Recap: Linearizability

- Linearizability
  - Should provide the behavior of a single client and a single copy
  - A read operation returns the most recent write, regardless of the clients according to their original actual-time order.
- Complication
  - In the presence of concurrency, read/write operations overlap.
  - There, you should be able to show that you’re using some ordering of requests, where you return the most recent write (every time there’s a read).

Linearizability

- Linearizability is all about client-side perception.
  - The same goes for all consistency models for that matter.
- If you write a program that works with a linearizable storage, it works as you expect it to work.
- There’s no surprise.

Linearizability Examples

- Example 1
  - a.write(x)
  - a.read() \rightarrow x
  - a.read() \rightarrow x

- Example 2
  - a.write(x)
  - a.read() \rightarrow 0
  - a.read() \rightarrow x

- Example 3
  - a.write(x)
  - a.read() \rightarrow x
  - a.read() \rightarrow x
  - a.read() \rightarrow y
  - a.write(y)

Implementing Linearizability
Implementing Linearizability

• Will this be difficult to implement?
  – It depends on what you want to provide.
  
You (NY) x.write(5)
Friend (CA) x.write(2) read(x) \rightarrow 5

• How about:
  – All clients send all read/write to CA datacenter.
  – CA datacenter propagates to NC datacenter.
  – A request never returns until all propagation is done.
  – Correctness (linearizability)? yes
  – Performance? No

Implementing Linearizability

• Importance of latency
  – Amazon: every 100ms of latency costs them 1% in sales.
  – Google: an extra .5 seconds in search page generation time dropped traffic by 20%.

• Linearizability typically requires complete synchronization of multiple copies before a write operation returns.
  – So that any read over any copy can return the most recent write.
  – No room for asynchronous writes (i.e., a write operation returns before all updates are propagated.)

• It makes less sense in a global setting.
  – Inter-datacenter latency: ~10s ms to ~100s ms
  – It might still makes sense in a local setting (e.g., within a single data center).

Passive (Primary-Backup) Replication

• Request Communication: the request is issued to the primary RM and carries a unique request id.
• Coordination: Primary takes requests atomically, in order, checks id (resends response if not new id.)
• Execution: Primary executes & stores the response
• Agreement: If update, primary sends updated state/result, req-id and response to all backup RMs (1-phase commit enough).
• Response: primary sends result to the front end

Chain Replication

• One technique to provide linearizability with better performance
  – All writes go to the head.
  – All reads go to the tail.

• Linearizability?
  – Clear-cut cases: straightforward
  – Overlapping ops?

CSE 486/586 Administrivia

• What ordering does this have for overlapping ops?
  – We have freedom to impose an order.
  – Case 1: A write is at either N0 or N1, and a read is at N2. The ordering we’re imposing is read then write.
  – Case 2: A write is at N2 and a read is also at N2. The ordering we’re imposing is write then read.

• Linearizability
  – Once a write becomes visible (at the tail), all following reads get the write result.
Relaxing the Guarantees

- Do we need linearizability?
- Does it matter if I see some posts some time later?
- Does everyone need to see these in this particular order?

Linearizability advantages
- It behaves as expected.
- There’s really no surprise.
- Application developers do not need any additional logic.

Linearizability disadvantages
- It’s difficult to provide high-performance (low latency).
- It might be more than what is necessary.

Relaxed consistency guarantees
- Sequential consistency
- Causal consistency
- Eventual consistency

It is still all about client-side perception.
- When a read occurs, what do you return?

Sequential Consistency

- A little weaker than linearizability, but still quite strong
  - Essentially linearizability, except that it allows writes from
    other processes to show up later.
- It still captures some reasonable expectation, but not
  the most natural one (which is captured by
  linearizability).
- For the remaining discussion,
  - Let’s assume that there are multiple processes.
  - Let’s also assume that each write has a unique value (just
    for the same of illustration).

Scenario 1: does this meet our natural expectation?
- We’ll think that there must be a write after the last write.
- Would we care which of these were true?
- Or.

Scenario 2: does this meet our natural expectation?
- No. Why? Not the most recent write.

Sequential consistency at least preserves this
expectation (each process’s program order).

In both cases, the logical ordering is this:
- Sequential consistency: Your storage should appear
  to process all requests in a single interleaved
  ordering, where...
  - ...each and every process’s program order is preserved,
  - ...and each process’s program order is only logically
    preserved w.r.t. other processes’ program orders, i.e., it
    doesn’t need to preserve its physical-time ordering.
- It works as if all clients are reading out of a single
  copy.
  - This meets the expectation from a (isolated) client, working
    with a single copy.
### Sequential Consistency Examples

**Example 1:** Can a sequentially consistent storage show this behavior? (i.e., can you come up with an interleaving that behaves like a single copy?)

- P1: `a.write(A)`
- P2: `a.write(B)`
- P3: `a.read() -> B, a.read() -> A`
- P4: `a.read() -> B, a.read() -> A`

**Example 2**

- P1: `a.write(A)`
- P2: `a.write(B)`
- P3: `a.read() -> B, a.read() -> A`
- P4: `a.read() -> A, a.read() -> B`

### Implementing Sequential Consistency

**In what implementation would the following happen?**

- P1: `a.write(A)`
- P2: `a.write(B)`
- P3: `a.read() -> B, a.read() -> A`
- P4: `a.read() -> A, a.read() -> B`

**Possibility**

- P3 and P4 use different copies.
- In P3's copy, P2's write arrives first and gets applied.
- In P4's copy, P1's write arrives first and gets applied.
- Writes are applied in different orders across copies.
- This doesn't provide sequential consistency.

### Implementing Sequential Consistency

**Typical implementation**

- You're not obligated to make the most recent write (according to actual time) visible (i.e., applied to all copies) right away.
- But you are obligated to apply all writes in the same order for all copies. This order should be FIFO-total.

### Active Replication

- A front end FIFO-orders all reads and writes.
- A read can be done completely with any single replica.
- Writes are totally-ordered and asynchronous (after at least one write completes, it returns).
  - Total ordering doesn’t guarantee when to deliver events, i.e., writes can happen at different times at different replicas.
  - Sequential consistency, not linearizability
    - Read/Write ops from the same client will be ordered at the front end (program order preservation).
    - Writes are applied in the same order by total ordering (single copy).
    - No guarantee that a read will read the most recent write based on actual time.

### Two More Consistency Models

**Even more relaxed**

- We don’t even care about providing an illusion of a single copy.

**Causal consistency**

- We care about ordering causally related write operations correctly.

**Eventual consistency**

- As long as we can say all replicas converge to the same copy eventually, we’re fine.

### Summary

**Linearizability**

- The ordering of operations is determined by time.
- Primary-backup can provide linearizability.
- Chain replication can also provide linearizability.

**Sequential consistency**

- The ordering of operations preserves the program order of each client.
- Active replication can provide sequential consistency.
Acknowledgements

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