Distributed Systems

Delhi Technological University
Consistency and Replication
Divyashikha Sethia
divyashikha@dce.edu

Introduction

•Data Replication:
- enhance reliability or improve performance
- scalability

•Issue of keeping replicas consistent

•Consistency models

•Implementation of consistency
- placement of replica servers and content distribution to these servers
- updating replicas for consistency

Reasons for Replication

•**Reliability**
- continue working after one replica crashes by simply switching to other replica
- multiple copies provide better protection against corrupted data

•**Performance**
- **scale in numbers** when an increasing number of processes needs to access data that are managed by a single server (replicated web servers)

- **scale in geographical area** - by placing copy of data in proximity of the process using them, the time to access the data decreases (web proxies)

- client performance may increase but more network bandwidth is now consumed keeping all replicas up to date

• **Problems: Consistency**
- modifications have to be carried out on all copies to ensure consistency

Reasons for Replication…

Example fetching a page from a remote Web server to improve performance, Web browsers locally store copy of previously fetched Web page
• Problem: if page has been modified, modifications will not have been propagated to cached copies, making those copies out of-date.

•**Solution:**
- Forbid browser to keep local copies, and server is fully in charge of replication. No replica near user – poor access time
- Web server invalidates or updates each cached copy, by keeping track of all caches and sending them messages. This, may degrade the overall performance of the server.

Replication as Scaling Technique

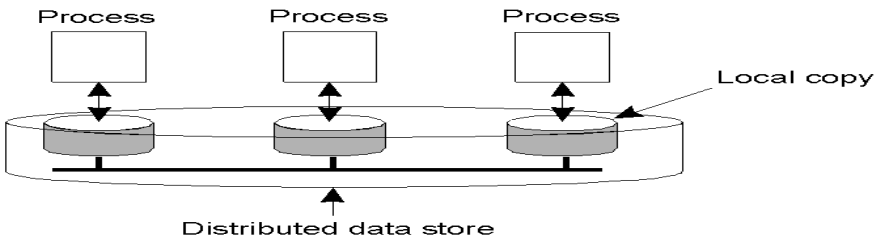
• Replication and caching are widely used for system scalability

• Multiple copies:
  – Improves performance by reducing access latency
  – But higher network overheads of maintaining consistency
  – Example: object is replicated N times
    • Read frequency R, write frequency W
    • If R<<W, high consistency overhead and wasted messages
  – Consistency maintenance is itself an issue

• Tight consistency requires globally synchronized clocks!

• Solution: loosen consistency requirements
  – Variety of consistency models possible based on access and update patterns as well as the application.

Data-Centric Consistency Models



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

Data store: large "set of files" distributed physically across multiple machines

Consistency Model

A **consistency model** is essentially a contract between processes and the data store. It says that if processes agree to obey certain rules, the store promises to work correctly.

-The data store specifies specifies precisely what the results of read and write operations are in the presence of concurrency

- All consistency models attempt to return the result of the last write for a read operation

– Differ in how “last” write is determined/defined

- Models with minor restrictions are easy to use while models with major restrictions are more difficult to use.

Consistency Models

- **Data-centric consistency models**
  - Sequential consistency
  - Causal consistency
  - Grouping operations
- **Client-centric consistency models**
  - Eventual consistency
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads

Strict Consistency

•It is most stringent consistency and is defined by following condition: Any read to a memory location X returns value stored by most recent write operation to X.
•Assumes existence of absolute global time to determine most recent write
•All writes are instantaneously visible to all processes and global time order is maintained.
•Difficult to achieve in real systems due to network delays

Behavior of two processes, operating on the same data item.
(a) A strictly consistent store.
(b) A store that is not strictly consistent.

Sequential Consistency

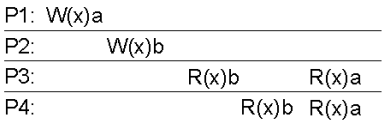
• Slightly weaker model and satisfies following condition:

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.

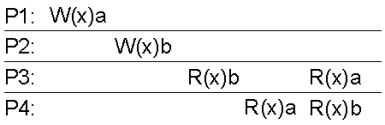
•Assumes all operations are executed in some sequential order and each process issues operations in program order.
– Any valid interleaving is allowed
– All processes see the same interleaving
– Each process preserves its program order
– Nothing is said about "most recent write"

• When processes run concurrently on (possibly) different machines, any valid interleaving of read and write operations is acceptable behavior, but *all processes see the same interleaving of operations.*

Sequential Consistency…



(a)



(b)

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

- Both processes P3 and P4 first read value b, and later value a.

Sequential Consistency…

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

Three concurrently executing processes.

•The data items in this example are formed by the three integer variables x, y, and z, which are stored in a (possibly distributed) shared sequentially consistent data store.
• Assumption: each variable is initialized to 0
•Assignment => write operation, whereas a print statement corresponds to a simultaneous read operation of its two arguments.
•All statements are assumed to be indivisible.

Sequential Consistency…

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

Three concurrently executing processes.

•For 6 statements, there are potentially 720 (6!) possible execution sequences

•Valid sequences are those in which the assignment statements are before the respective print statements.

Sequential Consistency…

```
x = 1;
print(y,z);
print(x,z); // invalid because it violates program
y = 1; // order of process P2.
z = 1;
print(x,y);
```

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

Sequential Consistency…

Common approaches to expression of sequential consistency are:

•Each process Pi has an associated **Execution** Ei, which includes its sequence of reads and writes on the data store.

