Chapter 23: Parallel and Distributed Transaction Processing

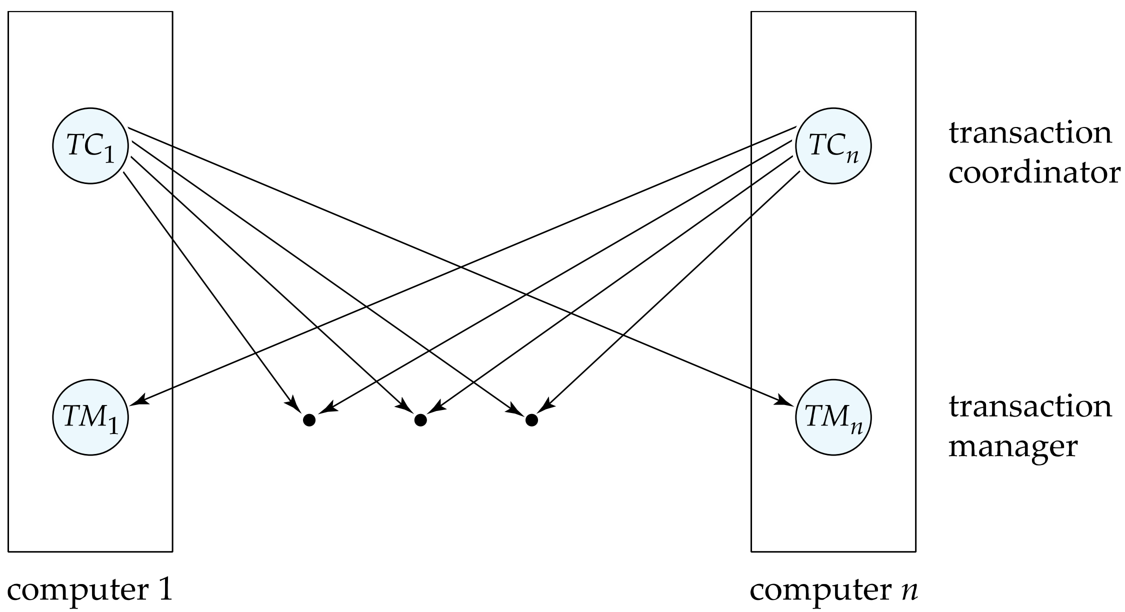
Database System Concepts, 7th Ed.

Distributed Transactions

- **Local transactions**
  - Access/update data at only one database
- **Global transactions**
  - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database

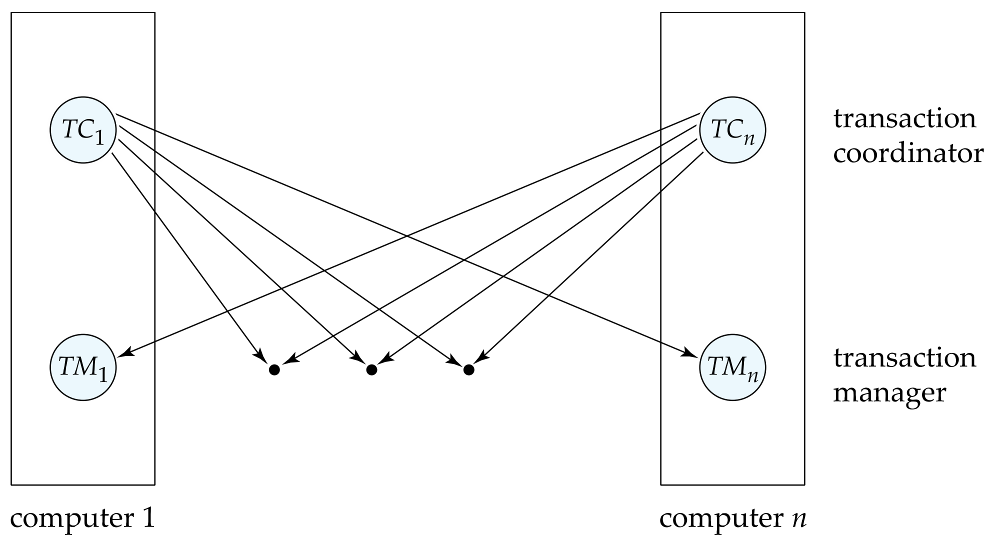
Distributed Transactions

- Transaction may access data at several sites.
  - Each site has a local **transaction manager**
  - Each site has a **transaction coordinator**
    - Global transactions submitted to any transaction coordinator



Distributed Transactions

- Each transaction coordinator is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - transaction must be committed at all sites or aborted at all sites.
- Each local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the execution and commit/abort of the transactions executing at that site.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
    - cannot have transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used
- *Three-phase commit* (3PC) protocol avoids some drawbacks of 2PC, but is more complex
- *Consensus protocols* solve a more general problem, but can be used for atomic commit
  - More on these later in the chapter
- The protocols we study all assume **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
  - Protocols that can tolerate some number of malicious sites discussed in bibliographic notes online

Two Phase Commit Protocol (2PC)

- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Protocol has two phases
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

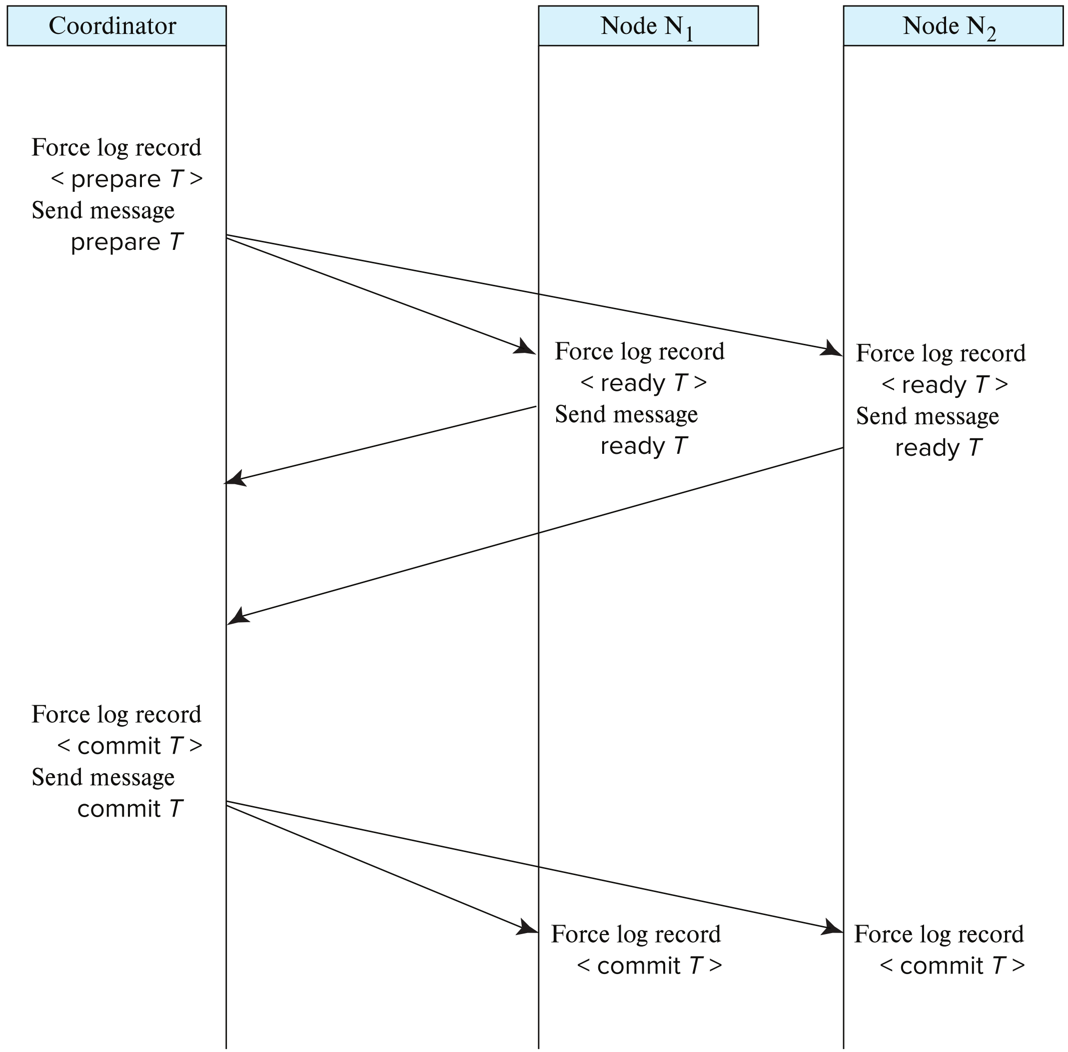
- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send **ready** *T* message to C_i

    Transaction is now in ready state at the site

Phase 2: Recording the Decision

- T can be committed of C_i received a **ready** T message from all the participating sites: otherwise T must be aborted.
- Coordinator adds a decision record, <**commit** T> or <a**bort** T>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Two-Phase Commit Protocol



