# Practical Byzantine Fault-Tolerance

## Impossibility of consensus with a faulty process

**Consensus problem**: each process starts with an input value from a set V and must decide a value from V

- **Safety** properties:
    - **Agreement** - no two correct processes decide differently
    - **Validity** - the decided value depends on the input values of the processes
- **Liveness**: every execution of a protocol decides a value

### Theorem

In an asynchronous system in which at least one process may crash, there is no deterministic consensus protocol that is both live and safe

- Even if the network is reliable
- **FLP's impossibility result**

#### Intuition-based Argument

- In an asynchronous distributed system we cannot distinguish a slow process from a crashed process
- For every consensus algorithm, there are executions in which the algorithm reaches a state such that:
    - If a process takes no decision, it may remain forever undecided, thus violating liveness
    - If a process makes a decision, independently of the decision rule, it may violate one of the safety properties
- Paxos favors safety at the cost of liveness
    - It never violates safety
    - But it may block, if it is unable to elect a single leader

## System Model and Problem Definition

### System Model (Asynchronous system)

**Network** may:

- fail to deliver messages
- delay messages
- duplicate messages

- deliver messages out of order

**Nodes/processes** may fail arbitrarily (Byzantine failure model)

- Processes use cruptographic techniques to prevent spoofing and replays and detect corrupted messages

### Service Properties

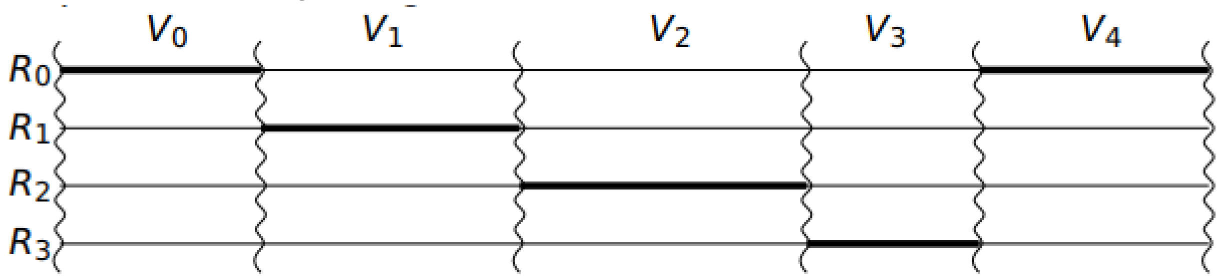
- State machine replication
- Safety - the replicated service satisfies linearizability
- Liveness - cannot be assured in an asynchronous system
- Reiliency - Tolerates f faulty replicas with n = 3f + 1 replicas

## Protocol Overview

### Views and Leaders

**View**: numbered system configuration

- Replicas move through a succession of views

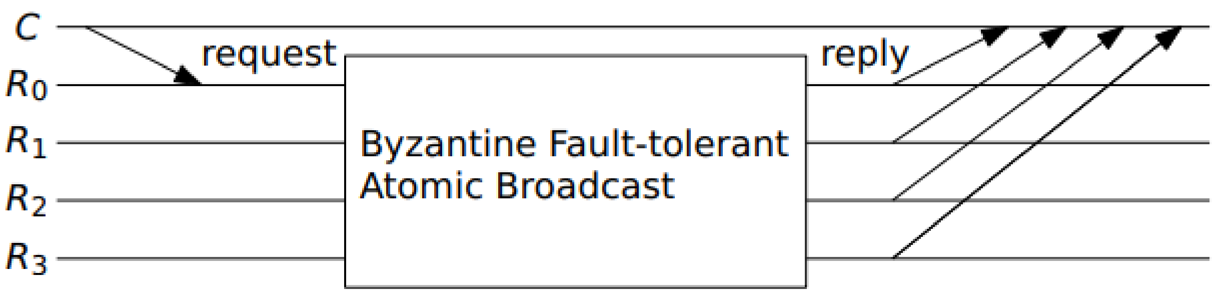


- Each view has a **leader**:

$$p = vmodn$$

where,
v → view number
n = 3f + 1 → number of replicas
- View changes occur upon suspicion of the current leader

