

Non-Standard Database Systems

Distributed Databases

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg



Sommersemester 2020

Version July 10, 2020

Adapted from slides for textbook "Database System Concepts"
by Silberschatz, Korth, Sudarshan
<http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/index.html>

Introduction

- A distributed database system consists of **loosely coupled sites** that share no physical component (like disk or RAM).
- Database systems that run on each site are **independent** of each other.
- **Transactions** may access data at **one or more sites**.

Homogeneous vs. Heterogeneous Distributed Databases

- In a **homogeneous** distributed database
 - All sites have **identical software**
 - Are **aware of each other** and agree to cooperate in processing user requests.
 - Each site surrenders part of its **autonomy** in terms of right to change schemas or software
 - Appears to user as a **single system**
- In a **heterogeneous** distributed database
 - Different sites may use **different schemas** and **software**
 - Difference in schema is a major problem for **query processing**
 - Difference in software is a major problem for **transaction processing**
 - Sites may **not be aware of each other** and may provide only limited facilities for cooperation in transaction processing

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 Commit Protocols
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 Concurrency Control
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 Commit Protocols
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 Concurrency Control
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

Distributed Data Storage

- Assume **relational data model**
- **Replication**
 - system maintains multiple copies of data, stored in different sites
- **Fragmentation**
 - relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be **combined**
 - relation is partitioned into several fragments
 - system maintains several identical replicas of each such fragment.

Data Replication/1

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- **Full replication**: relation is stored at all sites
- **Fully redundant databases**: every site contains copy of entire database

Data Replication/2

- Advantages of Replication
 - **Availability**: failure of site containing relation r does not result in unavailability of r as replicas exist.
 - **Parallelism**: queries on r may be processed by several nodes in parallel.
 - **Reduced data transfer**: relation r is available locally at each site containing a replica of r .
- Disadvantages of Replication
 - Increased **cost of updates**: each replica of relation r must be updated.
 - Increased **complexity of concurrency control**: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.

Data Fragmentation

- Division of relation r into fragments r_1, r_2, \dots, r_n which contain sufficient information to reconstruct relation r .
- **Horizontal fragmentation**: each tuple of r is assigned to one or more fragments:

$$r = \bigcup_{i=1}^n r_i$$

- **Vertical fragmentation**: schema of relation r is split into several smaller schemas.
 - All schemas must contain a common candidate key to ensure **lossless join property**.
 - A special attribute, the **tuple-id attribute** may be added to each schema to serve as a candidate key.
 - Let $sch(r_i) \cap sch(r_j)$ be the candidate key, then $r = r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$.

Horizontal Fragmentation of *account* Relation

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

Table: $account_1 = \sigma_{branch_name='Hillside'}(account)$

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

Table: $account_2 = \sigma_{branch_name='Valleyview'}(account)$

Vertical Fragmentation of *employee_info* Relation

<i>branch_name</i>	<i>customer_name</i>	<i>tuple_id</i>
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

Table: $deposit_1 = \Pi_{branch_name, customer_name, tuple_id}(employee_info)$

<i>account_number</i>	<i>balance</i>	<i>tuple_id</i>
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

Table: $deposit_2 = \Pi_{account_number, balance, tuple_id}(employee_info)$

Advantages of Fragmentation

- **Horizontal**:
 - allows **parallel processing** on fragments of a relation
 - allows a **relation to be split** so that tuples are located where they are most frequently accessed
- **Vertical**:
 - allows **tuples to be split** so that each part of the tuple is stored where it is most frequently accessed
 - tuple-id attribute allows efficient **joining of vertical fragments**
 - allows **parallel processing** on a relation
- Vertical and horizontal fragmentation can be **mixed**.
 - Fragments may be successively fragmented to an arbitrary depth.

Data Transparency

- **Data transparency:** Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system.
 - fragmentation transparency
 - replication transparency
 - location transparency

Naming of Data Items - Criteria

1. Every data item must have a system-wide **unique name**.
2. It should be possible to **find the location** of data items efficiently.
3. It should be possible to **change the location** of data items transparently.
4. Each site should be able to **create new data items** autonomously.

