CSE 486/586: Distributed Systems

Consistency

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Introduction to Consistency

A distributed system may store data in multiple places:

- to improve performance
- to increase availability
- to provide fault tolerance

Multiple copies of the same object are called replicas.

For some applications, it may be important that these copies are maintain some form of consistency.

For example:

- they are the same
- they share history up to some point
- if they are different, one happens-before the other
- *etc.*

Advantages of Replication I

Replication can be used to improve performance.

For example:

- Placing replicas “near” clients on the network can improve network latency and bandwidth. (This is the foundation of content distribution networks like Akamai.)
- Distributing load between replicas (load balancing) can reduce individual server loads and improve response times.

Advantages of Replication II

It can also be used to increase availability.

Assume that the probability of a single server failing is P.

- With one server, expected availability is $1 - P$.
- With n servers, expected availability is $1 - P^n$.

Thus, with $P = 5\%$:

Servers	Availability (%)
1	95.0000
2	99.7500
3	99.9875
4	99.9994

Consistency Guarantees

We will look at consistency guarantees in decreasing strength:

- Linearizability (or strong consistency)
- Sequential consistency
- Causal consistency
- Eventual consistency

The consistency required varies with the application.

Expectations within a Single Process

Consider a single process and a single datum:

```
write(x ← 2);
y = read(x);
```

What do we expect y to contain?

Expectations within a Single Process

Consider a single process and a single datum:

```
write(x ← 2);
y = read(x);
```

What do we expect y to contain?

Our general expectation is that
a read operation returns the most recent write.

Expectations with Multiple Processes

Consider a single datum across two processes:

Process P1:

```
write(x ← 2);

y = read(x)
```

Process P2:

```
write(x ← 5);
```

What do we expect y to contain?

Expectations with Multiple Processes

Consider a single datum across two processes:

Process P1:

```
write(x ← 2);

y = read(x)
```

Process P2:

```
write(x ← 5);
```

What do we expect y to contain?

- A read operation returns the most recent write regardless of who wrote.
- That most recent write is in physical time order.
- In other words, as if all reads and writes are serialized.

Expectation with Multiple Replicas

Similarly, with multiple replicas, we expect:

- Reads will return the most recent write regardless of the number of replicas.
- Read from any replica should return the same value.
- Write to any replica should modify all replicas.

Read and write operations should behave as if there were one replica.

Linearizability

Linearizability meets the foregoing expectations with the following guarantees:

- A read operation returns the most recent write,
- regardless of the number of clients, and
- according to the physical time ordering of requests.

This is equivalent to saying that linearization behaves as if a single client executed all operations in their physical time order on a single copy of the data.

A storage system guarantees linearizability if it provides the above single-client, single-copy semantics, such that a read returns the most recent write in physical time order.

Linearizing Real Operations I

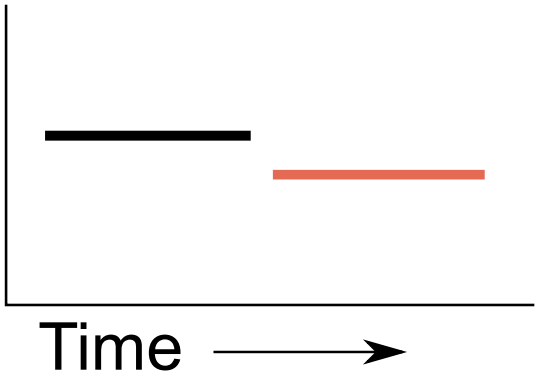
Unfortunately, real operations do not occur instantaneously.

For example, disk and network I/O have latency.

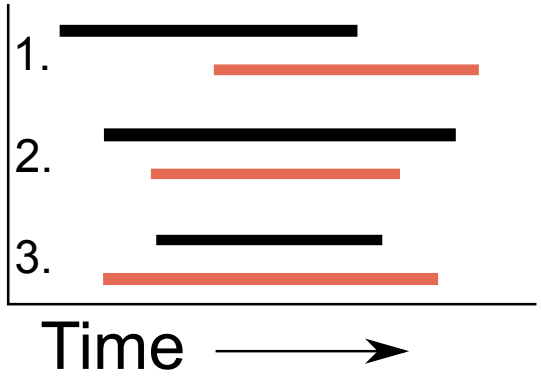
Read/write latency: The time from when a call for a read or write is invoked until the time the operation returns control to the client.

Linearizing Real Operations II

Clear ordering:



Unclear ordering:



(write, read)

With multiple processes and operations of measurable length, overlaps can occur.

In this case, what is the correct order?

- It's not clear!

Linearizing Real Operations III

Can overlaps happen with a single process and a single replica?

- No.
- Why not?