### Algorithm (SMR)



1. A client sends a request to execute a service operation to the leader
2. The leader **atomically broadcasts** the request to all replicas
3. Replicas execute the request and send the reply to the client
4. The client waits for replies with the **same result** from f + 1 replicas

### Client

1. A client c sends a request to execute a service operation o by sending a (REQUEST, o, t, c)σc message to the **leader**
2. The leader **atomically broadcasts** the request to all replicas
3. Replica i executes the request and sends (REPLY, v, t, c, i, r)σi , with the result r of the execution of the operation
4. The client waits for f + 1 replies with **valid signatures** from different replicas, and with the same t and r, before accepting r
    1. If the client does not receive replies on time, it **broadcasts** the request to all replicas

## Atomic Broadcast

### Quorums and Certificates

- PBFT uses quorums to implement atomic multicast
- These quorums satisfy two properties
    - **Intersection**: any 2 quorums have at least a correct replica in common (as they intersect in f + 1 replicas)
    - **Availability**: there is always a quorum with no faulty replicas
- Messages are sent to replicas
- Replicas collect **quorum certificates**
    - **Quorum Certificate**: set with one message for each element in quorum, ensuring that relevant information has been stored
    - **Weak certificate**: set with at least f + 1 messages from different replicas
        - The set of f + 1 replies a client must receive before returning the result is a weak certificate, the **reply certificate**

## Replicas

The state of each replica comprises:

- **The service state**
- **A message log** containing messages the replica has accepted
- **View id** an integer with the replica’s current view id

When the leader, l, receives a client request, m, it starts a three-phase protocol to atomically multicast the request to the replicas:

1. **Pre-prepare**
2. **Prepare** together with **pre-prepare** ensure total order of requests in a view
3. **Commit** together with **prepare** ensure total order of requests across views

## Pre-Prepare Phase

Upon receiving a client request m the leader:

1. Assigns a monotonically increasing sequence number, n, to m
2. The leader multicasts message ((PRE-PREPARE, v, n, di)σ , m) to the other replicas, where:
   (PRE-PREPARE, v, n, d)σ → is the pre-prepare message
   m → client's request with its signature
   d = D(m) → m's message digest

Upon receiving a PRE-PREPARE messsage a replica accepts it if:

- It is in view v
- The signatures in request, m, and in the PRE-PREPARE message are valid and d is the digest for m
- It has not accepted a PRE-PREPARE message for view v and sequence number n with a different digest d
- n is between a low water mark, h, and a high water mark, H

## Prepare Phase

- On accepting a PRE-PREPARE message replica i enters the **prepare** phase
- On receiving a PREPARE message a replica, including the leader, **accepts** it provided that

**Prepared Certificate** each replica collects:

1. A PRE-PREPARE message
2. 2f PREPARES messages from different replicas
   for request m in view v with sequence number n

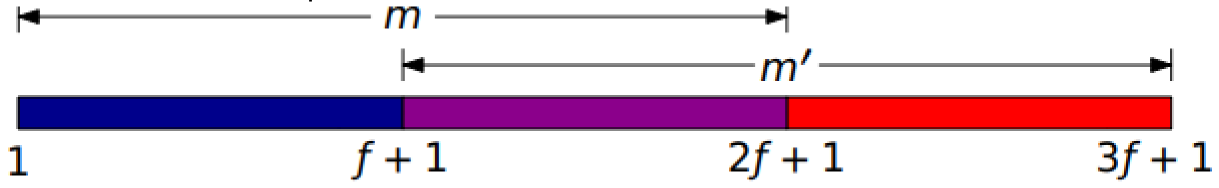
- After collecting a prepared certificate a replica prepared the request

### Total order within a view

**Lemma**: The pre-prepare and prepare phases guarantee that a replica cannot obtain prepare-certificates for the same view and sequence number and requests with different digests

**Proof sketch**:

- To obtain a prepare-certificate for a request in a view v with a sequence number n, 2f + 1 replicas need to send/accept a PRE-PREPARE message in view v with number n
- Because there are 3f + 1 replicas, the quorums of two prepare-certificates must have at least f + 1 replicas in common



- Thus at least one non-faulty replica would have to send/accept two PRE-PREPARE messages with the same sequence numbers in the same view but different digests
- By the protocol, this is not possible

