Sequential Consistency

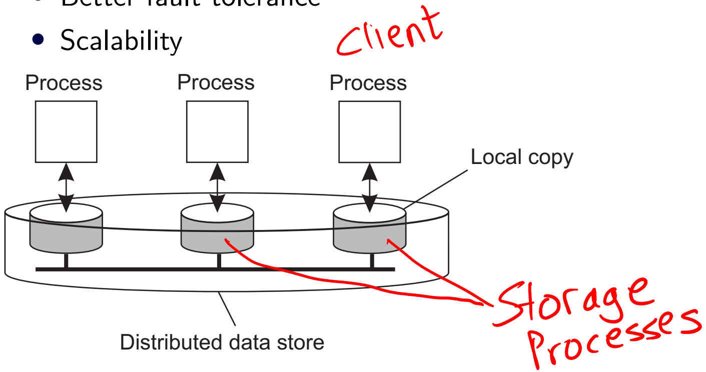
Distributed Systems Spring 2020

Lecture 13

Today's lecture:
1. Consistency models
2. Sequential consistency
3. Implement sequential consistency

Replication

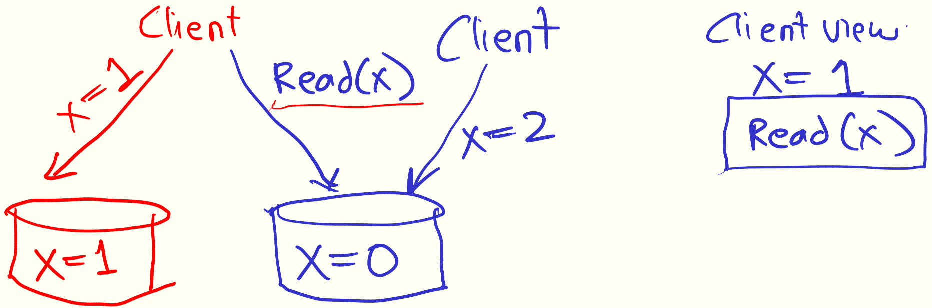
- Increase performance
- Increase system availability
- Better fault tolerance
- Scalability



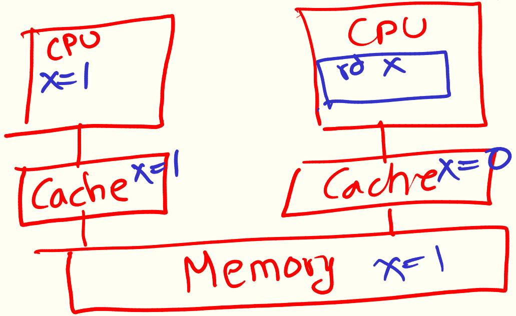
Consistency Models

A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.

- For many applications, we want that different clients making read/write requests to different replicas with the same logical data item should not obtain different results.
- Different consistency models dictate under what conditions different results can be obtained.
- Influence how concurrent reads and writes behave.
- Relevant in many contexts: shared multi-processor systems, cache coherence, databases, etc.



Read(x) outputs? [0, 1, ?]



Replication and Consistency

- Data stores can implement a range of consistency models with different tradeoffs
- Most intuitive model: **Program Order**
- Read(x) returns value of most recent write to x
- Also called **Strict Consistency**
- Strong assumption that we often make for sequential code.
- Replicas make it challenging:
  - What is "most recent" with many clients and many replicas?

wall clock time

Simple code example

x=1

x=2

rd(x) → 2

Consistency Model Tradeoffs

- Strict Consistency is ideal from programmer/user perspective
- Challenging/impossible to realize in many distributed system scenarios
- Usability vs. Performance vs. Fault-tolerance tradeoff.
- Many *relaxed* consistency models exist that dont always return the value of most recent write.
- This lecture: understanding and implementing **Sequential Consistency**

Sequential Consistency

Def

The result of any execution is the same as if the operations of 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.

- Concurrent operations can be "reordered" by the data store
- Operations can be *interleaved* > NOT Strictly cons

(a) Sequentially consistent. (b) Not sequentially consistent

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

(a)

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

(b)

W(x)a

R(x)a W(x)b R(x)b R(x)b

R(x)a

Execution History

Two processes P and Q share a queue.

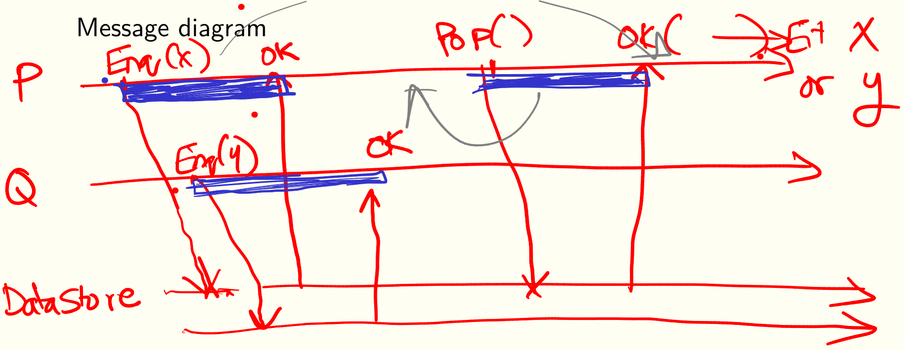
Observed execution history:

P:Enqueue(x), Q:Enqueue(y), P:ok(), Q:ok(), P:Dequeue(), P:ok(result=??)

