CSE 5306
Distributed Systems

Consistency and Replication

Jia Rao

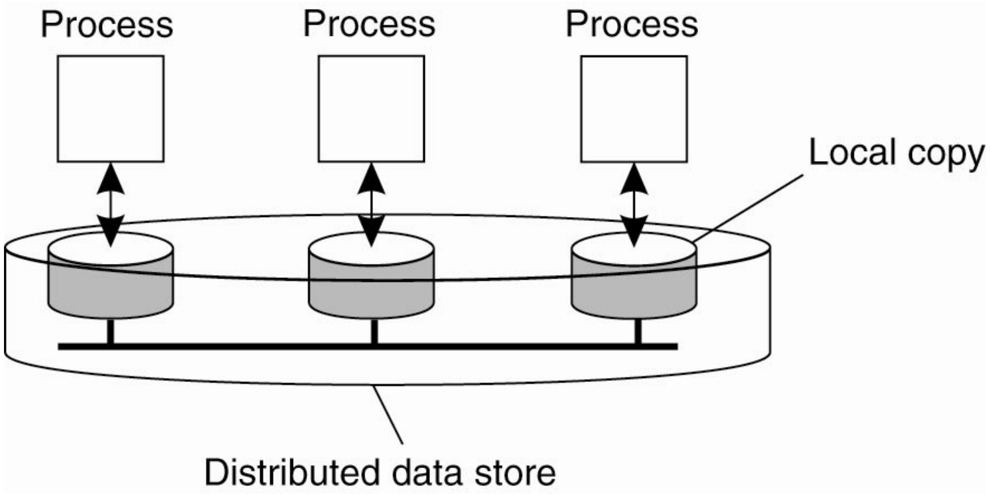
http://ranger.uta.edu/~jrao/

Reasons for Replication

- Data is replicated for
  - the reliability of the system
- Servers are replicated for performance
  - Scaling in numbers
  - Scaling in geographical area
- Dilemma
  - Gain in performance
  - Cost of maintaining replication
    - Keep the replicas up to date and ensure consistency

Data-centric Consistency Model (1/2)

- Consistency is often discussed in the context of read and write on
  - ✓ Shared memory, shared databases, shared files
- A more general term is: data store
  - ✓ A data store is distributed across multiple machines
  - ✓ Each process can access a local copy of the entire data store



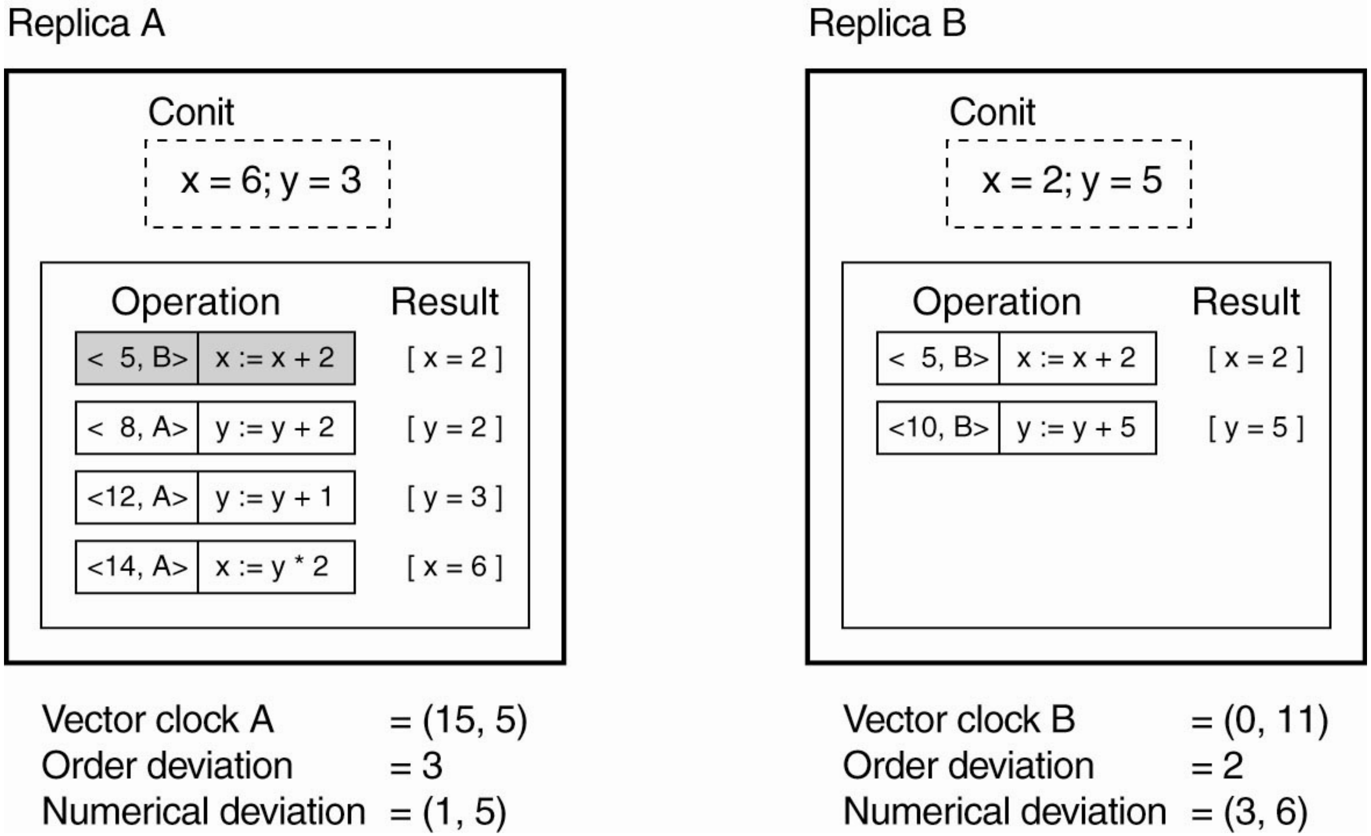
Data-centric Consistency Model (2/2)

- A consistency model is essentially a contract between processes and the data store
  - ✓ A process that performs a read operation on a data item expects the value written by the last write operation
- However, due to the lack of a global clock, it is hard to define which write operation is the last one

Continuous Consistency

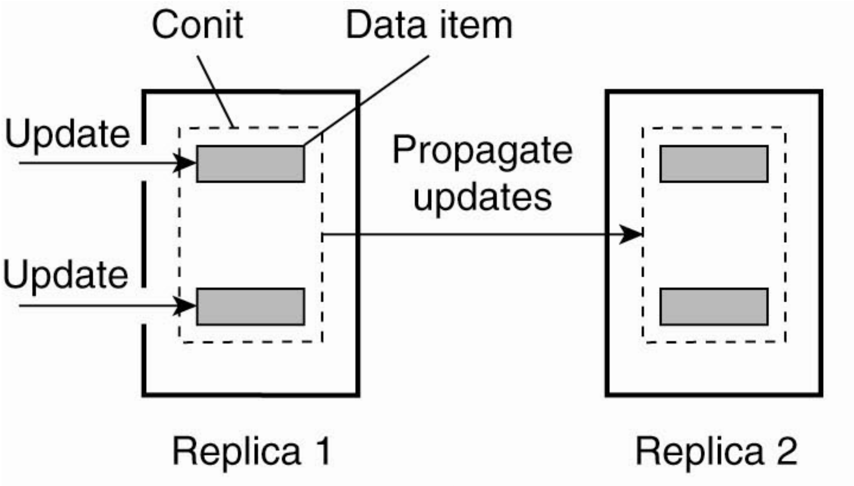
- Defines three independent axes of inconsistency
  - ✓ Deviation in numerical values between replicas
    - E.g., the number and values of updates
  - ✓ Deviation in staleness between replicas
    - Related to the last update
  - ✓ Deviation with respect to the ordering of updates
    - E.g., the number of uncommitted updates
- Measure inconsistency with “conit”
  - ✓ A conit specifies the unit over which consistency is to be measured
  - ✓ E.g., a record representing a stock, a weather report

Measuring Inconsistency: An Example

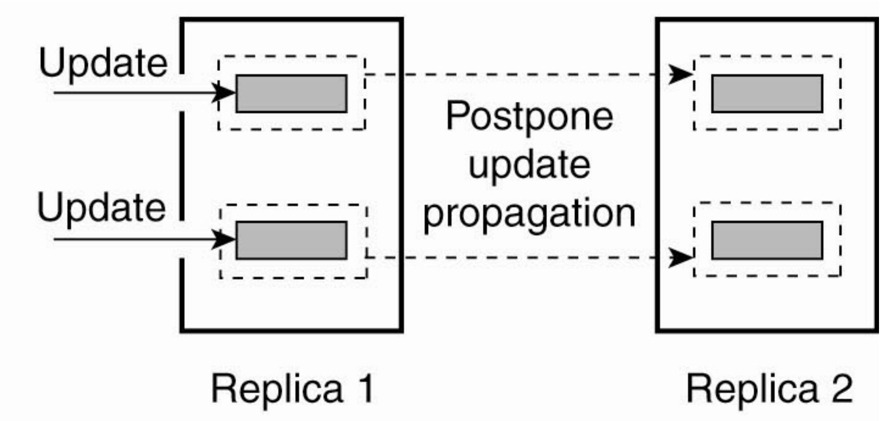


An example of keeping track of consistency deviations [Yu and Vahdat, 2002]

Conit Granularity



(a)



(b)

