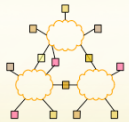
# 15-446 Distributed Systems Spring 2009



L-10 Consistency

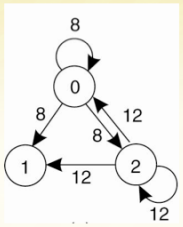
1

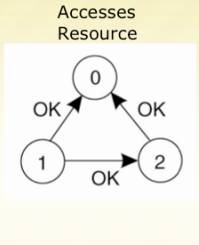
## Important Lessons

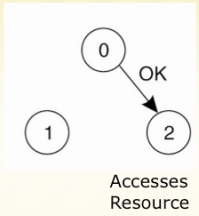
- Lamport & vector clocks both give a logical timestamps
  - Total ordering vs. causal ordering
- Other issues in coordinating node activities
  - Exclusive access to resources/data
  - Choosing a single leader

2

## A Distributed Algorithm (2)




- Two processes want to access a shared resource at the same moment.
- Process 0 has the lowest timestamp, so it wins
- When process 0 is done, it sends an OK also, so 2 can now go ahead.

3

## Today's Lecture - 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

4

# Outline

- Consistency Models
  - Data-centric
  - Client-centric
- Approaches for implementing Sequential Consistency
  - primary-backup approaches
  - active replication using multicast communication
  - quorum-based approaches

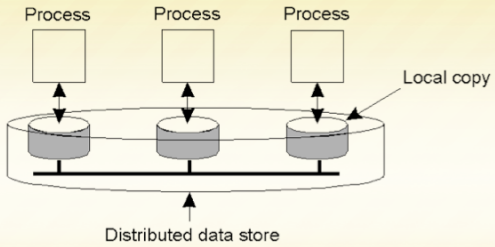
5

# 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 with respect to 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

6

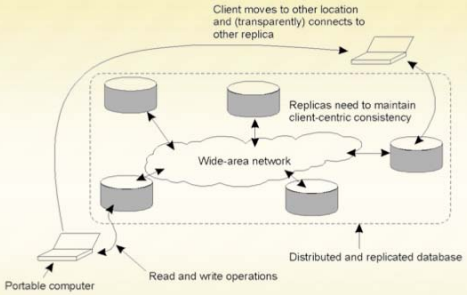
# Data-Centric Consistency Models




- The general organization of a logical data store, physically distributed and replicated across multiple processes. Each process interacts with its local copy, which must be kept 'consistent' with the other copies.

7

# Client-centric Consistency Models




- A mobile user may access different replicas of a distributed database at different times. This type of behavior implies the need for a view of consistency that provides guarantees for single client regarding accesses to the data store.

8

# Data-centric Consistency Models

- Strict consistency
- Sequential consistency
- Linearizability
- Causal consistency
- FIFO consistency
- Weak consistency
- Release consistency
- Entry consistency

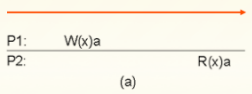
} use explicit synchronization operations

- Notation:
  - $W_i(x)a$ → process i writes value a to location x
  - $R_i(x)a$ → process i reads value a from location x

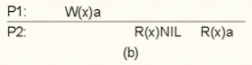
9

# Strict Consistency

Any read on a data item x returns a value corresponding to the result of the *most recent* write on x. "All writes are instantaneously visible to all processes"




A strictly consistent store




A store that is not strictly consistent.

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

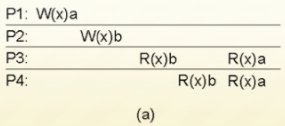
The problem with strict consistency is that it relies on *absolute global time* and is impossible to implement in a distributed system.

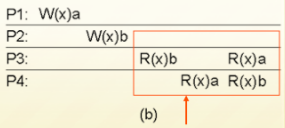
10

# 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 [Lamport, 1979].

Note: Any valid interleaving is legal but all processes must see the same interleaving.




