Basil: Breaking up BFT with ACID (transactions)

Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, Natacha Crooks

Presented By:
Radhika Gupta, Disha Narayan, Arvind S, Siva sai kumar P

UCDAVIS
COMPUTER SCIENCE

Current System and Its Problem

COMMUNICATION

- Redundant coordination across shards and replicas, leading to performance bottlenecks.

- Concentration of power in shard leaders, raising fairness concerns.

- Restrictions on transaction expressiveness due to the need for known read and write sets.

FLEXIBILITY

What is Basil?

- Leaderless
- Byzantine Fault Tolerant key-value store
- Partial Synchrony
- Single Round Trip

Current Problem	**Basil Solution**
Redundant Coordination	It integrates the process of distributed commit with the process of replication
Leader Dominance within Shards	Shifts transaction execution responsibility to clients
Restriction on Transaction Expressiveness	Support for general interactive transactions without the need for prior knowledge of reads and writes.

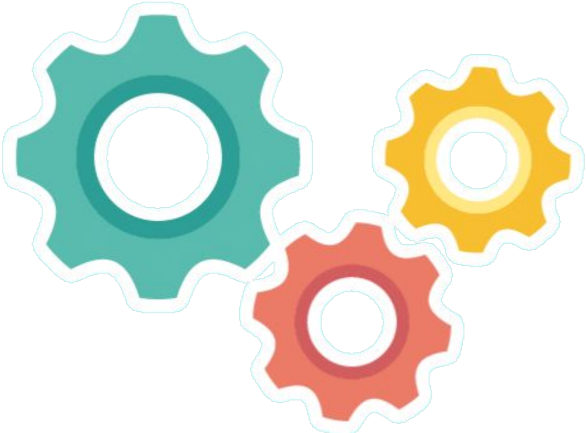
Foundation of Basil

- **Byzantine Isolation**
  Focuses on safety

- **Byzantine Independence**
  Focuses on liveness

Group Project with
Byzantine actor

Byzantine Isolation



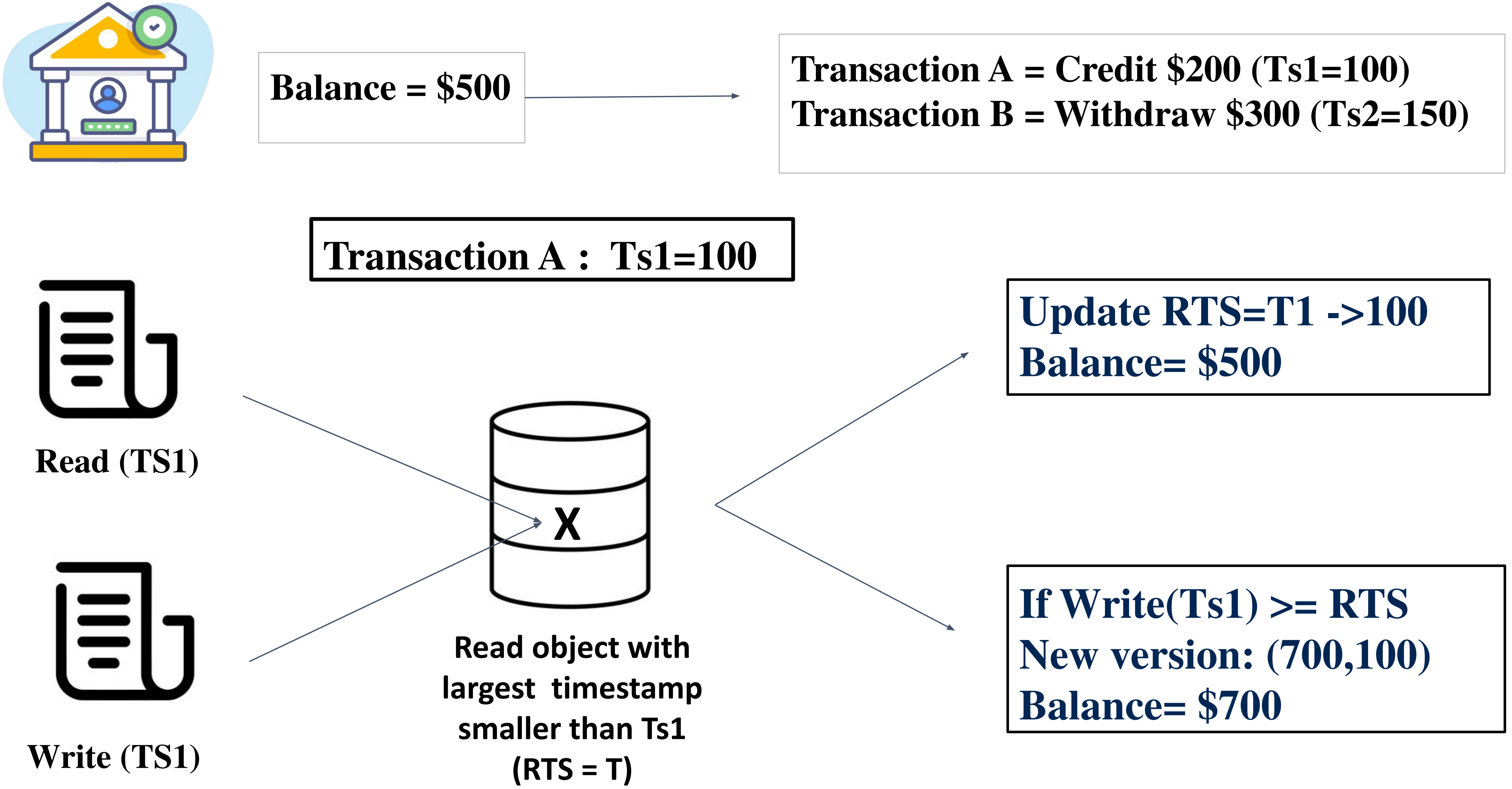
Byzantine Independence

MVTSO
Multiversion Timestamp Ordering

- **Optimistic concurrency control technique**

- **Maintains serializability**

Multiversioned Time stamped Ordering



Balance = $500

Transaction A = Credit $200 (Ts1=100)
Transaction B = Withdraw $300 (Ts2=150)

Transaction A : Ts1=100

Read (TS1)

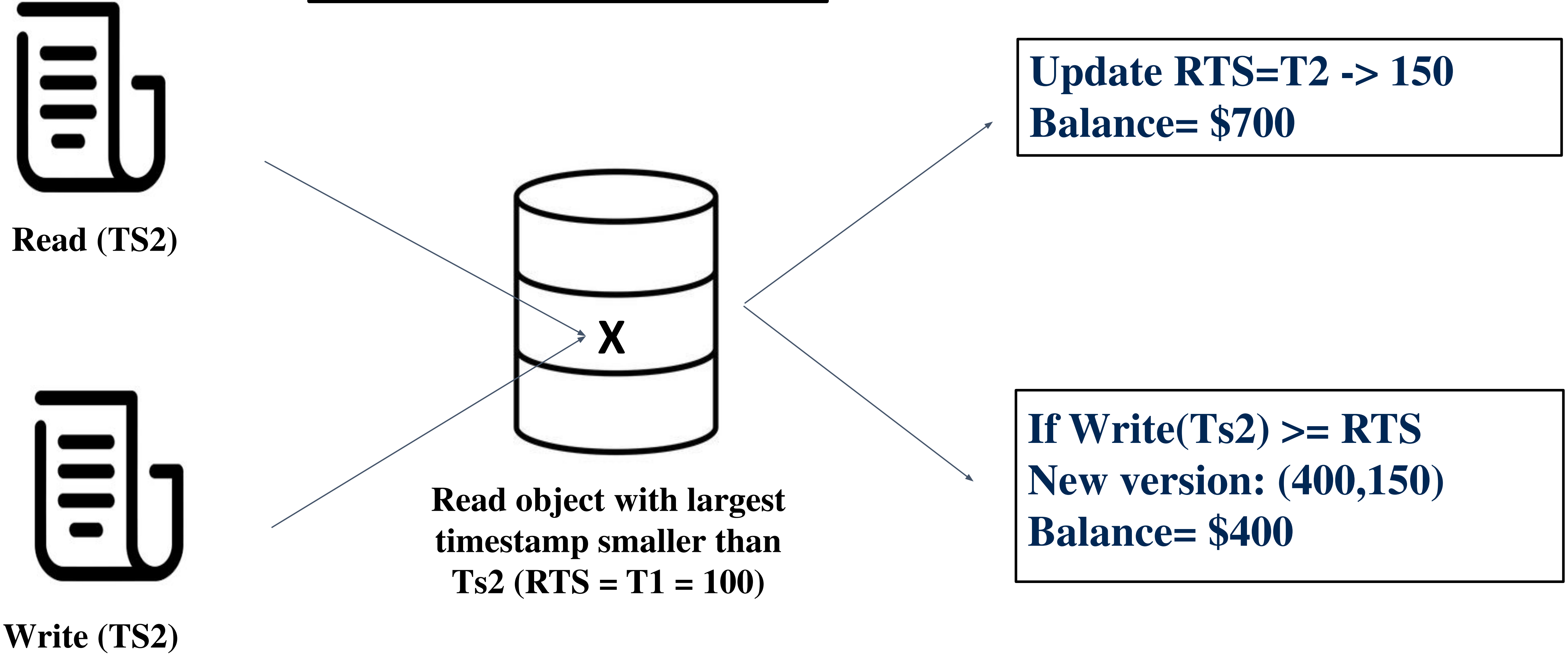
Write (TS1)