- Each operation can be thought of as sending a message to the data store
- The ok() message is the received response.

Concurrent Operations

- P doesnt communicate with Q explicitly
- P and Q's enqueue operations are thus concurrent



P: Enq(x) $<_H$ P: Dequeue() // only constraint

Both x & y are valid outputs of Dequeue

Checking for Sequential Consistency

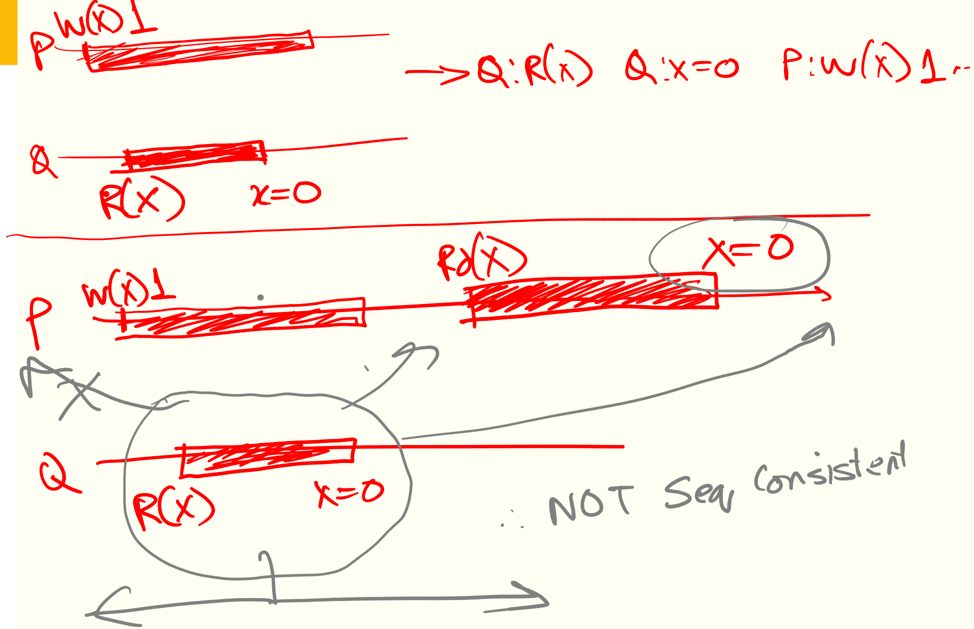
- With Sequential Consistency, data store can "move/slide" concurrent operations around
- "SC Legality Test": Given an execution history, could it have resulted from reordering concurrent operations such that the order of operations within a process is maintained.

Examples: *(assume vars initialized to 0)*

1. P:write(x,1)< Q:read(x)< Q:ok(0)< P:ok()
2. P:write(x,1), Q:read(x), Q:ok(0), P:read(x), P:ok(0)

↑ ROK()

Clearly not what you'd expect with only one client process.



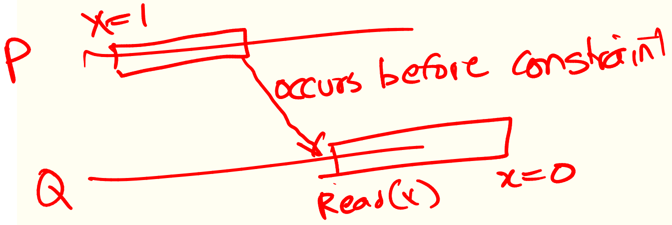
Linearizability

P1: W(x)=1
P2 W(x)=2
⋮

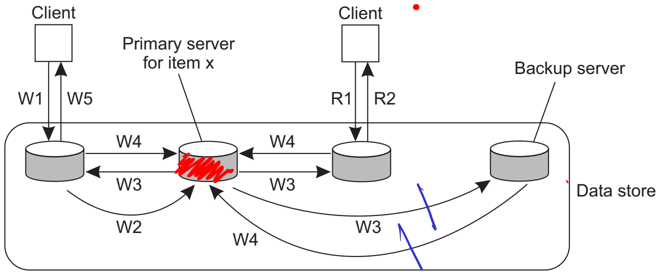
- Sequential consistency permits many valid outputs
- Stronger model: Linearizability
- An execution history is Linearizable if it is sequentially consistent and the original order of operations is preserved.
- Intuitively: cannot "slide" operations around any more
- Operations can be thought of as taking effect instantaenously.
- Data store has limited flexibility in reordering concurrent operations.
- Much more intuitive from user's perspective.