- A sequentially consistent data store.
- A data store that is not sequentially consistent.

*P3 and P4 disagree on the order of the w*

11

# Sequential Consistency - 2

| Process P1 | Process P2 | Process P3 |
|---|---|---|
| x = 1;<br>print ( y, z); | y = 1;<br>print (x, z); | z = 1;<br>print (x, y); |

```
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

(a) (b) (c) (d)

(a)-(d) are all legal interleavings.

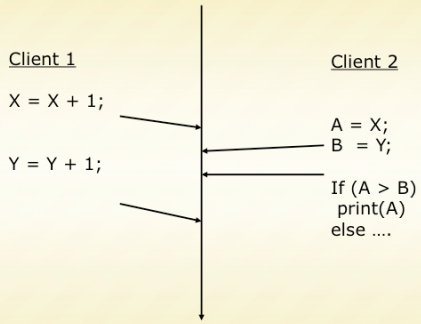
12

## Linearizability / Atomic Consistency

- 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 tsop1(x) < tsop2(y), then OP1(x) should precede OP2(y) in this sequence." [Herlihy & Wing, 1991]

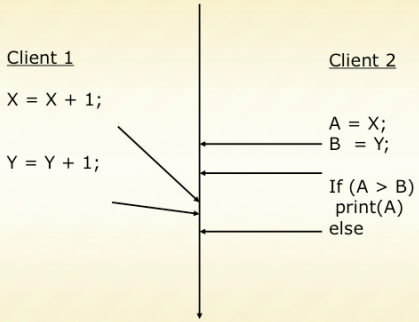
13

## Linearizable




14

## Not linearizable but sequentially consistent




15

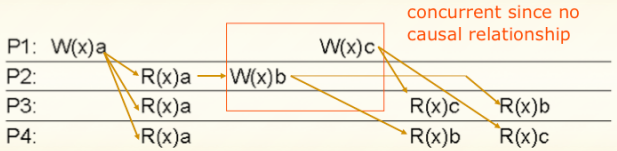
## Sequential Consistency vs. Linearizability

- Linearizability has proven useful for reasoning about program correctness but has not typically been used otherwise.
- Sequential consistency is implementable and widely used but has poor performance.
- To get around performance problems, weaker models that have better performance have been developed.

16

# Causal Consistency - 1

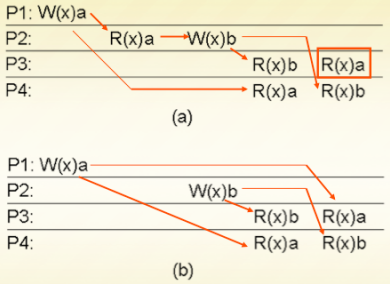
Necessary condition: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.




This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.
Can be implemented with vector clocks.

17

# Causal Consistency - 2




a) A violation of a causally-consistent store. The two writes are NOT concurrent because of the $R_2(x)a$.
b) A correct sequence of events in a causally-consistent store ($W_1(x)a$ and $W_2(x)b$ are concurrent).

18

# 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. Only requirement in this example is that P2's writes are seen in the correct order. FIFO consistency is easy to implement.

19

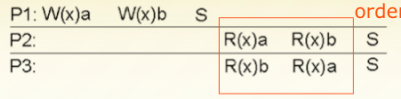
# Weak Consistency - 1

- Uses a synchronization variable with one operation synchronize(S), which causes all writes by process P to be propagated and all external writes propagated to P.
- Consistency is on groups of operations
- Properties:
  1. Accesses to synchronization variables associated with a data store are sequentially consistent (i.e. all processes see the synchronization calls in the same order).
  2. No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere.
  3. No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

