Consistency and Replication

Topics to be covered

- Introduction
- Consistency Models
- Distribution Protocols
- Consistency Protocols

Introduction

+ **Performance**

+ **Reliability**

Availability: proportion of time for which a service is accessible

- Delays due to pessimistic cc
- Server failures
- Network partitions and disconnected operation

N servers, independent probability p of failing

1- p(all failed or unreachable) = $1 - p^n$

p = 5%, n = 2, availability = 99.75%

Fault tolerance: guarantees strictly correct behavior despite a certain type and number of faults

(correct: e.g., freshness, timeliness of response)

Up to f of f+1 crash

Up to f byzantine failures, 2f + 1

Requirements

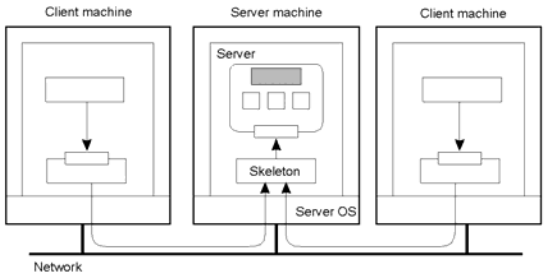
Replication Transparency

One logical object (data item) – many physical copies

Consistency Problems: keep replica consistent – in general, ensure that all *conflicting operations* (e.g., from the world of transactions: RW, and WW) are executed in the same order everywhere

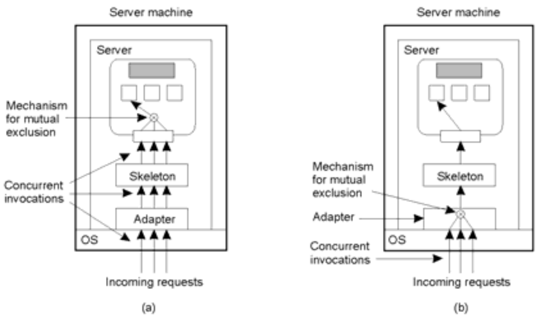
Guaranteeing global ordering costly operation, downgrade scalability

Weaken consistency requirements



Organization of a distributed remote object shared by two different clients.

Object Replication

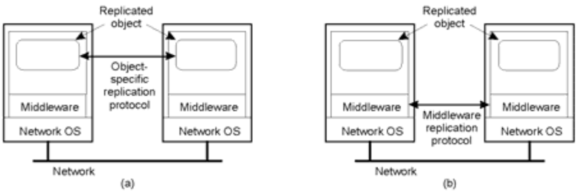


a) A remote object capable of handling concurrent invocations on its own.

b) A remote object for which an object adapter is required to handle concurrent invocations

---

Object Replication



a) A distributed system for replication-aware distributed objects.

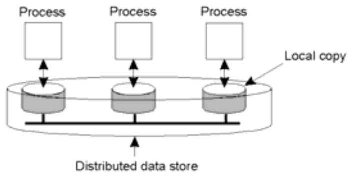
b) A distributed system responsible for replica management

Consistency Models

Data-Centric
Client-Centric

Data-Centric Consistency Models

Consistency Model: a contract between a (distributed) data store and processes, in which the data store specifies precisely what the result of read and write operations are in the presence of concurrency



Notation:

$W_i(x)a$ $R_i(x)b$

Data-Centric Consistency Models

There are no explicit synchronization operations

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	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

There are **explicit synchronization operations** – updates are propagated only when such operations are used

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.

Data-Centric Consistency Models

Strict Consistency

Any read on a data item x returns a value corresponding to the results of the **most recent write** on x

Absolute global time

Note: strict consistency is what we get in the normal sequential case, when each program does not interfere with any other program

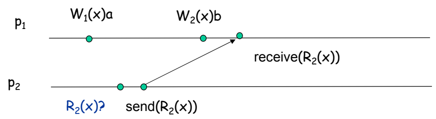
Example

(a) A strictly consistent store (b) A store that is not strictly consistent.