Handling of Failures - Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*' s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- **In-doubt transactions** have a <**ready** *T*>, but neither a <**commit** *T*>, nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of <**ready** *T*>, write out <**ready** *T*, *L*> *L* = list of locks held by *T* when the log is written (read locks can be omitted).
  - For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Avoiding Blocking During Consensus

- Blocking problem of 2PC is a serious concern
- Idea: involve multiple nodes in decision process, so failure of a few nodes does not cause blocking as long as majority don't fail
- More general form: **distributed consensus problem**
  - A set of *n* nodes need to agree on a decision
  - Inputs to make the decision are provided to all the nodes, and then each node votes on the decision
  - The decision should be made in such a way that all nodes will "learn" the same value for the even if some nodes fail during the execution of the
    protocol, or there are network partitions.
  - Further, the distributed consensus protocol should not block, as long as a majority of the nodes participating remain alive and can communicate with each other
- Several consensus protocols, Paxos and Raft are popular
  - More later in this chapter

Using Consensus to Avoid Blocking

- After getting response from 2PC participants, coordinator can initiate distributed consensus protocol by sending its decision to a set of participants who then use consensus protocol to commit the decision
  - If coordinator fails before informing all consensus participants
    - Choose a new coordinator, which follows 2PC protocol for failed coordinator
    - If a commit/abort decision was made as long as a majority of consensus participants are accessible, decision can be found without blocking
  - If consensus process fails (e.g., split vote), restart the consensus
    - Split vote can happen if a coordinator send decision to some participants and then fails, and new coordinator send a different decision
- The **three phase commit** protocol is an extension of 3PC which avoids blocking under certain assumptions
  - Ideas are similar to distributed consensus.
- Consensus is also used to ensure consistency of replicas of a data item
  - Details later in the chapter

Distributed Transactions via Persistent Messaging

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g., transaction crossing an organizational boundary
  - Latency of waiting for commit from remote site
- Alternative models carry out transactions by sending messages
  - Code to handle messages must be carefully designed to ensure atomicity and durability properties for updates
    - Isolation cannot be guaranteed, in that intermediate stages are visible, but code must ensure no inconsistent states result due to concurrency
  - Persistent messaging systems are systems that provide transactional properties to messages
    - **Persistent messages** are guaranteed to be delivered exactly once

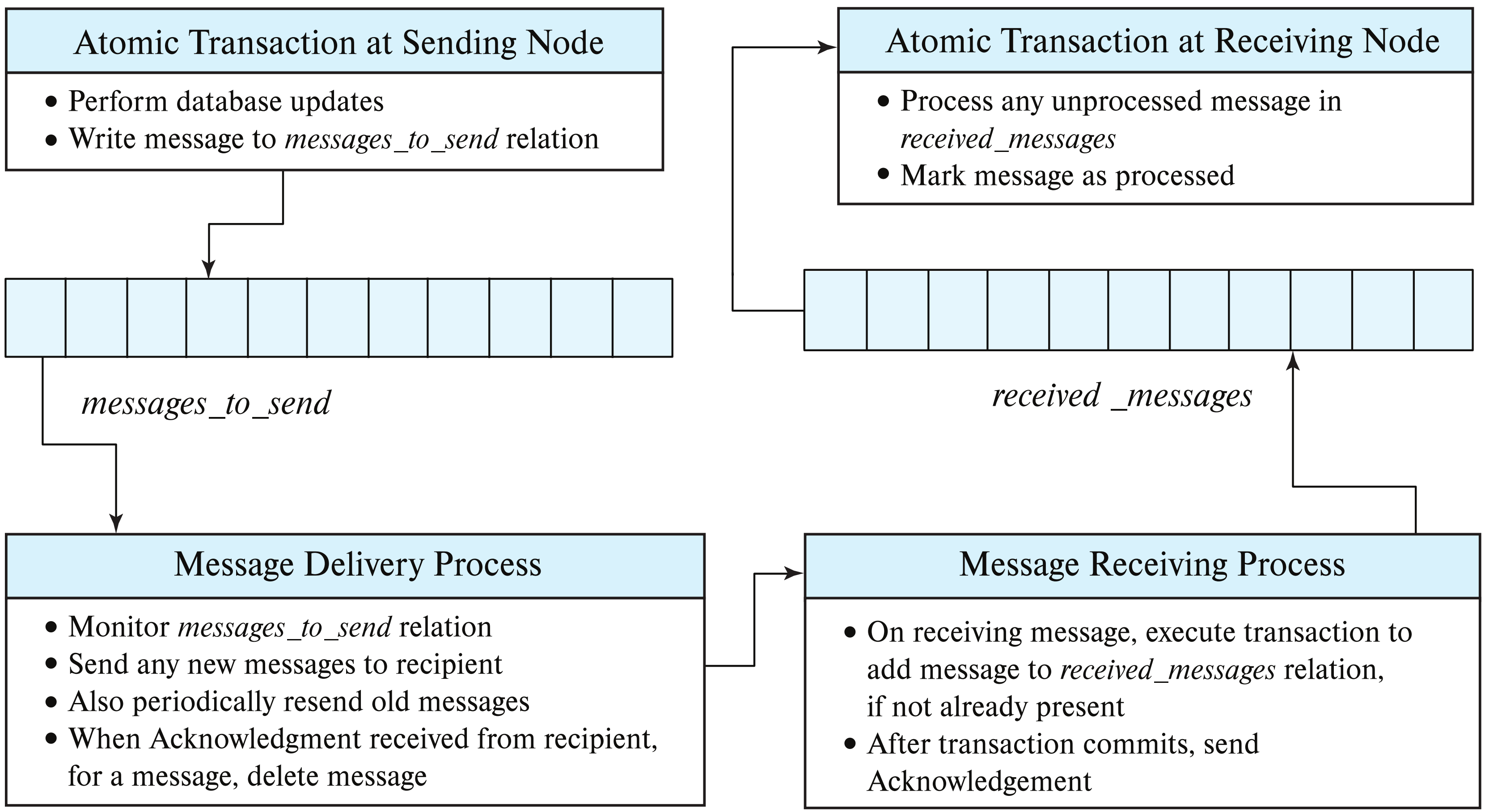
Persistent Messaging

- Example: funds transfer between two banks
  - Two phase commit would have the potential to block updates on the accounts involved in funds transfer
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue
  - once transaction sending a message is committed, message must guaranteed to be delivered
    - Guarantee as long as destination site is up and reachable, code to handle undeliverable messages must also be available
      - e.g., credit money back to source account.
  - If sending transaction aborts, message must not be sent

Error Conditions with Persistent Messaging

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - E.g., if destination account does not exist, failure message must be sent back to source site
  - When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed
      - get humans to take care of problem
- User code executing transaction processing using 2PC does not have to deal with such failures
- There are many situations where extra effort of error handling is worth the benefit of absence of blocking
  - E.g., pretty much all transactions across organizations

Persistent Messaging Implementation



Persistent Messaging (Cont.)

- Receiving site may get duplicate messages after a very long delay
  - To avoid keeping processed messages indefinitely
    - Messages are given a timestamp
    - Received messages older than some cutoff are ignored
    - Stored messages older than the cutoff can be deleted at receiving site
- **Workflows** provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps
  - E.g., when a bank receives a loan application, it may need to
    - Contact external credit-checking agencies
    - Get approvals of one or more managers

    and then respond to the loan application
  - Persistent messaging forms the underlying infrastructure for workflows in a distributed environment

Concurrency Control in Distributed Databases

Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

Single-Lock-Manager Approach

- In the **single lock-manager** approach, lock manager runs on a *single* chosen site, say S_i
  - All lock requests sent to central lock manager
- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure.

