Consistency and Replication

- Today:
  - Consistency models
    - Data-centric consistency models
    - Client-centric consistency models

Why replicate?

- Data replication versus compute replication

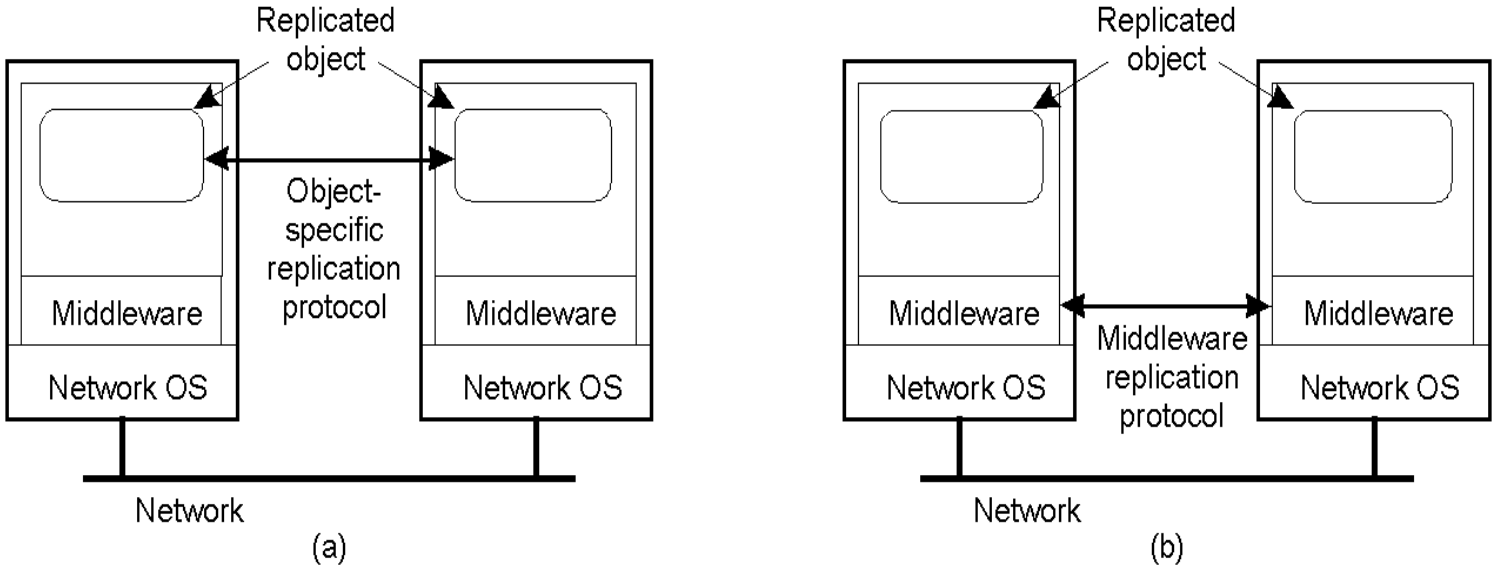
- Data replication: common technique in distributed systems
- Reliability
  - If one replica is unavailable or crashes, use another
  - Protect against corrupted data
- Performance
  - Scale with size of the distributed system (replicated web servers)
  - Scale in geographically distributed systems (web proxies)

Replication Issues

- When to replicate?
- How many replicas to create?
- Where should the replicas be located?

- Will return to these issues later (WWW discussion)
- Today: how to maintain *consistency*?
- Key issue: need to maintain *consistency* of replicated data
  - If one copy is modified, others become inconsistent