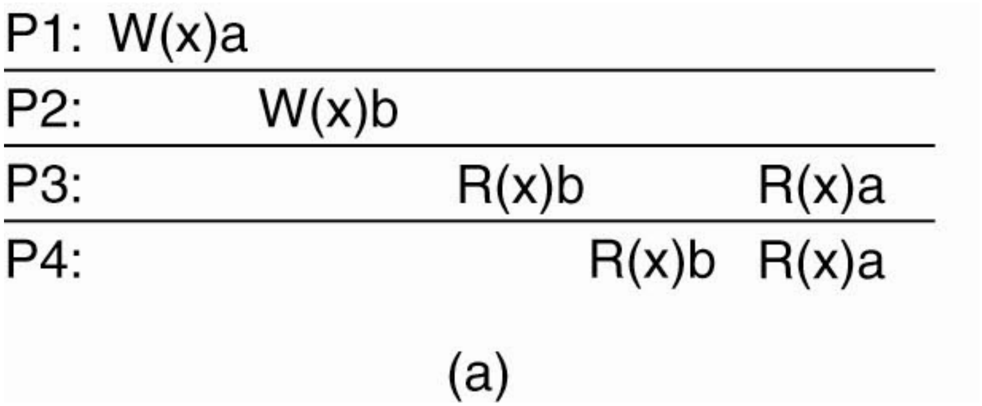
- Requirement: two replicas may differ in no more than ONE update
  - ✓ (a) Two updates lead to update propagation
  - ✓ (b) No update propagation is needed

Sequential Consistency

- The symbols for read and write operations

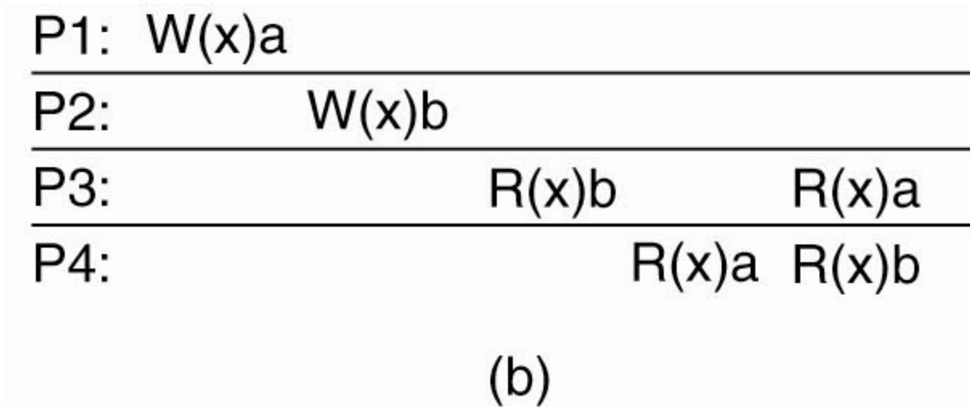
- A data store is sequentially consistent if
  - ✓ The result of any execution is the same, as if
  - ✓ The (read and write) operations on the data store were executed in some sequential order, and
  - ✓ The operations of each individual process appear in this sequence in the order specified by its program

Example 1



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

(a)

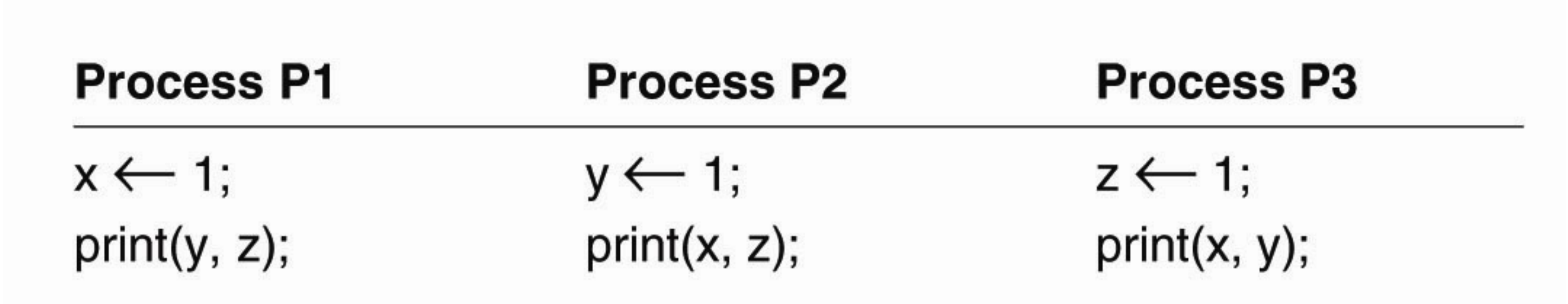


P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

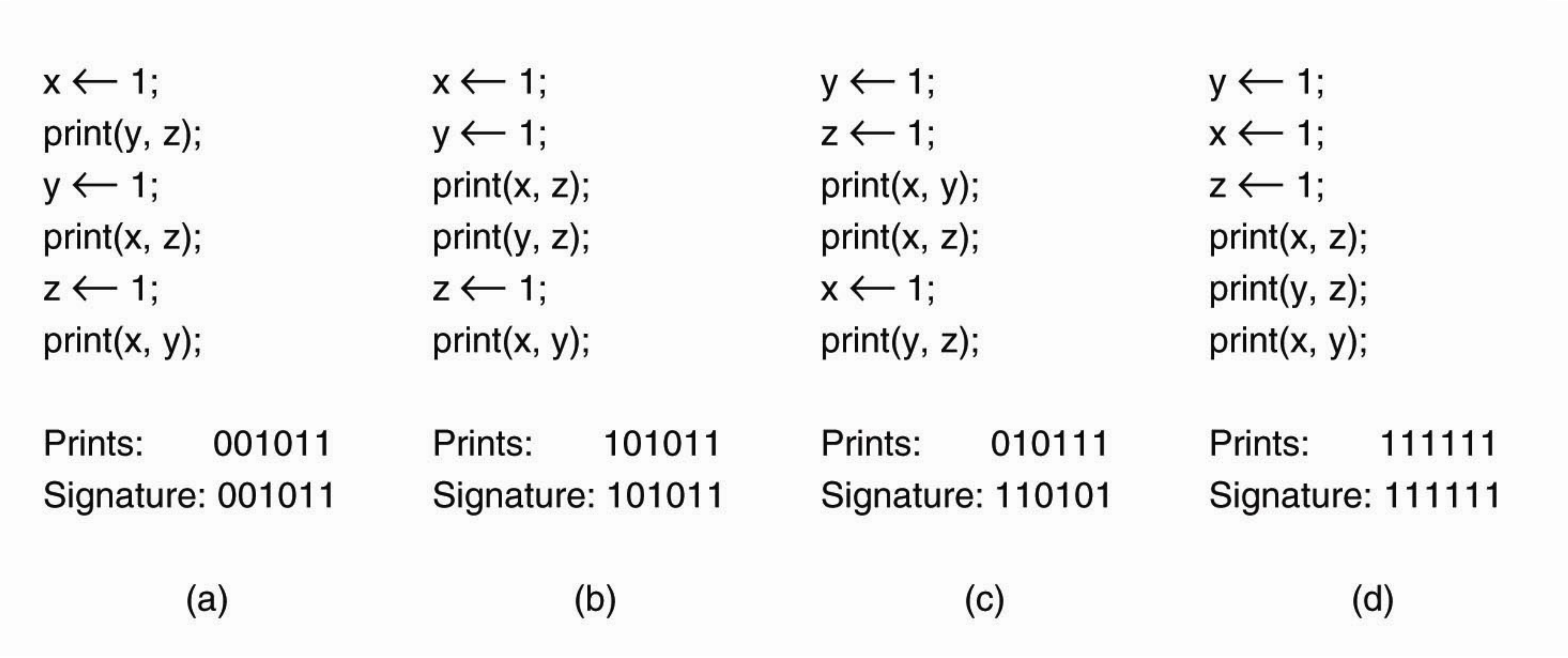
(b)

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

Example 2



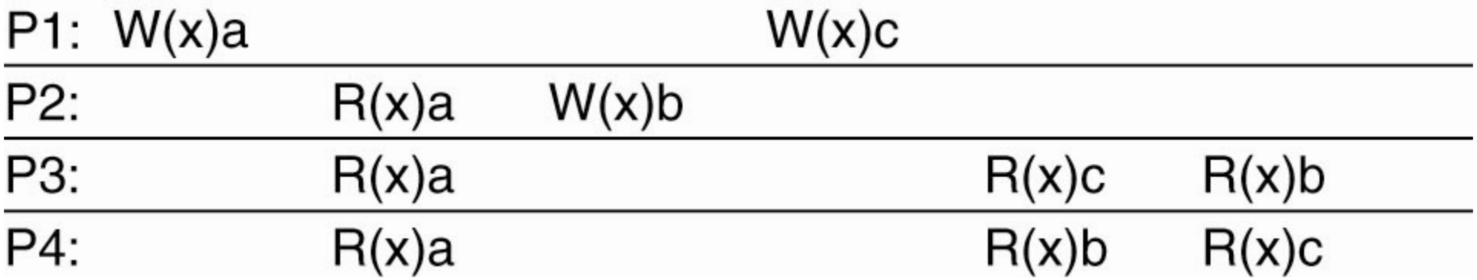
Process P1	Process P2	Process P3
x ← 1;	y ← 1;	z ← 1;
print(y, z);	print(x, z);	print(x, y);



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
Signature: 001011	Signature: 101011	Signature: 110101	Signature: 111111
(a)	(b)	(c)	(d)

Casual Consistency

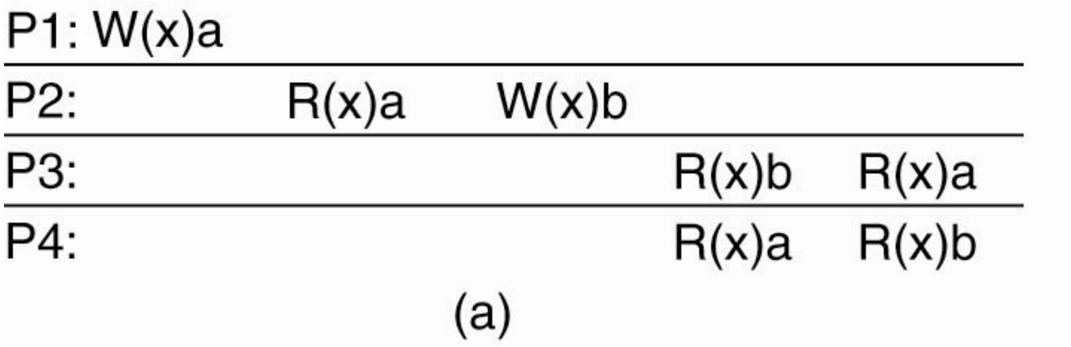
- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition
  - ✓ Writes that are potentially causally related
    - Must be seen by all processes in the same order
  - ✓ Concurrent writes
    - May be seen in a different order on different machines



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 a sequentially consistent store.

