05_02_Consensus: State Machine Ways

1. Introduction and Background

2. Paxos

2.1. Classification

1. single-value Paxos problem

a group of processes has to **agree on a single value** as proposed by one of the participants

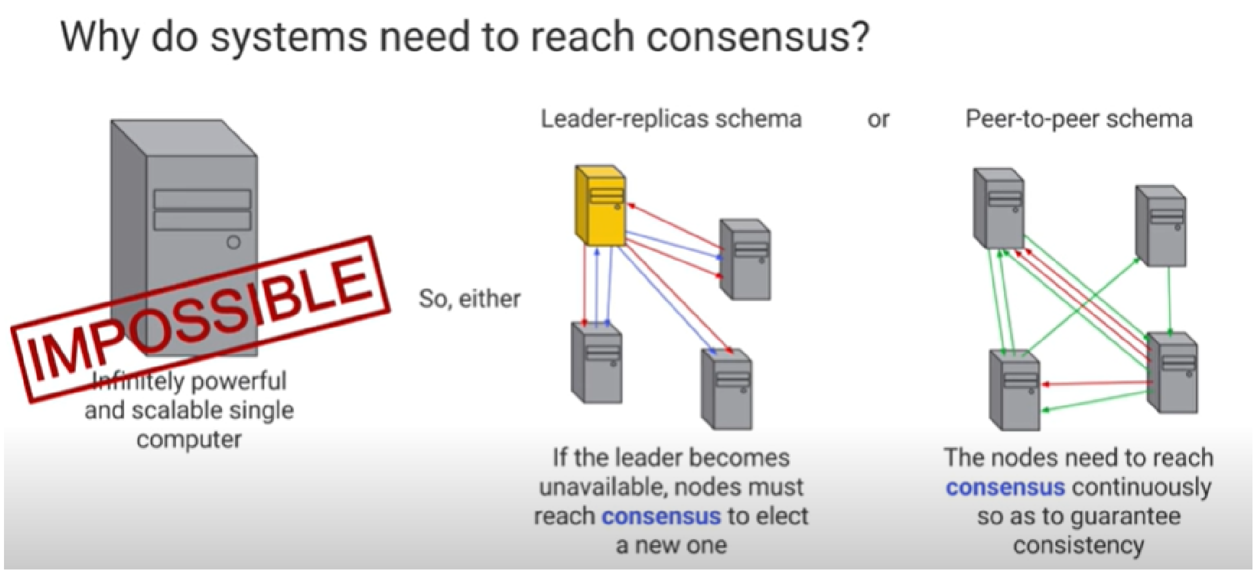
2. multi-value Paxos problem

a group of processes has to **agree on a sequence** of such values

2.2. Global Assumption

- Processes:
  - may crash or later starting, that means their only possible faulty behavior is **pausing**.
  - processes need some statble storage in order to rejoin
  - processes are **not malicious** (not Byzantine situation)
- Messages: may be
  - reordered
  - duplicated
  - received out of order
  - not corrupted
- Acknowledgement
  - Paxos nodes must know how many acceptors a majority is
  - Paxos nodes must be persistent: they can't froget what they accepted

2.3. Usage


Why do systems need to reach consensus?
IMPOSSIBLE
Infinitely powerful and scalable single computer
So, either
Leader-replicas schema
or
Peer-to-peer schema
If the leader becomes unavailable, nodes must reach consensus to elect a new one
The nodes need to reach consensus continuously so as to guarantee consistency

2.4. First stage question: Synod algorithm

2.5. Nodes Roles

- The processes have three types of roles:
    - **Proposers**: the processes that propose a value
    - **Acceptors**: the process that choose a value
    - **Learners**: the processes that learn the result