Object Replication

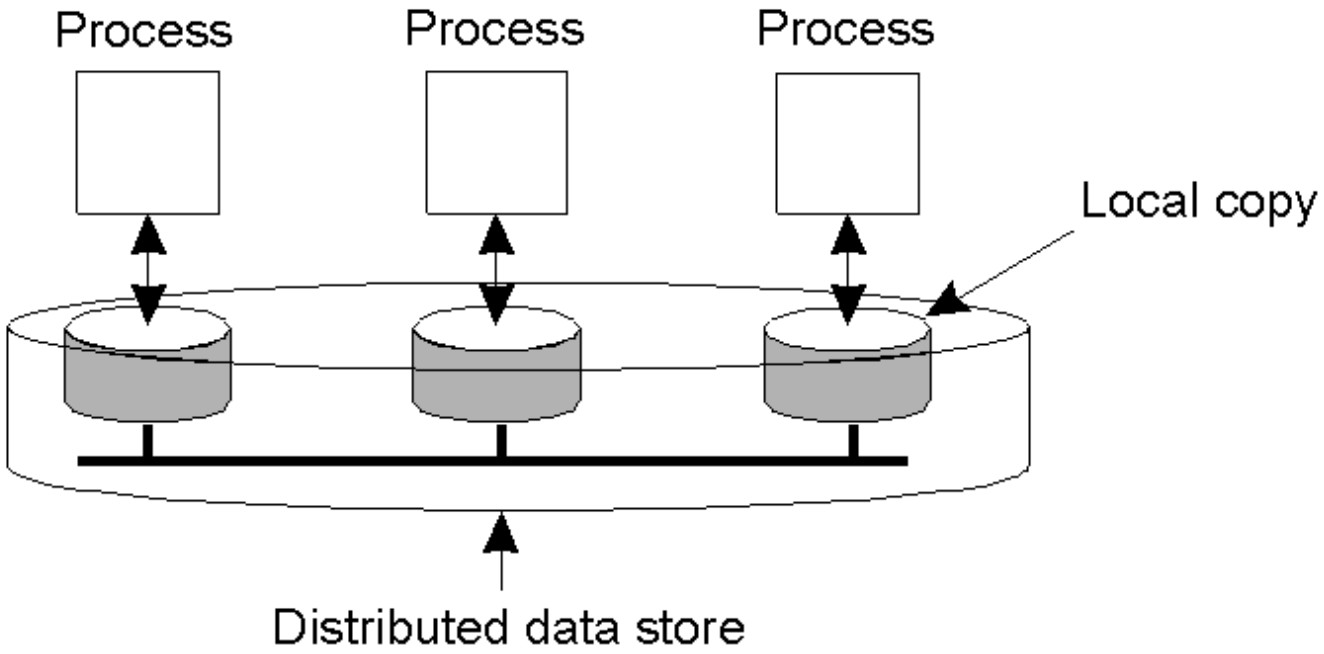


- Approach 1: application is responsible for replication
  - Application needs to handle consistency issues
- Approach 2: system (middleware) handles replication
  - Consistency issues are handled by the middleware
  - Simplifies application development but makes object-specific solutions harder

Replication and Scaling

- Replication and caching 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
      - What semantics to provide?
      - Tight consistency implies a global (logical) clock
- Solution: loosen consistency requirements
  - Variety of consistency semantics possible

Data-Centric Consistency Models



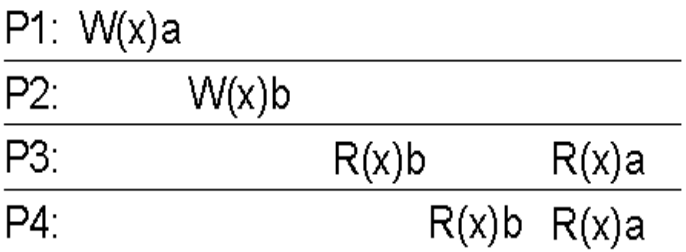
- Consistency model (aka *consistency semantics*)
  - Contract between processes and the data store
    - If processes obey certain rules, data store will work correctly
  - All models attempt to return the results of the last write for a read operation
    - Differ in how "last" write is determined/defined

Strict Consistency

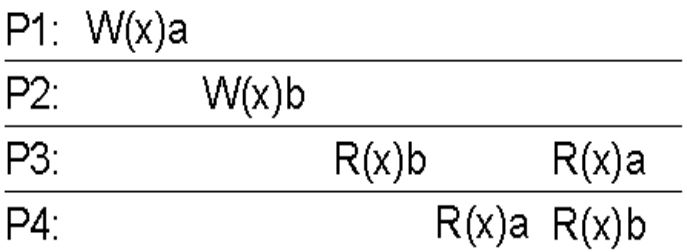
- Any read always returns the result of the most recent write
  - Implicitly assumes the presence of a global clock
  - A write is immediately visible to all processes
    - Difficult to achieve in real systems (network delays can be variable)

Sequential Consistency

- Sequential consistency: weaker than strict consistency
  - Assumes all operations are executed in some sequential order and each process issues operations in program order
    - Each process preserves its program order
    - Any valid interleaving is allowed
    - All processes agree on the same interleaving
    - Nothing is said about "most recent write"
- *Q:* Are these executions valid according to sequential consistency?



