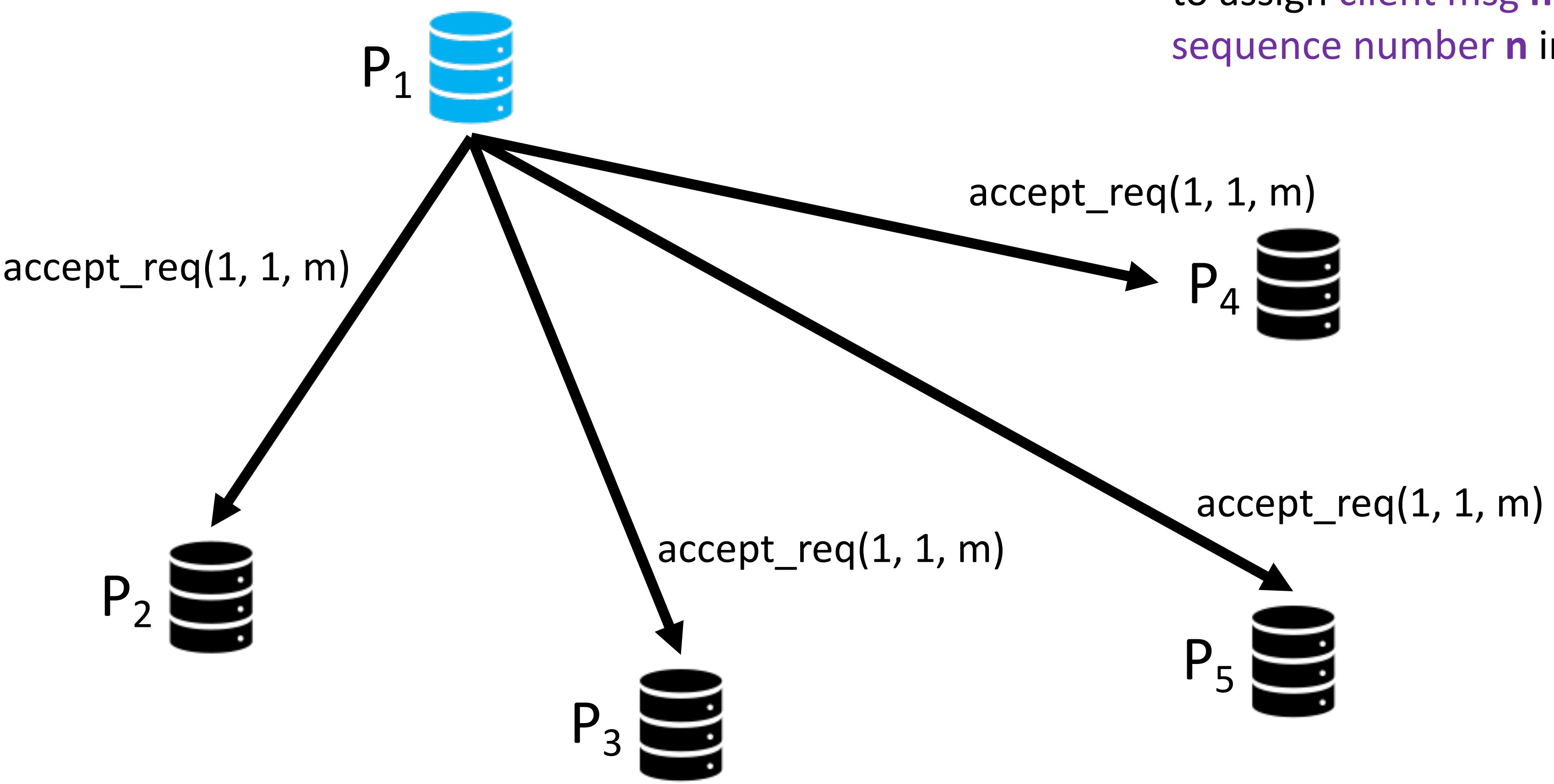
- *n* replicas
- Asynchronous network
  - Messages can be dropped, delayed, duplicated, delivered out of order
- Byzantine failures
  - Faulty replicas can behave arbitrarily (e.g. crash, recover, lie, collude, corrupt messages, delay messages)
- But,
  - At most f replicas are faulty, where $3f + 1 <= n$
  - Messages are authenticated (digital signatures) and faulty replicas are computationally bounded (cannot break crypto)

Recall: Paxos (SMR with benign failures)

- Key concept: Any two majorities must intersect in at least one replica
- So, to guarantee agreement we:
  1. Only allow a value to be *chosen* if it is accepted by a majority of replicas
  2. Require new leader to communicate with a majority of replicas before proposing a value to find out what they've previously accepted

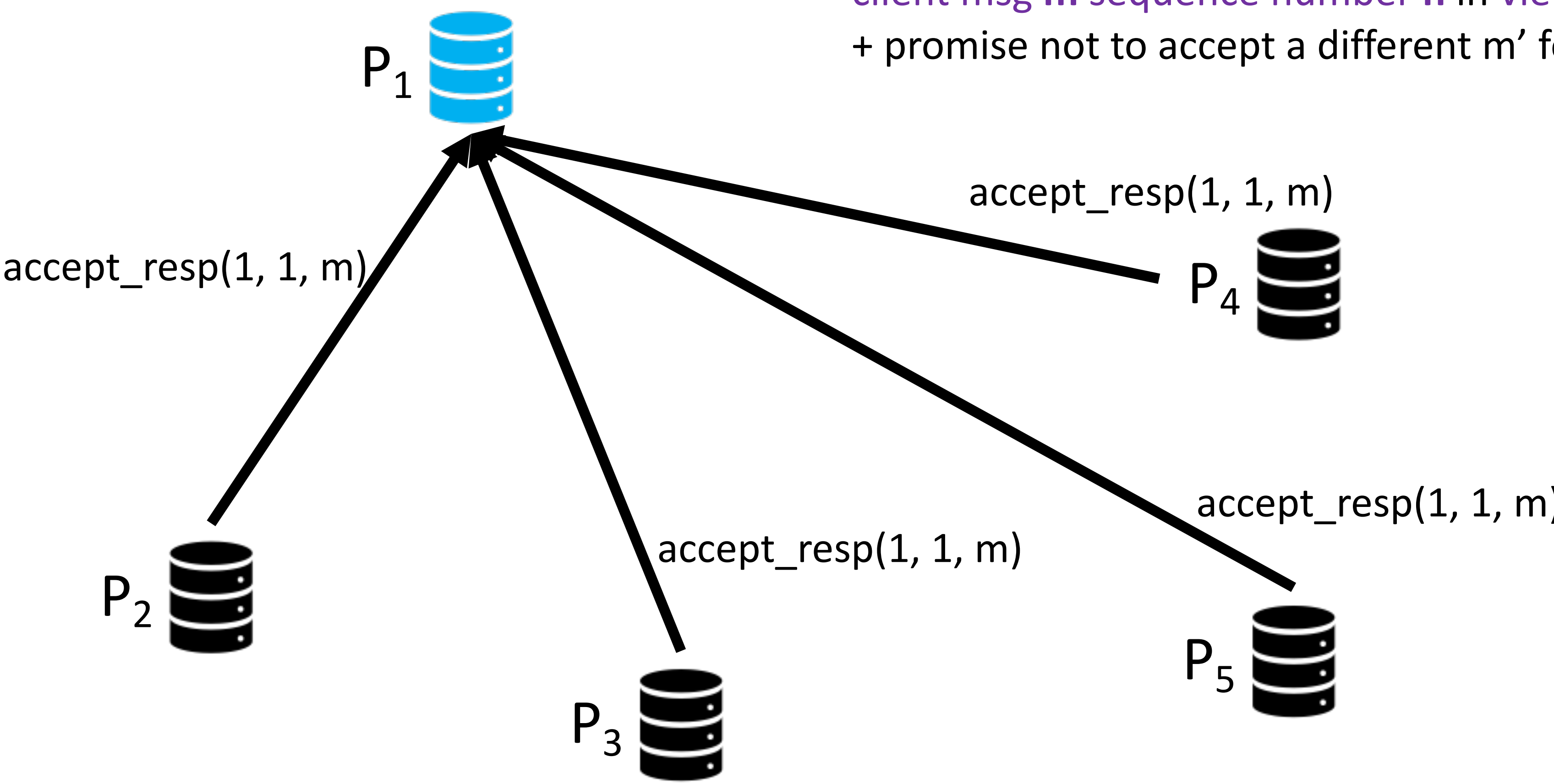
Recall: Paxos

accept_req(v, n, m): proposal to assign client msg **m** sequence number **n** in view **v**



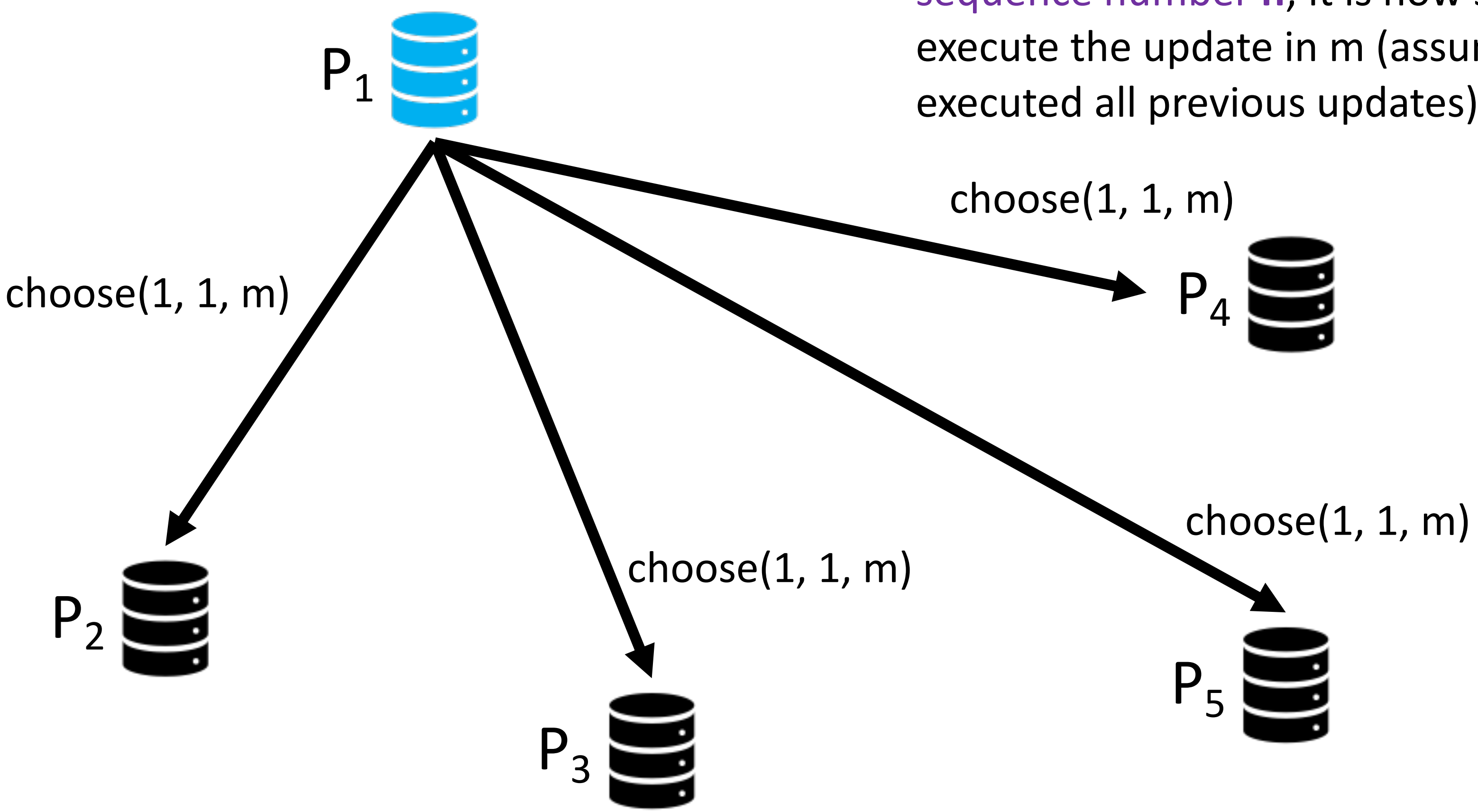
Recall: Paxos

accept_resp(v, n, m): agreement to assign client msg **m** sequence number **n** in view **v**
+ promise not to accept a different m' for n in v