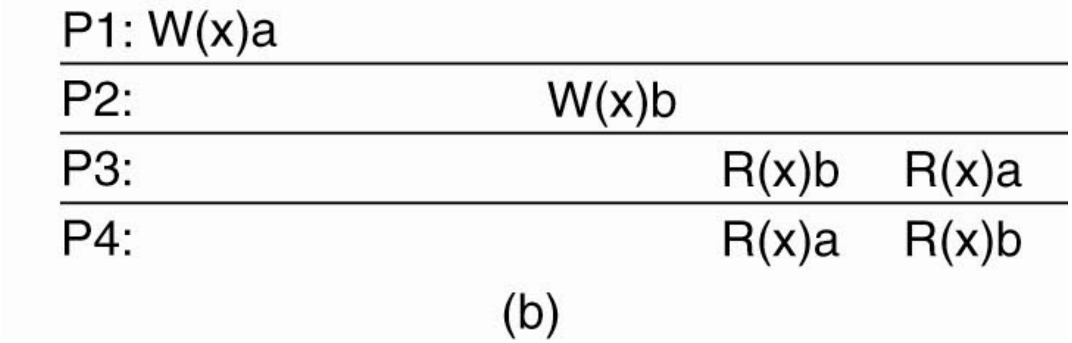
Another Example



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 causally-consistent store.



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

(b)

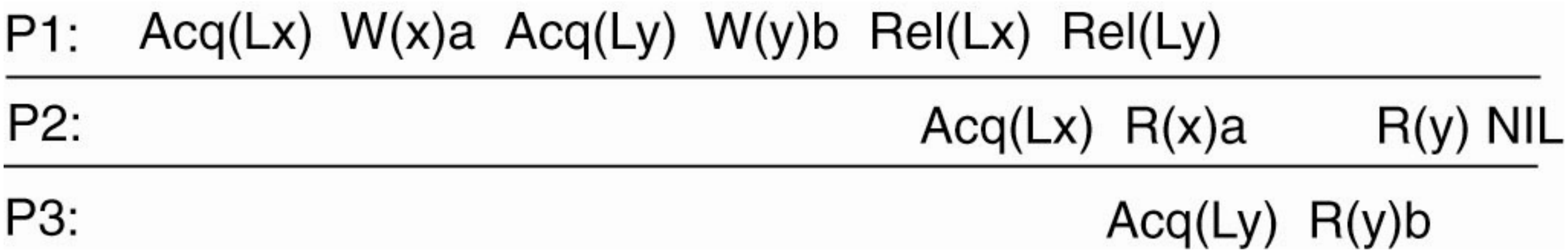
(b) A correct sequence of events in a causally-consistent store.

Grouping Operations

- Sequential and causal consistency is defined at the level of read and write operations
  - ✓ However, in practice, such granularity does not match the granularity provided by the application
    - Concurrency is often controlled by synchronization methods such as mutual exclusion and transactions
- A series of read/write operations, as one single unit, are protected by synchronization operations such as ENTER_CS and LEACE_CS
  - ✓ This atomically executed unit then defines the level of granularity in real-world applications

Entry Consistency

- It requires
  - ✓ The programmer to use acquire and release at the start and end of each critical section, respectively
  - ✓ Each ordinary shared variable to be associated with some synchronization variable



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

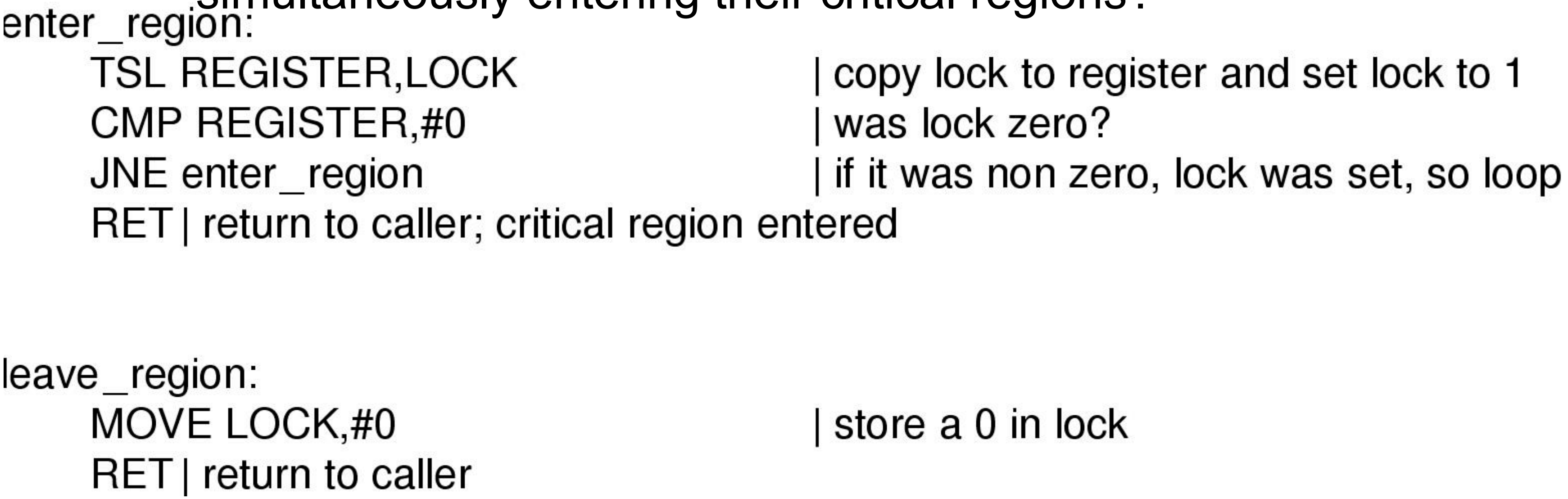
A valid event sequence for entry consistency.

Mutual Exclusion on Shared Memory

- Disabling interrupts:
  - OS technique, not users’
  - multi-CPU?
- Lock variables:
  - test-set is a two-step process, not atomic
- Busy waiting:
  - continuously testing a variable until some value appears (spin lock)

Busy Waiting: TSL

- TSL (Test and Set Lock)
  - Indivisible (atomic) operation, how? Hardware (multi-processor)
  - How to use TSL to prevent two processes from simultaneously entering their critical regions?



```
enter_region:
    TSL REGISTER,LOCK              | copy lock to register and set lock to 1
    CMP REGISTER,#0                | was lock zero?
    JNE enter_region               | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0                   | store a 0 in lock
    RET | return to caller
```

Entering and leaving a critical region using the TSL instruction

Mutexes

- Mutex:
  - a variable that can be in one of two states: unlocked or locked

```
mutex_lock:
        TSL REGISTER,MUTEX              | copy mutex to register and set mutex to 1
        CMP REGISTER,#0                 | was mutex zero?
        JZE ok                          | if it was zero, mutex was unlocked, so return
        CALL thread_yield               | mutex is busy; schedule another thread
        JMP mutex_lock                  | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0                   | store a 0 in mutex
        RET | return to caller
```

Give other chance to run so as to save self;
What is mutex_trylock()?

Monitors

- Monitor: a higher-level synchronization primitive
    - Only one process can be active in a monitor at any instant, with compiler’s help; thus, how about to put all the critical regions into monitor procedures for mutual exclusion?

