Consistency and Replication

Distributed Software Systems

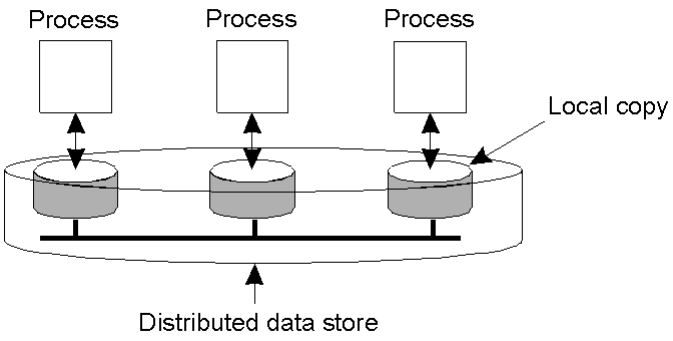
Outline

- Consistency Models
- Approaches for implementing Sequential Consistency
  - primary-backup approaches
  - active replication using multicast communication
  - quorum-based approaches
- Update Propagation approaches
- Approaches for providing weaker consistency
  - Gossip: casual consistency
  - Bayou: eventual consistency

Replication

- Motivation
  - Performance Enhancement
  - Enhanced availability
  - Fault tolerance
  - Scalability
    - tradeoff between benefits of replication and work required to keep replicas consistent
- Requirements
  - Consistency
    - Depends upon application
    - In many applications, we want that different clients making (read/write) requests to different replicas of the same logical data item should not obtain different results
  - Replica transparency
    - desirable for most applications

Data-Centric Consistency Models



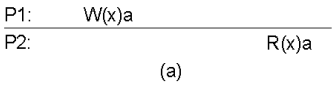
The general organization of a logical data store, physically distributed and replicated across multiple processes.

Consistency Models

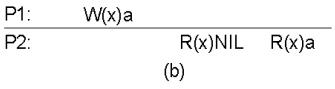
- Consistency Model is a contract between processes and a data store
  - if processes follow certain rules, then store will work "correctly"
- Needed for understanding how concurrent reads and writes behave wrt shared data
- Relevant for shared memory multiprocessors
  - cache coherence algorithms
- Shared databases, files
  - independent operations
    - our main focus in the rest of the lecture
  - transactions

Strict Consistency

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



A strictly consistent store



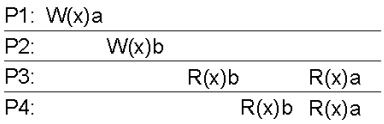
A store that is not strictly consistent.

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

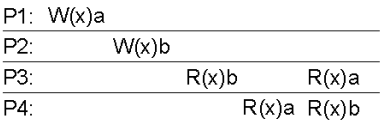
The problem with strict consistency is that it relies on absolute global time

Sequential Consistency (1)

Sequential consistency: the result of any execution is the same as if the read and write operations by all processes were executed *in some sequential order* and the operations of each individual process appear in this sequence in the order specified by its program



(a)



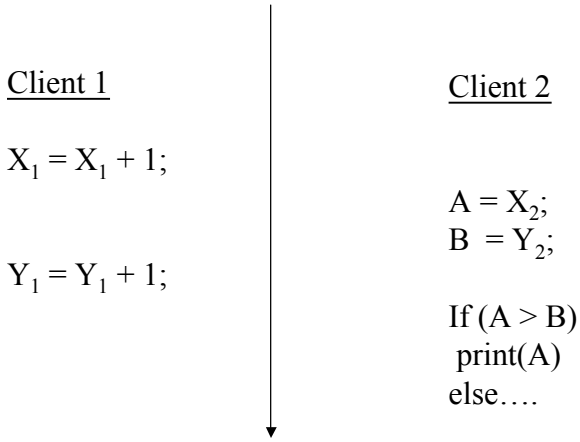
(b)