•It is necessary to merge all Ei into a **history** H with property of:

-Program order is maintained. This means that if an operation OP1 appears before another (called OP2) in Ei, then they must appear in the same relative order in the history H.

- Data coherence is maintained. This looks much like strict consistency: a read R(x) must always return the value most recently written to x.

(Difference:

- coherence examines single data item without regards to other data items

- consistency deals with different data items and their ordering.)

Sequential Consistency…

x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
Signature: 001011 (a)	Signature: 101011 (b)	Signature: 110101 (c)	Signature: 111111 (d)

Signature is string based on the order of prints of processes P1,P2 and P3.
(first instance of print)
Print(y,z)
Print(x,z)
Print(x,y)

Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

-The *signature* is the output of P1, P2, and P3, in that order.

-Usefulness of the signature; specifically, you can use it to determine

-There are actually less than 64 valid program results

Sequential Consistency…

•Not all 64 signature patterns are allowed:

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

Eg1: 000000 is not permitted, since it would imply that print statements ran before assignment statements, violating requirement that statements are executed in program order.
Eg2: 001001. The first two bits, 00, mean that
y and z were both 0 when PI did its printing. This situation occurs only when PI executes both statements before P2 or P3 starts.
The next two bits, 10, mean that P2 must run after P1, has started but before P3 has started.
The last two bits, 01,mean that P3 must complete before P1, starts, but we have already seen that PI must go first.
Hence 001001 is not allowed.

Sequential Consistency…

•Signature used to determine whether given execution sequence is valid.
•There are actually less than 64 valid program results (that's because there are 6 bits here in the output; 2^6 = 64, and not all signatures are valid so there must be less than 64).

•The contract between processes and distributed shared data store is that processes must accept all of these as valid results. (including the four previous results)

• All other valid results are proper answers, and must work correctly if any of them occurs.

•A program that works for some of these results and not for others violates the contract with the data store and is incorrect.

Casual Consistency

- Represents weakening of sequential consistency: makes distinction between events that are potentially causally related and those that are not
- Example of causality:
  - Discussion of relative merits of programming languages:
  - M1: “Anybody caught programming in FOTRAN must be shot”
  - M2: “I am against capital punishments”
  - Due to delays in message propagation, M3 may get M2 first and get confused.
  - M2 was influenced by M1 hence causality is violated.
  - If event B is caused or influenced by earlier event A, causality requires that everyone must see event A before B.

Casual Consistency

•Process P, writes a data item x and then P2 reads x and writes y.

- reading of x and writing of y are potentially causally related since computation of y may have depended on the value of x as read by P2 (i.e., the value written by P1)

```
a = a + 1; // First two are causally related
b = a * 5; // because b reads a after a was written.
c = c * 3; // concurrent statement
```

Casual Consistency..

- Two processes spontaneously and simultaneously write two different data items, *these are not causally related and are said to be concurrent.*

- Necessary condition:
  Writes that are potentially casually related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Casual Consistency…

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

•This sequence is allowed with a casually-consistent store, but not with sequentially or strictly consistent store.

-Not strictly consistent because P3 and P4 aren't executing "R(x)c)" in all cases, but they should.

-Not sequentially consistent because P3 and P4 don't read the same values in the same order.

•Writes W2(x)b and WI (x)c are concurrent, so it is not required that all processes see them in the same order

Casual Consistency…

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)

a) A violation of a casually-consistent store.

Violation of casually-consistent state

- *W(x)b* depends on W*(x)a because b may be result of computation involving* value read by *Rz(x)a.*
- *The two writes are causally related, so all processes* must see them in the same order.

Casual Consistency…

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

(b)

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

- read has been removed, so W*(x)a and W(x)b are now* concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered,
- - reflects a situation that would not be acceptable for a sequentially consistent store since the final values concluded by processes is different

Grouping Operations

•Sequential and casual consistency are defined at the level of read and write operations since they have been developed for share-memory multiprocessor system and are implemented at hardware level.

•Uses granularity of critical sections instead of individual read/write

– Send final result of critical section everywhere
– Intermediate results not propagated

• Entry and release consistency
– Shared data made consistent at entry/exit points of critical sections

• Entry and release consistency are two forms of weak consistency where shared data are made consistent at synchronization time

Group Operations…

•Each synchronization variable has a current owner, process that last acquired it.

•The owner may enter and exit critical sections repeatedly without having to send any messages on the network.

•A process not currently owning a synchronization variable but wanting to acquire has to send message to current owner for ownership and current values associated with synchronization variable.

•Several processes can own synchronization variable in nonexclusive mode (i.e can read, but not write, the associated data.)

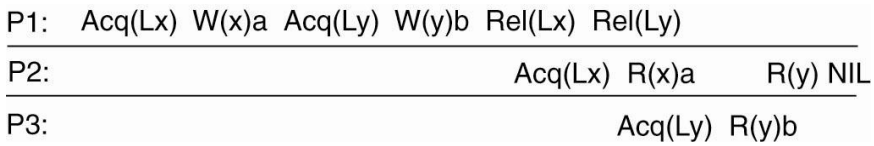
Entry Consistency

1. When process does an acquire, it will not be complete until all the guarded shared data are up to date. At acquire, all remote changes to the guarded data to be made visible.

2. Before updating shared data item, process must enter critical section in exclusive mode to share data exclusively.

3. If process wants to enter critical region in nonexclusive mode, it must first check with the owner of the synchronization variable guarding the critical region to fetch the most recent copies of the guarded shared data.

