Chapter 6

Consistency and Replication

November, 15th

Introduction

Reasons for replication

- Replication for reliability
  - Protection against failures
  - Protection against corrupted data
- Replication for performance
  - *Scaling in numbers*
  - *Scaling in geographical area*

- Scaling in numbers
  - Increasing number of processes need access to data
  - Performance improved by replicating server and dividing work
- Scaling in geographical area
  - Place replicas near processes using them
  - Client may perceive better performance (e.g., lower latency) but...
  - More network bandwidth is needed

The “problem” with replication

- Multiple copies may lead to consistency problems
- When and how to perform data modifications?
- Example: improving access times to Web pages
  - fetching a page from a remote Web server may take long
  - one solution is to locally cache recently fetched pages
  - next access can be served very efficiently, but...
  - client may not have most up-to-date page, or...
  - server updates cached copies, with a cost

Replication as scaling technique

- Replication and caching for performance are widely used
- How to solve the “staleness vs. bandwidth trade-off”?
  - keeping replicas up-to-date requires more network bandwidth
  - process P accesses local replica N times per second
  - replica is (fully) updated M times per second
  - if $N \ll M$ then replicating at P may not be a good idea
- The real problem is how to synchronize replica accesses
  - e.g., deciding on the order to execute reads & writes

The replication dilemma

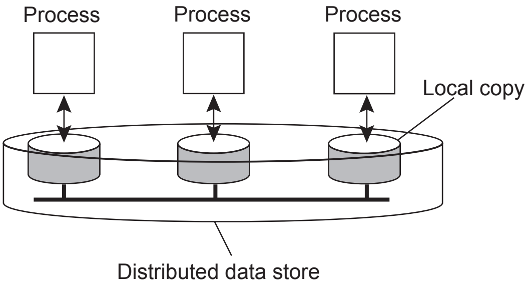
- Scalability problems can be alleviated with replication
- Keeping replicas consistent usually requires synchronization
- Synchronizing replicas is inherently expensive
- The solution may come from “loose consistency”
  - Avoids global synchronization
  - Replica states may diverge

Data-centric consistency models

Introduction

Data store

- Abstraction to reason about consistency
- Read and write operations

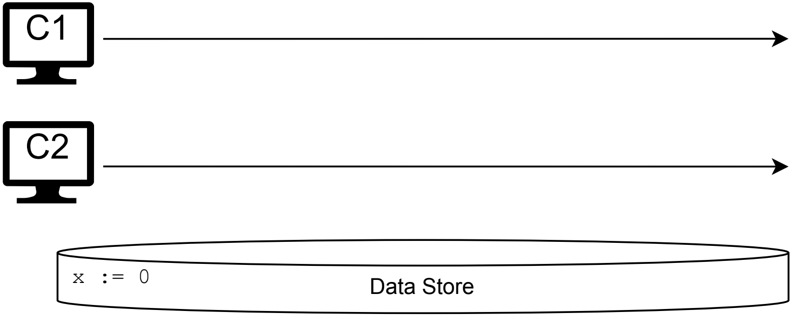


The general organization of a logical data store, physically distributed and replicated across multiple processes.

Introduction

Consistency model

- Contract between processes and data store
- If processes follow certain rules, the store will work “correctly”
- Consistency gives a more precise meaning to “correct”

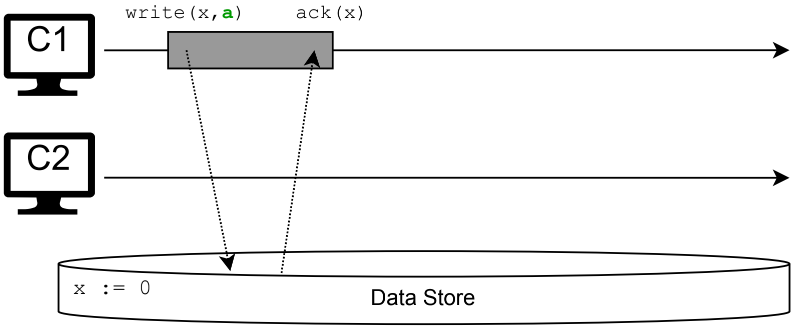


The general organization of a logical data store, physically distributed and replicated across multiple processes.

