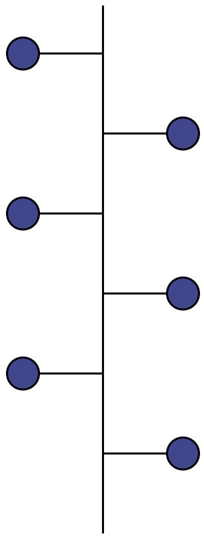
Introduction to Communication Networks and Distributed Systems



Unit 8: Algorithms for distributed systems

Prof. Dr.-Ing. Adam Wolisz

TKN Telecommunication Networks Group

Algorithms for distributed systems

- Overview
  - Transactions and Distributed coordination

    Very valuable additional set of slides:

    www.ics.uci.edu/~cs223/handouts/2-3pc.ppt

  - Replication

Transactions

- A special case of atomic actions, originally from databases
- Sequence of operations that transforms a current consistent state to a new consistent state
  - Without ever exposing an inconsistent state
- Example
  - Move $10 from my savings account to my checking account
  - Basically, subtract $10 from savings account, add $10 to checking account.

    Be aware: Both might run on different computers!

  - But never “lose” my $10 - and never give me an extra $10!

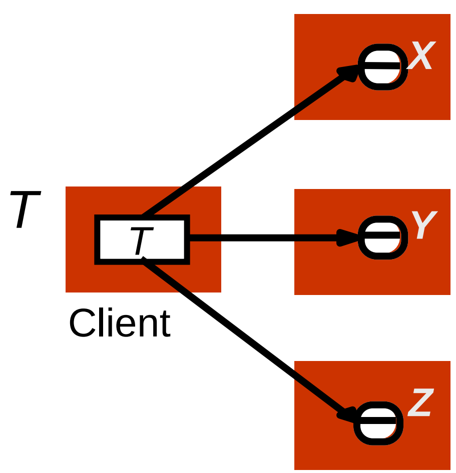
    Even if some computers will crash!

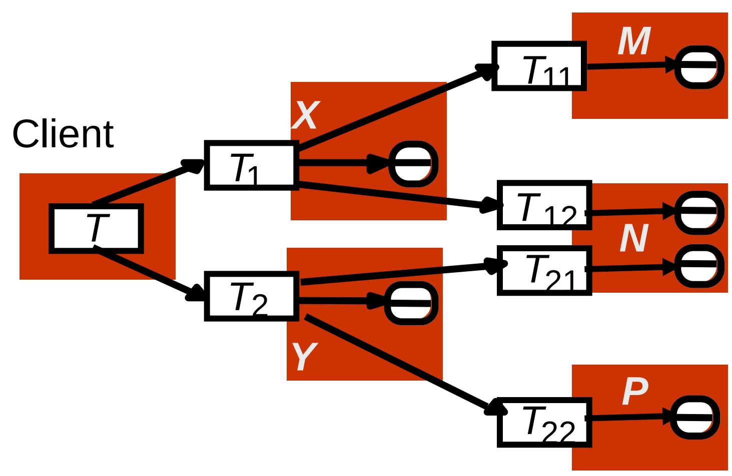
Basics

- Distributed transactions access objects located on spatially distributed servers (nodes)
  - Atomicity: Transaction is either committed by all nodes or aborted by all nodes

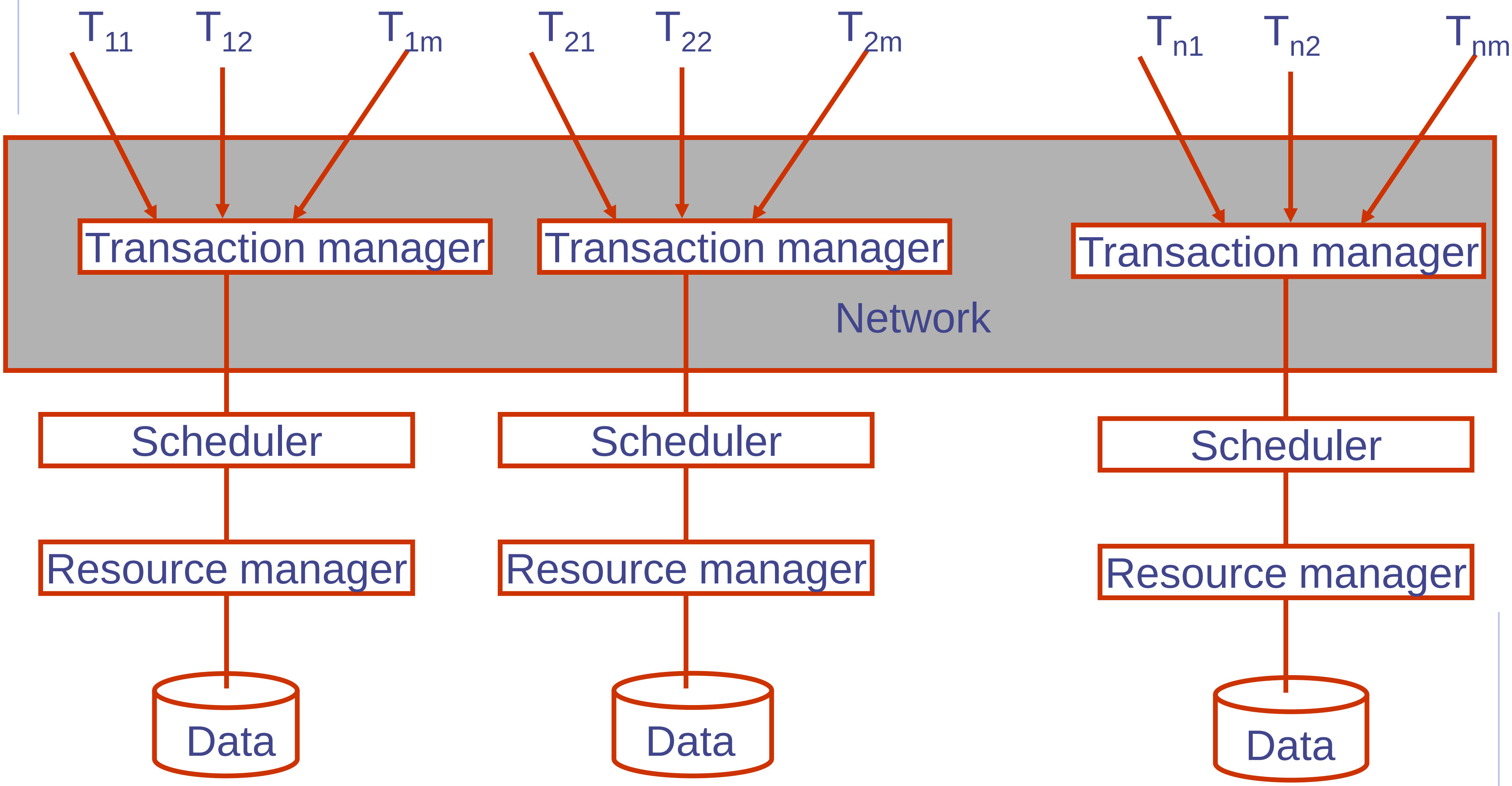
  ⇒Coordination by one or by a group servers (nodes) necessary

- Transaction types
  - Plain transaction: current transaction finished, before starting a new one
  - Nested transaction: top level transaction can start further transactions



