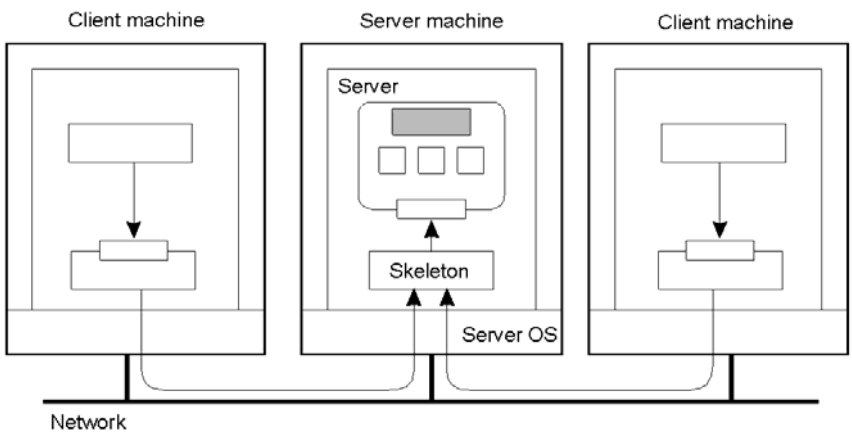
COP 6611 Advanced Operating System

# Consistency and Replication

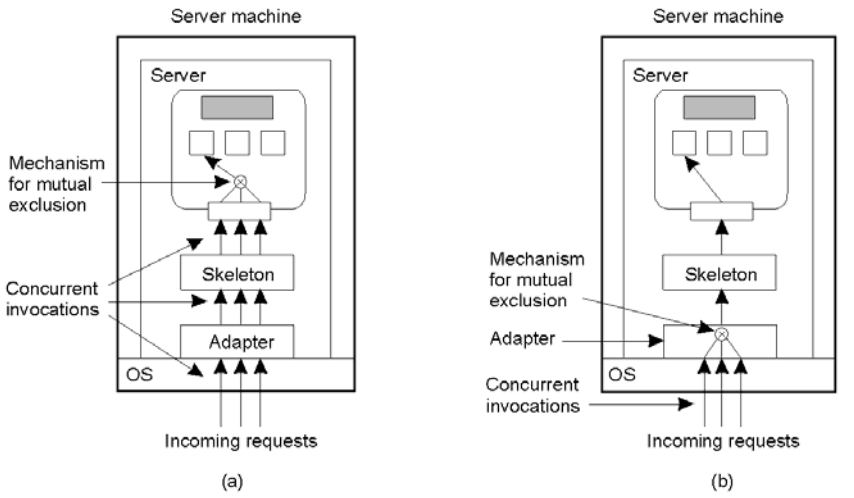
Chi Zhang
czhang@cs.fiu.edu

## Object Replication (1)




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

# Object Replication (2)

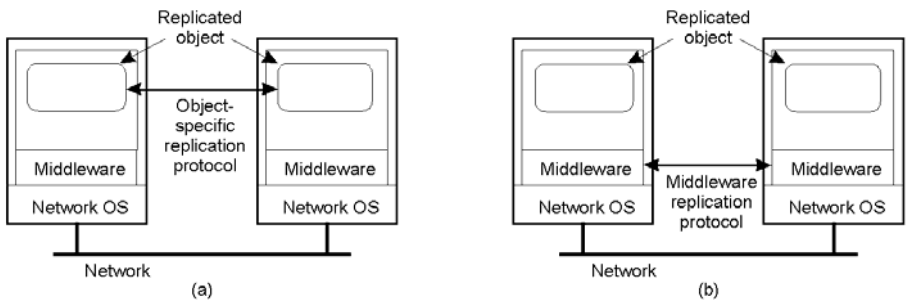



a) A remote object capable of handling concurrent invocations on its own.