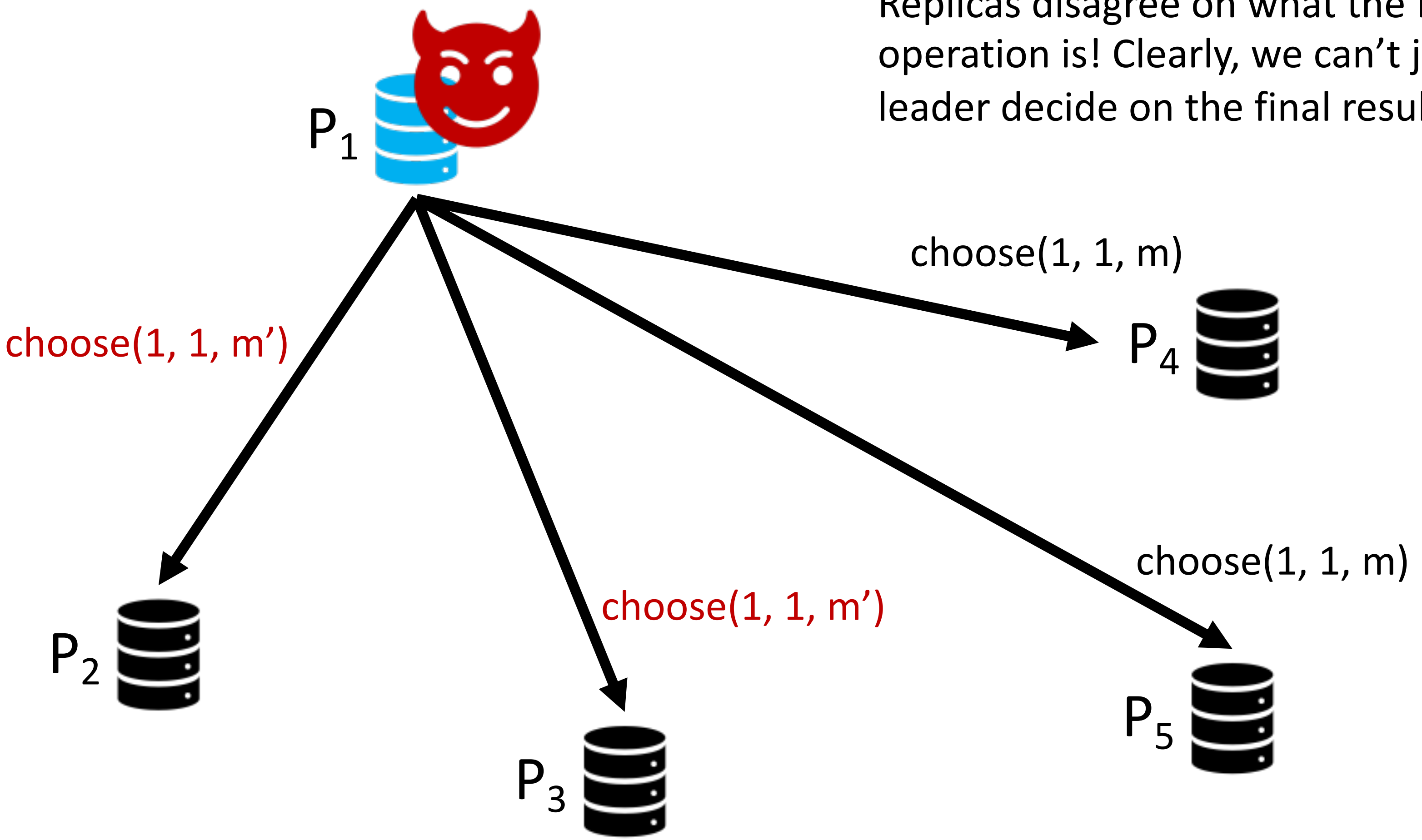
Recall: Paxos

choose(v, n, m): assign client msg **m** sequence number **n**; it is now safe to execute the update in m (assuming you've executed all previous updates)



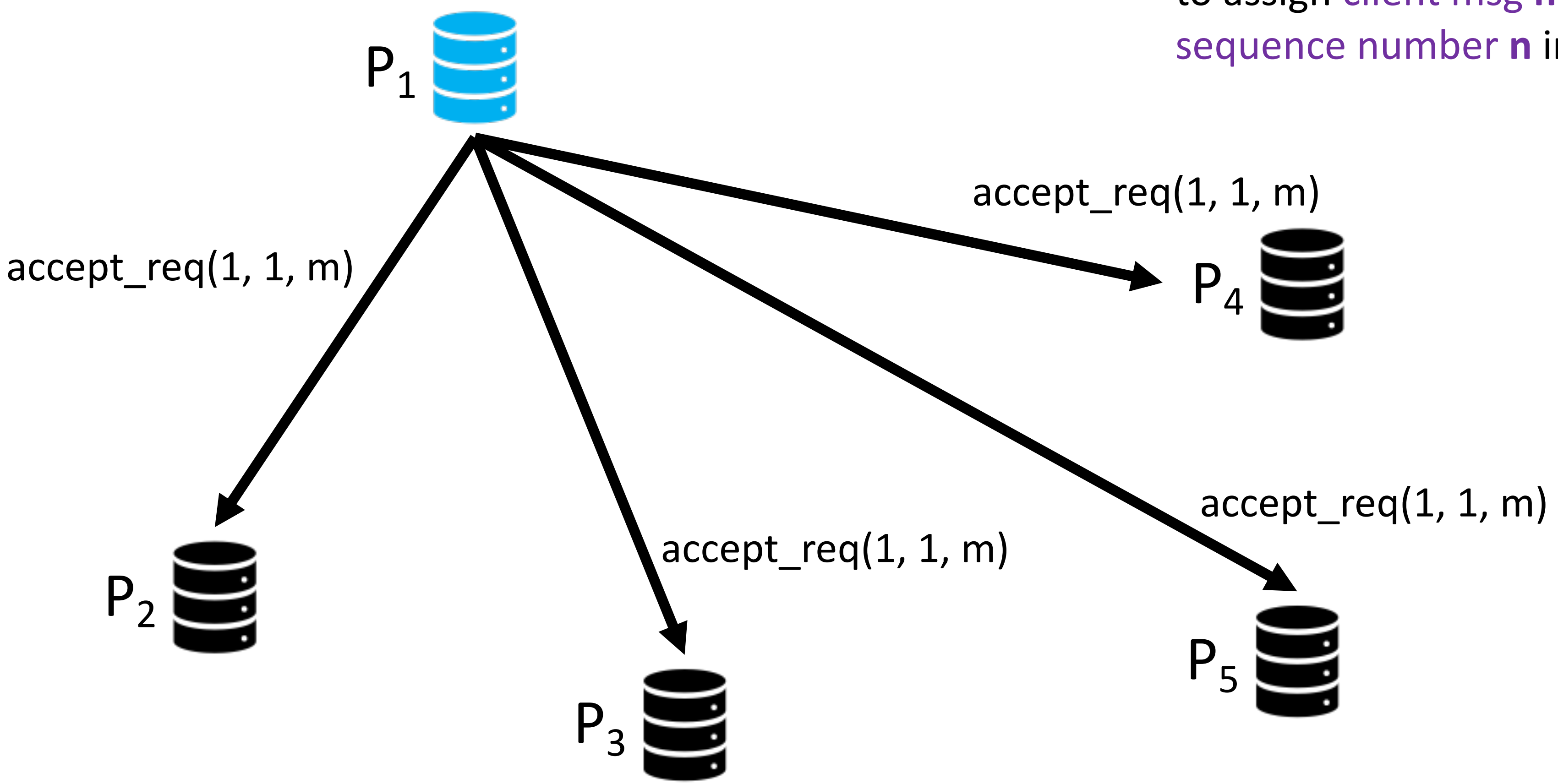
Paxos – What if our leader lies??

Replicas disagree on what the first operation is! Clearly, we can't just let the leader decide on the final result…



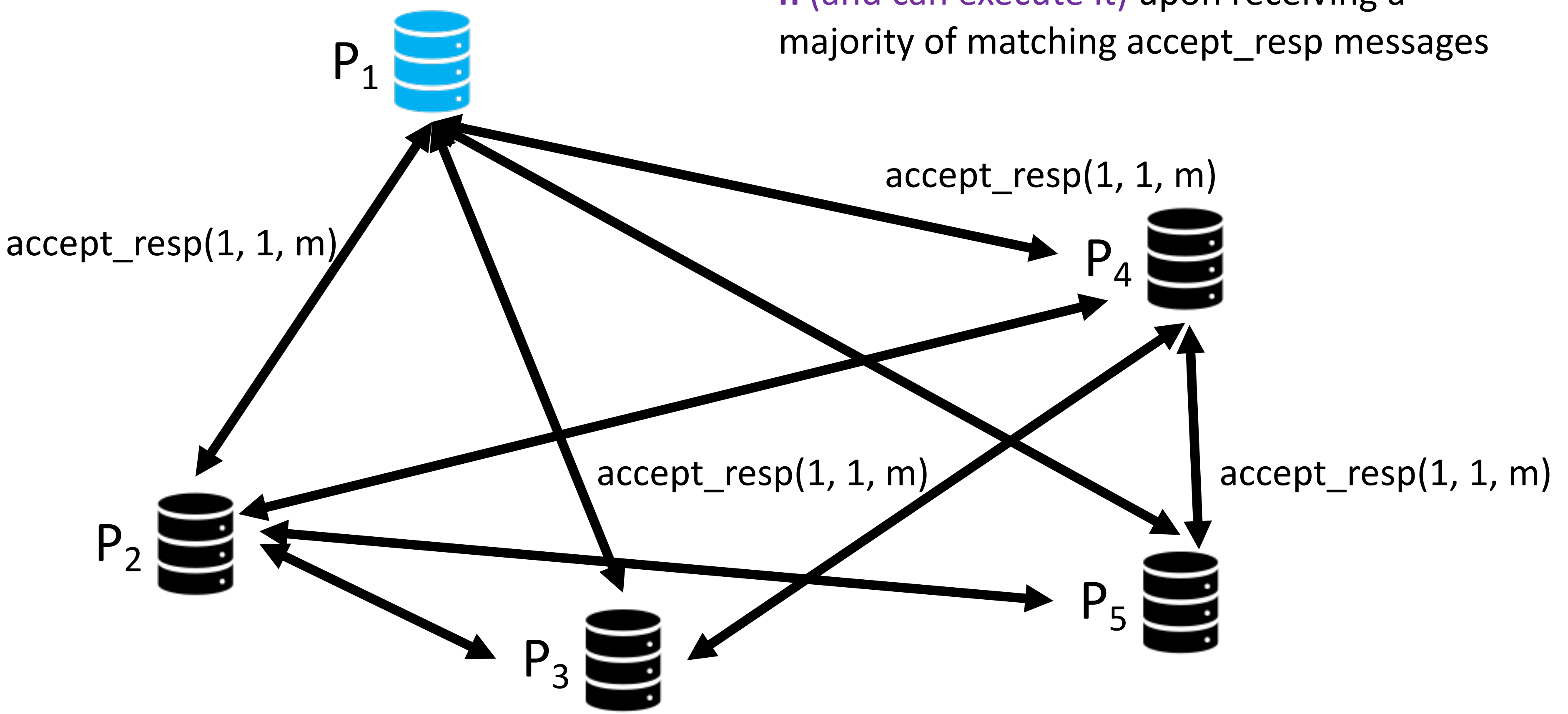
Paxos: Consider our other variant

accept_req(v, n, m): proposal to assign client msg **m** sequence number **n** in view **v**