(a)



(b)

Sequential Consistency Example

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

- Processes must accept all valid execution. *Q: are these valid?*

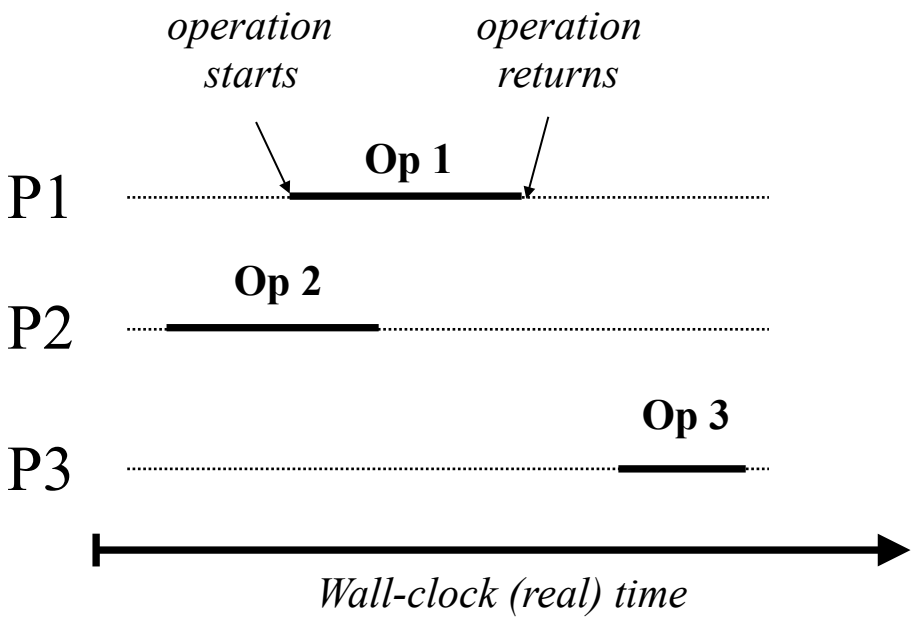
x = 1;	x = 1;	y = 1;	y = 1;
print ((y, z);	y = 1;	z = 1;	x = 1;
y = 1;	print (x,z);	print (x, y);	z = 1;
print (x, z);	print(y, z);	print (x, z);	print (x, z);
z = 1;	z = 1;	x = 1;	print (y, z);
print (x, y);	print (x, y);	print (y, z);	print (x, y);
Prints: 001011	Prints: 101011	Prints: 010111	Prints: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

Examples of Unacceptable Runs

- Signature 00 00 00 represents and unacceptable run
  - Q: why?
- Signature 00 10 01 represents and unacceptable run
  - Q: why?

Linearizability

- Requires sequential consistency *and* respect of real-time order
  - Operations that overlap in time can be reordered arbitrarily
  - The other operations need to be reordered according to real-time order



Q: Are these executions serializable/linearizable?
- Op 1 - Op 2 - Op 3
- Op 3 - Op 1 - Op 2

Serializability

- Consistency in the context of transactions
  - Grouping multiple operations in database management systems
- If we look at the database as a single object and at transactions as single operations
  - Serializability = sequential consistency
  - Strict serializability = linearizability

Causal consistency

- Causally related writes must be seen by all processes in the same order.
  - Concurrent writes may be seen in different orders on different machines
- *Q:* Are these execution valid according to causal 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)

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

(b)

Other models

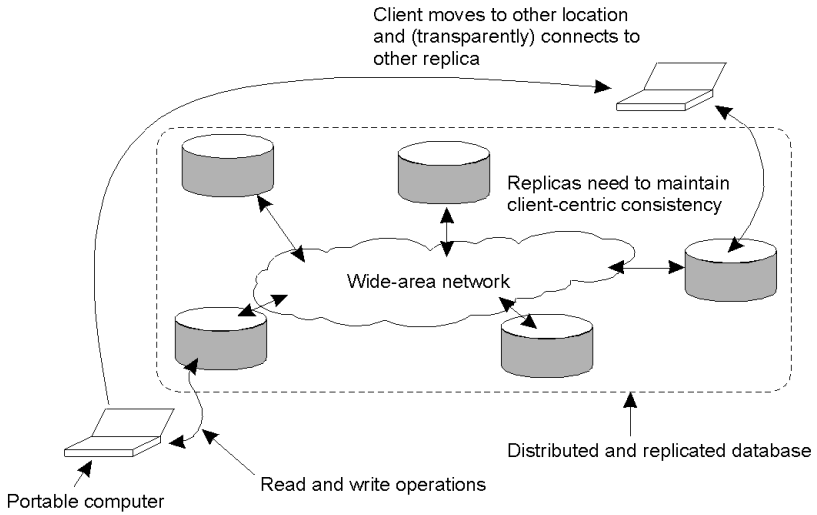
- FIFO consistency: writes from a process are seen by others in the same order. Writes from different processes may be seen in different order (even if causally related)
  - Relaxes causal consistency
  - Simple implementation: tag each write by (Proc ID, seq #)
- Entry consistency: Critical sections for *groups* of operations
  - Each process has a local copy of data item
  - Acquire per-item locks to enter critical sections
  - Upon exit of critical section, send results everywhere
  - Do not worry about propagating intermediate results
  - Lower level than transactions (explicitly handle locks)

