# *Computer Science 425 Distributed Systems (Fall 2009)*

## Lecture 24

## Transactions with Replication

## Reading: Section 15.5

Klara Nahrstedt

# *Acknowledgement*


- **The slides during this semester are based on ideas and material from the following sources:**
  - **Slides prepared by Professors M. Harandi, J. Hou, I. Gupta, N. Vaidya, Y-Ch. Hu, S. Mitra.**
  - **Slides from Professor S. Gosh's course at University o Iowa.**

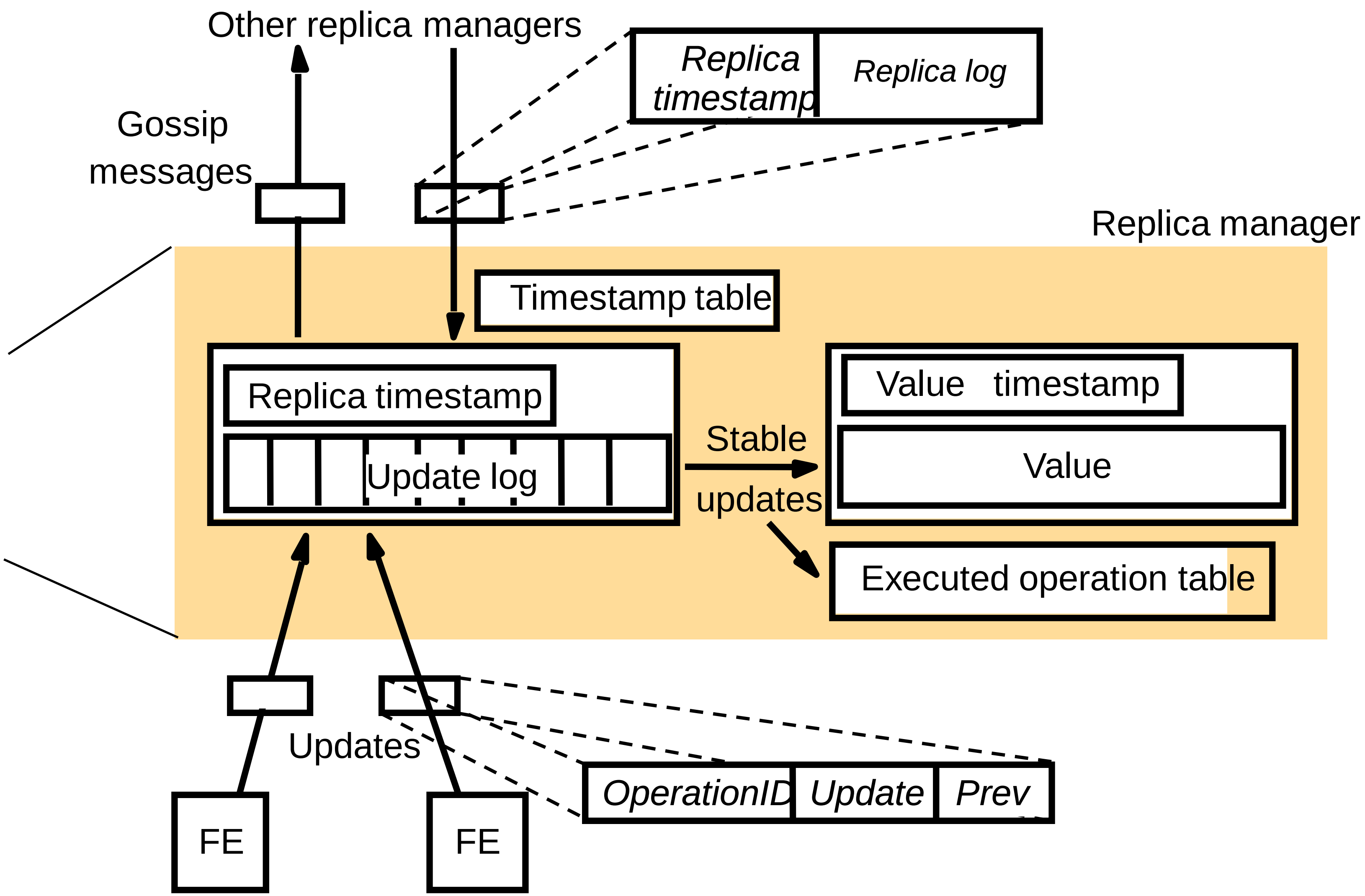
# *Administrative*

- **MP3 posted**
  - **Deadline  December 7 (Monday) – pre-competition**
    - **Top five groups will be selected for final demonstration on Tuuesday, December 8**
  - **Demonstration Signup Sheets for Monday, 12/7, will be made available**
  - **Main Demonstration in front of the Qualcomm Representative will be on Tuesday, December 8 afternoon - details will be announced.**
- **HW4 posted**
  - **Deadline December 1, 2009 (Tuesday)**

# *Plan for Today*

- Gossiping Architecture – Review
- Transactions on replicated objects
- Communication via 2PC Protocol for Replicated objects (no failures)
  - Primary copy replication approach
  - Read one/write all replication approach
  - Available copies replication approach
- Replications under failures
  - RM failure
  - Network partition
    - » Quorum-based approaches