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

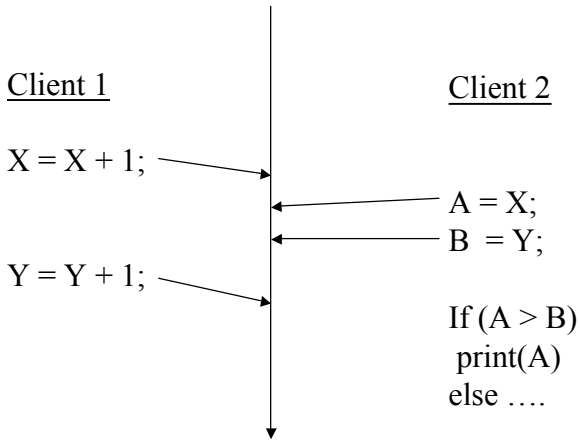
Linearizability

- Definition of sequential consistency says nothing about time
  - there is no reference to the "most recent" write operation
- Linearizability
  - weaker than strict consistency, stronger than sequential consistency
  - operations are assumed to receive a timestamp with a global available clock that is loosely synchronized
  - The result of any execution is the same as if the 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. In addition, if $ts_{op1}(x) < ts_{op2}(y)$, then OP1(x) should precede OP2(y) in this sequence

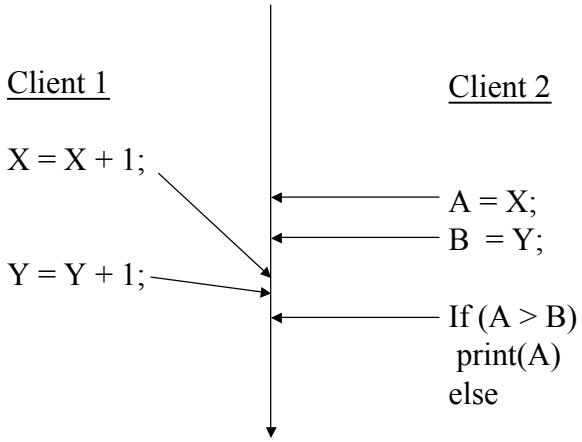
Example



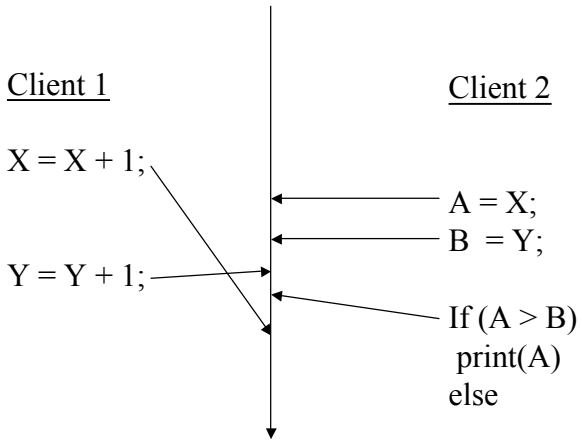
Linearizable



Not linearizable but sequentially consistent



Neither linearizable nor sequentially consistent



Causal Consistency

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

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 causally-consistent store, but not with sequentially or strictly consistent store.

Causal Consistency (2)

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

FIFO Consistency

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.

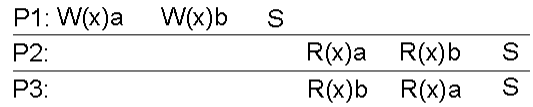
P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

A valid sequence of events of FIFO consistency

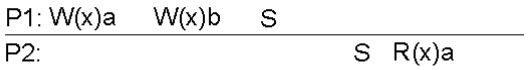
Weak Consistency (1)

- Properties:
  - 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)



(b)

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

Release Consistency (1)

- Rules:
  - 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)

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

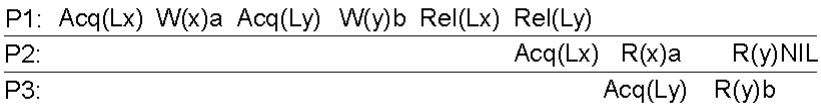
A valid event sequence for release consistency.

Entry Consistency (1)

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)



A valid event sequence for entry consistency.

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.