Eventual Consistency

- Many systems: one or few processes perform updates
  - How frequently should these updates be made available to other read-only processes?
- Examples:
  - DNS: single naming authority per domain
  - Only naming authority allowed updates (no write-write conflicts)
  - How should read-write conflicts (consistency) be addressed?
  - NIS: user information database in Unix systems
    - Only sys-admins update database, users only read data
    - Only user updates are changes to password

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: write-write conflicts do not result in coordination
    - Solve conflicts via merging. Q: How to detect conflicts?
  - Things work fine so long as user accesses same replica
  - What if they don't?



Client-centric Consistency Models

- Way to strengthen weak (e.g. eventual) data-centric consistency
- Apply to *sessions* of subsequent operations from the same client
- *Monotonic reads*
  - Once read, subsequent reads on that data items return same or more recent values
- *Monotonic writes*
  - A write must be propagated to all replicas before a successive write by the *same process*
  - Resembles FIFO consistency (writes from same process are processed in same order)
- *Read your writes*: read(x) always returns write(x) by that process
- *Writes follow reads*: write(x) following read(x) will take place on same or more recent version of x

Epidemic Protocols

- Used in Bayou system from Xerox PARC
- Bayou: weakly connected replicas
  - Useful in mobile computing (mobile laptops)
  - Useful in wide area distributed databases (weak connectivity)
  - Eventually-consistent systems use similar approaches to propagate updates
- Based on theory of epidemics *(spreading infectious diseases)*
  - Upon an update, try to "infect" other replicas as quickly as possible
  - Pair-wise exchange of updates (*like pair-wise spreading of a disease)*
  - Terminology:
    - Infective store: store with an update it is willing to spread
    - Susceptible store: store that is not yet updated
- Many algorithms possible to spread updates

Spreading an Epidemic

- Anti-entropy
  - Server *P* picks a server *Q* at random and exchanges updates
  - Three possibilities: only push, only pull, both push and pull
- Rumor mongering (aka *gossiping*)
  - Upon receiving an update, *P* tries to push to *Q*
  - If *Q* already received the update, stop spreading with prob *1/k*
  - Analogous to "hot" gossip items => stop spreading if "cold"
  - Does not guarantee that all replicas receive updates
    - Chances of staying susceptible (no update received): $s = e^{-(k+1)(1-s)}$

Removing Data

- Deletion of data items is hard in epidemic protocols
- Example: server deletes data item *x*
  - No state information is preserved
    - Can't distinguish between a deleted copy and no copy!
- Solution: death certificates
  - Treat deletes as updates and spread a death certificate
    - Mark copy as deleted but don't delete
    - Need an eventual clean up
      - Clean up dormant death certificates

CAP Theorem

- Conjecture by Eric Brewer at PODC 2000 conference
  - It is impossible for a web service to provide all three guarantees:
    - **Consistency** (nodes see the same data at the same time)
    - **Availability** (node failures do not the rest of the system)
    - **Partition-tolerance** (system can tolerate message loss)
  - A distributed system can satisfy any two, but not all three, at the same time
- Conjecture was established as a theorem in 2002 (by Lynch and Gilbert)

CAP Theorem Examples

- Consistency + Availability
  - Cannot ensure availability with message loss
  - Eg: Single database, cluster database, LDAP, xFS
- Consistency + Partition tolerance
  - Trivial: system does nothing
  - Advanced: provide availability only in "good runs"
  - E.g.: Distributed database, distributed locking
- Availability + Partition tolerance
  - Allow writes even if consistency cannot be achieved
  - E.g.: Cassandra, Web caching, DNS