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

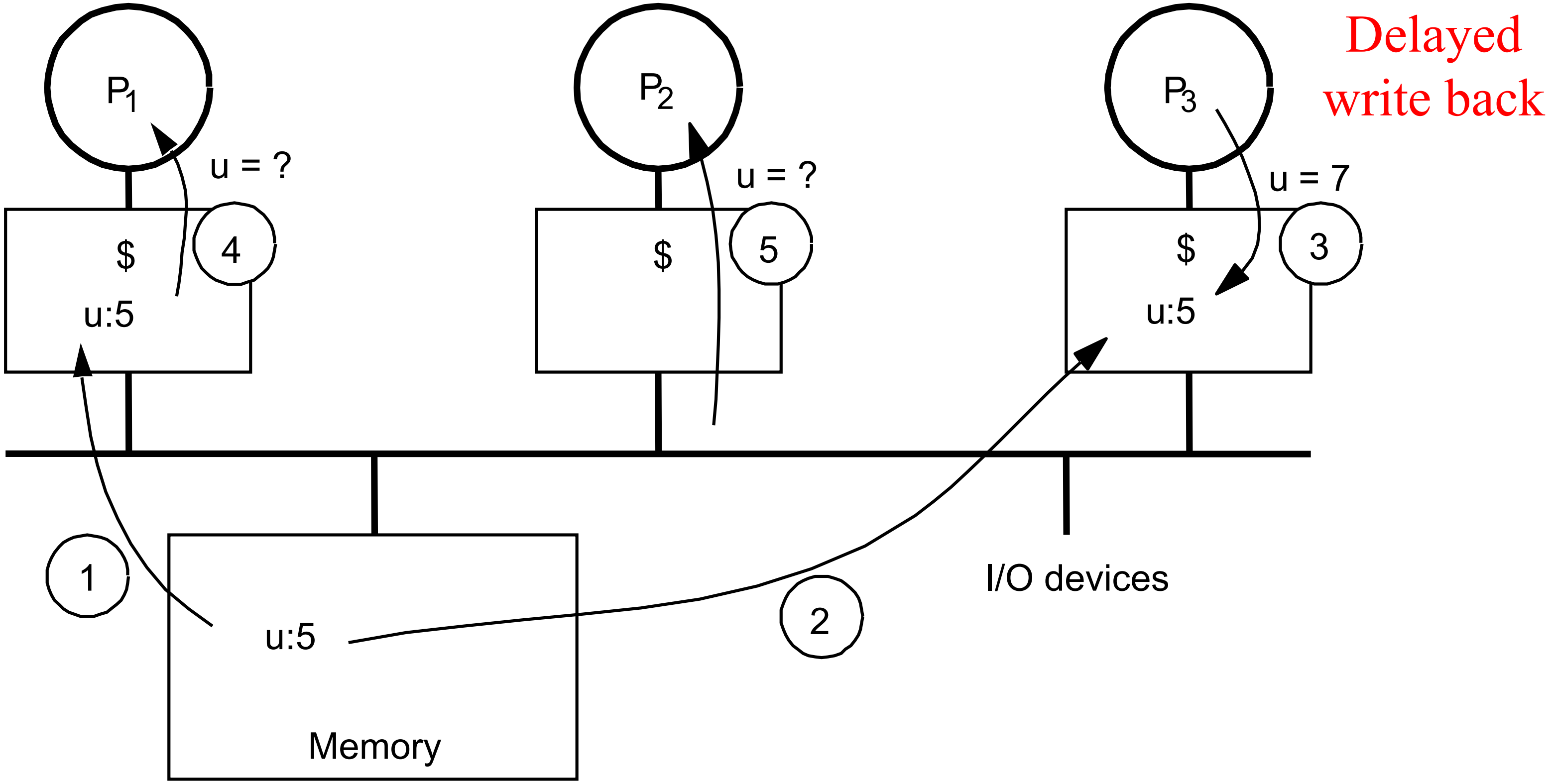
But, how processes block when they cannot proceed?

Condition variables, and two operations: *wait()* and *signal()*

Consistency v.s. Coherence

- Consistency deals with a set of processes operating on
  - ✓ A set of data items (they may be replicated)
  - ✓ This set is consistent if it adheres to the rules defined by the model
- Coherence deals with a set of processes operating on
  - ✓ A single data item that is replicated at many places
  - ✓ It is coherent if all copies abide to the rules defined by the model

Cache Coherence



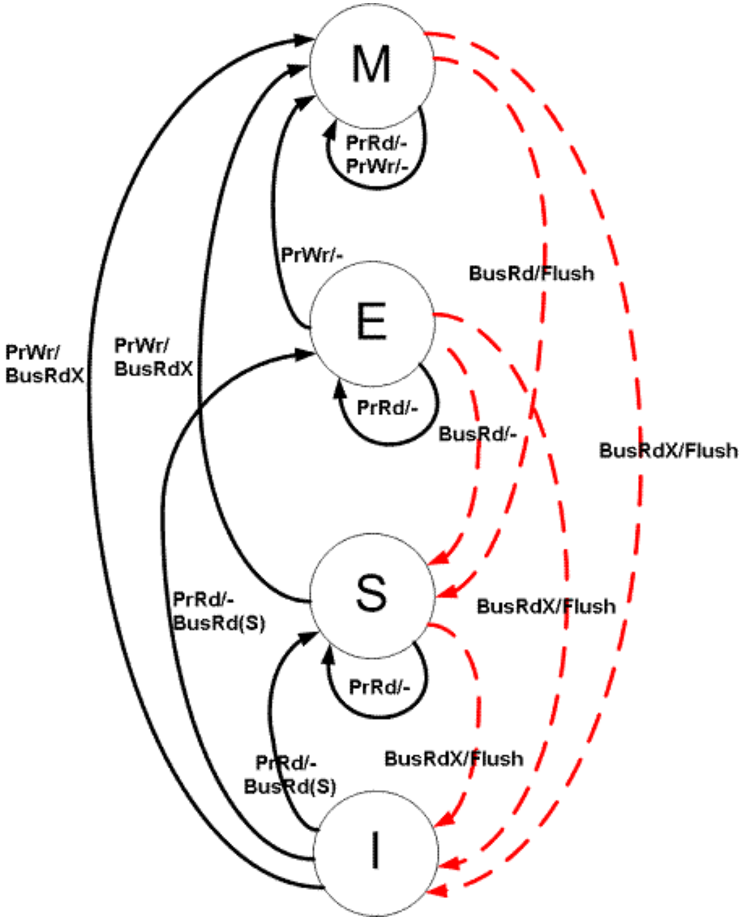
Processors see different values for u after event 3

The MESI Protocol (1/2)

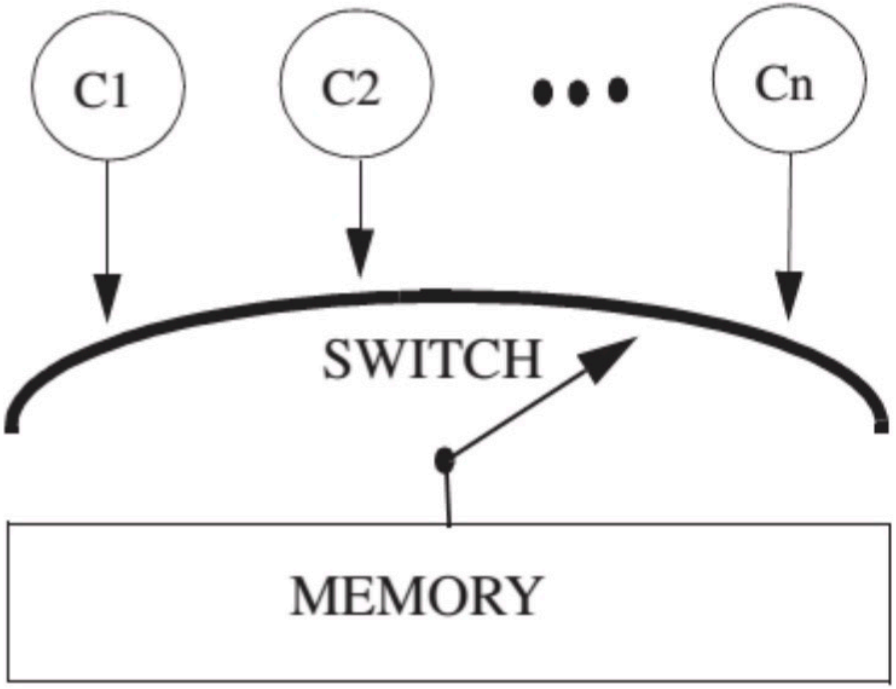
- All coherence related activities are broadcasted to all processors
- Every cache line has one of the four states
  - ✓ Modified — cache line is present only in the current cache, is dirty and has been modified from the value in memory
  - ✓ Exclusive — cache line is present only in the current cache, and is clean
  - ✓ Shared — cache line may be stored in other caches, and is clean
  - ✓ Invalid — cache line is invalid

The MESI Protocol (2/2)

- Processor events
  - ✓ PrRd — read
  - ✓ PrWr — write
- Bus transactions
  - ✓ BusRd — read request from the bus without intent to modify
  - ✓ BusRdX — read request from the bus with the intent to modify
  - ✓ BusWB — write line out to memory
- Access a cache line in I state will cause a cache miss
- A write can only be performed if the cache line is in E or M states. If it is in S state, the processor broadcasts a request for ownership (RFO) to invalidate other copies



Implementing SC on Multi-cores



Each core Ci seeks to do its next memory access in its program order <p.

The switch selects one core, allows it to complete one memory access, and repeats; this defines memory order <m.