Centralized Scheme - Name Server

- **Structure:**
 - name server assigns all names
 - each site maintains a record of local data items
 - sites ask name server to locate non-local data items
- **Advantages:**
 - satisfies naming criteria 1-3
- **Disadvantages:**
 - does not satisfy naming criterion 4
 - name server is a potential performance bottleneck
 - name server is a single point of failure

Use of Aliases

- Alternative to centralized scheme: each site **prefixes** its own site identifier to any name that it generates, e.g., site17.account.
 - Fulfills having a **unique identifier**, and avoids problems associated with central control.
 - However, fails to achieve **location transparency**.
- Solution: Create a set of **aliases** for data items; store the mapping of aliases to the real names at each site.
- The user can be unaware of the physical location of a data item, and is unaffected if the data item is moved from one site to another.

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 Commit Protocols
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 Concurrency Control
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

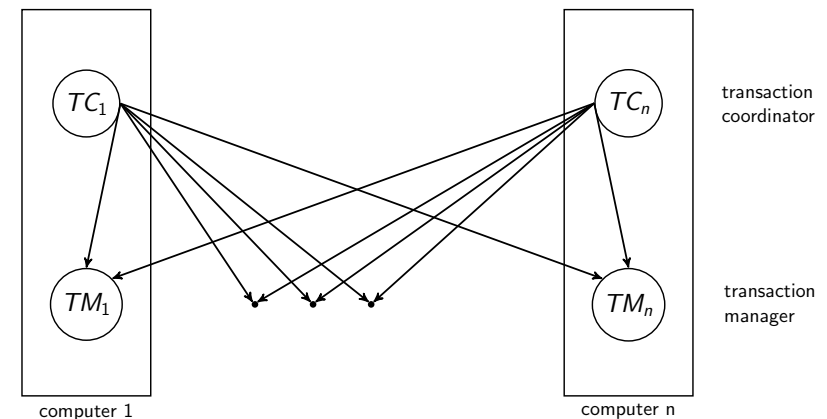
Local and Global Transactions

- Local transaction:
 - Accesses and/or updates data at **only one site**.
- Global transaction:
 - Accesses and/or updates data at **several different sites**.
 - Global transactions are split into local subtransactions for execution.

Distributed Transactions

- Each site has:
 - local transaction manager
 - transaction coordinator
- Local transaction manager:
 - ensures ACID for local transactions
 - maintains log for recovery purposes
 - coordinates concurrent execution of local transactions
- Transaction coordinator:
 - starts execution of transactions that originate at the site (local or global)
 - distributes subtransactions to appropriate sites for execution
 - coordinates termination of each transaction that originates at the site: either commit at all sites or aborted at all sites

Transaction System Architecture



System Failure Modes

- Failures unique to distributed systems:
 - **site failure:**
 - a site is down
 - **loss of messages:**
 - handled by network transmission control protocols such as TCP-IP
 - **communication link failure:**
 - handled by network protocols, by routing messages via alternative links
 - **network partition:**
 - network is split into two or more disconnected subsystems
 - a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 **Commit Protocols**
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 Concurrency Control
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

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.
 - not acceptable to have a transaction committed at one site and aborted at another
- The **two-phase commit** (2PC) protocol is widely used
- The **three-phase commit** (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol. This protocol is not used in practice.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop model** — failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- 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
- Let T be a transaction initiated at site S_i , and let the transaction coordinator at S_j be C_j

Phase 1: Obtaining a Decision

- Coordinator C_i asks all participants to **prepare to commit** transaction T .
 - C_i adds record $\langle \text{prepare } T \rangle$ 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 $\langle \text{abort } T \rangle$ to the log and send abort T message to C_i
 - if the transaction can be committed, then:
 - add the record $\langle \text{ready } T \rangle$ to the log and force all records for T to stable storage
 - send ready T message to C_i

Phase 2: Recording the Decision

- T can be committed if C_i received a **ready T message** from all the participating sites, otherwise T must be aborted.
- Coordinator adds a **decision record**, $\langle \text{commit } T \rangle$ or $\langle \text{abort } T \rangle$, to the log and forces record onto stable storage. Once the record is on 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.

Handling of Failures - Site Failure

When site S_k ($k \neq i$) **recovers**, it examines its log to determine the **fate of transactions** active at the time of the failure.