20

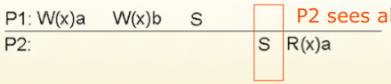
# Weak Consistency - 2

P2 and P3 have not synchronized, so no guarantee about what order they see.




(a)

This S ensures that P2 sees all updates



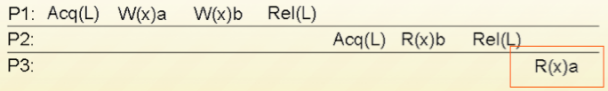

(b)

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

21

# Release Consistency

- Uses two different types of synchronization operations (*acquire* and *release*) to define a critical region around access to shared data.
- Rules:
  - Before a read or write operation on shared data is performed, all previous *acquire*s 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).



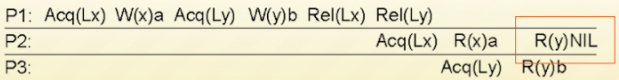

No guarantee since operations not used.

22

# Entry Consistency

Associate locks with individual variables or small groups.
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.




No guarantees since y is not acquired.

23

# 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.

24

## Outline

- Consistency Models
  - Data-centric
  - Client-centric
- Approaches for implementing Sequential Consistency
  - primary-backup approaches
  - active replication using multicast communication
  - quorum-based approaches

25

## Consistency Protocols

- Remember that a consistency model is a contract between the process and the data store. If the processes obey certain rules, the store promises to work correctly.
- A consistency protocol is an implementation that meets a consistency model.

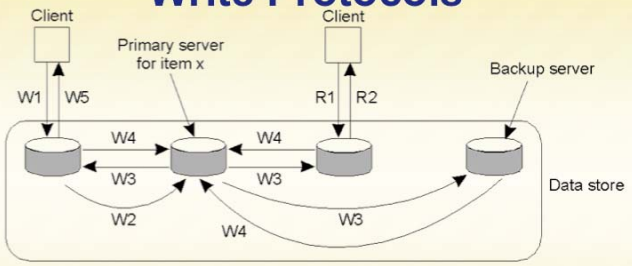
26

## Mechanisms for Sequential Consistency

- Primary-based replication protocols
  - Each data item has associated primary responsible for coordination
  - Remote-write protocols
  - Local-write protocols
- Replicated-write protocols
  - Active replication using multicast communication
  - Quorum-based protocols

27

## Primary-based: Remote-Write Protocols




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

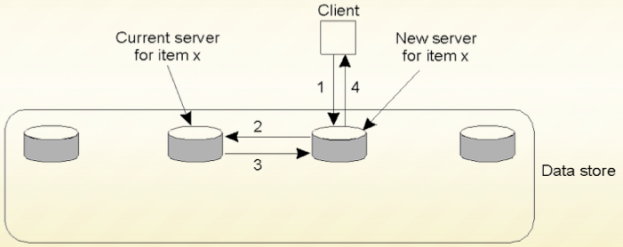
R1. Read request
R2. Response to read

- The principle of primary-backup protocol.

28

## Primary-based: Local-Write Protocols (1)

- Primary-based local-write protocol in which the single copy of the shared data is migrated between processes. One problem with approach is keeping track of current location of data.

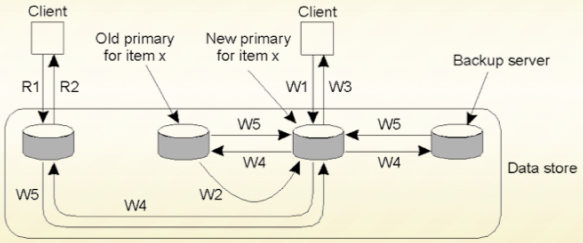



1. Read or write request
2. Forward request to current server for x