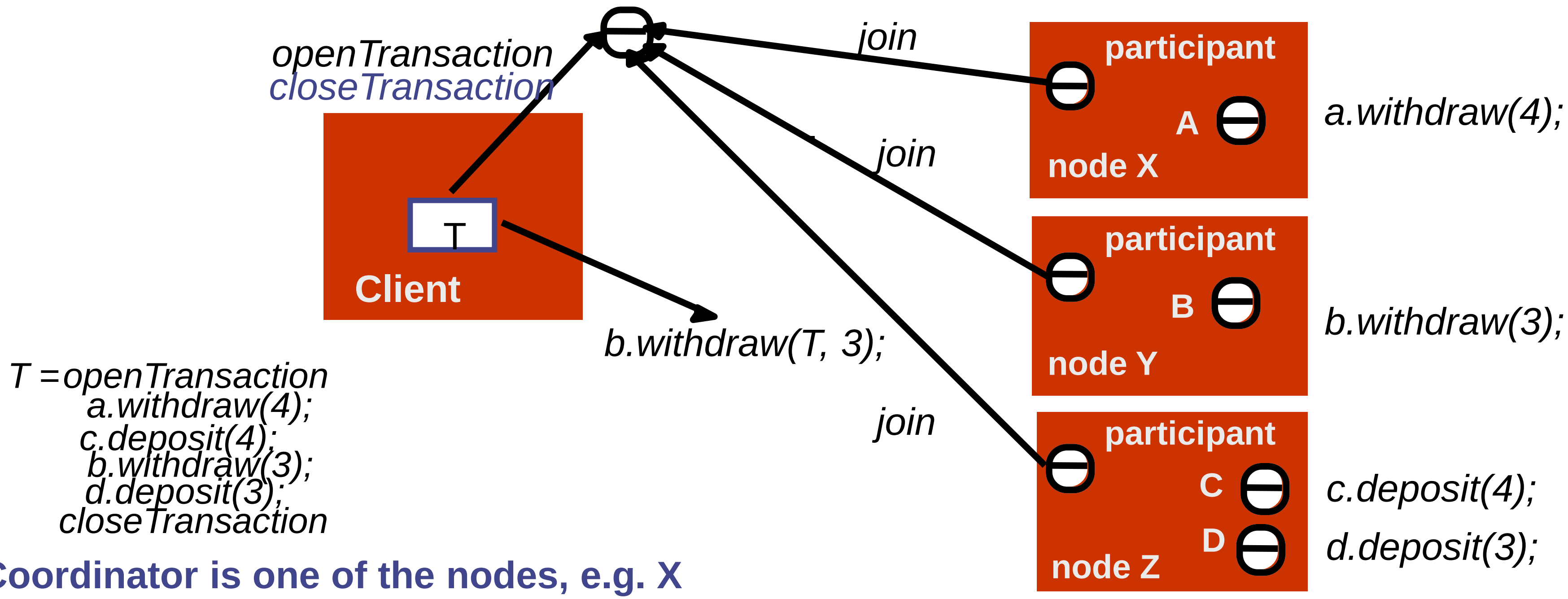


Transaction systems architecture



Coordinating distributed transactions

- Client starts a distributed transaction by openTransaction request
  - Manager opens the transaction, assigns a unique ID
  - Manager coordinates the transaction processing $\Rightarrow$ responsible for commit/abort of the transaction
  - participant = server (node) with required objects (also called *cohort*)



Coordinator is one of the nodes, e.g. X

Coordinator (2)

- During the execution of a distributed transaction
  - Coordinator creates a list with references to all participants
  - Participants acknowledge the coordinator
  - New participants can be inserted with functions such as join (Trans-ID, reference to a new participant)
    - Coordinator must be informed about new participants
- Coordinator calls closeTransaction

What Problems Could Arise?

- Other processes could write the variables
- Other processes could read the variables
- Failures could interrupt the process
- How can we avoid these problems?

Running Transactions

- Multiple transactions must not interfere
- You can run them one at a time
  - Or run them concurrently…
  - But avoiding all interference
- Serializability avoids interference
  - A property of a proposed schedule of transactions
  - A serializable schedule produces the same results as some serial execution of those transactions
  - Even though the actions may be have been performed in a different order

Indivisible actions, atomic action…

- If several users access a common resource with a sequence of actions at the same time, race conditions occur and the system can become inconsistent.
- Indivisible action
  - some actions are indivisible (e.g. actions on semaphores)
  - these can be used to group a sequence of actions into an indivisible action
- Atomic action (or transaction) with following characteristics
  - Indivisible, i.e. performed in one step;
  - Either successful or „has never happened“,
    - ⇒data is not corrupted by only partially executed action sequences
  - Can consist of „normal“ actions or embedded atomic actions.

Commit and Abort

- A commit is an unconditional guarantee that a transaction will be completed
- An abort is an unconditional guarantee that a transaction will be completely backed out
  - Requires returning system to its pre-transaction state
  - Typically done
    - either by logging/rolling back the changes or
    - by delaying updates until commit point is reached
- In both cases, regardless of multiple failures
- All transactions eventually either commit or abort

Distributed Commitment Mechanisms

- Mechanisms to guarantee atomicity of actions in a distributed system
- Important mechanisms for building distributed transactions
- Works with mechanisms to permit backing out of transactions

Distributed coordination

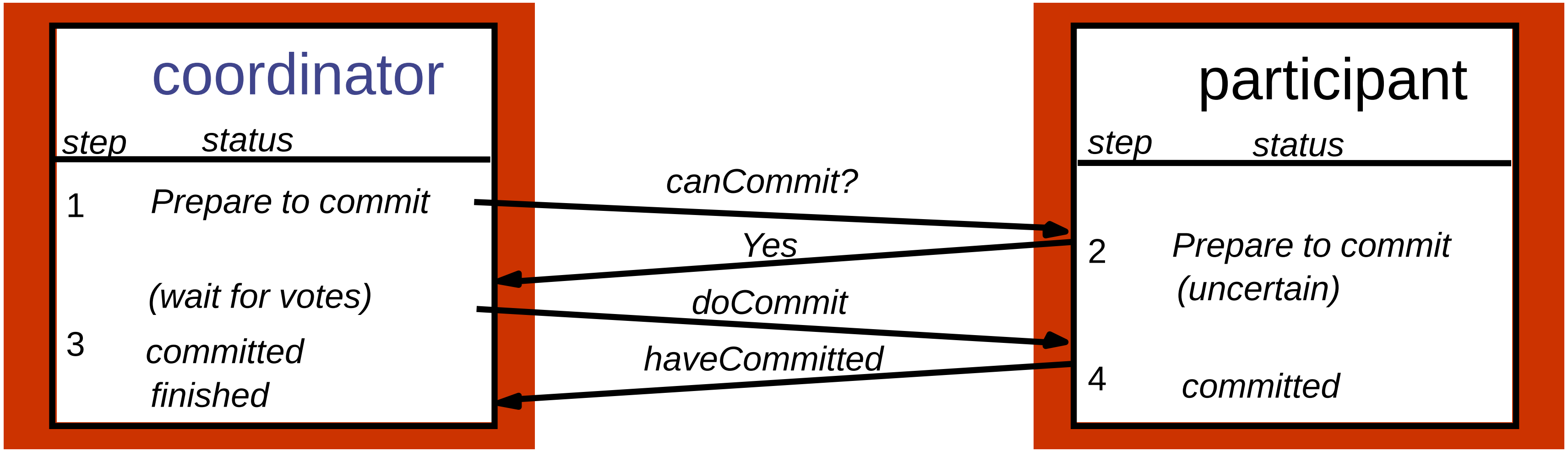
- Guaranteed correctness for distributed transactions is a complex issues, as node and networks failures can occur at any time
- Demands on commit protocol
  - All nodes meet the same decision
  - One node is not allowed to change the decision once the decision was acknowledged
  - Decision to commit the transaction is allowed if and only if all nodes can commit
- Simplest implementation: 1 Phase commit (1PC):
  - Coordinator requests all participants to commit
  - Useless in real world
- Two phase commit protocol:
  - Phase 1: ask all participants about the status
  - Phase 2: decide on commit/abort, send the decision, request acknowledgments

Two Phase commit protocol (2PC)

- 2PC protocol with four layers
  - vote request
  - Vote
  - Decision
  - Acknowledge
- Operations
  - canCommit?(trans)-> Yes / No: Coordinator asks if the participant can commit the local transaction, participants vote
  - doCommit(trans): Coordinator -> participant: commit transaction
  - doAbort(trans): see doCommit(), now with abort
  - haveCommitted(trans, participant): Participant -> coordinator: Transaction committed
  - getDecision(trans) -> Yes / No: Participant -> coordinator, participants wants to know the decision on a certain transaction after the participant voted but did not receive an answer $\Rightarrow$ Time out to recognize server failures

2PC procedure

- Coordinator asks all participants canCommit(), get the replies
- Coordinator evaluates the votes (also its own vote)
  - All votes Yes, then doCommit() for all nodes
  - One No vote, then call doAbort() to roll-back the transaction
- All participant wait for the command, acknowledge the execution
- If committed, $\Rightarrow$ haveCommited message sent to the coordinator



Characteristics of Two-Phase Commit

- Conditions to agree:
  - Access locked for other transactions
  - Possibility for atomic activation/roll back assured!
    (writing the state to permanent storage!!!)
- timeouts handle lost/delayed messages