### Ensuring order across view changes

- The pre-prepare and the prepare phases are not sufficient to ensure total order for requests across view-changes
- Replicas may receive PREPARE messages in the same view with the same sequence and different requests
- Replicas may collect prepared certificates in different views with the same sequence number and different requests

## Commit Phase

**On collecting a prepared certificate** replica i enters the commit phase and Multicasts message (COMMIT, v, n, d, i)σi to all replicas
**On receiving a COMMIT message** a replica, including the leader, accepts it provided that:

- The view v is the same as the replica's current view
- Its signature is correct
- The sequence number is between h and H

**Commit certificate** is a set of accepted 2f + 1 COMMIT messages with the same view, sequence number and digest, received from different replicas (inlcuding the acceptor)
**Committed request** by a replica, if the replica has both the prepared and the committed certificates

### Invariant

The commit phase ensures that if a replica committed a request that request is prepared by at least f + 1 non-faulty replicas
**With the view change protocol** this invariant guarantees that:

1. Non-faulty replicas agree on the sequence numbers of a committed request, even if different replicas may commit it in different views
2. Any request committed at a non-faulty replica, will commit at all non-faulty replicas, possibly in different views.

## Request Delivery and Execution

- Each replica i executes the operation requested by m after:
    - It has committed that request.
    - It has executed all requests with a lower sequence number
- Replicas send a reply to the client after executing the requested operation

## Garbage Collection and Checkpoints

- To ensure safety a replica cannot discard the messages with a sequence number as soon as it executes that request
- For replica repair or replacement, we need state synchronization
- A replica periodically, i.e. every K requests, checkpoints its state
- After generating such a proof, the checkpoint becomes stable
- A checkpoint proof requires exchanging messages
- A replica maintains several copies of the service state

### Checkpoint Proof Generation

- **Upon a checkpoint** replica i multicasts a (CHECKPOINT, v, n, d, i)σi message to all replicas
- **Upon receiving a CHECKPOINT message** a replica saves it in its log until it has collected a weak certificate, the **stable certificate**
- **Stable certificate** is a set of f + 1 CHECKPOINT messages signed by different replicas (including itself) for sequence number n with the same digest d – this is the **checkpoint's proof**
- **Upon collecting a stable certificate** a replica discards:
    - all PRE-PREPARE, PREPARE and COMMIT messages with sequence number less than or equal to n
    - all earlier checkpoints and respective CHECKPOINT messages

### Updates to the low and high water marks

A replica advances the low and high water marks every time it runs the checkpoint protocol

h → set to the sequence number n of the last stable checkpoint
H → set to h + L, where L is a small multiple (e.g. 2) of K, the checkpoint period

# View Change

## View Change Protocol

### First Phase

- **Purpose**: ensure liveness upon failure of the leader
- **Leader Failure** is suspected with the help of a timer
  - The timer prevents a replica from waiting indefinitely for requests to execute
    - A replica is *waiting* for a request, if it received a valid request but has not executed it yet
- **Upon timeout** in view v, replica i starts a view change to advance to view v + 1
  1. It stops accepting messages (other than CHECKPOINT, VIEW-CHANGE and NEW-VIEW)
  2. . It multicasts a (VIEW-CHANGE, v + 1, n, C,P, i)σi message

n → seq. number of the last stable checkpoint s known to i
C → checpoint's stable certificate
P → set with a prepared certificate for each request prepared at replica i with sequence number greater than n

### Second Phase

- **New-view Certificate** is a set with 2f + 1 valid VIEW-CHANGE messages for view v + 1 each signed by a different replica
- **On collecting a VIEW-CHANGE certificate** the leader l of view v + 1

1. Updates its log and/or service state, if necessary
2. Multicasts (NEW-VIEW, v + 1, V, O, Niσl to other replicas, where
   V → new-view certificate
   O and N → sets of PRE-PREPARE messages (without the respective requests) that propagate sequence number assignments from previous views
3. Finally, l enters view v + 1, and starts accepting messages