Formulating SC

- All cores insert their loads and stores into the memory order (<m) respecting their program order (<p), regardless of whether they are to the same or different addresses.
  - ✓ If L(a) <p L(b) → L(a) <m L(b) /* load → load*/
  - ✓ If L(a) <p S(b) → L(a) <m S(b) /* load → store*/
  - ✓ If S(a) <p S(b) → S(a) <m S(b) /* store → store*/
  - ✓ If S(a) <p L(b) → S(a) <m L(b) /* store → load*/
- Every load gets its value from the latest store before it in global memory order to the same address

Too expensive

Total Store Order (TSO)

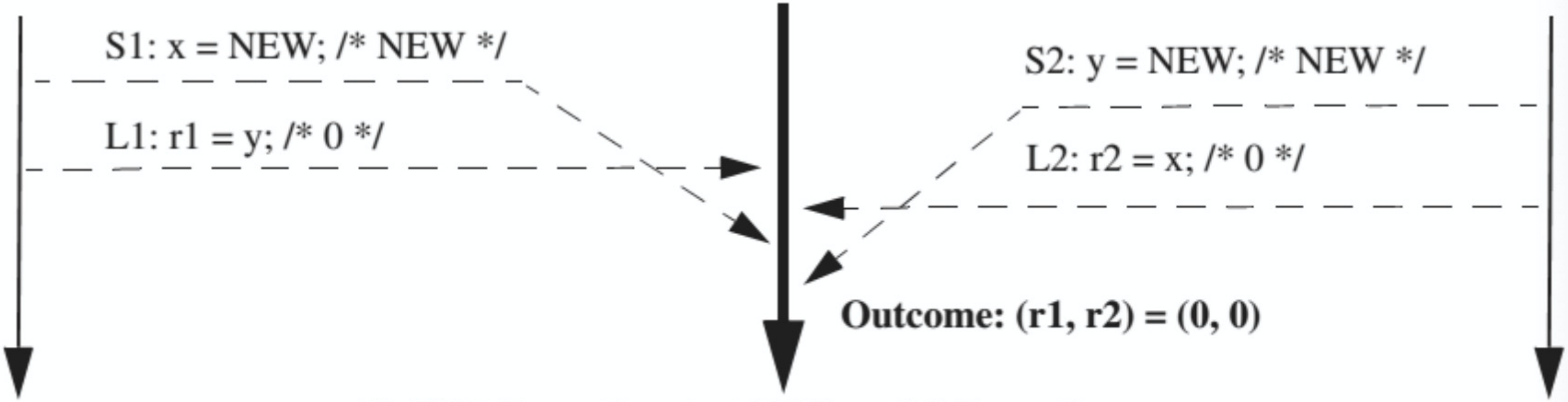
- Processors use write buffers to hold committed stores until the memory system can process them.
- A store enters the write buffer when the store commits, and a store exits the write buffer when the block to be written is in the cache in a read–write coherence state.

Formulating TSO

- All cores insert their loads and stores into the memory order (<m) respecting their program order (<p), regardless of whether they are to the same or different addresses.
  - ✓ If L(a) <p L(b) → L(a) <m L(b) /* load → load*/
  - ✓ If L(a) <p S(b) → L(a) <m S(b) /* load → store*/
  - ✓ If S(a) <p S(b) → S(a) <m S(b) /* store → store*/
  - ✓ If S(a) <p L(b) → S(a) <m L(b) /* store → load*/ no longer enforced

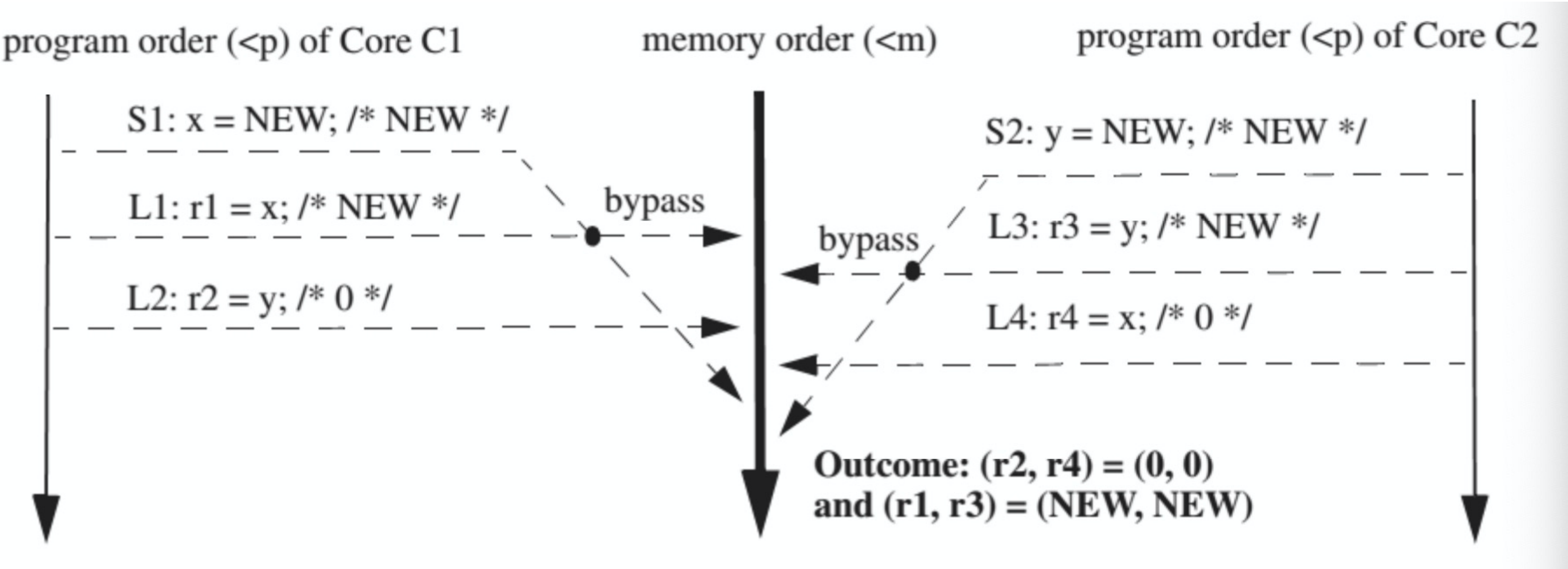
Comparing SC and TSO

Core C_1	Core C_2
S1: x = NEW	S2: y = NEW
L1: r1 = y	L2: r2 = x



(d) TSO Execution, but NOT an SC Execution

TSO Bypass Example



Eventual Consistency

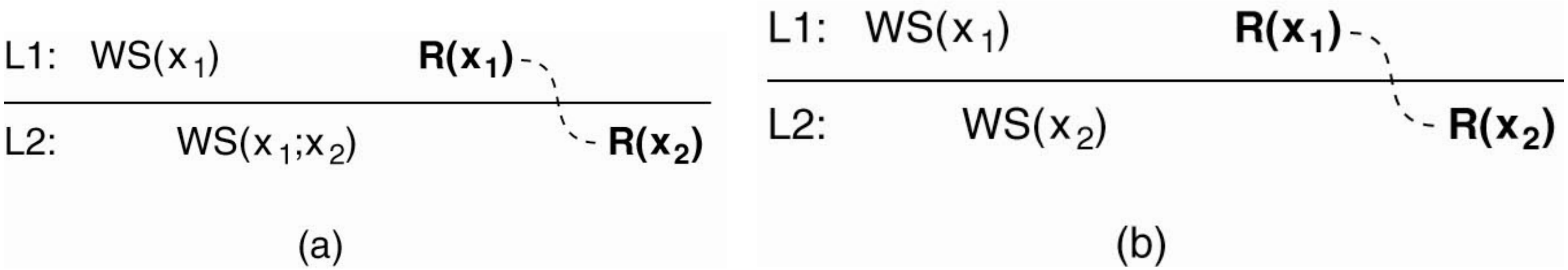
- In many distributed systems such as DNS and World Wide Web,
  - ✓ Updates on shared data can only be done by one or a small group of processes
  - ✓ Most processes only read shared data
  - ✓ A high-degree of inconsistency can be tolerated
- Eventual consistency
  - ✓ If no updates take place for a long time, all replicas will gradually become consistent
  - ✓ Clients are usually fine if they only access the same replica
- However, in some cases, clients may access different replicas
  - ✓ E.g., a mobile user moves to a different location
- Client-centric consistency:
  - ✓ Guarantee the consistency of access for a single client

Monotonic-Read Consistency

- A data store is said to provide monotonic-read consistency if the following condition holds:
  - ✓ If a process reads the value of a data item x, then
  - ✓ Any successive read operation on x by that process will always return
    - That same value or
    - A more recent value
- In other words
  - ✓ If a process has seen a value of x at time t, it will never see an older version of x at any later time

An Example

- Notations
  - ✓ $x_i[t]$: the version of x at local copy L_i at time t
  - ✓ $WS(x_i[t])$: the set of all writes at L_i on x since initialization



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

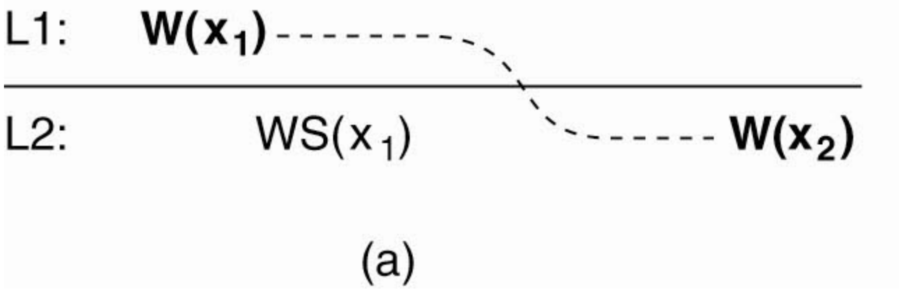
(a) A monotonic-read consistent data store.

