Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

October 25, 2012

Lecture 18

Replication Control

Reading: Sections 18.1-18.3, 18.5

Replication

- So far we've discussed
  - Operations between 1 client and 1 server
  - Operations between multiple clients and 1 server
    - Concurrency Control
  - Operations between multiple clients and multiple servers, with each object having one copy
    - 2 PC and Paxos
- Next: What if each object is replicated at multiple servers?
- Replication = Multiple copies of the same object/data
- Copies are called replicas

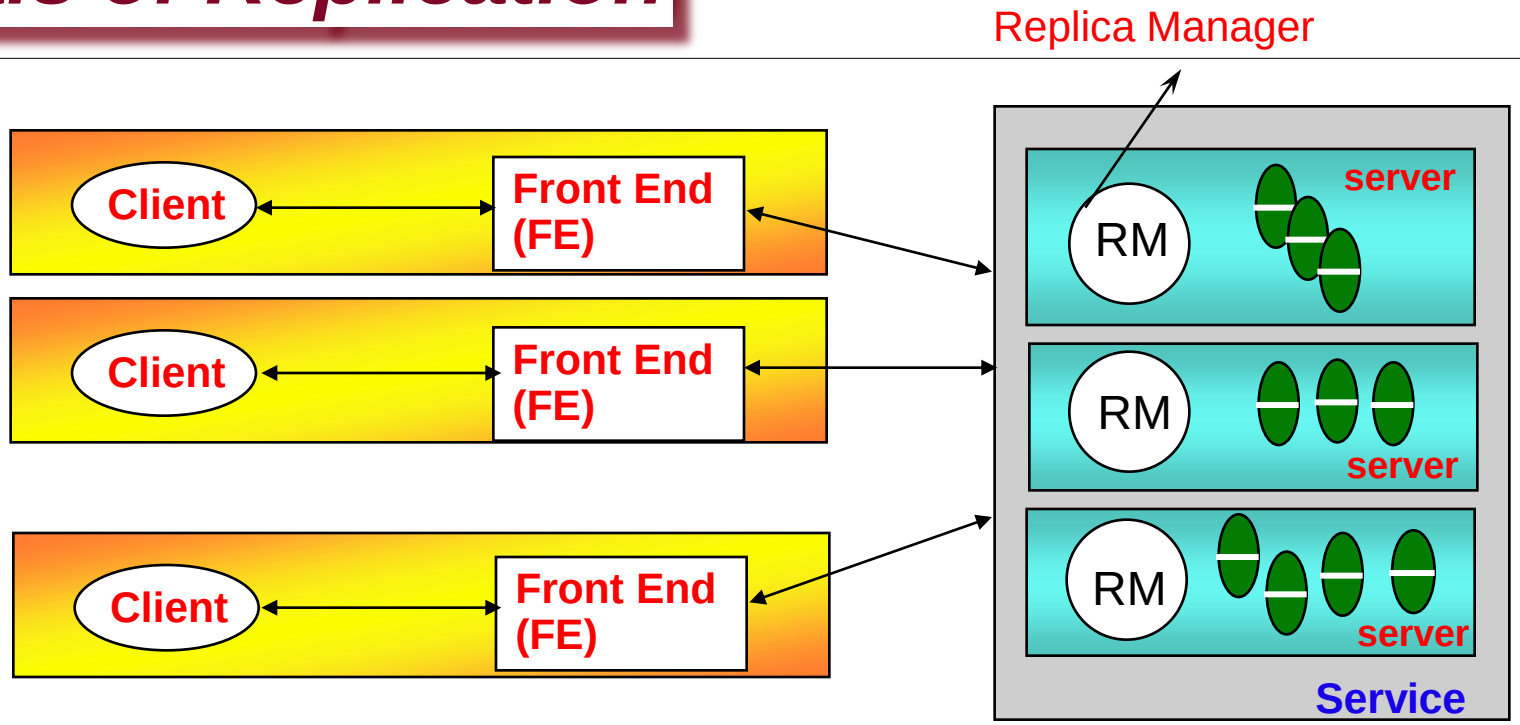
Why Replication

- **Enhances a service (object/data/service)**
  - **Increased Availability**
    - **Of service. When servers fail or when the network is partitioned, service still available at at least once server.**
  - **Fault Tolerance**
    - **Under the fail-stop model, if up to *f* of *f+1* servers crash, at least one is alive.**
  - **Load Balancing**
    - **One approach: Multiple server IPs can be assigned to the same name in DNS, which returns answers/IPs round-robin.**

P: probability that one server fails= 1 – P= availability of service. e.g. P = 5% => service is available 95% of the time.

P^n: probability that n servers fail= 1 – P^n= availability of replicated service. e.g. P = 5%, n = 3 => service available 99.875% of the time

Goals of Replication



- **Replication Transparency**

  User/client need not know that multiple physical copies of data exist.

- **Replication Consistency**

  Data is consistent on all of the replicas of an object (or is converging towards becoming consistent).

Replication Management – First Cut

- **Request Communication**
  - Requests made from client are handled by FE. FE sends requests to either a single RM or to multiple RMs
- **Coordination: The RMs decide**
  - whether the request is to be applied
  - the order of requests
    - **FIFO ordering:** If a FE issues *r* then *r′*, then any correct RM handles *r* and then *r′*.
    - **Causal ordering**: If the issue of *r* "happened before" the issue of *r′*, then any correct RM handles *r* and then *r′*.
    - **Total ordering**: If a correct RM handles *r* and then *r′*, then any correct RM handles *r* and then *r′*.
- **Execution:** The RMs execute the request (often they do this tentatively – why?).