Paxos: Consider our other variant

replicas assign client msg **m** sequence number **n** (and can execute it) upon receiving a majority of matching accept_resp messages



Paxos: Consider our other variant

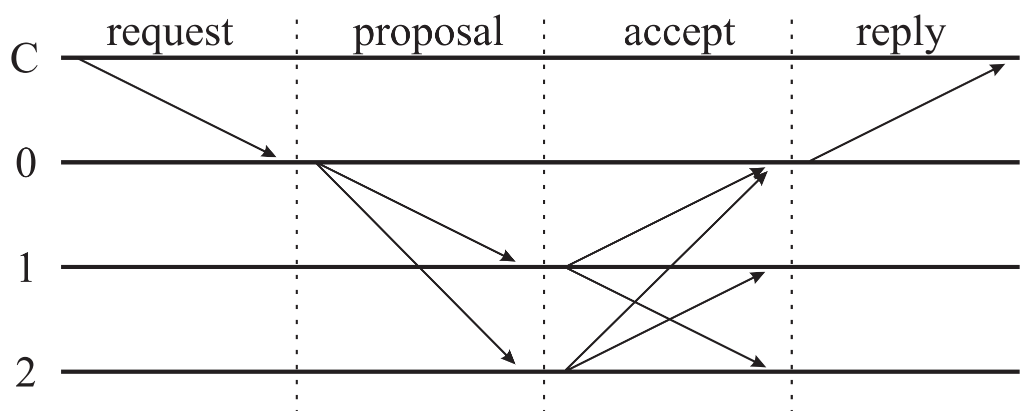
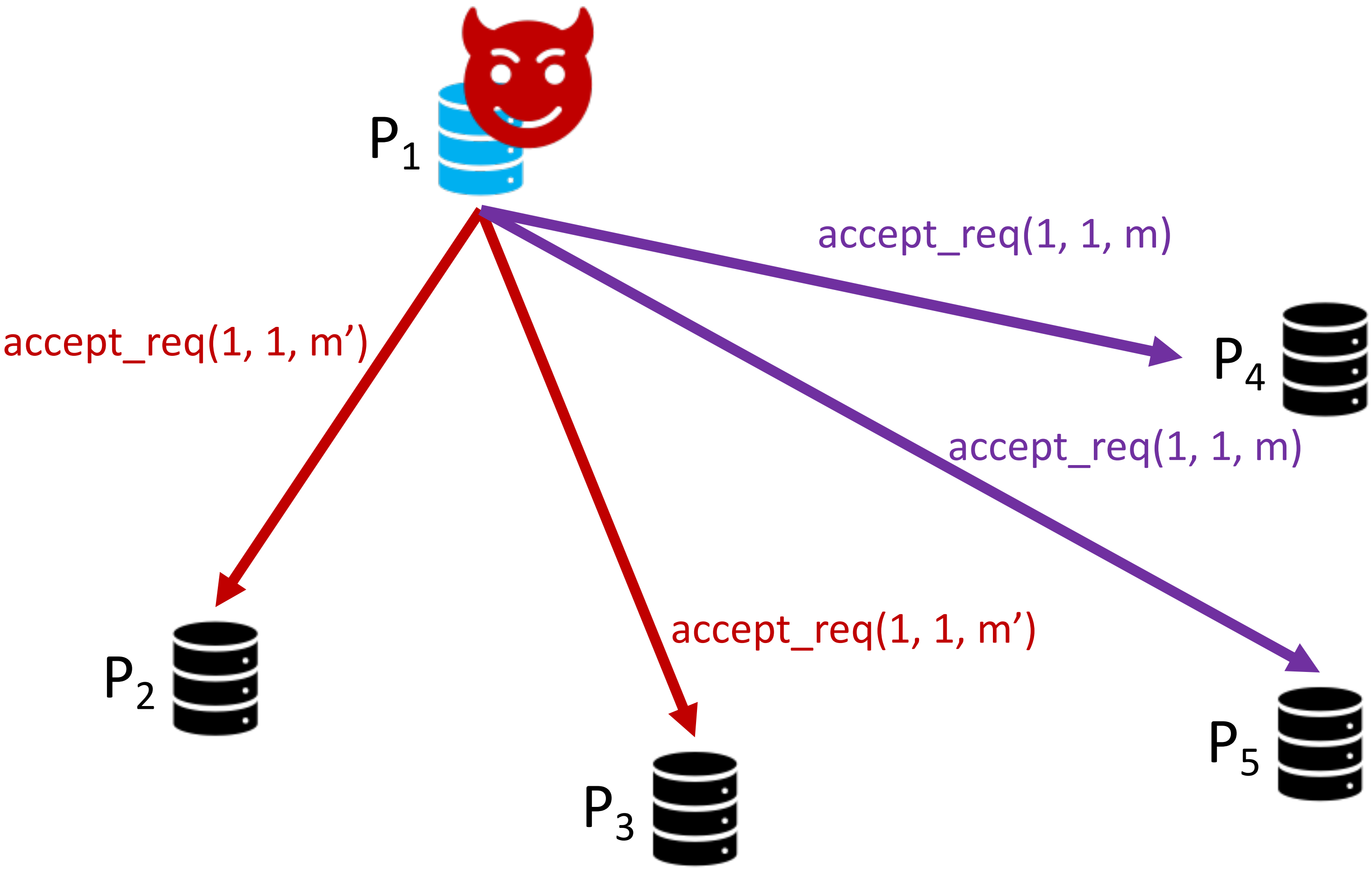


Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

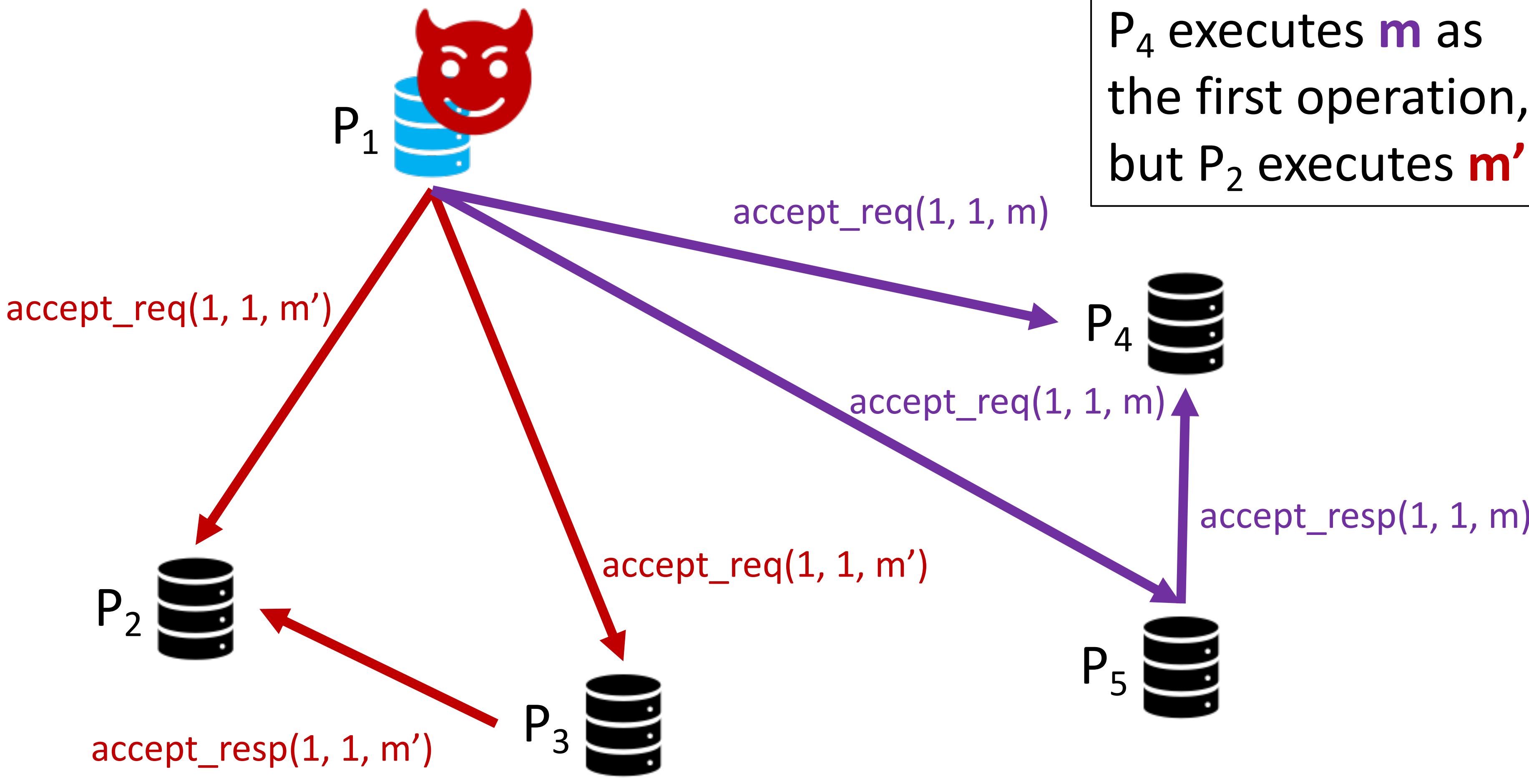
- From "Paxos for System Builders"

What can a malicious leader do?



What can a malicious leader do?

P_4 executes **m** as the first operation, but P_2 executes **m'**



Handling Byzantine Faults

- Simple majority is NOT enough
  - Malicious replicas may not keep their promise to only vote once
- New idea: require a majority of **correct** replicas
  - How many total replicas do we need?

