Distributed Systems

(3rd Edition)

Chapter 07: Consistency & Replication

Version: February 25, 2017

Performance and scalability

Main issue

To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the the same order everywhere

Conflicting operations: From the world of transactions

- Read–write conflict: a read operation and a write operation act concurrently
- Write–write conflict: two concurrent write operations

Issue

Guaranteeing global ordering on conflicting operations may be a costly operation, downgrading scalability Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

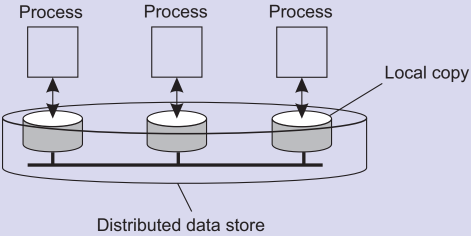
Data-centric consistency models

Consistency model

A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

Essential

A data store is a distributed collection of storages:



Continuous Consistency

We can actually talk about a degree of consistency

- replicas may differ in their numerical value
- replicas may differ in their relative staleness
- there may be differences with respect to (number and order) of performed update operations

Conit

Consistency unit $\Rightarrow$ specifies the data unit over which consistency is to be measured.

Example: Conit

Replica A

Conit:
d = 558 // distance
g = 95 // gas
p = 78 // price

Operation		Result
< 5, B>	g ← g + 45	[g = 45]
< 8, A>	g ← g + 50	[g = 95]
< 9, A>	p ← p + 78	[p = 78]
<10, A>	d ← d + 558	[d = 558]

Vector clock A = (11, 5)
Order deviation = 3
Numerical deviation = (2, 482)

Replica B

Conit:
d = 412 // distance
g = 45 // gas
p = 70 // price

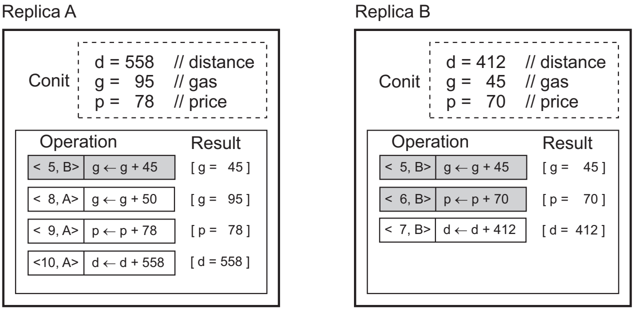
Operation		Result
< 5, B>	g ← g + 45	[g = 45]
< 6, B>	p ← p + 70	[p = 70]
< 7, B>	d ← d + 412	[d = 412]

Vector clock B = (0, 8)
Order deviation = 1
Numerical deviation = (3, 686)

Conit (contains the variables *g*, *p*, and *d*)

- Each replica has a vector clock: ([known] time @ A, [known] time @ B)
- *B* sends *A* operation [$\langle 5, B\rangle : g \leftarrow d + 45$]; *A* has made this operation permanent (cannot be rolled back)

Example: Conit



Vector clock A = (11, 5)
Order deviation = 3
Numerical deviation = (2, 482)

Vector clock B = (0, 8)
Order deviation = 1
Numerical deviation = (3, 686)

Conit (contains the variables *g*, *p*, and *d*)

- *A* has three pending operations ⇒ order deviation = 3
- *A* missed two operations from *B*; max diff is 70 + 412 units ⇒ $(2, 482)$

Sequential consistency

Definition

The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

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

P1:	W(x)a			
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)b	R(x)a

(a)

P1:	W(x)a			
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)a	R(x)b

(b)

Causal consistency

Definition

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

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

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Grouping operations

Definition

- Accesses to locks are sequentially consistent.
- No access to a lock is allowed to be performed until all previous writes have completed everywhere.
- No data access is allowed to be performed until all previous accesses to locks have been performed.

Basic idea

You don't care that reads and writes of a series of operations are immediately known to other processes. You just want the effect of the series itself to be known.

Grouping operations

A valid event sequence for entry consistency

P1:	L(x)	W(x)a	L(y)	W(y)b	U(x)	U(y)		
P2:						L(x)	R(x)a	R(y) NIL
P3:							L(y)	R(y)b

Observation

Entry consistency implies that we need to lock and unlock data (implicitly or not).

Question

What would be a convenient way of making this consistency more or less transparent to programmers?

Consistency for mobile users

Example

Consider a distributed database to which you have access through your notebook. Assume your notebook acts as a front end to the database.