Replication Management – First Cut

- **Agreement**: The RMs attempt to reach consensus on the effect of the request.
  - E.g., Two phase commit or Paxos (this is per-object!)
  - If this succeeds, effect of request is made permanent
- **Response**
  - One or more RMs responds to the FE.
  - The first response to arrive is good enough because all the RMs will return the same answer.
  - Thus each RM is a replicated state machine

    "Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs." [Wikipedia, Schneider 90]

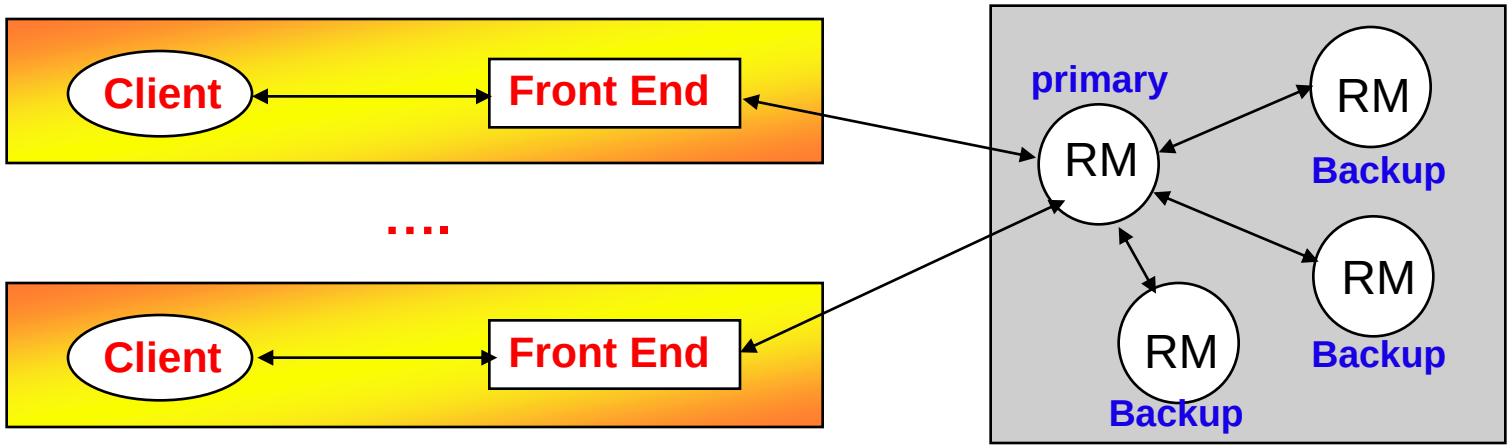
What the Client Sees - Linearizability

- Let the sequence of read and update operations that client *i* performs in some execution be o_{i1}, o_{i2},….
  - "Program order" for the client
- A replicated shared object service is linearizable if for any execution (real), there is some interleaving of operations (virtual) issued by all clients that:
  - meets the specification of a single correct copy of objects
  - is consistent with the *real times* at which each operation occurred during the execution
- Main goal: any client will see (at any point of time) a copy of the object that is correct and consistent

Sequential Consistency

- The real-time requirement of linearizability is hard, if not impossible, to achieve in real systems (Why?)
- A less strict criterion is sequential consistency: A replicated shared object service is sequentially consistent if for any execution (real), there is some interleaving of clients' operations (virtual) that:
  - meets the specification of a single correct copy of objects
  - is consistent with the program order in which each individual client executes those operations.
- This approach does not require absolute time or total order. Only a partial order so that each client's ops in the sequence be consistent with that client's program order (~ FIFO).
- Linearilizability implies sequential consistency. Not vice-versa!
- Challenge with guaranteeing sequential consistency?
  - Ensuring that all replicas of an object are consistent.

Passive (Primary-Backup) Replication

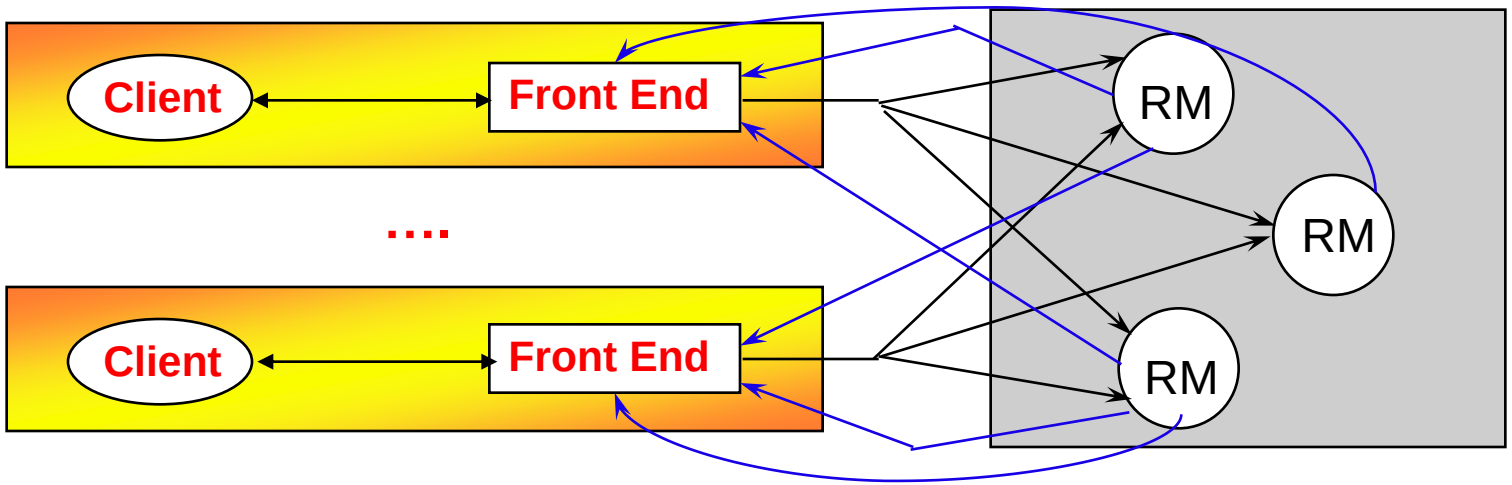


- **Request Communication:** the request is issued to the primary RM and carries a unique request id.
- **Coordination:** Primary takes requests atomically, in order, checks id (resends response if not new id.)
- **Execution:** Primary executes & stores the response
- **Agreement:** If update, primary sends updated state/result, req-id and response to all backup RMs (1-phase commit enough).
- **Response:** primary sends result to the front end