Introduction

Consistency model

- Contract between processes and data store
- If processes follow certain rules, the store will work “correctly”
- Consistency gives a more precise meaning to “correct”

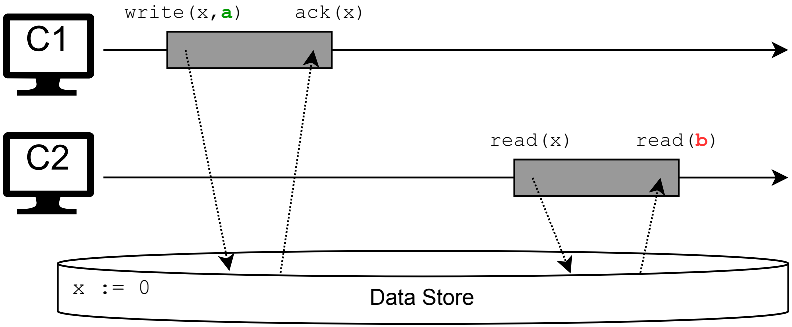


The general organization of a logical data store, physically distributed and replicated across multiple processes.

Introduction

Consistency model

- Contract between processes and data store
- If processes follow certain rules, the store will work “correctly”
- Consistency gives a more precise meaning to “correct”



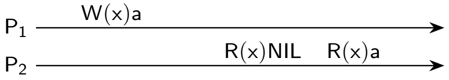
The general organization of a logical data store, physically distributed and replicated across multiple processes.

Consistent ordering of operations

- Important class of consistency models comes from concurrent programming
- Useful in distributed and parallel computing when shared resources are replicated
- These models deal with consistently ordering operations on shared replicated data

Notation

- $W_i(x)a$: Process P_i writes value a to x
- $R_i(x)b$: Process P_i reads value b from x
- Initial values NIL
- Sometimes $_i$ omitted if clear from context



Behavior of two processes operating on the same data item. The horizontal axis is time.

Sequential consistency

Originally defined by Lamport (1979)

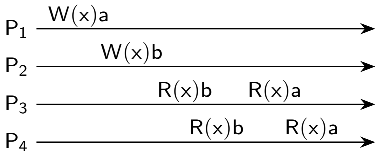
Definition (Sequential consistency)

A data store is sequentially consistent when the result of any execution is the same as if the (read and write) **operations by all processes** on the data store were executed in **some sequential order** and the operations of each individual process appear in this sequence in the **order specified by its program**

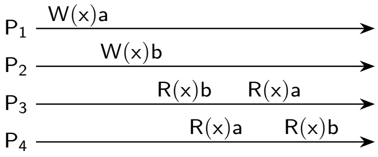
Nothing about time

Sequential consistency

Example that time plays no role



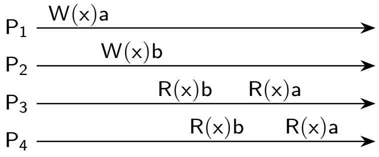
(a)



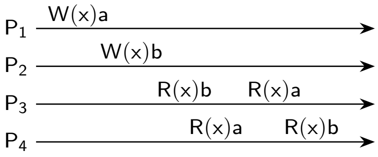
(b)

Sequential consistency

Example that time plays no role



(a)



(b)

(a) A sequentially consistent data store.

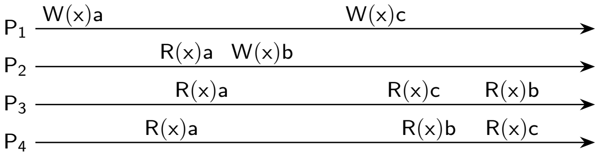
(b) A data store that is not sequentially consistent.

Causal consistency

Definition (Causal consistency)

Writes that are potentially causally related must be seen by all processes in causal order. Concurrent writes may be seen in a different order on different machines

- Writes $W_2(x)b$ and $W_1(x)c$ are concurrent
- Is this execution sequentially consistent?

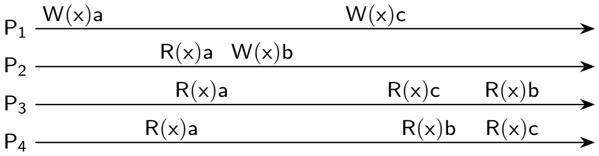


Causal consistency

Definition (Causal consistency)

Writes that are potentially causally related must be seen by all processes in causal order. Concurrent writes may be seen in a different order on different machines

- Writes $W_2(x)b$ and $W_1(x)c$ are concurrent
- Is this execution sequentially consistent?



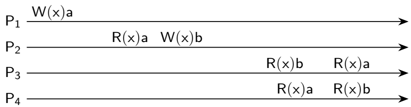
A causally-consistent store.

Causal consistency

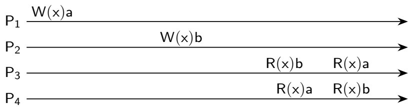
Definition (Causal consistency)

Writes that are potentially causally related must be seen by all processes in causal order. Concurrent writes may be seen in a different order on different machines

- What about the following two executions?