  Processes may have multiple roles

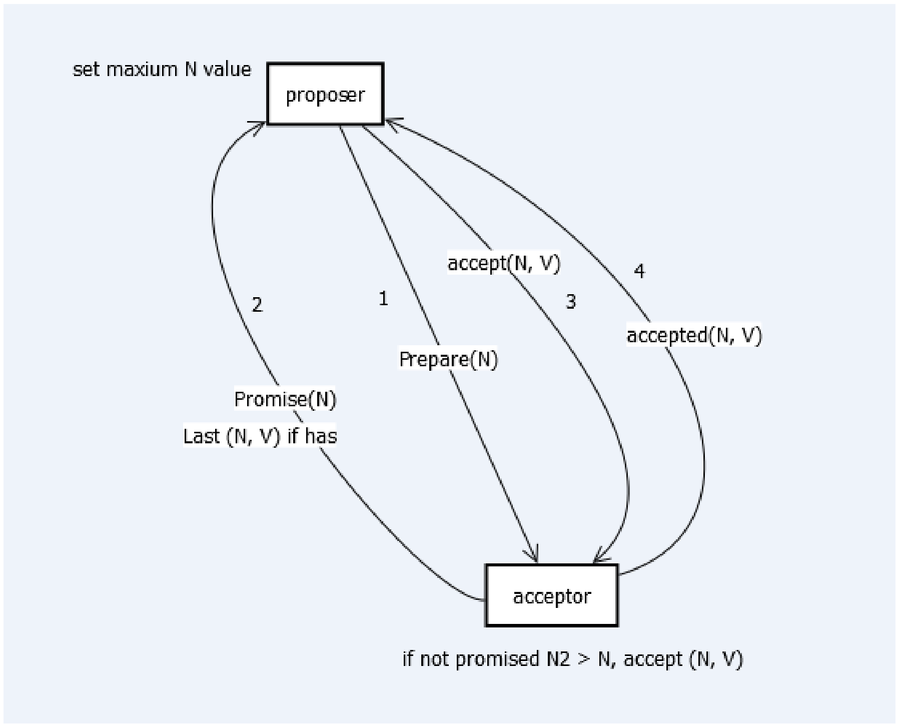
2.6.Single-value Paxos

Preparation(Structure)

- proposer:
    - **prnd:** the highest-numbered ruond it has started
    - **pval:** the value it proposes in round
- acceptor:
    - **rnd:** highest-numbered round in which it has participated
    - **vrnd:** highest-numbered round in which it has voted
    - **vval:** the value of which it voted in round **vrnd**

Main Idea

- The algorithm is structured **in rounds**
    - different rounds **do not have a specific sequence**, that means rounds can be executed simultaneously, out of order with respect to round numbers, and can even be (partially) skipped altogether.
    - each proposer has a **unique** rounds number
    - so the number can be seen as an **non-fixed ID**
- Message Types:



-

- The aglorithm proceeds in **two phases**

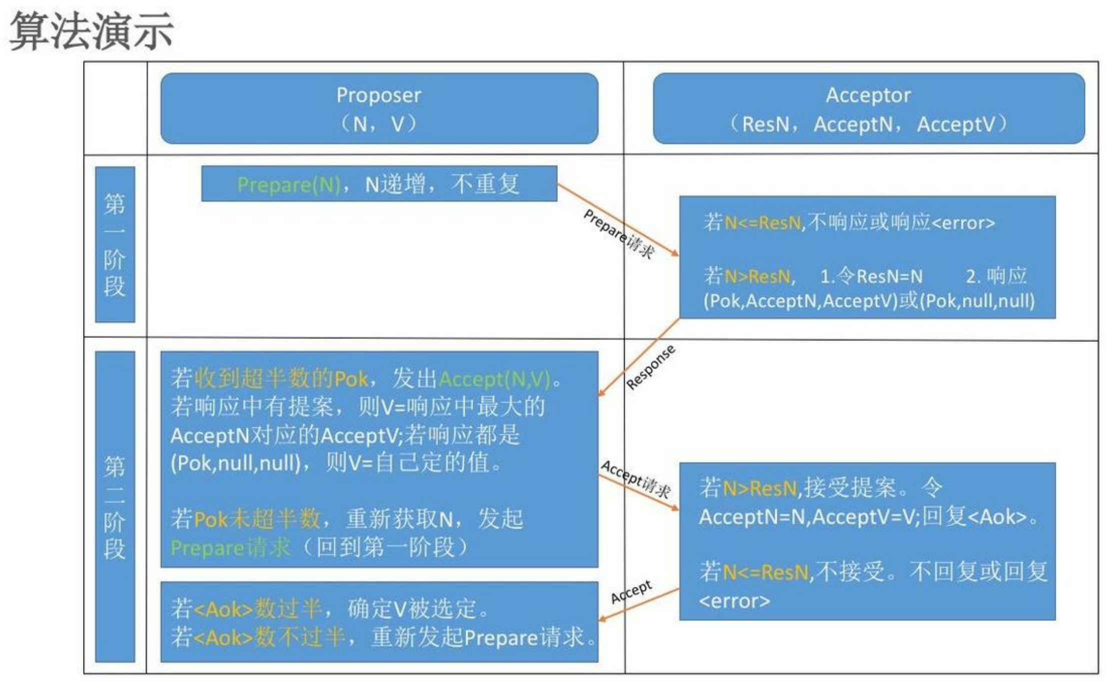
1. phase 1

- In phase 1, the **coordinator (means proposer**) of attempts to build a majority of acceptors willing to accept a proposal on its part (without already settling on the actual value to be proposed) with a **REQUEST TO PARTICIPATE message (Prepare(N))** to some set of processes
- When an **acceptor** receives **REQUEST TO PARTICIPATE message (PREPARE(N))** and the round number associated with it is **new (means new or larger than previous)** to it, it responds with a **PARTICIPATE message (Promise(N))** that includes its **latest vote (if any) and the round number of that vote**

2. phase 2:

- the **coordinator (means proposer)**
  - selects the most recent vote from **the votes of a majority of the acceptors**, and proposes the corresponding value to (a subset of) the acceptors with a **REQUEST TO VOTE (accept)** message
  - When **none of the acceptors** in the responding majority has voted sofar, the coordinator is free to pick **any value to propose (**this situation only means no responsed acceptor has voted already**)**
  - if no majority acceptor vote, increase round number, restart phase 1
- the acceptor receives a **REQUEST TO VOTE (accept)** message
  - if the round number is smaller than max round number recorded, ignore it
  - if the round number is larger, than sending a VOTE message to the **learners** if **it has not in the mean time voted for a yet more recent proposal**

3. A round, has completed successfully when **at least one learner has learnt a value** by receiving votes for it from a majority of the acceptors.


算法演示

Proposer（N，V）

Acceptor（ResN，AcceptN，AcceptV）

第一阶段

Prepare(N)，N递增，不重复

若N<=ResN,不响应或响应<error>

若N>ResN， 1.令ResN=N   2. 响应(Pok,AcceptN,AcceptV)或(Pok,null,null)

第二阶段

若收到超半数的Pok，发出Accept(N,V)。若响应中有提案，则V=响应中最大的AcceptN对应的AcceptV;若响应都是(Pok,null,null)，则V=自己定的值。