# A Gossip Replica Manager



# *Update Operations*

- **Each update request *u* contains**
  - **The update operation, *u.op***
  - **The FE's timestamp, *u.prev***
  - **A unique id that the FE generates, *u.id*.**
- **Upon receipt of an update request, the RM *i***
  - **Check if *u* has been processed by looking up *u.id* in the executed operation table and in the update log and executed operation table.**
  - **If not, increment the *i*-th element in the replica timestamp by 1 to keep track of the number of updates directly received from FEs.**
  - **Place a record for the update in the RM's log.**

    ***logRecord := <i, ts, u.op, u.prev, u.id>***

    ***where ts is derived from u.prev by replacing u.prev's ith element by the ith element of its replica timestamp.***
  - **Return *ts* back to the FE, which merges it with its timestamp.**

# *Update Operation (Cont'd)*

- **The stability condition for an update *u* is**

  ***u.prev <= valueTS***

  **i.e., All the updates on which this update depends have already been applied to the value.**
- **When the update operation *u* becomes stable, the RM does the following**
  - ***value := apply(value, u.op)***
  - ***valueTS := merge(valueTS, ts) (update the value timestamp)***
  - ***executed := executed U {u.id} (update the executed operation table)***

# *Exchange of Gossiping Messages*

- **A gossip message *m* consists of the log of the RM, *m.log*, and the replica timestamp, *m.ts*.**
  - **Replica timestamp contains info about non-stable updates**
- **An RM that receives a gossip message has three tasks:**
  - **(1) Merge the arriving log with its own.**
    - » **Let *replicaTS* denote the recipient RM's replica timestamp. A record *r* in *m.log* is added to the recipient's log unless *r.ts* <= *replicaTS*.**
    - » ***replicaTS* ← *merge(replicaTS, m.ts)***
  - **(2) Apply any updates that have become stable but not been executed (stable updates in the arrived log may cause some pending updates become stable)**
  - **(3) Garbage collect: Eliminate records from the log and the executed operation table when it is known that the updates have been applied everywhere.**

# *Query Operations*

- A query request *q* contains the operation, *q.op*, and the timestamp, *q.prev*, sent by the FE.
- Let *valueTS* denote the RM's value timestamp, then *q* can be applied if

  ***q.prev* <= *valueTS***
- The RM keeps *q* on a hold back queue until the condition is fulfilled.
  - *If valueTs is (2,5,5) and q.prev is (2,4,6), then one update from $RM_3$ is missing.*
- Once the query is applied, the RM returns

  ***new* ← *valueTS***

  to the FE (along with the value), and the FE merges *new* with its timestamp.