Read object with largest timestamp smaller than Ts1 (RTS = T)

Update RTS=T1 ->100
Balance= $500

If Write(Ts1) >= RTS
New version: (700,100)
Balance= $700

Transaction B : Ts2=150



Read (TS2)

Write (TS2)

Read object with largest timestamp smaller than Ts2 (RTS = T1 = 100)

Update RTS=T2 -> 150
Balance= $700

If Write(Ts2) >= RTS
New version: (400,150)
Balance= $400

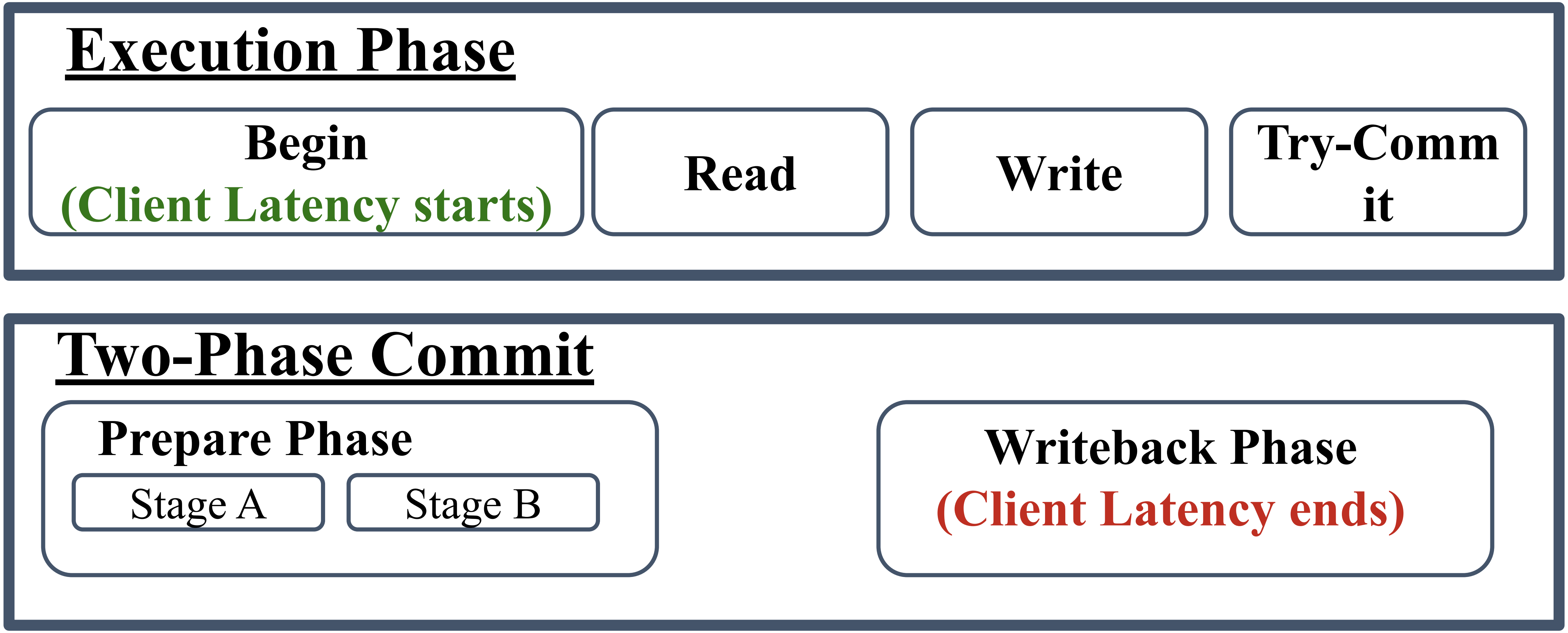
Drawbacks of MVTSO:

1. **Manipulation of Timestamps.**

2. **Transactions dependent on Uncommitted Transactions gets blocked too.**

   [Leaves open the possibility that blocked transactions may be rescued and brought to commit]

System Overview



Execution Phase

Begin
(Client Latency starts)

Read

Write

Try-Commit

Two-Phase Commit

Prepare Phase

Stage A

Stage B

Writeback Phase
(Client Latency ends)

How execution in Basil resolves Drawback 1?

Begin()

- Ts := (Time, ClientID)
- Accept Transaction if Ts is no greater than *R*Time + δ
- *R*Time= Replica own local clock.
  δ = skew of NTP's clock

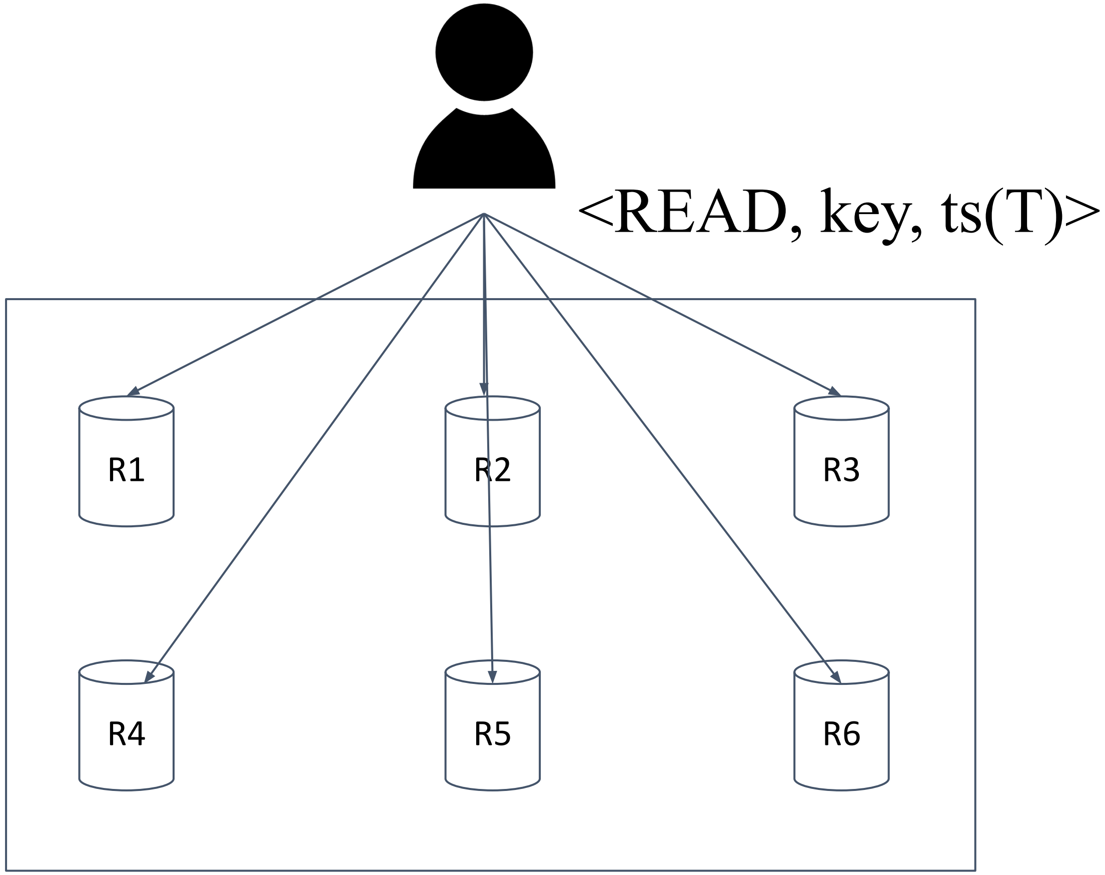
How execution in Basil resolves Drawback 2?