Details: Abort by Write-Ahead Logs

- An operation-based abort mechanism
  - Record operations and undo them, if necessary
- For each operation performed, record enough information to completely undo it
  - Either old state or other information
- Undo log

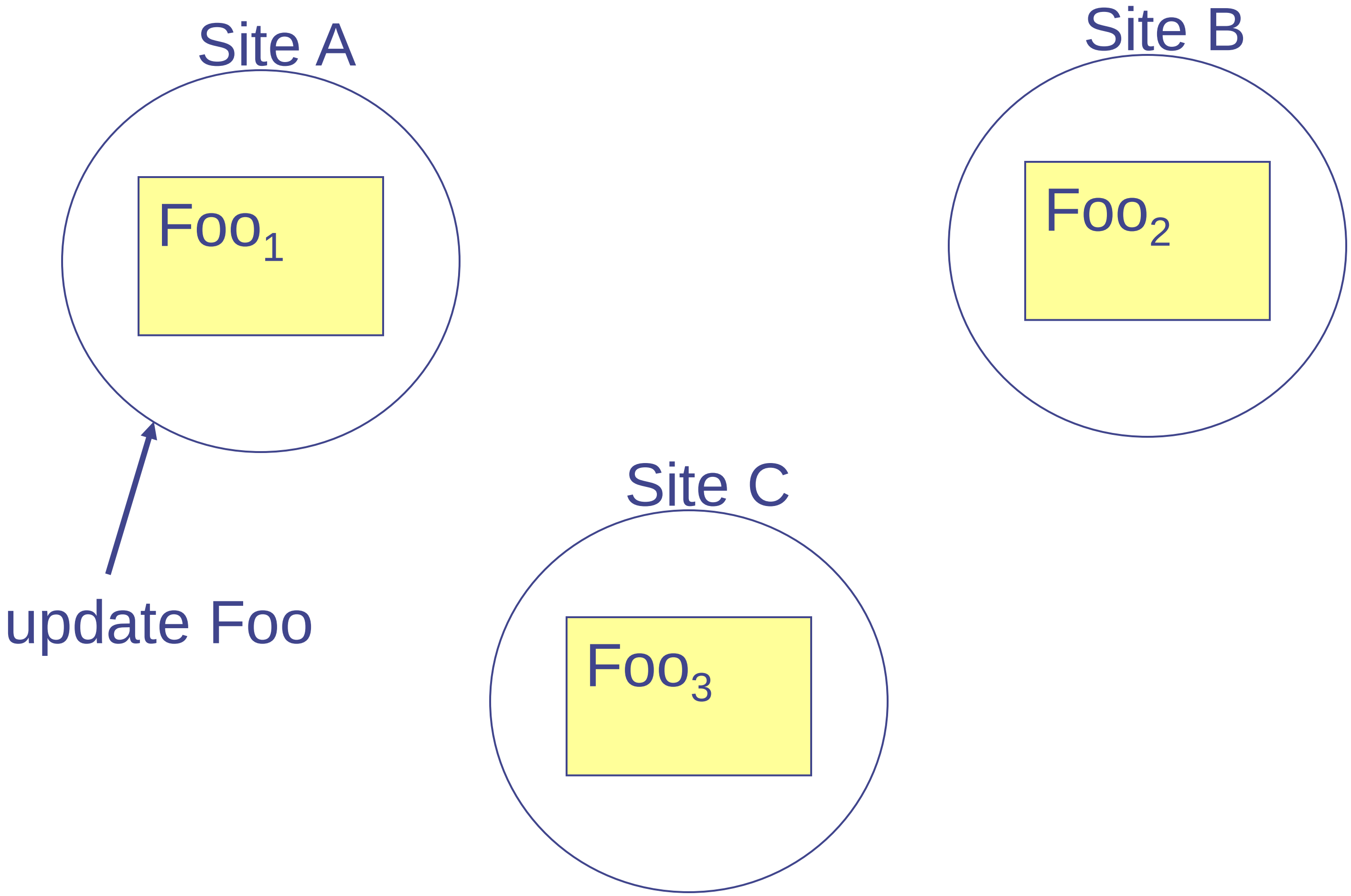
- Write-ahead Log Protocol
  - Write the undo log to stable storage
  - Make the update
  - If transaction commits, undo log can be deleted/garbage collected
  - If transaction aborts, use undo log to roll back operation
  - And then delete/garbage collect the log

Details: Abort by Shadow Pages

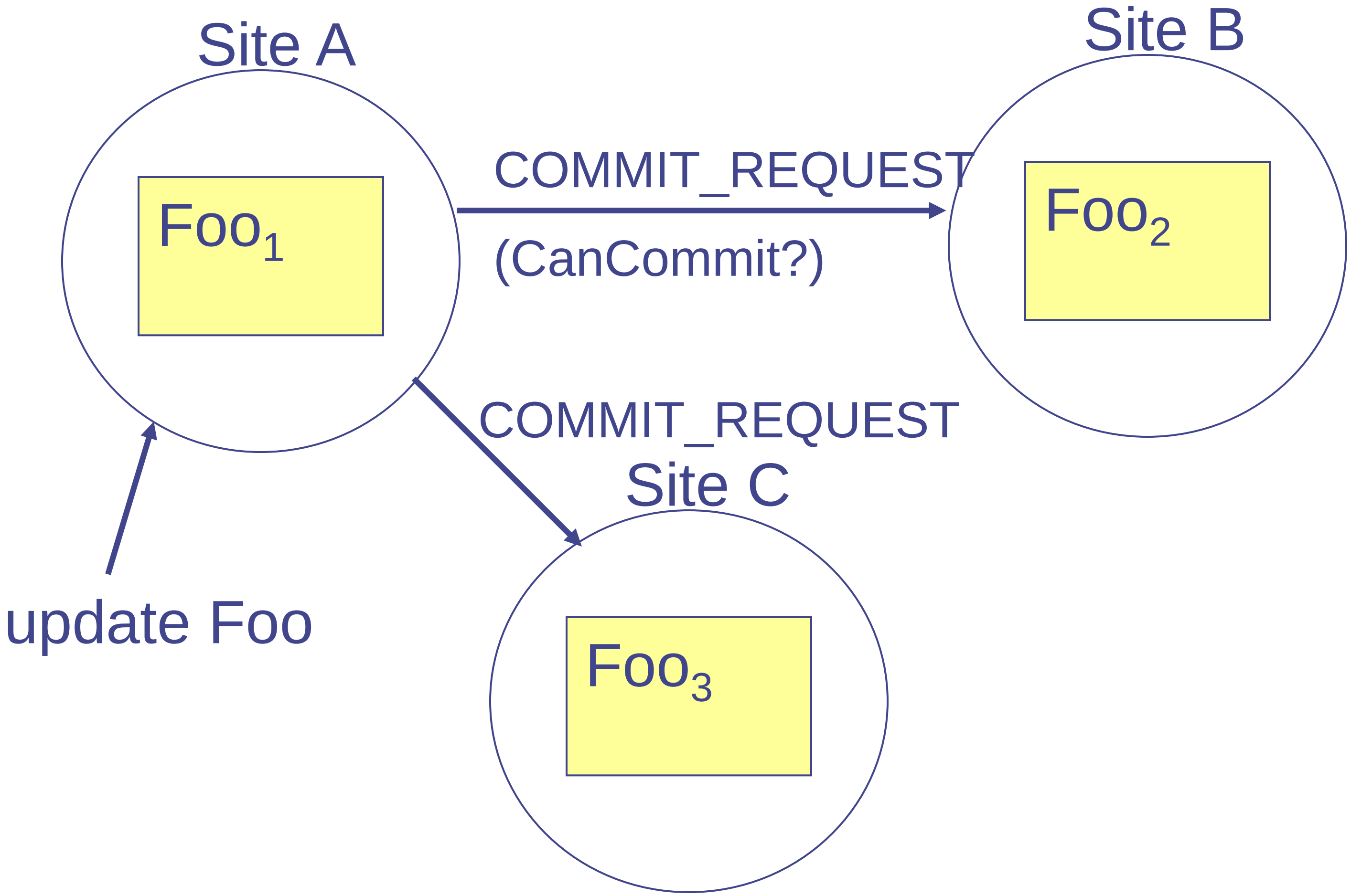
- State-based approach
- Save a checkpoint of the state before performing transaction operations
- If transaction aborts, restore old state
- Can be expensive if state is large
- Shadow paging limits the cost