Fault Tolerance in Passive Replication

- The system implements linearizability, since the primary sequences operations in order.
- If the primary fails, a backup becomes primary by leader election, and the replica managers that survive agree on which operations had been performed at the point when the new primary takes over.
  - The above requirement can be met if the replica managers (primary and backups) are organized as a group and if the primary uses view-synchronous group communication to send updates to backups.
- Thus the system remains linearizable in spite of crashes
- However, overhead of election

Active Replication



- **Request Communication:** The request contains a unique identifier and is multicast to all by a reliable totally-ordered multicast.
- **Coordination:** Group communication ensures that requests are delivered to each RM in the same order (but may be at different physical times!).
- **Execution:** Each replica executes the request. (Correct replicas return same result since they are running the same program, i.e., they are *replicated state machines*)
- **Agreement:** No agreement phase is needed, because of multicast delivery semantics of requests
- **Response:** Each replica sends response directly to FE

Fault Tolerance in Active Replication

- RMs work as replicated state machines, playing equivalent roles. That is, each responds to a given series of requests in the same way.
- If any RM crashes, state is maintained by other correct RMs.
- This system implements sequential consistency
  - Use FIFO-total ordering in multicasts from FE to RM group
- Caveat (Out of band): If clients are multi-threaded and communicate with one another while waiting for responses from the service, we may need to incorporate causal-total ordering.

Transactions - One Copy Serializability

- **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**

Replication + Concurrency Control: Primary Copy Replication

- **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**
- **Disadvantage? Performance is low since primary RM is bottleneck.**

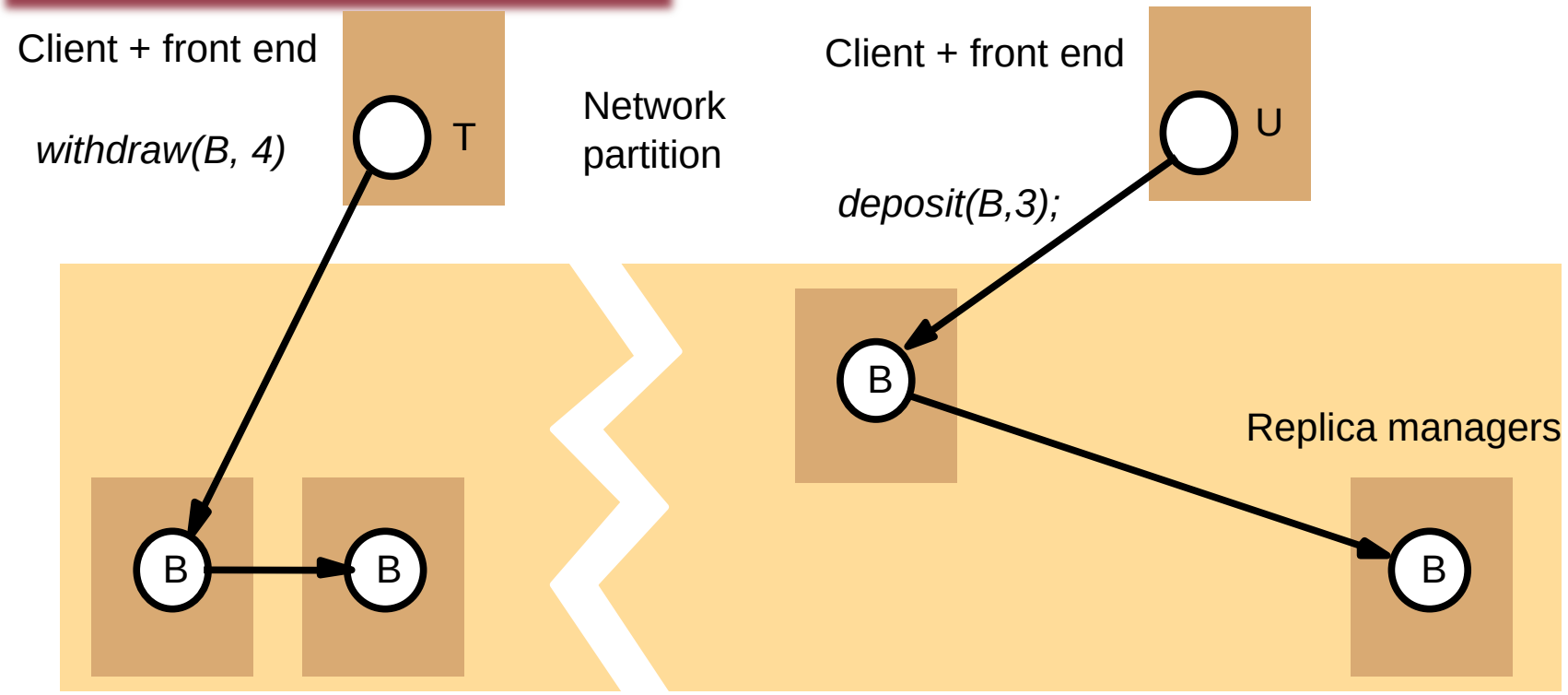
Read One/Write All Replication

- **An FE (front end) 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 ➔ not allowed**
  - **A read operation and a write operation will require conflicting locks at some RM ➔ not allowed**

  ➔One-copy serializability is achieved.