Entry Consistency



•Associate locks with each data item.

•P1 does an acquire for x, changes x once, after which it also does an acquire for y

•Process P2 does an acquire for x but not for y, so that it will read value a for x, but may read NIL for y.

•Because process P3 first does an acquire for y, it will read the value b when y is released by P1

CLIENT-CENTRIC CONSISTENCY MODELS

•The consistency models described in previous section aim at providing system wide consistent view on a data store
- Assume concurrent processes may be simultaneously updating data store, and that it is necessary to provide consistency for such concurrency

•We look at special class of distributed data stores which are characterized by lack of simultaneous updates, or when such updates happen, they can easily be resolved
- Most operations involve reading data.
- These data stores offer a very weak consistency model, called **eventual consistency.**
- many inconsistencies can be hidden in a relatively cheap way

Eventual Consistency

• Many systems have one or few updaters and many readers
– No write-write conflicts
– How fast should the updates be made available to other read-only processes?
• Examples:
– DNS: single naming authority per domain
  • Only naming authority is allowed updates
– Web:
  • Web pages are updated by a single authority such as a webmaster or owner of the page
- Handle read-write conflicts, in which one process wants to update a data item while another is concurrently attempting to read that item.

Eventual Consistency

• It is acceptable to propagate an update in a *lazy fashion*, meaning that a reading process will see an update only after some time has passed since the update took place

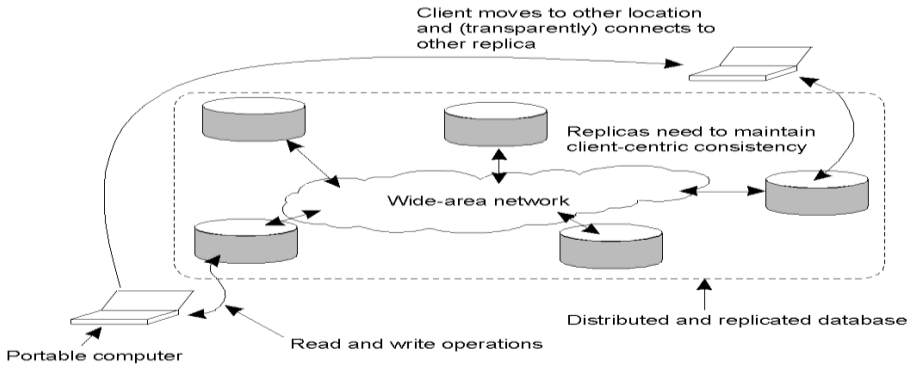
•These distributed and replicated databases tolerate a relatively high degree of inconsistency

•They have in common that if no updates take place for a long time, all replicas will gradually become consistent. This form of consistency is called **eventual consistency.**

• In the absence of updates, all replicas converge toward identical copies of each other

Eventual Consistency

• 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 (use epidemic protocols)
– Things work fine so long as user accesses same replica
–



Principle of mobile user accessing different replicas of distributed database.

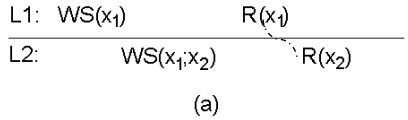
Client Centric Consistency

Client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client

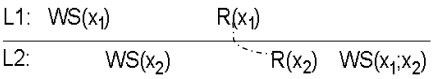
• *Monotonic reads*
– Once read, subsequent reads on that data item return same or more recent values
• *Monotonic writes*
– A write must be propagated to all replicas before a successive write by the same process
• *Read your writes*
– read(x) always returns last write(x) by that process
• *Writes follow reads*
– write(x) following read(x) will take place on same or more recent version of x

Monotonic Reads

Once read, subsequent reads on data item return same/ more recent value



(a)



(b)

The read operations performed by a single process *P* at two different local copies L1 and L2 of the same data store.

a) A monotonic-read consistent data store

- WS(x1;x2) means that set WS(x1) is part of set WS(x2) (i.e., on time line, WS(x1;x2) is a synchronization operation showing that the operations on data x in L1 are now also reflected in L2's version of x).

b) A data store that does not provide monotonic reads.

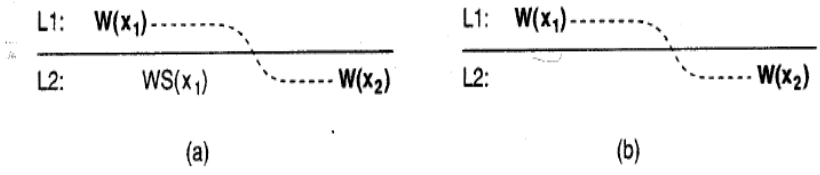
- no guarantee that operations at local store L1 have been propagated local store L2

Monotonic Read..

Ex: a user reads email x1 in New York, and then flies to Toronto, open the copy of email box there, monotonic reads consistency guarantees that x1 will be in the mail box in Toronto.

Monotonic Writes

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



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.

Monotonic Writes…

-Write on a copy of data item x is performed only if that copy has been updated by any preceding write operation that occurred in other copies of the data store

Monotonic Write

Example of a software library : consider a single source code file containing 10 functions implementing some sort of simple library.
-The programmer updates function1 and "checks it in" in Flint.
- Then he/she updates function7 and checks that change in at the Detroit.
- Each of these updates represents a different version of the library, but from the programmer's standpoint, the update to function7 took place after the update to function 1.
- The programmer should observe the changes to function 1 in the source code file he/she works with in Detroit.
- Else, there'll be two different versions of the library in two locations, and there will be no way of reconciling them
-Thus, the second update (to function 7) can only take place after all local copies have seen the first update to function 1.