- Before writing a data item in a transaction
    1. Make a complete copy of its page
    2. Allow transaction to use the copy (transparently)
    3. On commit, switch shadow page for new page
    4. On abort, restore shadow page

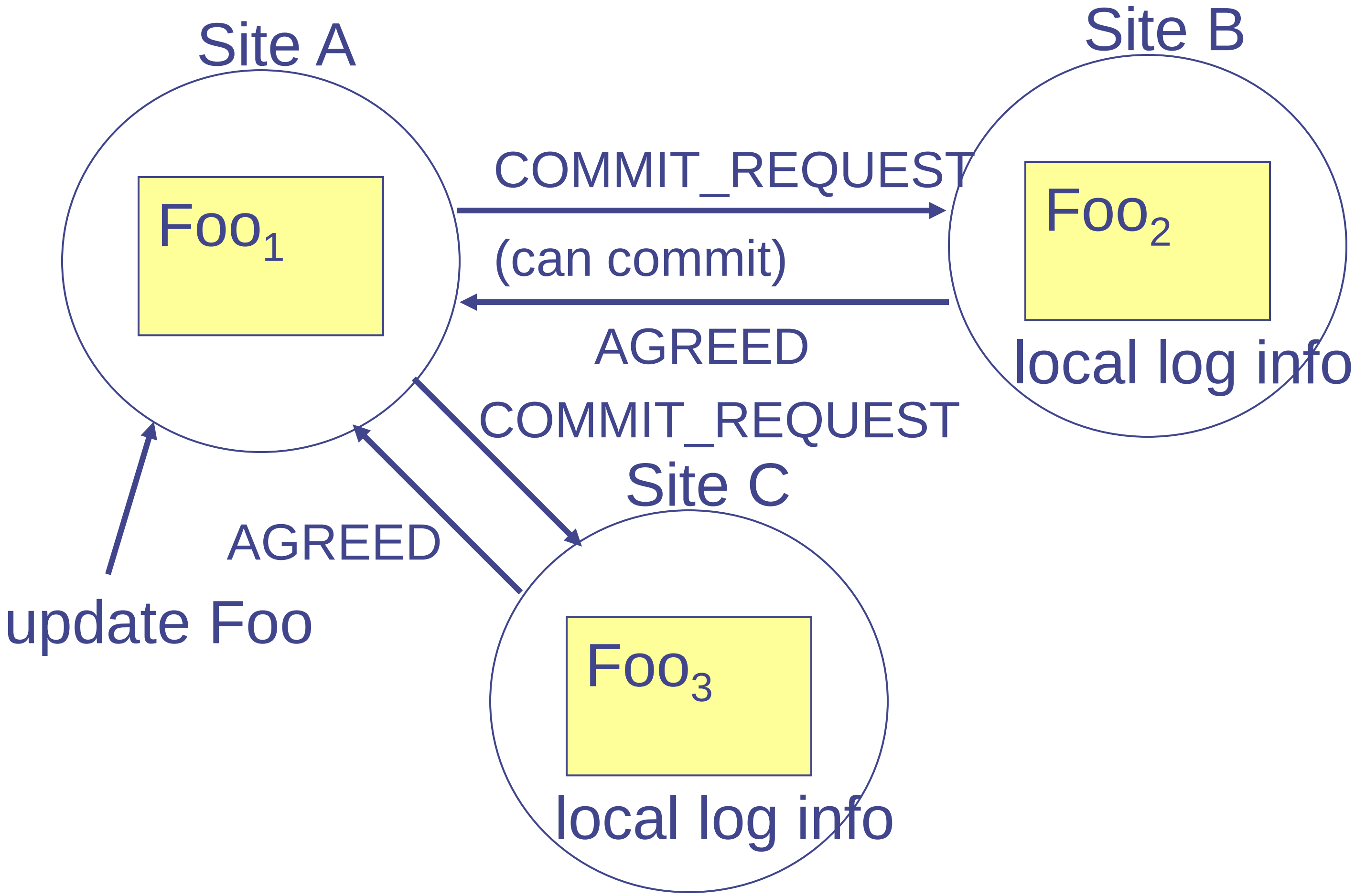
Two-Phase Commit Diagram (Phase 1)



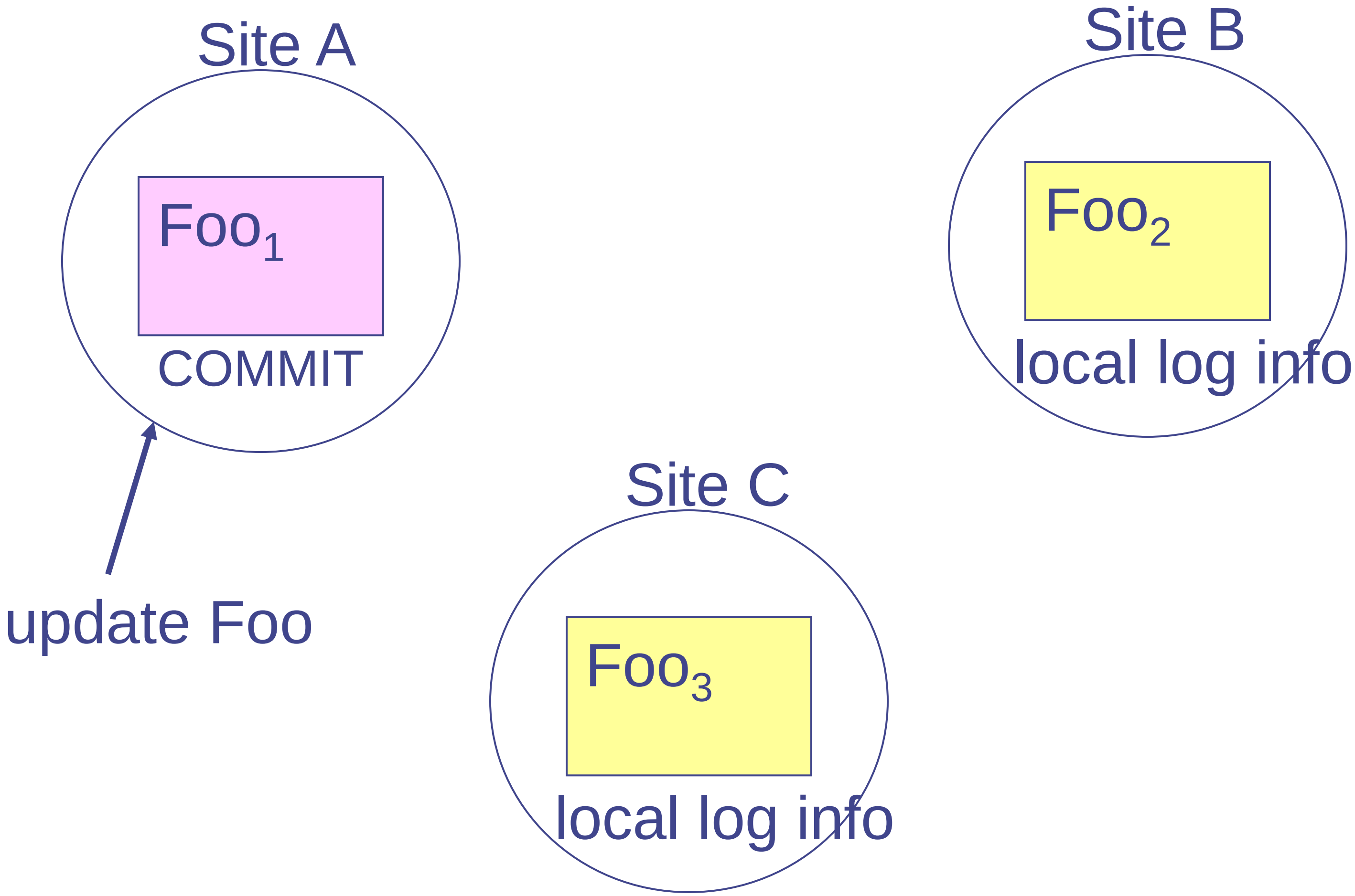
Two-Phase Commit Diagram (Phase 1)