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

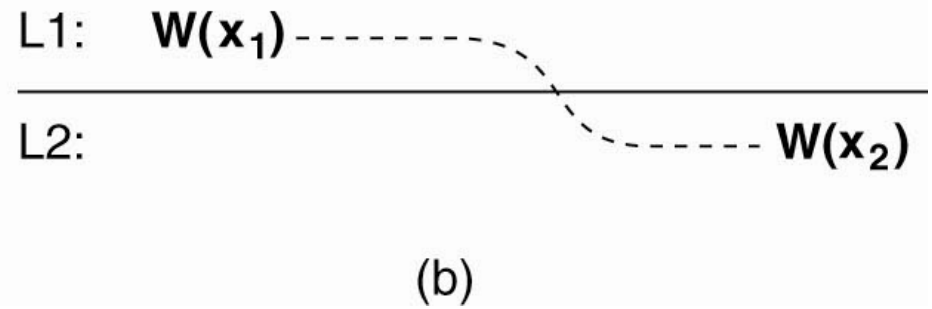
Monotonic-Write Consistency

- In a monotonic-write consistent store, the following condition holds
  - ✓ A write operation by a process on a data item x is completed before
    - Any successive write operation on x by the same process
- In other words
  - ✓ A write on a copy of x is performed only if this copy is brought up to date by means of
    - Any preceding write on x, which may take place at other replicas, by the same process

An Example



(a) A monotonic-write consistent data store.

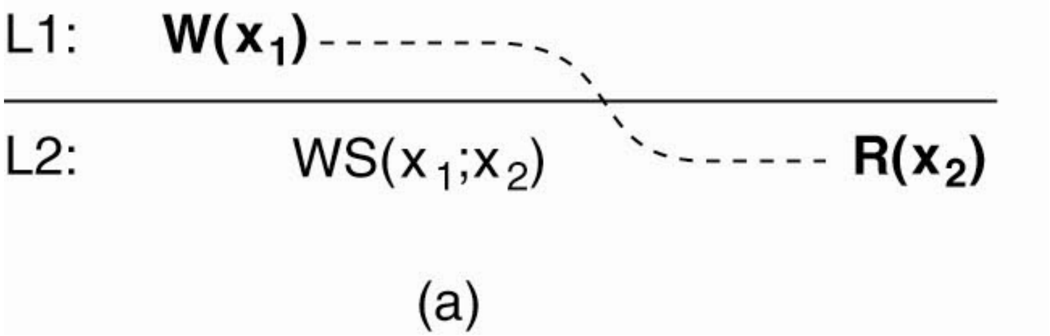


(b) A data store that is not.

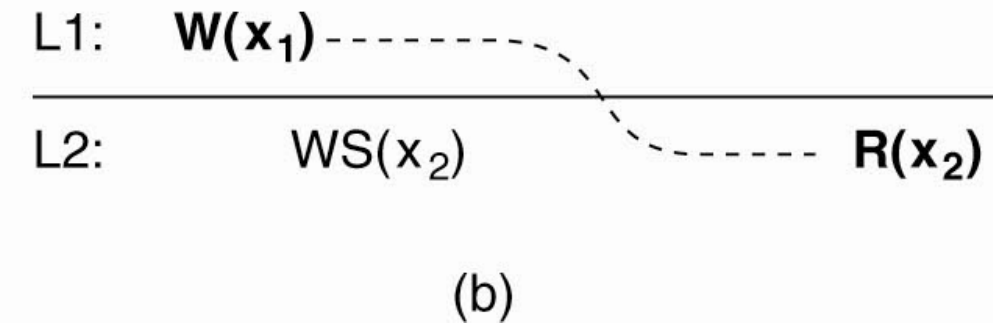
Read-Your-Write Consistency

- A data store is said to provide read-your-write consistency, if the following condition holds:
  - ✓ 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
- In other words,
  - ✓ A write operation is always completed before a successive read operation by the same process
    - No matter where the read takes place

An Example



(a) A data store that provides read-your-writes consistency.

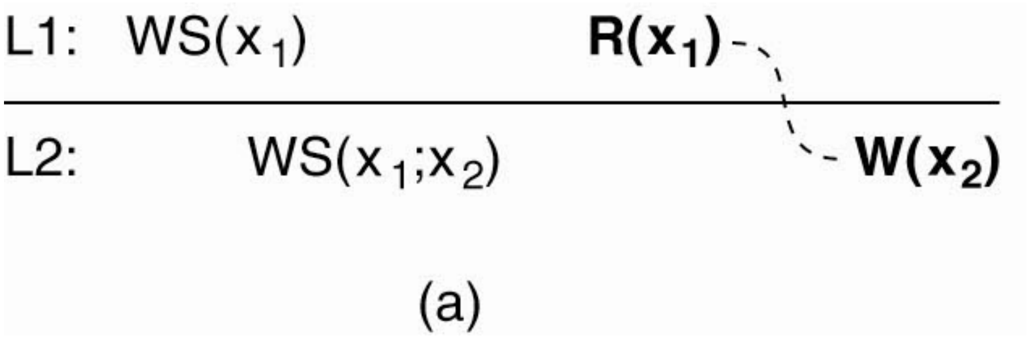


(b) A data store that does not.

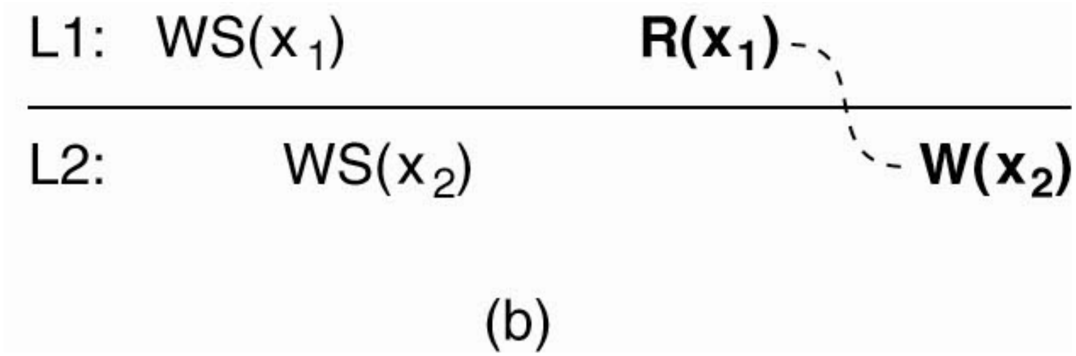
Write-Follow-Read Consistency

- A data store is said to provide write-follow-reads consistency, if the following holds:
  - ✓ 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
- In other words,
  - ✓ Any successive write operation by a process on a data item x will be performed on a copy of x that
    - Is up to date with the value most recently read by that process

An Example



(a) A writes-follow-reads consistent data store.



(b) A data store that does not

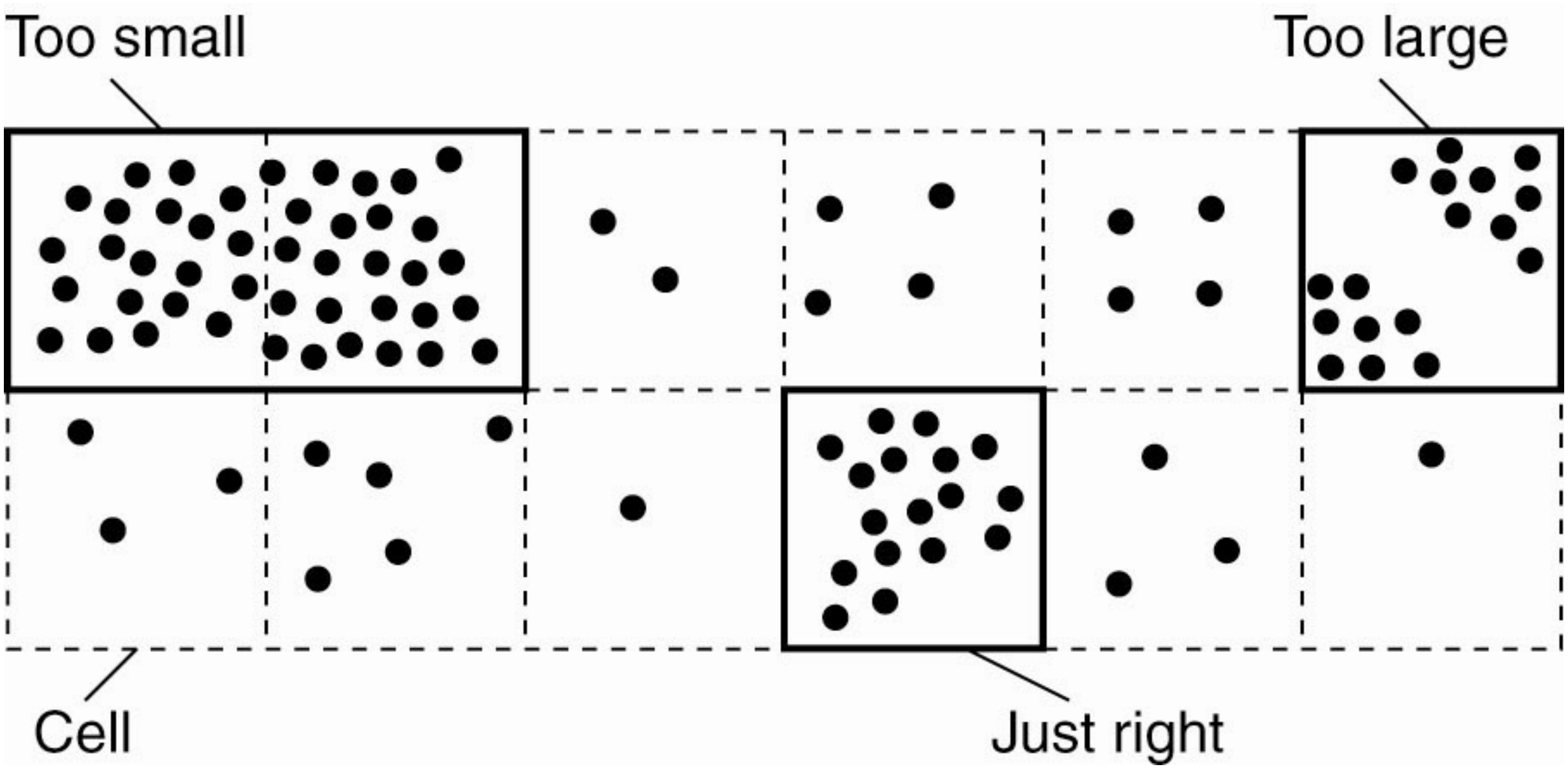
Replica Management

- Two key issues for distributed systems that support replication
- Where, when, and by whom replicas should be placed? Divided into two sub-problems:
  - ✓ Replica server placement: finding the best location to place a server that can host a data store
  - ✓ Content placement: find the best server for placing content
- Which mechanisms to use for keeping replicas consistent