- Log contain $\langle \text{commit } T \rangle$ record: T had completed
- Log contains $\langle \text{abort } T \rangle$ record: T had failed
- Log contains $\langle \text{ready } T \rangle$ record: site must consult C_i to determine the fate of T .
 - if T committed, $\text{redo}(T)$; write $\langle \text{commit } T \rangle$ record
 - if T aborted, $\text{undo}(T)$
- The log contains **none of the above log records** concerning T :
 - implies that S_k failed before responding to prepare T message from C_i
 - since S_k did not send ready T message, coordinator C_i must have aborted T (or will abort after timeout)
 - S_k executes $\text{undo}(T)$

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:
 - If an active site contains a $\langle \text{commit } T \rangle$ record in its log, then T must be committed.
 - If an active site contains an $\langle \text{abort } T \rangle$ record in its log, then T must be aborted.
 - If some active participating site does not contain a $\langle \text{ready } T \rangle$ record in its log, then the failed coordinator C_i cannot have decided to commit T .
 - Can therefore abort T ; however, such a site must reject any subsequent $\langle \text{prepare } T \rangle$ message from C_i
 - If none of the above cases holds, then all active sites must have a $\langle \text{ready } T \rangle$ record in their logs, but no additional control records (such as $\langle \text{abort } T \rangle$ or $\langle \text{commit } T \rangle$).
 - 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 in the **same partition** as the coordinator (and the coordinator) think that the sites in the other partitions have failed, and follow the usual commit protocol.
 - **No harmful results**
 - Sites that are **not in the same partition** as the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
 - **No harmful results**, but sites may still have to wait for decision from coordinator.

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 be slow and potentially **block recovery**.
- Recovery algorithms can note **lock information** in the log.
 - Instead of `<ready T>`, write out `<ready T, L>`, where 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.

Three Phase Commit (3PC)/1

- Assumptions:
 - No network partitioning
 - At any point, at least one site must be up.
 - At most K sites (participants as well as coordinator) can fail
- **Phase 1**: Identical to 2PC Phase 1.
 - Outcome: Every site is ready to commit if instructed to do so.
- Phase 2 of 2PC is split into 2 phases, Phase 2 and Phase 3 of 3PC:
 - In **Phase 2** coordinator makes a decision as in 2PC (called the **pre-commit decision**) and records it in multiple (at least K additional) sites.
 - In **Phase 3**, coordinator sends commit/abort message to all participating sites.

Three Phase Commit (3PC)/2

- Under 3PC, knowledge of pre-commit decision can be used to commit despite **coordinator failure**
 - Avoids blocking problem as long as at most K sites fail
- Drawbacks:
 - higher overheads
 - assumptions may not be satisfied in practice

Three Phase Commit (3PC)/3

- Phase 1: Obtaining **Preliminary Decision**: Identical to 2PC Phase 1.
 - Every site is ready to commit if instructed to do so
 - Under 2PC each site is obligated to wait for decision from coordinator.
 - Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure.

3PC: Phase 2. Recording the Preliminary Decision

- Coordinator adds a **decision record** (`<abort T>` or `<precommit T>`) in its log and forces record to stable storage.
- Coordinator sends a message to each participant informing it of the decision
- Participant records decision in its log
- If abort decision reached then participant aborts locally
- If pre-commit decision reached then participant replies with `<acknowledge T>`

3PC: Phase 3. Recording Decision in the Database

- Executed only if decision in phase 2 was to precommit
- Coordinator **collects acknowledgements**. It sends `<commit T>` message to the participants as soon as it receives K acknowledgements.
- Coordinator adds the record `<commit T>` in its log and forces record to stable storage.
- Coordinator sends a commit T message to each participant
- Participants take appropriate action locally

3PC: Handling Site Failure/1

- **Site Failure**: Upon recovery, a participating site examines its log and does the following:
 - Log contains `<commit T>` record: no action
 - Log contains `<abort T>` record: no action
 - Log contains `<ready T>` record, but no `<abort T>` or `<precommit T>` record: site consults C_i to determine the fate of T .
 - if C_i says T aborted, site executes `undo(T)` (and writes `<abort T>` record)
 - if C_i says T committed, site executes `redo(T)` (and writes `<commit T>` record)
 - if C_i says T pre-committed, site resumes the protocol from receipt of `precommit T` message (thus recording `<precommit T>` in the log, and sending `acknowledge T` message sent to coordinator).