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



# Object Replication (3)




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

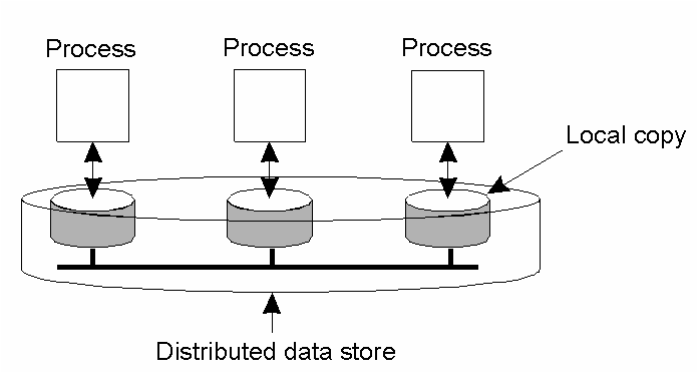
b) A distributed system responsible for replica management

# Why Replication?

- Performance Scalability
  - Clients can access a nearby copy
  - Ease the traffic load on the network and the server
- Reliability
  - Failure
  - Denial of Service Attacks
- Problem: Consistency
  - Updates need extra bandwidth!
  - Loosen consistency condictions.
    - Depends on the data type and the application.

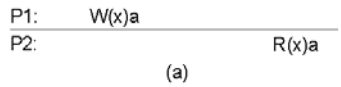


# Data-Centric Consistency Models

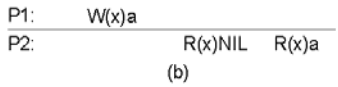



The general organization of a logical data store, physically distributed and replicated across multiple processes.

# Strict Consistency




(a)




(b)

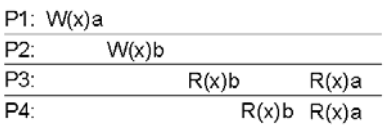
Behavior of two processes, operating on the same data item.

- A strictly consistent store.
- A store that is not strictly consistent (non-zero delay).

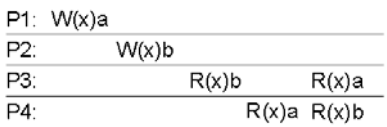
Assumes an absolute global time and zero propagation delay!



# Linearizability and Sequential Consistency (1)




(a)




(b)

a) A sequentially consistent data store.$W_1$(x)a delayed!

b) A data store that is not sequentially consistent.

Any interleaving of R/W is acceptable, but all processes see the same interleaving! Assumes a logical time.

## Linearizability and Sequential Consistency (2)

E1: W1(x)a

E2: W2(x)b

E3: R3(x)b, R3(x)a

E4: R4(x)b, R4(x)a

Merge $E_i$ into a single history string H, such that

- Program order must be maintained (order within $E_i$)
- Data coherence must be respected.
  - W1(x)a, W2(x)b, R3(x)a ☹



## Causal Consistency (1)

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.

Implementation:

"multicast" writes and vector timestamps.

# Causal Consistency (2)

| | | | | | |
|---|---|---|---|---|---|
| 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.



# Causal Consistency (3)

| | | | | |
|---|---|---|---|---|
| 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)

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

# FIFO Consistency (1)

Necessary Condition:

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

Processes do not have to stall before the next writing

Sometimes Counter-intuitive



# FIFO Consistency (2)

| | | | | | | |
|---|---|---|---|---|---|---|
| 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

# Weak Consistency (1)

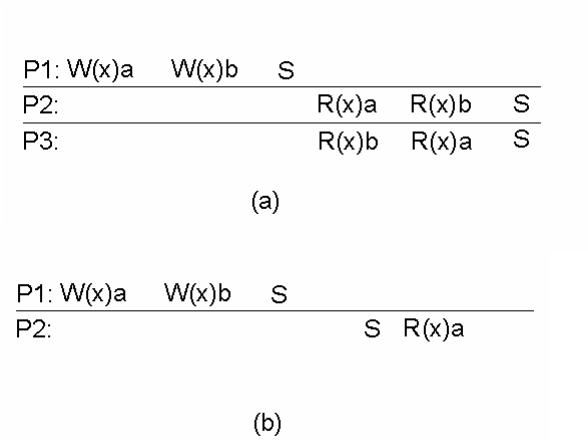
Sequential consistency on a synchronization variable.

Intermediate results need not be propagated upon synchronization: copy remote data to local stores, and copy local date to remote stores

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



# Weak Consistency (2)




a) A valid sequence of events for weak consistency.

b) An invalid sequence for weak consistency.