Write (key, value)

- Buffering writes locally.
- Visible during Prepare Phase.

Read(key)

1. **C→R: Client C sends read requests to replicas.**

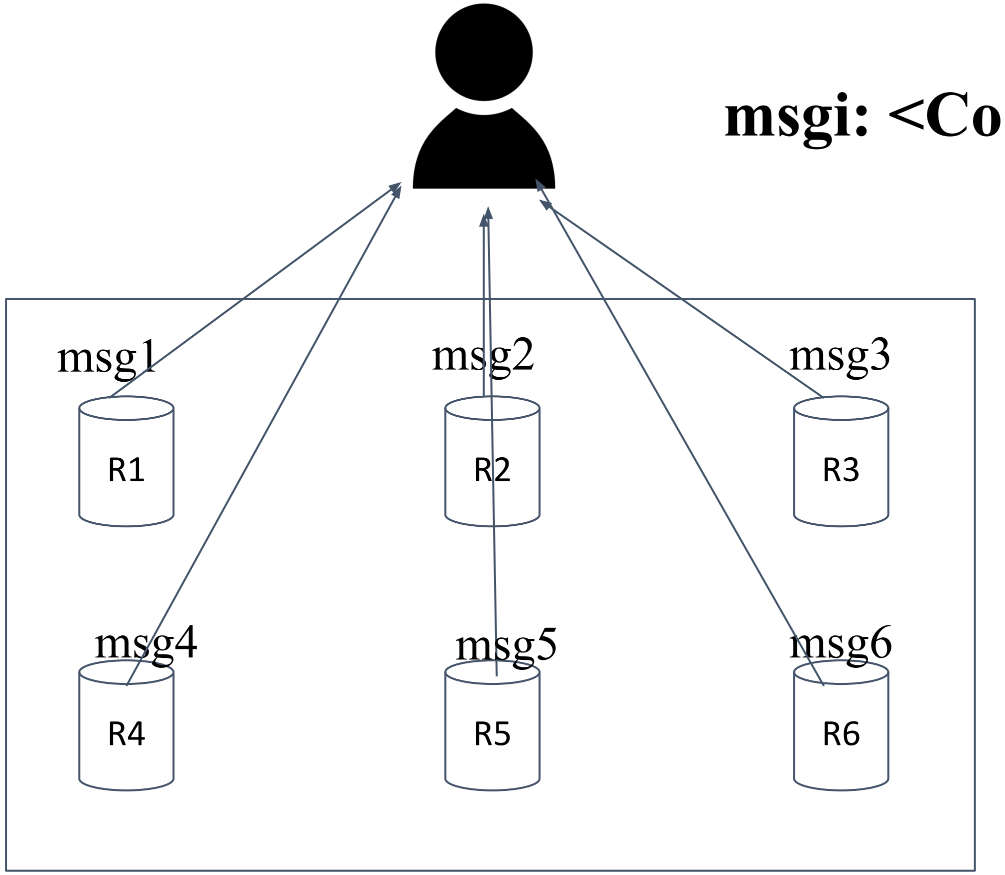


Shard S

Read(key)

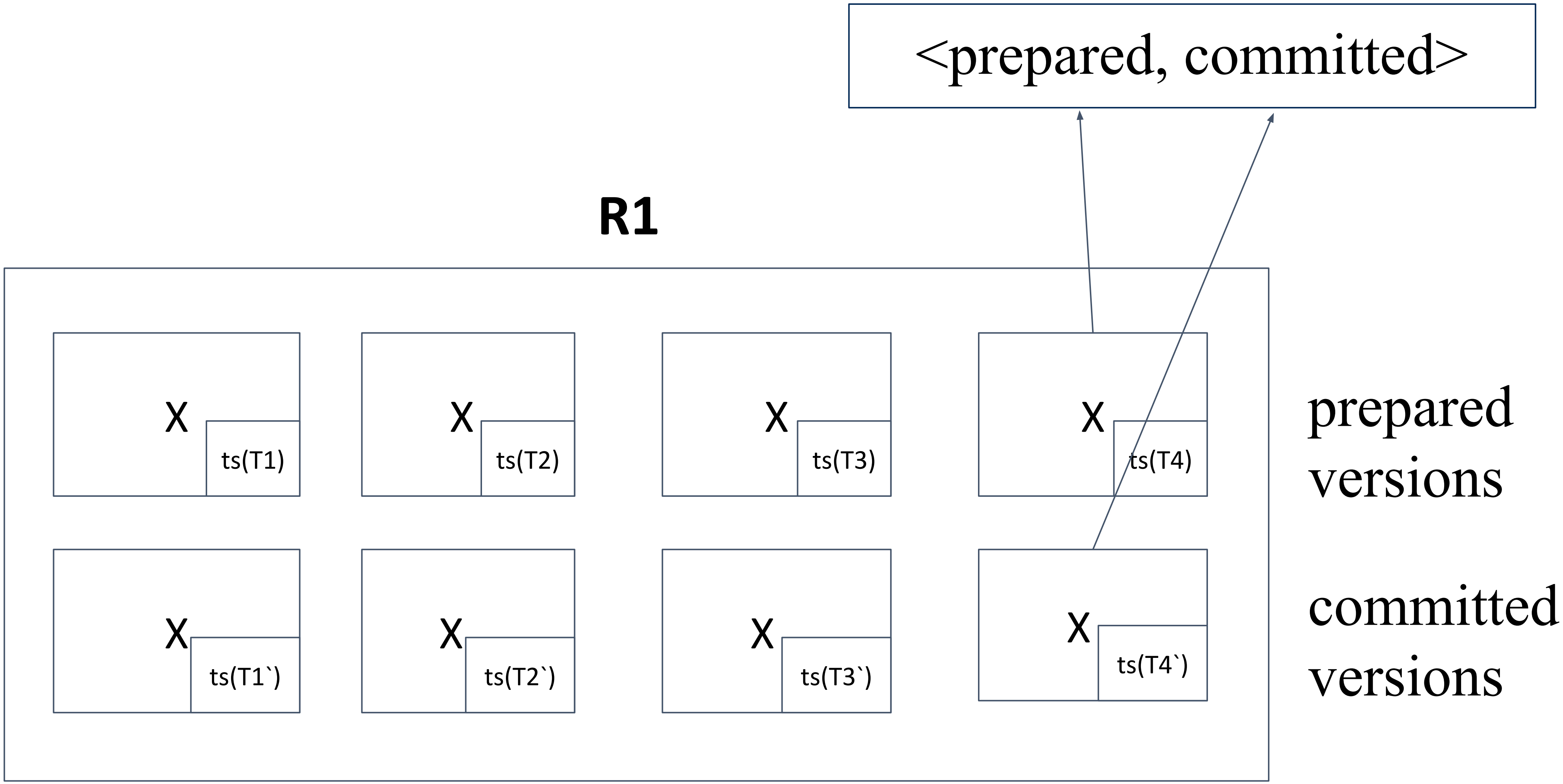
2. R→C: Replica processes client read and replies

msgi: <Committed,Prepared>Ri



Shard S

UCDAVIS COMPUTER SCIENCE



<prepared, committed>

R1

prepared versions

committed versions

What Msg contains?