若Pok未超半数，重新获取N，发起Prepare请求（回到第一阶段）

若N>ResN,接受提案。令AcceptN=N,AcceptV=V;回复<Aok>。

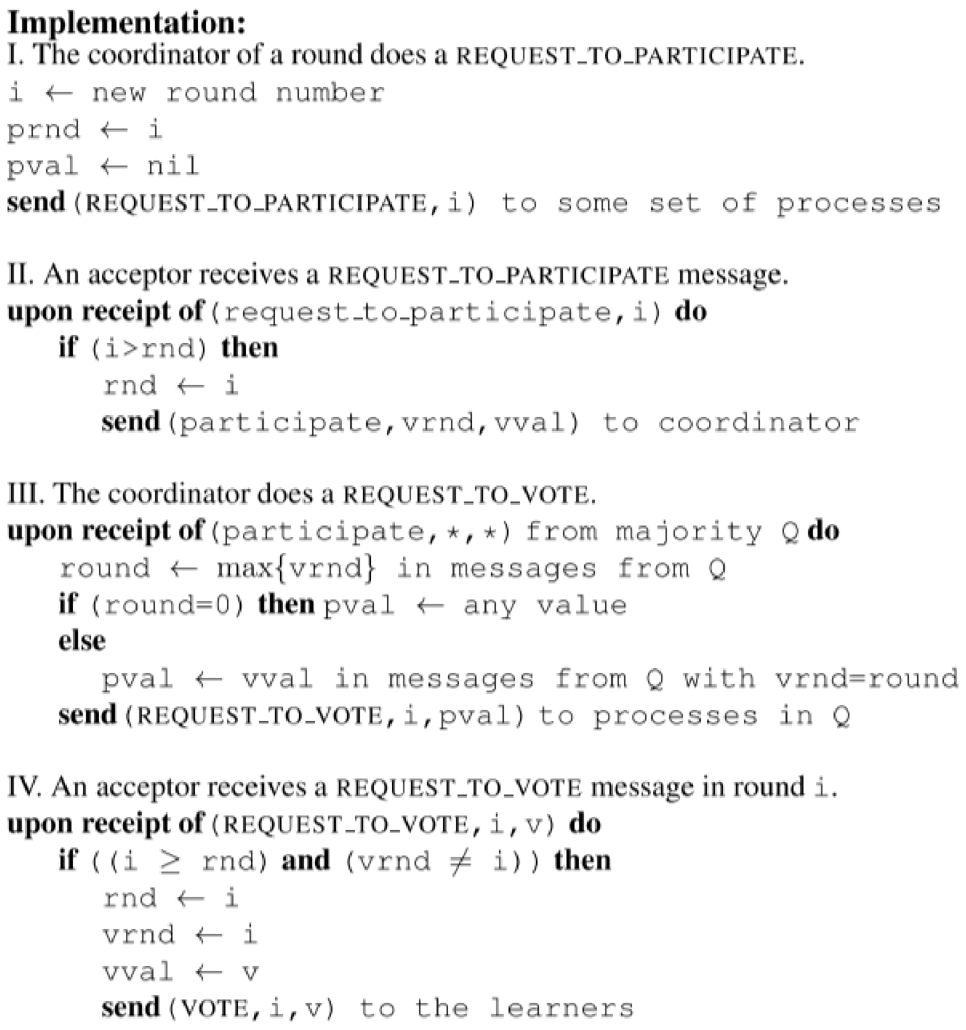
若N<=ResN,不接受。不回复或回复<error>

若<Aok>数过半，确定V被选定。
若<Aok>数不过半，重新发起Prepare请求。

Important Points

- All messages pertaining to a certain round are identified by the number of the round to which they belong
- processes **ignore message** that carry a round number smaller than the largest they know about. Moreover, as soon as a process receives a message with a round number larger than any it has seen sofar, **it skips sending any message related to previous rounds**
- nodes do not agree on round numbers, they final agree on a value
- a coordinator may in phase 2 send a REQUEST TO VOTE to an acceptor that has not indicated its willingness to participate in phase 1

Implementation



Implementation:

I. The coordinator of a round does a REQUEST_TO_PARTICIPATE.

```
i ← new round number
prnd ← i
pval ← nil
send(REQUEST_TO_PARTICIPATE,i) to some set of processes
```

II. An acceptor receives a REQUEST_TO_PARTICIPATE message.

```
upon receipt of(request_to_participate,i) do
  if (i>rnd) then
    rnd ← i
    send(participate,vrnd,vval) to coordinator
```

III. The coordinator does a REQUEST_TO_VOTE.

```
upon receipt of(participate,*,*) from majority Q do
  round ← max{vrnd} in messages from Q
  if (round=0) then pval ← any value
  else
    pval ← vval in messages from Q with vrnd=round
  send(REQUEST_TO_VOTE,i,pval) to processes in Q
```