# Release Consistency (1)

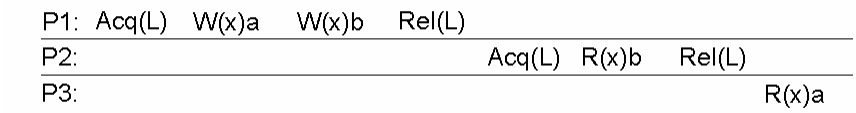
Acquire: copy remote data to local stores

Release: copy local date to remote stores

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



# Release Consistency (2)




A valid event sequence for release consistency.

With lazy release consistency, nothing is done at the time of a release. Data are brought up to date when a require is done.

# Entry Consistency (1)

Fine-Grained Lazy Release Consistency

Conditions:

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



# Entry Consistency (2)

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| P1: | Acq(Lx) | W(x)a | Acq(Ly) | W(y)b | Rel(Lx) | Rel(Ly) | | | | | |
| P2: | | | | | | | Acq(Lx) | R(x)a | | R(y)NIL | |
| P3: | | | | | | | | Acq(Ly) | R(y)b | | |

A valid event sequence for entry consistency.

Extra overhead and complexity of associating every shared data item with some synchronization variable.

# Summary of Consistency Models

| 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)

a) Consistency models not using synchronization operations.

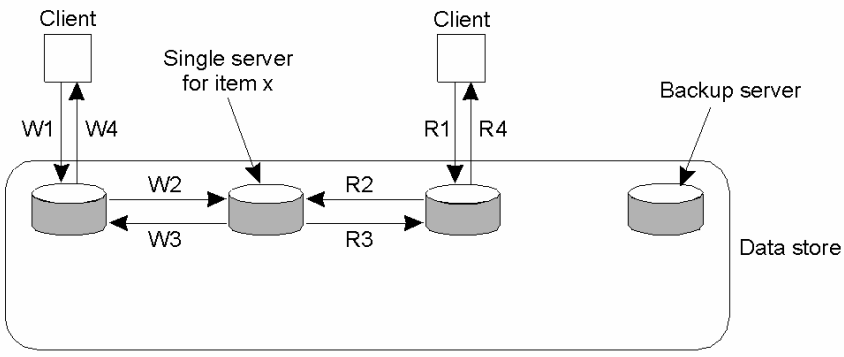
b) Models with synchronization operations.



# Consistency Protocols

- Primary-based Protocols
  - Each date item has an associated primary, serving as the coordinator.
  - Remote write protocols (write to remote primary)
  - Local write protocols.
- Replicated writes
  - Write operations can be carried out at multiple replicas instead of one.
  - Active replication.
    - Totally-ordered Multicast
  - Quorum-based Protocols.
- Cache Coherent Protocols
  - Lazy consistency

# Remote-Write Protocols (1)



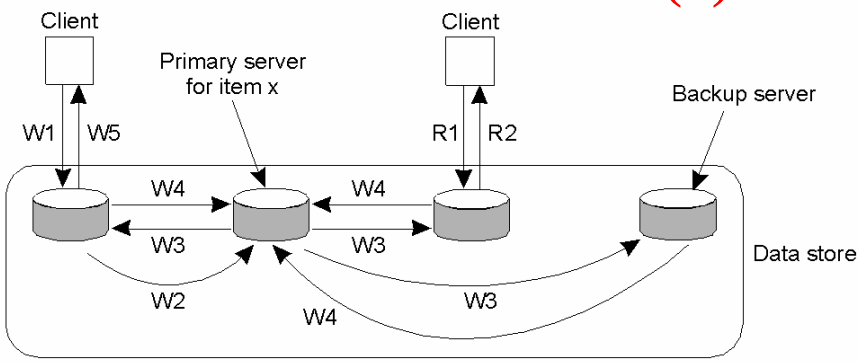

W1. Write request
W2. Forward request to server for x
W3. Acknowledge write completed
W4. Acknowledge write completed

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

Primary-based remote-write protocol with a fixed server.
Data are not replicated!