3PC: Handling Site Failure/2

- Log contains <precommit T> record, but no <abort T> or <commit T>: site consults C_i to determine the fate of T .
 - if C_i says T aborted, site executes *undo* (T)
 - if C_i says T committed, site executes *redo* (T)
 - if C_i says T still in precommit state, site resumes protocol at this point
- Log contains no <ready T> record for a transaction T : site executes *undo*(T) writes <abort T> record

Alternative Models of Transaction Processing/1

- Single transaction spanning multiple sites may be inappropriate for some applications:
 - E.g. transaction crossing an organizational boundary: No organization would like to permit an externally initiated transaction to block local transactions for an indeterminate period.
- Alternative models carry out transactions by sending messages.
- Persistent messaging systems:
 - provide transactional properties for messages
 - messages are guaranteed to be delivered exactly once

Alternative Models of Transaction Processing/2

- Example: funds transfer between two banks
 - 2PC potentially blocks 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 be 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 and Workflows

- **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

Implementation of Persistent Messaging/1

- **Sending site protocol**
 - When a transaction wishes to send a persistent message, it writes a record containing the message in a **special relation** *messages_to_send*; the message is given a unique message identifier.
 - A **message delivery process** monitors the relation, and when a new message is found, it sends the message to its destination.
 - The message delivery process deletes a message from the relation only after it receives an acknowledgment from the destination site.
 - If it receives no acknowledgement from the destination site, after some time it sends the message again. It repeats this until an acknowledgment is received.
 - If after some period of time, that the message is undeliverable, exception handling code provided by the application is invoked to deal with the failure.
- Writing the message to a relation and processing it only after the transaction commits ensures that the message will be delivered if and only if the transaction commits.

Implementation of Persistent Messaging/2

- **Receiving site protocol**
 - When a site receives a persistent message, it runs a transaction that adds the message to a *received_messages* relation
 - provided message identifier is not already present in the relation
 - After the transaction commits, or if the message was already present in the relation, the receiving site sends an **acknowledgment** back to the sending site.
 - sending the acknowledgment before the transaction commits is not safe since a system failure may then result in loss of the message.
 - In many messaging systems, it is possible for messages to get **delayed arbitrarily**, although such delays are very unlikely.
 - Each message is given a **timestamp**, and if the timestamp of a received message is older than some cutoff, the message is discarded.
 - All messages recorded in the received messages relation that are older than the cutoff can be deleted.

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 Commit Protocols
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 **Concurrency Control**
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

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/1

- System maintains a **single lock manager** that resides in a single chosen site, say S_i
- When a transaction needs to lock a data item, it sends a **lock request** to S_i and **lock manager** determines whether the lock can be granted immediately
 - If yes, lock manager sends a message to the site which initiated the request
 - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

Single-Lock-Manager Approach/2

- 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 this approach, functionality of locking is implemented by **lock managers** at each site
 - Lock managers control access to local data items
- Advantage: work is distributed and can be made **robust to failures**
- Disadvantage: **deadlock detection** is more complicated
 - Lock managers cooperate for deadlock detection
- Several variants of this approach
 - **Primary copy**
 - **Majority protocol**
 - **Biased protocol**
 - **Quorum consensus**

Primary Copy

- Choose one replica of data item to be the **primary copy**.
 - Site containing the replica is called the **primary site** for that data item
 - Different data items can have different primary sites
- When a transaction needs to lock a data item Q , it requests a lock at the primary site of Q .
 - Implicitly gets lock on all replicas of the data item
- Benefit
 - Concurrency control for replicated data handled similarly to unreplicated data — **simple implementation**.
- Drawback
 - If the primary site of Q fails, Q is **inaccessible** even though other sites containing a replica may be accessible.

Majority Protocol/1

- **Local lock manager** at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item Q residing at site S_i , a **message** is sent to S_i 's lock manager.
 - If Q is locked in an incompatible mode, then the request is **delayed** until it can be granted.
 - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

Majority Protocol/2

- In case of **replicated data**
 - If Q is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which Q is stored.
 - The transaction does not operate on Q until it has obtained a lock on a majority of the replicas of Q .
 - When writing the data item, transaction performs writes on all replicas.
- Benefit
 - Can be used even when **some sites are unavailable**
 - details on how handle writes in the presence of site failure later
- Drawback
 - Requires $2(n/2 + 1)$ messages for handling **lock requests**, and $(n/2 + 1)$ messages for handling **unlock requests**.
 - Potential for **deadlock** even with single item — e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data.

Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
- **Shared locks**: When a transaction needs to lock data item Q , it simply requests a lock on Q from the lock manager at one site containing a replica of Q .
- **Exclusive locks**: When transaction needs to lock data item Q , it requests a lock on Q from the lock manager at all sites containing a replica of Q .
- Advantage — imposes less overhead on read operations.
- Disadvantage — additional overhead on writes

Quorum Consensus Protocol

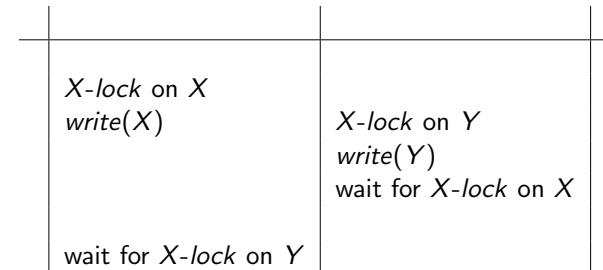
- A generalization of both majority and biased protocols
- Each site is assigned a weight.
 - Let S be the total of all site weights
- Choose two values read quorum Q_r and write quorum Q_w
 - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
 - Quorums can be chosen (and S computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is $\geq Q_w$
- For now we assume all replicas are written
 - Extensions to allow some sites to be unavailable described later

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(Y)
write(X)

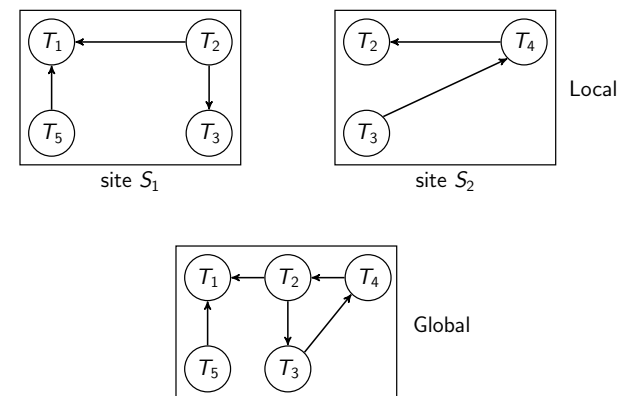


Result: deadlock which cannot be detected locally at either site

Centralized Approach

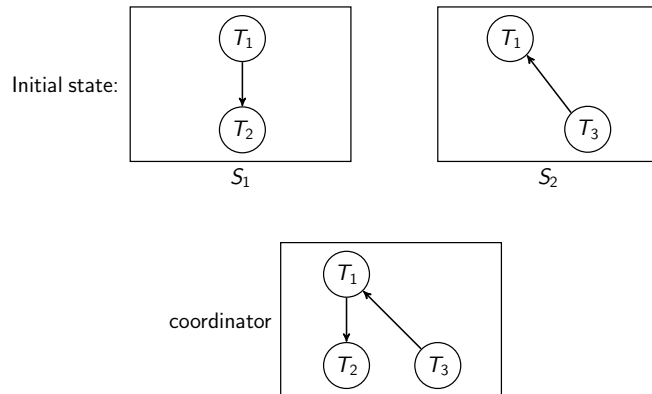
- A global wait-for graph is constructed and maintained in a single site: the deadlock-detection coordinator
 - Real graph: Real, but unknown, state of the system.
 - Constructed graph: Approximation generated by the controller during the execution of its algorithm.
- The global wait-for graph can be 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



Example Wait-For Graph for False Cycles

Initial state:



False Cycles

- Suppose that starting from the state shown in figure,
 - 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)
 - 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.

Unnecessary Rollbacks

- 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.
- Unnecessary rollbacks can result from **false cycles** in the global wait-for graph; however, likelihood of false cycles is low.

Timestamp-Based Protocols/1

- Each **transaction** is issued a **timestamp** when it enters the system. If an old transaction T_i has time-stamp $TS(T_i)$, a new transaction T_j is assigned time-stamp $TS(T_j)$ such that $TS(T_i) < TS(T_j)$.
- The protocol manages concurrent execution such that the **time-stamps determine the serializability order**.
- In order to assure such behavior, the protocol maintains for **each data Q** **two timestamp values**:
 - $W\text{-timestamp}(Q)$ is the largest time-stamp of any transaction that executed **write(Q)** successfully.
 - $R\text{-timestamp}(Q)$ is the largest time-stamp of any transaction that executed **read(Q)** successfully.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are **executed in timestamp order**.