Distributed Lock Manager

- In the **distributed lock-manager** approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
  - Locking is performed separately on each site accessed by transaction
    - Every replica must be locked and updated
    - But special protocols may be used for replicas (more on this later)
- Advantage: work is distributed and can be made robust to failures
- Disadvantage:
  - Possibility of a global deadlock without local deadlock at any single site
  - Lock managers must cooperate for deadlock detection

Deadlock Handling

Consider the following two transactions and history, with item X and transaction T_1 at site 1, and item Y and transaction T_2 at site 2:

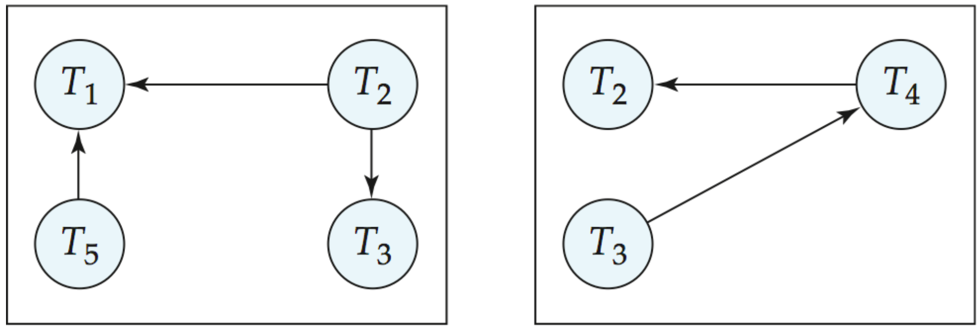
T_1: write (X) write (Y)	T_2: write (X) write (Y)
X-lock on X write (X)	
	X-lock on Y write (Y) wait for X-lock on X
Wait for X-lock on Y	

Result: deadlock which cannot be detected locally at either site

Deadlock Detection

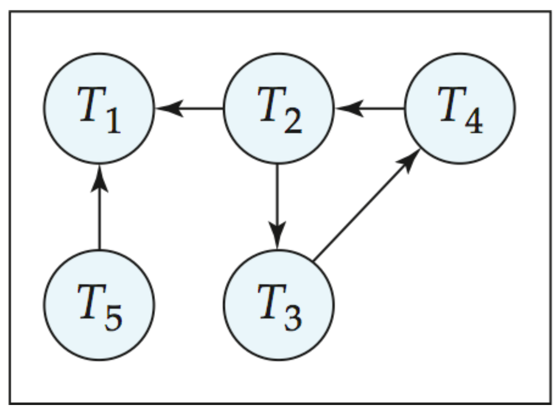
- In the **centralized deadlock-detection** approach, a global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.
  - a number of changes have occurred in a local wait-for graph.
  - the coordinator needs to invoke cycle-detection.
- If the coordinator finds a cycle, it selects a victim and notifies all sites. The sites roll back the victim transaction.

Local and Global Wait-For Graphs



site S_1 site S_2

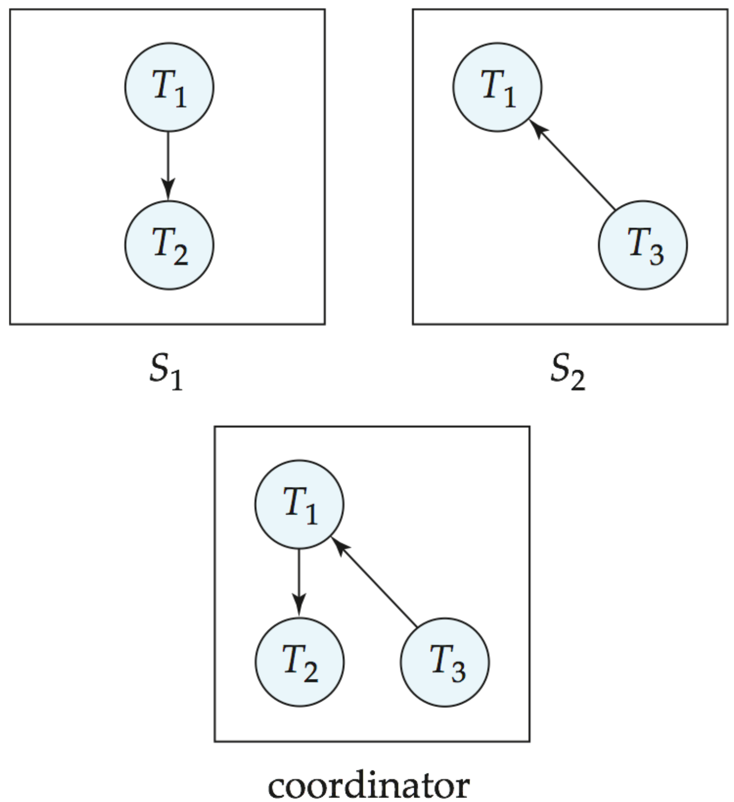
Local



Global

Example Wait-For Graph for False Cycles

Initial state:



False Cycles (Cont.)

- Suppose that starting from the state shown in figure,

  1. T_2 releases resources at S_1
     - resulting in a message remove $T_1 \rightarrow T_2$ message from the Transaction Manager at site S_1 to the coordinator)
  2. And then T_2 requests a resource held by T_3 at site S_2
     - resulting in a message insert $T_2 \rightarrow T_3$ from S_2 to the coordinator

- Suppose further that the insert message reaches before the **delete** message
  - this can happen due to network delays
- The coordinator would then find a false cycle

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

- The false cycle above never existed in reality.
- False cycles cannot occur if two-phase locking is used.

Distributed Deadlocks

- Unnecessary rollbacks may result
  - When deadlock has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.
  - Due to false cycles in the global wait-for graph; however, likelihood of false cycles is low.
- In the **distributed deadlock-detection** approach, sites exchange wait-for information and check for deadlocks
  - Expensive and not used in practice

Leases

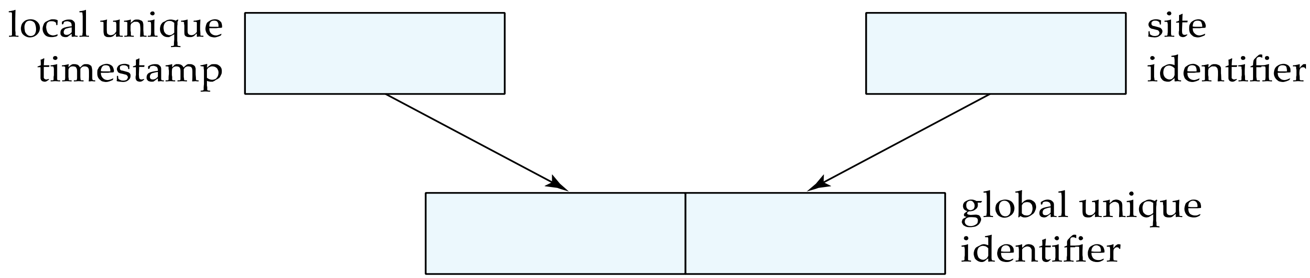
- A **lease** is a lock that is granted for a specific period of time
- If a process needs a lock even after expiry of lease, process can **renew** the lease
- But if renewal is not done before end time of lease, the lease **expires,** and lock is released
- Leases can be used to that there is only one coordinator for a protocol at any given time
  - Coordinator gets a lease and renews it periodically before expire
  - If coordinator dies, lease will not be renewed and can be acquired by backup coordinator

Leases (Cont.)

- Coordinator must check that it still has lease when performing action
  - Due to delay between check and action, must check that expiry is at least some time *t'* into the future
    - *t'* includes delay in processing and maximum network delay
    - Old messages must be ignored
- Leases depend on clock synchronization

Distributed Timestamp-Based Protocols

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a *unique* timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - Each site generates a unique local timestamp using either a logical counter or the local clock.
  - Global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier.



Distributed Timestamps

- A node with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: "disadvantages" transactions
- To fix this problem
  - Keep clocks synchronized using network time protocol
  - Or, define within each node N_i a **logical clock** (LC_i), which generates the unique local timestamp
    - Require that N_i advance its logical clock whenever a request is received from a transaction Ti with timestamp $< x,y >$ and x is greater that the current value of LC_i.
    - In this case, site N_i advances its logical clock to the value $x + 1$

Distributed Timestamp Ordering

- Centralized TSO and multiversion TSO easily extended to distributed setting
  - Transactions use a globally unique timestamp
  - Each site that performs a read or write performs same checks as in centralized case
- Clocks at sites should be synchronized
  - Otherwise a transaction initiated at a site with a slower clock may get restarted repeatedly.

Distributed Validation

- The validation protocol used in centralized systems can be extended to distributed systems
- Start/validation/finish timestamp for a transaction T_i may be issued by any of the participating nodes
  - Must ensure StartTS(T_i) < TS(T_i) < FinishTS(T_i)
- Validation for T_i is done at each node that performed read/write
  - Validation checks for transaction T_i are same as in centralized case
    - Ensure that no transaction that committed after T_i started has updated any data item read by T_i.
  - A key difference from centralized case is that may T_i start validation after a transaction with a higher validation timestamp has already finished validation
    - In that case T_i is rolled back

Distributed Validation (Cont.)

- Two-phase commit (2PC) needed to ensure atomic commit across sites
  - Transaction is validated, then enters prepared state
  - Writes can be performed (and transaction finishes) only after 2PC makes a commit decision
  - If transaction T_i is in prepared state, and another transaction T_k reads old value of data item written by T_i, T_k will fail if T_i commits
    - Can make the read by T_k wait, or create a commit dependency for T_k on T_i.

Distributed Validation (Cont.)