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

Network Partition



Bad News Bears – CAP Theorem

- In a distributed system we desire
  - Consistency: all copies of data should be alike at all times (e.g., one-copy serializability/sequential consistency)
  - Availability: at least one copy of data should be readable and writable at all times
  - Partition-tolerance: in spite of partitions in the system/network, the system continues to operate
- Unfortunately, you cannot achieve all three
- Can only choose at most 2 out of 3
  - Good way to read CAP theorem: under Partition, would you choose Consistency or Availability
- Conjectured by Eric Brewer in 2000, proved by Gilbert and Lynch in 2002
- Different systems take different ways around it
  - CP: Paxos; AP: Many key-value stores; CA: View Synchrony

Healing from 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 to optimize performance, better to avoid executing operations that will eventually lead to aborts…how?**

Quorum Approaches

Quorum Approaches

- **Quorum approaches used to decide whether reads and writes are allowed**
- **There are two types: pessimstic quorums and optimistic quorums**
- **In the *pessimistic quorum philosophy*, updates are allowed only in a partition that has the majority of RMs**
  - **Updates are then propagated to the other RMs when the partition is repaired.**
  - **Any two majority sets intersect, thus consistency is ensured.**

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

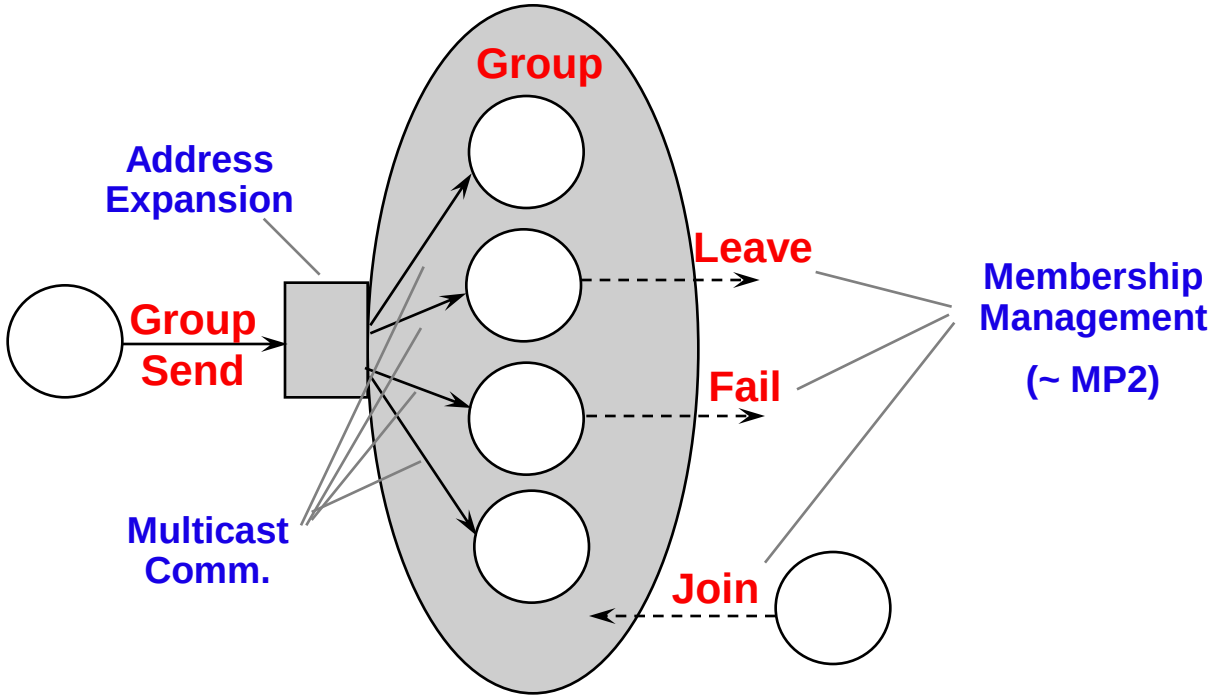
- A version of quorum selection where each replica has a number of votes. Quorum is reached by majority of votes (N is the total number of votes)

  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 r = w = 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 can be in the range 175ms to 850ms – why?.

Group Communication: A bulding block



- "Member"= process (e.g., an RM)
- **Static Groups**: group membership is pre-defined
- **Dynamic Groups**: Members may join and leave, as necessary

Views

- A *group membership service* maintains group *views*, which are lists of current group members.
  - This is NOT a list maintained by a one member, but…
  - *Each member maintains its own local view*
- A view $v_{p.i}(g)$ is process *p*'s understanding of its group (list of members) in "view number" i:
  - Example: $v_{p.0}(g) = \{p\}$, $v_{p.1}(g) = \{p, q\}$, $v_{p.2}(g) = \{p, q, r\}$, $v_{p.3}(g) = \{p,r\}$
- Whenever a member joins or leaves or fails, a new group view is disseminated, throughout the group.
  - Member detecting failure of another member reliable multicasts a "view change" message (requires at least causal-total ordering for multicasts)
  - The goal: the order of views received at each member is the same (i.e., view deliveries are "virtually synchronous")

Views

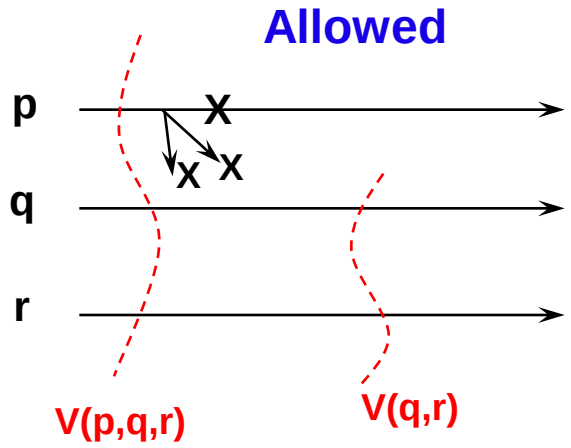
- An event is said to occur in a view $v_{p,i}(g)$ *if the event occurs at p, and at the time of event occurrence, p has delivered* $v_{p,i}(g)$ *but has not yet delivered* $v_{p,i+1}(g)$.
- Messages sent out in a view *i* need to be delivered in that view at ***all*** members in the group ("What happens in the View, stays in the View")
- Requirements for view delivery
  - Order: If *p* delivers $v_i(g)$ and then $v_{i+1}(g)$, then no other process *q* delivers $v_{i+1}(g)$ before $v_i(g)$.
  - Integrity: If *p* delivers $v_i(g)$, then *p* is in all $v_{*,i}(g)$.
  - Non-triviality: if process q joins a group and becomes reachable from process p, then eventually, q will always be present in the views that delivered at p.
    - *Exception: partitioning of group. Solutions to partitioning:*
      - *Primary partition: allow only majority partition to proceed*
      - *Allow any and all partitions to proceed*
      - *Choice depends on consistency requirements.*