Timestamp-Based Protocols/2

- Transaction T_i issues a **read(Q)**:
 - If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was **already overwritten**.
 - Hence, the **read** operation is **rejected**, and T_i is **rolled back**.
 - Otherwise the **read operation is executed**, and $R\text{-timestamp}(Q)$ is set to $\max(R\text{-timestamp}(Q), TS(T_i))$.
- Transaction T_i issues **write(Q)**:
 - If $TS(T_i) < R\text{-timestamp}(Q)$, then the **value of Q** that T_i is producing was **needed previously**, and the system assumed that that value would never be produced.
 - Hence, the **write(Q)** operation is **rejected**, and T_i is rolled back.
 - If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to **write an obsolete value of Q**.
 - Hence, this **write(Q)** operation is **rejected**, and T_i is rolled back.
 - Otherwise, the **write(Q)** operation is **executed**, and $W\text{-timestamp}(Q)$ is set to $TS(T_i)$.

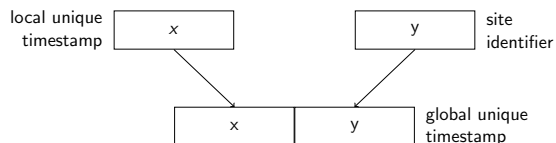
Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
	read(Y)			read(X)
read(Y)		write(Y) write(Z)		
	read(Z) abort			read(Z)
read(X)		write(W) abort	read(W)	
				write(Y) write(Z)

Timestamping/1

- 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** $\langle x, y \rangle$ is obtained by concatenating the unique local timestamp x with the unique identifier y .



Timestamping/2

- A site with a **slow clock** will assign smaller timestamps
 - still logically correct: serializability not affected
 - but: “disadvantages” transactions
- Lamport-Clocks** fix this problem:
 - each site S_i defines a **logical clock** LC_i , which generates the unique local timestamp;
 - increment timestamp LC_i for each new transactions issued by S_i ;
 - whenever a read or write **request is received** from a transaction T_i with timestamp $\langle x, y \rangle$ and $x > LC_i$, then set LC_i to $x + 1$.

Replication with Weak Consistency/1

- Many commercial databases support replication of data with **weak degrees of consistency** (i.e., without a guarantee of serializability)
- Example: **master-slave replication**: updates are performed at a single “master” site, and propagated to “slave” sites.
 - Propagation is not part of the update transaction: its is decoupled
 - May be immediately after transaction commits
 - May be periodic
 - Data may only be read at slave sites, not updated
 - No need to obtain locks at any remote site
 - Particularly useful for distributing information
 - E.g. from central office to branch-office
 - Also useful for running read-only queries offline from the main database

Replication with Weak Consistency/2

- Replicas should see a **transaction-consistent snapshot** of the database
 - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- Example: Oracle provides a create snapshot statement to create a snapshot of a relation or a set of relations at a remote site
 - snapshot refresh either by recomputation or by incremental update
 - automatic refresh (continuous or periodic) or manual refresh

Multimaster and Lazy Replication

- With **multimaster replication** (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all other replicas
 - basic model in distributed databases, where transactions are unaware of the details of replication
 - database system propagates updates as part of the same transaction
 - coupled with **2 phase commit**
- Many systems support **lazy propagation** where updates are transmitted after transaction commits
 - allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency

Outline

- 1 Distributed Data Storage
- 2 Distributed Transactions
- 3 Commit Protocols
 - Two Phase Commit (2PC)
 - Three Phase Commit (3PC)
 - Persistent Messaging
- 4 Concurrency Control
 - Locking
 - Deadlocks
 - Timestamping
 - Weak Consistency
- 5 Availability

Availability

- **High availability:** time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- **Robustness:** ability of system to function spite of failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must
 - Detect failures
 - Reconfigure the system so computation may continue
 - Recovery/reintegration when a site 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/1

- Reconfiguration:
 - Abort all transactions that were active at a failed site
 - making them wait could interfere with other transactions since they may hold locks on other sites
 - however, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data 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/2

- 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 should be able to continue even if some sites are down
- Solution: majority based approach
 - alternative of “read one write all available” is tantalizing but causes problems

Majority-Based Approach/1

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable.
- Each replica of each item has a version number which is updated when the replica is updated, as outlined below.
- A lock request is sent to at least 1/2 the sites at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites.
- Read operations look at all replicas locked, and read the value from the replica with largest version number.
 - may write this value and version number back to replicas with lower version numbers (no need to obtain locks on all replicas for this task)