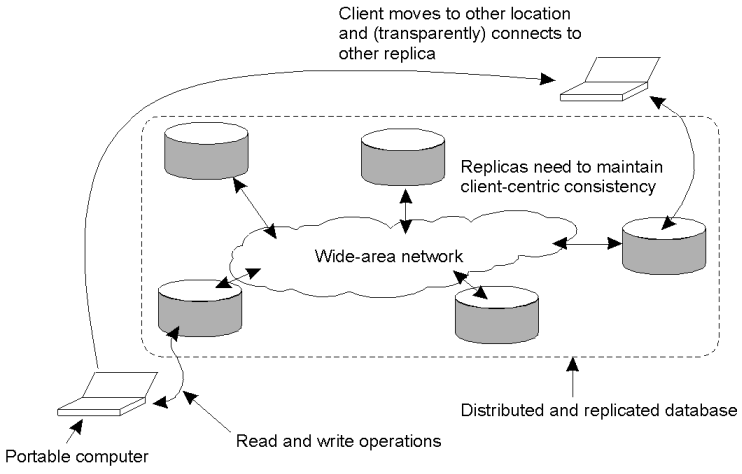
(b)

a) Consistency models not using synchronization operations.
b) Models with synchronization operations.

Weak Consistency Models

- The weak consistency models that use synchronization variables (release, entry consistency) are mostly relevant to shared multiprocessor systems
  - also modern CPUs with multiple pipelines, out-of-order instruction execution, asynchronous writes, etc.
- In distributed systems, weak consistency typically refers to weaker consistency models than sequential consistency
  - causal consistency, e.g. as used in the Gossip system
  - optimistic approaches such as those used in Bayou, Coda that use application-specific operations to achieve *eventual consistency*

Eventual Consistency



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

Sequential Consistency

- Good compromise between utility and practicality
  - We can do it
  - We can use it
- Strict consistency: too hard
- Less strict: replicas can disagree forever

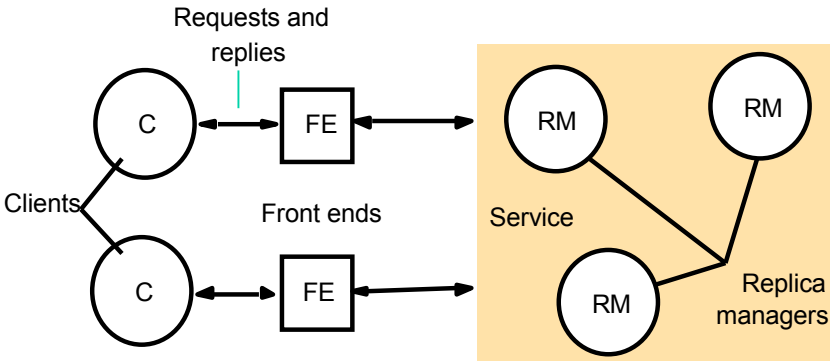
Mechanisms for Sequential Consistency

- Primary-based replication protocols
- Replicated-write protocols
  - Active replication using multicast communication
  - Quorum-based protocols

System model

- Assume replica manager apply operations to its replicas *recoverably*
- Set of replica managers may be static or dynamic
- Requests are reads or writes (updates)

A basic architectural model for the management of replicated data

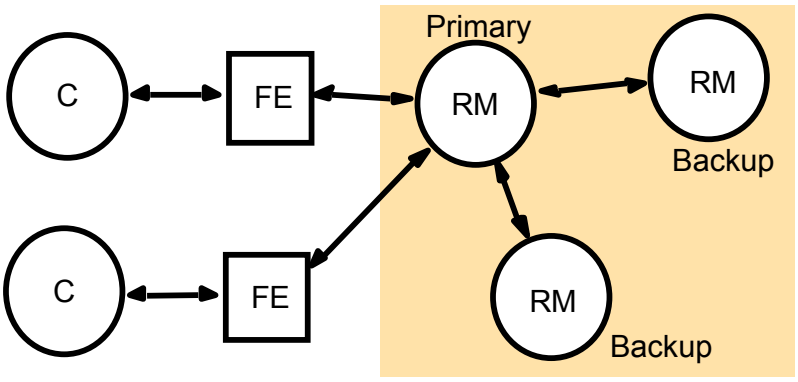


System model

Five phases in performing a request

- Front end issues the request
  - Either sent to a single replica or ***multicast*** to all replica mgrs.
- Coordination
  - Replica managers coordinate in preparation for the execution of the request, I.e. agree if request is to be performed and the ordering of the request relative to others
    - FIFO ordering, Causal ordering, Total ordering
- Execution
  - Perhaps tentative
- Agreement
  - Reach consensus on effect of the request, e.g. agree to commit or abort in a transactional system
- Response