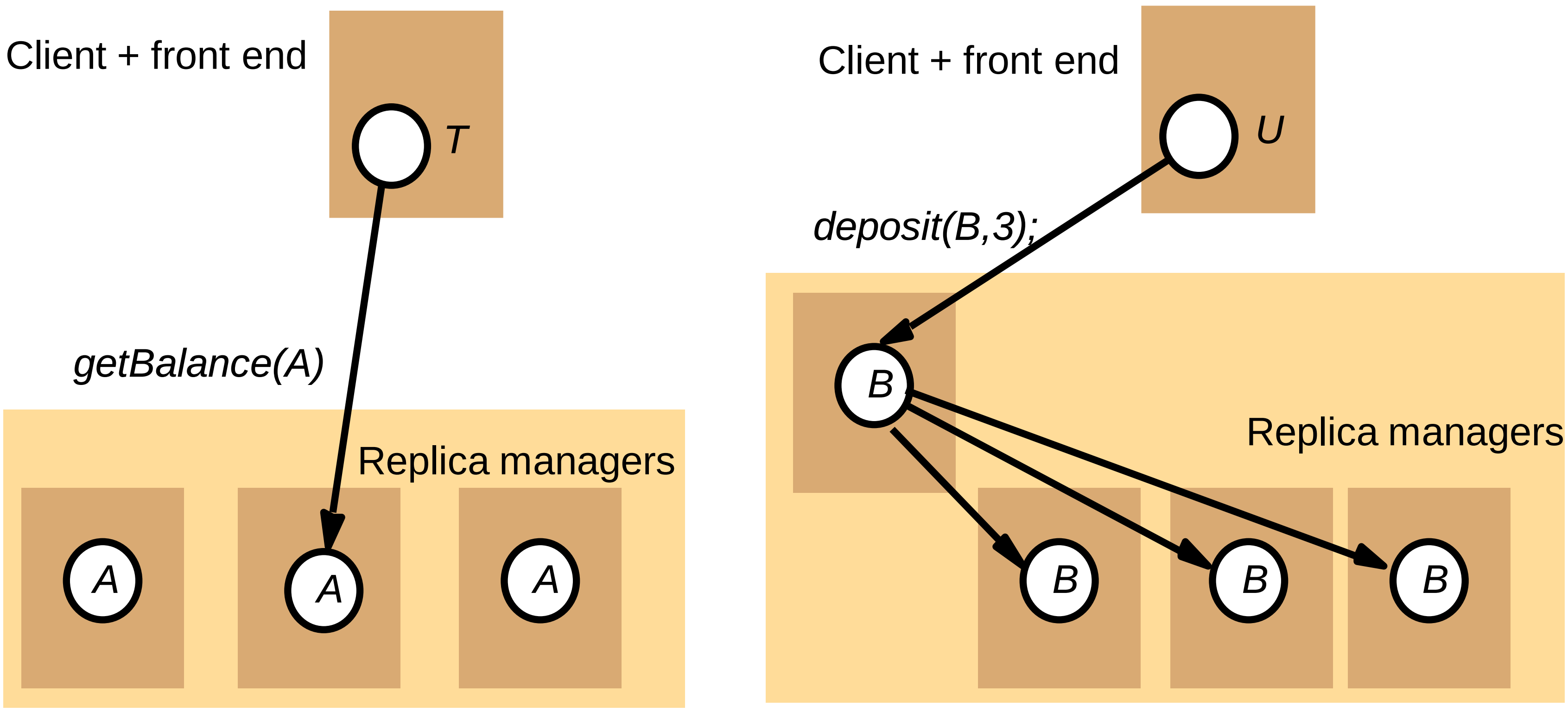
# *Selecting Gossip Partners*

- **The frequency with which RMs send gossip messages depends on the application.**
- **Policy for choosing a partner to exchange gossip with:**
  - **Random policies: choose a partner randomly (perhaps with weighted probabilities)**
  - **Deterministic policies: a RM can examine its timestamp table and choose the RM that is the furthest behind in the updates it has received.**
  - **Topological policies: arrange the RMs into an overlay graph. Choose graph edges based on small round-trip times (RTTs), e.g., ring or Chord.**
    - » **Each has its own merits and drawbacks. The ring topology produces relatively little communication but is subject to high transmission latencies since gossip has to traverse several RMs.**
- ***Example: Network News Transport Protocol (NNTP) uses gossip communication. Your updates to class.cs425 are spread among News servers using the gossip protocol!***
- **Gives probabilistically reliable and fast dissemination of data with very low background bandwidth**
  - **Analogous to the spread of gossip in society.**

# *Examples of Highly Available Services*

- **Bayou**
  - **Replicated database with weaker guarantees than sequential consistency**
    - » **Uses gossip, timestamps and concept of *anti-entropy***
  - **Anti-Entropy - Anti-entropy protocols for repairing replicated data, which operate by comparing replicas and reconciling differences.**
  - **Complete anti-entropy (CAE)**
    - » **Reconcile all inconsistent states**
  - **Selective anti-entropy (SAE)**
    - » **Selectively reconcile inconsistent states**
- **Coda**
  - **Provides high availability in spite of disconnected operation, e.g., roving and transiently-disconnected laptops**
  - **Based on AFS**
  - **Aims to provide *Constant data availability***