Majority-Based Approach/2

- **Write operations**
 - find highest version number like read, 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
- **Failures** (network and site) cause no problems as long as
 - sites at commit contain a majority of replicas of any updated data items
 - during reads a majority of replicas are available to find version numbers
 - subject to above, **2 phase commit** can be used to update replicas

Read One Write All (Available)

- **Quorum consensus algorithm** can be similarly extended
- **Biased protocol** is a special case of quorum consensus
 - allows reads to read any one replica but updates require all replicas to be available at commit time (called read one write all)
- **Read one write all available** (ignoring failed sites) is attractive, but incorrect

Link Failure and Network Partitioning

- **Link failure:**
 - 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.
 - If site was aware of failure, reintegration could have been performed, but no way to guarantee this.
- **Network partitioning:**
 - With network partitioning, sites in each partition may update same item concurrently (believing sites in other partitions have all failed).

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.
 - other solutions with better concurrency also available

Comparison with Remote Backup

- **Remote backup (hot spare) systems** are also designed to provide high availability.
 - simpler and lower overhead
 - all actions performed at a single site, and only log records shipped
 - no need for distributed concurrency control or 2 phase commit
- **Distributed databases** with replicas of data items
 - provide higher availability by having multiple (> 2) replicas and using the majority protocol
 - avoid failure detection and switchover time associated with remote backup systems

Coordinator Selection

- **Backup coordinators**
 - site which 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.
- **Election algorithms**
 - used to elect a new coordinator in case of failures
 - Example: **Bully Algorithm** — applicable to systems where every site can send a message to every other site.

Bully Algorithm

- **Bully algorithm:**
 - all nodes S_i are numbered
 - node with highest i -value is coordinator
- **Coordinator election algorithm** (started by S_i):
 - S_i sends an **election message** to every site S_k with $k > i$ and waits for response within T .
 - **no response:** S_i elects itself and informs all $S_j, j < i$.
 - **response:** Wait for the outcome of the coordinator election. (After timeout interval T' , restart election from scratch.)
- S_i **starts coordinator election** (tries to elect itself coordinator) if
 - **coordinator failure:** coordinator does not answer within time interval T
 - **recovery:** when S_i recovers from failure
 - even if there is already a coordinator in the system
 - **election message received:** S_i is not coordinator and receives election message from some node $S_j, j < i$
 - if S_i is coordinator there is no need for election and S_j is informed

What is Consistency?

- **Consistency in Databases (ACID):**
 - database has a set of integrity constraints
 - a database state is consistent when all **integrity constraints are satisfied**
 - each **transaction run individually** on a consistent database state must leave the database in a consistent state
- **Consistency in distributed systems** with replication
 - **Strong consistency**¹: a schedule with read and write operations on a replicated object should give results and final state equivalent to some schedule on a single copy of the object, with the order of operations from a single site preserved
 - replicated data item appears to be a single data item stored in shared memory to which different sites have sequential access
 - **Weak consistency** (several forms)

¹Also “sequential consistency”, defined by L. Lamport, 1979

Availability

- Traditionally, availability of **centralized server**
- For distributed systems: **availability of system to process requests**
- In large distributed system **failures frequently happen**:
 - a node is down
 - network partitioning
- **Distributed consensus algorithms will block** during partitions to ensure consistency
- Some applications require **high availability even at cost of consistency**

Brewer's CAP Theorem

- Three properties of a system
 - **Consistency** (all copies have same value)
 - **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
- Very large systems will partition at some point
 - ⇒ choose one of **consistency or availability**
 - traditional databases choose consistency
 - most Web applications choose availability (except for specific parts such as order processing)