### Computation of O and N

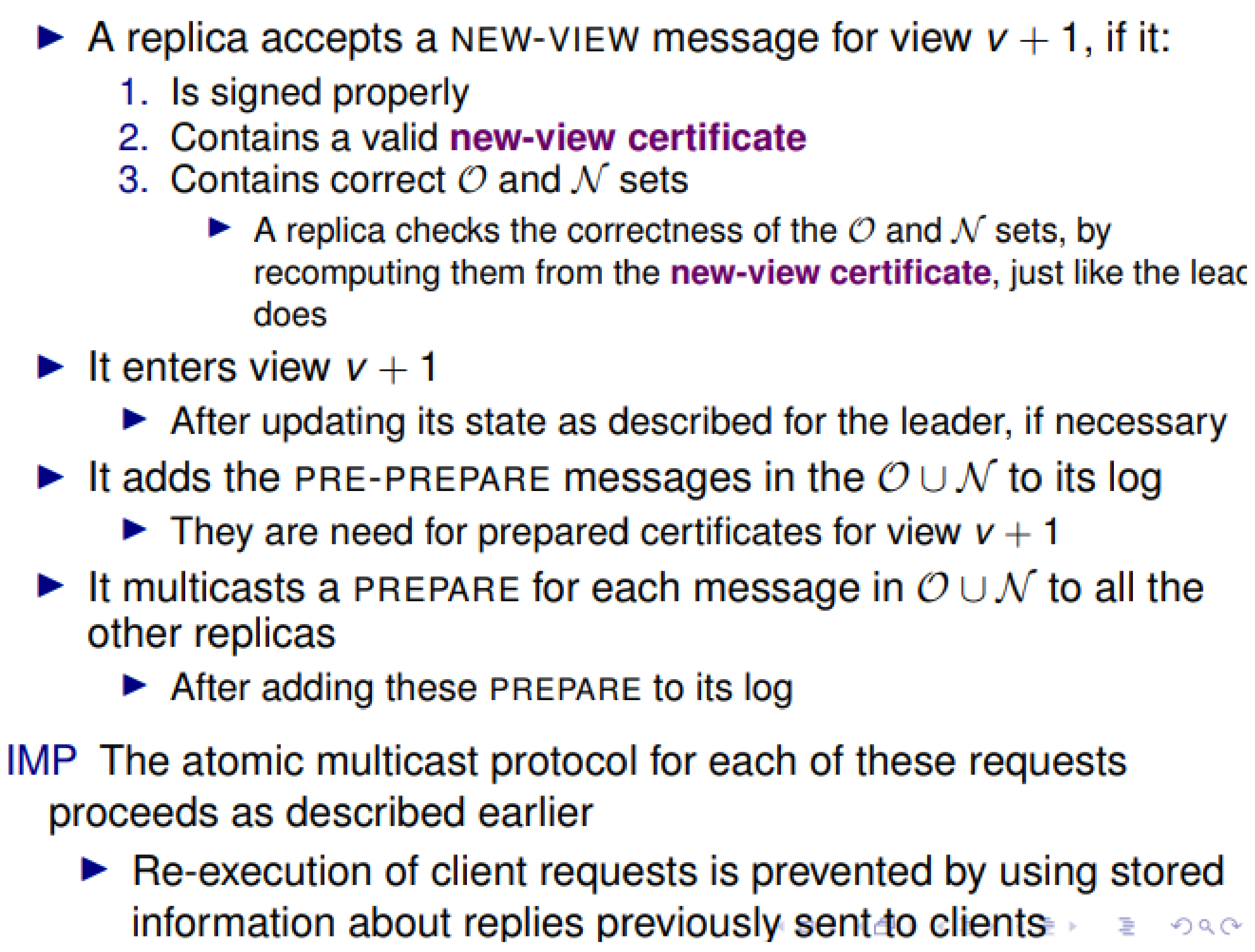
1. The leader, $\ell$, determines the sequence numbers
   - *h* of the latest stable checkpoint in $\mathcal{V}$
   - *H* the highest in a message in a prepared certificate in $\mathcal{V}$
2. $\ell$ creates a new PRE-PREPARE message for view $v + 1$ and sequence number $n$, s.t. $h < n \leq H$. There are two cases:
   - There is a prepared certificate in $\mathcal{V}$ for sequence number $n$, the leader adds a new message $\langle \text{PRE-PREPARE}, v + 1, n, d\rangle_{\sigma ell}$ to $\mathcal{O}$, where $d$ is the digest in the prepared certificate with sequence number $n$ and with the highest view number in $\mathcal{V}$
   - Othewise it adds a new message $\langle \text{PRE-PREPARE}, v + 1, n, null\rangle_{\sigma_\ell}$ to $\mathcal{N}$, where *null* is a special digest for a no-op request (this is similar to what happens in Paos upon a new leader election)
   - IMP the leader appends the messages in $\mathcal{O}$ and $\mathcal{N}$ to its log, as they belong to the pre-prepare phase for these requests in the new view