Monotonic Writes..

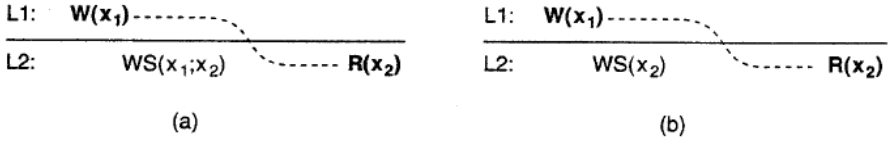
•Monotonic-write consistency applies more appropriately to situations in which the data, x, is partly updated.

•If a single data item is written in on location then a new value of that single data item is written in a different location by the same process, then this problem doesn't really occur

•Consider changing a single integer variable x to 5 and later on change it to 7. The value 5 isn't really that important since it has been *completely overwritten*

Read Your Writes

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.

- Client writes to data x and moves locations to another local copy of the store, then it will see the write that it made earlier.

- Similar to Monotonic Read except that consistency is now determined by the last write operation by process *P, instead of its last read.*

Read Your Writes..

Example updating Web documents and subsequently viewing the effects:

• Update operations on word processor, saves the new version on a file system that is shared by the Web server.

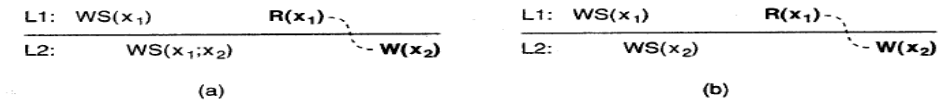
•User's Web browser accesses same file, possibly after requesting it from the local Web server. However, once the file has been fetched, either the server or the browser often caches a local copy for subsequent accesses.

•When the Web page is updated, the user will not see the effects if the browser or the server returns the cached copy instead of the original file.

• Read-your-writes consistency can guarantee that if the editor and browser are integrated into a single program, the cache is invalidated when the page is updated, so that the updated file is fetched and displayed.

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



a) A writes-follow-reads consistent data store

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

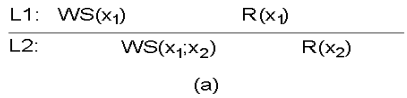
- If a process reads a "most recent" copy of x at local store L1 then moves to some remote store L2 and issues a write to x, then the process *must* use the "most recent" copy it obtained at L1, not a less recent copy that may exist at L2.

Writes Follow Reads

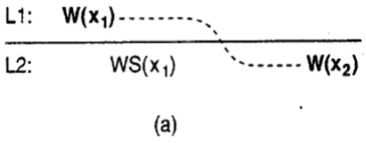
Internet News and Writes Follow Reads in newsgroups:
Comments on news group, let A an article read recently, R the response to that article, then R must follows A.

•user first reads an article A.
•Reacts by posting a response B.
•writes-follow-reads consistency: B will be written to any copy of the newsgroup only after A has been written as well.
•The writes-follows-reads consistency assures that reactions to articles are stored at a local copy only if the original is stored there as well.

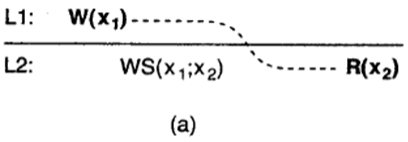
Comparitive



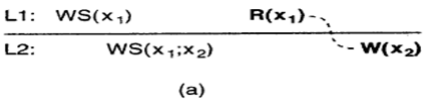
Monotonic Read



Monotonic Writes



Read Your Writes



Writes Follow Reads

Replication

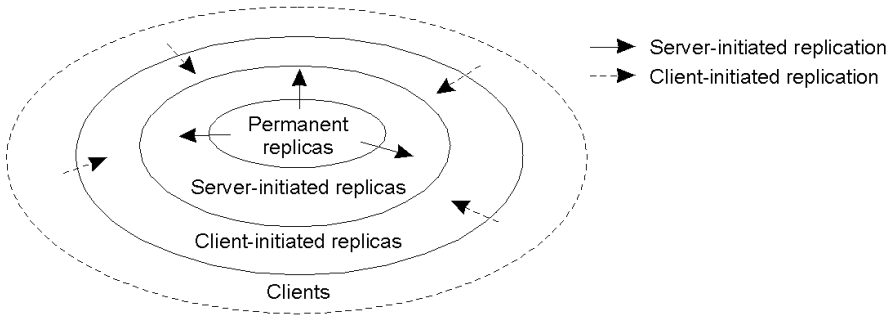
•How to distribute updates to replicas, regardless of how consistency is handled?

•How to propagate updated content to the relevant replica servers?

•How updates are propagated in a distributed data store?

Replica Placement

•How to distribute updates to replicas, regardless of how consistency is handled



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

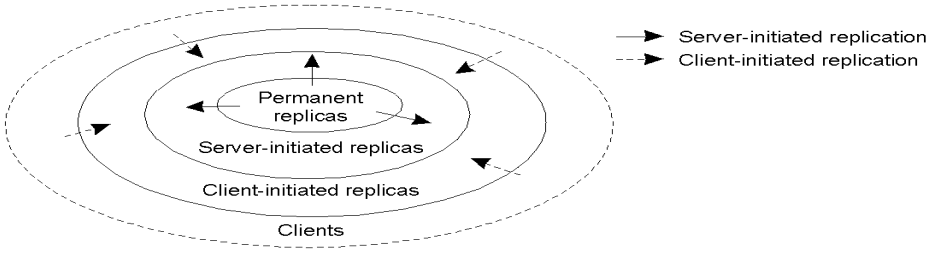
Replica Placement

•**Permanent replicas** are created when the distributed data store is created.