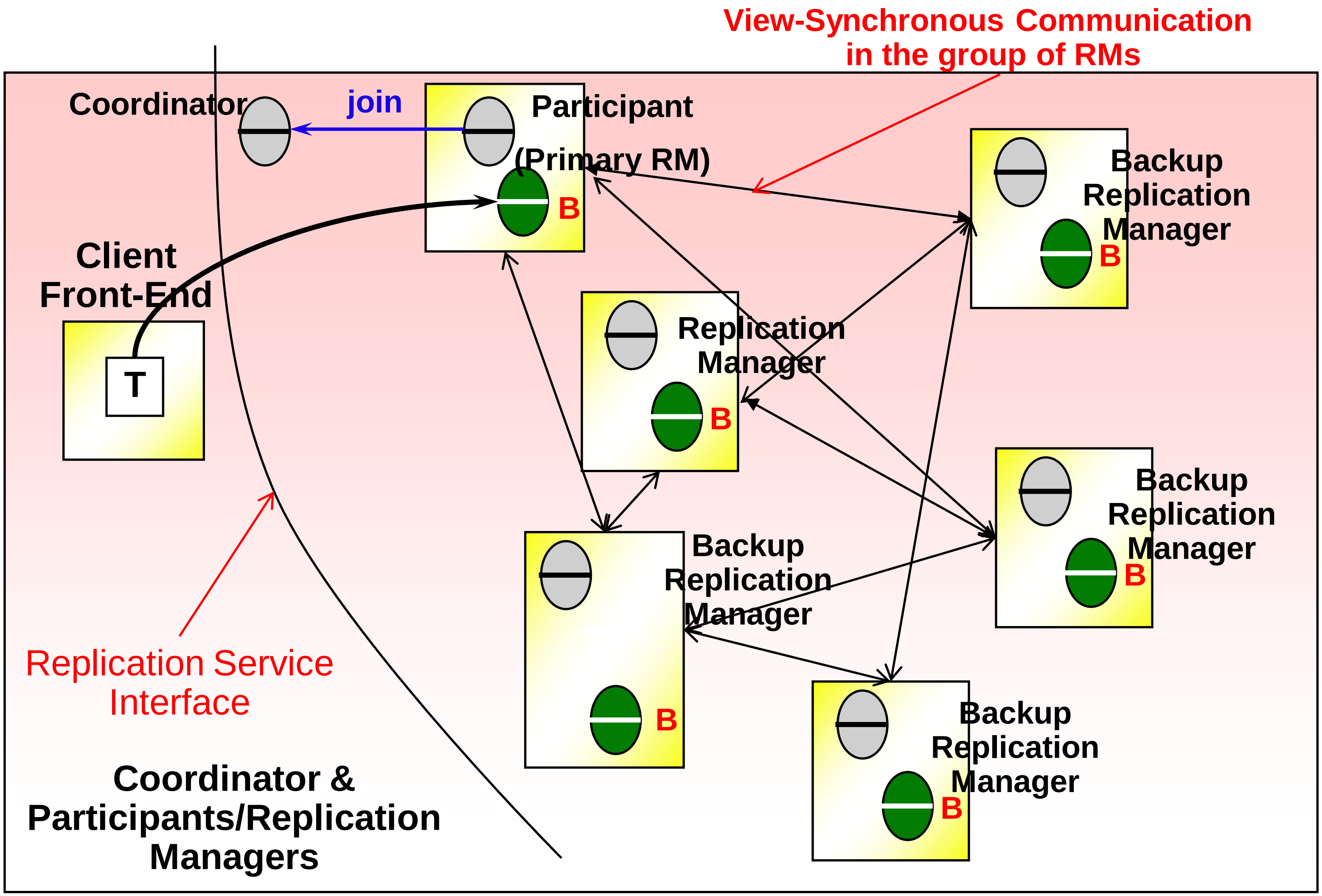
# *Transactions on Replicated Data*



# *One Copy Serialization*

- In a non-replicated system, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of transaction operations.
- One-copy serializability: The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).
  – Equivalent to combining serial equivalence + replication transparency/consistency

# Coordination in Primary-Backup Replication Service

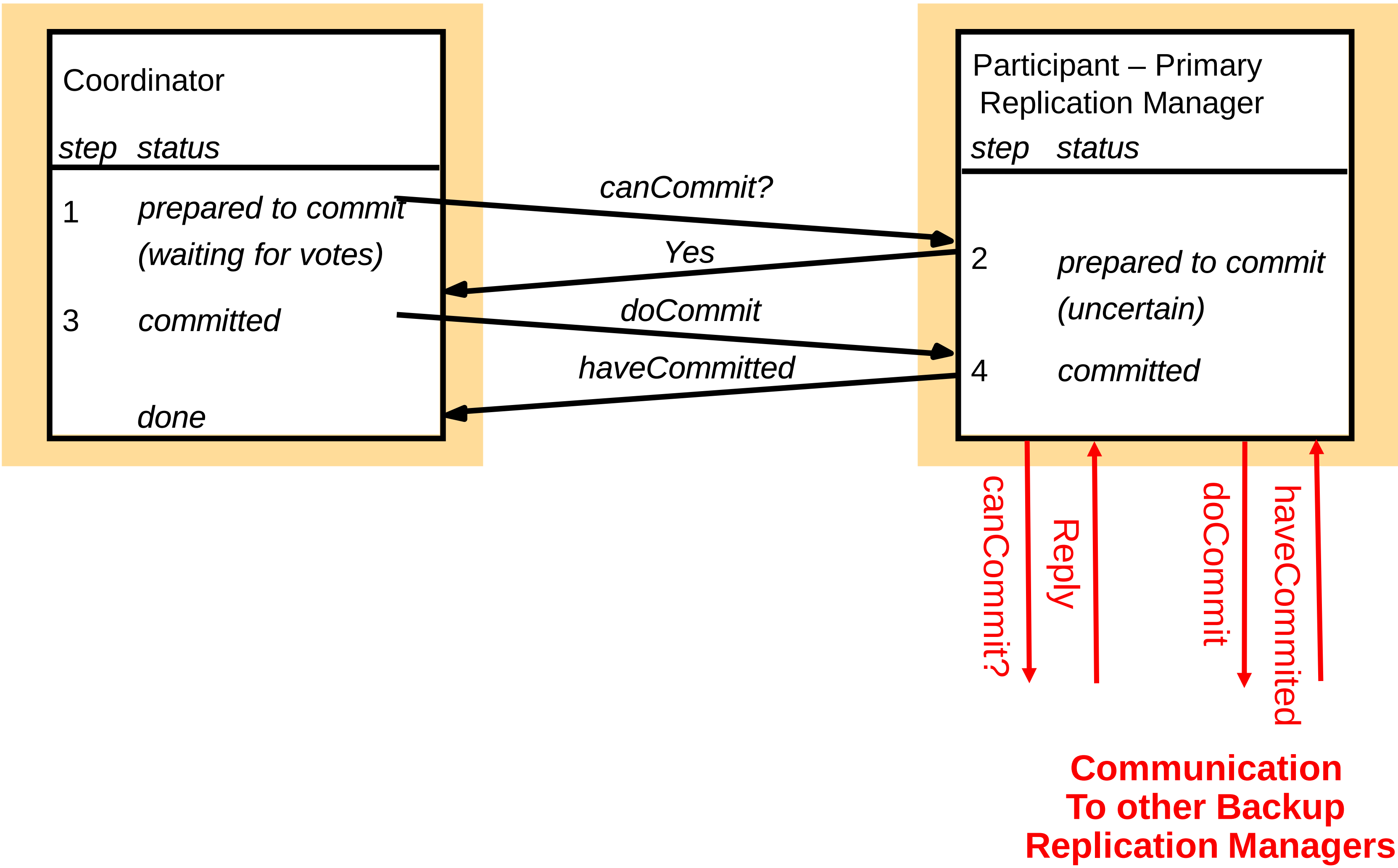


# *Two Phase Commit Protocol For Replicated Objects*

**Two level nested 2PC**

- **In the first phase, the coordinator sends the canCommit? command to the participants, each of which then passes it onto the other RMs involved (e.g., by using view synchronous communication) and collects their replies before replying to the coordinator.**
- **In the second phase, the coordinator sends the doCommit or doAbort request, which is passed onto the members of the groups of RMs.**