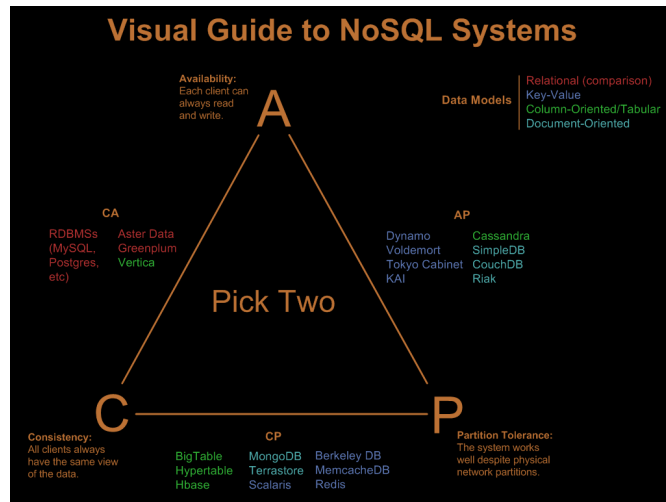
Replication with Weak Consistency

- Many systems support replication of data with **weak degree of consistency** (i.e., without a guarantee of serializability)
 - $Q_r + Q_w \leq S$ or $2 * Q_w < S$
- **Trade off consistency** for:
 - **availability**: when not enough sites are available to ensure quorum
 - **low latency**: small Q_r -values allow fast local reads
- **Key issues**:
 - Reads may get **old versions**
 - Writes may occur in parallel, leading to **inconsistent versions**
 - Question: how to detect, and how to resolve

Example: Trade off Consistency for Availability or Latency

- Real systems may use a mix of tradeoff options.
- **Example**: Yahoo!'s PNUTS distributed database
 - allows **inconsistent reads** to reduce latency (critical for many applications)
 - but **consistent updates** (via master) to ensures consistency over availability

Example: CAP Choice of Various Systems



Source: <http://blog.nahurst.com/visual-guide-to-nosql-systems>

BASE Properties

- **BASE** is an acronym for
 - **Basically Available:** availability is given priority over consistency
 - **Soft state:** copies of a data item may be inconsistent
 - **Eventual Consistency:** copies becomes consistent at some later time if there are no more updates to that data item.
- BASE is an **alternative to ACID** as used in traditional databases.

Eventual Consistency

- **Definition 1:** 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.
- **Definition 2:** For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service.

How to converge?

- **Anti entropy:** exchange versions
- **Conflict detection:**
 - **timestamp:** can identify last writer, but cannot distinguish sequential from branching version history
 - **vector clocks:** detects branching histories (i.e. conflicting updates)
- **Reconciliation:** decide on final state
 - **last updater wins:** data item with highest time stamp is final state
 - **user defined:** user must solve conflict
- **When to reconcile?**
 - **read repair:** fix conflicts at read time
 - **write repair:** fix conflicts at write time
 - **asynchronous repair:** separate process fixes conflicts

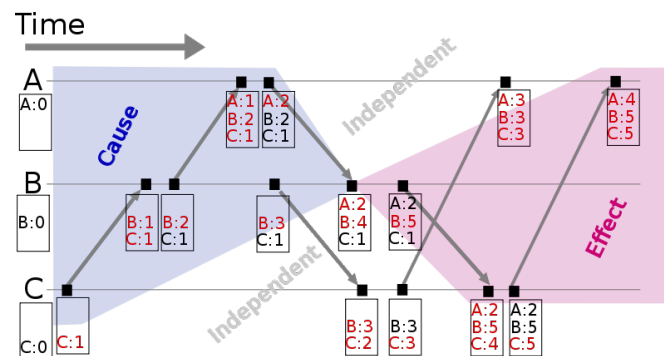
Vector Clock/1

- **Replica:** each data item is replicated at n sites S_i , $1 \leq i \leq n$
- **Data item:** d_i is the copy of data item d at site S_i
- **Vector clock:**
 - each d_i has vector $V_i[j]$, $1 \leq j \leq n$
 - $V_i[j]$: timestamp of data item d at site S_j as known by S_i
 - initialization: $V_i[j] \leftarrow 0$, $1 \leq i, j \leq n$
- **Local update** at site S_i : $V_i[i] \leftarrow V_i[i] + 1$
- **Copy from remote site S_k** with vector V_k to S_i :
 - $V_i[i] \leftarrow V_i[i] + 1$
 - for all $1 \leq j \leq n$: $V_i[j] \leftarrow \max(V_i[j], V_k[j])$

Vector Clock/2

- **Exchange versions** between replica S_i , S_j
 - d_i with vector V_i from site S_j
 - d_j with vector V_j from site S_i
- **Conflict detection:**
 - $\exists x, y : V_i[x] < V_j[x] \wedge V_i[y] > V_j[y]$: branching history
 - otherwise: linear history
- **Linear History:** d_j is a newer version of d_i
 - the updates of d_j include the updates of d_i
 - reconciliation: keep new version, $d_i \leftarrow d_j$
- **Branching history:** conflicting updates
 - d_i and d_j have received independent updates in parallel
 - reconciliation: some sort of conflict resolution (e.g. user interaction)

Vector Clock/3 – Example



Source: https://commons.wikimedia.org/wiki/File:Vector_Clock.svg