Chapter 8: Replication and Consistency

- Replication: A key to providing good performance, high availability and fault tolerance in distributed systems (passive and active).
- The important issue is keeping replicas consistent.
- Consistency models and protocols
- The Gossip architecture: an approach to propagate updates.

Enhancing Services by replicating data

- Performance

When a distributed system needs to scale in numbers and geographical area, performance can be improved by replicating servers.

- Fault Tolerance

Under the fail-stop model, if up to *N* of *N +1* servers crash, at least one remains to supply the service.

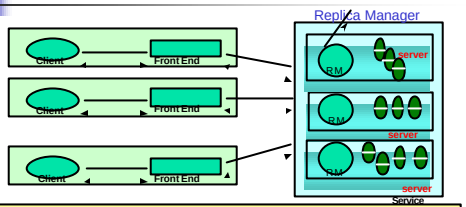
- Increased Availability

Service may not be available when servers fail or when the network is partitioned.

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

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

Basic Model of Replication



Replication Transparency: User/client need not know that multiple physical copies of data exist.
Replication Consistency: Data is consistent on all of the replicas (or is in the process of becoming consistent)

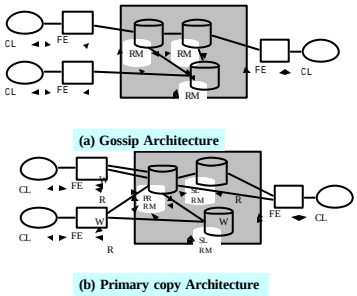
Replication Management (1)

- Front End: Request Communication
  - Requests can be made to a single RM or to multiple RMs
- Coordination: The RMs decide
  - whether the request is to be applied
  - the order of requests
    - FIFO ordering: If a FE issues *r* then *r'*, then any correct RM handles *r* and then *r'*.
    - Causal ordering: If the issue of *r* "happened before" the issue of *r'*, then any correct RM handles *r* and then *r'*.
    - Total ordering: If a correct RM handles *r* and then *r'*, then any correct RM handles *r* and then *r'*.
- Execution: The RMs execute the request tentatively.

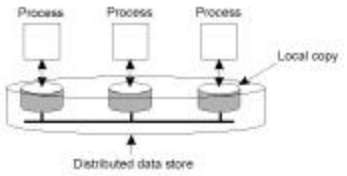
Replication Management (2)

- Agreement: The RMs attempt to reach consensus on the effect of the request.
  - E.g., Two phase commit through a coordinator
- Response
  - One or more RMs responds to the front end.
  - In the case of fail-stop model, the FE returns the first response to arrive.

Basic Replication Architecture

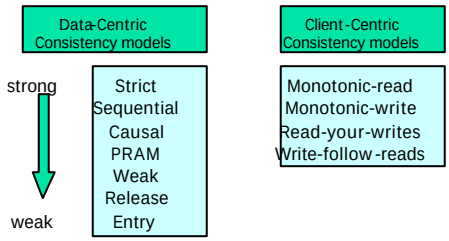


Consistency models (1)

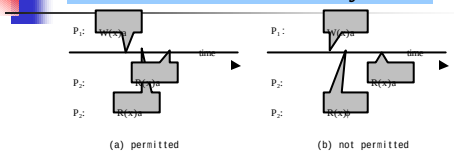


- Consistency model (or *consistency semantics*)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly
  - All models attempt to return the results of the last write for a read operation
    - Differ in how "last" write is determined/defined

Consistency models (2)

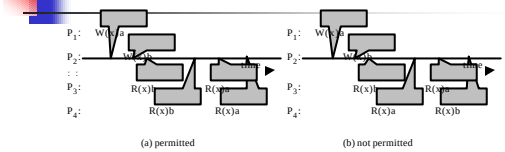


Strict Consistency



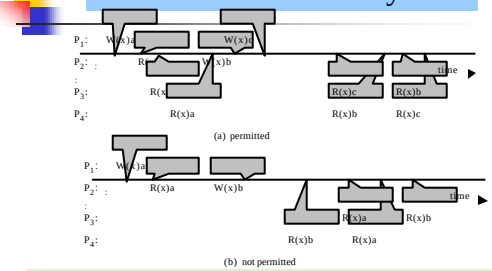
- Any read always returns the result of the most recent write
  - Implicitly assumes the presence of a global clock
  - A write is immediately visible to all processes
    - An ideal model, but difficult to achieve in real systems (network delays can be variable)