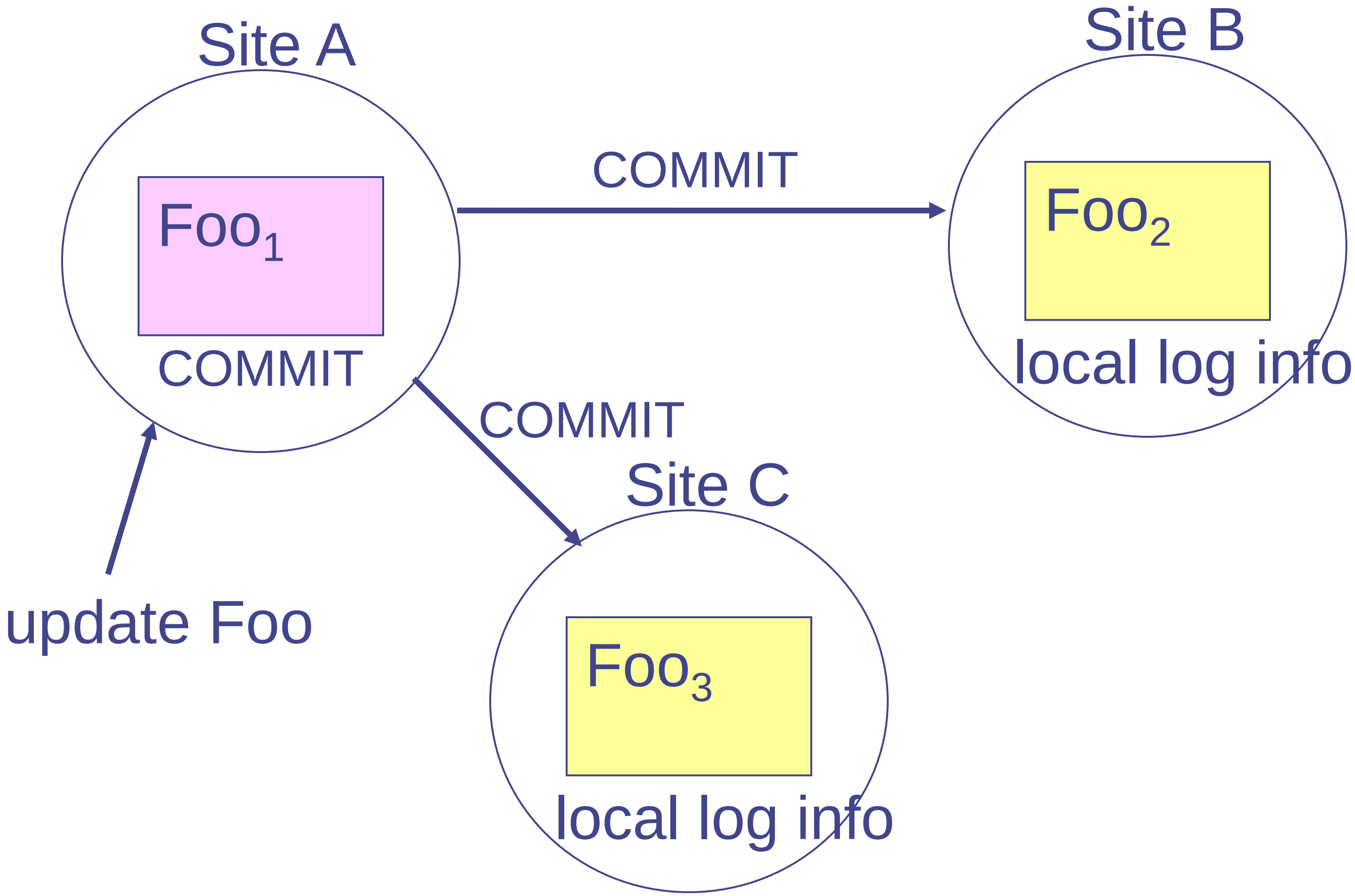
Two-Phase Commit Diagram (Phase 1)



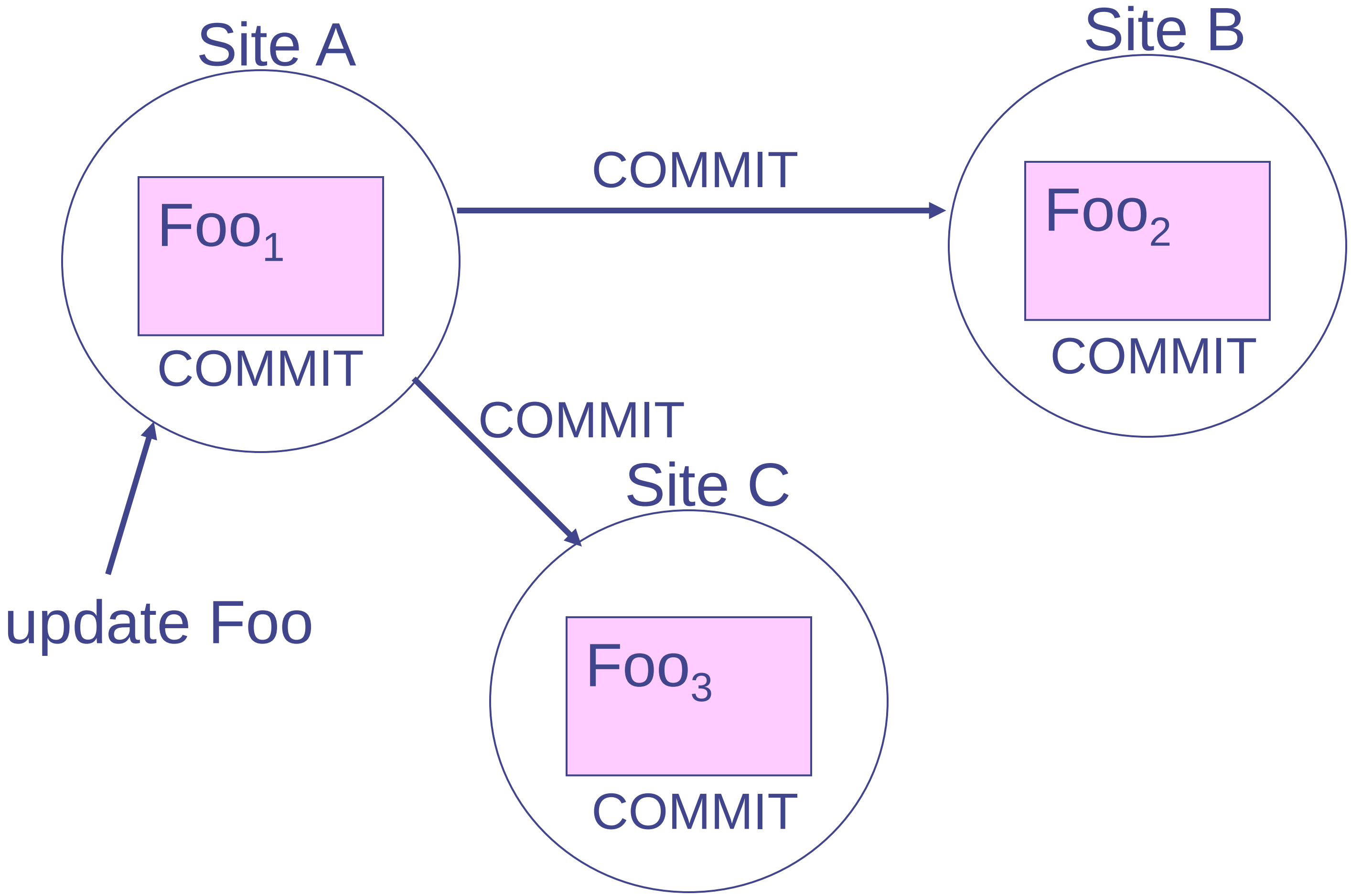
Two-Phase Commit Diagram (Phase 2)