•**Server-initiated replicas** are created to enhance performance, and are created by owner of data store.
- Created in static fashion - "manually" copying data store from one location to another
- Created in dynamic fashion - server initiates copy in response to increased load

•**Client-initiated replicas** are caches that are stored on the client machines or stored on other machines in close proximity to the client.

Replica Placement



• What Client encounters in a data store if all three types of replicas exist:
- It will consult its own cache (client-initiated replica) first.
- That failing, it will consult a server-initiated replica (which, like the cache, may also not contain a complete copy of the data store).
- When the cache and server-initiated replica fail to provide the data, the client can access the permanent replica.

Permanent Replicas

•Initial set of replicas that constitute a distributed data store

•Distribution of Web site:

i) Files for a site replicated across limited number of servers at a location:
- Whenever a request comes in, it is forwarded to one of the servers, for instance, using a round-robin strategy.

ii) Mirroring: a Web site is copied to a limited number of servers, called mirror sites which are geographically spread across the Internet.:
- Clients choose one of the mirror sites from list offered to them.
- There are only a few number of replicas, which are more or less statically configured.

Server Initiated Replicas

•Server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store

•Can Install a number of temporary replicas in regions where requests are coming from

•Dynamically placing replicas by web hosting services which:
- Offer a (relatively static) collection of servers spread across the Internet that can maintain and provide access to Web files belonging to third parties
- Can dynamically replicate files to servers where those files are needed to enhance performance, that is, close to demanding (groups of) clients

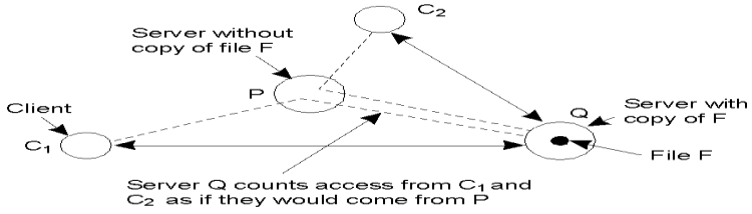
Server Initiated Replicas

Dynamic replication:

- reduces the load on a server

- specific files on a server can be migrated or replicated to servers placed in the proximity of clients that issue many requests for those files

Server-Initiated Replicas (Migrating files in proximity of clients)

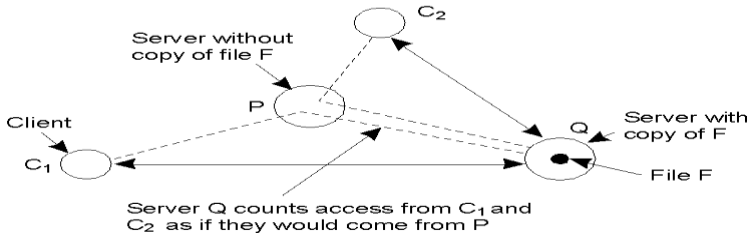


Each server keeps track of access counts per file, and where access requests come from

-For a given a client C, each server can determine which of the servers in the Web hosting service is closest to C

- If client C1 and client C2 share the same "closest" server P, all access requests for file F at server Q from C1 and C2 are jointly registered at Q as a single access count cntQ(P,F).

Server-Initiated Replicas (Migrating files in proximity of clients)



Counting access requests from different clients.

- (1) system maintains two limits: del(S, F) and rep(S, F)
  rep (S,F) > del (S,F)
- (2) if countQ(P, F) > rep(Q, F) , then replicates F on P
- if countQ(P,F) < del(Q,F) then delete replica at Q unless it is the last copy

Server-Initiated Replicas

•The usage of the count works as follows:
•There are two thresholds: a *deletion threshold* and a *replication threshold*. The replication threshold always has a higher value than the deletion threshold (i.e., these are lower and upper limits on the count).

•On a given server hosting a file (server Q in the diagram at the top), the deletion threshold represents a lower limit on access requests. It makes the server remove the file unless it is the last overall copy of the file.

•The replication threshold determines conditions under which the file may be replicated or migrated.

•Requests from clients C1 and C2 are coming independent of server P, but since P is "closest" to them, the server Q counts them as if the requests came from P.

Server Initiated Replicas

Q decides to "reevaluate the placement of files it stores" (it determines whether to move anything), it checks the access counts and applies these rules:
- If count from remote server P > replication threshold, then server P is asked to replicate file.

-If count from some remote server P is above half of all requests but not above the replication threshold, then server P is asked to take the file over (i.e., migration *may* take place).

-Migration or replication may not take place on a remote server if that server is too heavily loaded or simply out of file space. In that case, if the overall count meets either of the above conditions, then server may try to migrate/replicate the file to/on a different server.

- If the overall count drops below the deletion threshold, then the file is removed unless it is the last copy.

Server Initiated Replicas

•Permanent replicas are still often useful as a back-up facility, or to be used as the only replicas that are allowed to be changed to guarantee consistency.

•Server-initiated replicas are then used for placing read only copies close to clients.