The passive (primary-backup) model



Front ends only communicate with primary

Passive (primary-backup) replication

- Request: FE issues a request containing a unique identifier to the primary replica manager
- Coordination: The primary takes each request in the order in which it receives it
- Execution: The primary executes the request and stores the response
- Agreement: If the request is an update, the primary sends the updated state, the response, and the unique id to all backups. The backups send an acknowledgement
- Response: The primary responds to the front end, which hands the response back to the client

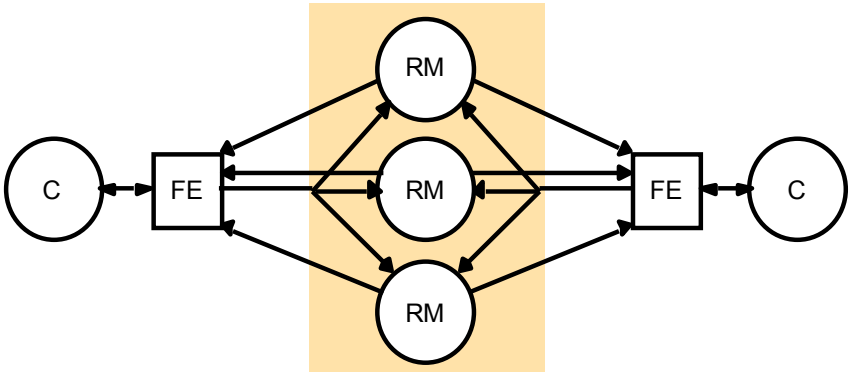
Passive (primary-backup) replication

- Implements linearizability if primary is correct, since primary sequences all the operations
- If primary fails, then system retains linearizability if a single backup becomes the new primary and if the new system configuration takes over exactly where the last left off
  - If primary fails, it should be replaced with a unique backup
  - Replica managers that survive have to agree upon which operations had been performed when the replacement primary takes over
  - Requirements met if replica managers organized as a group and if primary uses **view-synchronous communication** to propagate updates to backups
    - Will discuss view-synchronous communication in next class

Active replication using multicast

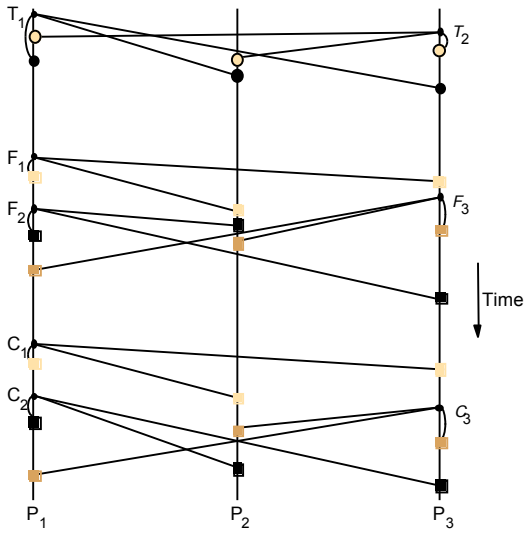
- Active replication
  - Front end multicasts request to each replica using a *totally ordered reliable multicast*
  - System achieves sequential consistency but not linearizabilty
    - Total order in which replica managers process requests may not be same as real-time order in which clients made requests