IV. An acceptor receives a REQUEST_TO_VOTE message in round i.

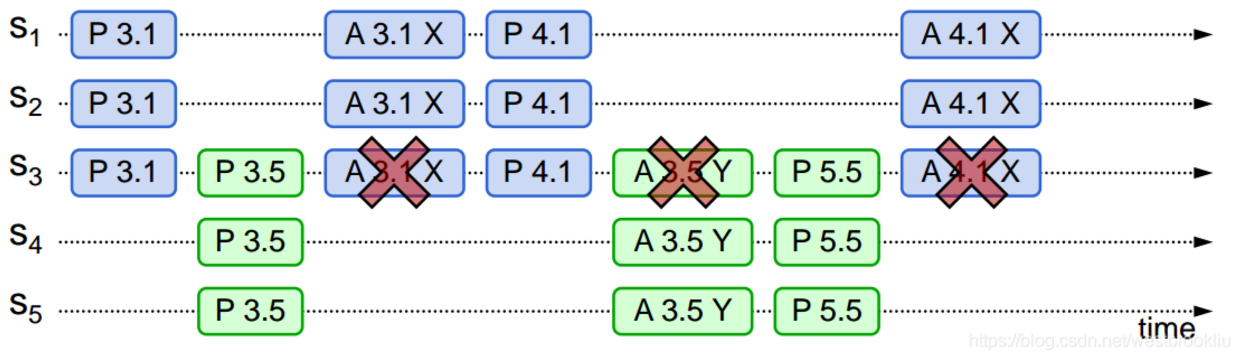
```
upon receipt of(REQUEST_TO_VOTE,i,v) do
  if ((i ≥ rnd) and (vrnd ≠ i)) then
    rnd ← i
    vrnd ← i
    vval ← v
    send(VOTE,i,v) to the learners
```

Intuition

- It is simple to understand if proposer do not overlap their round
- For overlapping situation:
    - if a smaller round number proposer first capture majority, the later larger one will receive the value of the node with smaller round number from accpetors. Then the smaller one cannot receive enough Accepeted message. But the larger one will, and the value will still achive agreement

Livelock

Livelock is still going but no possible to stop follow this step, always trapped in a cycle



Correctness

We do not fully prove the correctness, here, we try to prove: When the coordinator proposes value v in round i, in no previous round (means smaller round number) another value than v has been or can still be decided

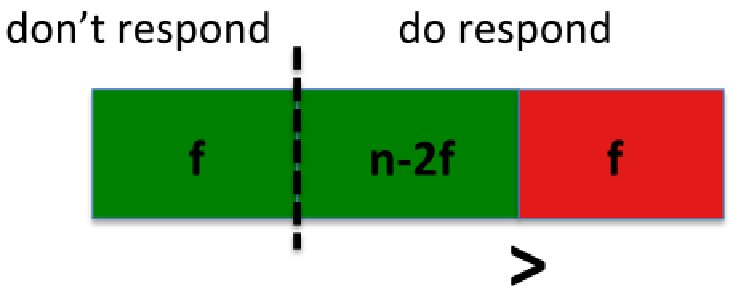
Proof:

suppose pval=v in round i, and v was last voted for by an acceptor in a majority Q in round k < i; consider any previous round with number j (j < i)

3. Two Other Ways

3.1. Assumptions

- Required **n>3f**
- When waiting for reply from **all** replicas, proceed after **n-f** replies
    - Replies from non-faulty ones should out-number replies from faulty ones



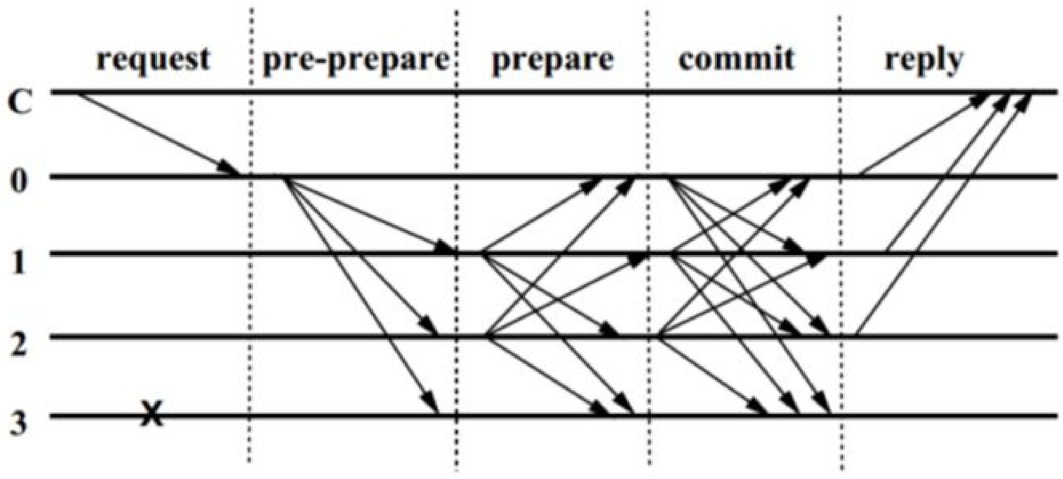
    - Replicas numbered 0,1, …, n-1 and **n=3f+1**