# Communication in Two-Phase Commit Protocol



# *Primary Copy Replication (Approach 1)*

- **For now, assume no crashes/failures**
- **All the client requests are directed to a single primary RM.**
- **Concurrency control is applied at the primary.**
- **To commit a transaction, the primary communicates with the backup RMs and replies to the client.**
- **View synchronous comm. gives → one-copy serializability (Why?)**
- **Disadvantage? Performance is low since primary RM is bottleneck.**

# *Read One/Write All Replication (Approach 2)*

- **An FE may communicate with any RM.**
- **Every write operation must be performed at <u>all</u> of the RMs**
  - **Each contacted RM sets a write lock on the object.**
- **A read operation can be performed at <u>any</u> single RM**
  - **A contacted RM sets a read lock on the object.**
- **Consider pairs of conflicting operations of different transactions on the same object.**
  - **Any pair of write operations will require locks at all of the RMs**
  - **A read operation and a write operation will require conflicting locks at some RM**

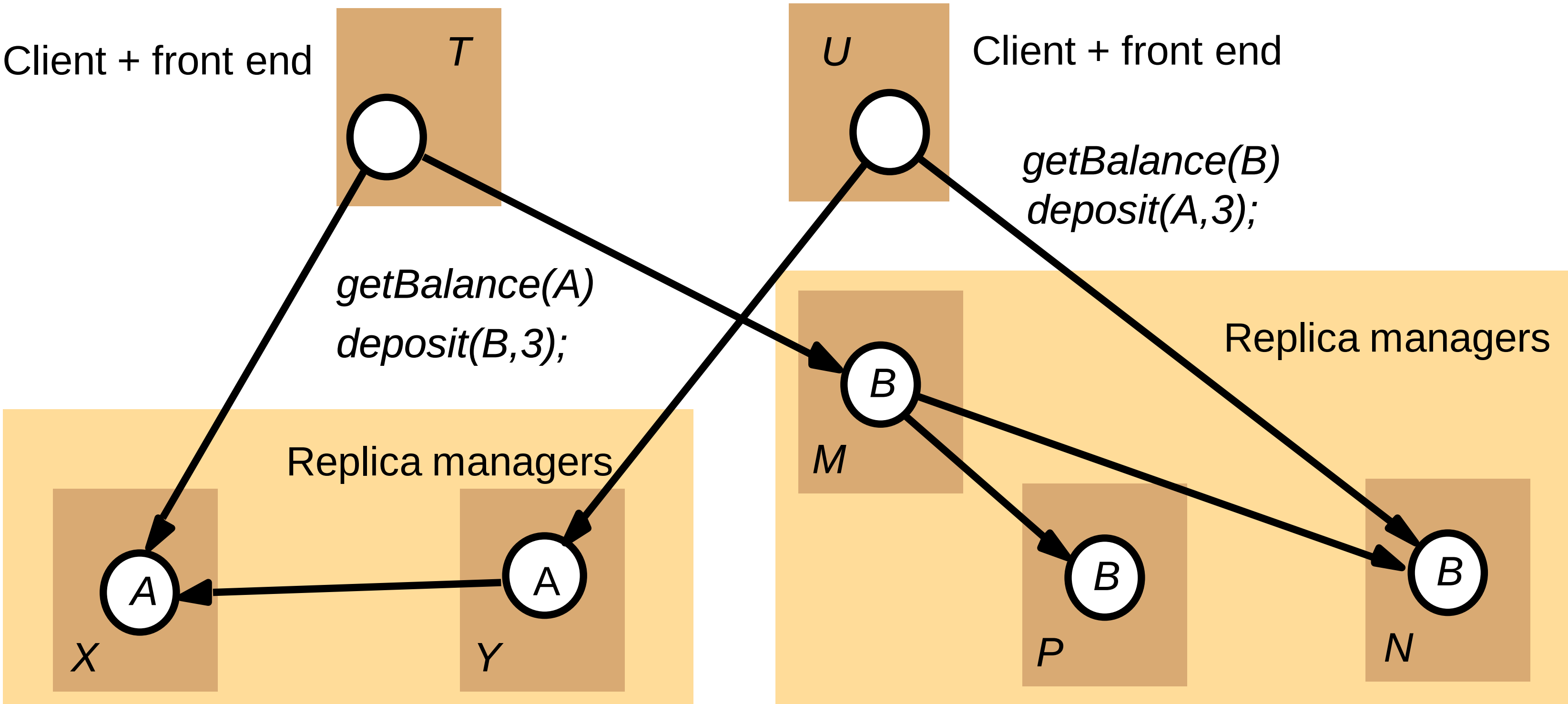
  **- One-copy serializability is achieved.**

**Disadvantage? Failures block the system (esp. writes).**

# *Available Copies Replication (Approach 3)*

- **A client's read request on an object can be performed by any RM, but a client's update request must be performed *across all available* *(i.e., non-faulty)* RMs in the group.**
- **As long as the set of available RMs does not change, local concurrency control achieves one-copy serializability in the same way as in read-one/write-all replication.**
- **May not be true if RMs fail and recover during conflicting transactions.**