Active replication



Total, FIFO and causal ordering of multicast messages

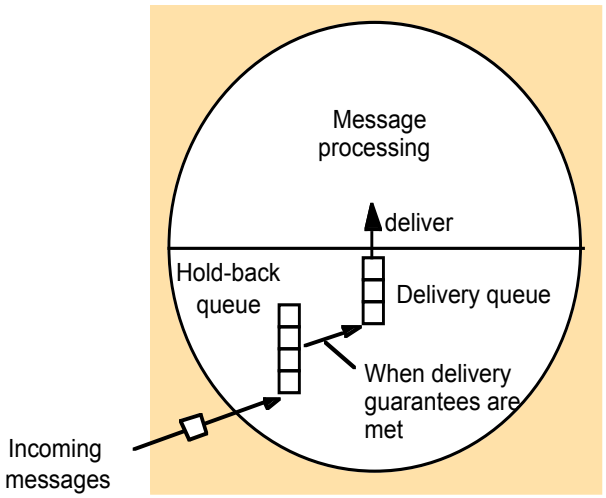
Notice the consistent ordering of totally ordered messages T_1 and T_2, the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_3 – and the otherwise arbitrary delivery ordering of messages.



Implementing ordered multicast

- Incoming messages are held back in a queue until delivery guarantees can be met
- Coordination between all machines needed to determine delivery order
- FIFO-ordering
  - easy, use a separate sequence number for each process
- Total ordering
  - Use a sequencer
  - Distributed algorithm with three phases
- Causal ordering
  - use vector timestamps

The hold-back queue for arriving multicast messages



Total ordering using a sequencer

1. Algorithm for group member *p*

On initialization: $r_g := 0$;

To TO-multicast message m to group g
B-multicast($g \cup \{sequencer(g)\}$, <*m, i*>);

On B-deliver(<*m, i*>) *with g = group*(*m*)
Place <*m, i*> in hold-back queue;

On B-deliver(<"order", *i, S*>) *with g = group*(*m*)
wait until <*m, i*> in hold-back queue and $S = r_g + 1$;
TO-deliver m; // (after deleting it from the hold-back queue)
$r_g = S$;

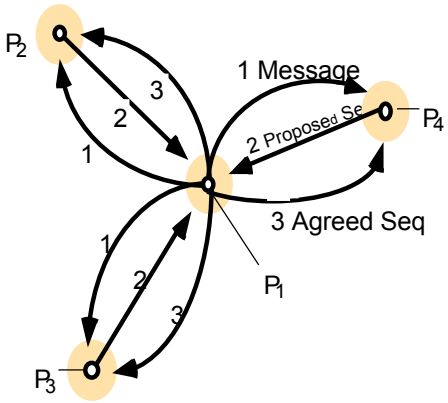
B-deliver simply means that the message is guaranteed to be delivered if the multicaster does not crash

2. Algorithm for sequencer of *g*

On initialization: $s_g := 0$;

On B-deliver(<*m, i*>) *with g = group*(*m*)
B-multicast(*g*, <"order", *i*, s_g>);
$s_g := s_g + 1$;

The ISIS algorithm for total ordering



- Each process keeps the largest agreed sequence number it has observed (O) for the group and its own largest proposed sequence number (A)
- Each process replies to a message from p with a proposed sequence number that is one larger than max(O,A)
- p collects all proposed sequence numbers and selects the largest one as the next agreed sequence number

Causal ordering using vector timestamps

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization
$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

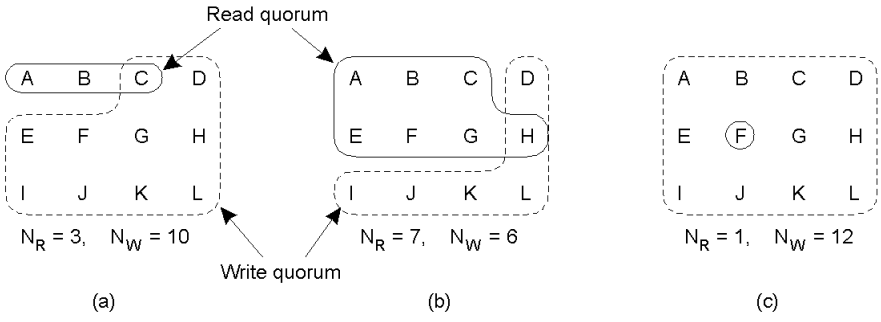
To CO-multicast message m to group g
$V_i^g[i] := V_i^g[i] + 1$;
B-multicast$(g, <V_i^g, m>)$;