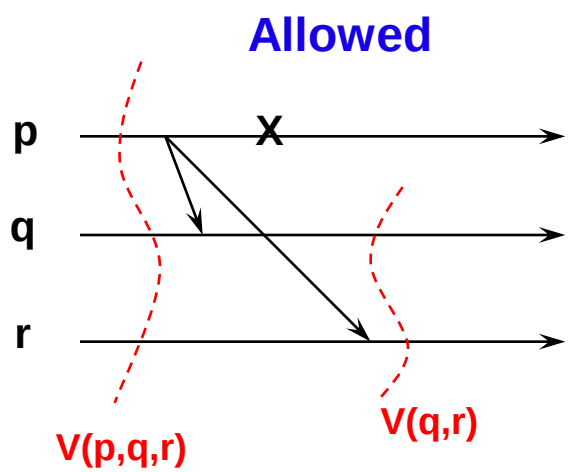
View Synchronous Communication

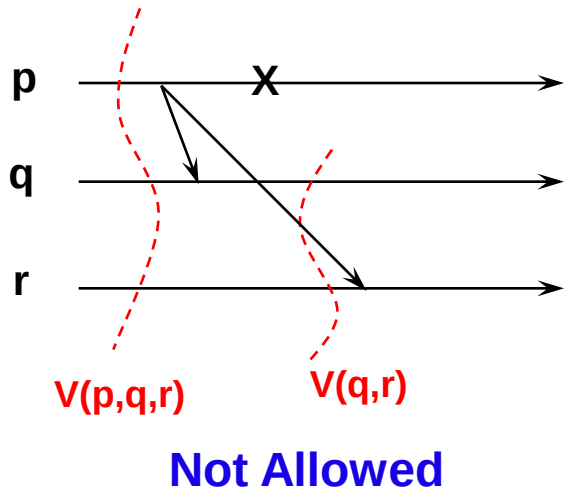
- **View Synchronous Communication = Views + Reliable multicast**
- **The following guarantees are provided for multicast messages:**
  - **Integrity: If *p* delivered message *m*, *p* will not deliver *m* again. Also *p* $\in$ *group (m), i.e., p is in the latest view.***
  - **Validity: Correct processes always deliver all messages. That is, if p delivers message m in view *v(g)*, and some process q $\in$ *v(g)* does not deliver m in view *v(g)*, then the next view *v′(g)* delivered at *p* will not include *q*.**
  - **Agreement: Correct processes deliver the same sequence of views, and the same set of messages in any view.**

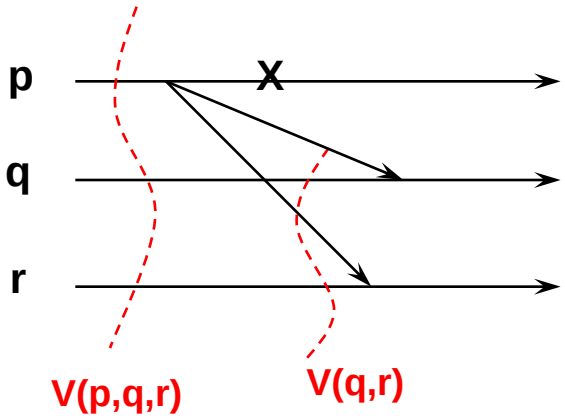
    if *p* delivers *m* in *V*, and then delivers *V′*, then all processes in *V* $\cap$ *V′* deliver *m* in view *V*
  - **All View Delivery conditions (Order, Integrity and Non-triviality conditions, from last slide) are satisfied**
- **"What happens in the View, stays in the View"**
- **View and message deliveries are allowed to occur at different physical times at different members!**

Example: View Synchronous Communication









Announcements

- **Please collect graded midterms, MPs and HWs**
- **MP3: By now, you should have started and have an initial design.**

State Transfer

- When a new process joins the group, state transfer may be needed (at view delivery point) to bring it up to date
  - "state" may be list of all messages delivered so far (wasteful)
  - "state" could be list of current server object values (e.g., a bank database) – could be large
  - Important to optimize this state transfer
- View Synchrony = "Virtual Synchrony"
  - Provides an *abstraction* of a synchronous network that hides the asynchrony of the underlying network from distributed applications
  - But does not violate FLP impossibility or CAP (since does not deal well with partition)
- Used in ISIS toolkit (NY Stock Exchange)

Optimistic Quorum Approaches

- An Optimistic *Quorum selection* allows writes to proceed in any 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 (and it's likely the partition may have healed by then)
- Optimistic Quorum is practical when:
  - Partitions are relatively short-lived
  - 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