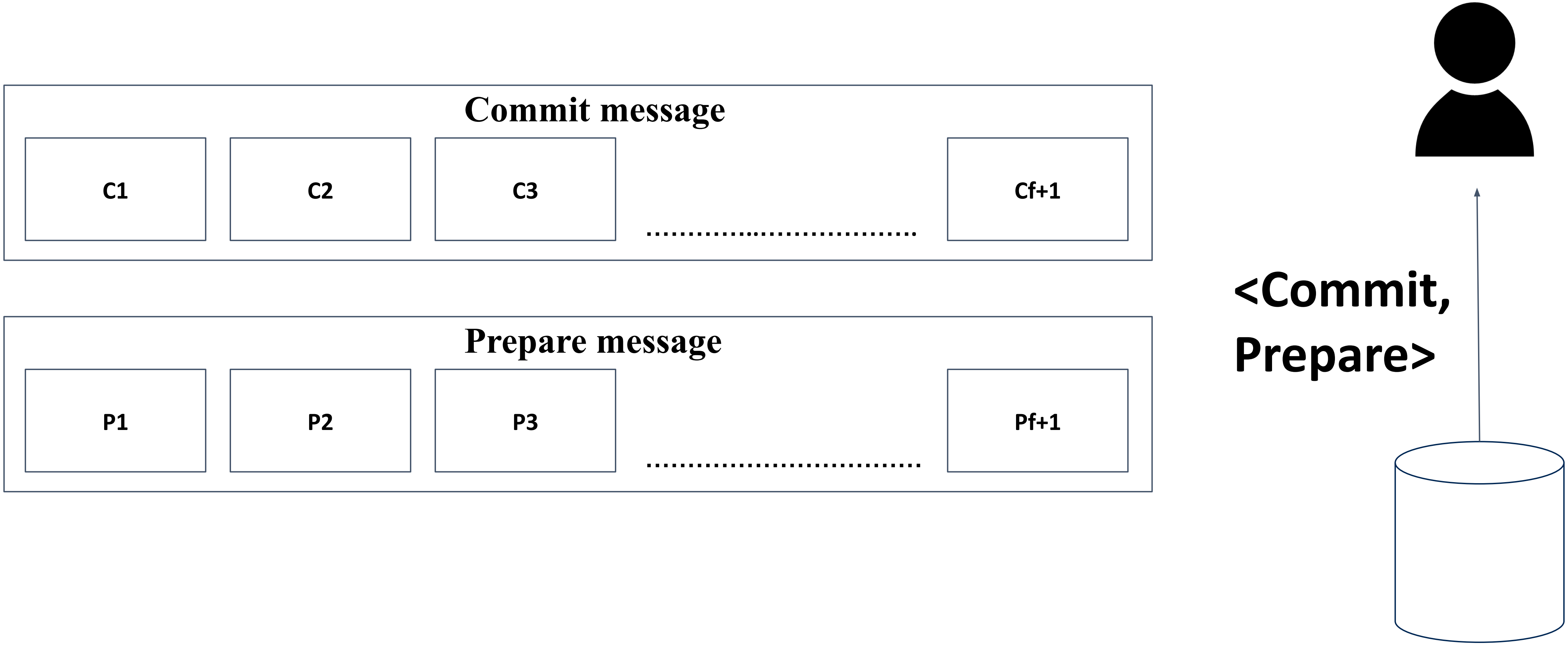
Committed Version

Version	C-CERT

Prepared Version

Version	id(T`)	Dep(T`)

Client chooses highest timestamped version



Commit message

C1 C2 C3 .... Cf+1

Prepare message

P1 P2 P3 .... Pf+1

<Commit, Prepare>

Commit Version

Read Set
(Key, Version)

Prepared Version

Read Set
(Key, Version)

Dependency Set
(Version, id_t)

Try Commit

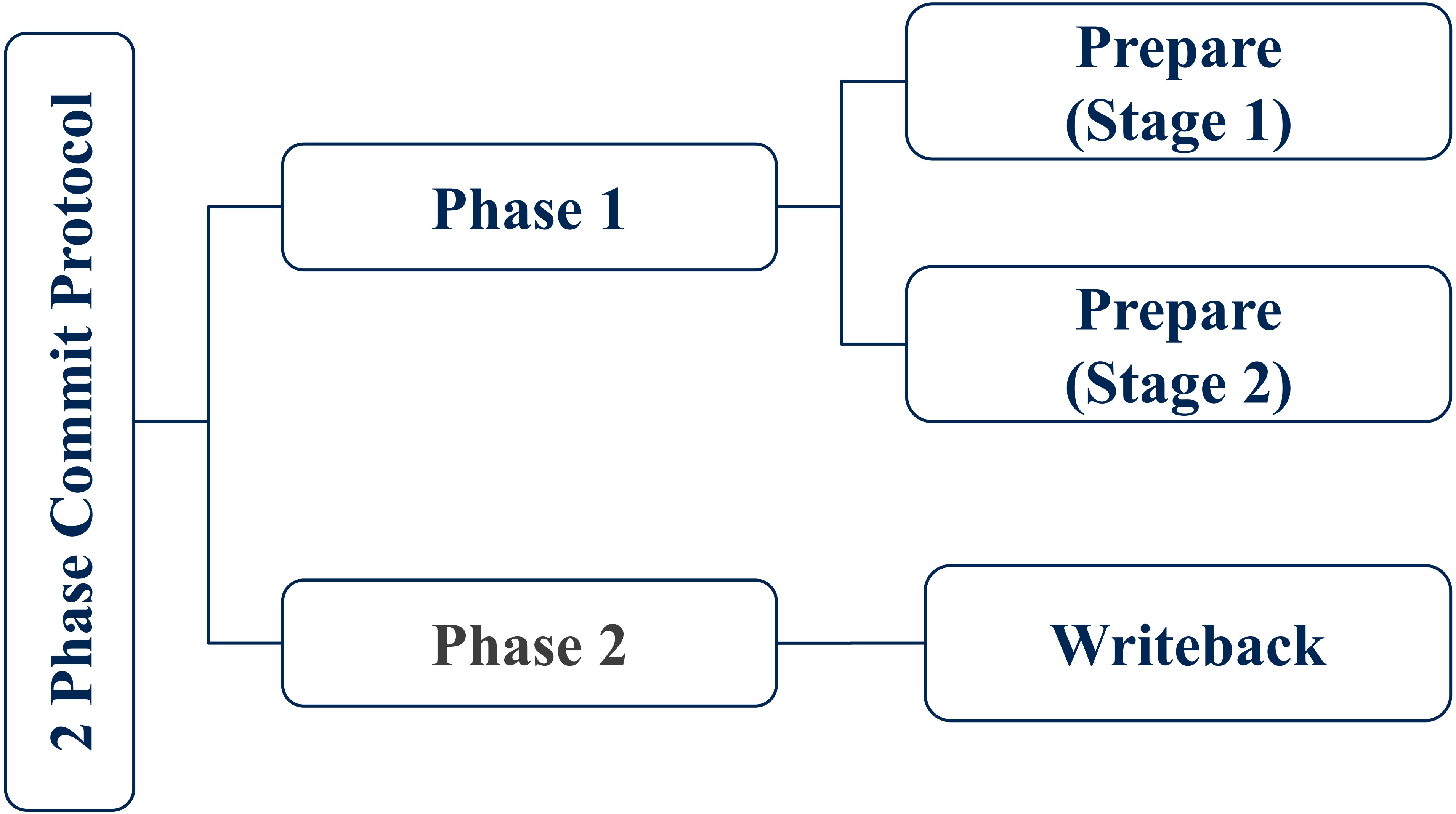
Abort()

- Remove RTS from all keys in ReadSet(T)
- No changes for writes as it buffers during execution.

