CSE 486/586 Distributed Systems
Consistency --- 1

Steve Ko
Computer Sciences and Engineering
University at Buffalo

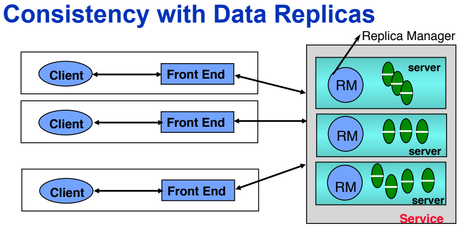
Recap: Concurrency (Transactions)

- Question: How to support transactions (with locks)?
  - Multiple transactions share data.
- First strategy: Complete serialization
  - One transaction at a time with one big lock
  - Correct, but at the cost of performance
- How to improve performance?
  - Let's see if we can interleave multiple transactions.

Recap: Concurrency (Transactions)

- Problem: Not all interleavings produce a correct outcome
  - Serial equivalence & strict execution must be met.
- How do we meet the requirements using locks?
  - Overall strategy: using more and more fine-grained locking
  - No silver bullet. Fine-grained locks have their own implications.
  - Exclusive locks (per-object locks)
  - Non-Exclusive locks (read/write locks)
  - Other finer-grained locks (e.g., two-version locking)
- Atomic commit problem
  - Commit or abort (consensus)
  - 2PC

Consistency with Data Replicas



- Consider that this is a distributed storage system that serves read/write requests.
- Multiple copies of a same object stored at different servers
- Question: How to maintain consistency across different data replicas?

Consistency

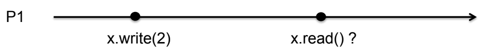
- Why replicate?
- Increased availability of service. When servers fail or when the network is partitioned.
  - P: probability that one server fails= 1 – P= availability of service. e.g. P = 5% => service is available 95% of the time.
  - P^n: probability that n servers fail= 1 – P^n= availability of service. e.g. P = 5%, n = 3 => service available 99.875% of the time
- Fault tolerance
  - Under the fail-stop model, if up to f of f+1 servers crash, at least one is alive.
- Load balancing
  - One approach: Multiple server IPs can be assigned to the same name in DNS, which returns answers round-robin.

This Week

- We will look at different consistency guarantees (models).
- We'll start from the strongest guarantee, and gradually relax the guarantees.
  - Linearizability (or sometimes called strong consistency)
  - Sequential consistency
  - Causal consistency
  - Eventual consistency
- Different applications need different consistency guarantees.
- This is all about client-side perception.
  - When a read occurs, what do you return?
- First
  - Linearizability: we'll look at the concept first, then how to implement it later.

Our Expectation with Data

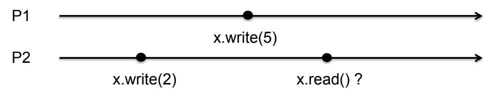
- Consider a single process using a filesystem
- What do you expect to read?



- Our expectation (as a user or a developer)
  - A read operation returns the most recent write.
  - This forms our basic expectation from any file or storage system.
- Linearizability meets this basic expectation.
  - But it extends the expectation to handle multiple processes…
  - …and multiple replicas.
  - The strongest consistency model

Expectation with Multiple Processes

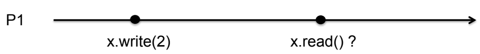
- What do you expect to read?
  – A single filesystem with multiple processes



- Our expectation (as a user or a developer)
  - A read operation returns the most recent write, regardless of the clients.
  - We expect that a read operation returns the most recent write according to the single actual-time order.
  - In other words, read/write should behave as if there were a single (combined) client making all the requests.
  - It's easiest to understand and program for a developer if your storage appears to process one request at a time.

Expectation with Multiple Copies

- What do you expect to read?
  – A single process with multiple servers with copies



- Our expectation (as a user or a developer)
  - A read operation returns the most recent write, regardless of how many copies there are.
  - Read/write should behave as if there were a single copy.

Linearizability

- Three aspects
  – A read operation returns the most recent write,
  – …regardless of the clients,
  – …according to the single actual-time ordering of requests.
- Or, put it differently, read/write should behave as if there were,
  – …a single client making all the (combined) requests in their original actual-time order (i.e., with a single stream of ops),
  – …over a single copy.
- You can say that your storage system guarantees linearizability when it provides single-client, single-copy semantics where a read returns the most recent write.
  – It should *appear* to all clients that there is *a single order (actual-time order) that your storage uses* to process all requests.