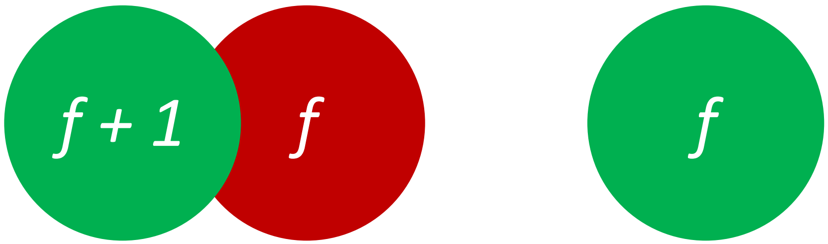
Handling Byzantine Faults

- How many total replicas do we need?
  - faulty replicas may never respond at all, so we can't require more than $n - f$ responses
  - but, it could be that our $n - f$ responses actually **do** include f faulty replicas
  - so, we are only guaranteed $n - 2f$ out of $n - f$ responses come from correct replicas
  - we need correct replicas to outnumber faulty ones in making our decision, so we need $n - 2f > f$, which implies $\boldsymbol{n > 3f}$

Handling Byzantine Faults

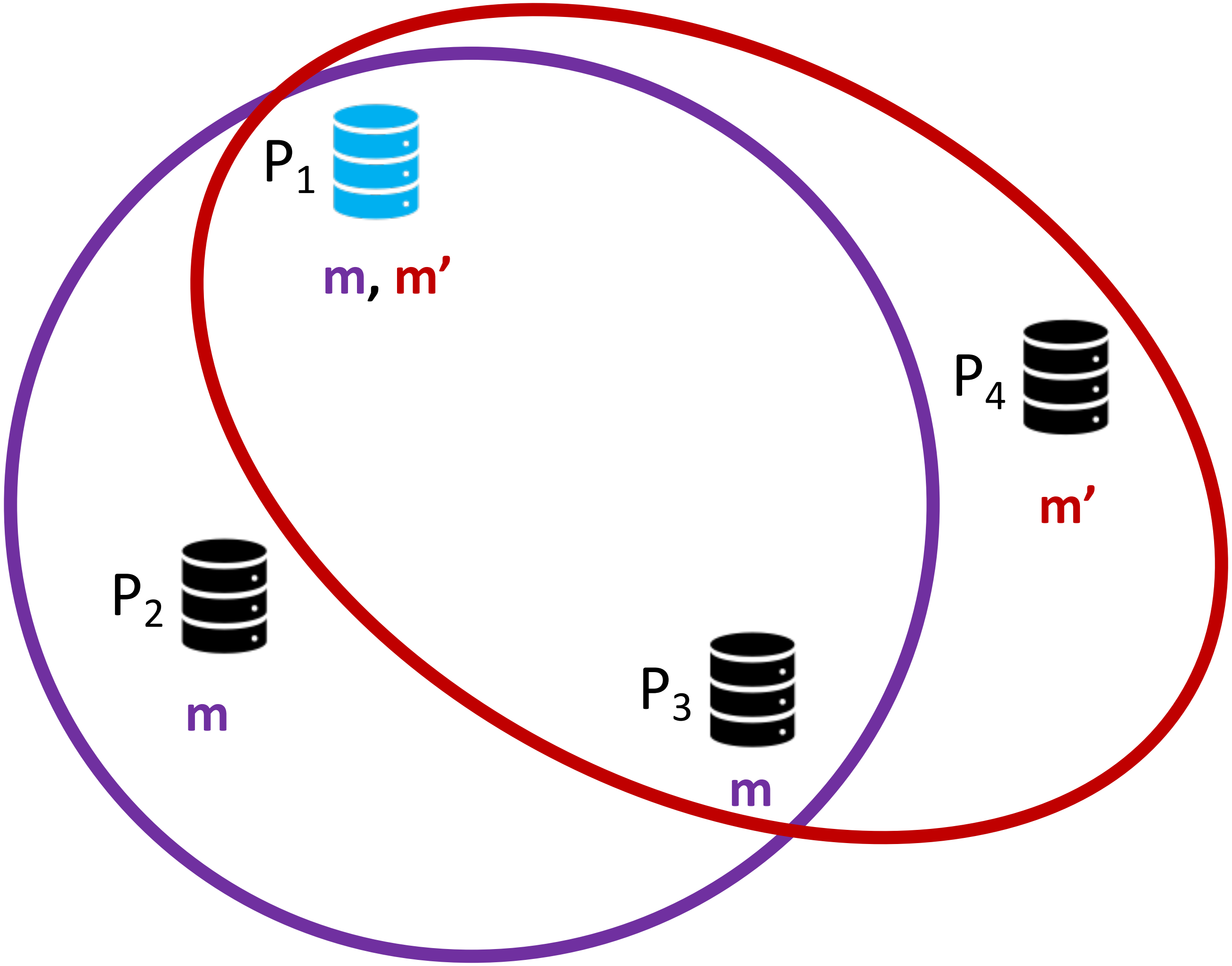
- Let our total number of replicas $\boldsymbol{n = 3f + 1}$
  - e.g. $f = 1 \rightarrow n = 4$; $f = 2 \rightarrow n = 7$
- Let a **quorum** $\boldsymbol{= 2f + 1}$
  - e.g. $f = 1 \rightarrow 2f+1 = 3$; $f = 2 \rightarrow 2f+1 = 5$
- Any 2 sets of $2f + 1$ replicas MUST share at least 1 **correct** replica

$2f+1 =$



No way to get *2f+1* without drawing at least one replica from other green circle

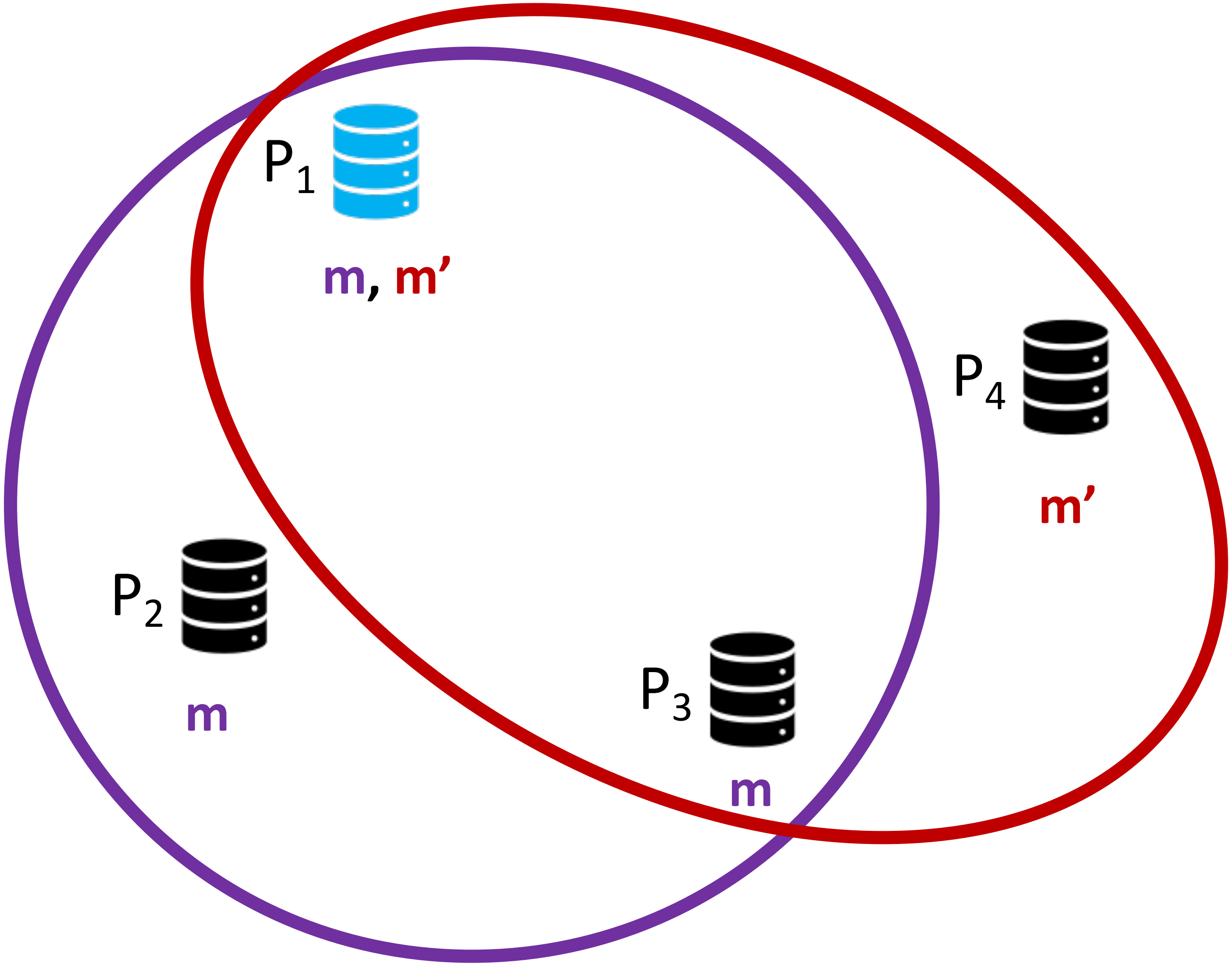
Quorums in BFT



***f* = 1**
3f+1 = 4
2f+1 = 3

There is no way to get 3 replicas to agree on m AND get 3 replicas to agree on m'

Quorums in BFT



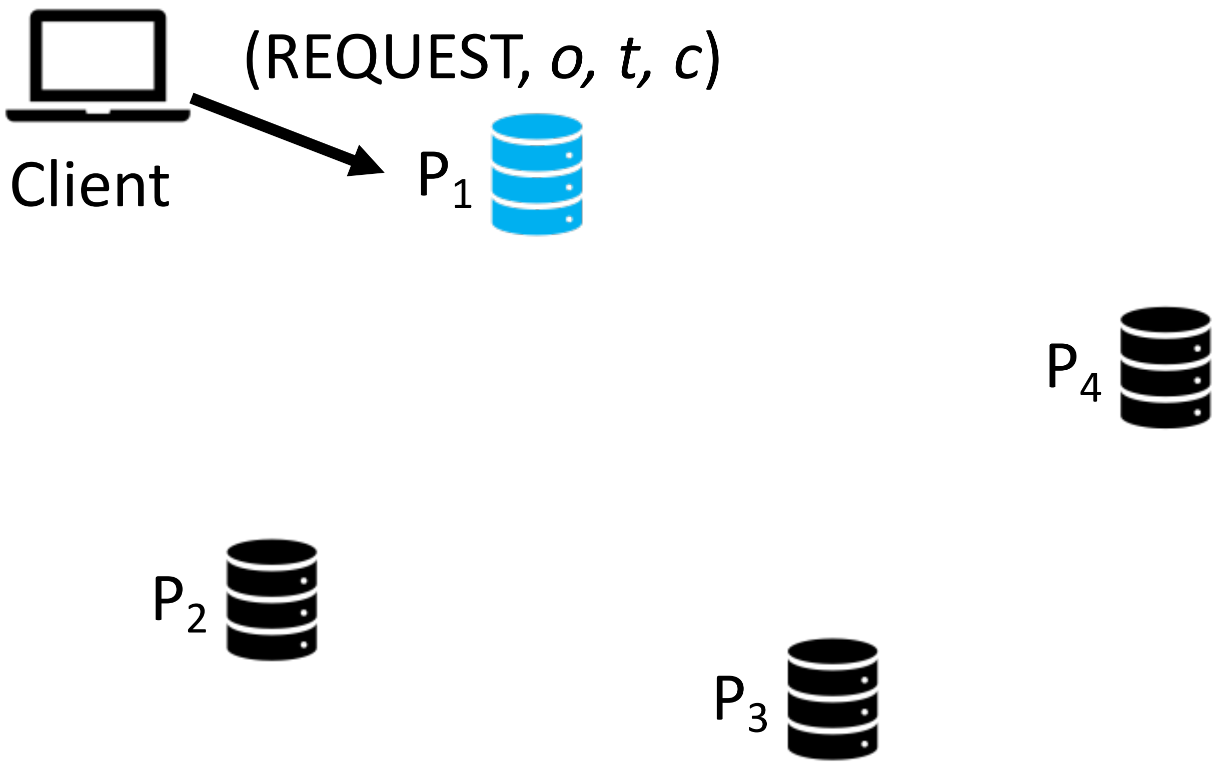
***f* = 1**
3f+1 = 4
2f+1 = 3

In the worst case, with 3f+1 replicas:

- **f+1 correct replicas vote m**
- **f correct replicas vote m'**
- **f faulty replicas vote m AND m'**

We can't get 2f+1 votes for conflicting messages!