(a)



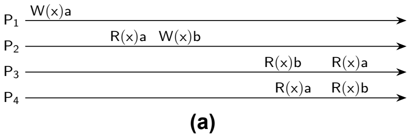
(b)

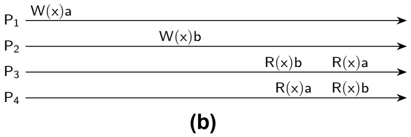
Causal consistency

Definition (Causal consistency)

Writes that are potentially causally related must be seen by all processes in causal order. Concurrent writes may be seen in a different order on different machines

- What about the following two executions?





(a) A violation of a causally-consistent store.

(b) A correct sequence of events in a causally-consistent store.

Sequential vs causal consistency vs serializability

Causal consistency vs sequential consistency?

- Does either imply/is either stronger than the other?

Serializability:

- Transactions (ACID) are sequences of read/write operations

Definition (Serializability)

Execution (possibly interleaved) of operations of a set of transactions is serializable iff there exists an *equivalent* strictly sequential/serialized execution of the transactions (w/o interleaving)

Equivalence: Every read by every transaction gets value written by same write of same transaction (and hence same value)

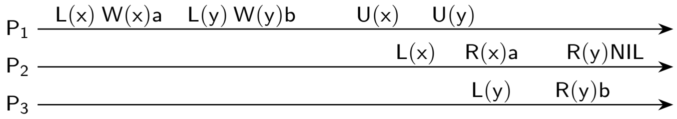
Grouping operations

- Sequential and causal consistency initially developed for shared-memory multiprocessor systems
- Not always a good match for distributed applications
- Synchronization variables
  - To enter a CS, a process must `acquire()` the relevant sync variables
  - To leave a CS, the process `release()`s these variables
- In brief:
  - `acquire()` may not complete (i.e., return to next statement) until all the guarded shared data have been updated
  - before updating shared data a process must enter a critical section (CS) in exclusive mode (no concurrent updates)
  - if a process wants to enter a CS in nonexclusive mode, it must check with the owner of the synchronization variable guarding the CS to fetch the most recent copies of the guarded shared data

Grouping operations

Entry consistency

- Locks are associated with each data item
- Accesses to locks are sequentially consistent



A valid event sequence for entry consistency.

Client-centric consistency models

Eventual consistency

- Special class of distributed data stores
- Workload composed mostly of reads
- Few (if any) concurrent writes (i.e., SWMR[1] model)
- Conflicts solved when needed
- Weak consistency model
- **Client-centric consistency**:
  - guarantees for a single client only;
  - no guarantees for concurrent accesses by different clients

[1] single writer multiple readers

Eventual consistency

Examples

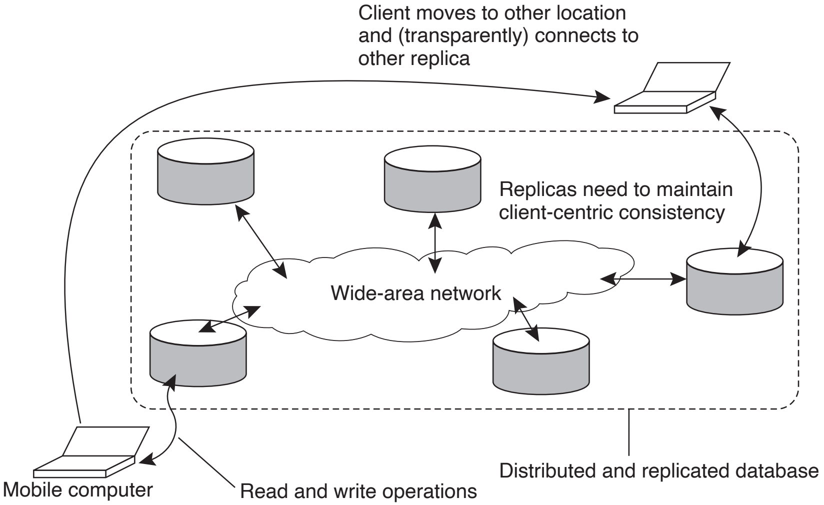
- Databases that are mostly read and updated by one process
- Worldwide naming system (DNS)
  - No write-write conflicts: each domain has it own naming authority, responsible for any updates
  - Read-write conflicts handled in a lazy manner
- Word Wide Web (WWW)
  - No write-write conflicts: updates handled by a single authority
  - Reading stale data is usually acceptable

Eventual consistency

Basic idea

- Distributed and replicated databases that tolerate a relatively high degree of inconsistency
- In the absence of updates, all replicas converge to the same state (and thus the name “eventual consistency”)
- Consequently, usually cheap to implement

Eventual consistency (in Bayou)


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Mobile computer
Read and write operations

The principle of a mobile user accessing different replicas of a distributed database.