Linearizability Exercise

- Assume that the following happened with object x over a linearizable storage.
  – C1: x.write(A)
  – C2: x.write(B)
  – C3: x.read() → B, x.read() → A
  – C4: x.read() → B, x.read() → A
- What would be an actual-time ordering of the events?
  – One possibility: C2 (write B) -> C3 (read B) -> C4 (read B) -> C1 (write A) -> C3 (read A) -> C4 (read A)
- How about the following?
  – C1: x.write(A)
  – C2: x.write(B)
  – C3: x.read() → B, x.read() → A
  – C4: x.read() → A, x.read() → B

CSE 486/586 Administrivia

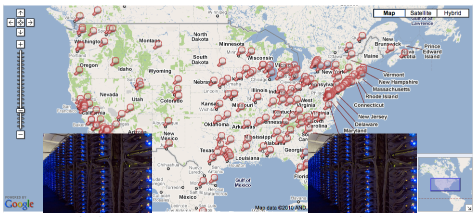
- PA3 deadline: 4/8 (Friday)
- This Friday and next Monday
  – No lectures
  – PA3 help from the TAs (still in the lecture room)

Linearizability Subtleties

- Notice any problem with the representation?

You (NY) ——— x.write(5) ———>

Friend (CA) ——— x.write(2) ——— read(x) ? ———>



California — North Carolina

Linearizability Subtleties

- A read/write operation is never a dot!
  – It takes time. Many things are involved, e.g., network, multiple disks, etc.
  – Read/write latency: the time measured right before the call and right after the call from the client making the call.
- Clear-cut (e.g., black---write & red---read)
- Not-so-clear-cut (parallel)
  – Case 1:
  – Case 2:
  – Case 3:

Linearizability Subtleties

- With a single process and a single copy, can overlaps happen?
  – No, these are cases that do not arise with a single process and a single copy.
  – "Most recent write" becomes unclear when there are overlapping operations.
- Thus, we (as a system designer) have freedom to impose an order.
  – As long as it appears to all clients that there is a single, interleaved ordering for all (overlapping and non-overlapping) operations that your implementation uses to process all requests, it's fine.
  – I.e., this ordering should still provide the single-client, single-copy semantics.

Linearizability Subtleties

- Definite guarantee
- Relaxed guarantee when overlap
- Case 1
- Case 2
- Case 3

Linearizability Examples

- Example 1: if your system behaves this way…

a.write(x)

a.read() -> x

a.read() -> x

- Example 2: if your system behaves this way…

a.write(x)

a.read() -> 0 — a.read() -> x ← If this were a.read() -> 0, would it support linearizability?

a.read() -> x

No

Linearizability Examples

- In example 2, what are the constraints?

a.write(x)

a.read() -> 0 — a.read() -> x

a.read() -> x

- Constraints
  – a.read() → 0 happens before a.read() →x (you need to be able to explain why that happens that way).
  – a.read() → x happens before a.read() →x (you need to be able to explain why that happens that way).
  – The rest are up for grabs.
- Scenario
  – a.write(x) gets propagated to (last client's) a.read() -> x first.
  – a.write(x) gets propagated to (the second process's) a.read() -> x, right after a.read() -> 0 is done.

Linearizability Examples

- In example 2, why would a.read() return 0 and x when they're overlapping?

a.write(x)
a.read() -> 0   a.read() -> x
a.read() -> x

- This assumes that there's a particular storage system that shows this behavior.
- At some point between a read/write request sent and returned, the result becomes visible.
  - E.g., you read a value from physical storage, *prepare it for return (e.g., putting it in a return packet, i.e., making it visible)*, and actually return it.
  - Or you *actually write a value to a physical disk, making it visible* (out of multiple disks, which might actually write at different points).

Linearizability Examples

- Example 3

a.write(x)
a.read() -> x   a.read() -> x
a.read() -> y
a.write(y)

- Constraints
  - a.read() → x and a.read() → x: we cannot change these.
  - a.read() → y and a.read() → x: we cannot change these.
  - The rest is up for grabs.

Linearizability (Textbook Definition)

- Let the sequence of read and update operations that client i performs in some execution be oi1, oi2,….
  - "Program order" for the client
- A replicated shared object service is linearizable if for any execution (real), there is some interleaving of operations (virtual) issued by all clients that:
  - meets the specification of a single correct copy of objects
  - is consistent with the actual times at which each operation occurred during the execution
- Main goal: any client will see (at any point of time) a copy of the object that is correct and consistent
- The strongest form of consistency

Summary

- Linearizability
  - Single-client, Single-copy semantics
- A read operation returns *the most recent* write, regardless of the clients, according to their actual-time ordering.

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).