## State Update at New Leader

- If $h$ is greater than the sequence number of its latest stable checkpoint, the leader adds the stable certificate for the checkpoint with sequence number $h$ to its log
  - And discards log information (as usual, on checkpointing)
- If $h$ is greater than the leader's current state, it also updates its current state to that of the checkpoint with sequence number $h$
- The leader may be missing:
  - The latest stable checkpoint, i.e. the service state
  - Request messages committed or prepared since that checkpoint

  These are not sent in NEW-VIEW messages
- The leader may obtain the missing information from another replica. E.g.:
  - Missing checkpoint state, $s$ can be fetched from a replica whose CHECKPOINT messages certified its correctness in $\mathcal{V}$
    - Since $f + 1$ replicas certified its correctness, at least one of these will be correct, and replica $i$ will always be able obtain $s$ or a later checkpoint

## NEW-VIEW at replicas



- A replica accepts a NEW-VIEW message for view $v + 1$, if it:
  1. Is signed properly
  2. Contains a valid **new-view certificate**
  3. Contains correct $\mathcal{O}$ and $\mathcal{N}$ sets
     - A replica checks the correctness of the $\mathcal{O}$ and $\mathcal{N}$ sets, by recomputing them from the **new-view certificate**, just like the leac does
- It enters view $v + 1$
  - After updating its state as described for the leader, if necessary
- It adds the PRE-PREPARE messages in the $\mathcal{O} \cup \mathcal{N}$ to its log
  - They are need for prepared certificates for view $v + 1$
- It multicasts a PREPARE for each message in $\mathcal{O} \cup \mathcal{N}$ to all the other replicas
  - After adding these PREPARE to its log

IMP The atomic multicast protocol for each of these requests proceeds as described earlier

- Re-execution of client requests is prevented by using stored information about replies previously sent to clients

# Correctness Arguments

## Safety

**Safety** depends on all non-faulty replicas agreeing on the sequence numbers of requests that commit locally

- For local commits at the **same view** this is ensured by the **pre-prepare and the prepare phases**
- For local commits at **different views** this is ensured by **view-change protocol

## Liveness

### Goals

- **Replicas must move to a new view** if they are unable to execute a request
- **Maximize the time interval** with at least 2f + 1 non-faulty replicas in the same view