Strict Consistency

Problem: It relies on absolute global time

Example



If strict, p_2 should read the value a

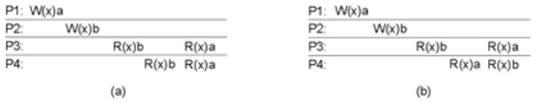
All writes are instantaneously visible to all processes and an absolute global time order is maintained

---

Sequential Consistency

The result of any execution is the same as if the (read and write) operations by all processes on the data store are executed in the **same sequential order** and the operations of each individual process appear in this sequence in the order specified by its program.

Example



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

- Note: a process sees writes from all processes but only its own reads
- Similar with (transaction) serializability but difference in granularity (transactions vs single read and write operations)

*All processes see the **same** interleaving of operations.*

Serializability for replicated data

- x logical data item
- $x_1, x_2, \ldots, x_n$ physical data items

- Replica control protocols: maps each read/write on a logical data item x to a read/write on one (or more) of the physical data items

- One-copy serializability (*equivalence* with a serial execution on an one-copy database – view equivalence same reads-from and same set of final writes)

(assumption: unique reads-from relationships on data items)

A consistency model between strict consistency and sequential consistency that uses *loosely synchronized clocks*.

In particular, we assume that operations receive a *timestamp* using a loosely synchronized clock (a finite precision global clock)

Notation: $ts_{OP}(x)$ where OP = R or W

Linearizability

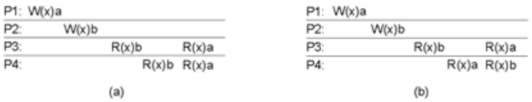
The result of any execution is the same as if the (read and write) operations by all processes on the data store ere executed in *some sequential order* and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if $ts_{OP1}(x) < ts_{OP2}(y)$ then operation OP1(X) should precede OP2(y) in this sequence.

- A linearizable data store is also sequentially consistent.
- The additional requirements of ordering according to timestamps makes it more expensive to implement

---

Ways to express consistency

Consider an associated history (execution)



H1: $W_1(x)a$

H2: $W_2(X)b$

H3: $R_3(x)b\ R_3(x)a$

H4: $R_4(x)R_4(x)a$

Merge individual histories to get the execution history H

$W_1(x)a\ W_2(x)b\ R_3(x)b\ R_4(x)b\ R_3(x)a\ R_4(x)a$

Legal history H, if

Rules:

1. Present program order (order of individual histories)
2. A read to x must always return the value most recently written to x (data coherency)

Data coherency: a R(x) must return the value most recently written to x; that is, the value written by the W(x) immediately preceding it in H

Coherence examines each data item in isolation

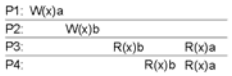
Called memory coherence when dealing with memory locations instead of data items

Sequential Consistency (2nd definition)

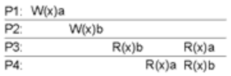
All legal values for history H must:

(i) Maintain the program order

(ii) Data coherency must be respected



(a)



(b)

$W_1(x)a$ $W_2(x)b$ $R_3(x)b$ $R_4(x)b$ $R_3(x)a$ $R_4(x)a$

$W_1(x)a$ $W_2(x)b$ $R_3(x)b$ $R_4(x)a$ $R_3(x)a$ $R_4(x)b$

Legal history:

$W_2(x)b$ $R_3(x)b$ $R_4(x)b$ $W_1(x)a$ $R_3(x)a$ $R_4(x)a$

No legal history

It has been proved that: for any sequentially consistent store, changing the protocol to improve read performance makes write performance worse and vice versa.

Sequential Consistency

Example

Assume the following three concurrently executing processes (assign = write and print = read). Assume, initially x = y = z = 0

Process P1	**Process P2**	**Process P3**
x = 1; print (y, z);	y = 1; print (x, z);	z = 1; print (x, y);

- How many interleaved executions are possible?

  With 6 statements: 6! = 720

- How many of them are valid, i.e., do not violate program order?

  90 (why?)