Replica-Server Placement

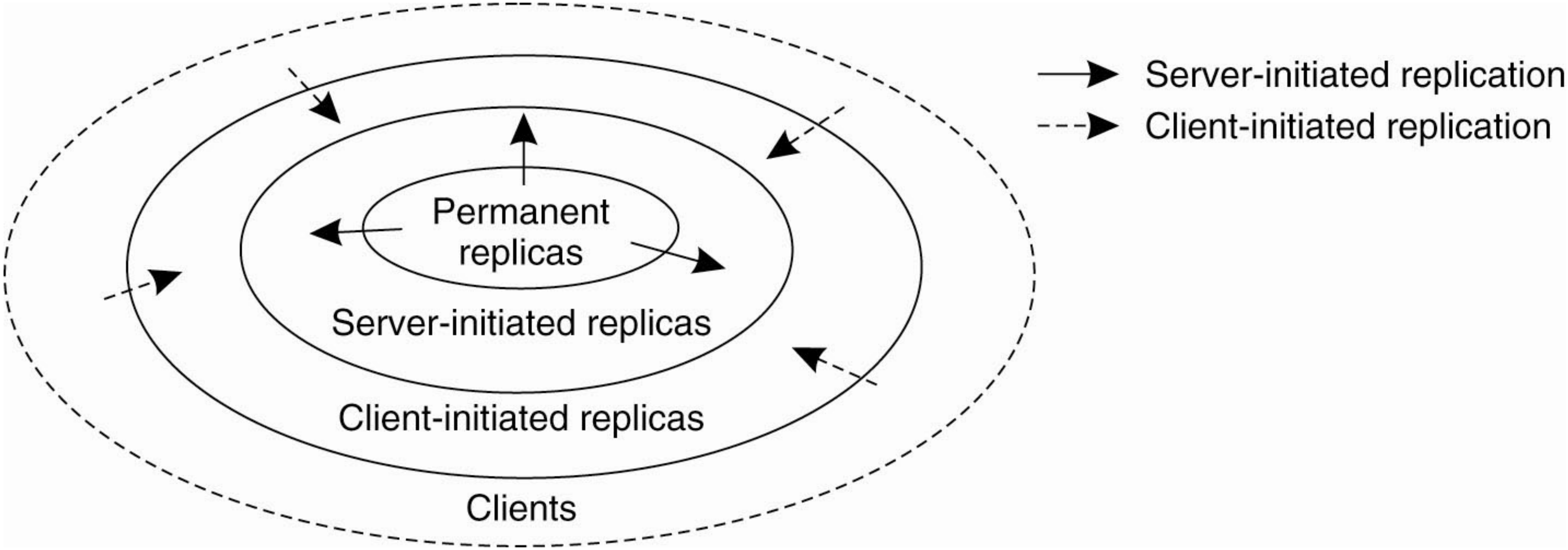
- Some typical approaches
  - ✓ Select K out of N: select the one that leads to the minimal average latency to all clients, and repeat
  - ✓ Ignore the client, only consider the topology, i.e., the largest AS, the second largest AS …
  - ✓ However, these approaches are very expensive
- Region-based approach
  - ✓ A region is identified to be a collection of nodes accessing the same content, but for which the internode latency is low

Region-based Approach



Choosing a proper cell size for server placement.

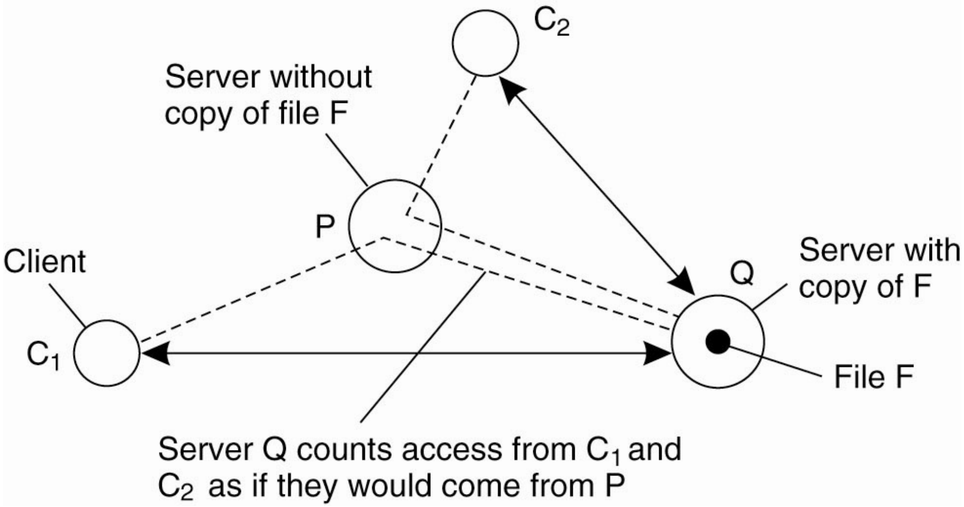
Content Replication and Placement



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

Server-Initiated Replicas

- Observe the client access pattern and dynamically add or remove replicas to improve performance
- One example algorithm
  - ✓ Count the access request of F from clients
  - ✓ If the request drops significantly, delete replica F
  - ✓ If a lot of requests from one certain location, replicate F at this location



Client-Initiated Replicas

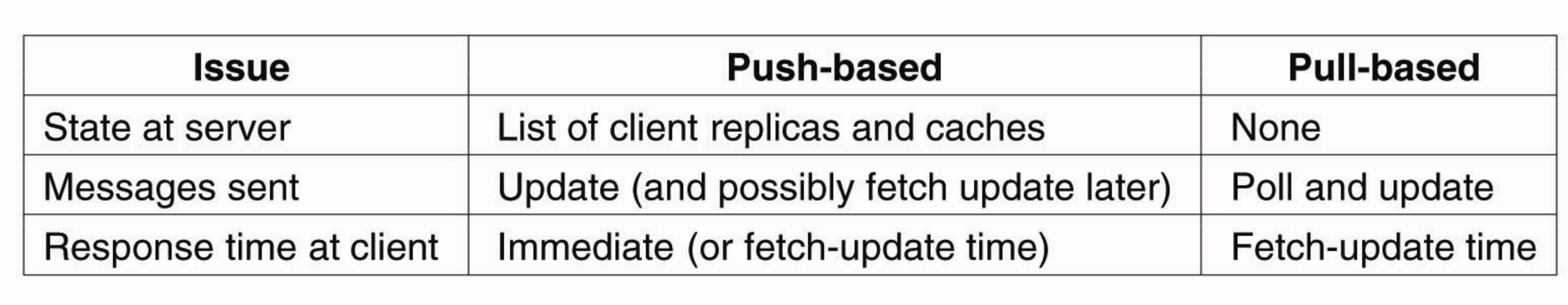
- Mainly deals with client cache
  - ✓ i.e., a local storage facility that is used by a client to temporarily store a copy of the data it has just requested
- The cached data may be outdated
  - ✓ Let the client checks the version of the data
- Multiple clients may use the same cache
  - ✓ Data requested by one client may be useful to other clients as well, e.g., DNS look-up
  - ✓ This can also improve the chance of cache hit

Content Distribution

- Deals with the propagation of updates to all relevant replicas
- Two key questions
  - ✓ What to propagate (state v.s. operations)
    - Propagate only a notification of an update
    - Transfer data from one copy to another
    - Propagate the update operation to other copies
  - ✓ How to propagate the updates
    - Pull v.s. push protocols
    - Unicast v.s. multicast

Pull v.s. Push Protocols

- Push-based approach
  - ✓ It is server-based, updates are propagated to other replicas without those replicas even asking for
  - ✓ It is usually used for high degree of consistency
- Pull-based approach
  - ✓ It is client-based, updates are propagated when a client or a replication server asks for it



Issue	Push-based	Pull-based
State at 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

Consistency Protocols

- A consistency protocol describes
  - ✓ An implementation of a specific consistency model
- Will discuss
  - ✓ Continuous consistency protocols
    - Bounding numerical, staleness, ordering deviation
  - ✓ Primary-based protocols
    - Remote-write and local-write protocols
  - ✓ Replication-write protocols
    - Active replication and quorum-based protocols