# Available Copies Approach



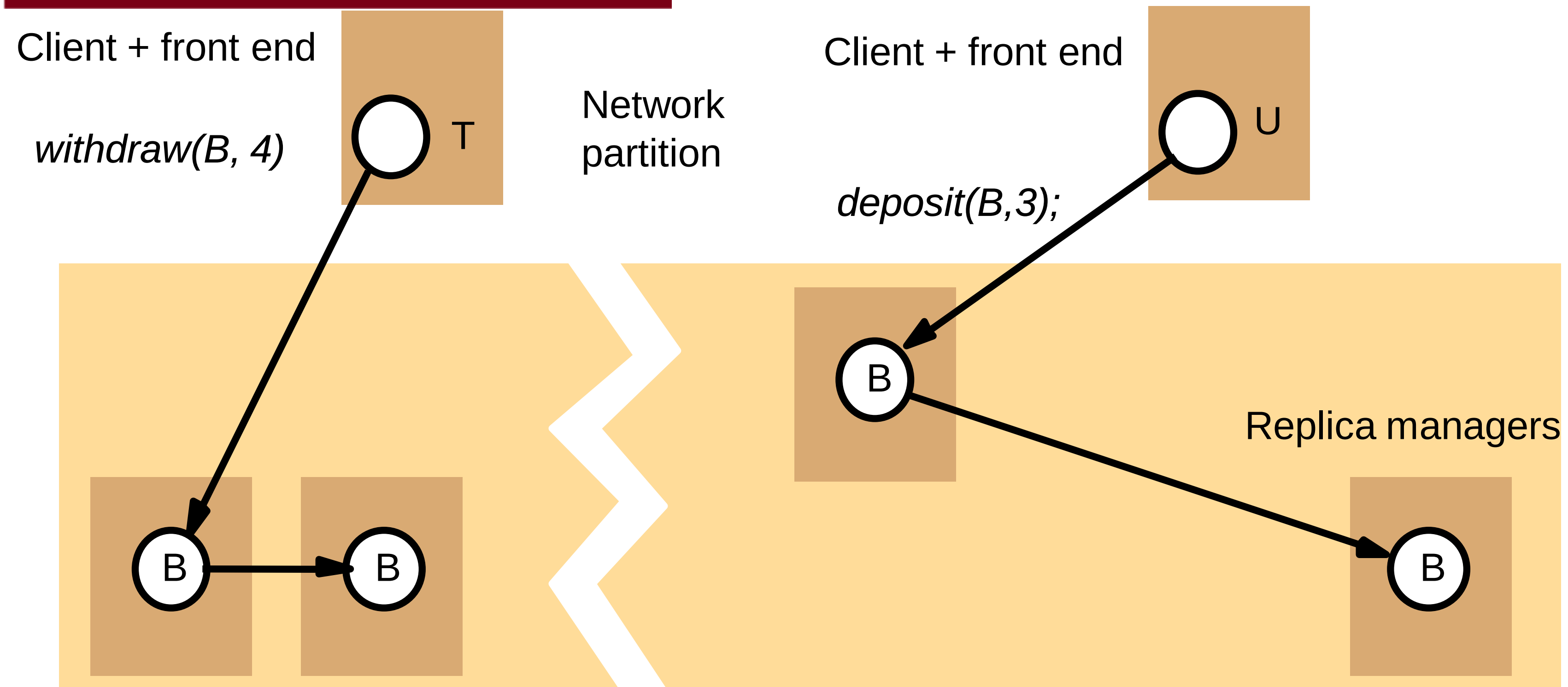
# *The Impact of RM Failure*

- **Assume that (i) RM X fails just after T has performed *getBalance(A)*; and (ii) RM N fails just after U has performed *getBalance(B)*. Both failures occur before any of the *deposit()*s.**
- **Subsequently, T's deposit will be performed at RMs M and P, and U's deposit will be performed at RM Y.**
- **The concurrency control on A at RM X does not prevent transaction U from updating A at RM Y.**
- **Solution: Must also serialize *RM crashes and recoveries* with respect to transactions.**

# *Local Validation (using Our Example)*

- **From T's perspective,**
  - **T has read from an object at X ➔ X must have failed after T's operation.**
  - **T observes the failure of N when it attempts to update the object B ➔ N's failure must be before T.**
  - **N fails → T reads object A at X; T writes objects B at M and P → T commits → X fails.**
- **From U's perspective,**
  - **X fails → U reads object B at N; U writes object A at Y → U commits → N fails.**
- **At the time T tries to commit,**
  - **it first checks if N is still not available and if X, M and P are still available. Only then can T commit.**
  - **It then sees if the failure order is consistent with that of other transactions (T cannot commit if U has committed)**
  - **If T commits, U's validation will fail because N has already failed.**
- **Can be combined with 2PC.**
- **Local validation may not work if partitions occur in the network**

# Network Partition



# *Dealing with Network Partitions*

- **During a partition, pairs of conflicting transactions may have been allowed to execute in different partitions. The only choice is to take corrective action after the network has recovered**
  - **Assumption: Partitions heal eventually**
- **Abort one of the transactions after the partition has healed**
- **Basic idea: allow operations to continue in partitions, but finalize and commit trans. only after partitions have healed**
- **But need to avoid executing operations that will eventually lead to aborts…**

# *Quorum Approaches for Network Partitions*

- **Quorum approaches used to decide whether reads and writes are allowed**
- **In the *pessimistic quorum philosophy*, updates are allowed only in a partition that has the <u>majority</u> of RMs**
  - **Updates are then propagated to the other RMs when the partition is repaired.**

# *Static Quorums*

- The decision about how many RMs should be involved in an operation on replicated data is called ***Quorum selection***
- Quorum rules state that:
  - At least ***r*** replicas must be accessed for read
  - At least ***w*** replicas must be accessed for write
  - ***r*** + ***w*** > N, where N is the number of replicas
  - ***w*** > N/2
  - Each object has a version number or a consistent timestamp
- Static Quorum predefines ***r*** and ***w*** , & is a