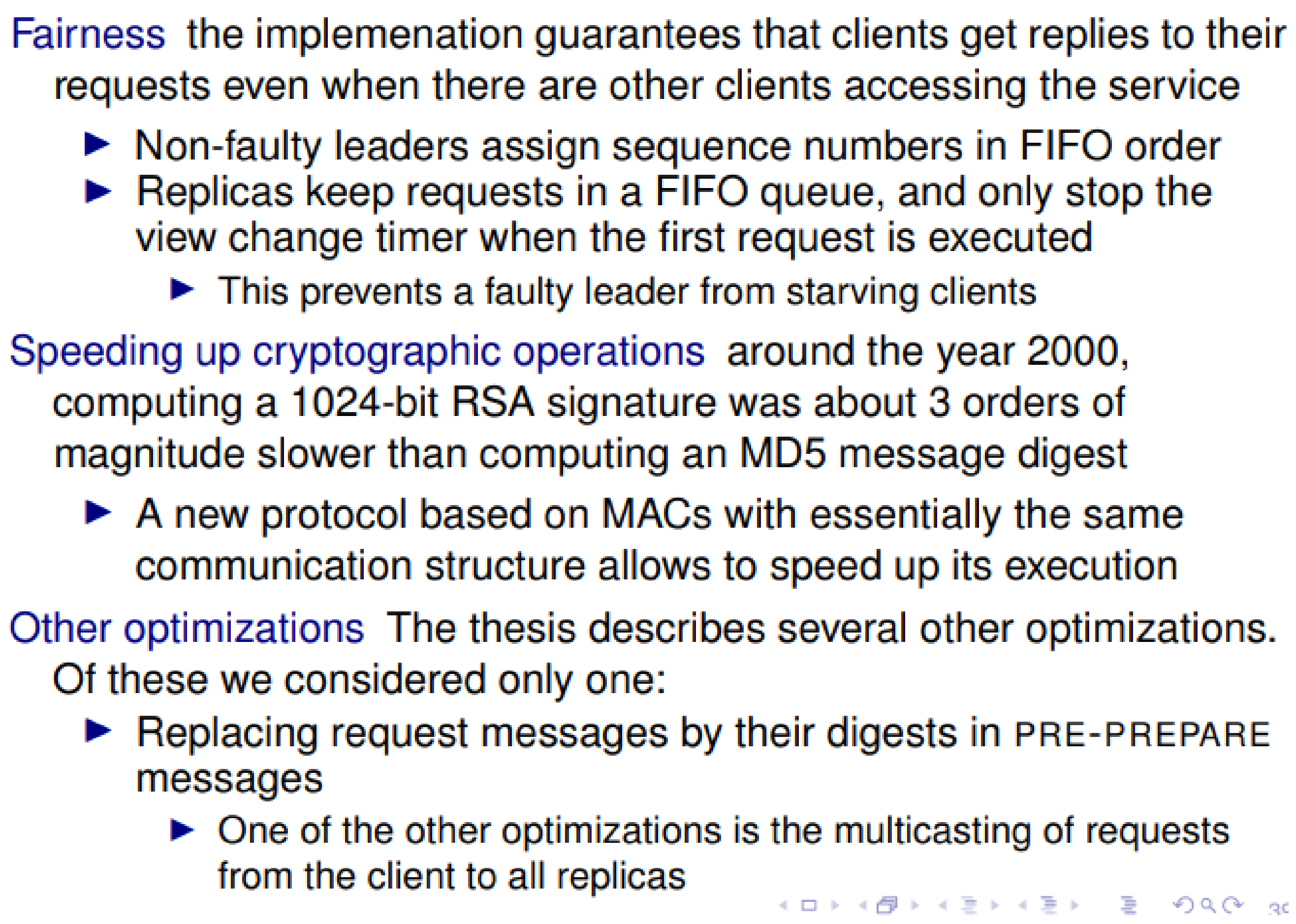
### Ensurance Measures

- **Increasing view change timeouts** to avoid starting a view change too early
- **View change without timeout**

- **Faulty replicas cannot force too-frequent view changes** unless they are leaders, because a non-faulty replica changes its view only
    - On timeout
    - On recception of f + 1 VIEW-CHANGE messages

These techniques guarantee liveness unless message delays grow faster than the timeout value indefinitely

## Final Remarks



Fairness the implemenation guarantees that clients get replies to their requests even when there are other clients accessing the service

- Non-faulty leaders assign sequence numbers in FIFO order
- Replicas keep requests in a FIFO queue, and only stop the view change timer when the first request is executed
    - This prevents a faulty leader from starving clients

Speeding up cryptographic operations around the year 2000, computing a 1024-bit RSA signature was about 3 orders of magnitude slower than computing an MD5 message digest

- A new protocol based on MACs with essentially the same communication structure allows to speed up its execution

Other optimizations The thesis describes several other optimizations. Of these we considered only one:

- Replacing request messages by their digests in PRE-PREPARE messages
    - One of the other optimizations is the multicasting of requests from the client to all replicas

## Byzantine Quorums vs. PBFT

- Compared with state machine replication, Byzantine Quorums appear to require fewer messages, by a large margin
- But the protocols we have seen assume that:
  - Either clients can be trusted
  - Or the information stored is self-verifiable
- To handle other cases, in Phalanx (the SRDS 1998 paper) Malkhi and Reiter use **consensus-objects**, which appear to require about the same number of messages as Byzantine SMR
- Furthermore, these quorum protocols support only read/write operations.
  - Although, we can build more complex operations on top of read/write operations, the number of messages will increase
  - Also, although read/write operations are atomic, and performed in the same order, consistency problems may arise when we build more complex operations on top of read/write
    - In Phalanx (the SRDS 1998 paper) Malkhi and Reiter use **mutual exclusion** objects