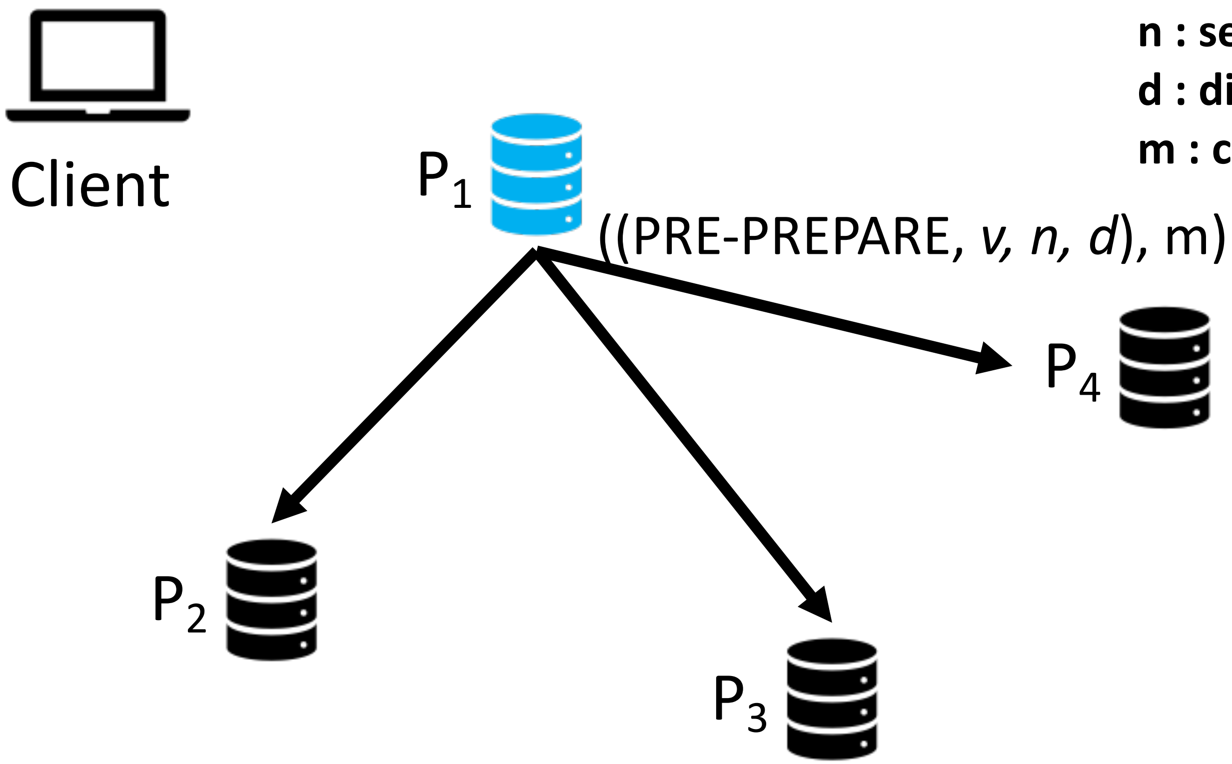
Basic BFT Protocol



o : operation to execute
t : timestamp
c : client id

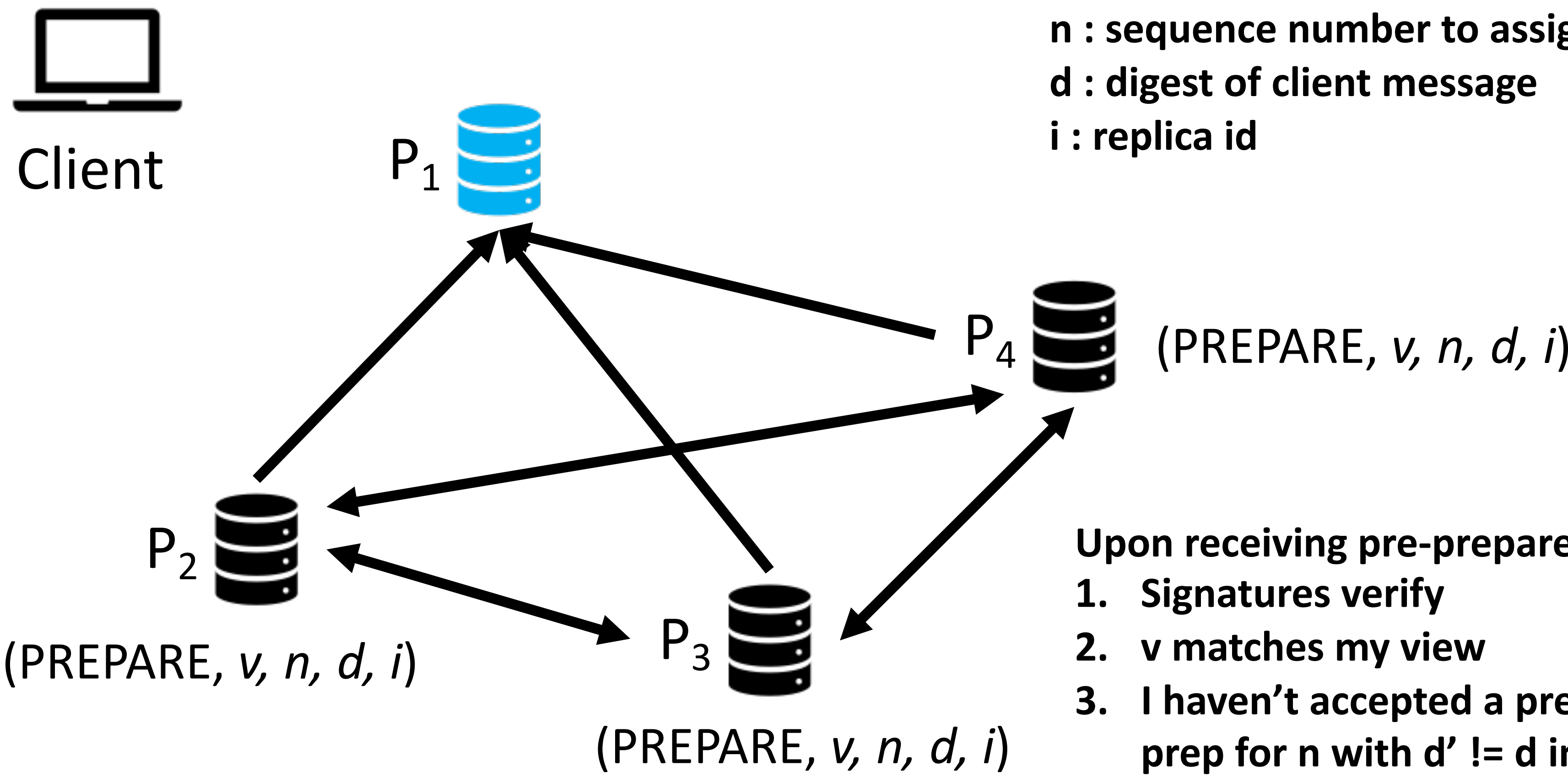
Basic BFT Protocol

v : view
n : sequence number to assign
d : digest of client message
m : client message



Basic BFT Protocol

v : view
n : sequence number to assign
d : digest of client message
i : replica id



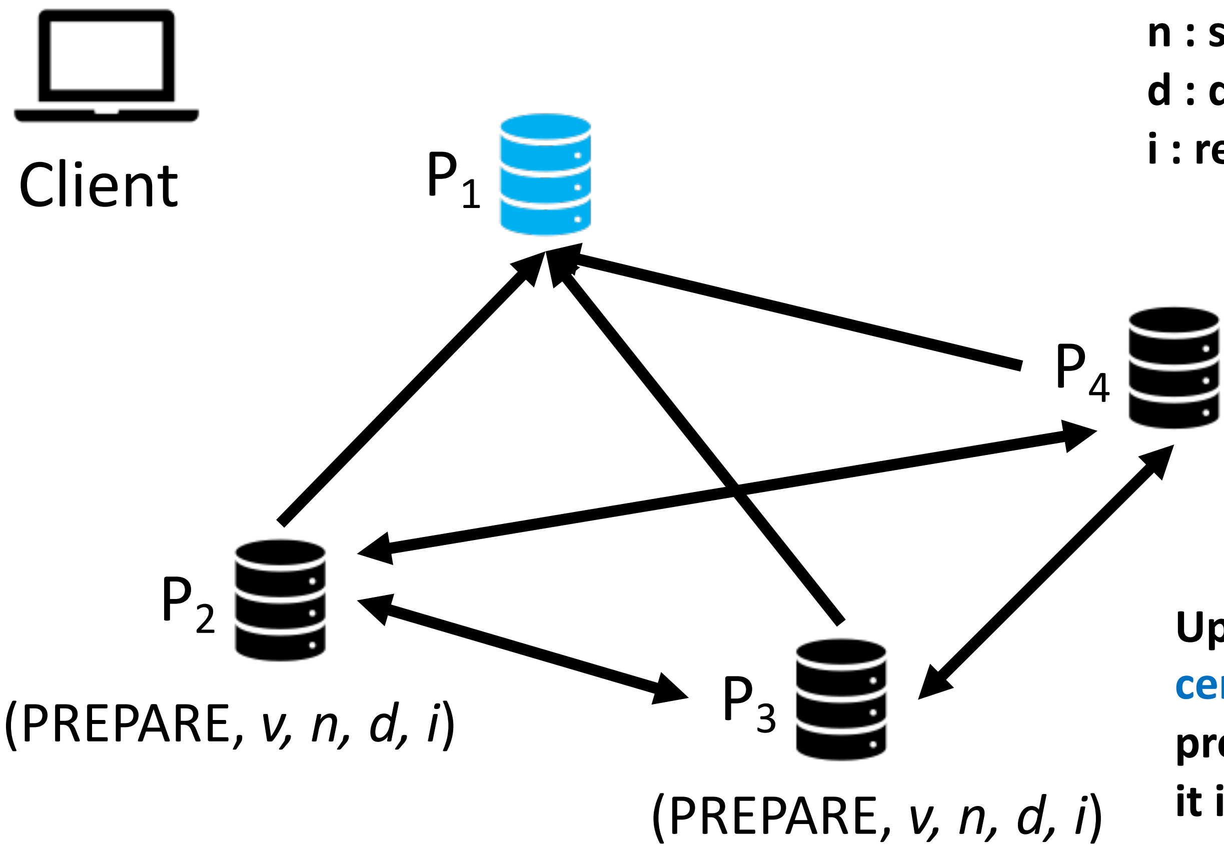
Upon receiving pre-prepare, if:

1. **Signatures verify**
2. **v matches my view**
3. **I haven't accepted a pre-prep for n with d' != d in v**
4. **h < n < H**

then send prepare

Basic BFT Protocol

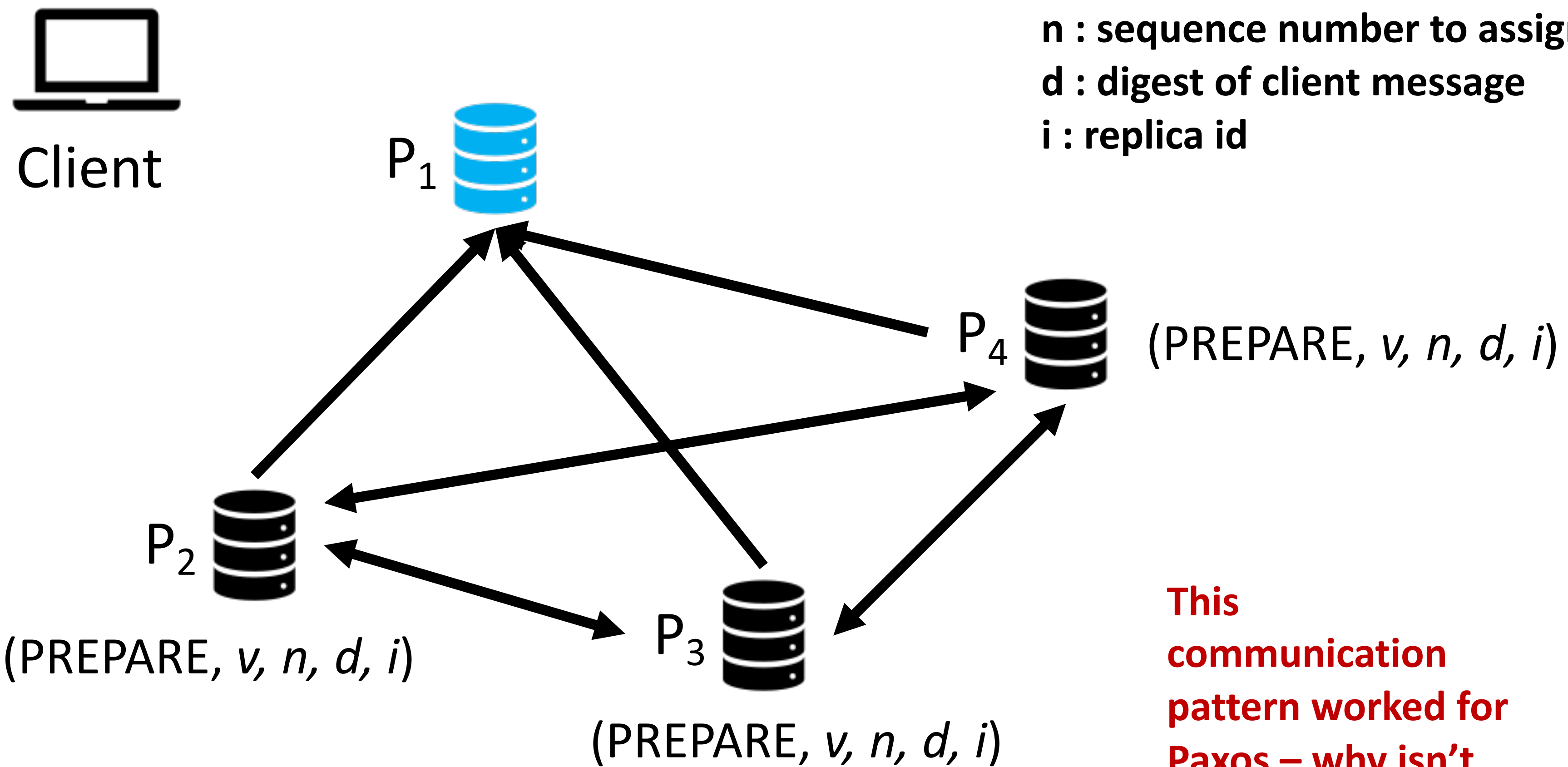
v : view
n : sequence number to assign
d : digest of client message
i : replica id



Upon collecting prepare certificate (pre-prepare + 2f prepares) for (n,m), I know that it is impossible for another replica to get a prepare certificate for (n,m')

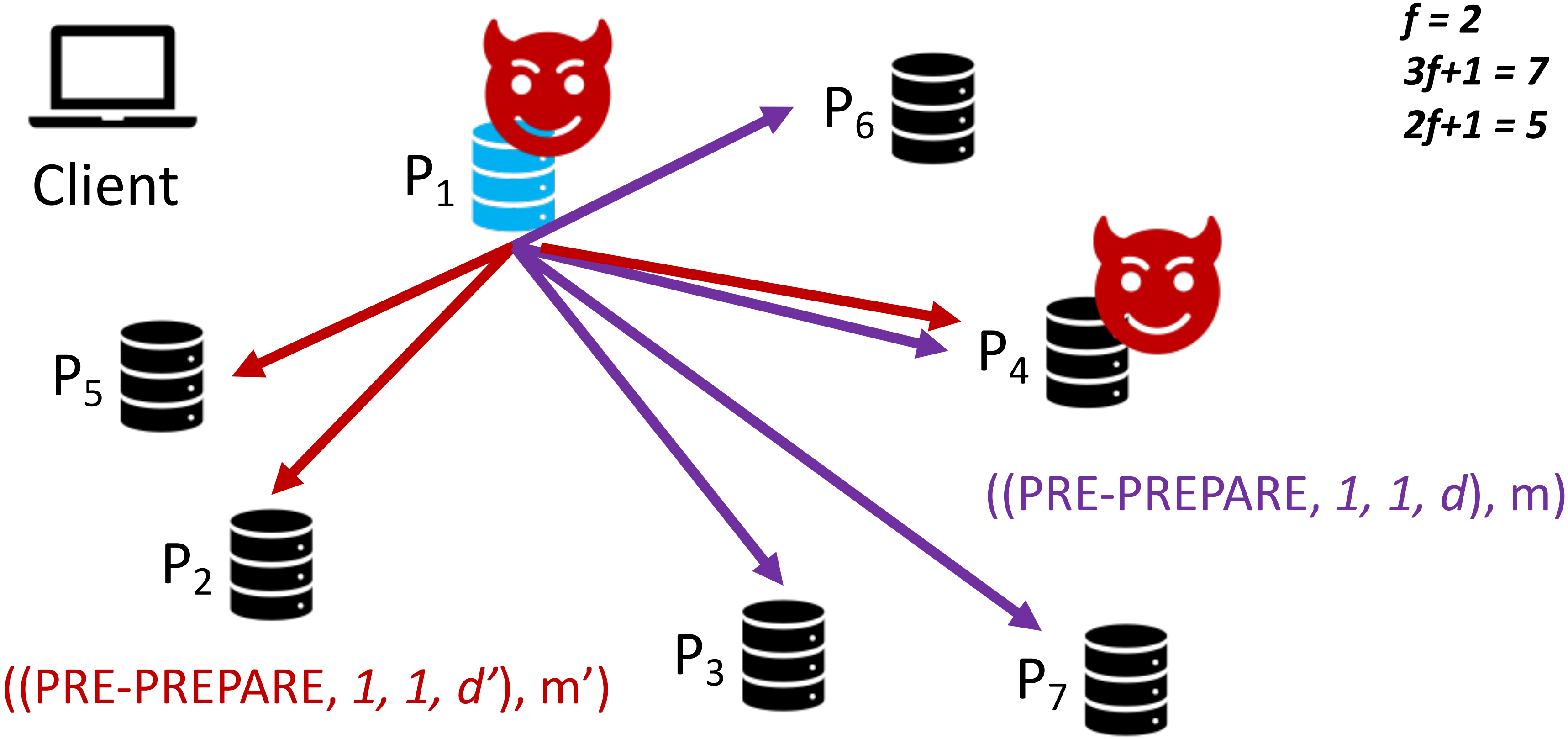
Basic BFT Protocol

v : view
n : sequence number to assign
d : digest of client message
i : replica id



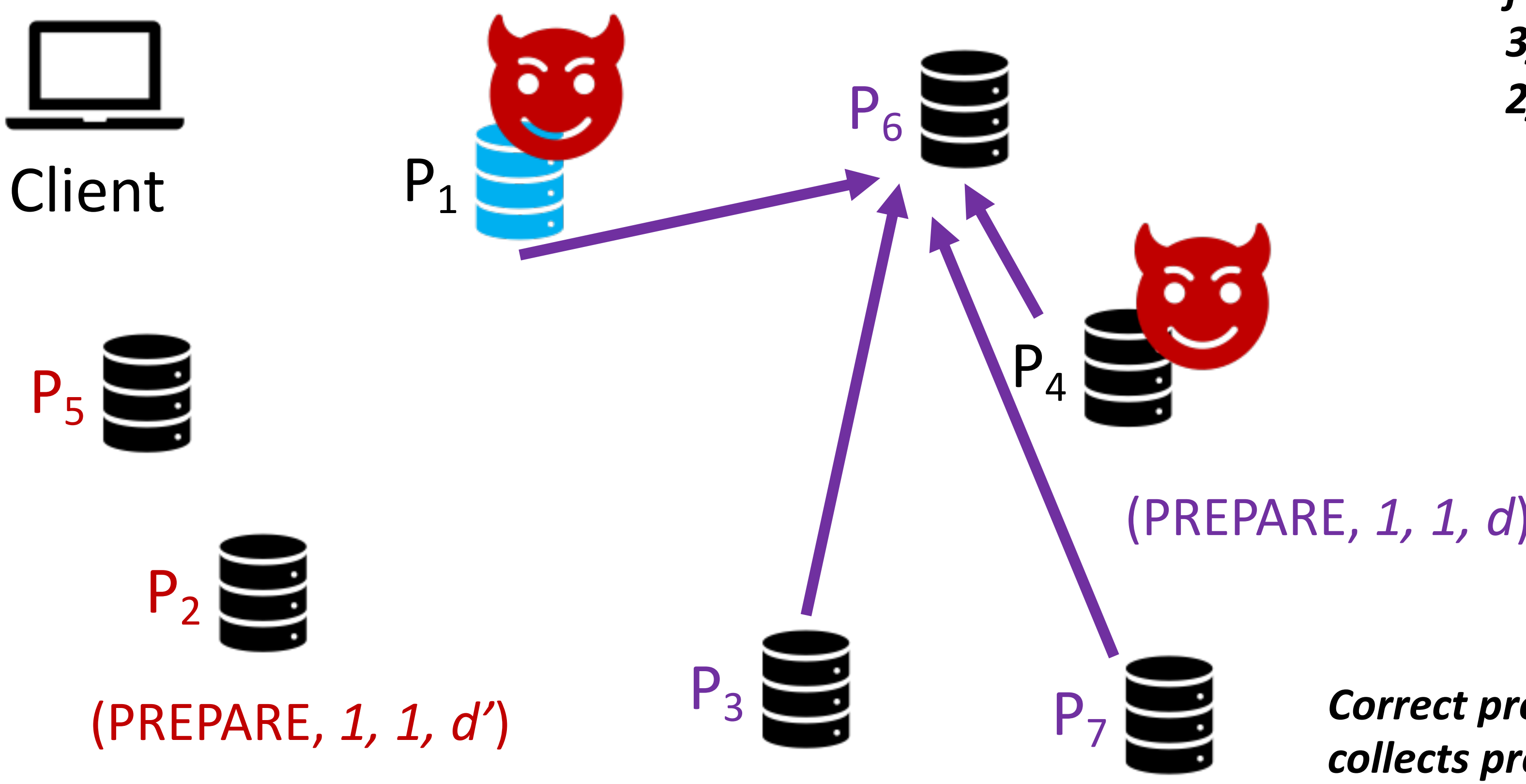
This communication pattern worked for Paxos – why isn't it enough here?