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$
place $<V_j^g, m>$ in hold-back queue;
wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ $(k \neq j)$;
CO-deliver m; // after removing it from the hold-back queue
$V_i^g[j] := V_i^g[j] + 1$;

Quorum-based Protocols

- Assign a number of votes to each replica
- Let N be the total number of votes
- Define R = read quorum, W=write quorum
- R+W > N
- W > N/2
- Only one writer at a time can achieve write quorum
- Every reader sees at least one copy of the most recent read (takes one with most recent version number)

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)

Possible Policies

- ROWA: R=1, W=N
  - Fast reads, slow writes (and easily blocked)
- RAWO: R=N, W=1
  - Fast writes, slow reads (and easily blocked)
- Majority: R=W=N/2+1
  - Both moderately slow, but extremely high availability
- Weighted voting
  - give more votes to “better” replicas

Scaling

- None of the protocols for sequential consistency scale
- To read or write, you have to either
  - (a) contact a primary copy
  - (b) use reliable totally ordered multicast
  - (c) contact over half of the replicas
- All this complexity is to ensure sequential consistency
  - Note: even the protocols for causal consistency and FIFO consistency are difficult to scale if they use reliable multicast
- Can we weaken sequential consistency without losing some important features?

Highly available services

- Emphasis on giving clients access to the service with reasonable response times, even if some *results do not conform to sequential consistency*
- Examples
  - Gossip
    - Relaxed consistency
      - Causal update ordering
  - Bayou
    - Eventual consistency
    - Domain-specific conflict detection and resolution
  - Coda (file system)
    - Disconnected operation
    - Uses vector timestamps to detect conflicts

Distribution Protocols

How are updates propagated to replicas (independent of the consistency model)?

- State versus operations
  1. Propagate only notification of update, i.e, invalidation
  2. Transfer data from one copy to another
  3. Propagate the update *operation* to other copies
- Push versus pull protocols

Pull versus Push Protocols

A 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	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Leases: a hybrid form of update propagation that dynamically switches between pushing and pulling

- Server maintains state for a client for a TTL, i.e., while lease has not expired

Epidemic Protocols

- Update propagation for systems that only need eventual consistency
- Randomized approaches based on the theory of epidemics
  - *infective, susceptible*, and *removed* servers
- Anti-entropy propagation model
  - A server P picks another server Q at random, and exchanges updates
  - Three approaches
    - P only pushes updates to Q
    - P only pulls new updates from Q
    - P and Q send updates to each other
  - If many infective servers, pull-based approach is better
  - If only one infective server, either approach will eventually propagate all updates
  - **Rumor spreading (gossiping)** will speed up propagation
    - If server P has been updated, it randomly contacts Q and tries to push the update to Q; if Q was already updated by another server, with some probability (1/k), P loses interest in spreading the update any further

The Gossip system

- Guarantees
  - Each client obtains a consistent service over time, i.e. replica managers only provide a client with data that reflects the updates the client has observed so far
  - Relaxed consistency between replicas
    - primarily causal consistency, but support also provided for sequential consistency
    - choice up to the application designer