- Distributed validation is not widely used, but optimistic concurrency control without read-validation is widely used in distributed settings
  - Version numbers are stored with data items
  - Writes performed at commit time ensure that the version number of a data item is same as when data item was read
  - Hbase supports atomic checkAndPut() as well as checkAndMutate() operations; see book for details

Replication

Replication

- **High availability** is a key goal in a distributed database
  - **Robustness**: the ability to continue function despite failures
- Replication is key to robustness
- Replication decisions can be made at level of data items, or at the level of partitions

Consistency of Replicas

- Consistency of replicas
  - Ideally: all replicas should have the same value ➔ updates performed at all replicas
    - But what if a replica is not available (disconnected, or failed)?
  - Suffices if reads get correct value, even if some replica is out of date
  - Above idea formalized by **linearizability**: given a set of read and write operations on a (replicated) data item
    - There must be a linear ordering of operations such that each read sees the value written by the most recent preceding write
    - If o_1 finishes before o_2 begins (based on external time), then o_1 must precede o_2 in the linear order
- Note that linearizability only addresses a single (replicated) data item; serializability is orthogonal

Consistency of Replicas

- Cannot differentiate node failure from network partition in general
  - Backup coordinator should takeover if primary has failed
  - Use multiple independent links, so single link failure does not result in partition, but it is possible all links have failed
- Protocols that require all copies to be updated are not robust to failure
- Will see techniques that can allow continued processing during failures, whether node failure or network partition
  - Key idea: decisions made based on successfully writing/reading majority
- Alternative: **asynchronous replication:** commit after performing update on a *primary copy* of the data item, and update replicas *asynchronously*
  - Lower overheads, but risk of reading stale data, or lost updates on primary failure

Concurrency Control With Replicas

- Focus here on concurrency control with locking
  - Failures addressed later
  - Ideas described here can be extended to other protocols
- **Primary copy**
  - one replica is chosen as primary copy for each data item
    - Node containing primary replica is called **primary node**
  - concurrency control decisions made at the primary copy only
- Benefit: Low overhead
- Drawback: primary copy failure results in loss of lock information and non-availability of data item, even if other replicas are available
  - Extensions to allow backup server to take over possible, but vulnerable to problems on network partition

Concurrency Control With Replicas (Cont.)

- **Majority protocol**:
  - Transaction requests locks at multiple/all replicas
  - Lock is successfully acquired on the data item only if lock obtained at a majority of replicas
- Benefit: resilient to node failures and node failures
  - Processing can continue as long as at least a majority of replicas are accessible
- Overheads
  - Higher cost due to multiple messages
  - Possibility of deadlock even when locking single item
    - How can you avoid such deadlocks?

Concurrency Control With Replicas (Cont.)

- **Biased protocol**
  - Shared lock can be obtained on any replica
    - Reduces overhead on reads
  - Exclusive lock must be obtained on *all* replicas
    - Blocking if any replica is unavailable

Quorum Consensus Protocol

Quorum consensus protocol for locking

- Each site is assigned a weight; let S be the total of all site weights
- Choose two values **read quorum** Q_R and **write quorum** Q_W
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
- Each read must lock enough replicas that the sum of the site weights is ≥ Q_r
- Each write must lock enough replicas that the sum of the site weights is ≥ Q_w
- Can choose Q_r and Q_w to tune relative overheads on reads and writes
  - Suitable choices result in majority and biased protocols.
    - What are they?

Dealing with Failures

- Read one write all copies protocol assumes all copies are available
  - Will block if any site is not available
- *Read one write all available* (ignoring failed sites) is attractive, but incorrect
  - Failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an incorrect value
  - With network partitioning, sites in each partition may update same item concurrently
    - believing sites in other partitions have all failed

Handling Failures with Majority Protocol

- The majority protocol with version numbers
  - Each replica of each item has a **version number**
  - Locking is done using majority protocol, as before, and version numbers are returned along with lock allocation
  - Read operations read the value from the replica with largest version number
  - Write operations
    - Find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
- Read operations that find out-of-date replicas may optionally write the latest value and version number to replicas with lower version numbers
  - no need to obtain locks on all replicas for this task

Handling Failures with Majority Protocol

- Atomic commit of updated replicas must be ensured using either
  - *2 phase commit on all locked replicas*, or
  - distributed consensus protocol such as Paxos (more on this later)
- Failure of nodes during 2PC can be ignored as long as majority of sites enter prepared state
- Failure of coordinator can cause blocking
  - Consensus protocols can avoid blocking

Handling Failures with Majority Protocol

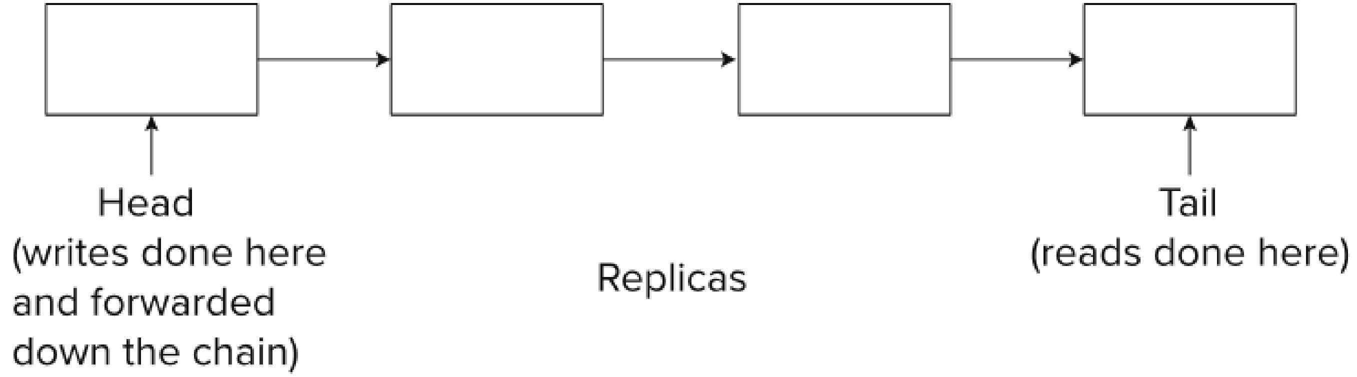
- Benefits of majority protocol
  - Failures (network and site) do not affect consistency
    - Reads are guaranteed to see latest successfully written version of a data item
  - Protocol can proceed as long as
    - Sites available at commit time contain a majority of replicas of any updated data items
    - During reads a majority of replicas are available to find version numbers
  - No need for any special reintegration protocol: nothing needs to be done if nodes fail and subsequently recover
- Drawback of majority protocol
  - Higher overhead, especially for reads

Reducing Read Cost

- Quorum consensus can be used to reduce read cost
  - But at increased risk of blocking of writes due to failures
- Use primary copy scheme:
  - perform all updates at primary copy
  - reads only need to be done at primary copy
  - But what if primary copy fails
    - Need to ensure new primary copy is chosen
      - Leases can ensure there is only 1 primary copy at a time
    - New primary copy needs to have latest committed version of data item
      - Can use consensus protocol to avoid blocking

Reducing Read Cost

- **Chain replication**:
  - Variant of primary copy scheme
  - Replicas are organized into a chain
  - Writes are done at head of chain, and passed on to subsequent replicas
  - Reads performed at tail
    - Ensures that read will get only fully replicated copy
  - Any node failure requires reconfiguration of chain



Reconfiguration and Reintegration

- To be robust, a distributed system must either
  - Follow protocols like the majority protocol that work in spite of failures or
  - Use other protocols like primary copy protocol, but
    - Detect failures (failed/non-reachable nodes)
    - Reconfigure the system to remove failed nodes, and assign their tasks to other sites, so computation may continue
    - Recover/reintegrate nodes a node or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure

Reconfiguration

- Reconfiguration:
  - Abort all transactions that were active at a failed site
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g., name server, concurrency coordinator, global deadlock detector

Reconfiguration (Cont.)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two or more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- Solution: majority based approach

Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to latest version, then release locks
    - Can do this for one partition at a time

Comparison with Remote Backup

- Remote backup systems (Section 19.7) are also designed to provide high availability
- Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and only log records shipped
  - No need for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can provide higher availability by having multiple (> 2) replicas and using the majority protocol
  - Also avoid failure detection and switchover time associated with remote backup systems

Extended Concurrency Control Protocols

Multiversion 2PL and Globally Consistent Timestamps

- Recall multiversion 2PL protocol:
  - Read only transactions get timestamp at start
    - T_i reads latest committed version of data items with TS < startTS(T_i)
  - Update transactions perform 2PL, and also get timestamp at commit
  - Serialization order defined by timestamp
- Question: can we use MV2PL in a distributed system
- Answer: yes, *but a lot of conditions apply*
  - If commits are serialized at central coordinator, timestamps can be given based on counter
  - But if commits are distributed, how to give timestamps in a consistent manner?
    - Clocks may not be in sync, later commit may get lower timestamp
    - Out of order timestamp issual may result in serialization order not matching timestamp order

Multiversion 2PL and Globally Consistent Timestamps

- Centralized coordinator to assign consistent timestamps
  - Can be done, but becomes bottleneck
- Google Spanner ideas:
  - In an ideal world, clocks are synchronized, and can be used to assign commit timestamps to transactions
  - In reality, clocks are out of sync
  - Key ideas
    - Use atomic clocks, GPS etc to periodically get precise time
    - Derive bound on how out-of-sync a node's clock *t'* can be w.r.t. to actual time *t*
      - $t' - \varepsilon \leq t \leq t' + \varepsilon$
    - Introduce **commit wait**: hold locks for some period and assign timestamp *ts* such that locks were definitely held at actual time *ts*