Client Initiated Replicas

- Commonly known as (client) caches

- Local storage facility that is used by a client to temporarily store a copy of the data it has just requested

-Client caches are used only to improve access times to data.

-Scheme works fine as long as the fetched data have not been modified in the meantime

- Data are generally kept in a cache for a limited amount of time,

- Placement of client caches is relatively simple: a cache is normally placed on the same machine as its client, or otherwise on a machine shared by clients on the same local-area network

Content Distribution

•How to propagate updated content to the relevant replica servers?

•Three possibilities:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.

Content Distribution – Propagate Notification

•**Propagating a notification similar to invalidation protocols:**

-Other copies informed about the update place and that data they contain is no longer valid.

-May specify which part of data store has been updated, so that only part of copy is invalidated

-Whenever operation on invalidated copy is requested, update copy

• Advantage: use little network bandwidth.

•Works best when many update operations compared to read operations, i.e. read-to-write ratio is small.

Content Distribution – Transferring modified data

•Useful when the read-to-write ratio is relatively high

• Could log changes and transfer only those logs to save bandwidth.

• Transfers are often aggregated - multiple modifications packed into single message to save communication overhead

Content Distribution – Propagate Update Operations

•Tell each replica which update operation at should perform and send only parameter values required by those - **Active Replication**

•Assumes each replica is represented by process capable of "actively" keeping associated data up to date by performing operations

•Updates can be propagated at minimal bandwidth costs, if size of parameters associated with operation are small.

Pull versus Push Protocols

• How updates are propagated in a distributed data store?

- A pull-based approach means that the clients initiate the change ("pulling" the data from the server), so these are also referred to as client-based protocols.
- A push-based approach means that the server actively seeks to make updates to its clients, so it is "pushing" data to them.

Pull versus 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.
•Pull-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 (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.

Consistency Protocol

A consistency protocol describes an implementation of a specific consistency model.

1. Primary Based Protocols
2. Active Replications

Consistency Protocol: Primary Based

•In sequential consistency, primary-based protocols prevail:
- Each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x.

- **Remote- Write Protocols** : Primary could be fixed at a remote server

- **Local- Write Protocols**: Write operations could be carried out locally after moving primary to the process where the write operation is initiated

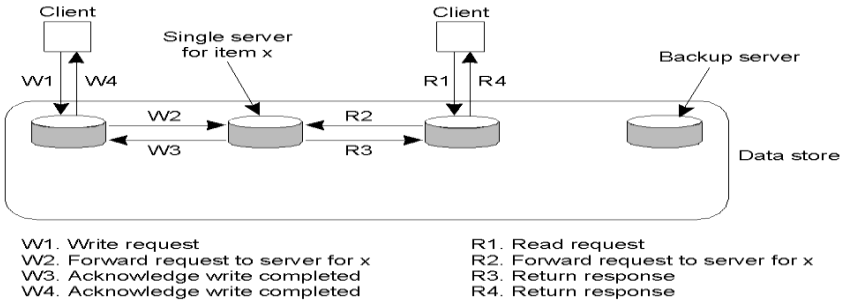
Consistency Protocol: Replicated-Write Protocols

Write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas.

•**Active replication**, in which an operation is forwarded to all replicas

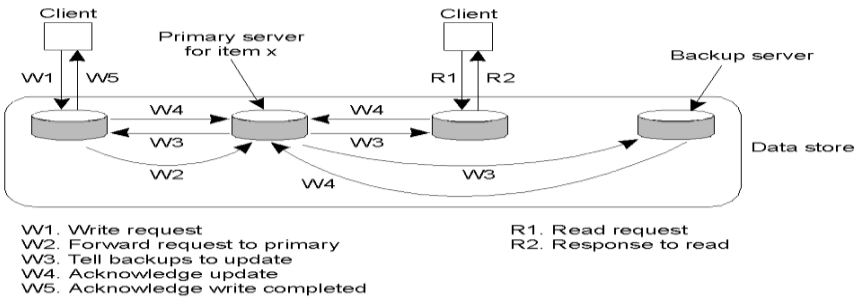
•**Based on Majority voting**

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 Based: Remote-Write Protocols…



- Read operations are on local copies, where updates must be propagated to backup server and other copies. Problem: long time for a update propagation.
- The primary performs the update on its local copy of x, and subsequently forwards the update to the backup servers.
- Each backup server performs the update as well, and sends an acknowledgment back to the primary.
- When all backups have updated their local copy, the primary sends an acknowledgment back to the initial process.

Primary Based: Remote-Write Protocols…

•Non blocking approach:
- When primary has updated local copy of x, it returns an acknowledgment.
- Primary tells the backup servers to perform the update as well

- Disadvantage: In a blocking scheme, the client process knows for sure that the update operation is backed up by several other servers. This is not the case with a non blocking solution.

- Advantage: write operations may speed up considerably.

Primary Based: Remote-Write Protocols

Primary-backup protocols provide implementation of sequential consistency:

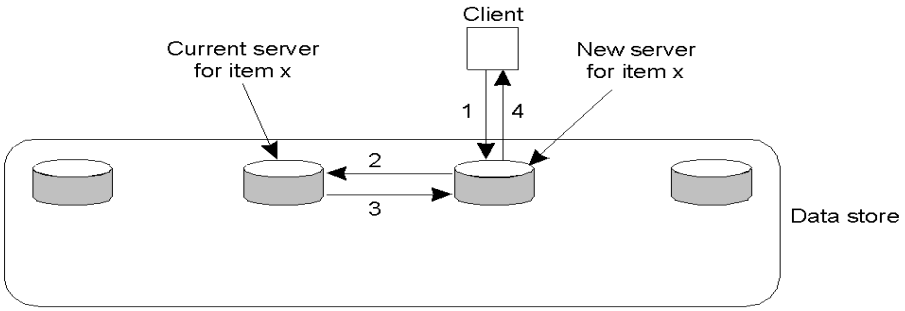
- Primary can order all incoming writes in a globally unique time order.