# Remote-Write Protocols (2)

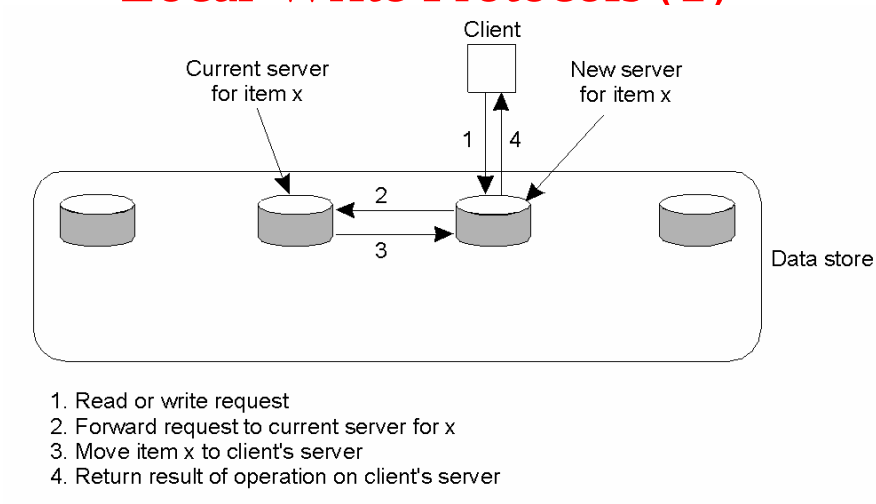



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

The principle of primary-backup protocol.
Write to primary (blocking or non-blocking?); read local.

# Local-Write Protocols (1)

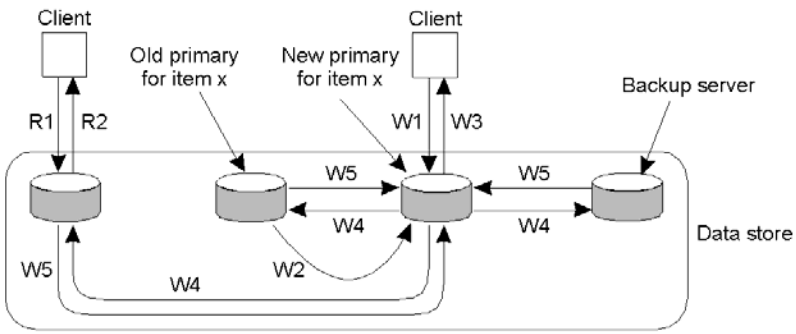



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

Primary-based local-write protocol in which a single copy is migrated between processes. Data are not replicated!



# Local-Write Protocols (2)



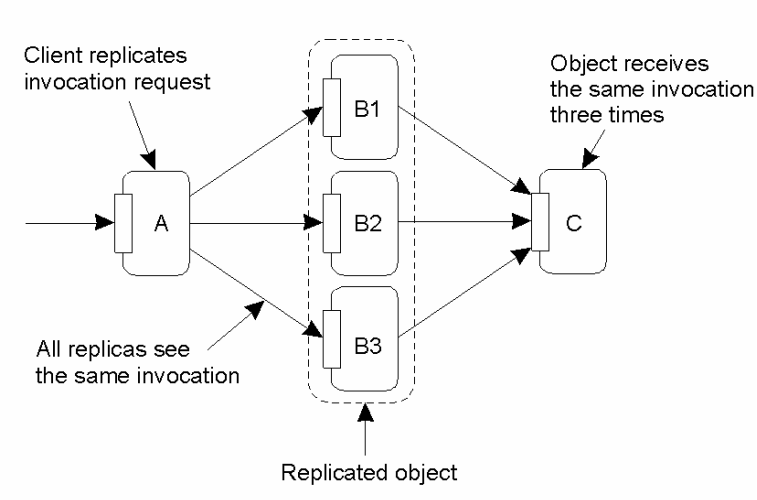

W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

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

Multiple writes by a client can be carried out locally.