Commit()

2 Phase Protocol

Commit in BASIL



2 Phase Commit Protocol

Phase 1

Prepare (Stage 1)

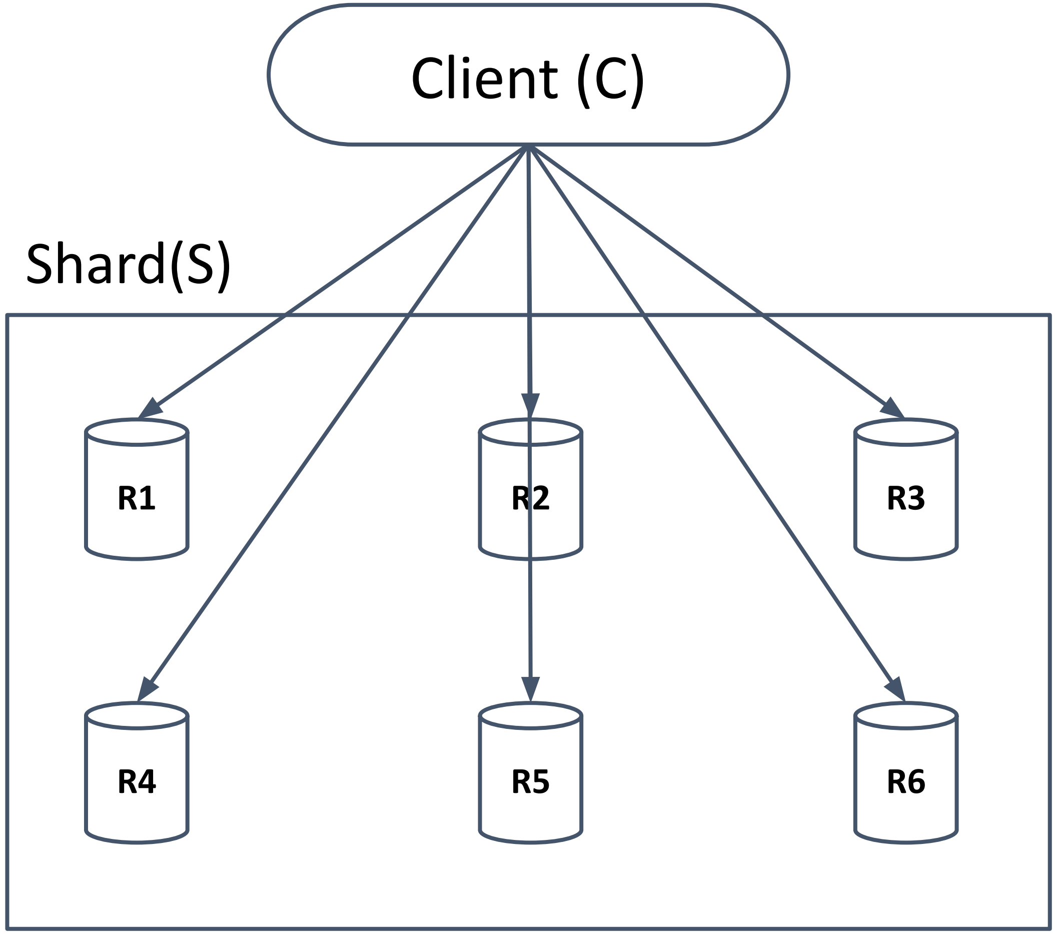
Prepare (Stage 2)

Phase 2

Writeback

Prepare Phase (Stage 1)

1. **C→R: Client(C) sends an authenticated ST1 request to all replicas(R) in Shard(S)**



ST1 := <PREPARE, T>

ts_T
ReadSet$_T$
$WriteSet_T$
Dep_T
id(T)

Transaction Metadata

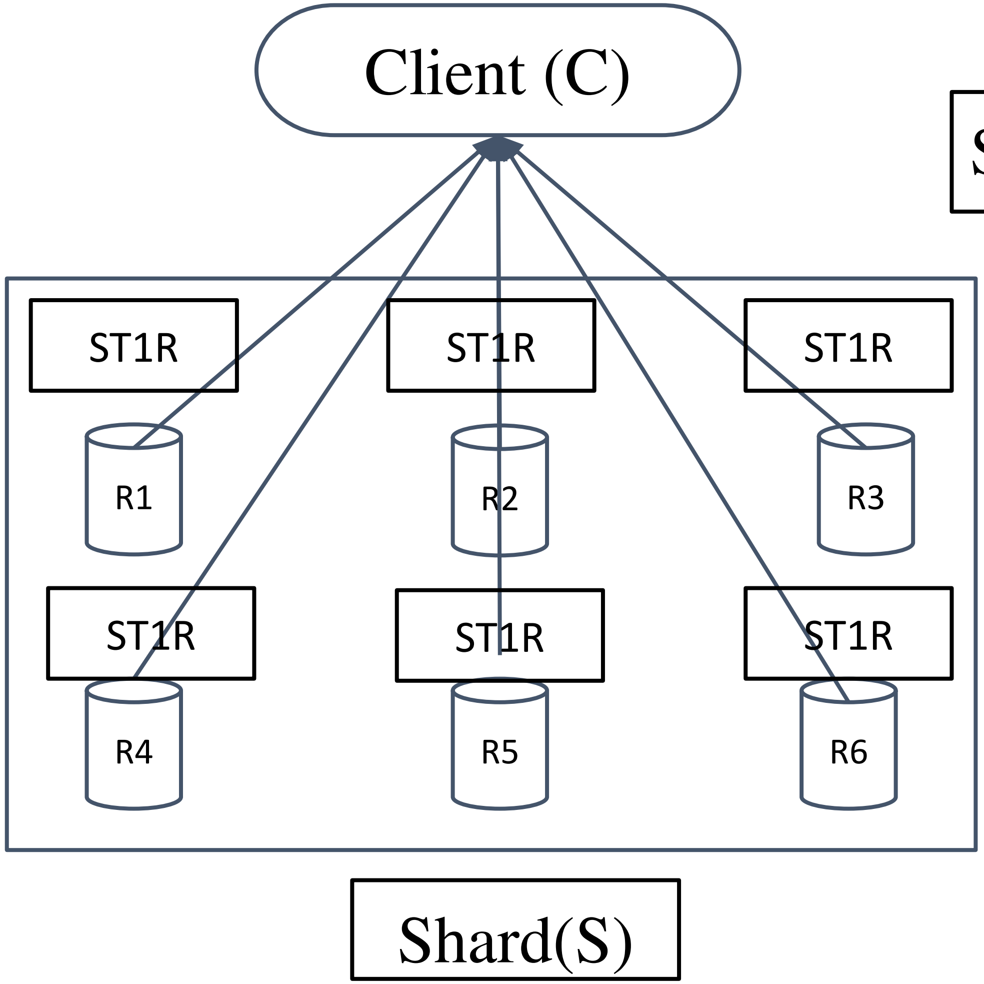
Prepare Phase (Stage 1)

2. R←C: Replica R receives a ST1 request and executes concurrency control check.

Basil thus runs an additional concurrency control check to determine whether a transaction *T* should commit and preserve serializability

Prepare Phase (Stage 1)