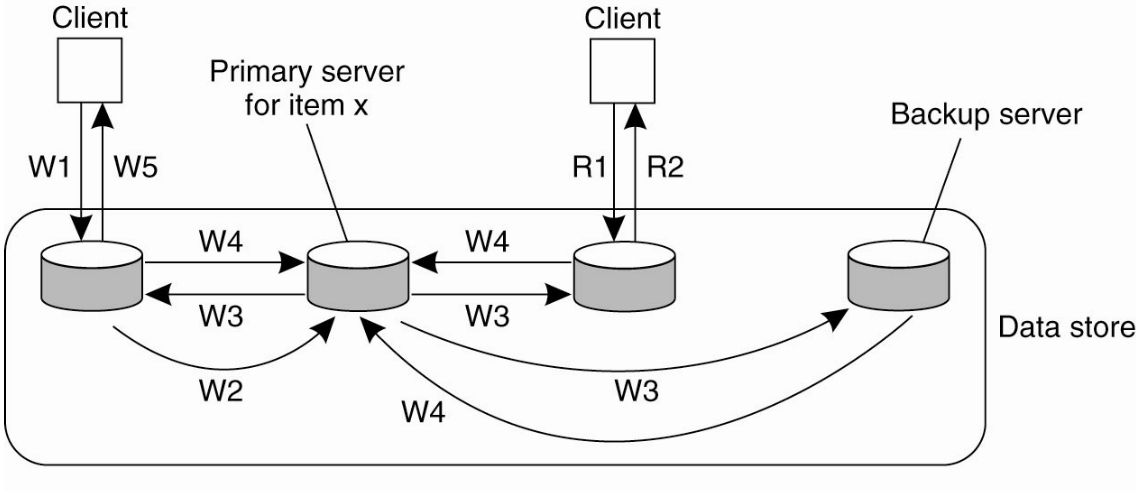
Continuous Consistency Protocols (1/2)

- Bounding numerical deviation
    - ✓ The number of unseen updates, the absolute numerical value, or the relative numerical value
    - ✓ E.g., the value of a local copy of x will never deviate from the real value of x by a threshold
- Let us concern about the number of updates unseen
    - ✓ i.e., the total number of unseen updates to a server shall never exceed a threshold
- A simple approach for N replicas
- Every server i tracks every other server j's state about i's local writes, i.e., the number of i's local writes not been seen by j
- If this number exceeds $\delta/(N-1)$, i will propagate its writes to j

Continuous Consistency Protocols (2/2)

- Bounding staleness deviation
  - ✓ Each server maintains a clock T(i), meaning that this server has seen all writes of i up to T(i)
  - ✓ Let T be the local time. If server i notices that T-T(j) exceeds a threshold, it will pull the writes from server j
- Bounding ordering deviation
  - ✓ Each server keeps a queue of tentative, uncommitted writes
  - ✓ If the length of this queue exceeds a threshold,
    - The server will stop accepting new writes and
    - Negotiate with other servers in which order its writes should be executed, i.e., enforce a globally consistent order of tentative writes
  - ✓ Primary-based protocols can be used to enforce a globally consistent order of tentative writes

Remote-Write Protocols

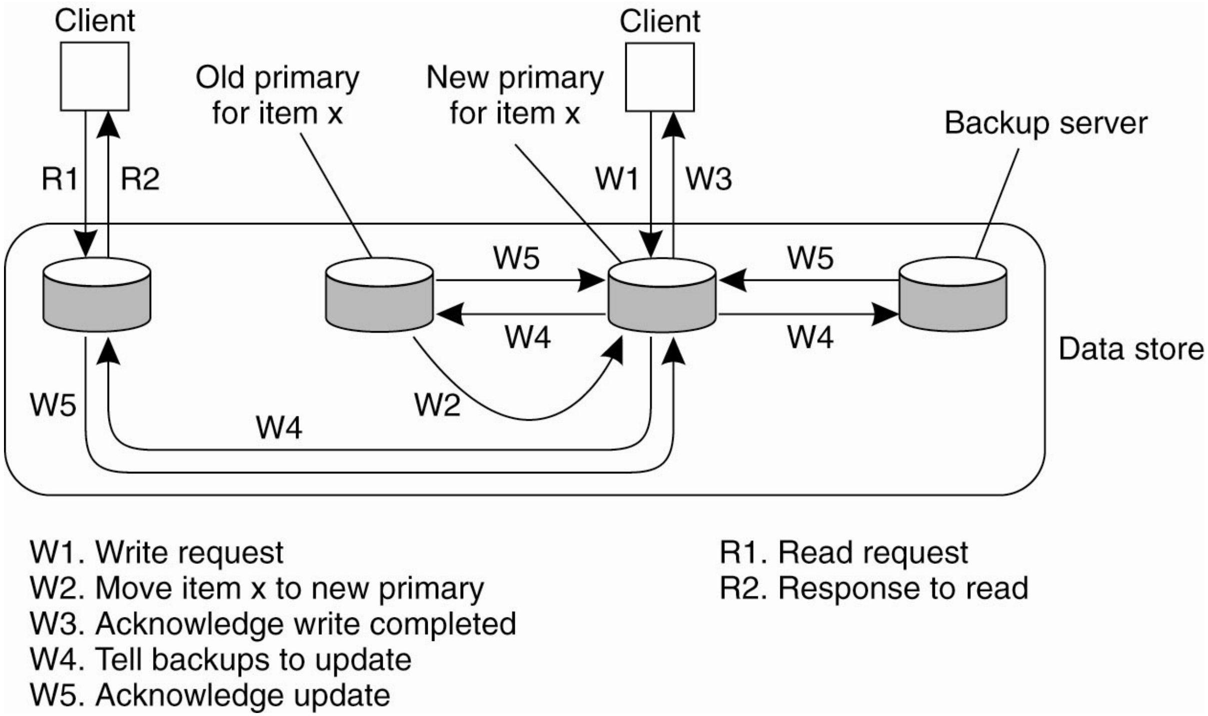


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

- Problem: it is a blocking operation at the client
- Replace it with a non-blocking update, i.e., update the local copy immediately and then the local server asks the backup server to perform the update
- However, the non-blocking version does not have fault tolerance

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

- The difference is that the primary copy migrates between processes
- Benefit: multiple successive writes can be performed locally, while others can still read
  - ✓ If a non-blocking protocol is followed by which updates are propagated to the replicas after the primary has finished the update

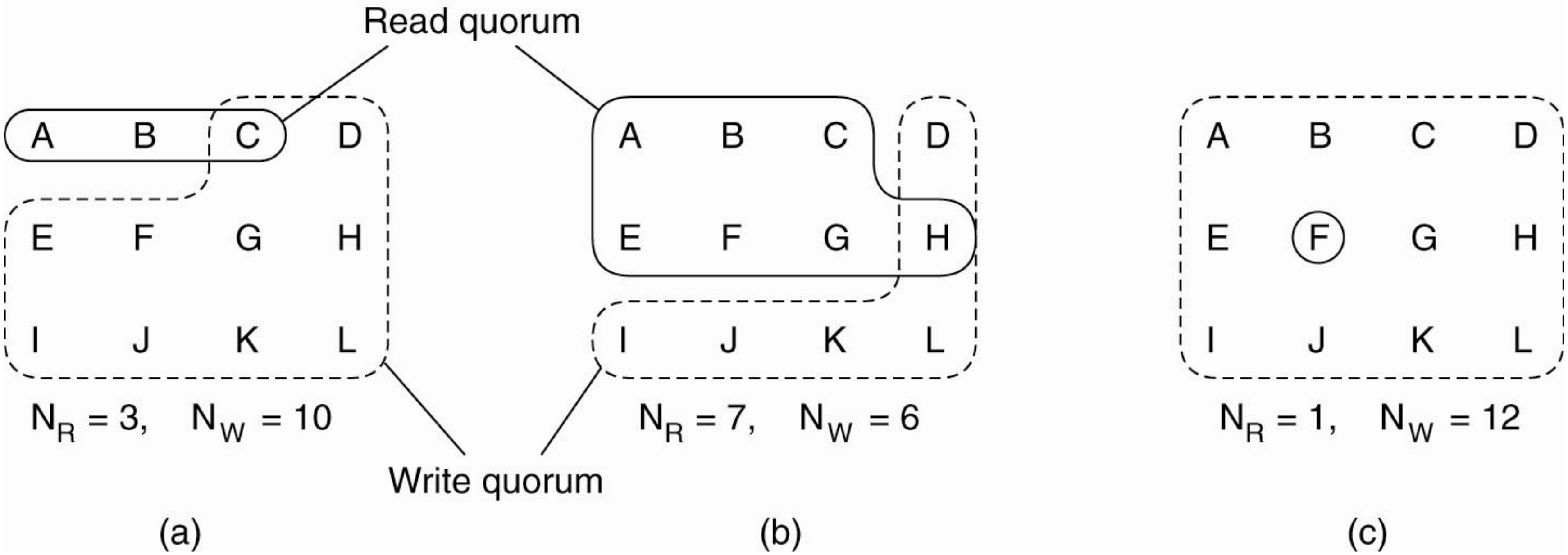
Replicated-Write Protocols (1/2)

- Active replication
  - ✓ Update are propagated by means of the write operation that causes the update
- The challenge is that the operations have to be carried out in the same order everywhere
  - ✓ Need a totally-ordered multicast mechanism such as the one based on Lamport's logical clocks
    - However, this algorithm is expensive and does not scale
- An alternative is to use a central sequencer
  - ✓ However, this central sequencer does not solve the scalability problem

Replicated-Write Protocols (2/2)

- Quorum-based protocols
  - ✓ Require a client to get permission from multiple servers before a read or write
- A simple version
  - ✓ A read or write has to get permission from half plus 1 servers
- A better version: a client must get permission from
  - ✓ A read quorum: an arbitrary set of Nr servers
  - ✓ A write quorum: an arbitrary set of Nw servers
  - ✓ Such that Nr+Nw>N and Nw>N/2

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).