Consider a Malicious leader (again)



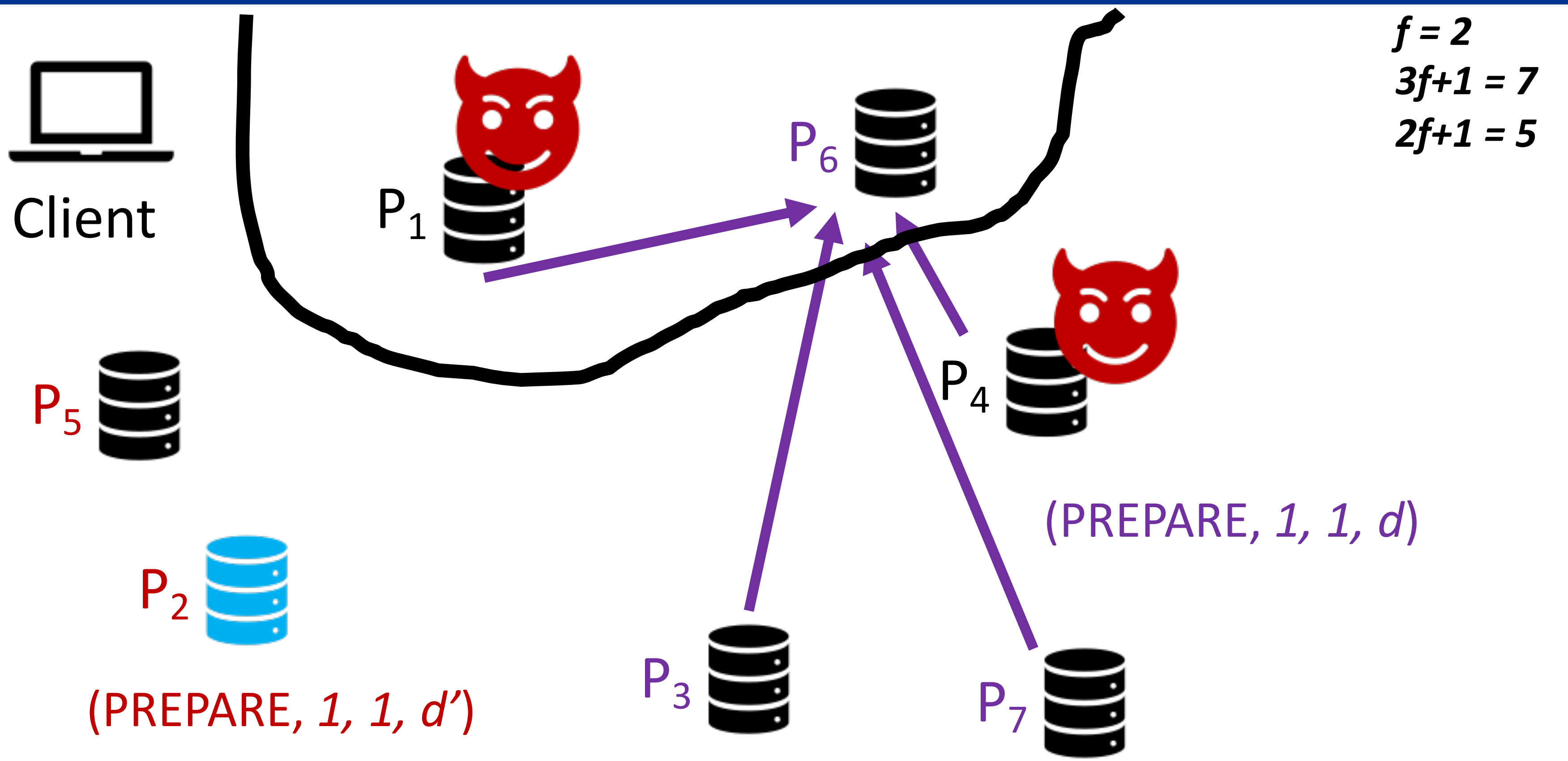
Consider a Malicious leader (again)

f = 2
3f+1 = 7
2f+1 = 5



Correct process P_6 collects prepare certificate for (1,m) (commits to m for seq 1)

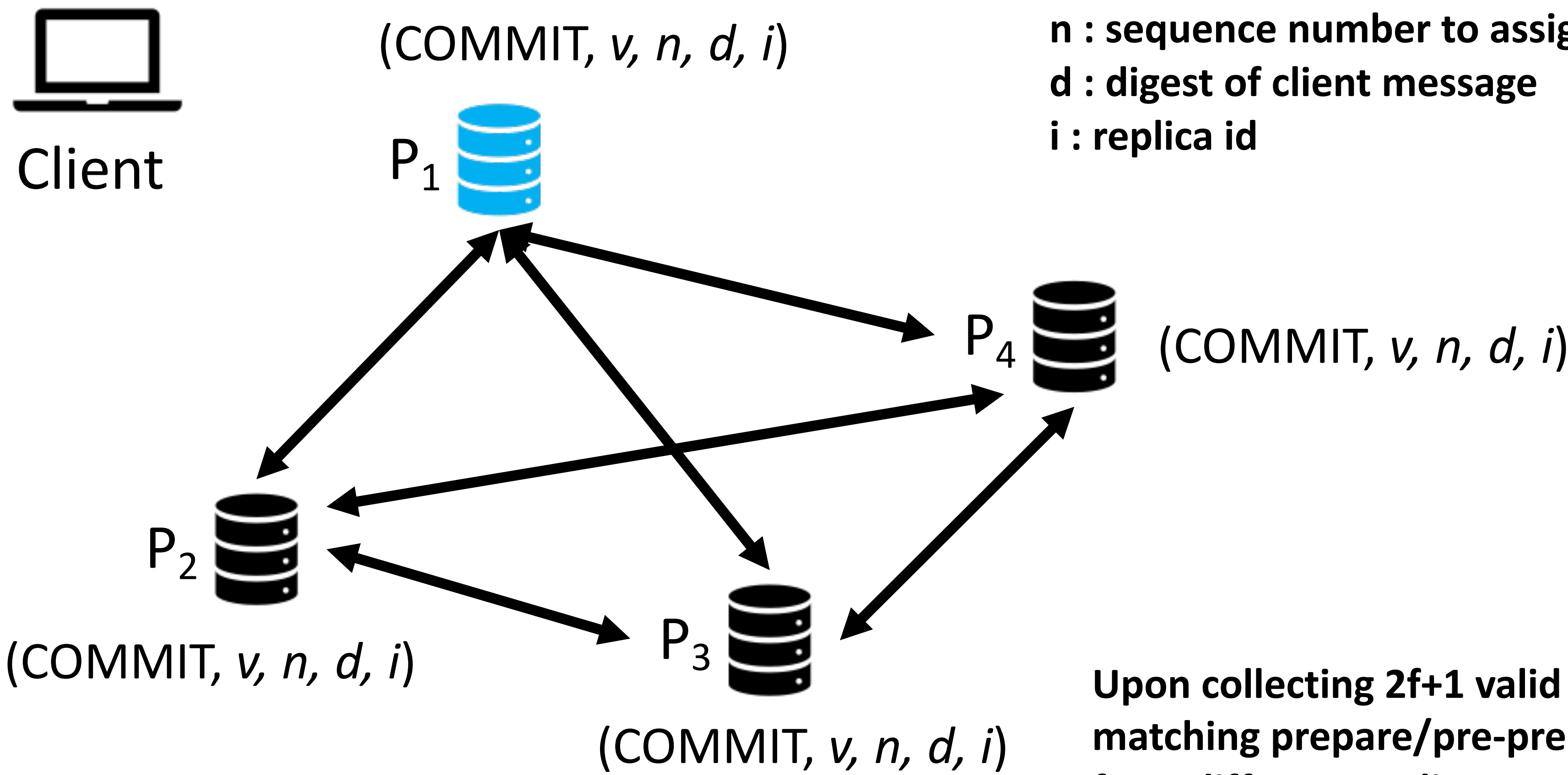
Consider a Malicious leader (again)



Now consider that P_1 and P_6 get partitioned away. New leader P_2 must protect P_6 by choosing m for seq 1, but it doesn't have enough information to know that (either m or m' could have been chosen). The protocol gets stuck!

Basic BFT Protocol: Commit Phase

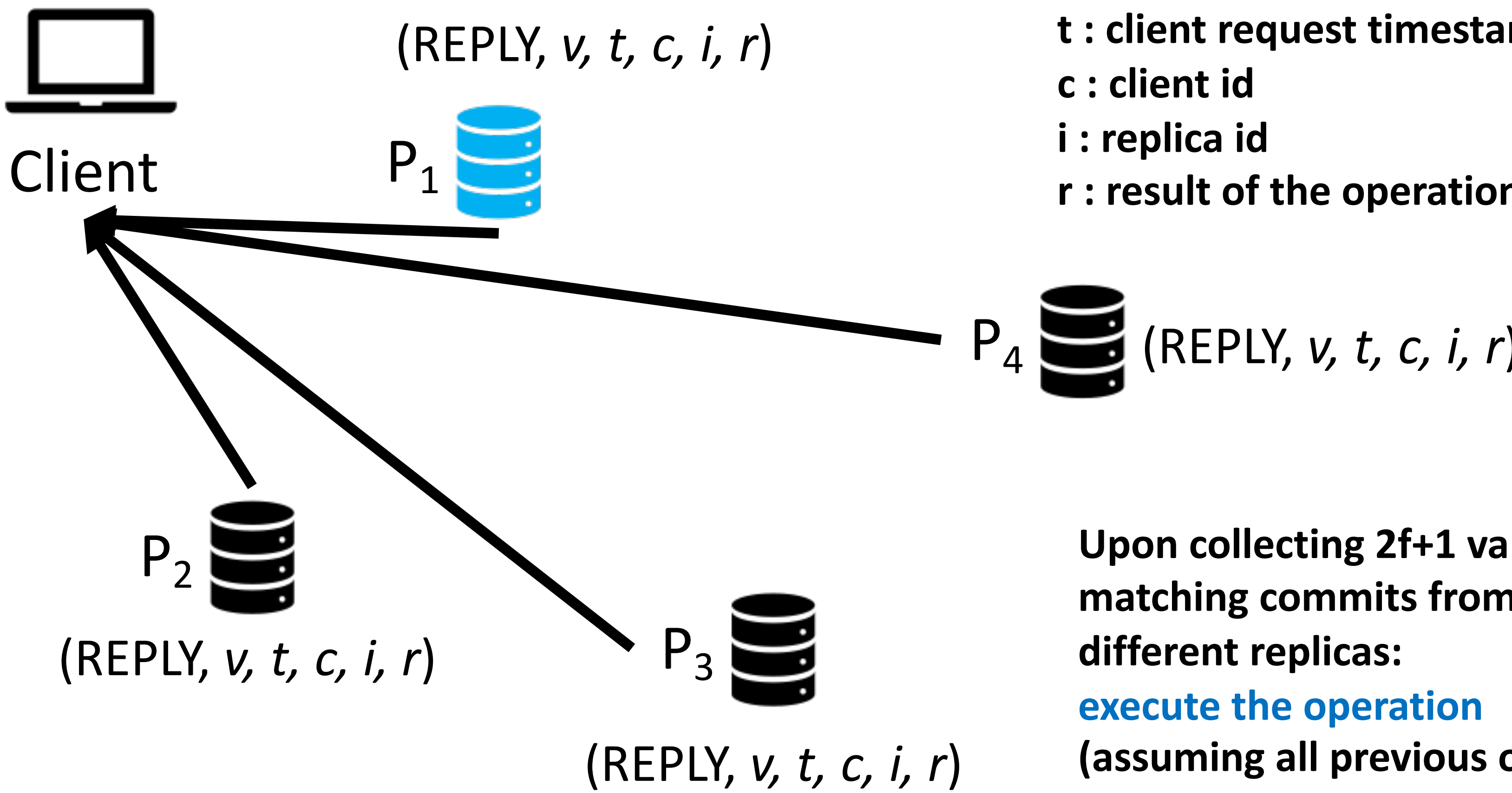
v : view
n : sequence number to assign
d : digest of client message
i : replica id



Upon collecting 2f+1 valid matching prepare/pre-prepare from different replicas:
send commit

Basic BFT Protocol

v : view
t : client request timestamp
c : client id
i : replica id
r : result of the operation



Upon collecting 2f+1 valid matching commits from different replicas:
execute the operation
(assuming all previous ops have been executed) and reply to the client