3. R→C: Replica returns its **vote** in a ST1R message.



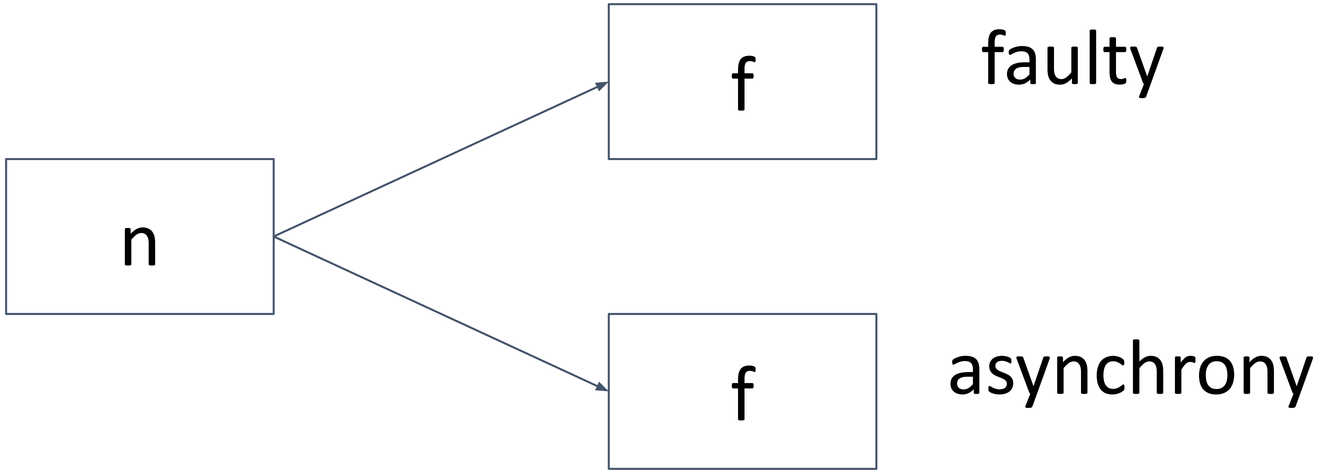
ST1R := <T, vote>

Prepare Phase (Stage 1)

4. C←R: The client receives replicas' votes.

- Client(C) waits for ST1R messages from the replicas of each shard S touched by T.
- C decides whether shard S voted to commit or abort the transaction T.
- C marks the Shard(S) as:
    - **Fast Shard**: Votes are contributed to V-CERT.
    - **Slow Shard**: Votes are contributed to record in a vote tally.

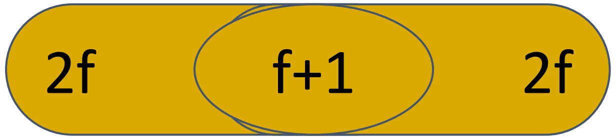
Why 5f + 1 replicas in a shard?



- We need (n-2f) votes to form the Commit Quorum
- And we need at least 1 correct replica to overlap.

Prepare Phase (Stage 1)

Commit Slow Path (3f+1 <= Commit votes < 5f+1):



- When a **Commit Quorum(CQ)** is reached. (at least one honest replica will ensure that two CQs do not independently commit conflicting Ts.)
- Client C adds S to set of ‘slow shards’ and records the votes in a vote tally. (to make the decision durable)

UCDAVIS
COMPUTER SCIENCE

Prepare Phase (Stage 1)

Abort Slow Path (f+1 <= Abort votes < 3f+1):

- When an **Abort Quorum(AQ)** is reached. (ensures at least one honest replica thinks T is a conflicting transaction)
- Client C adds S to set of ‘slow shards’ and records the votes in a vote tally. (to make the decision durable)

Prepare Phase (Stage 1)

Commit Fast Path (5f+1 Commit votes)

V-CERT
<id(T),S,COMMIT,{ST1R}>

- All replicas in shard S vote to commit.
- C records the votes from S into a Vote-certificate**(V-CERT)** (makes the decision durable)
- C adds S into the set of ‘fast shards’

Prepare Phase (Stage 1)

Abort Fast Path (Abort votes >= 3f+1)

V-CERT
<id(T),S,ABORT,{ST1R}>

- At least 3f+1 abort votes.
- C records the votes from S into a Vote-certificate**(V-CERT)** (makes the decision durable)
- C adds S into the set of ‘fast shards’

Prepare Phase (Stage 1)

Abort Fast Path
(C-CERT for a conflicting transaction T`)

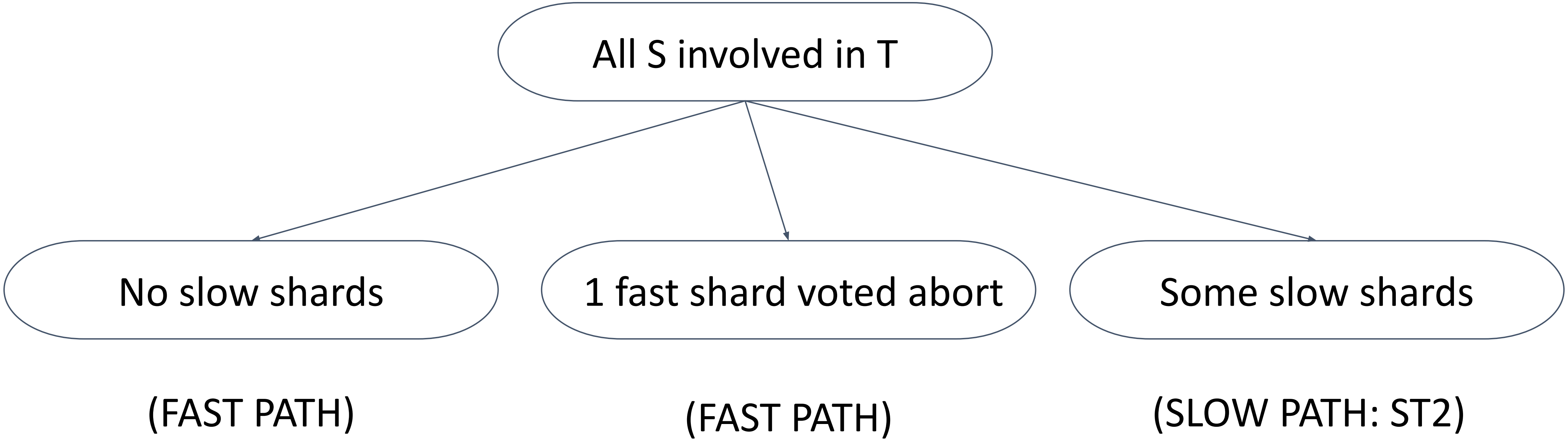
V-CERT

<id(T),S,ABORT,id(T`),C-CERT>

- Client C receives an abort vote and a Commit certificate for a conflicting transaction T` from shard S.
- C creates a Vote-certificate(V-CERT) for shard S (makes the decision durable)
- C adds S into the set of ‘fast shards’

Prepare Phase (Stage 1)