Sequential Consistency



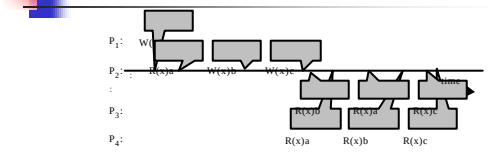
- Sequential consistency: weaker than strict consistency
  - Assumes all operations are executed in some sequential order and each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order
    - Nothing is said about "most recent write "

Causal consistency



- Causally related writes must be seen by all processes in the same order.
  - Concurrent writes may be seen in different orders on different machines

PRAM consistency

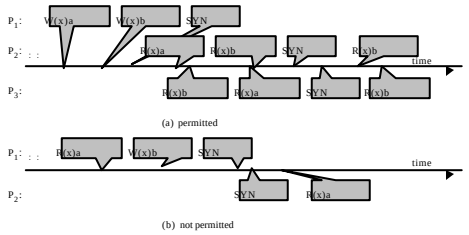


- Pipelined Random Access Memory Consistency: writes from a process are seen by others in the same order. Writes from different processes may be seen in different order (even if causally related)
  - Relaxes causal consistency
  - Simple implementation: tag each write by (Proc ID, seq #)

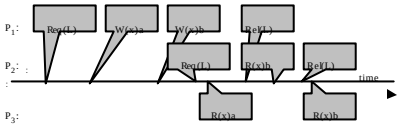
Weak consistency(1)

- Weak consistency
  - Accesses to synchronization variables associated with a data store are sequentially consistent
  - No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  - No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Weak consistency(2)

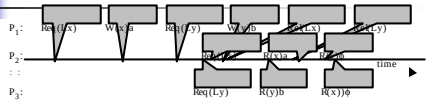


Release consistency



- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed
- Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

Entry consistency



- An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
- Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
- After an exclusive mode access to a synchronization variable has been performed, any other process's next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Summary of Data-Centric Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters (Global physical time)
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time (Total ordering)
Causal	All processes see causally-related shared accesses in the same order (causal ordering)
PRAM	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order (Single-process ordering)

Consistency models not using synchronization operations.

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

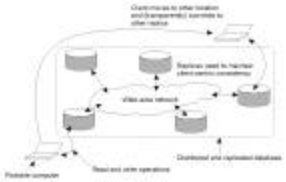
Models with synchronization operations.

Client-Centric Consistency models

- Many systems: one or few processes perform updates
  - How frequently should these updates be made available to other read-only processes?
- Examples:
  - DNS: single naming authority per domain
  - Only naming authority allowed updates (no write-write conflicts)
  - How should read-write conflicts (consistency) be addressed?
  - NIS: user information database in Unix systems
    - Only sys-admins update database, users only read data
    - Only user updates are changes to password

Eventual Consistency

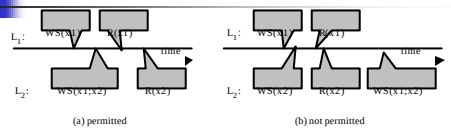
- In absence of updates, all replicas converge towards identical copies
  - Only requirement: an update should eventually propagate to all replicas
  - Cheap to implement: no or infrequent write-write conflicts
  - Things work fine so long as user accesses same replica
  - What if they don't:



Semantics of Client-Centric Models

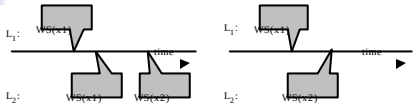
- Assume read operations by a single process *P* at two *different* local copies of the same data store
  - Four different consistency semantics
- ***Monotonic reads***
  - Once read, subsequent reads on that data items return same or more recent values
- ***Monotonic writes***
  - A write must be propagated to all replicas before a successive write by the *same process*
  - Resembles FIFO consistency (writes from same process are processed in same order)
- ***Read your writes***: read(x) always returns write(x) by that process
- ***Writes follow reads***: write(x) following read(x) will take place on same or more recent version of x

Monotonic Reads



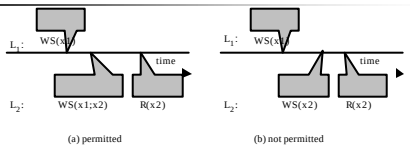
- The read operations performed by a single process *P* at two different local copies (L1 and L2) of the same data store.
- Where x_i denotes the version of x at local copy Li, and WS represents a write sequence, WS(x_1; x_2) denotes that x_1 version is formed before x_2.
- Ex: a user reads email x_1 in New York, and then flies to Toronto, open the copy of email box there, monotonic reads consistency guarantees that x_1 will be in the mail box in Toronto.

Monotonic Writes



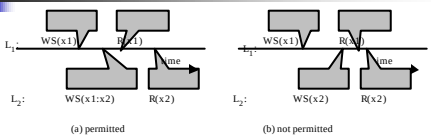
- The write operations performed by a single process *P* at two different local copies of the same data store
- Resembles to PRAM, but here we are considering consistency only for a single process (client) instead of for a collection of concurrent processes.

Read-Your-Writes



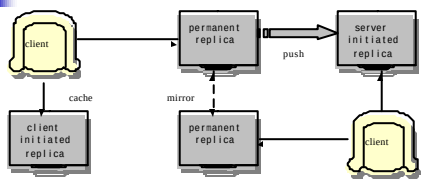
- Closed related to monotonic reads
- A write operation is always completed before a successive read operation by the same process
- Ex: editor and browser, if not integrated, you may not read-your-writes of an HTML page

Write-follow-reads



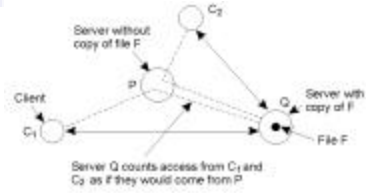
- Updates are propagated as the result of previous read operation
- Any successive write operation on x by a process will be performed on a copy of x that is most recently read by that process
- Ex: comments on news group, let A an article read recently, R the response to that article, then R must follows A.

Replica Placement



The logical organization of different kinds of copies of a data store into three concentric rings.

Server-Initiated Replicas



Counting access requests from different clients:

(1) system maintains two limits: del(S, F) and rep(S, F)

(2) if $count_Q(P, F) > rep(Q, F)$, then replicates F on P

Update Propagation

- Propagate only a notification of an update: a so called invalidation protocol, only informs other copies that their data are no longer valid. A copy updates itself when needed. Useful when reads/writes is small.
- Transfer data from one copy to another: useful when reads/writes is relatively high. Pack multiple modifications into a single update package will save communication overhead.
- Propagate the update operation to other copies: also referred to as active replication. Let every copy do the same update operation.

Pull *vs* Push Protocols

- Push-based (server-based): updates are propagated to other copies actively. Useful for replicas need to maintain a relatively high degree of consistency.
- Push-based (client-based): a server or client requests another server to send it any updates it has at that moment. Efficient when reads/writes is low.