Sequential Consistency

Example: Four valid execution sequences for the processes of the previous slide.

x = 1;
print ((y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
	x = 1;
	y = 1;
	print (x,z);
	print(y, z);
	z = 1;
	print (x, y);
		y = 1;
		z = 1;
		print (x, y);
		print (x, z);
		x = 1;
		print (y, z);
			y = 1;
			x = 1;
			z = 1;
			print (x, z);
			print (y, z);
			print (x, y);

Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111

Signature: output of P1 Output of P2 output of P3 – 64 different signatures, valid ones?

90 different valid statement orderings produce a variety of different signatures

---

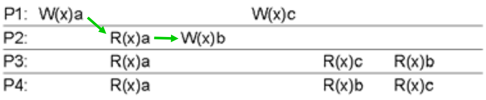
No need to preserve the order of non-related (that is, of concurrent) events (= writes in our case, since reads depend on previous writes)

Casual relation = related say by happened-before

Casual Consistency

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

Example:

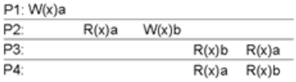


sequence allowed with a casually-consistent store, but not with sequentially or strictly consistent store, assume $W_2(x)b$ and $W_1(x)c$ are concurrent

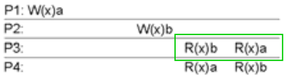
---

Casual Consistency

Example:



(a)



(b)

(a) A violation of a casually-consistent store. (b) A correct sequence of events in a casually-consistent store. – assume $W_2(x)b$ depends on $W_1(x)a$

Implementation

Dependency graph: need to know for each operation, the operation it depends on

FIFO Consistency

Writes of a single process are seen by all other processes **in the order** in which they were issued, but writes of different processes may be seen in a different order by different processes.

In other words: There are no guarantees about the order in which different processes see writes, except that two or more writes of the same process must arrive in order (*that is, all writes generated by different processes are concurrent*).

Also called PRAM consistency in the case of distributed shard memory

Pipelined RAM

FIFO Consistency

Example:

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

A valid sequence of events of FIFO consistency but not for casual

Implementation: need just to guarantee that writes from the same process arrive in order, tag writes with (process-id, sequence-number)

Perform writes with the same id based on sequence-number

FIFO Consistency

(a) P1's view	(b) P2's view	(c) P3's view
x = 1;	x = 1;	y = 1;
print (y, z);	y = 1;	print (x, z);
y = 1;	**print(x, z);**	z = 1;
print(x, z);	print (y, z);	**print (x, y);**
z = 1;	z = 1;	x = 1;
print (x, y);	print (x, y);	print (y, z);
Prints: 00	Prints: 10	Prints: 01

Statement execution as seen by the three processes from the previous slide. The statements in bold are the ones that generate the output shown.

FIFO Consistency

Example

Initially, x = y = 0

Process P1	Process P2
x = 1;	y = 1;
if (y == 0) kill (P2);	if (x == 0) kill (P1);

$W1(x)_1$ $R_1(y)0$

$W_2(y)1$ $R_2(x)0$

Strong Consistency Models: Operations on shared data are synchronized:

- Strict consistency (related to time)
- Sequential Consistency (similar to database serializability, what we are used to)
- Causal Consistency (maintains only casual relations)
- FIFO consistency (maintains only individual ordering)

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	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

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.

Weak Consistency

Don't care that the reads and writes of a **series of operations** are immediately known to other processes. Just want the effect of the series itself to be known.

Each process operates on its own local copy of the data store.

Changes are propagated only when an explicit synchronization takes place

A synchronization variable S with one associated operation **synchronize(S)** which synchronizes all local copies of the data store.

When the data store is synchronized al local copies of process P are propagated to the other copies, whereas writes by other processes are brought into P's copies.