- All processes see all write operations in same order, no matter which backup server they use to perform read operations.

- Blocking protocols: processes will always see effects of their most recent write operation

- Nonblocking protocol: Recent write operation is not guaranteed without taking special measures

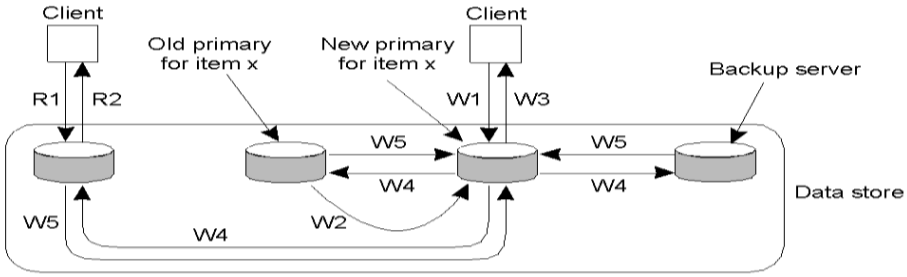
Primary Based: Local-Write Protocols



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

- 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 Based: Local-Write Protocols…



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. Read local copy, whereas updates must be propagated to all replicas. Applicable to mobile computers .

Primary Based: Local-write protocol…

•Nonblocking local-write primary-based protocols are used for distributed file systems in general.

•Fixed central server at which all write operations take place, as in the case of remote-write primary backup.

•Server temporarily allows one of the replicas to perform a series of local updates, as this may considerably speed up performance.

•When replica server is done, the updates are propagated to the central server, from where they are then distributed to the other replica servers

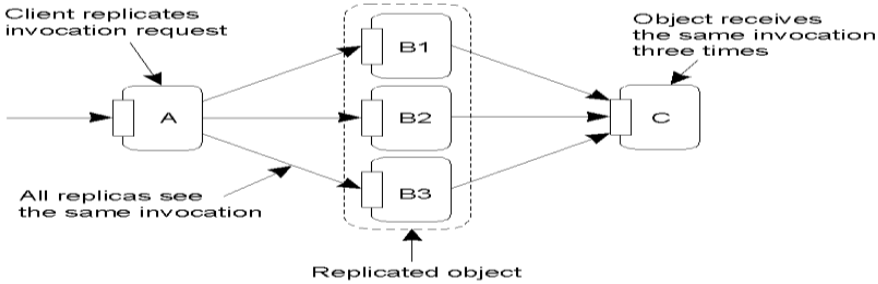
Replicated-Write Protocols

Write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas.

•Active replication, in which an operation is forwarded to all replicas

•Consistency Protocols: Based on Majority voting

Active Replication



- Each replica has an associated process that carries out update operations.

•Contrast to other protocols, updates are generally propagated by means of write operation that causes update. i.e. operation is sent to each replica

Active Replication…

-Operations need to be carried out in the same order everywhere and hence require totally-ordered multicast mechanism using Lamport's logical clocks

- does not scale well in large distributed systems

- total ordering can be achieved using a central coordinator/sequencer

  - first forward each operation to sequencer, which assigns it a unique sequence number and subsequently forwards the operation to all replicas

  -Operations are carried out in order of their sequence number

  - Resembles primary-based consistency protocols

Quorum-Based Protocols

•Require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.
•To read a file of which N replicas exist, a client needs to assemble a read quorum, an arbitrary collection of any NR servers, or more.

•To modify a file, a write quorum of at least Nw servers is required.

•Two constraints:

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

- The first constraint is used to prevent read-write conflicts
- Second constraint prevents write-write conflicts.

• Only after the appropriate number of servers has agreed to participate can a file be read or written.

Quorum-Based Protocols…

•Consider a distributed file system and suppose a file is replicated on *N servers.*
-*Rule :* to update a file, client must first contact at least half the servers plus one (a majority) and get them to agree to do the update.

•When file is changed a new version number is associated with new file. Version number is used to identify version of the file - same for all newly updated files.

• To read a replicated file, a client must contact at least half the servers plus one and ask them to send the version numbers associated with the file.

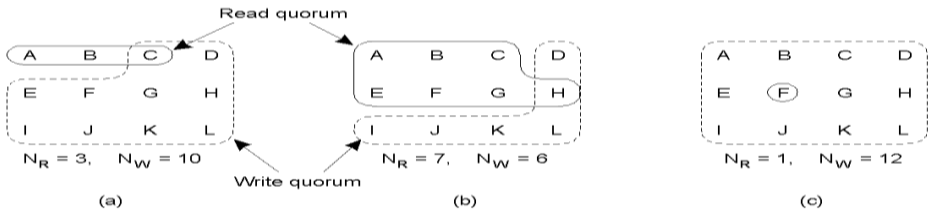
Quorum-Based Protocols….

•If all the version numbers are the same, this must be the most recent version because an attempt to update only the remaining servers would fail because there are not enough of them.