P:W(x,1), P:ok() < Q:R(x), Q:ok(0)
This is SC but not linearizable.

If 10 procs, then 10! valid interleavings



Primary-based protocol for Sequential Consistency

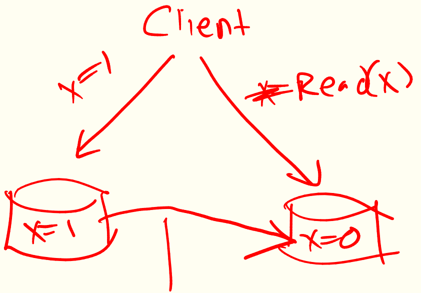


W3: Broadcast

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

- Each object has a single primary (can change over time)
- **Remote-writes:** All writes are **blocking** and forwarded to primary for serialization
- Need to be careful about faults with non-blocking protocols



Broadcast write to other replicas

Reads: Connect to any replica
(Local Reads..)

Writes: Broadcast .. (Remote writes)

Local Read Protocol

[No need for a master replica]

Each replica process (P) runs the following algorithm:

1 - Upon read(x): Generate Ok(v). v is value of P's copy of x; — Local Read
2 - Write(x, v): Totally ordered broadcast(x, v);
- Receiving a broadcast message(x, v) from Q:
  1. Set local copy of x to v ;
  2. If P==Q, then generate Ok() for write(x,v)

Generating a Sequentially consistent history:

- All writes are totally ordered
- Reads are inserted between appropriate writes based on value read

Writes are slow because they only return after the broadcast is completed. Reads concurrent with a write can get different values based on which replica they hit

Recall Totally ordered broadcast/multicast:

- Two rounds of messages.
- Guarantees that all processes "see" the same total order of operations.

① Total order of writes within a client

Blocking & Total property

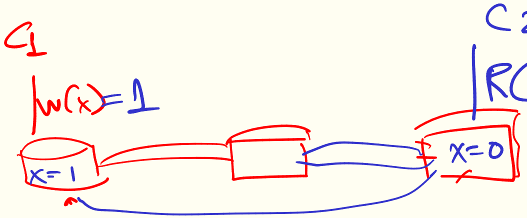
② R(x) R(y)

③ $W_1(x)$ $R_2(x)$ $W_3(x)$

Implementing Linearizability

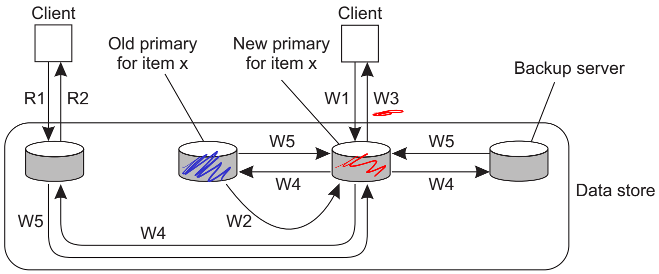
- Modify the local read algorithm
- All operations (including reads) require a total order broadcast
- A total order of all read and write operations that all processes agree on, is a linearizable history.

x=0 init



R(x)=0 even with Local read if Broadcast hasnt reached the replica

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

- Primary copy migrates between processes that want to write
- Example:Mobile computing in disconnected mode (ship all relevant files to user before disconnecting, and update later).
- Lowers write latency

Replicated-write protocols

- Writes can be performed by multiple replicas
- Active replication typically used (operations sent to replicas via total-order multicasting)
- “Centralization”: Use a sequencer for multicasting.
- All updates sent to a centralized sequencer that serializes the updates and broadcasts them

Local Write Algorithm

Key idea: Generate ok() for write Immediately.

Maintain counter *num* for pending writes

Upon read(x):

- If(num == 0), then generate Ok(v)

v is value of local copy of x

Upon write(x,v):

- num = num+1
- totally ordered broadcast(x, v)
- Generate Ok() for write

Upon receive of broadcast (x,v) from Q:

- set local copy of x to v
- If (P==Q), then
  - num = num -1
  - If(num==0), then generate Ok(v)

Reads now have to wait until all pending writes clear

Proof-sketch for Local write algorithm

1. Let $w_j(x, a) < r_i(x, a)$. We need to show that another write $w_k(x, b)$ cannot get between w_j, r_i.
2. 1st case: w_k, r_i are on same process P. But read only returns when all broadcasting operations (including for w_k) have finished, in which case the read would return value as b, and not a, which is a contradiction.
3. w_k occurs on Q, and r_i on P. Two cases again:
   - 3.1 w_k's broadcast phase is ongoing when read is issued. This cannot happen since all broadcasts must finish before reads return.
   - 3.2 w_k's broadcast finishes, and P knows that x is b. Which contradicts r_i