3. Move item x to client's server
4. Return result of operation on client's server

29

## Primary-based: Local-Write Protocols (2)

- Primary-backup protocol where replicas are kept but in which the role of primary migrates to the process wanting to perform an update. In this version, clients can read from non-primary copies.




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

30

## Replica-based protocols

- Active replication: Updates are sent to all replicas
- Problem: updates need to be performed at all replicas in same order. Need a way to do totally-ordered multicast
- Problem: invocation replication

31

## 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
- Causal ordering
  - use vector timestamps

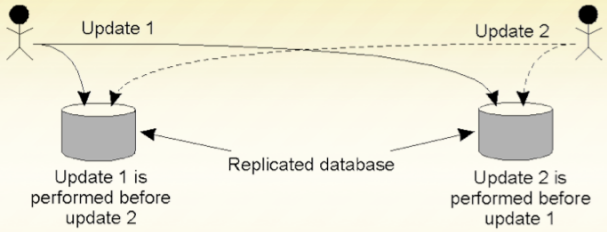
32

# Totally Ordered Multicast

- Use Lamport timestamps
- Algorithm
  - Message is timestamped with sender's logical time
  - Message is multicast (including sender itself)
  - When message is received
    - It is put into local queue
    - Ordered according to timestamp
    - Multicast acknowledgement
  - Message is delievered to applications only when
    - It is at head of queue
    - It has been acknowledged by all involved processes
  - Lamport algorithm (extended) ensures total ordering of events

33

# Totally-Ordered Multicasting



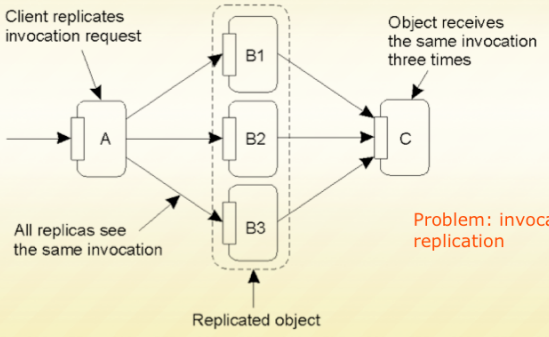

At DB 1:
Received request$_1$ and request$_2$ with timestamps 4 and 5, as well as acknowledgements from ALL processes of request$_1$. DB1 performs request$_1$.

At DB 2:
Received request$_2$ with timestamp 5, but not the request$_1$ with timestamp 4 and acknowledgements from ALL processes of request$_2$. DB2 performs request$_2$.

Why can't this happen??

34

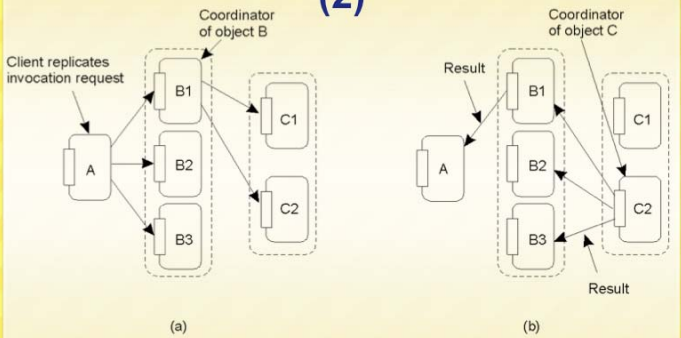
# Replica-based: Active Replication (1)




Problem: invocation replication

35

# Replica-based: Active Replication (2)




Assignment of a coordinator for the replicas can ensure that invocations are not replicated.

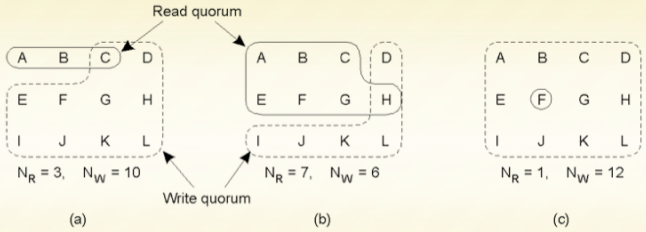
36

## Quorum-based protocols - 1

- 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)

37

## Quorum-based protocols - 2




Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, known as ROWA (read one, write all)

38

## Quorum-based protocols - 3

- 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

39

## 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

40