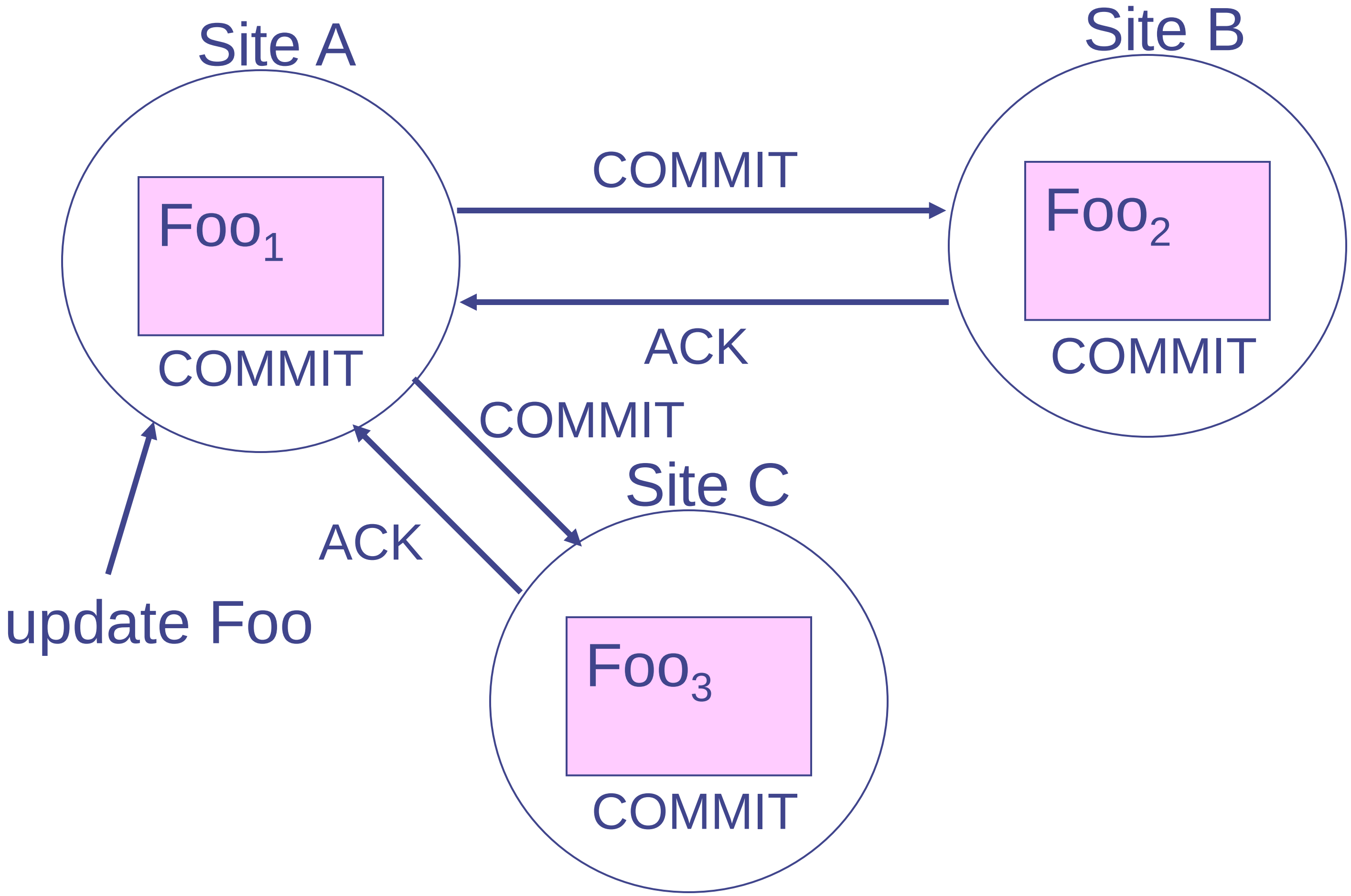
Two-Phase Commit Diagram (Phase 2)



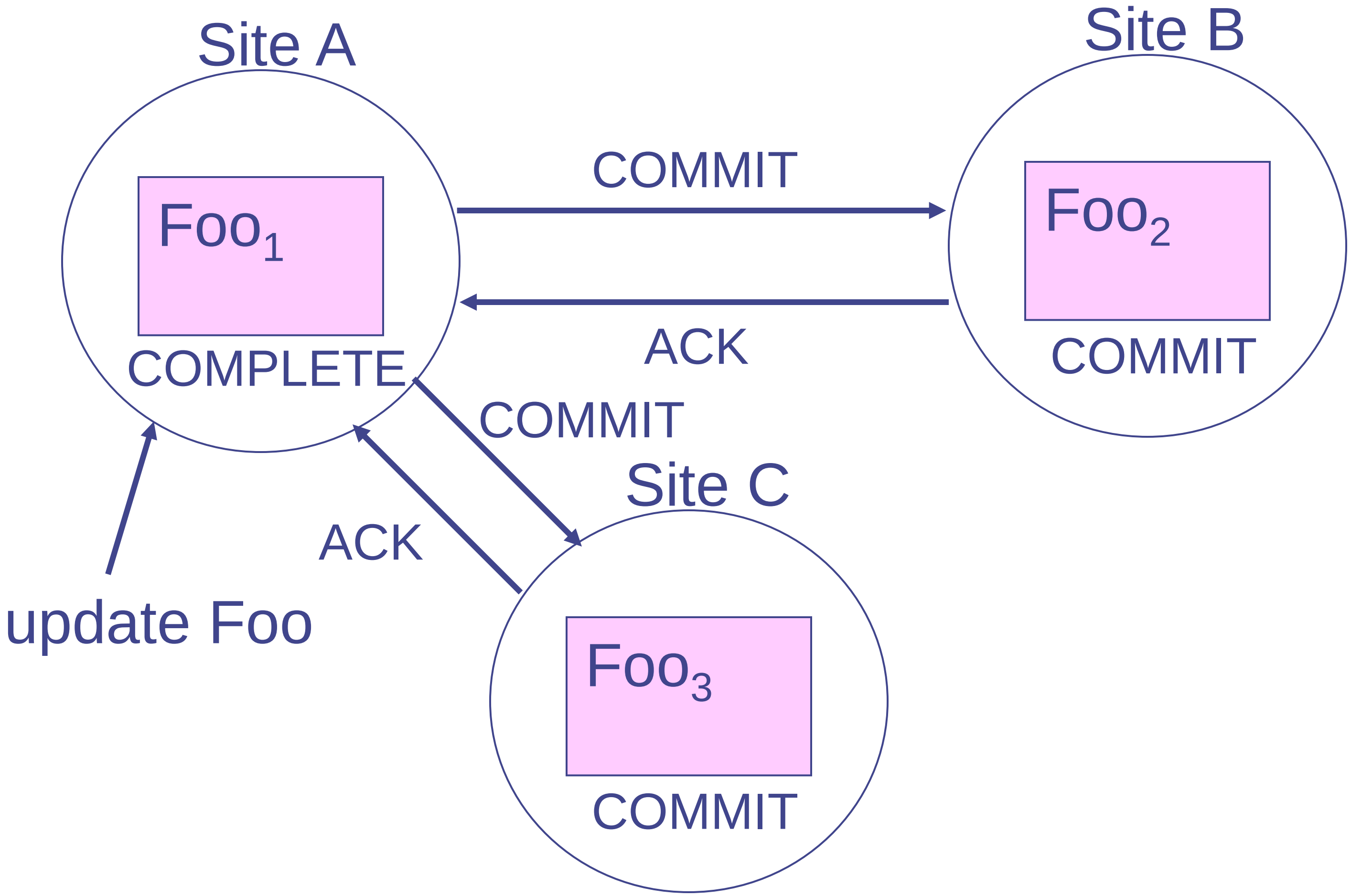
Two-Phase Commit Diagram (Phase 2)



Two-Phase Commit Diagram (Phase 2)



Two-Phase Commit Diagram (Phase 2)



Two-Phase Commit Failure Recovery

- Coordinator fails before writing COMMIT record to log
  - On recovery, broadcast ABORT
- Coordinator fails between COMMIT and COMPLETE
  - On recovery, broadcast COMMIT
- Coordinator fails after writing COMPLETE
  - Transaction succeeded, do nothing
- Cohort crashes in Phase 1
  - Coordinator aborts transaction
- Cohort crashes in Phase 2
  - On recovery, check with coordinator whether to commit or abort

Three-Phase-Commit- Protocol (3PC)

- 2PC problem: some nodes may remain in state uncertain for long time, if the coordinator fails and no
  - ➔ getDecision() requests are replied
- 3PC: non-blocking improvement of 2PC
- Prerequisites
  - Failure of K<N nodes at most, where K is a user-set parameter
- Idea
  - Blocking in 2PC results from a situation, where not all failed nodes are in state „uncertain“ and can not decide „abort“, because some nodes maybe already decided to commit
  - 3PC avoids this situation by introducing a new layer, so a node can not decide to commit as long as a non failed node is in state „uncertain“
- Price
  - Management overhead much higher than in case of 2PC:
    - 6N messages and 3(N1) logging processes necessary