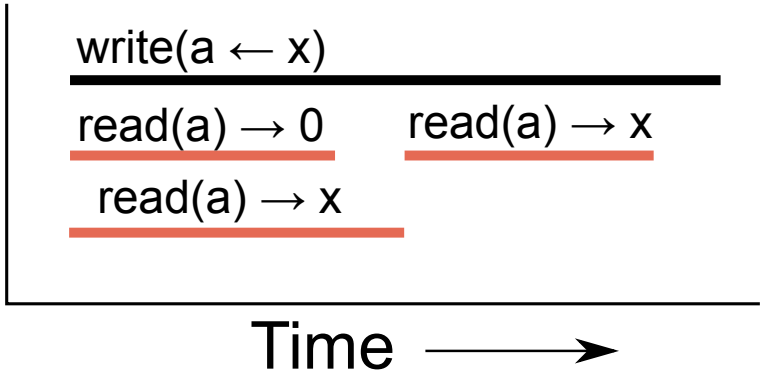
Linearizability picks something and defines it as correct:

- When overlaps occur, if it appears to all clients that there is a single, interleaved ordering for all operations, the ordering is valid.
- Once this ordering is defined, the correct value of each read is clear.

This means that there may be more than one valid linearization with overlap, but not within any one system.

Linearizability with Overlap

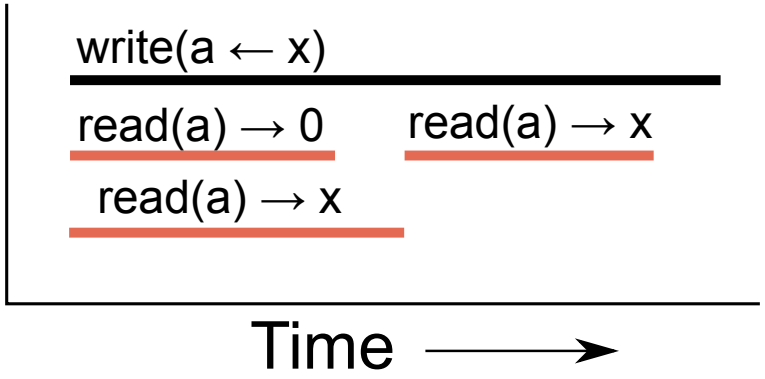
Consider three processes performing operations:



What are the constraints here?

Linearizability with Overlap

Consider three processes performing operations:

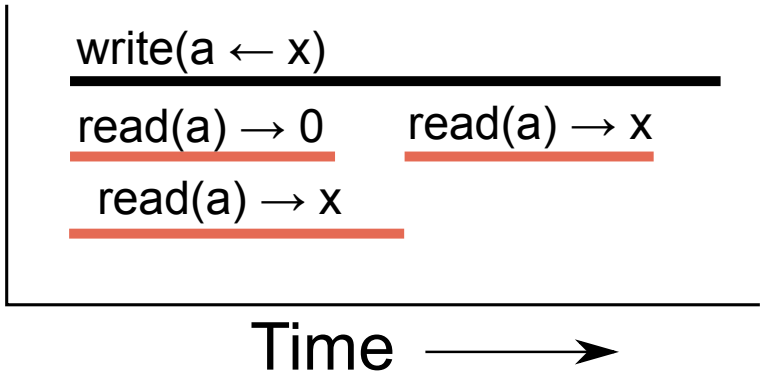


What are the constraints here?

- read(a) → 0 happens before either read(a) → x
- write(a ← x) happens before either read(a) → x
- The rest of the ordering can be implementation-defined

Linearizability with Overlap Analysis

Consider three processes performing operations:

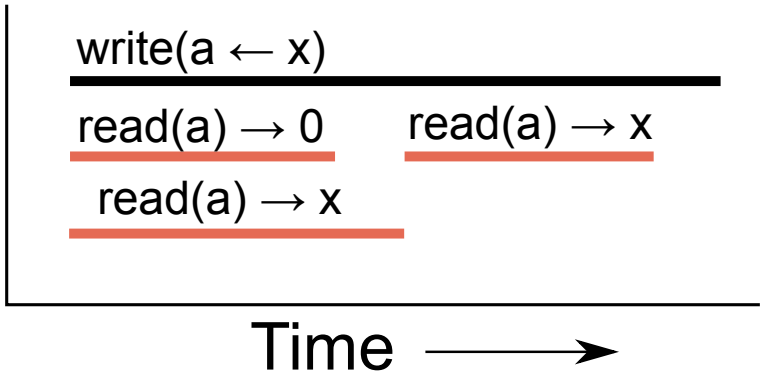


How can this happen?

- The write(a ← x) propagates to the bottom process first.
- The write(a ← x) propagates to the middle process only after its read(a) → 0 has determined a value for a.