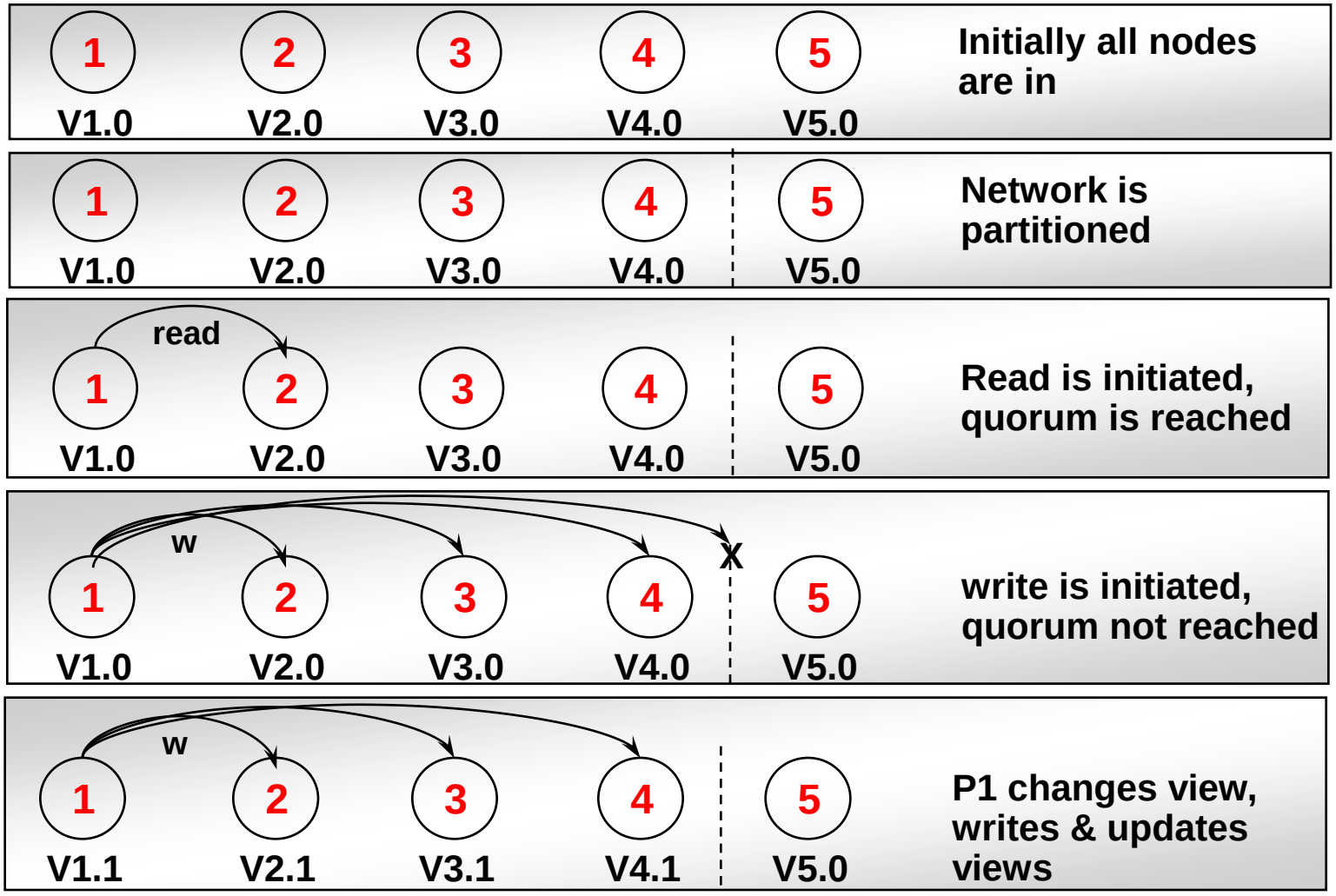
- An *optimistic* approach
- Quorum is based on views at any time
  - Uses view-synchronous group communication as a building block
- Once the partition is repaired, participants in the smaller partition know whom to contact for updates.

View-based Quorum - details

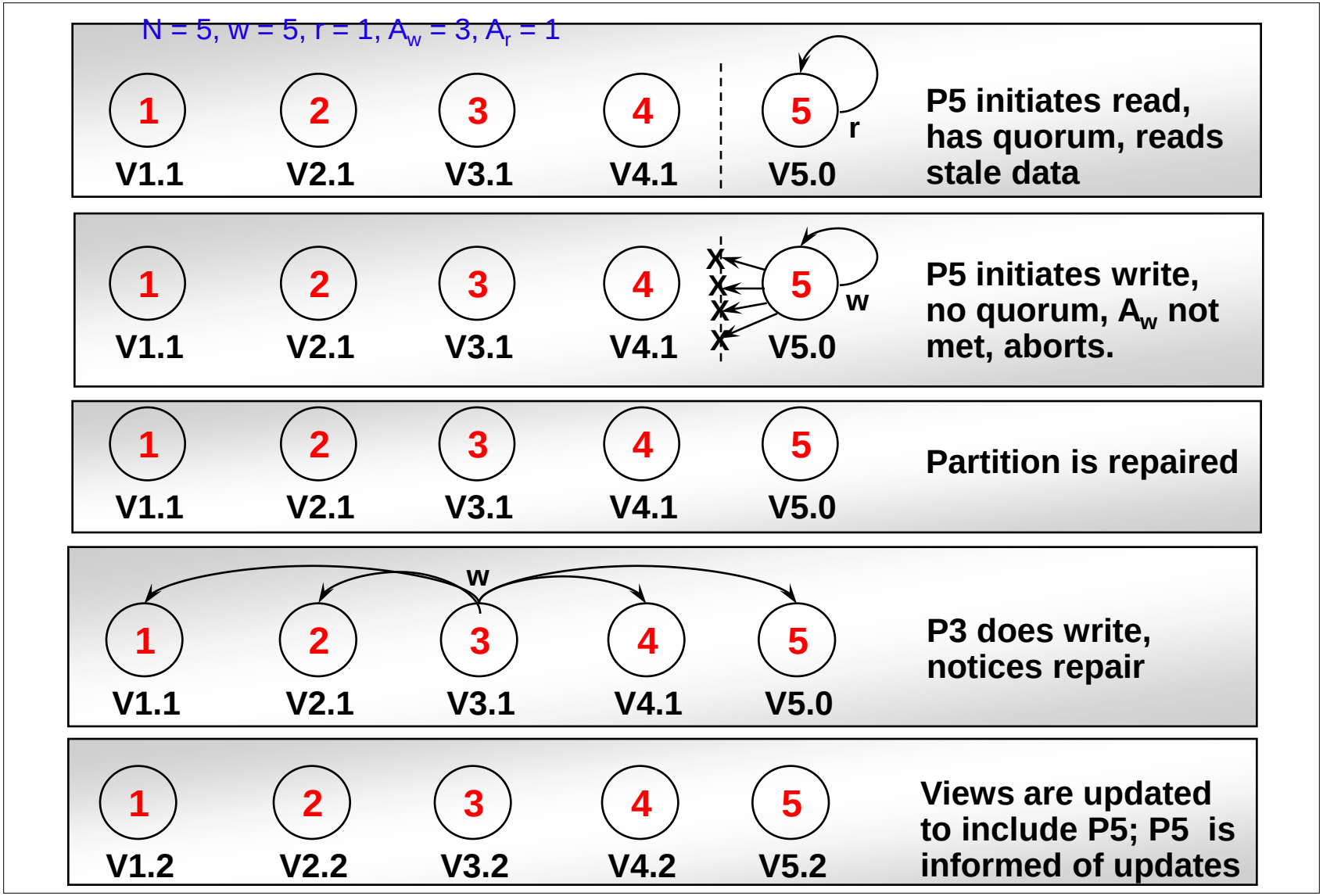
- Views are per object, numbered sequentially and only updated if necessary
- We define thresholds for each of read and write :
  - A_w: minimum nodes in a view for write, e.g., $A_w > N/2$
  - A_r: minimum nodes in a view for read
  - E.g., $A_w + A_r > N$
- If ordinary quorum cannot be reached for 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, $View_{size} \geq A_w$
- The first update after partition repair forces restoration for nodes in the smaller partition