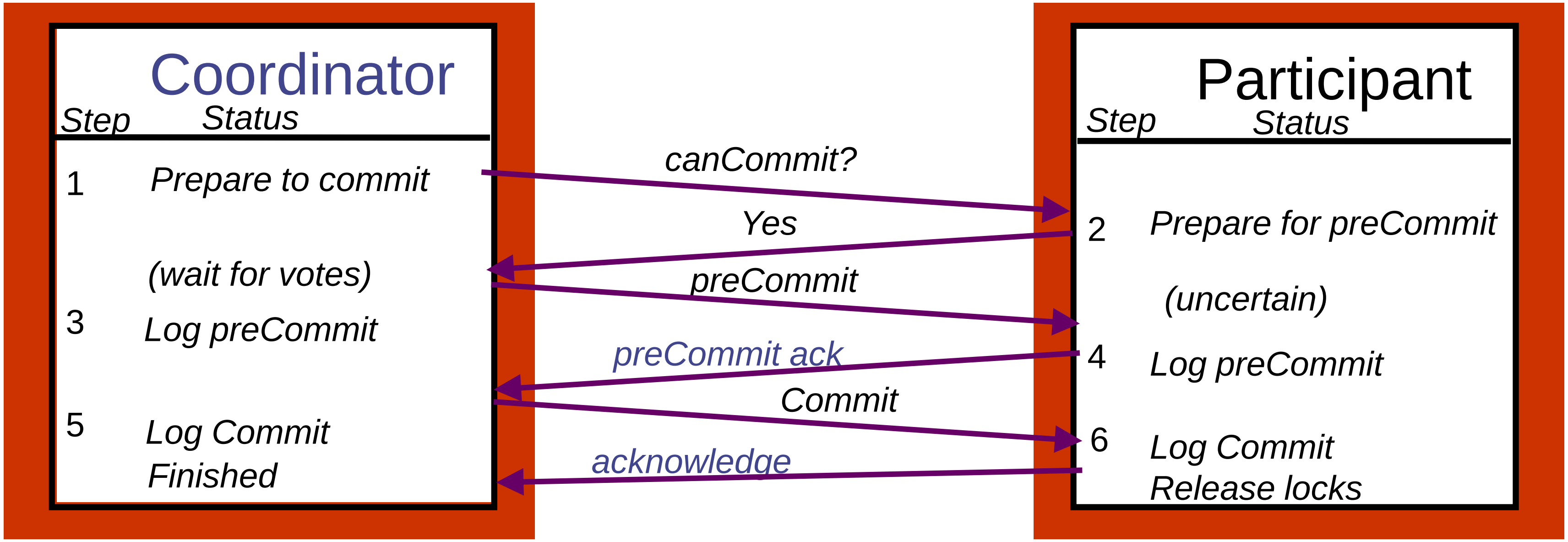
3PC procedure

- Phase 1: analogously to 2PC
  - The coordinator collects all votes and makes a decision
  - No: Abort and information to all participants that voted yes
  - Yes: all nodes answer Yes $\Rightarrow$ additional layer $\Rightarrow$ Application 3PC
- Phase 2:
  - Coordinator changes mode to preCommit and guarantees that the transaction will not be terminated

    (If the coordinator fails, a transaction roll back is still possible!)
  - Coordinator logs the new state and sends preCommit to all participants
  - Participants log the new state and acknowledge the preCommit request
- Phase 3:
  - Coordinator collects all answers: if all messages arrived, a commit is executed and doCommit is sent to all participants $\Rightarrow$ Participants write the data in the database and confirm transaction

3PC procedure (2)

- Coordinator failure recognized by applying time-out
  - All active participants elect a new coordinator
  - commit protocol starts, so the new coordinator can proceed the commit handling



3PC commit protocol

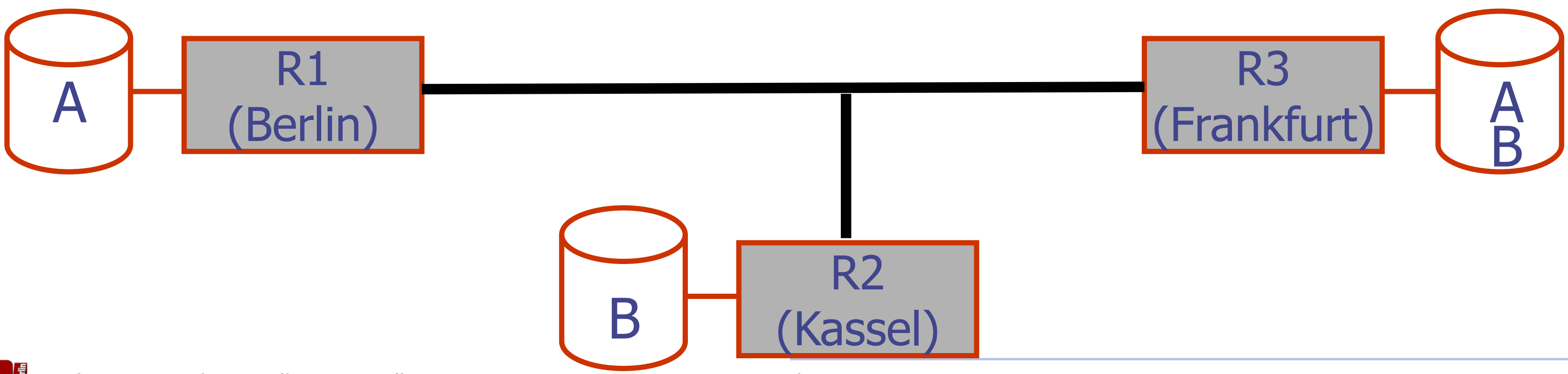
- New coordinator collects status information from all participants and proceeds according to the following rules
  - R1: If one node is in state finished or aborted, decide abort
  - R2: If one node is in state commit, decide commit
  - R3: If all nodes are in state uncertain, decide abort
  - R4: If some nodes in state preCommit, but no node in state commit, send preCommit to all nodes
- Wait for all acknowledgments and decide commit
- Ignore all participants that do not transmit their status

3PC example

- All participants vote Yes
- Coordinator sends preCommit messages. Node failure occurs during the process, so only a part of the nodes receives the messages
- Now, some nodes are in state preCommit, all other nodes are in state uncertain
- Assume, nodes in state preCommit crash too
  - Remaining nodes start the commit protocol
    - The (elected) new coordinator collect the state information of all nodes that are still operational (all uncertain) and decide according rule R3 abort
  - Failed nodes start after reboot the commit protocol and receive the decision abort
    - Although they already received preCommit, nodes decide abort

Replication

- Replication goals
  - Increased availability
    - Copy data to k nodes $\Rightarrow$ Data available even if k-1 nodes crash
  - Performance increase
    - Parallel execution of read access for the same data
    - Reduction of necessary communication by supporting data locality
- Example: Replicated account information



Demands and approaches

- Management of replicated data increases the storage demands and the communication load during write access
- Increased implementation effort to hide the existence of replicated data for the user
  - Automatic, transparent update of all replicated data after modification of an object
  - Ensuring data consistency considering all replicated data sets
- Three well-known approaches for update and synchronization of replicated databases
  - Write-All: Synchronous updates of all replicated nodes
  - Primary-Copy: Immediate update of a master copy, modifications submitted with certain delay
  - Voting: Each replicated set receives one or more votes. For each read or write access, a certain quorum for read or write access must be collected before the operation is performed

Write-All approach

- Write-All-Read-Any or Read-Once-Write-All (ROWA) strategy
  - Synchronous modification of all replicated date before the transaction is committed
  - Each replicated data is updated at any time and can be used for read access (in parallel to the other sets)
  - Selection of data set for reading based on criteria such as minimal network load or least node utilization
  - Singular node failures easy to compensate

- Advantages of ROWA strategy
  - All read accesses of the primary data set (R2) can be performed on replicated data without any delay
  - Data access for all data despite failures of individual nodes

ROWA disadvantages

- Locking protocols have to be changed
  - All replicated data sets on all involved nodes have to be locked before a write access can occur
  - All nodes must participate on the commit protocol
- Crucial disadvantage
  - Availability decreased compared to non-replicated databases
  - If a single node crashes, then the entire database fails, because all nodes with replicated data sets have to be considered
- Relaxed demand: Write-All-Available-Read-Any
  - The modified data sets are updated on the available nodes only
  - In case of crashed nodes, the updates are logged and recovered after reboot
  - Problems in case of network separations: node crashed or messages lost?