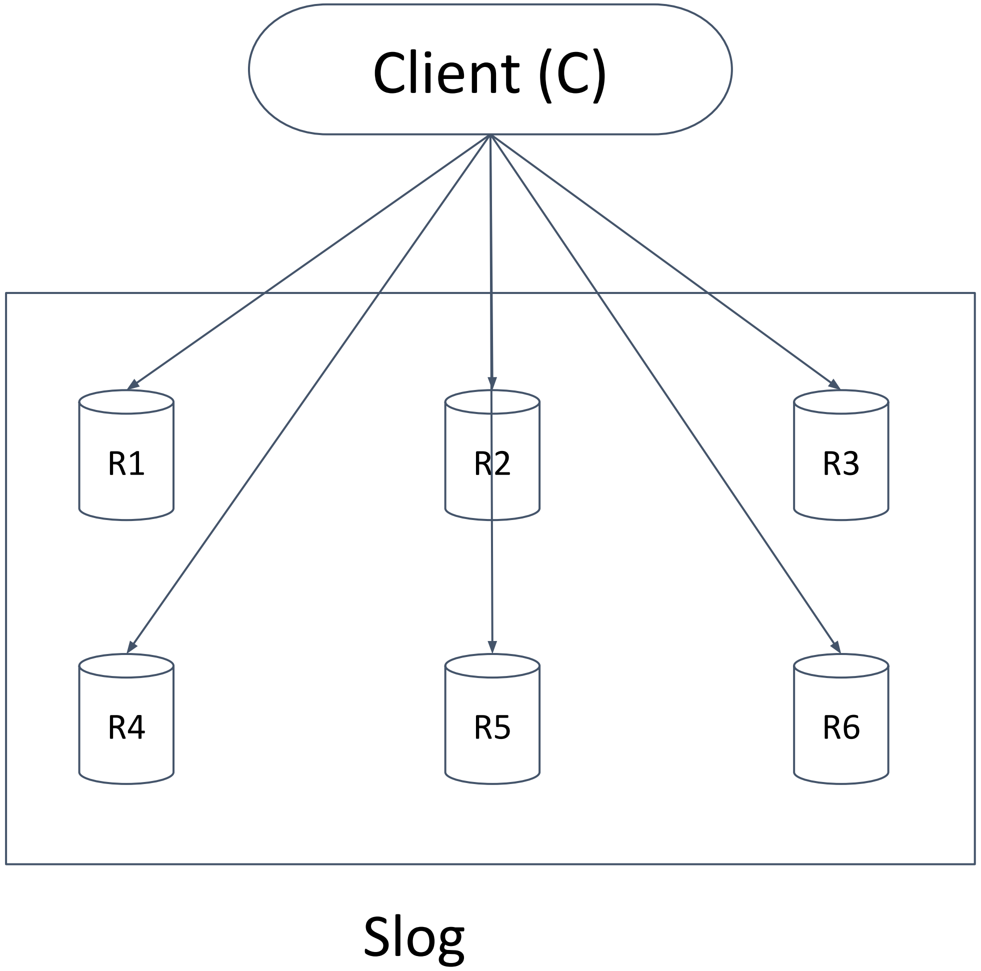
- After all shards voted, C decides whether to commit or abort T. (by making the decision durable)



Prepare Phase (Stage 2)

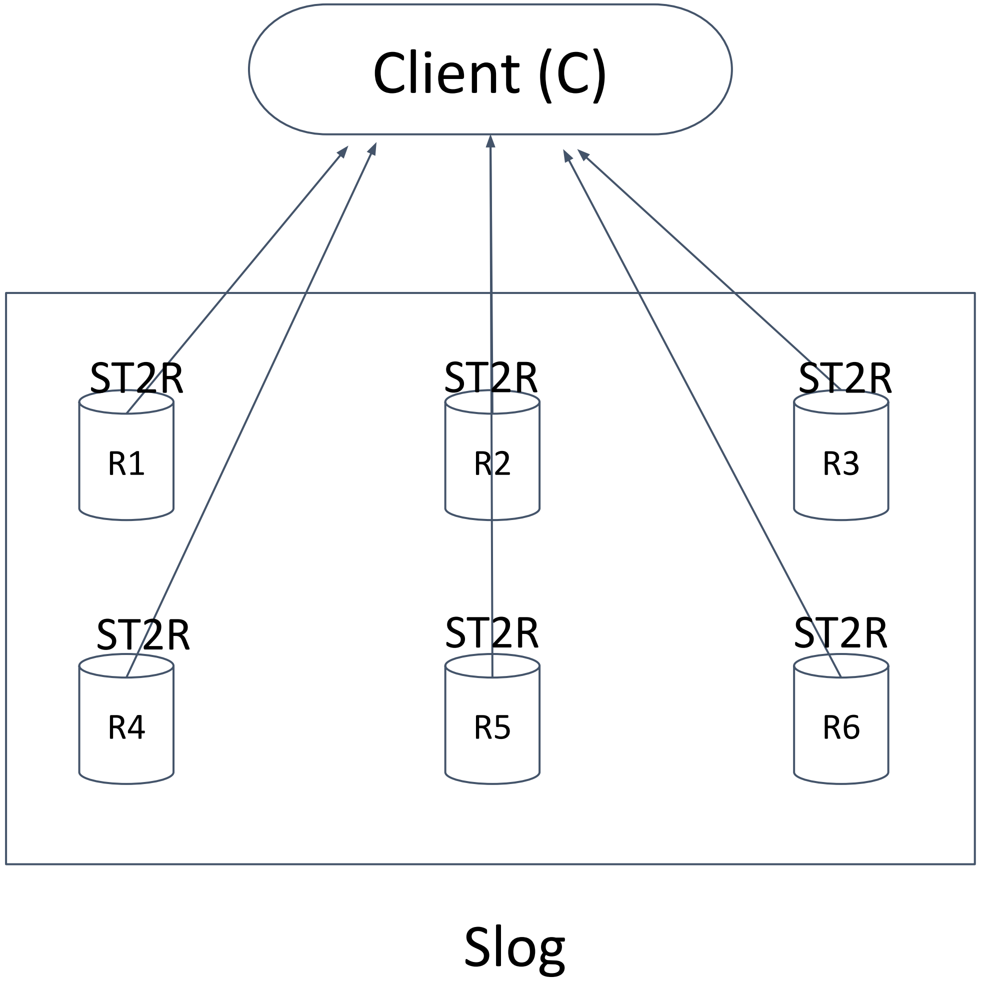
5. C→R: The client attempts to make its tentative 2PC decision **durable**

ST2 := <id(T), decision, {shard votes}, view=0>



Prepare Phase (Stage 2)

6. R→C: Replicas in Slog receives ST2 message, validates decision and returns ST2R message.



ST2R := <id(T), decision, view(decision), view(current)>

Prepare Phase (Stage 2)

7. C←R: The client receives a sufficient number of matching replies to confirm a decision was logged.

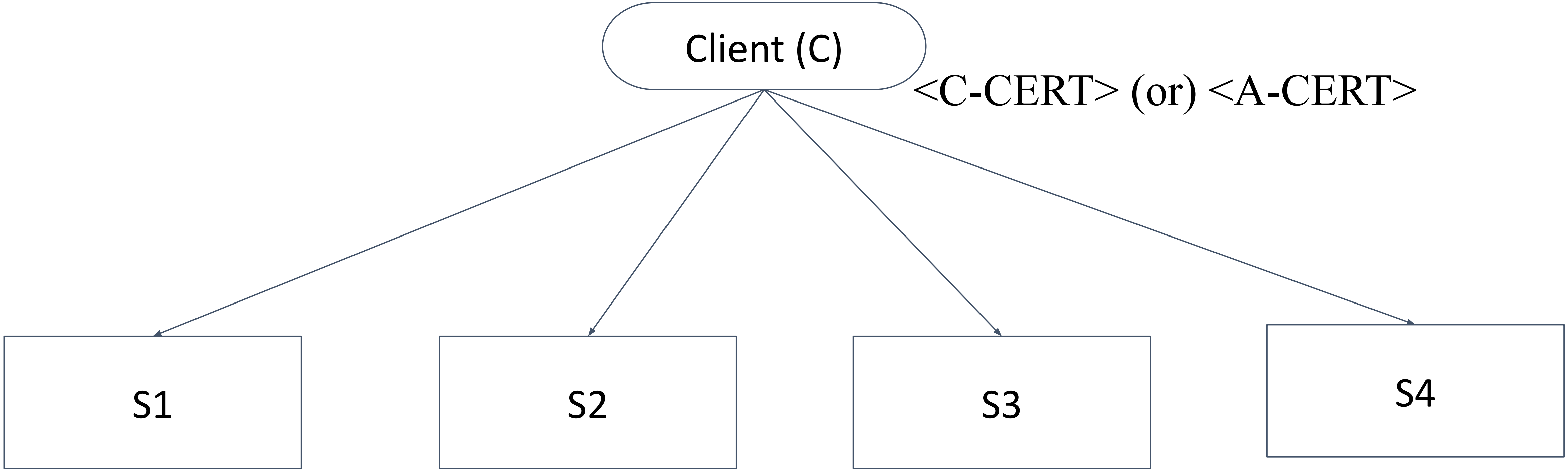
- C waits for (n-f) ST2R messages whose decision and view(decision) match.
- C creates a single shard certificate **V-CERT$_{\textbf{Slog}}$** for the logging shard.