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update to all clients	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Epidemic Protocols

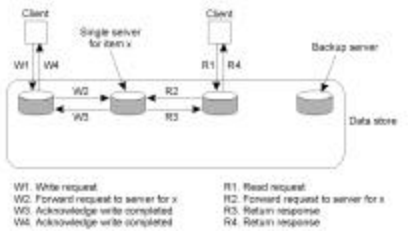
- Used in Bayou system from Xerox PARC
- Bayou: weakly connected replicas
  - Useful in mobile computing (mobile laptops)
  - Useful in wide area distributed databases (weak connectivity)
- Based on theory of epidemics *(spreading infectious diseases)*
  - Upon an update, try to "infect" other replicas as quickly as possible
  - Pair-wise exchange of updates (*like pair-wise spreading of a disease)*
  - Terminology:
    - Infective store: store with an update it is willing to spread
    - Susceptible store: store that is not yet updated
- Many algorithms possible to spread updates

Spreading an Epidemic

- Anti-entropy
  - **Server *P* picks a server *Q* at random and exchanges updates**
  - **Three possibilities: only push, only pull, both push and pull**
  - **Claim: A pure push-based approach does not help spread updates quickly (Why?)**
    - Pull or initial push with pull work better, O(log N)
- Gossiping (Rumor mongering)
  - **Upon receiving an update, *P* tries to push to *Q***
  - **If *Q* already received the update, stop spreading with probability *1/k*, where *k* is a predefined constant**
  - **Analogous to "hot" gossip items => stop spreading if "cold"**
  - **Does not guarantee that all replicas receive updates**
    - Chances of staying susceptible: $s = e^{-(k+1)(1-s)}$

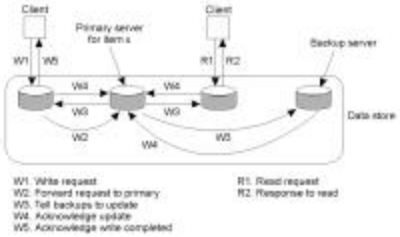
k	1	2	3	4	5
s	0.203188	0.059520	0.019827	0.006977	0.002516

Primary-Based Remote-Write Protocols



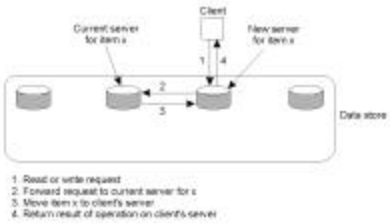
- Primary -based remote-write protocol with a fixed server to which all read and write operations are forwarded. Low efficiency if many read operations involved.

Primary-Backup Remote-Write Protocol



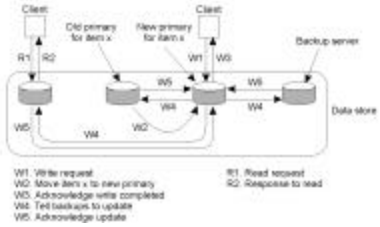
- Read operations are on local copies, where updates must be propagated to backup server and other copies. Problem: long time for a update propagation.

Primary-Based Local-Write Protocols



- Primary -based local-write protocol in which a single copy is migrated between processes. A fully distributed non -replicated version of the data store. Must locate where each data item currently is.

Primary-Backup Local-Write Protocol



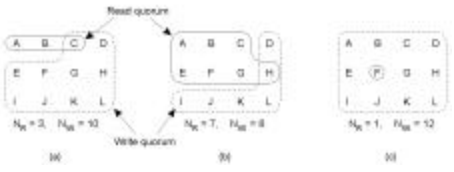
- Primary -backup protocol in which the primary migrates to the process wanting to perform an update. Read local copy, whereas updates must be propagated to all replicas. Applicable to mobile computers .

Quorum-Based Protocol

- Suppose a data item x is replicated on N servers.
- Each server S_i assigns x a voting weight $v_i(x)$.
- Define R(x) as read quorum and W(x) write quorum
- To read x, a client must get enough votes: $\sum_{i} v_i(x) \geq R(x)$
- To write x, a client must satisfy: $\sum_{i} v_i(x) \geq W(x)$
- The value of R(x) and W(x) must follow the following two constraints:
- (1) $R(x) + W(x) > \sum_{i=1..N} v_i(x)$ (prevent read/write conflict)

  (2) $2 * W(x) > \sum_{i=1..N} v_i(x)$ (prevent write/write conflict)

Quorum-Based Protocol



- Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write -write conflicts
c) A correct choice, known as ROWA (Read One, Write All)