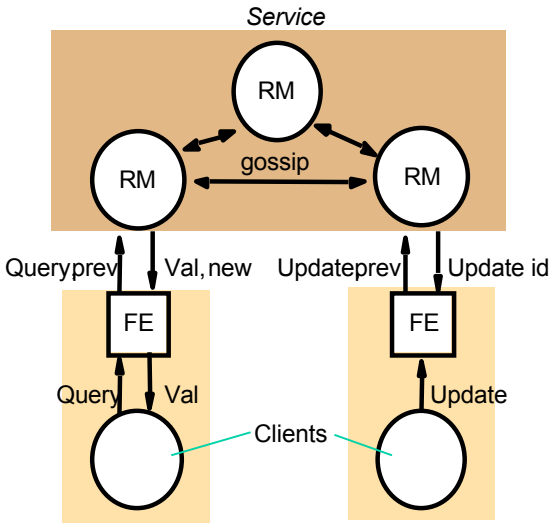
Display from bulletin board program

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

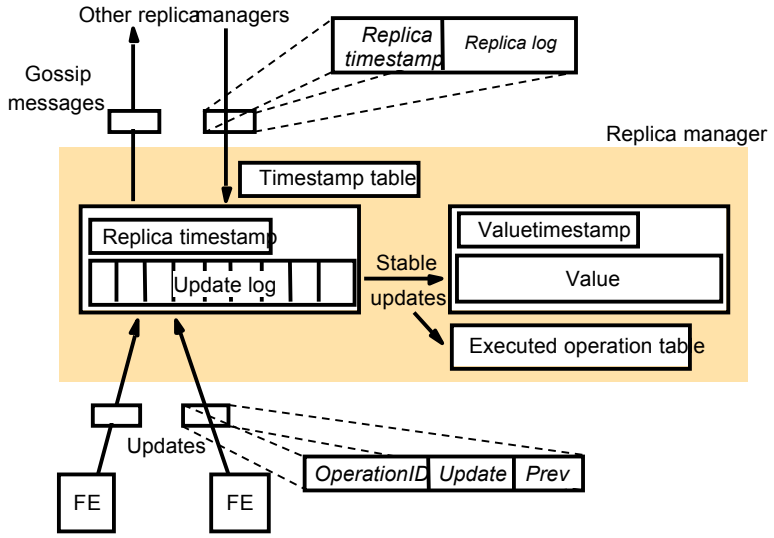
Gossip service operation

1. Request: Front End sends a query or update request to a replica manager that is reachable
2. Update Response: RM replies as soon as it receives update
3. Coordination: RM does not process the request until it can meet the required ordering constraints.
   - This may involve receiving updates from other replica managers in gossip messages
4. Execution
5. Query Response: If the request is a query, the RM replies at this point
6. Agreement: The replica managers update each other by exchanging gossip messages, which contain the most recent updates they have received. This is done in a *lazy* fashion

Query and update operations in a gossip service



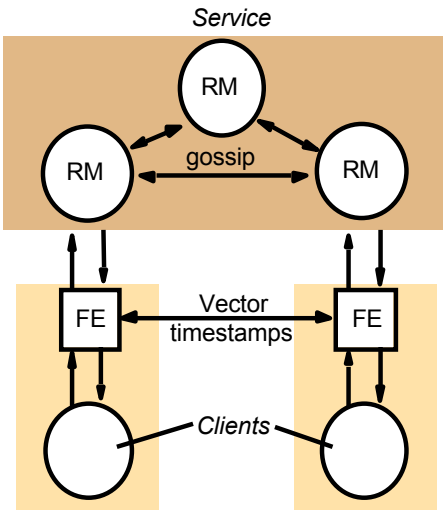
A gossip replica manager, showing its main state components



Version timestamps

- Each front end keeps a vector timestamp that reflects the version of the latest data values accessed by the front end
  - one timestamp for every replica manager
  - included in queries and updates
- Replica manager
  - value timestamp: reflects updates that have been applied (stable updates)
  - replica timestamp: reflects updates that have been placed in the log
- Example:
  - if query's timestamp = (2,4,6) and replica's value time stamp = (2,5,5), then RM is missing an update, and the query will not return until the RM receives that update (perhaps in a gossip message)

Front ends propagate their timestamps whenever clients communicate directly



Bayou: an approach for implementing eventual consistency

- System developed at Xerox PARC in the mid-90's
- Data replication for high availability despite disconnected operation
- Eventual consistency
  - if no updates take place for a long time, all replicas will gradually become consistent
- Domain specific conflict detection and resolution
  - appropriate for applications like shared calendars

Motivation for eventual consistency

- Sequential consistency requires that at every point, every replica has a value that could be the result of the globally-agreed sequential application of writes
- This does not require that all replicas agree at all times, just that they always take on the same sequence of values
- Why not allow temporary out-of-sequence writes?
  - Note: all forms of consistency weaker than sequential allow replicas to disagree forever
- We want to allow out-of-order operations, but only if the effects are temporary
- All writes eventually propagate to all replicas
- Writes, when they arrive, are applied in the same order at all replicas
  - Easily done with timestamps

Motivating Scenario: Shared Calendar

- Calendar updates made by several people
  - e.g., meeting room scheduling, or exec+admin
- Want to allow updates offline
- But conflicts can't be prevented
- Two possibilities:
  - Disallow offline updates?
  - Conflict resolution?