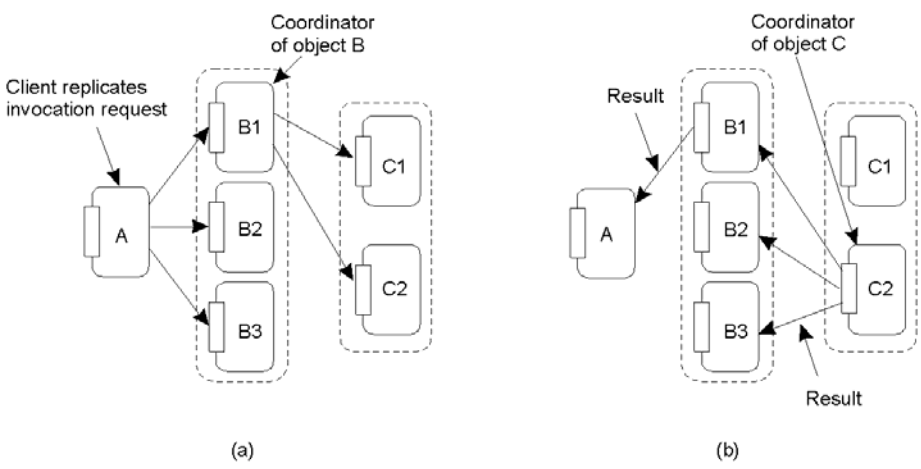
# Active Replication (1)




The problem of replicated invocations.

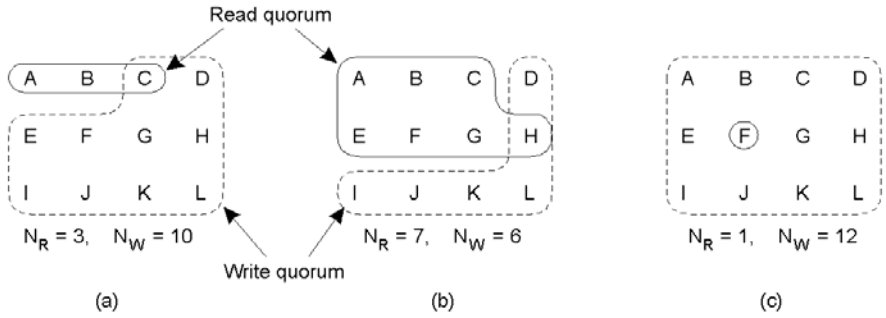


# Active Replication (2)




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

# Quorum-Based Protocols




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)

$N_R + N_W > N$ (no r/w conflicts) ; $N_W > N/2$ (no w/w conflicts).

When writing, update the version number on the servers that give permissions. When reading, read the server with the latest version number.



# Cache Coherent Protocols

- e.g. caches in distributed systems; shared-memory multi-processor.
- When to detect the consistency
- A transaction can choose to
  - Block until the cache is consistent
  - Proceed while verification is taking place and abort later if necessary.
  - Verify the consistency when it commits

# Eventual Consistency(1)

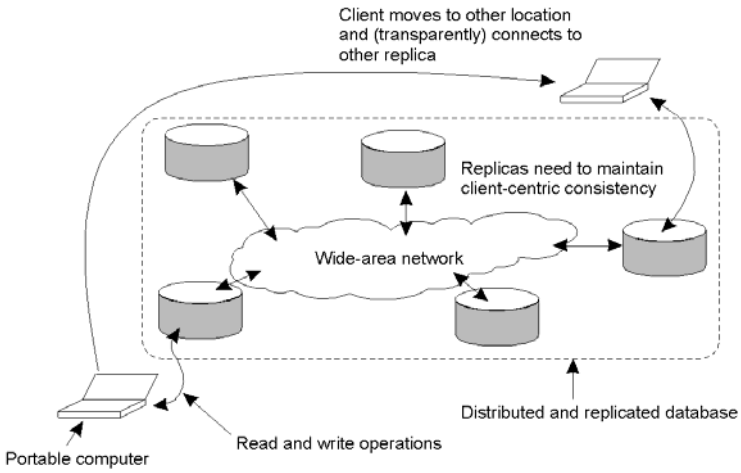
- Very few processes (even one) can perform write operations
  - Write/Write conflicts are relatively easy to solve.
  - Cheap to implement
- A relatively high degree of inconsistency can be tolerated.
- All replicas will gradually (eventually!) become consistent.



# Epidemic Protocols

- Eventual Consistency.
  - Updates for a specific data item are initiated at a single server. (no w/w conflicts)
- Propagate updates in as few as messages possible, **and** as fast as possible.
- A server P picks another server Q randomly.
  - An infective P pushed updates.
    - When many servers are infective, a waste of messages
    - Gossip algorithm: if Q is already infective, P may lose interests.
  - A susceptible P pulls updates.
    - Works better when many servers are infective.