3.2. Practical Byzantine Fault Tolerance (PBFT)

3.2.1. High-level process

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

3.2.2. Normal Process (Primary does not fail)



PBFT算法流程

Three Phases:

- pre-prepare
- prepare
- commit
- All three types of messages contain a **view number** and a **request number**

<u>Accepting a pre-prepare</u>

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

If **accept**, then it enters the prepare phase and broadcasts a **prepare** message

<u>Get prepared</u>

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

When **prepared(m,v,n,i) is true**, replica i broadcasts a **commit** message

Get commited

Predicate **committed-local(m,v,n,i)** is true if:

- prepared(m,v,n,i) is true
- replica i has accepted **2f+1** commit messages (then it executes the request)

Predicate **committed(m,v,n)** is true if

- prepared(m,v,n,i) is true in at least **f+1** correct replicas

After **committed,** backup will reply to the request

Checkpoint

- **checkpoint** contain the state after the execution of every K client requests for some generally known integer K
- Replicas **broadcast checkpoints** in separate messages containing a digest of the corresponding state and the number of the last request executed in it
- a replica has received **2f +1 matching** checkpoint messages (proof for the correctness of the checkpoint), which is then called **stable**.
- Upon a checkpoint becoming stable:
    - discard previous checkpoints
    - discard all messages related to earlier requests

3.2.3. (View Change)The primary is suspected to be failed

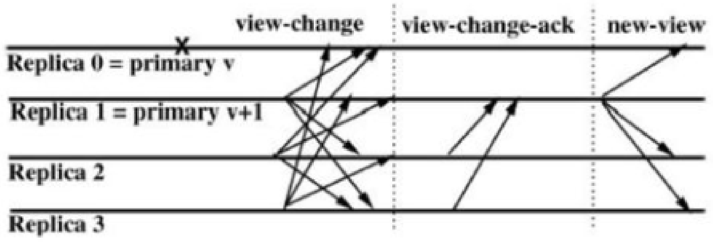


Fig. 2. View-change protocol: the primary for view v (replica 0) fails causing a view change to view $v + 1$.

- 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 multicast** the request to the backups, it is suspected of failure
  - The backups then initiate a **view change**

<u>View Change</u>

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 stable checkpoint
- for every request that prepared at the backup with **request number higher than n**, the corresponding **pre-prepare** message and **2f prepare** messages

<u>New View</u>

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:**

- derive on which of these messages it still has to act

    - 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 (from pre-pare messages contains in the new-view message "**catch up with faster nodes**")
    - it may have to retrieve requests or checkpoints from other replicas

Broadcast by new primary:

- “minimum” of histories of backups
- every backup compares its own history with this “minimum”