Weak Consistency

1. Accesses to synchronization variables with a data store are sequentially consistent. *(All processes see all operations on synchronization variables in the same order)*
2. No operation on a synchronized variable is allowed to be performed until all previous writes are completed everywhere.
3. 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

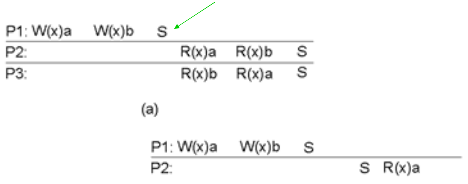
```
int a, b, c, d, e, x, y;           /* variables */
int *p, *q;                        /* pointers */
int f( int *p, int *q);            /* function prototype */

a = x * x;                         /* a stored in register */
b = y * y;                         /* b as well */
c = a*a*a + b*b + a * b;           /* used later */
d = a * a * c;                     /* used later */
p = &a;                            /* p gets address of a */
q = &b                             /* q gets address of b */
e = f(p, q)                        /* function call */
```

A program fragment in which some variables may be kept in registers.

Weak Consistency

Example



Two things: (i) propagate own updates, finish writing shared data – leave CR, (ii) get all other writes, start, enter CR

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

(a)

P1: W(x)a W(x)b S
P2: S R(x)a

(b)

(a) A valid sequence of events for weak consistency. (b) An invalid sequence for weak consistency.

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

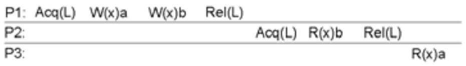
Release Consistency

Divide access to a synchronization variable into two parts: an acquire (for entering a critical region) and a release (for leaving a critical region) phase.

Acquire forces a requestor to wait until the shared data can be accessed. Release sends requestor's local value to other servers in data store.

1. When a process does an acquire, the store will ensure that all the local copies of the protected data are brought up to date
2. When a release is done, protected data that have been changed are propagated to other local copies of the store.

Example



P1: Acq(L) W(x)a W(x)b Rel(L)
P2: Acq(L) R(x)b Rel(L)
P3: R(x)a

A valid event sequence for release consistency.

Release Consistency

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

Entry Consistency

With release consistency, all local updates are propagated to other copies/servers during release of shared data.

With entry consistency: **each shared data item is associated with a synchronization variable**.

When acquiring the synchronization variable, the most recent of its associated shared data are fetched.

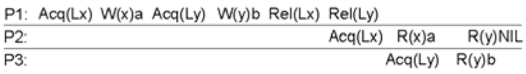
Whereas release consistency affects all data, entry consistency affects only those shared data associated with a synchronization variable.

Entry Consistency

1. Any 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.
2. 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.
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Entry Consistency

Example



A valid event sequence for entry consistency.

Weak Consistency Models: Synchronization occurs only when shared data are locked and unlocked:

- General Weak Consistency
- Release Consistency
- Entry consistency

barriers: synchronization mechanism that prevents any process from starting phase n +1 until all processes have finished phase n

When a process reaches a barrier, it must wait others to get there

When all arrive, data are synchronized, all processes are resumed

---

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	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

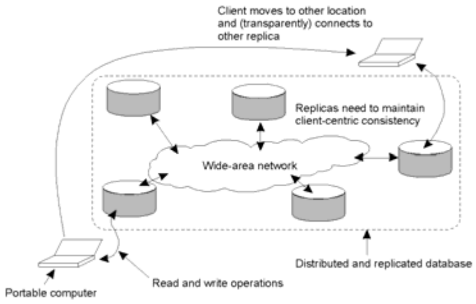
(a)

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.

(b)

Show how we can avoid system-wide consistency, by concentrating on what specific clients want, instead of what should be maintained by the servers.

Eventual consistency: if no updates take place for a long time, all replicas will gradually become consistent



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

Monotonic Reads

If a process reads the value of a data item x, any successive read operation on x by that process will always return the same or a more recent value.

Notation

WS($x_i[t]$): the set of write operations (at site L_i) that lead to version x_i of x (at time t);

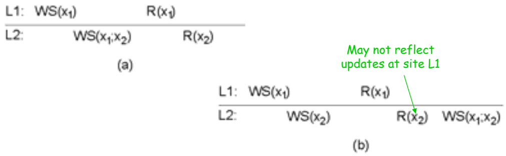
WS($x_i[t_1]$;$x_j[t_2]$) indicates that is known that WS($x_i[t_1]$) is part of WS($x_j[t_2]$)

Example: reading incoming email while on the move; each time you connect: monotonic reads

---

Monotonic Reads

Example



May not reflect updates at site L1

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.