Multiversion 2PL and Globally Consistent Timestamps

- Google Spanner ideas (cont):
  - If version of *x* has timestamp ts, then *x* definitely had that value at time *ts*
  - System can generate transactionally consistent snapshop as of actual time *ts* (**external consistency**)
  - Commit processing can still take time
    - With 2PC status of transaction may not be known for a while
      - Reads may have to wait till status of transaction is known
    - But read-only transactions can use a snapshot timestamp *ts* such that all transactions before that timestamp have been committed or aborted
      - Read can proceed without waiting
      - But perhaps with older versions of data

Other Concurrency Control Techniques

- **Distributed snapshot isolation**
  - Running Snapshot Isolation separately on each node may result in different serialization orders at different nodes
  - Extensions to SI to ensure consistent ordering have been proposed
- **Concurrency control in federated databases**
  - Local transactions
  - Global transactions
  - Local serializability may not guarantee global serializability unless all nodes use 2PL
  - Use idea of **tickets** to create conflicts that will ensure serializability

Replication With Weak Degrees of Consistency

Consistency

- Recall: Consistency in Databases (ACID):
  - Database has a set of integrity constraints
  - A consistent database state is one where all integrity constraints are satisfied
  - Each transaction run individually on a consistent database state must leave the database in a consistent state
- Recall: Consistency in distributed systems with replication
  - **Strong consistency**: a schedule with read and write operations on an object should give results and final state equivalent to some schedule on a single copy of the object, with order of operations from a single site preserved
  - Weak consistency (several forms)

Availability

- Traditionally, availability of centralized server
- For distributed systems, availability of system to process requests
  - For large system, at almost any point in time there's a good chance that
    - a node is down or even
    - Network partitioning
- Availability: ability to continue operations despite node and network failures.

CAP "Theorem"

- Three properties of a system
  - Consistency
    - an execution of a set of operations (reads and writes) on replicated data is said to be **consistent** if its result is the same as if the operations were executed on a single node, in a sequential order that is consistent with the ordering of operations issued by each process (transaction)
  - Availability (system can run even if parts have failed)
    - Via replication
  - Partitions (network can break into two or more parts, each with active systems that can't talk to other parts)
- Brewer's CAP "Theorem": You can have at most two of these three properties for any system

CAP "Theorem" (Cont.)

- Very large systems will partition at some point
- Choose one of consistency or availability
  - Traditional database choose consistency
  - Many web applications choose availability
    - Except for specific parts such as order processing
- Latency is another factor
  - Many applications choose to serve potentially stale data to reduce latency

Replication with Weak Consistency

- Many systems support replication of data with weak degrees of consistency (I.e., without a guarantee of serializabiliy)
  - In quorum consistency notation: allow Q_R and Q_W to be set such that $Q_R + Q_W \leq S$ or $2^*Q_W \leq S$
    - E.g., can be set in MongoDB and Cassandra
  - Usually only when not enough sites are available to ensure quorum
    - But sometimes to allow fast local reads
  - Tradeoff of consistency versus availability or latency
- Key issues:
  - Reads may get old versions
  - Some replicas may not get updated
  - **Different updates may be applied to different replicas**
    - Question: how to detect, and how to resolve
    - Will see in detail later

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - **Soft state**: copies of a data item may be inconsistent
  - **Eventually Consistent :** Copies may be allowed to become inconsistent, but (once partitioning is resolved) eventually all copies become consistent with each other
    - at some later time, if there are no more updates to that data item

Asynchronous Replication

- With **asynchronous replication,** updates are done at the primary node (also known as master node), and then propagated to replicas
  - Transaction can commit once update is done at primary node
  - Propagation after commit is also referred to as **lazy propagation**
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency
- Replicas may not be up-to-date
  - Transactions that can live with old data can read from replicas
  - Snapshot reads at a point in time can also be served from replicas that are sufficiently up-to-date
    - E.g., in Google Spanner
      - each replica maintains a timestamp t_{safe} such that all updates with timestamp $t < t_{safe}$ have already been received
      - Reads of a transaction can be satisfied by a replicate if transaction timestamp $t < t_{safe}$

Asynchronous Replication

- **Master-slave replication**: updates performed only at master, and asynchronously propagated to replicas
  - replicas can only satisfy reads
- **Multimaster replication** (or **update-anywhere replication**): updates can be performed at any replica, and propagated synchronously or asynchronously to other replicas
- Updates must be propagated to replicas even if there are failures, and processed in the correct order at the replicas
  - Persistent messaging systems can be used for this, with minor extensions to ensure in-order delivery
  - Publish-subscribe systems such as Kafka can also be used for this task
    - More flexible, support parallelism by having multiple topics and partitions of topics
  - Fault-tolerance is important
    - Can use log-replication with two-safe protocol (Section 19.7)

Asynchronous View Maintenance

- Materialized views can be useful in distributed systems
  - Secondary indices can be considered as a simple form of materialized view in a parallel database
    - E.g., given relation *r(A,B,C)* where *A* is the primary key on which *r* is partitioned, a secondary index on *B* is simply a projection of *r* on *(B,A),* partitioned on *B*.
  - Materialized aggregate views are also very useful in many contexts
- Performing view maintenance as part of the original transaction may not be possible (if the underlying database does not support distributed transactions), or may be expensive
- Asynchronous maintenance of materialized views, after the original transaction commits, is a good option in such a case
  - Applications using the view/index must then be aware that it may be a little out-of-date

Requirements for Asynchronous View Maintenance

Requirements:

1. Updates must be delivered and processed exactly once despite failures
2. Derived data (such as materialized views/indices) must be updated in such a way that it will be consistent with the underlying data
   - Formalized as **eventual consistency**: if there are no updates for a while, eventually the derived data will be consistent with the underlying data
3. Queries should get a transactionally consistent view of derived data
   - Potentially a problem with long queries that span multiple nodes
   - E.g., without transactional consistency, a scan of relation may miss some older updates and see some later updates
   - Not supported by many systems, supported via snapshots in some systems

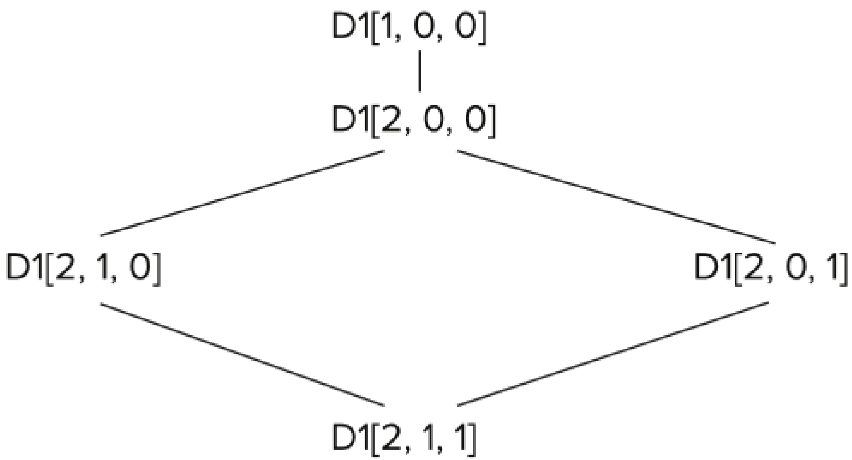
Detecting Inconsistency

- Data items are versioned
- Each update creates a new immutable version
- In absence of failure, there is a single latest version
- But with failures and weak consistency, versions can diverge
  - Different nodes may perform different updates on same data
  - Need to detect, and fix such situations
- Key idea: **vector-vector** identifies each data version
  - Set of (node, counter) pairs
    - E.g., with two nodes N1 and N2, ([N1,2],[N2,1])
    - Represented as a vector [2, 1]
  - An update to a data item at a node increments the counter for that node
  - Define a partial order across versions

Vector Vectors

- Examples of vector vectors
  - ([Sx,1]): data item created by site Sx
  - ([Sx,2]): data item created by site Sx, and later updated
  - ([Sx,2],[Sy,1]): data item updated twice by site Sx and once by Sy
    - Update by a site Sx increments the counter for Sx, but leaves counters from other sites unchanged
  - ([Sx,4],[Sy,1]) **newer than** ([Sx,3],[Sy,1])
  - But ([Sx,2],[Sy,1]) **incomparable with** ([Sx,1],[Sy,2])
  - Read operation may find **incomparable versions**
    - Such versions indicate inconsistent concurrent updates
    - All such versions returned by read operation
    - Up to application to reconcile multiple versions

Example of Vector Clock in action



- Item D1 created by Node N1
- D1 updated by Node N1
- D1 concurrently updated by node N2 and N3 (usually due to network partitioning)
- Subsequent read from N2 and N3 returns two incomparable versions
- Application merges versions and writes new version

Extensions for Detecting Inconsistency

- Two replicas may diverge, and divergence is not detected until the replicas is read
  - To detect divergence early, one approach is to scan all replicas of all items periodically
    - But requires a lot of network, CPU and I/O load
    - Alternative approach based on Merkle trees covered shortly

How to Reconcile Inconsistent Versions?