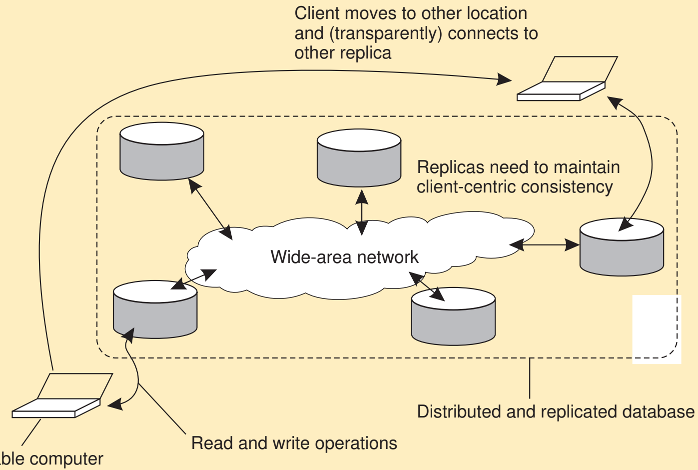
- At location *A* you access the database doing reads and updates.
- At location *B* you continue your work, but unless you access the same server as the one at location *A*, you may detect inconsistencies:
  - your updates at *A* may not have yet been propagated to *B*
  - you may be reading newer entries than the ones available at *A*
  - your updates at *B* may eventually conflict with those at *A*

Note

The only thing you really want is that the entries you updated and/or read at *A*, are in *B* the way you left them in *A*. In that case, the database will appear to be consistent to you.

Basic architecture

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

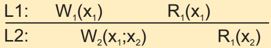


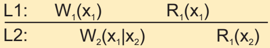
Monotonic reads

Definition

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 or a more recent value.

The read operations performed by a single process *P* 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





Client-centric consistency: notation

Notation

- $W_1(x_2)$ is the write operation by process P_1 that leads to version x_2 of x
- $W_1(x_i; x_j)$ indicates P_1 produces version x_j based on a previous version x_i.
- $W_1(x_i|x_j)$ indicates P_1 produces version x_j concurrently to version x_i.

Monotonic reads

Example

Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Example

Reading (not modifying) incoming mail while you are on the move. Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.

Monotonic writes

Definition

A write operation by a process on a data item *x* is completed before any successive write operation on *x* by the same process.

(a) A monotonic-write consistent data store. (b) A data store that does not provide monotonic-write consistency. (c) Again, no consistency as $WS(x_1|x2)$ and thus also $WS(x_1|x_3)$. (d) Consistent as $WS(x_1;x_3)$ although x_1 has apparently overwritten x_2.

L1: $W_1(x_1)$
L2: $W_2(x_1;x_2)$ $W_1(x_2;x_3)$

(a)

L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_1|x_3)$

(b)

L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_2;x_3)$

(c)

L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $W_1(x_1;x_3)$

(d)

Monotonic writes

Example

Updating a program at server S_2, and ensuring that all components on which compilation and linking depends, are also placed at S_2.

Example

Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

Read your writes

Definition

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.

(a) A data store that provides read-your-writes consistency. (b) A data store that does not.

L1: $W_1(x_1)$

L2: $W_2(x_1;x_2)$ $R_1(x_2)$

(a)

L1: $W_1(x_1)$

L2: $W_2(x_1|x_2)$ $R_1(x_2)$

(b)

Read your writes

Definition

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.

(a) A data store that provides read-your-writes consistency. (b) A data store that does not.

L1: $W_1(x_1)$
L2: $W_2(x_1;x_2)$ $R_1(x_2)$

(a)

L1: $W_1(x_1)$
L2: $W_2(x_1|x_2)$ $R_1(x_2)$

(b)

Example

Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.

Writes follow reads

Definition

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.

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

L1: $W_1(x_1)$ $R_2(x_1)$

L2: $W_3(x_1;x_2)$ $W_2(x_2;x_3)$

(a)

L1: $W_1(x_1)$ $R_2(x_1)$

L2: $W_3(x_1|x_2)$ $W_2(x_1|x_3)$

(b)

Example

See reactions to posted articles only if you have the original posting (a read “pulls in” the corresponding write operation).

Replica placement

Essence

Figure out what the best K places are out of N possible locations.

Replica placement

Essence

Figure out what the best K places are out of N possible locations.

- Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.

Replica placement

Essence

Figure out what the best K places are out of N possible locations.

- Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.
- Select the K-th largest autonomous system and place a server at the best-connected host. Computationally expensive.