Monotonic Writes

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

Example

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.

Similar to FIFO but for a single process

Monotonic Writes

Examples

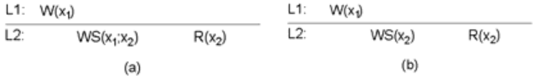
Updating a program at server S2 and ensuring that all components on which compilation and linking depends are also placed at S2

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

The effect of a write operation by a process on data item x will be always seen by a successive read operation on x by the same process.

Example



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

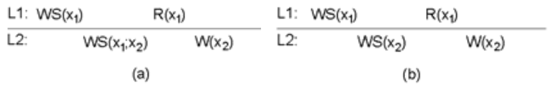
Example: Updating your web page and guaranteeing that your web browser shows the newest version instead of the cached copy.

Similar with changing passwords

Writes Follow Reads

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

Example



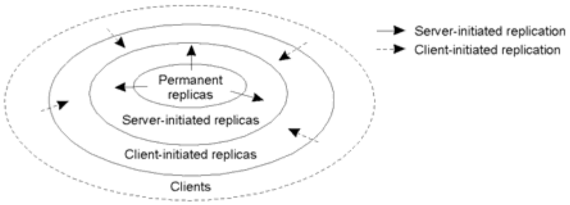
(a) A writes-follow-reads consistent data store (b) A data store that does not provide writes-follow-reads consistency

Example: See reactions to posted articles only if you have the original posting (a read "pulls in" the corresponding write operation)

Distribution Protocols

Replica Placement
Update Propagation
Epidemic Protocols

- Where, when and by whom data copies are placed in a distributed system?



Permanent Replicas

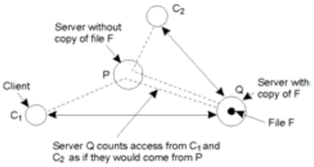
The initial set of replicas that constitute a distributed data store

---

Server-Initiated Replicas

Copies of data to enhance performance, created by the servers

- Keep track of access counts *per file* plus the client that requested the file aggregated by considering server closest to requesting clients



Example, when two clients (C1 and C2 share the same closest server (P)

- Number of accesses drop below threshold D: drop file
- Number of accesses exceeds threshold R: replicate file
- Number of accesses between D and R: file can only be migrated (no drop or replication), when? If the requests from a specific server exceeds half of the total requests

---

Client-Initiated Replicas

Client initiated replicas or (client) caches

Generally kept for a limited amount of time (replaced or become stale)

Cache hit

Share caches among clients

Normally placed at the same machine as the client

State vs Operation

- Propagate *only notification/invalidation* of update
  - Often used for caches
  - Called invalidation protocols
  - Works well when read-to-write ratio is small
- Transfer *values/copies* from one copy to the other
  - Works well when read-to-write ratio is relatively high
  - Log the changes, aggregate updates
- Propagate the *update operation* to other copies (aka active replication)
  - less bandwidth, more processing power

Push vs Pull

Push or server based (update is propagated without a client request)

Pull or client based

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

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

---

A Hybrid Protocol: Leases

Lease: A contract in which the server promises to push updates to the client until the lease expires

Make lease expiration time depended on system behavior (adaptive leases)

- *Age-based leases*: an object that has not changed for 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

---

Overview

Basic idea: assume there are no write-write conflicts (e.g., updates for a specific item are initiated at a single server)

- Update operations are initially performed at one or only a few replicas
- A replica passes its updated state to a limited number of neighbors
- Update propagation is lazy. i.e., not immediate
- Eventually, each update should reach every replica

Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards

Gossiping: A replica that has just been updated (i.e., has been contaminated) tells a number of other replicas about its update (contaminating them as well).