- Reconciliation is application specific
  - E.g., two sites concurrent insert items to cart
    - Merge adds both items to the final cart state
  - E.g., S1 adds item A, S2 deletes item B
    - Merge adds item A, but deleted item B resurfaces
    - Cannot distinguish S2 deletes B from S1 add B
    - Problem: operations are inferred from states of divergent versions
  - Better alternative:
    - Keep track of history of operations
    - Merge operation histories

Order Independent Operations

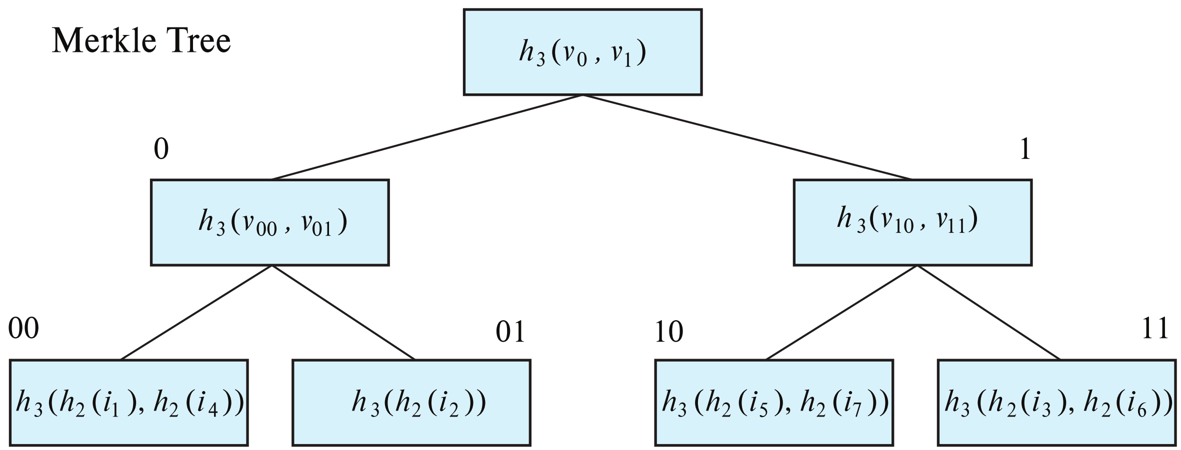
- Basic idea: updates are performed as logical operations
  - Data store is aware of the set of operations that can be carried out
  - Operation performed at place where data (replica) is stored
- If result of a sequence of operations is independent of the operation ordering
  - Independent update operations can be merged in different orders at different replicas, but will lead to same result
  - Eventual consistency can be ensured relatively easily

Detecting Differences Using Merkle Trees

- **Merkle Tree**: A data structure that can
  - Efficiently sign contents of a tree
  - Efficiently find differences (if any) between two replicas
- Example

Hash values of data items
$h_1(i_1)$=00
$h_1(i_2)$=01
$h_1(i_3)$=11
$h_1(i_4)$=00
$h_1(i_5)$=10
$h_1(i_6)$=11
$h_1(i_7)$=10

Merkle Tree



Node identifier shown above node, and has value shown inside node,
v_i denotes stored hash value in node i

Detecting Differences Using Merkle Trees (Cont.)

- Overall cost of finding differences with Merkle tree
  - $O(m \log_2 N)$ with N data items and m differences using binary tree
  - Each operation requires communication between the two trees (nodes)
  - Use wider trees to reduce height/cost
    - Cost is $O(m \log_K N)$ if each node has K children instead of 2 children
    - Particularly important due to high network latency
- Merkle trees originally used for **verification of contents** of a collection
  - Include digital signature at root in this case.

Weak Consistency Models for Applications

- **Read-your-writes**
  - if a process has performed a write, a subsequent read will reflect the earlier write operation
- **Session consistency**
  - Read-your-writes in the context of a session, where application connects to storage system
- **Monotonic consistency**
  - For reads: later reads never return older version than earlier reads
  - For writes: serializes writes by a single process
    - Minimum requirement
- **Sticky sessions**: all operations from a session on a data item go to the same node
- Can be implemented by specifying a version vector in get() operations
  - Result of get guaranteed to be at least as new as specified version vector

Coordinator Selection

Coordinator Selection

- **Backup coordinators**
  - Backup coordinator maintains enough information locally to assume the role of coordinator if the actual coordinator fails
    - executes the same algorithms and maintains the same internal state information as the actual coordinator
  - allows fast recovery from coordinator failure but involves overhead during normal processing.
- Backup coordinator approach vulnerable to two-site failure
  - Failure of coordinator and backup leads to non-availability
  - Key question: how to choose a new coordinator from a set of candidates
    - Choice done by a master: vulnerable to master failure
    - Election algorithms are key

Coordinator Selection

- **Coordinator selection using a fault-tolerant lock manager**
  - Coordinator gets a lease on a coordinator lock, and renews the lease as long as it is alive
  - If coordinator dies or gets disconnected, lease is losk
  - Other nodes can detect coordinator failure using heart-beat messages
  - Nodes request coordinator lock lease from lock manager; only 1 node gets the lease, and becomes new coordinator
- Fault-tolerant coordination services such as **ZooKeeper, Chubby**
  - Provide fault-tolerant lock management services
  - And are widely used for coordinator section
  - Store (small amounts) of data in files
  - Create and delete files
    - Which can be used as locks/leases
      - Coordinator releases lease if it is not renewed in time
  - Can watch for changes on a file
  - But these services themselves need a coordinator……..

Election of Coordinator

- **Election algorithms**
  - Used to elect a new coordinator in case of failures
    - Heartbeat messages used to detect failure of coordinator
  - One-time election protocol
    - **Proposers**: Nodes that propose themselves as coordinator and send vote requests to other nodes
    - **Acceptors**: Nodes that can vote for candidate proposers
    - **Learners**: Nodes that ask acceptors who they voted for, to find winner
      - A node can perform all above roles
    - Problems with this protocol
      - What if no one won the election due to split vote?
      - If election is rerun, need to identify which election a request is for
  - General approach
    - Candidates make a proposal with a **term number**
      - Term number is 1 more than term number of previous election known to candidate

Election of Coordinator

- **Election algorithms (Cont.)**
      - **Stale messages** corresponding to old terms can be ignored
    - If a candidate wins majority vote it becomes coordinator
    - Otherwise election is rerun with term number incremented
    - Minimizing chances of split elections:
      - Use node IDs to decide who to vote for
        - e.g., max node ID (**Bully algorithm**)
        - Candidates withdraw if they find another candidate with higher ID
      - **Randomized retry**: candidates wait for random time intervals before retrying
        - High probability that only one node is asking to be elected at a time
  - Special case of distributed consensus

Issues with Multiple Coordinators

- Coordinator may get disconnected, and new coordinator elected, without old coordinator ever knowing about the election
  - Multiple nodes may thus believe they are coordinators
    - Called a **split-brain** situation
- Solutions:
  - **Term numbers** can be used to identify coordinator
    - Majority of node will know of latest coordinator term since they voted for it
    - Messages with old term number (**stale messages**) can be ignored
  - **Leases** can be used to ensure only one coordinator at a time
    - Delayed messages may still be received from old coordinator
      - Term numbers can be used to ignore such delayed messages

Distributed Consensus

Distributed Consensus

- Motivating example: commit decision in two-phase commit (2PC)
  - Decision made by coordinator alone: vulnerable to blocking problem
    - If coordinator fails/gets disconnected at certain key points, rest of system does not know if the decision was to commit/abort, and must block till coordinator recovers
  - Multiple nodes must participate in decision process to ensure fault tolerance
    - Although initial proposal for decision may be made by a single node
  - Goal: A decision making protocol that is non-blocking as long as a majority of participating nodes are up and reachable
- 2PC is a special case of a more general class of decision problems that must be made by a collection of nodes in a fault-tolerant, non-blocking manner

Distributed Consensus

- **Distributed consensus problem**: A set of *n* nodes (called **participants**) need to agree on a decision by executing a protocol such that
  - All participants “learn” the same value for the decision
    - even if some nodes fail during the execution of the protocol, messages are lost, or there are network partitions
  - The protocol should not block, and must terminate, as long as some majority of nodes are alive and can communicate with each other

Distributed Consensus: Overview

- An real system needs to make a series of decisions: **multiple consensus protocol**
- Problem can be abstracted as adding a record to a log
  - Each node has a copy of the log, and log records are appended at each node
  - Potential for conflicts between the nodes on what record is appended at what point in the log
  - The multiple consensus protocol must ensure that the log is uniquely defined
    - Copies of the log may temporarily differ, but must be made consistent subsequently
      - May require deleting parts of the log on a node
    - Actions can be taken on a log record only after consensus has been reached for that position in the log

Distributed Consensus: Overview (Cont.)