Eventual consistency (in Bayou)

- Network connectivity is unreliable and performance poor (e.g., wireless networks, Internet)
- Distributed database: processes may connect to different copies and submit read and write commands
- Updates are eventually propagated to all replicas

Notation:

- $W(x_1)$: sequence of updates to produce x_1
- $W(x_1; x_2)$: sequence of updates to produce x_2, preceded by the updates to produce x_1
- $W(x_1|x_2)$: sequence of updates to produce x_2 in parallel to produce x_1
- $W_j(\ldots)$: sequence performed by p_j

Eventual consistency (in Bayou)

Four types of consistency:

- Monotonic reads
- Monotonic writes
- Read your writes
- Writes follow reads

Monotonic reads

Definition (Monotinic reads)

A data store is said to provide *monotonic-read consistency* if the following condition holds: if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value

In brief: if a process sees a value of x at time t, it will never see an older value of x at a later time $t' > t$.

Example: e-mail system

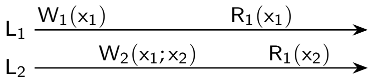
Monotonic reads

Definition (Monotinic reads)

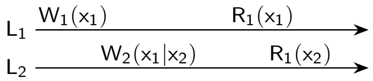
A data store is said to provide *monotonic-read consistency* if the following condition holds: if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value

In brief: if a process sees a value of x at time t, it will never see an older value of x at a later time $t' > t$.

Example: e-mail system



(a)



(b)

The read operations performed by a single process p_1 at two different local copies of the same data store. (a) A monotonic-read consistent data store. (b) A data store that does not provide monotonic reads.

Monotonic writes

Definition (Monotonic writes)

In a *monotonic-write consistent* store, the following condition holds: A write operation by a process on a data item x is completed before any successive write operation on x by the same process (similar to FIFO constraint)

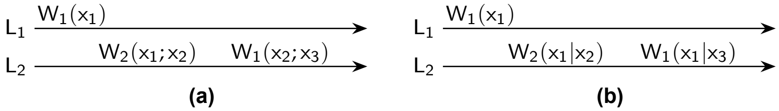
Example: updating a software library

Monotonic writes

Definition (Monotonic writes)

In a *monotonic-write consistent* store, the following condition holds: A write operation by a process on a data item x is completed before any successive write operation on x by the same process (similar to FIFO constraint)

Example: updating a software library



The write operations performed by a single process P at two different local copies of the same data store. (a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency.

Read your writes

Definition (Read your writes)

A data store is said to provide *read-your-writes* consistency, if the following condition holds: the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

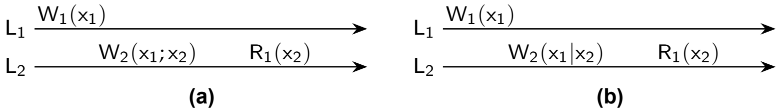
Example (of not reading your writes): updating a web page in a server and later accessing it may result in a previous state to the update due to the caching mechanism implemented by the local browser

Read your writes

Definition (Read your writes)

A data store is said to provide *read-your-writes* consistency, if the following condition holds: the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

Example (of not reading your writes): updating a web page in a server and later accessing it may result in a previous state to the update due to the caching mechanism implemented by the local browser



(a) A data store that provides read-your-writes consistency.

(b) A data store that does not.

Writes follow reads

Definition (Writes follow reads)

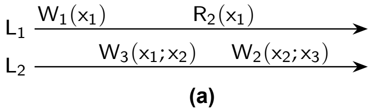
A data store is said to provide *writes-follow-reads* consistency, if the following holds: a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

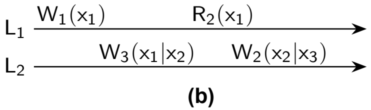
In brief: any successive write by a process p_i on a data item x will be performed on a copy of x that is up to date with the value read by p_i.

Example: Users of a network newsgroup see a posting of a reaction to an article only after they have seen the original article

- A user reads an article A and reacts by posting a response B. With writes-follows-reads, B will be written to any copy of the newsgroup only after A has been written as well

Writes follow reads





(a) A writes-follow-reads consistent data store.

(b) A data store that does not provide writes-follow-reads consistency.