System Model

- A collection of servers, each storing a number of objects
- Each object O has a primary server at which updates for O are initiated
- An update of an object O at server S is timestamped

Notation: timestamp T(O, S), value VAL(O, S)

Infective server/susceptible server

Anti-Entropy

A server S picks another server S* randomly and exchange updates with it

When S contacts S* to exchange state information, three strategies:

PUSH: S only forwards all its updates to S*

if T(O, S*) < T(O, S) then VAL(O, S*) = VAL(O, S)

PULL: S only fetches updates from S*

if T(O, S*) > T(O, S) then VAL(O, S) = VAL(O, S*)

PUSH&PULL: S and S* exchange their updates by pushing an pulling values

If each server randomly chooses another server for exchanging updates, an update is propagated in O(log(N)) time units

Why pushing alone is not efficient when many servers have already been infected?

Gossiping

A server S having an update to report, contacts other servers. If a server is contacted to which the update has already been propagated, *S stops contacting other servers with probability 1/k.*

IF s is the fraction of susceptible servers (i.e., which are unaware of the updates), it can be shown that with many servers:

$$s = e^{-(k+1)(1-s)}$$

k	s
1	0.2
2	0.06
3	0.02
4	0.007
5	0.0025

If we really have to ensure that *all* servers are eventually updated, gossiping alone is not enough.

Deleting Values

We cannot remove an old value from a server and expect the removal to propagate. Why?

Treat removal as a special update by inserting a death certificate

When to remove a death certificate:

- Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
- Assume that death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at the re=isk of not reaching all servers)

It is necessary that a removal actually reaches all servers.

Scalability?

Implementation

Use timestamps and maintain read and write sets

Sessions!

Consistency Protocols

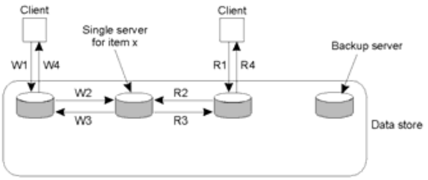
Primary-Based Protocols
Replicated-Write Protocols
Cache-Coherence Protocols

Implementation of a specific consistency model. We will concentrate on sequential consistency.

Primary-based protocols: each data item x has an associated primary responsible for coordinating write operations on x

Remote-Write protocols

Simplest model: no replication, all read and writes operations are forwarded to a single server

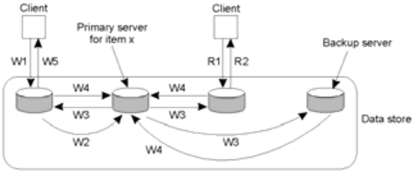


W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

R1. Read request
R2. Forward request to server for x
R3. Return response
R4. Return response

Remote-Write protocols

Primary-backup protocol: reads on local copies, but writes at a (fixed) primary copy



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

An updated is applied as a blocking operation.

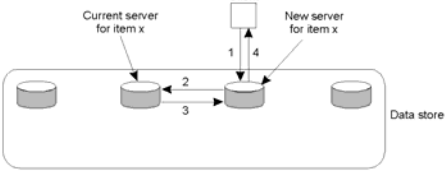
Sequential consistency. Why?

Non-blocking write variant: as soon as the primary has updated its local copy, it returns an ack, then it tells the backup to perform the update as well. Consistency?

Local-Write protocols

Case 1: there is only a single copy of each data item x (no replication) a single copy is migrated between processes

a single copy is migrated between processes

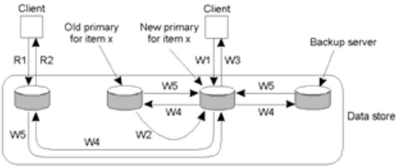


1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

- Useful when writes are expected to come in series from the same client (e.g., mobile computing without replication)

Local-Write protocols

Case 2 (primary back-up): the primary copy migrates



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

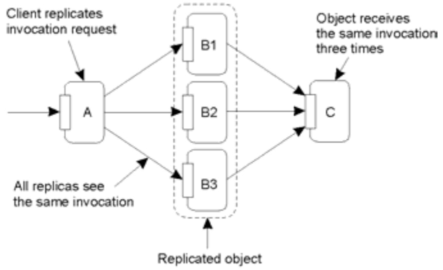
R1. Read request
R2. Response to read

- distributed shared memory systems, but also mobile computing in disconnected mode