- Several protocols proposed
  - We outline key ideas behind **Paxos** and **Raft**
  - The **Zab** protocol used in ZooKeeper is another widely used consensus protocol
- Key idea: voting to make a decision
  - A particular decision succeeds only if a majority of the participating nodes have voted for it
    - Prevents more than one decision being chosen in a round
    - If majority of nodes are up and agree on a decision voting will not block
      - But devil in the details!

Paxos Consensus Protocol

- Assume a collection of processes that can **propose** values
  - Different processes may propose different values
  - Proposals are sent to **acceptors** which collectively choose from among the proposals
- A single execution of a **distributed consensus protocol** must ensure that:
  - At most a single value from amongst those proposed is chosen collectively by the acceptors
  - If a value has been chosen, then **learner** processes should be able to learn the chosen value
    - In case no value is chosen (split-voting), protocol reexecutes
  - Protocol should not block, and must terminate, as long as some majority of the nodes participating remain alive and can communicate with each other

Paxos Consensus Protocol: Overview

- Key idea: Consensus is reached when a *majority* of acceptors have accepted a particular proposal
  - Learner finds what value (if any) was accepted by a majority of acceptors
- If a majority vote for a particular value, all is fine, BUT
  - Vote may get split, requiring further rounds to reach a majority
  - Worse, even if a majority accept a value (and even if a learner learns of the majority), some of the acceptors (and the learner) may die or get disconnected
    - Remaining nodes may not be a majority
    - If this is treated as failure and another round is run, a different proposal may get accepted, with different learners learning different values!
  - Once acceptor has voted for a particular proposal in a round, it cannot change its mind for that round
    - Decision must be logged to ensure no change in decision if acceptor dies and comes back up

Paxos: Overview

- To deal with split vote Paxos uses a coordinator
  - Proposals serialized through coordinator, so only one value is typically proposed in a round
  - Paxos works correctly (but less efficiently) even if there are multiple coordinators
  - Coordinator can be elected
- Different values getting majorities in different nodes is a more serious problem. To solve it further rounds should give same result.
- Key idea:
  - Each proposal in Paxos has a unique number
  - Acceptors accept highest numbered proposal received in a round
  - Proposers will not create new proposals with a different number
  - Two phase protocol

Paxos Made Simple

- Phase 1
  - **Phase 1a**: A proposer selects a proposal number *n* and sends a prepare request with number *n* to a majority of acceptors
    - Number has to be chosen in some unique way
  - **Phase 1b**: If an acceptor receives a prepare request with number *n*
    - If *n* is less than that of any prepare request to which it has already responded then it ignores the request
    - Else it remembers *n* and responds to the request
      - If it has already accepted a proposal with number *m* and value *v,* it sends (*m, v*) with the response
      - Otherwise it indicates to the proposer that it has not accepted any value earlier
      - *NOTE: responding is NOT the same as accepting*

Paxos Made Simple

- Phase 2
  - **Phase 2a:** *Proposer Algorithm***:** If the proposer receives a response to its prepare requests (numbered *n*) from a majority of acceptors
    - then it sends an *accept* request to each of those acceptors for a proposal numbered *n* with a value *v*, where v is
      - the value selected by the proposer if none of the acceptors indicated it had already accepted a value.
      - Otherwise *v* is the value of the highest-numbered proposal among the responses
        - i.e., proposer backsoff from its own proposal and votes for highest numbered proposal already accepted by at least one acceptor
    - If proposer does not hear from a majority it takes no further action in this round

Paxos Made Simple

- Phase 2
  - **Phase 2b:** *Acceptor Algorithm***:** If an acceptor receives an accept request for a proposal numbered *n*,
    - If it has earlier responded to a prepare message with number *n1* > *n* it ignores the message
    - Otherwise it *accepts* the proposed value *v* with number *n*.
      - Note: acceptor may accept different values with increasing *n*

Paxos Details

- Key idea: if a majority of acceptors accept a value *v* (with number *n*), then even if there are further proposals with number *n1* > *n*, the value proposed will be value *v*
  - *Why?:*
    - A value can be accepted with number *n* only if a majority of nodes (say *P*) respond to a prepare message with number *n*
    - Any subsequent majority (say *A*) will have nodes in common with the first majority *P*, and at least one of those nodes would have responded with value *v* and number *n*
      - If a higher numbered proposal *p* was accepted earlier by even one node majority would have responded to *p*, and will ignore *n*
    - Further rounds will use this value *v* (since highest accepted value is used in Phase 2a)

Paxos Details (Cont.)

- At end of phase 2, it is possible that there is no majority have agreed on a value
  - Learners that believe majority was not reached can initiate a fresh proposal
  - If majority had actually been reached, same value will be chosen again
- Many more details under cover
- Above is for a single decision. **Multi-Paxos**: extension which deals with a series of decisions
- Many variants of Paxos optimized for different scenarios

The Raft Consensus Protocol

The Log-Based Consensus Protocols

- Fault-tolerant log, to which records are appended
- Each participating node maintains a replica of a log
- Key goal: keep the log replicas in sync
  - Logical view of atomically appending records to all copies of the log
  - Can't actually be done atomically; logs may diverge
- Consensus protocols must ensure
  - Even if a log replica is temporarily inconsistent with another, it will be brought back to sync
    - May require log deletion and replacement
  - A log entry will not be treated as committed until the algorithm guarantees that it will never be deleted

The Raft Consensus Algorithm

- Raft is based on having a coordinator, called a **leader**
  - Essential in Raft, unlike Paxos, where coordinator is an optimization
- Other nodes are called **followers**
- Leaders may die and get replaced
  - Time divided into **terms,** each term has a unique leader
  - Terms have increasing numbers

The Raft Leader Election

- Leaders are elected using randomized retry algorithm outlined in Section 23.7.2
  - Recall that algorithm already uses notion of term
  - Voting is done for a specific term
    - Can change in another term
  - Nodes track *currentTerm* based on messages received
- Leader *N1* may get disconnected and get reconnected after new leader *N2* is elected
  - *N1* may not even know it was disconnected and may continue leader actions

Example of Raft Logs

- Number in each entry indicates term
- Example log entries are assignments to variables

log index	1	2	3	4	5	6	7
leader	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	3 $z \leftarrow 4$
follower 1	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$		
follower 2	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	3 $z \leftarrow 4$
follower 3	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$				
follower 4	1 $x \leftarrow 2$	1 $z \leftarrow 2$	1 $x \leftarrow 3$	2 $x \leftarrow 4$	3 $x \leftarrow 1$	3 $y \leftarrow 6$	

committed entries

Raft Log Replication

- Appending a log entry done by sending log append request to leader
- Leader sends *AppendEntries* request to all followers, with these parameters
  - *term*
  - *previousLogEntryPosition*
  - *previousLogEntryTerm*
  - *logEntries*: array allowing multiple log records to be appended
  - *leaderCommitIndex*: an index such that all log records before the index are committed
- Followers carry out checks and respond (next slide)
- If majority of nodes respond with true, leader can report successful log append to initiating node
  - Otherwise more work is needed, explained later

Raft AppendEntries Procedure

- Follower that receives *AppendEntries* message does the following
  1. If term in message is less than followers *currentTerm*, Return false
  2. If log does not have an entry at *previousLogEntryPosition* with term matching *previousLogEntryTerm*, Return false
  3. If entry at *previousLogEntryPosition* is different from first log record in *AppendEntries* message, delete existing entry and all subsequent entries in log
  4. Any log records in *logEntries* that are not already in log are appended to log
  5. Follower maintains local *commitIndex*
     - if *leaderCommitIndex* > *commitIndex*, set *commitIndex*=min(*leaderCommitIndex*, last log entry index)
  6. Return true

Raft AppendEntries Procedure (Cont.)