# Client-Centric Consistency




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

Client-centric consistency is needed for a single client accessing different replicas.



# Monotonic Reads / Writes

Monotonic Read Consistency

- If a process reads the value of a data item *x*, Any successive read operations on *x* by that process will always return the same value or a more recent version
- E.g. A distributed email database

Monotonic Write Consistency

- A write operation on a data item *x* is completed before any successive write operation (no matter the location) on *x* by the same process. (like data-centric FIFO consistency)
- E.g. compiling a software library.

## Read Your Writes / Writes Follows Reads

- Read-Your-Writes Consistency
  - The effect of a write operation by a process on a data item *x* will always be seen by a successive read operation by the same process.
  - E.g. Update HTML page.
- Writes-Follow-Reads Consistency
  - A write operation by a process on a data item *x* following a previous read on *x* by the same process, is guaranteed to take place on the same or a more recent value of *x* that was read.
  - E.g. forbidding a reply to be posted before the original message.



## Implementation of Monotonic Consistency

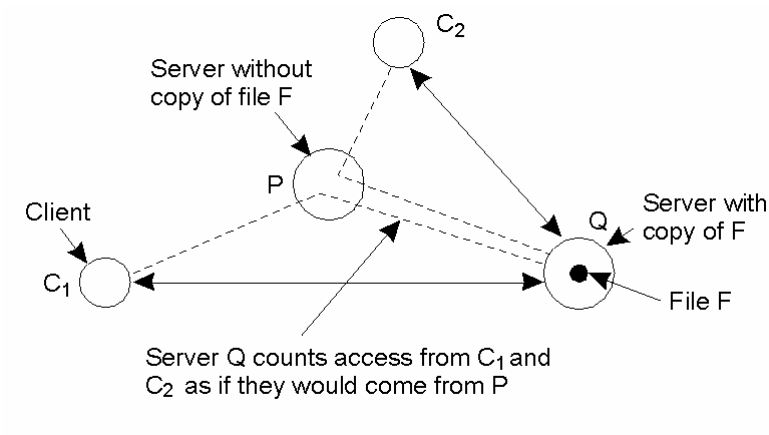
- Each write is assigned a unique ID.
- Each client keeps a read set (of write IDs) and a write set.
- When reading, the client handed the current server its read set. The server brings the local copy up-to-date, if necessary.
  - Write ID should include server ID where the write is initiated, and a unique number that can reflect write sequence.
- How to reduce the size of read /write set?
  - Associate a WID with a timestamp
  - Each server keeps records of the latest timestamp it observes
  - Read sets can be represented by vector timestamps

# Replica Propagation Strategy

- Independent of consistency model
- Server-initiated replicas
  - The server **pushes** a number of temporary copies in regions where (read) requests are coming from.
  - Less messages are sent when r / w ratio is high (Efficient in CDN )
- Each server keeps track of access counts per {file, remote replica server).
- If the count drops below *threshold1*, delete the replica. If the count increases above *threshold2*, push the replica.
- In between, migrate the data.



# Server-Initiated Replicas




Counting access requests from different clients.

# Client-Initiated Replicas

- Client **pulls** the data to a local cache.
- The client can poll the server to see whether an update has occurred.
- Less messages are sent when r / w ratio is relatively low.
  - E.g. Efficient for client-side cache
- Response time increases in the case of a miss.



# Pull versus Push Protocols

- Push-based
  - High degree of consistency / Less client response time / Multicast can be utilized.
  - Server needs to maintain a list of client replicas!
- A hybrid form: lease
  - A lease is a promise by the server that it will push updates to the client for a specified time.
  - When a lease expires, the client has to pull.
  - Dynamically adapted
    - r/w high $\Rightarrow$ lease $\uparrow$ (closer to push-based protocols)
    - r/w low $\Rightarrow$ lease $\downarrow$ (closer to pull-based protocols)
    - Server overhead high $\Rightarrow$ lease $\downarrow$

# State versus Operations

Three choices for sending updates

- Notification of an update (or invalidation)
  - Writes are applied to the local copy of a server, and only invalidation is sent.
  - Use little bandwidth
  - Works best when r / w ratio is low.
- Transfer the modified data.
  - Efficient when r / w is high.
  - Could be aggregated.
- Transfer the update operation.
  - The receiver replays the update.