Example: View-based Quorum

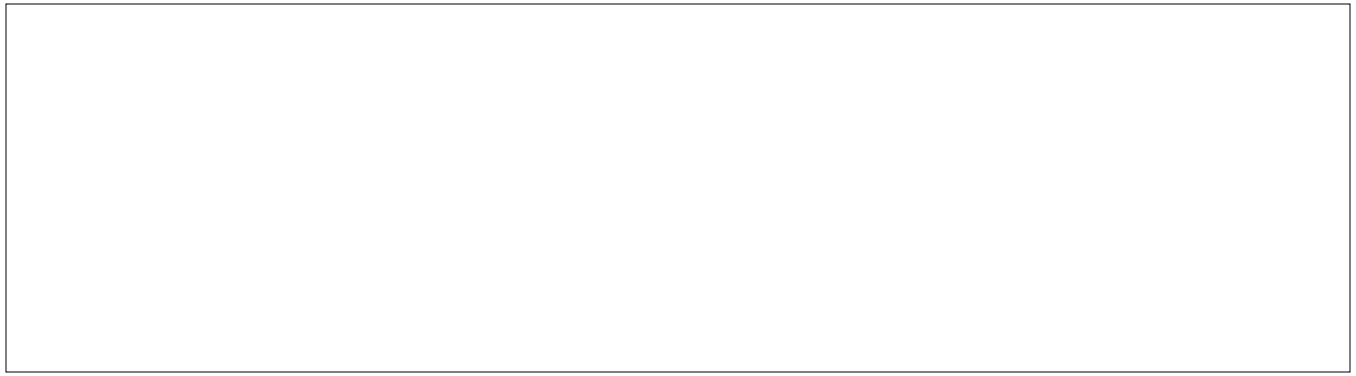
❖ **Consider: N = 5, w = 5, r = 1, A_w = 3, A_r = 1**



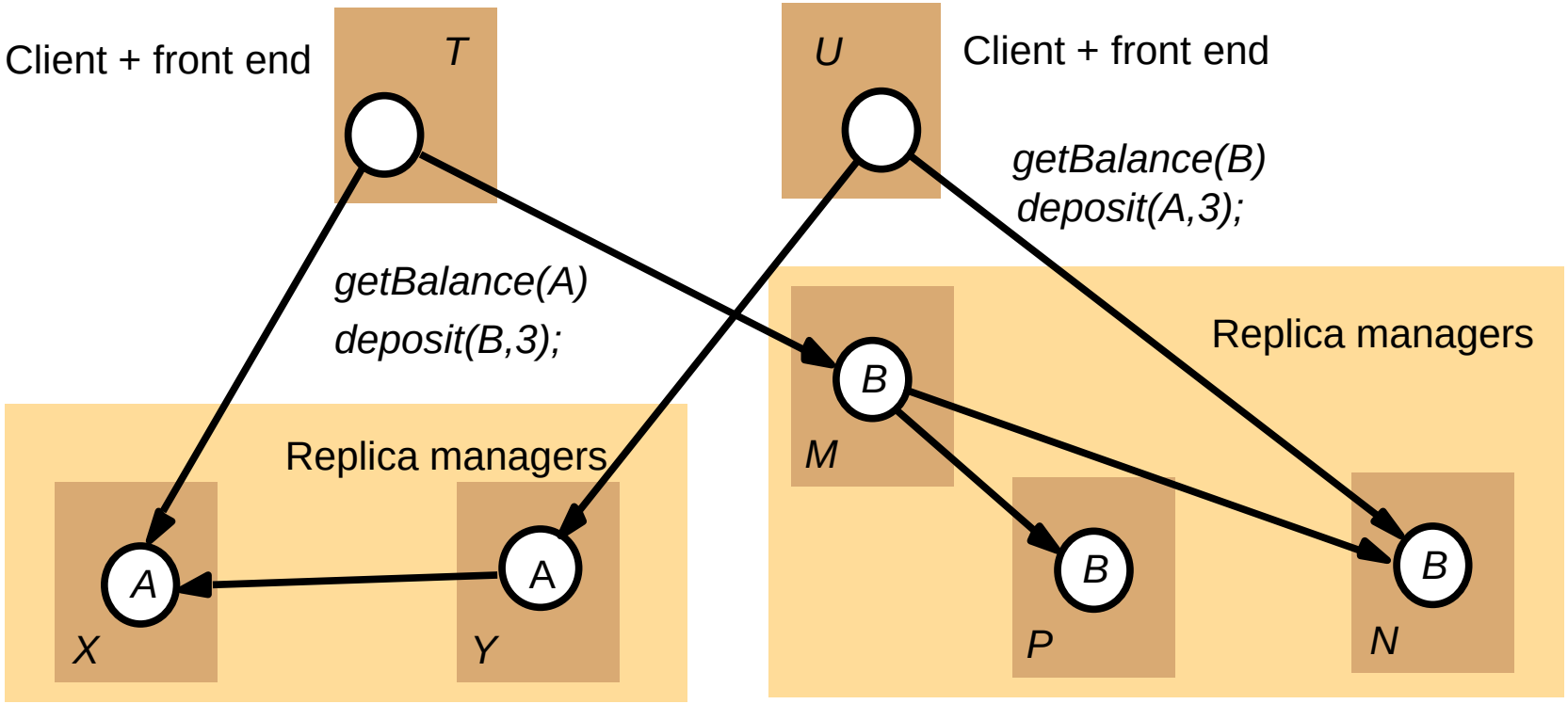
Example: View-based Quorum (cont'd)