Replica placement

Essence

Figure out what the best K places are out of N possible locations.

- Select best location out of $N - K$ for which the average distance to clients is minimal. Then choose the next best server. (Note: The first chosen location minimizes the average distance to all clients.) Computationally expensive.
- Select the K-th largest autonomous system and place a server at the best-connected host. Computationally expensive.
- Position nodes in a d-dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one. Computationally cheap.

Content replication

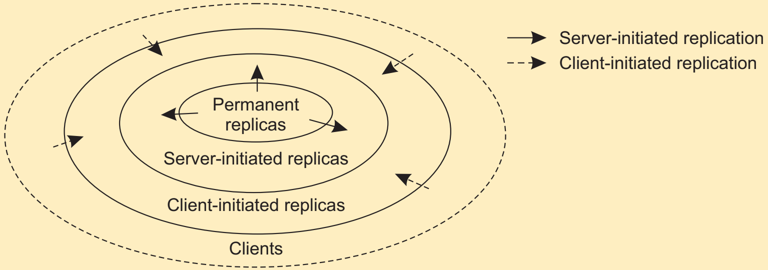
Distinguish different processes

A process is capable of hosting a replica of an object or data:

- Permanent replicas: Process/machine always having a replica
- Server-initiated replica: Process that can dynamically host a replica on request of another server in the data store
- Client-initiated replica: Process that can dynamically host a replica on request of a client (client cache)

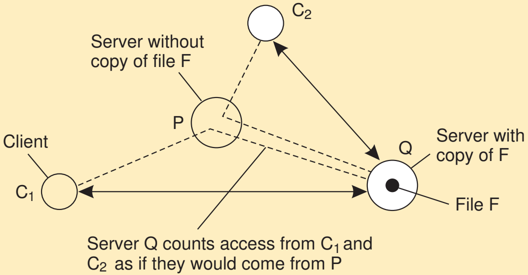
Content replication

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



- Keep track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below threshold $D \Rightarrow$ drop file
- Number of accesses exceeds threshold $R \Rightarrow$ replicate file
- Number of access between D and $R \Rightarrow$ migrate file

Content distribution

Consider only a client-server combination

- Propagate only notification/invalidation of update (often used for caches)
- Transfer data from one copy to another (distributed databases): passive replication
- Propagate the update operation to other copies: active replication

Note

No single approach is the best, but depends highly on available bandwidth and read-to-write ratio at replicas.

Content distribution: client/server system

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

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it.
- Pulling updates: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
1:	List of client caches	None
2:	Update (and possibly fetch update)	Poll and update
3:	Immediate (or fetch-update time)	Fetch-update time

1: State at server

2: Messages to be exchanged

3: Response time at the client

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system's behavior (adaptive leases)

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system's behavior (adaptive leases)

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system's behavior (adaptive leases)

- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system's behavior (adaptive leases)

- State-based leases: The more loaded a server is, the shorter the expiration times become

Content distribution

Observation

We can dynamically switch between pulling and pushing using leases: A contract in which the server promises to push updates to the client until the lease expires.

Make lease expiration time dependent on system's behavior (adaptive leases)

- Age-based leases: An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- Renewal-frequency based leases: The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- State-based leases: The more loaded a server is, the shorter the expiration times become

Question

Why are we doing all this?

Continuous consistency: Numerical errors

Principal operation

- Every server S_i has a log, denoted as L_i.
- Consider a data item x and let $val(W)$ denote the numerical change in its value after a write operation W. Assume that

$$\forall W : val(W) > 0$$

- W is initially forwarded to one of the N replicas, denoted as $origin(W)$. $TW[i,j]$ are the writes executed by server S_i that originated from S_j:

$$TW[i,j] = \sum\{val(W) | origin(W) = S_j \ \& \ W \in L_i\}$$

Continuous consistency: Numerical errors

Note

Actual value $v(t)$ of x:

$$v(t) = v_{init} + \sum_{k=1}^{N} TW[k,k]$$

value v_i of x at server S_i:

$$v_i = v_{init} + \sum_{k=1}^{N} TW[i,k]$$

Continuous consistency: Numerical errors

Problem

We need to ensure that $v(t) - v_i < \delta_i$ for every server S_i.

Continuous consistency: Numerical errors

Problem

We need to ensure that $v(t) - v_i < \delta_i$ for every server S_i.

Approach

Let every server S_k maintain a view $TW_k[i,j]$ of what it believes is the value of $TW[i,j]$. This information can be gossiped when an update is propagated.