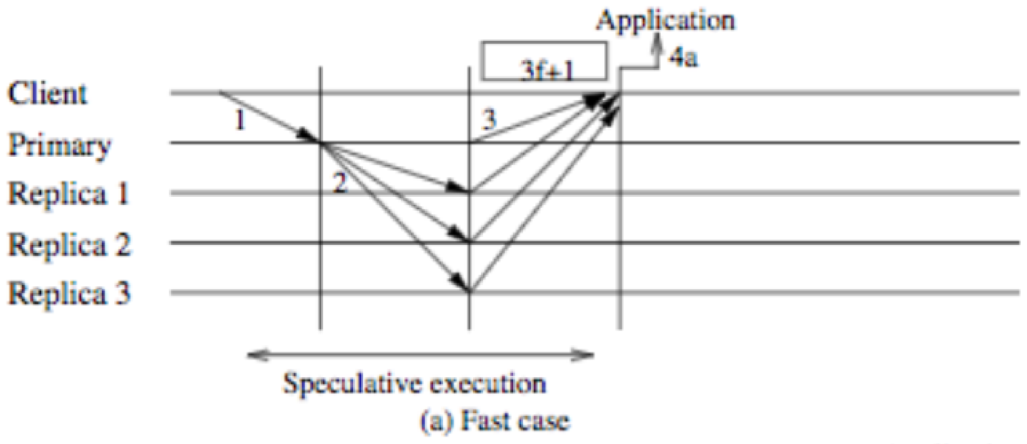
3.2.4. Understanding

3.3. Kotla et al.’s Speculative Byzantine Fault Tolerance (Zyzzyva)

3.3.1. Basic idea(improvement)

it places part of the burden of resolving conflicts on the clients

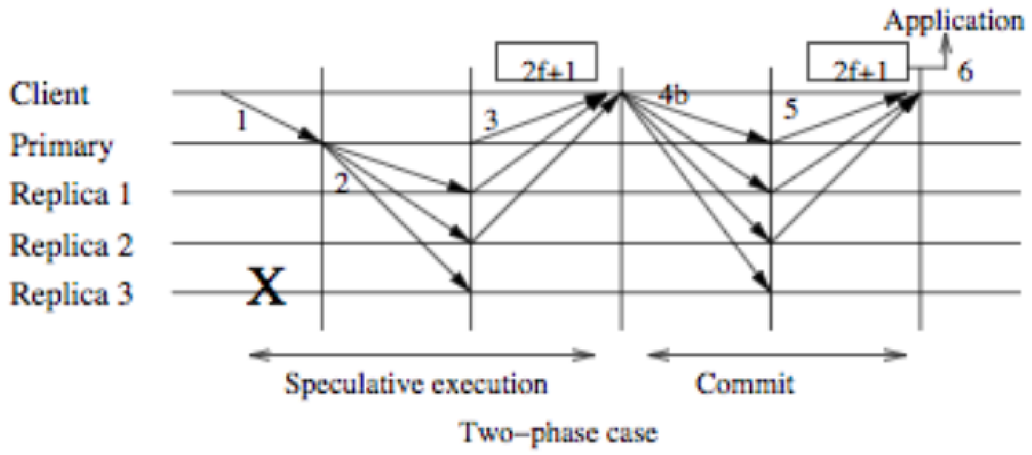
3.3.2. Normal Case



(a) Fast case

1. A client **sends its request** to the primary
2. The primary assigns a **sequence number** to request and **forwards** it to the backups
3. The replicas **execute** the request and **respond** to the client (response includes history at replica)
4. The client **receives 3f+1** matching responses from the replicas

3.3.3. Failure/Delay Case 1



receives **between 3f and 2f+1** matching responses from the replicas (thus from a majority of the correct replicas),

- Because lacking of communication between replicas, replicas are not aware of this quorum of matching replicas
- The client sends a **commit certificat**e to all replicas (with proof of order)
- The client waits for **2f+1 local-commits** of the certificate from the replicas

when the client has received at least **2f +1 such messages**, it again considers the request to be **completed**

3.3.4. View Change Case

The client receives **fewer than 2f+1** matching responses from the replicas

- The client **sends its request** to all replicas
  - if the primary still reacts, fine
  - Otherwise the backups will invoke a **view change**
- A replica initiates a **view change** by sending an **I-hate-the-primary** message to all replicas
- When a replica receives **f+1 such messages**, it
  - commits to the view change
  - sends the **I-hate-the-primary messages** plus a **view-change message** to all replicas
- When the new primary receives **2f+1 view-change** messages, it
  - broadcasts **a new-view message**,
  - along with the supporting view-change messages
- When a replica receives **2f+1 view-change** messages, it
  - waits for a **new-view message**

    - if that takes too long, it **initiates** a change to **view v+2**