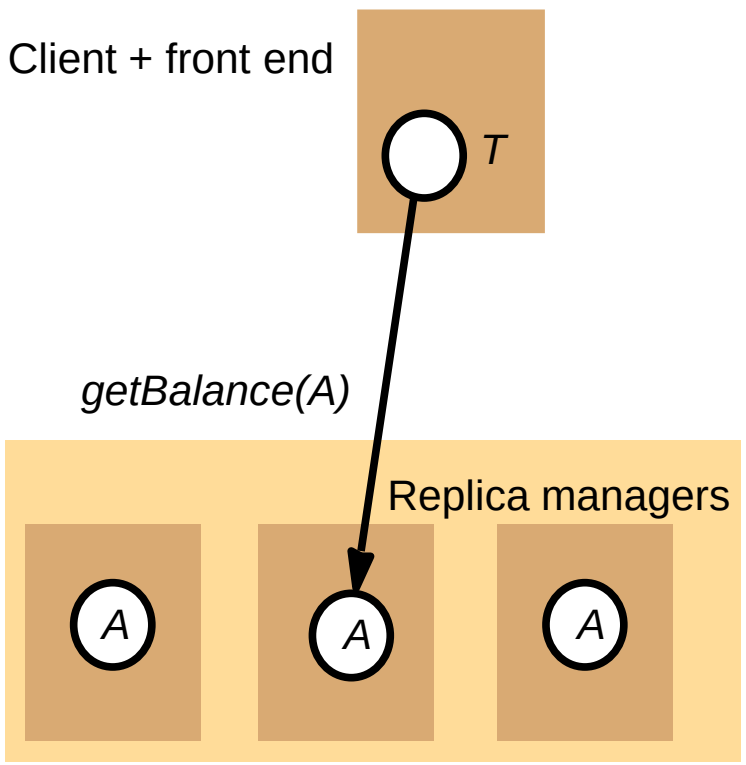
Optional Slides (Not Covered)

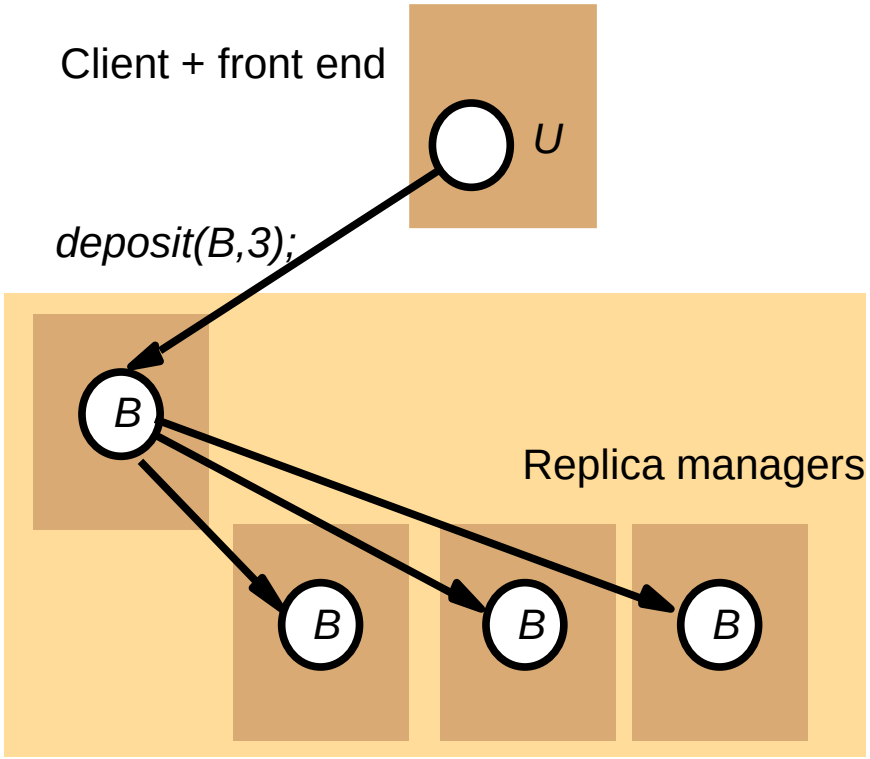


Available Copies Approach



Transactions on Replicated Data





The Impact of RM Failure

- **Assume that (i) RM X fails just after T has performed *getBalance*; and (ii) RM N fails just after U has performed *getBalance*. 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 entire 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.
  - Thus: N fails → T reads object A at X; T writes objects B at M and P → T commits → X fails.
- From U's perspective,
  - Thus: 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 checks 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.
- Caveat: Local validation may not work if partitions occur in the network

Two Phase Commit Protocol For Transactions on 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.**

Available Copies Replication

- **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.**