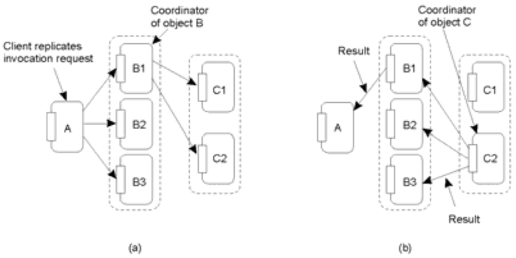
---

Active Replication: updates are propagated to multiple replicas where they are carried out.

The problem of replicated invocations.



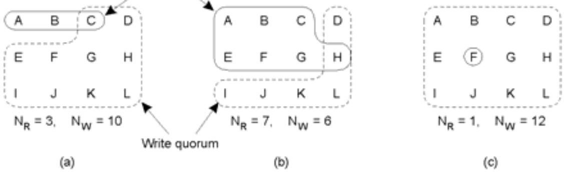
Assign a coordinator on each side (client and server) that ensures that only one invocation and one reply is sent



(a) Forwarding an invocation request from a replicated object. (b) Returning a reply to a replicated object.

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

1. $N_R + N_W > N$
2. $N_W > N/2$



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)

Cache Consistency Protocols

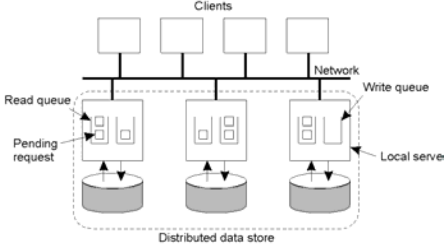
- Write-through caches: clients directly modify the cached data and forward the update to the servers
- Write-back caches: delay the propagation of updates by allowing multiple writes to take place

---

Casually-Consistent Lazy Replication

Number of replica servers jointly implement a causal-consistent data store. Clients normally talk to front ends which maintain data to ensure causal consistency (eventual consistency but also causal relationships between operations)

The general organization of a **distributed data store**. Clients are assumed to also handle consistency-related communication.



Casually-Consistent Lazy Replication

Vector Timestamps

Two vector timestamps per local copy L_i:

VAL(i): VAL(i)[i] denotes the total number of write operations sent directly by a front end (client). VAL(i)[j] denotes the number of updates sent from local copy L_j

WORK(i): WORK(i)[i] total number of write operation directly from front ends, including the pending ones. WORK(i)[j] is the total number of updates from local copy L_j, including pending ones

Each client:

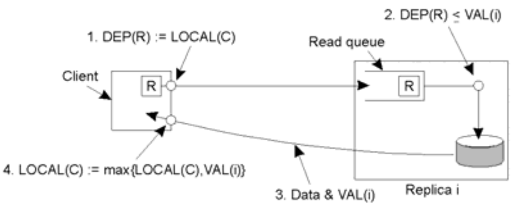
LOCAL(C): LOCAL(C)[j] is (almost) most recent value of VAL(j)[j] known to front end C (will be refined …)

DEP(R): Timestamp associated with a request, reflecting what the request depends on.

Casually-Consistent Lazy Replication

Client C performs R

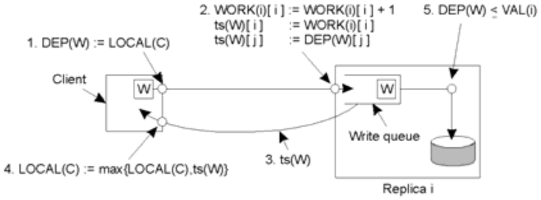
It is necessary that L_i knows about that state



Performing a read operation at a local copy.

Casually-Consistent Lazy Replication

Client C performs W



Performing a write operation at a local copy.