Continuous consistency: Numerical errors

Problem

We need to ensure that $v(t) - v_i < \delta_i$ for every server S_i.

Approach

Let every server S_k maintain a view $TW_k[i,j]$ of what it believes is the value of $TW[i,j]$. This information can be gossiped when an update is propagated.

Note

$$0 \leq TW_k[i,j] \leq TW[i,j] \leq TW[j,j]$$

Continuous consistency: Numerical errors

Solution

S_k sends operations from its log to S_i when it sees that $TW_k[i,k]$ is getting too far from $TW[k,k]$, in particular, when

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

Continuous consistency: Numerical errors

Solution

S_k sends operations from its log to S_i when it sees that $TW_k[i,k]$ is getting too far from $TW[k,k]$, in particular, when

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

Question

To what extent are we being pessimistic here: where does $\delta_i/(N-1)$ come from?

Continuous consistency: Numerical errors

Solution

S_k sends operations from its log to S_i when it sees that $TW_k[i,k]$ is getting too far from $TW[k,k]$, in particular, when

$$TW[k,k] - TW_k[i,k] > \delta_i/(N-1)$$

Question

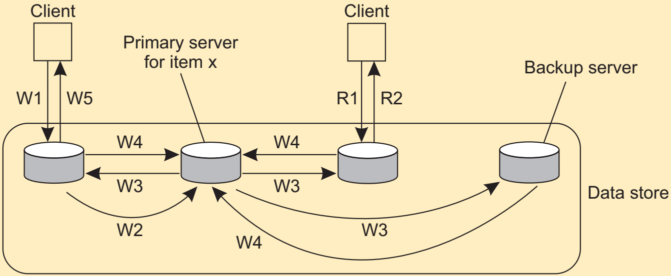
To what extent are we being pessimistic here: where does $\delta_i/(N-1)$ come from?

Note

Staleness can be done analogously, by essentially keeping track of what has been seen last from S_i (see book).

Primary-based protocols

Primary-backup protocol

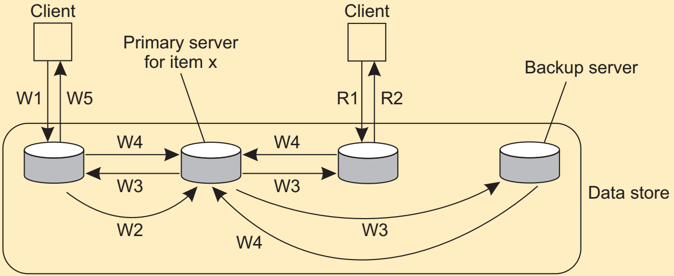


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Primary-based protocols

Primary-backup protocol



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

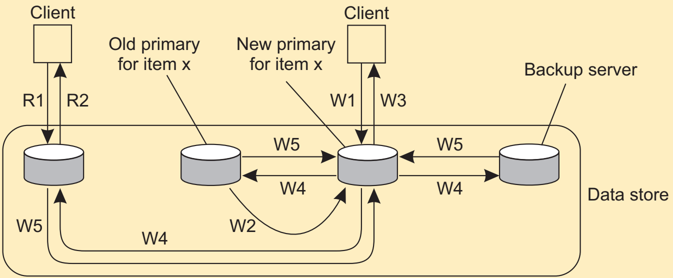
R1. Read request
R2. Response to read

Example primary-backup protocol

Traditionally applied in distributed databases and file systems that require a high degree of fault tolerance. Replicas are often placed on same LAN.

Primary-based protocols

Primary-backup protocol with local writes

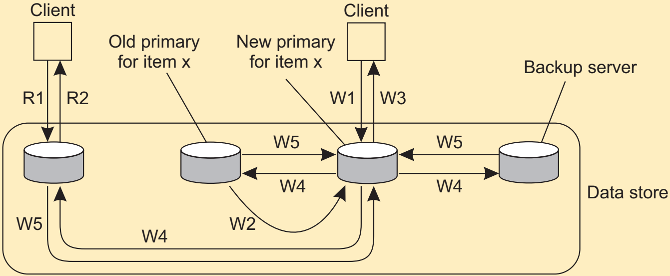


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Primary-based protocols

Primary-backup protocol with local writes



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

Example primary-backup protocol with local writes

Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later on).

Replicated-write protocols

Quorum-based protocols

Ensure that each operation is carried out in such a way that a majority vote is established: distinguish read quorum and write quorum

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)