- If leader *N1* receives a false message from follower with a higher currentTerm, *N1* realizes it is no longer a leader and becomes a follower
- Different followers may have different log states
- If leader receives false from a node, log in that node is out of date and needs updating
  - Leader retries AppendEntries for that node, starting from an earlier point in its own log
  - May get false several times, until it goes far enough back in log to find a matching log entry
- Key remaining issue: if a leader dies, and another one takes over, the log must be brought to consistent state
  - New leader may have an older log

Raft Leader Replacement

- Raft protocol ensures any node elected as leader has all committed log entries
  - Candidate must send information about its own log state when seeking votes
  - Node votes for candidate only if candidates log state is at least as up-to-date as its own (we omit details)
  - Since majority have voted for new leader, any committed log entry will be in new leaders log
- Raft forces all other nodes to replicate leaders log
  - Log records at new leader may get committed when log gets replicated
  - Leader *cannot* count number of replicas with a record from an earlier term and declare it committed if it is at majority
    - Details are subtle, and omitted
    - Instead, leader must replicate a new log record in its current term

Raft Protocol

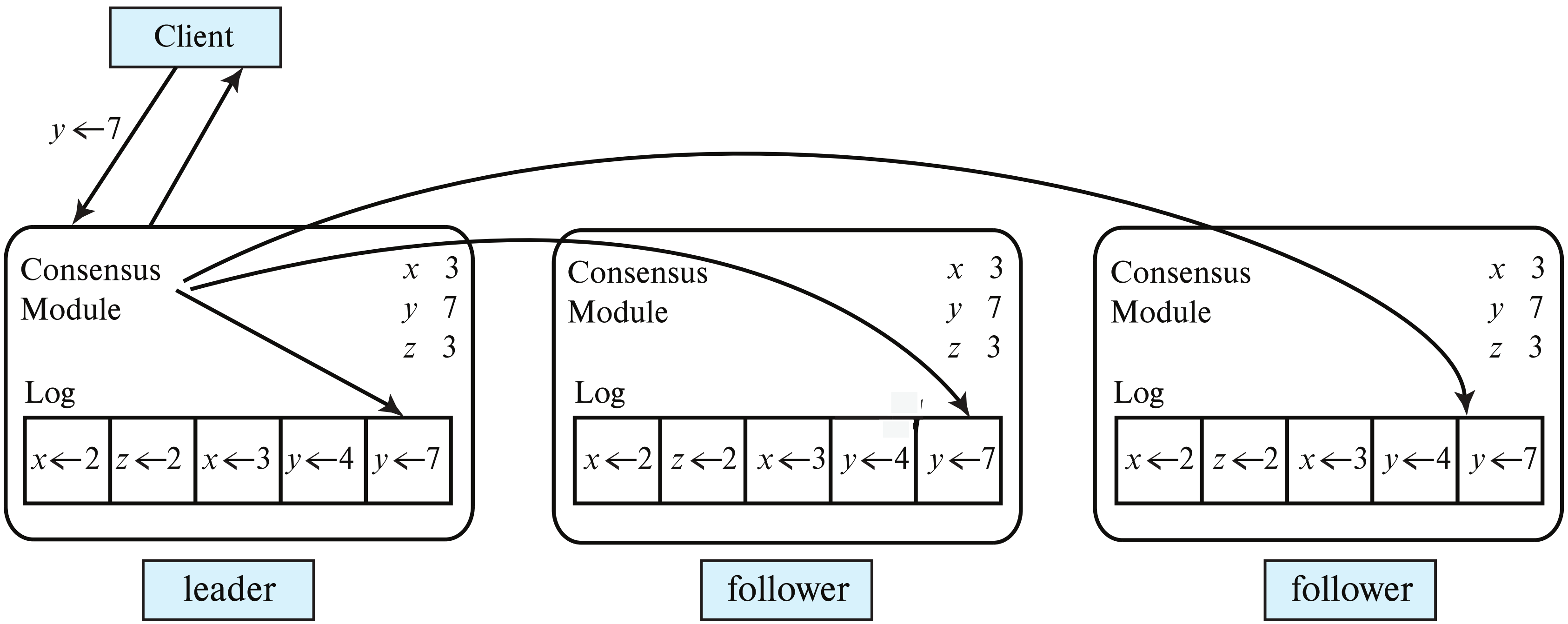
- There are many more subtle details that need to be taken care of
  - Consistency even in face of multiple failures and restarts
  - Maintaining cluster membership, cluster membership changes
- Raft has been proven formally correct
- See bibliographic notes for more details of above

Fault-Tolerant Services using Replicated State Machines

- Key requirement: make a service fault tolerant
  - E.g., lock manager, key-value storage system, ….
- State machines are a powerful approach to creating such services
- A **state machine**
  - Has a stored state, and receives inputs
  - Makes state transitions on each input, and may output some results
    - Transitions and output must be deterministic
- A **replicated state machine** is a state machine that is replicated on multiple nodes
  - All replicas must get exactly the same inputs
    - Replicated log! State machine processes only committed inputs!
  - Even if some of the nodes fail, state and output can be obtained from other nodes

Replicated State Machine

- Replicated state machine based on replicated log
- Example commands assign values to variables



Leader declares log record committed after it is replicated at a majority of nodes. Update of state machine at each replica happens only after log record has been committed.

Uses of Replicated State Machines

- Replicated state machines can be used to implement wide variety of services
  - Inputs can specify operations with parameters
  - But operations must be deterministic
  - Result of operation can be sent from any replica
    - Gets executed only when log record is committed in replicated log
    - Usually sent from leader, which knows which part of log is committed
- Example: **Fault-tolerant lock manager**
  - State: lock table
  - Operations: lock requests and lock releases
  - Output: grant, or rollback requests on deadlock
  - Centralized implementation is made fault tolerant by simply running it on a replicated state machine

Uses of Replicated State Machines

- **Fault tolerant key-value store**
  - State: key-value storage state
  - Operations: get() and put() are first logged
    - Operations executed when the log record is in committed state
    - Note: even get() operations need to be processed via log
- Google Spanner uses replicated state machine to implement key-value store
  - Data is partitioned, and each partition is replicated across multiple nodes
  - Replicas of a partition form a *Paxos group* with one node as leader
  - Operations initiated at leader, and replicated to other nodes

Two-Phase Commit Using Consensus

- Basic two-phase commit can result in blocking
- **Non-blocking two-phase commit** can be implemented using consensus
  - Key idea: Record commit decisions using consensus protocol instead of logging it at coordinator
  - As long as majority of sites are up and reachable, decision will be known
    - Blocking is then avoided
- Used e.g. in Google spanner, for transactions that span partitions
  - 2PC is coordinated by Paxos group leader at any 1 partition
  - Lock table is implemented using replicated state machine
    - Even if leader fails, new leader can see up-to-date lock state

End of Chapter 23

Extra Slides – Material Not in Text

- Weak Consistency
- Miscellaneous

Dynamo: Basics

- Provides a key-value store with basic get/put interface
  - Data values entirely uninterpreted by system
    - Unlike Bigtable, PNUTS, Megastore, etc.
- Underlying storage based on DHTs using consistent hashing with virtual processors
- Replication (N-ary)
  - Data stored in node to which key is mapped, as well as N-1 consecutive successors in ring
  - Replication at level of key range (virtual node)
  - Put call may return before data has been stored on all replicas
    - Reduces latency, at risk of consistency
    - Programmer can control degree of consistency (Q_R, Q_W and S) per instance (relation)

Performing Put/Get Operations

- Get/put requests handled by a coordinator (one of the nodes containing a replica of the item)
- Upon receiving a put() request for a key
  - the coordinator generates the vector clock for the new version and writes the new version locally
  - The coordinator then sends the new version (along with the new vector clock) to the N highest-ranked reachable nodes.
  - If at least Q_W-1 nodes respond then the write is considered successful.
- For a get() request
  - the coordinator requests all existing versions of data for that key from the N highest-ranked reachable nodes in the preference list for that key,
  - Waits for Q_R responses before returning the result to the client.
  - Returns all causally unrelated (incomparable) versions
  - Application should reconcile divergent versions and write back a reconciled version superseding the current versions

How to Reconcile Inconsistent Versions?

- Reconciliation is application specific
  - E.g., two sites concurrent insert items to cart
    - Merge adds both items to the final cart state
  - E.g., S1 adds item A, S2 deletes item B
    - Merge adds item A, but deleted item B resurfaces
    - Cannot distinguish S2 deletes B from S1 add B
    - Problem: operations are inferred from states of divergent versions
    - Better solution (not supported in Dynamo) keep track of history of operations

Availability vs Latency

- Abadi's classification system: **PACELC**
  - CAP theorem only matters when there is a partition
  - Even if partitions are rare, applications may trade off consistency for latency
    - E.g. PNUTS allows inconsistent reads to reduce latency
      - Critical for many applications
    - But update protocol (via master) ensures consistency over availability
  - Thus Abadi asks two questions:
    - If there is **P**artitioning, how does system tradeoff **A**vailability for **C**onsistency
    - **E**lse how does system trade off **L**atency for **C**onsistency
  - E.g., Megastore: PC/EC
    PNUTS: PC/EL
    Dynamo (by default): PA/EL

Amazon Dynamo

- Distributed data storage system supporting very high availability
  - Even at cost of consistency
  - E.g., motivation from Amazon: Web users should always be able to add items to their cart
    - Even if they are connected to an app server which is now in a minority partition
    - Data should be synchronized with majority partition eventually
    - Inconsistency may be visible (briefly) to users
      - preferable to losing a customer!
- DynamoDB: part of Amazon Web Service, can subscribe and use over the Web

Bully Algorithm Details

- If site S_i sends a request that is not answered by the coordinator within a time interval *T*, assume that the coordinator has failed S_i tries to elect itself as the new coordinator.
- S_i sends an election message to every site with a higher identification number, S_i then waits for any of these processes to answer within *T*.
- If no response within *T*, assume that all sites with number greater than *i* have failed, S_i elects itself the new coordinator.
- If answer is received S_i begins time interval *T*', waiting to receive a message that a site with a higher identification number has been elected.

Bully Algorithm (Cont.)

- If no message is sent within T' , assume the site with a higher number has failed; S_i restarts the algorithm.
- After a failed site recovers, it immediately begins execution of the same algorithm.
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.