• If there are five servers and a client determines that three of them have version 8, it is impossible that the other two have version 9. any successful update from version 8 to version 9 requires getting three servers to agree to it, not just two.

Quorum-Based Protocols

•Require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.

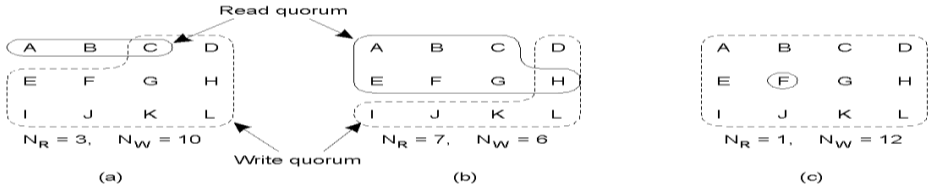


Three examples of the voting algorithm:

a) A correct choice of read and write set which has *NR = 3 and Nw = 10.*

- Let the most recent write quorum consisted of the 10 servers 0 through 10.
- All of these get the new version and the new version number.
- Any subsequent read quorum of three servers will have to contain at least one member of this set.
- When the client looks at the version numbers, it will know which is most
- recent and take that one.

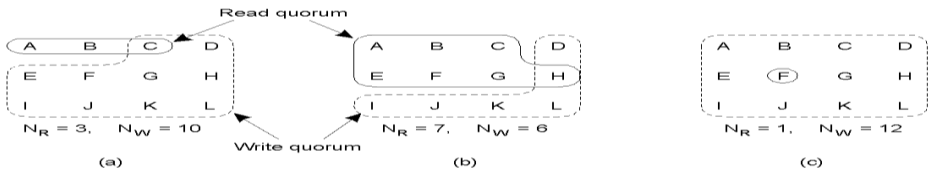
Quorum-Based Protocols



b) A choice that may lead to write-write conflicts

- a write-write conflict may occur because Nw =6 !> N/2.
- if one client chooses {A,B, C,E,F,G} as its write set and another client chooses {D,H,I.J,K,L} as its write set, then clearly we will run into trouble as the two updates will both be accepted without detecting that they actually conflict.

Quorum-Based Protocols



c) A correct choice, known as ROW A (read one, write all)

is especially interesting because it sets *NR = 1* making it possible to read a replicated file by finding any copy and using it.

The price paid for this good read performance, however, is that write updates need to acquire all copies.

This scheme is generally referred to as Read-One, Write-All (ROWA).

There are several variations of quorum-based replication protocols.

Cache Coherence problem

•Caches form a special case of replication, and are controlled by clients instead of servers.
• Need to ensure that cache is consistent with server-initiated replicas
Issues:
• Coherence detection strategies
• Cache coherence enforcement strategy
•Process modifies data

Coherence detection strategies

• Coherence detection strategies:
- Static solutions:
  - compiler is assumed to perform necessary analysis prior to execution, and to determine which data may actually lead to inconsistencies because they may be cached
  - compiler simply inserts instructions that avoid inconsistencies
- Dynamic solutions:
  - inconsistencies detected runtime
  - check is made with the server to see whether the cached data have been modified since they were cached

Coherence detection strategies…

Dynamic detection-based protocols - when during a transaction , detection is done:

i) During transaction when cached data item is accessed, client needs to verify whether data item is still consistent with version stored at server and must processed with the transaction until cached version's consistency has been verified.

ii) Let transaction proceed while verification is taking place – assuming cache was updated before transaction started- if cache data is not up to date then transaction will have to abort

iii) Cached data are up to date only when the transaction commits – hopes for the best. After all the work has been done, accessed data are verified for consistency. When stale data were used, the transaction is aborted

Coherence enforcement strategy

•Coherence enforcement strategy: how caches are kept consistent with the copies stored at servers:

- Disallow shared data to be cached at all.(only private data is cached)

- Shared data are kept only at servers, which maintain consistency using one of the primary-based or replication-write protocols

- When shared data can be cached, there are two approaches to enforce cache coherence:
  i) Server sends invalidation to all caches whenever data is modified
  ii) Simply propagate update.

Process modifying cached data

•Process modified cached data:

i) When read-only caches are used:
- update operations can be performed only by servers, which follow distribution protocol to ensure that updates are propagated to caches.
- a pull-based approach may be followed: that its cache is stale, and requests a server for an update.

ii) When write through caches are used
- clients directly modify cached data, and forward update to server
- write-through caching is similar to a primary-based local-write protocol in which the client's cache has become a temporary primary.
- To guarantee (sequential) consistency, it is necessary that the client has been granted exclusive write permissions, or otherwise write-write conflicts may occur

Process modifying cached data….

- Write-through caches potentially offer improved performance over other scheme since all operations can be carried out locally.

- Further improvements - delay the propagation of updates for multiple writes to take place before informing the servers. - write-back cache applied in distributed file systems.

Summary

•Reasons for replicating data: improving the reliability of a distributed system and improving performance introduces a consistency

•Problem: whenever a replica is updated, that replica becomes different from the others.

•Keep replicas consistent, we need to propagate updates in such a way that temporary inconsistencies are not noticed

•Relax consistency somewhat. Different consistency models exist Data centric and Client centric

Summary…

•Replica placement

•Different methods to propagate updated content to the relevant replica servers

•

•Transferring modified data – Pull and Push techniques

•Consistency Protocols – Primary Based and Active Replication

•Cache Coherence strategy

THANKS