V-CERT$_{\textbf{Slog}}$

<id(T), S, decision, {ST2R}>

Writeback Phase

1. C→R: The client asynchronously forwards decision certificates to all participating shards.



(Shards involved in this transaction)

Writeback Phase

C-CERT
<id(T), Commit, {V-CERT$_S$}>

A-CERT
<id(T), Abort, {V-CERT$_S$}>

Writeback Phase

2. R←C: Replica validates C-CERT (or) A-CERT and updates store accordingly.

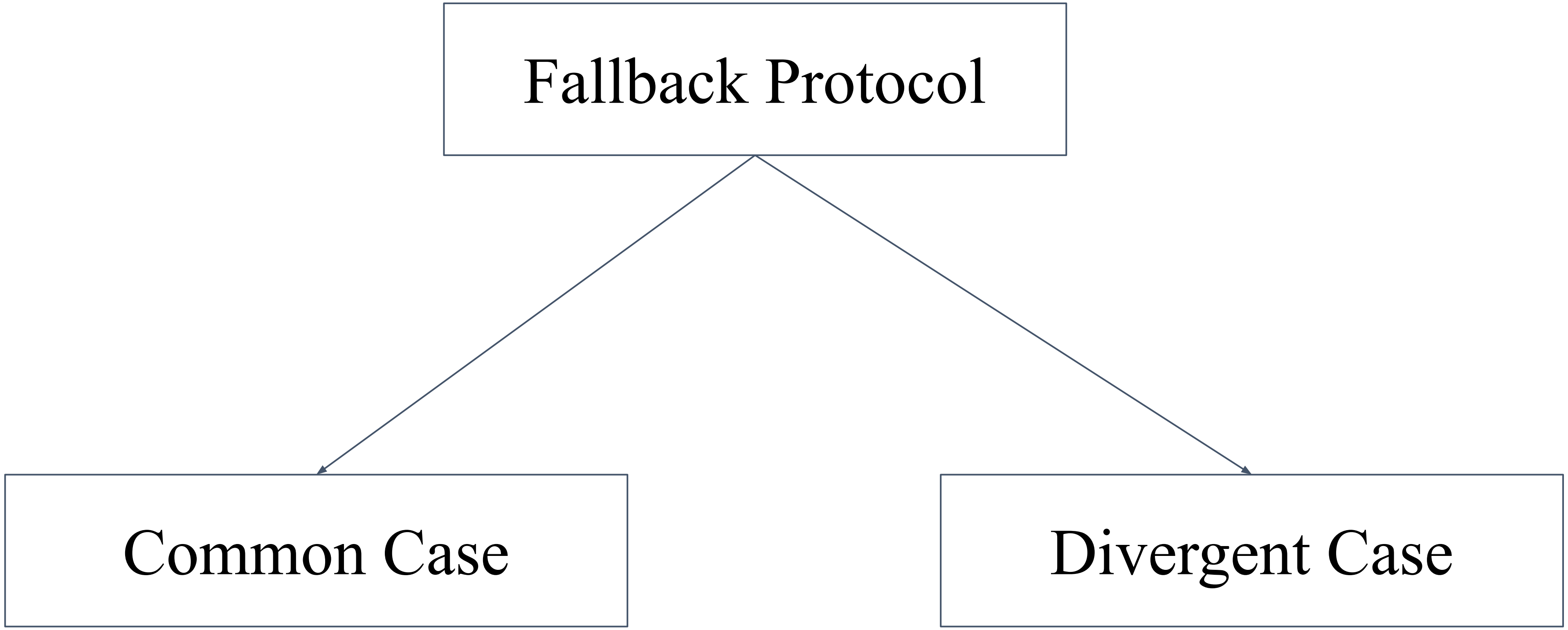
- R updates all local data structures, including applying writes
- R notifies pending dependencies

hase (Stage I)

Transaction Recovery

Let,

- T be a transaction by a client that is being stalled.
- S_{log} be the shard where the vote tally for a decision is stored.
- ST1 = $\langle \text{PREPARE}, T \rangle$
- ST1R = ST1R $\langle T, vote \rangle$
- ST2 = $<id_T, \textit{decision}, \{\text{SHARDVOTES}\}, \textit{view}>$
- ST2R = $<id_T, \textit{decision}, \textit{view}_{decision}, \textit{view}_{current}>$



Fallback Protocol

Common Case

Divergent Case

Common Case

- Clients resend ST1 message
- Receive one of three response:
    - ST1R ⟨*T* , *vote*⟩
    - ST2R <id*T* , *decision*, *viewdecision*, *viewcurrent*>
    - C-Cert or A-Cert
- Client fast forwards to the next phase.

Divergent Case

- Occurs when there are conflicting ST2R messages.

- Reasons:

    - Byzantine client issued T and sent deliberately conflicting ST2 messages to S_{log}

    - multiple correct clients tried to finish T concurrently, that led them to reach different decisions

- normal method cannot be used to recover the transactions in this case

Client sends a Invoke message

- Client sends an **InvokeFB** message of the form **<id$_T$, views>** where,

- **id$_T$** is the identifier for the transaction

- **views** represents the set of current views associated with each replica

Replicas start Election Process after getting an Invoke message

- Every replica R starts the process of determining the most recent view as soon as it receives the InvokeFB message.

- The new view is determined based in certain rules:

1. If a view ‘**w**’ is seen ‘**3f + 1**’ times in the views then,

   $\mathbf{view_{current}} = \mathbf{max(w + 1, view_{current})}$

   where,

   $\mathbf{view_{current}}$ is the current view of the replica

2. Else $\mathbf{view_{current}} = \mathbf{w}$

   where, **w > view_current** and

   w appears at least **f + 1** times in the views set

After all the replicas are on the same view, they send a message **ELECTFB ⟨*id_T* , *decision*, *viewcurrent*⟩** to a replica with id $\mathbf{view_{current} + (id_T \bmod n)}$

Fallback leader aggregates election messages and sends decisions to replicas.

If a replica $\mathbf{R_{FL}}$ receives **4f + 1 ELECTFB** messages, with matching views, it will consider it self the leader of the fallback mechanism

DECFB: <($\mathbf{id_T}$, *decnew*, $\boldsymbol{view_{elect}}$), {ELECTFB}>

where,

- $\mathbf{id_T}$ is the identifier for the transaction,
- ***decnew*** is the new decision,
- $\boldsymbol{view_{elec}}$ is the view on the basis of which it was elected and
- finally it also broadcasts the **ELECTFB** messages it received as proof of leadership.

Replicas send ST2R message to the client

- Replicas receive a **DECFB** message.
- Each replica checks if their own view is smaller or equal to viewelect.
- If so, the replica then updates its current state to viewelect.
- Finally, replicas forward this decision to the clients in an ST2R message: $\langle \mathbf{id}_{T}, \boldsymbol{decision}, \boldsymbol{view}_{decision}, \boldsymbol{view}_{current} \rangle$

Client creates a V-Cert of restarts fallback process

- The client waits to receive **n - f** ST2R messages from different replicas.

- These ST2R messages need to have a matching decision and view.

- If the client successfully receives these consistent ST2R messages, it creates a V-CERT certificate that can be used to commit the decision as part of the protocol's Commit phase.

Conclusion

Basil improves throughput over traditional BFT systems by four to five times

Basil's novel recovery mechanism further minimizes the impact of failures: with 30% Byzantine clients, throughput drops by less than 25% in the worst-case.

References

1. Suri-Payer, Florian, et al. "Basil: Breaking up BFT with ACID (transactions)." Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021.

2. https://www.youtube.com/watch?v=RKZvsW-p4P0

3. https://www.youtube.com/watch?v=mPNWKG7BUoM

4. https://www.youtube.com/watch?v=Rz4Bnpt_hHE