Replica management

Introduction

- Where, when and how replicas should be placed?
- What mechanisms for keeping replicas consistent?
- Placing replica servers
  - Finding best location for a server that can host (part of) a data store
- Placing content
  - Finding best servers for placing content

Replica-server placement

Optimization problem:

- The best K out of N locations need to be selected ($K < N$)
- Usually computationally complex, solved with heuristics
- Example: Take distance (e.g., latency) between all clients and all locations l as start point; select one server at a time such that the average distance between that server's location l and its clients is minimal given the already selected servers:
  - Round 1: take l_1 s.t. average latencies from all clients is minimal
  - Round 2: take 2nd location l_2 s.t. average latencies from all clients to respective closest locations l_i is minimal
  - $\ldots$

Content distribution (update propagation)

State versus operations

Possibilities for what is to be propagated:

- Propagate only a notification of an update
  - invalidate cached copy; bandwidth economical
- Transfer data from one copy to another
  - better with high read-to-write ratio; batched propagation
- Propagate the update operation (vs updated data) to copies
  - “active replication”

Content distribution (update propagation)

Push versus pull protocols

- Pushed-based model (server-based protocol)
  - updates propagated without replicas asking for them
  - normally used for high degree of consistency
- Pull-based approach (client-based protocol)
  - server/client asks another server for updates
  - normally used to update caches (i.e., client polls server to see whether an update is needed)

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

A comparison between push-based and pull-based protocols in the case of multiple-client, single-server systems.

Content distribution (update propagation)

Unicasting versus multicasting

- Unicasting communication
  - Updating N servers implies sending the update N times
  - More suitable for pull-based propagation
- Multicasting communication
  - Underlying network takes care of propagating updates
  - Very effective if replicas are in the same LAN
  - More suitable for push-based propagation

Consistency protocols

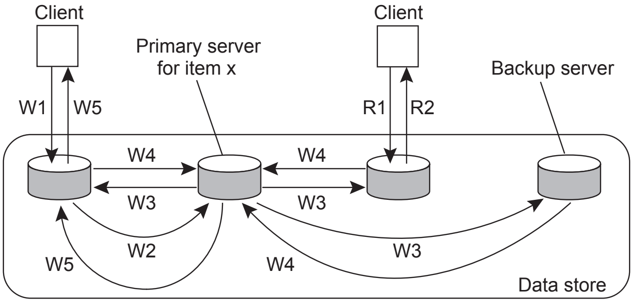
Primary-based protocols

- Each data item x in the data store has a primary, responsible for coordinating write operations on x
  e.g., used to implement sequential consistency
- Two main classes:
  - remote-write protocols
  - local-write protocols

Primary-based protocols

Remote-write protocols (primary-backup)

Write operations are forwarded to a fixed single server



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

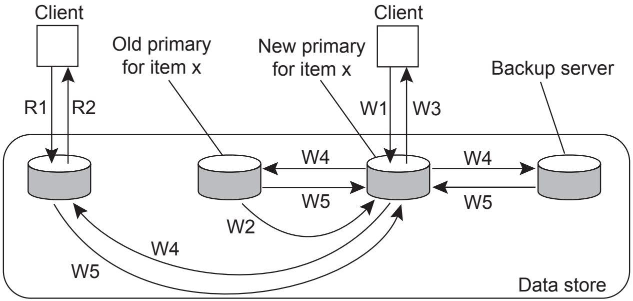
R1. Read request
R2. Response to read

The principle of a primary-backup protocol.

Primary-based protocols

Local-write protocols

Primary migrates between servers



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Primary-backup protocol in which the primary migrates to the process wanting to perform an update.

Replicated-write protocols

- Writes are executed at multiple replicas (not only primary)
- Two classes:
  - Active replication
  - Quorum-based protocols

Replicated-write protocols

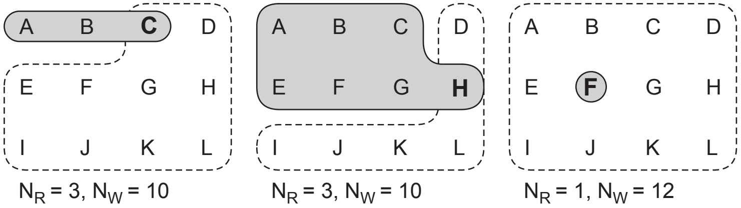
Active replication

- Each replica receives and executes all commands
- Commands need be executed in the same order by all replicas
- Needs mechanism to totally order commands
    - Lamport’s total order protocol
    - Sequencer-based mechanism

Replicated-write protocols

Quorum-based protocols

- Clients request and acquire permission of multiple servers before reading and writing a replicated data item
- Read quorum N_R and write quorum N_W
  - $N_R + N_W > N$
  - $N_W > N/2$



(a) **(b)** **(c)**

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).