Basic BFT Protocol

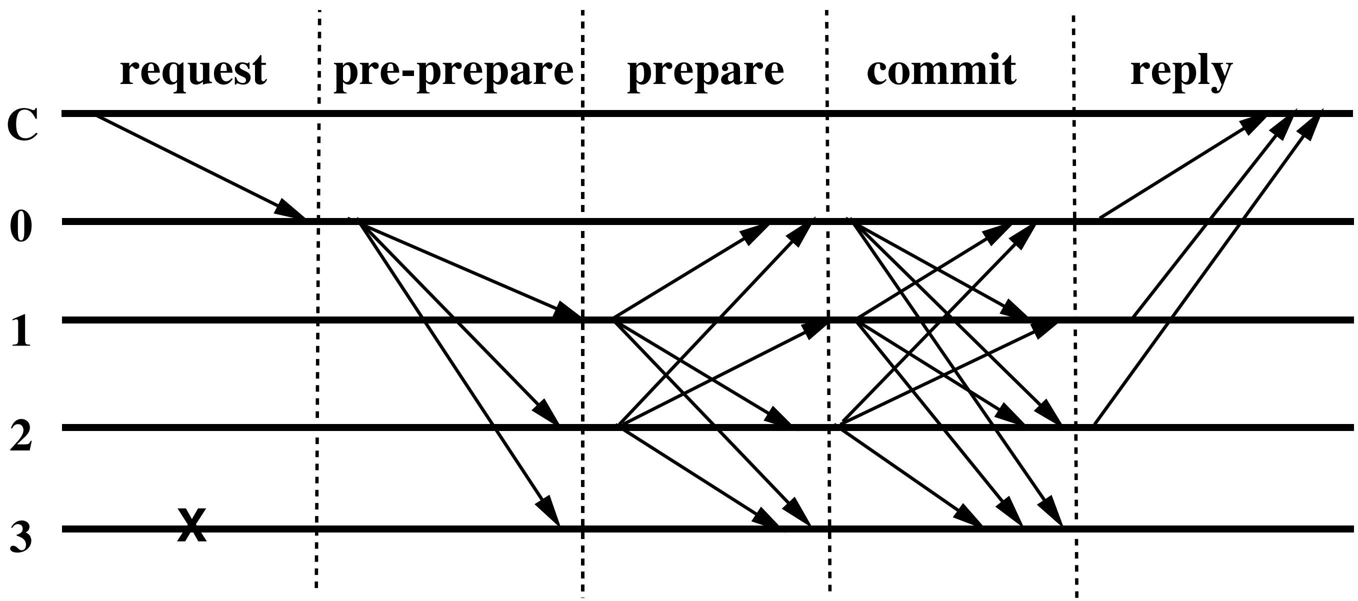


Figure 1: Normal Case Operation

Basic BFT Protocol

- **(One) Key difference from Paxos**:
  - It is possible to have **more than one value proposed in a given view** (by faulty leader)
  - So, we can’t rely only on view numbers to break ties

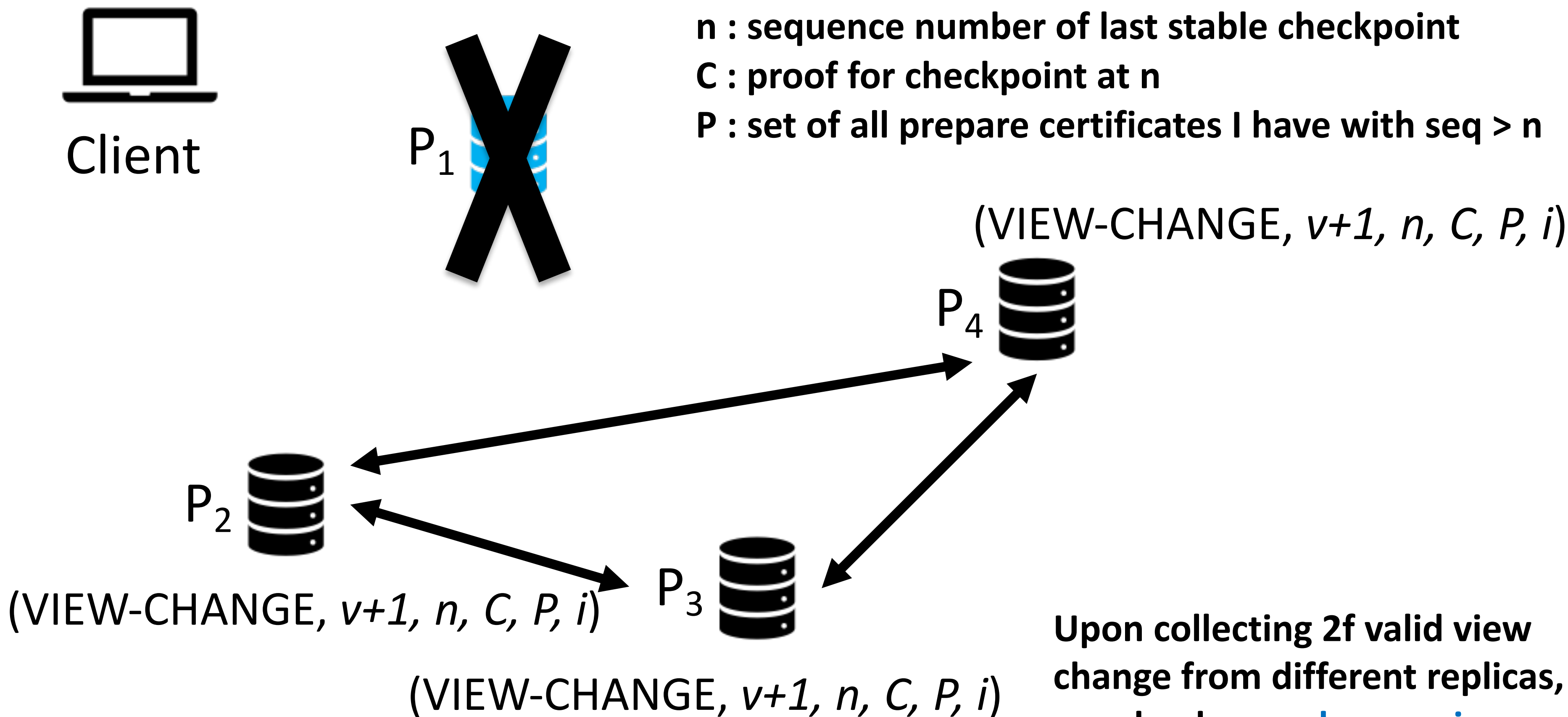
- Why does the **commit phase** solve our problem?
  - By collecting 2f+1 valid commits for sequence number n and message m, a replica learns that a **quorum has collected a prepare certificate for (n,m) in view v**
  - This implies that **NO correct replica can collect a prepare certificate for (n,m’) in view v**
  - This also implies that **ANY future quorum must include at least one correct replica with a prepare certificate for (n,m)**

BFT View Change

- **Same principle as Paxos**: **new leader must communicate with a quorum** to find out what might have been ordered/executed
- **But**,
  - Quorum = **2f+1** (not n/2)
  - We use **prepare certificates** to identify **unique** operations that may have been executed in a view
  - Replicas must be able to **verify** that the leader preserves ordering operations that may have been executed in a previous view

BFT View Change

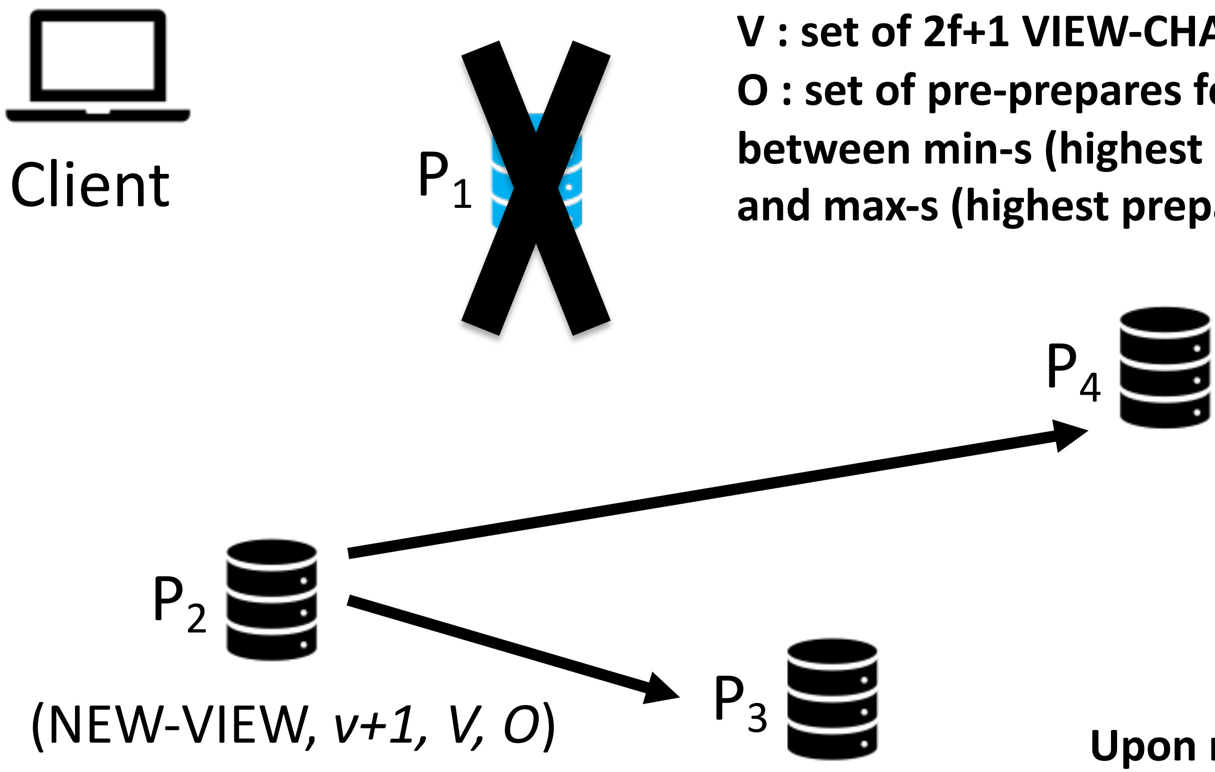
v+1 : next view
n : sequence number of last stable checkpoint
C : proof for checkpoint at n
P : set of all prepare certificates I have with seq > n



Upon collecting 2f valid view change from different replicas, new leader sends new view

BFT View Change

v+1 : next view
V : set of 2f+1 VIEW-CHANGE msgs
O : set of pre-prepares for all sequence numbers between min-s (highest stable seq n in a VC msg in V) and max-s (highest prepare cert in a VC msg in V)



Upon receiving valid new-view message, each replica sends prepare for each pre-prepare in O

Additional Considerations

- **Garbage Collection**
  - Checkpoint state periodically to keep message log from growing indefinitely
  - In Byzantine environment, checkpoints must be **coordinated**
- **Non-determinism**
  - Transform non-deterministic operations into deterministic calculation based result from one or more replica(s)
    - e.g. BFS “time-last-modified”: take max of leader’s proposed value or highest value seen so far + 1
    - Observation: median of 2f+1 values must be >= some value proposed by a correct process and <= some value proposed by a correct process
- **Communication Optimizations**
  - Reduce # of replies, tentative execution, read-only optimization
- **Crypto Optimizations**
  - MACs vs Digital Signatures: MACs are faster but weaker; requires non-trivial changes to view change protocol (view-change acks)

Additional Considerations

- **Liveness**
  - To guarantee progress, we need at least 2f+1 correct replicas in a stable view with a correct leader
  - **Tension**: if leader fails, we want to replace quickly, BUT if our timeout is too aggressive, we will view change before a (correct) leader has a chance to make progress
  - **Solution approach**: double the timeout each time we view change
    - Clean theoretical guarantee: we make progress as long as message delay doesn't grow faster than timeout forever
    - Problematic in practice: that progress might be extremely slow

Summary

- Introduced the first practical state-machine replication protocol that tolerates Byzantine faults in an asynchronous network
- Optimized the protocol to achieve dramatically better performance than state-of-the-art (at the time) – replacing signatures with MACs
- Implemented replicated NFS service on top of the BFT library