  **pessimistic approach:** if partition occurs, update will be possible in at most one partition

# Voting with Static Quorums

- A version of quorum selection where each replica has a number of votes. Quorum is reached by majority of votes (N is the total number of votes)

e.g., a cache replica may be given a 0 vote

| Replica | votes | access time | version chk | P(failure) |
|---|---|---|---|---|
| Cache | 0 | 100ms | 0ms | 0% |
| Rep1 | 1 | 750ms | 75ms | 1% |
| Rep2 | 1 | 750ms | 75ms | 1% |
| Rep3 | 1 | 750ms | 75ms | 1% |

- with <u>r</u> = <u>w</u> = 2, **Access time for write is 750 ms (parallel writes). Access time for read without cache is 750 ms. Access time for read with cache is 175ms to 825ms. (chk – check time)**

# Quorum Consensus Examples

[Gifford]'s examples
for a replicated file system

*Ex1:*
High R to W ratio
Single RM on Replica 1

*Ex2:*
Moderate R to W ratio
Accessed from local LAN of RM 1

*Ex3:*
V. High R to W ratio
All RM's equidistant

| | | *Example 1* | *Example 2* | *Example 3* |
|---|---|---|---|---|
| *Latency* | Replica 1 | 75 | 75 | 75 |
| *(milliseconds)* | Replica 2 | 65 | 100 | 750 |
| | Replica 3 | 65 | 750 | 750 |
| *Voting* | Replica 1 | 1 | 2 | 1 |
| *configuration* | Replica 2 | 0 | 1 | 1 |
| | Replica 3 | 0 | 1 | 1 |
| *Quorum* | *R* | 1 | 2 | 1 |
| *sizes* | *W* | 1 | 3 | 3 |

Derived performance of file suite:

| | | | | |
|---|---|---|---|---|
| *Read* | Latency | 75 | 75 | 75 |
| | Blocking probability | 0.01 | 0.0002 | 0.000001 |
| *Write* | Latency | 75 | 100 | 750 |
| | Blocking probability | 0.01 | 0.0101 | 0.03 |

0.01 failure prob. per RM

# *Optimistic Quorum Approaches*

- An Optimistic *Quorum selection* allows writes to proceed in <u>any</u> partition.
- This might lead to write-write conflicts. Such conflicts will be detected when the partition heals
  - Any writes that violate one-copy serializability will then result in the transaction (that contained the write) to abort
  - Still improves performance because partition repair not needed until commit time
- Optimistic Quorum is practical when:
  - Conflicting updates are rare
  - Conflicts are always detectable
  - Damage from conflicts can be easily confined
  - Repair of damaged data is possible or an update can be discarded without consequences

# *View-based Quorum*

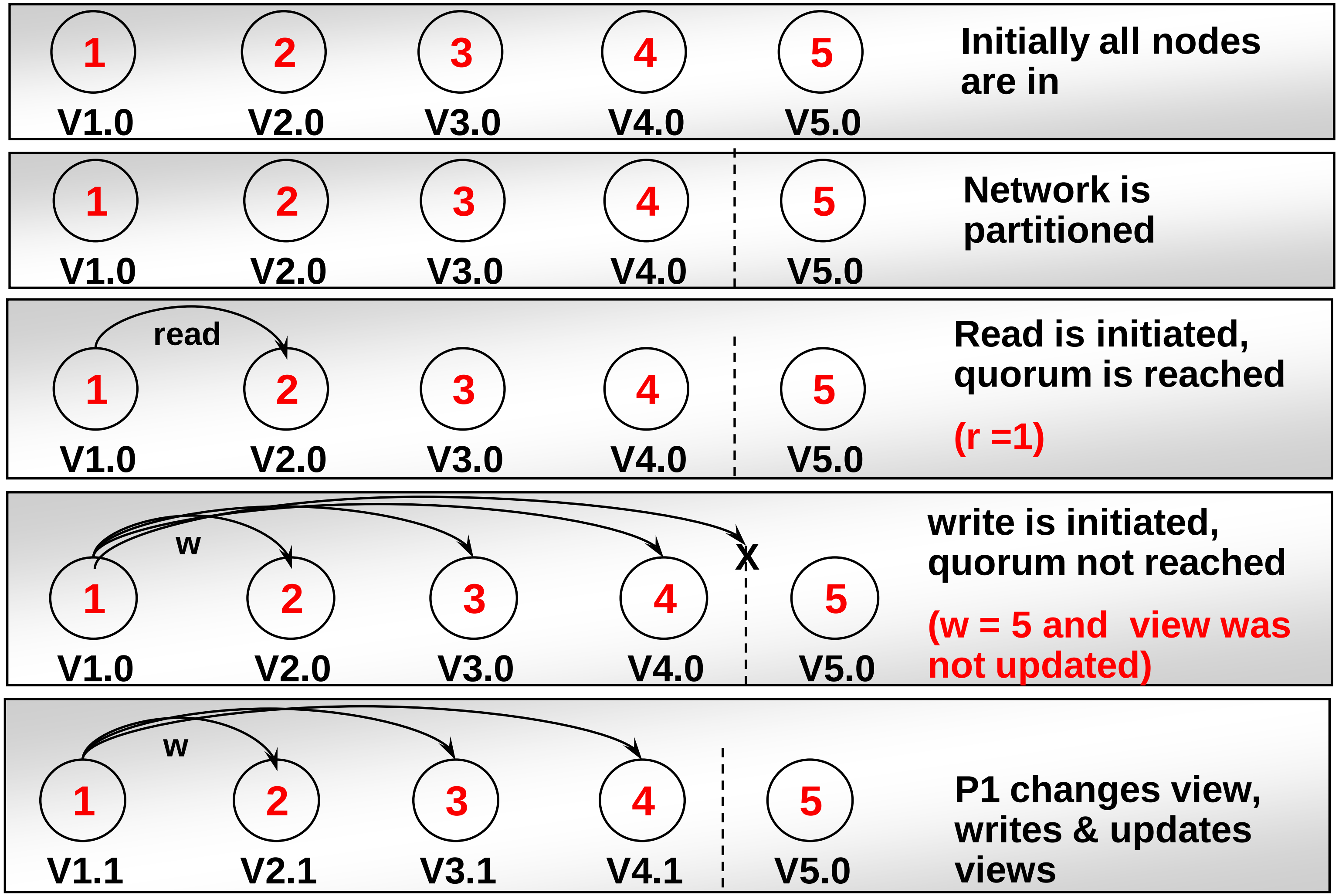
- An ***optimistic*** approach
- Quorum is based on views at any time
- In a partition, inaccessible nodes are considered in the quorum as ghost participants that reply "Yes" to all requests.
  - Allows operations to proceed if the partition is large enough (need not be majority)
- Once the partition is repaired, participants in the smaller partition know whom to contact for updates.