Two Basic Issues

- Flexible update propagation
- Dealing with inconsistencies
  - detecting and resolving conflicts
  - every Bayou update contains a dependency check and a merge procedure in addition to the operation's specification

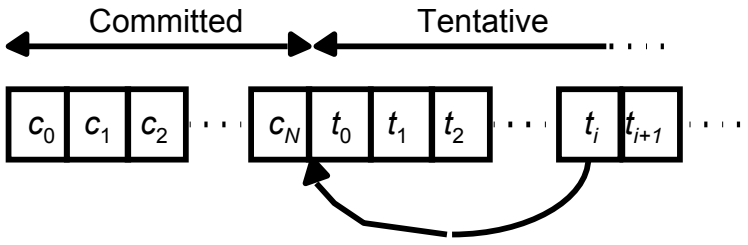
Conflict Resolution

- Replication **not** transparent to application
  - Only the application knows how to resolve conflicts
  - Application can do record-level conflict detection, not just file-level conflict detection
  - Calendar example: record-level, and easy resolution
- Split of responsibility:
  - Replication system: propagates updates
  - Application: resolves conflict
- Optimistic application of writes requires that writes be "undo-able"

Rolling Back Updates

- Keep log of updates
- Order by some timestamp
- When a new update comes in, place it in the correct order and reapply log of updates
- Need to establish when you can truncate the log
- Requires old updates to be “committed”, new ones tentative
- Committed order can be achieved by designating a replica manager as the primary replica manager

Committed and tentative updates in Bayou



Tentative update t_i becomes the next committed update and is inserted after the last committed update c_N.

Flexible Update Propagation

Requirements:

- Can deal with arbitrary communication topologies
- Can deal with low-bandwidth links
- Incremental progress (if get disconnected)
- Eventual consistency
- Flexible storage management
- Can use portable media to deliver updates
- Lightweight management of replica sets
- Flexible policies (when to reconcile, with whom, etc.)

Update Mechanism

- Updates time-stamped by the receiving server
- Writes from a particular server delivered in order
- Servers conduct anti-entropy exchanges
- State of database is expressed in terms of a timestamp vector
- By exchanging vectors, can easily identify which updates are missing
- Because updates are eventually "committed" you can be sure that certain updates have been spread everywhere

Session Guarantees

- When client move around and connects to different replicas, strange things can happen
  - Updates you just made are missing
  - Database goes back in time
- Design choice:
  - Insist on stricter consistency
  - Enforce some “session” guarantees

Read Your Writes

- Every read in a session should see all previous writes in that session
- Example error: deleted email messages re-appear

Monotonic reads

- Disallow reads to a DB less current than previous read
- Example error:
  - Get list of email messages
  - When attempting to read one, get “message doesn’t exist” error

Monotonic writes

- Writes must follow any previous writes that occurred within their session
- Example error:
  - Update to library made
  - Update to application using library made
  - Don’t want application depending on new library to show up where new library doesn’t show up

Writes Follow Reads

- If a write W followed a read R at a server X, then at all other servers
  - If W is in Y's database then any writes relevant to R are also there

Writes follow reads

- Affects users outside session
- Traditional write/read dependencies preserved at all servers
- Two guarantees: ordering and propagation
  - Order: If a read precedes a write in a session, and that read depends on a previous non-session write, then previous write will never be seen after second write at any server. It may not be seen at all.
  - Propagation: Previous write will actually have propagated to any DB to which second write is applied.

Writes follow reads, continued

- Ordering - example error:
  - Modification made to bibliographic entry, but at some other server original incorrect entry gets applied after fixed entry
- Propagation - example error:
  - Newsgroup displays responses to articles before original article has propagated there

Supporting Session Guarantees

- Responsibility of “session manager”, not servers
- Two sets:
  - Read-set: set of writes that are relevant to session reads
  - Write-set: set of writes performed in session
- Update dependencies captured in read sets and write sets
- Causal ordering of writes
  - Use Lamport clocks