Primary-Copy

- Goal
  - Efficient processing of updates
  - One selected data set is determined as primary (master) copy and it is updated immediately
  - Other replicated data sets are updated asynchronously from the primary node as soon as possible
- Efficiency
  - Update messages are transferred as bundle to the target node
  - Primary copies stored on different nodes to avoid bottle necks and hot spots
- Disadvantage: Deferred modification of replicated sets
- Implementation
  - Write locks requested for all copies (same as ROWA), but only the primary copy is updated immediately
  - Primary copy node updates as soon as possible all other objects and releases the locks after finishing all operations

Alternatives

- Write locks requested only for the primary copy $\Rightarrow$ reduced number of locking conflicts
- Handling for read requests on old, possibly inconsistent data necessary
  - Read on primary copy: All read transactions refer the primary copy $\Rightarrow$ fault tolerance, but no locality and parallelism
  - Read access to local copies, lock requests for primary copy node: Solely locks increase the load of the primary copy, the reading occurs on other nodes
    - Check during locking, if the object has to be updated and if so then update with highest priority
    - Reduced load for primary copy node
  - Local reads: inconsistent data possible, but in special cases might be tolerable
- Failure of primary copy node
  - No transactions possible
  - Election of new primary copy node with suitable algorithms

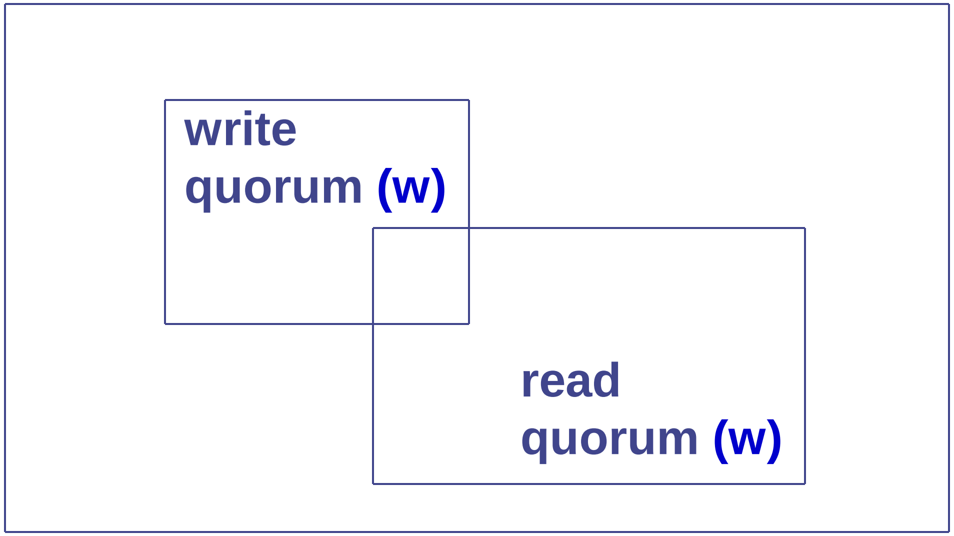
Voting

- Before a read or write access to an object, a sufficient number of votes has to be collected
- Majority votes
  - Write: Transaction hast to lock the majority of the needed objects (lock = vote)
  - Read: Majority of replicated data sets is locked for reading and one specific object is referenced
  - Guaranteed, that the referenced object will not be modified by another transaction concurrently
  - At least one replicated data set is updated
  - Assigned counter reflects the update status (version) of each replicated data set
- Advantage: Objects usable in case of crash of several nodes
- Disadvantage: Each access requires several messages to guarantee the vote majority

Weighted Voting (Quorum Consensus)

- We assign a certain weight to each replicated data set (number of votes)
- For read and write access a certain, pre-defined number of votes (read quorum or write quorum) is necessary and need to be collected
- If v votes are available, the following rules apply for read quorum r and write quorum w
  - $w>v/2$: guarantees that no object will be modified simultaneously in two transactions
  - $r+w>v$ prevents, that an object will be read and modified simultaneously. Furthermore, it is guaranteed that at least one object from the last write quorum is involved
- We use the weights to determine the costs for write / read access as well as the availability
  - The smaller r and w, the faster read and write access
  - Increased availability, because some node may crash
  - Preferred reading with increased write complexity and vice versa

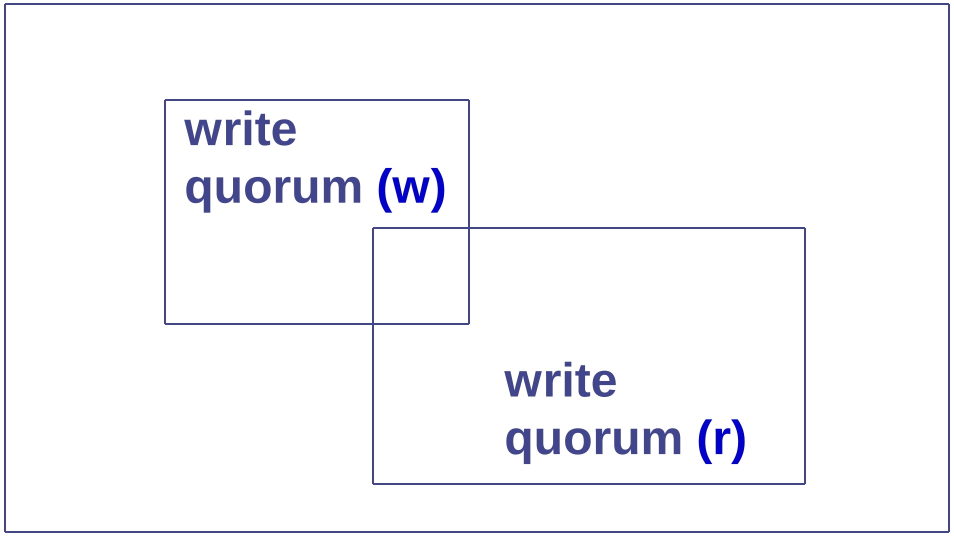
Quorum Consensus Replica Control



Set of all replicas of an item (v)

- **Read/write conflict: r + w > v**
- **An intersection between any read and any write quorum**

Quorum Consensus Replica Control



Set of all replicas of an item (v)

- **Read/write conflict: w > n/2**
- **An intersection between any two write quorums**

Example Quorum

- Object A replicated on four nodes R1 to R4

- Vote distribution <2,1,1,1>, i.e. R1 with 2 votes

- In case of r = 3 and w = 3, then at least two nodes must be involved in the transaction
  - Preferring R1 means that a faster access to data from R1 is provided
  - Access also after node failure provided. In case that R1 is still alive, then also two nodes can fail without affecting the overall system

- Read access preferred versus write access, in case of following parameters: r=2 and w=4
  - Read access locally on R1
  - For write access at least 3 nodes are necessary
  - Overall failure (no modification possible), if R1 fails

Pros and Cons

- Voting approach can emulate – suitable parameter selection assumed – all other approaches
  - Majority approach: each replicated data set gets the same weight (1 vote)
  - ROWA: same as majority, each data set with one vote. In addition r=1 and w=v=number of replicated data sets
  - Primary copy: primary copy gets one vote, all other replicated data sets have no votes and r=w=1

  ⇒Read access has to be requested from the primary copy

- Disadvantage
  - Complex definition of suitable parameters

Snapshot replication

- Replication over WANs leads to high overhead
- Slower networks increase the demands on replication mechanisms
- Lower requirements regarding update speed open space for additional techniques such as snapshots
- Definition
  - Certain database view is provided
  - Query result is exported as an DB image and provided as an object with a specific name
  - Snapshot access with the used query language, only read access allowed
- Example

```
CREATE SNAPSHOT underflow AS

SELECT CustomerNR, AccountNR, AccountBal
FROM Account WHERE AccountBal <0

REFRESH EVERY DAY
```

Snapshot properties

- In example
  - Snapshot corresponds to a copy of all accounts with negative balance
  - Snapshot refreshed in given intervals, here every day
- Advantages
  - Coupling to DB query languages allows the summary of any information, also aggregated information, dependencies, …
  - Lower load to the primary copy node, as all accesses are performed locally and without communication
  - Consistency problems avoided, as only read operations allowed
- Disadvantages
  - Lower quality than in case of “real” replication $\Rightarrow$ Data older than the real data, but in guaranteed interval
- Examples
  - Lists of spare parts in shops
  - Book catalogues
  - Phone and e-mail directories