# View-based Quorum - details

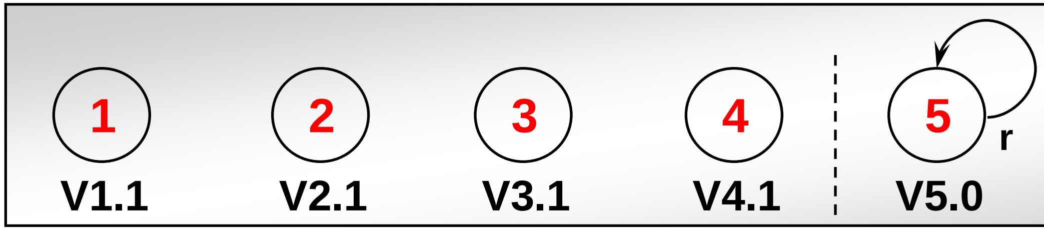
- We define **thresholds** for each of read and write :
  - $A_w$: minimum nodes in a view for write, e.g., $A_w > N/2$
  - $A_r$: minimum nodes in a view for read
  - E.g., $A_w + A_r > N$
- If ordinary quorum cannot be reached for an operation, then we take a straw poll, i.e., we update views
- In a large enough partition for read, $View_{size} \geq A_r$ In a large enough partition for write, $View_{size} \geq A_w$ (inaccessible nodes are considered as ghosts that reply Yes to all requests.)
- Views are per object, numbered sequentially and only updated if necessary
- The first update after partition repair forces restoration for nodes in the smaller partition

# Example: View-based Quorum

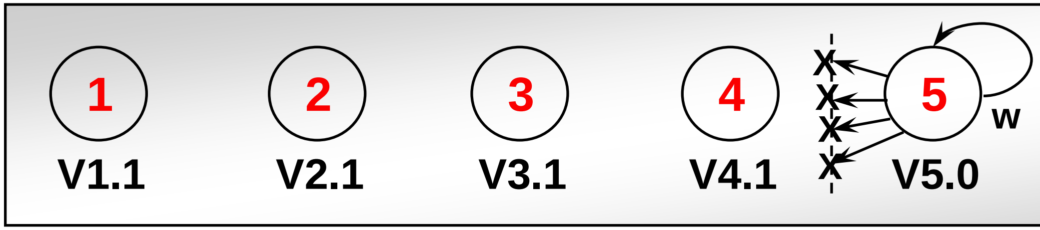
## ❖ Consider: N = 5, w = 5, r = 1, $A_w$ = 3, $A_r$ = 3



# *Example: View-based Quorum (cont'd)*

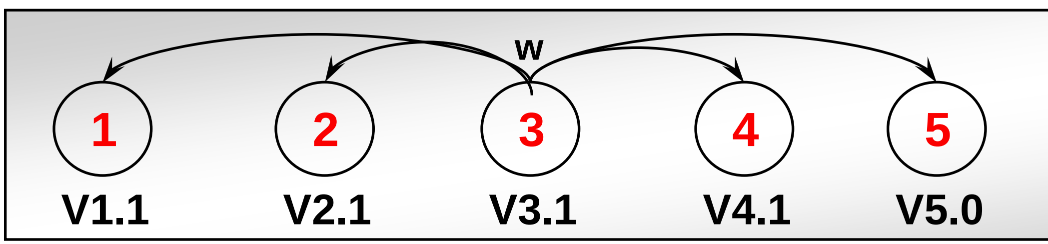



P5 initiates read, has quorum, reads stale data




P5 initiates write, no quorum, $A_w$ not met, aborts.


Partition is repaired




P3 initiates write, notices repair


Views are updated to include P5; P5 is informed of updates

# Summary

- Replication is a very important distributed service to enhance availability and performance
- Replication and concurrency control are needed
- Replication with Transactions