Linearizability with Overlap Analysis

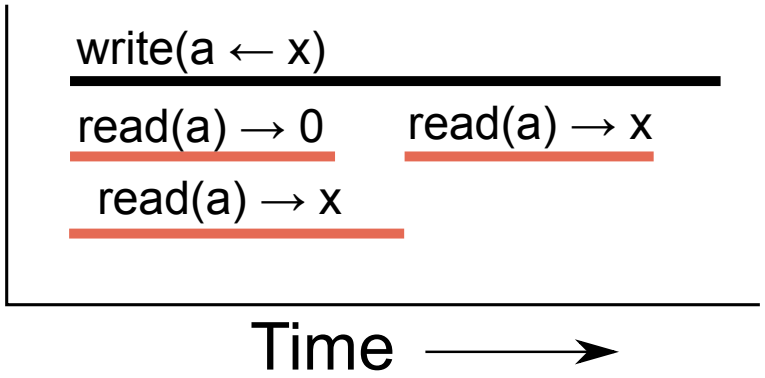
Consider three processes performing operations:



Why might the middle process read different values for a *during* the top process's write?

Linearizability with Overlap Analysis

Consider three processes performing operations:



Why might the middle process read different values for a *during* the top process's write?

- At some point, the write becomes visible to other processes. (*e.g.*, the value is actually stored to disk).
- The same is true for a read; at some point the read value is determined from the underlying storage.

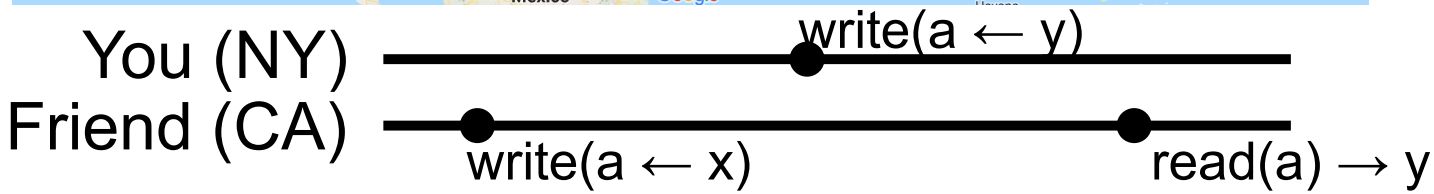
Linearizability (Textbook Definition)

- Let the sequence of read and update operations that client *i* performs in some execution be o_{i1}, o_{i2}, …
- A replicated shared object service is linearizable if, for any execution (in physical time), there is some interleaving of operations issued by all clients that:
  - meets the specification of a single correct copy of objects
  - is consistent with the physical times at which each operation occurred during execution

This is the strongest form of consistency.

Practical Linearizability

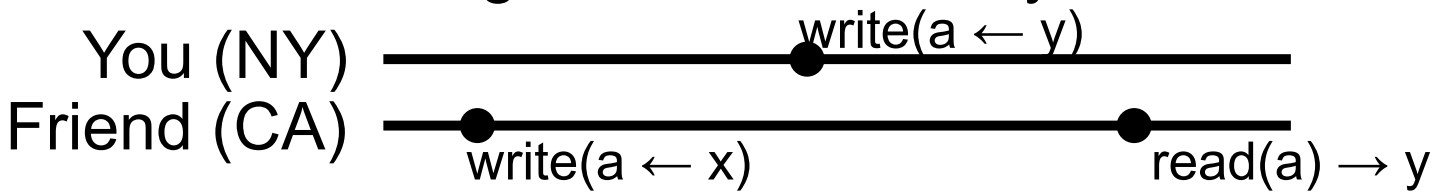
Consider the following scenario:



What are the challenges to linearizability?

Practical Linearizability

What are the challenges to linearizability?



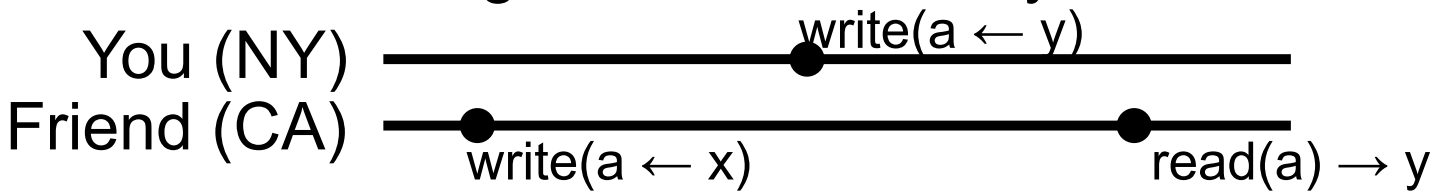
What if:

- All clients send all operations to the CA data center.
- The CA data center propagates writes to the NY data center.
- No request returns until all propagation is finished.

Is this correct (does it display linearizability)?

Practical Linearizability

What are the challenges to linearizability?



What if:

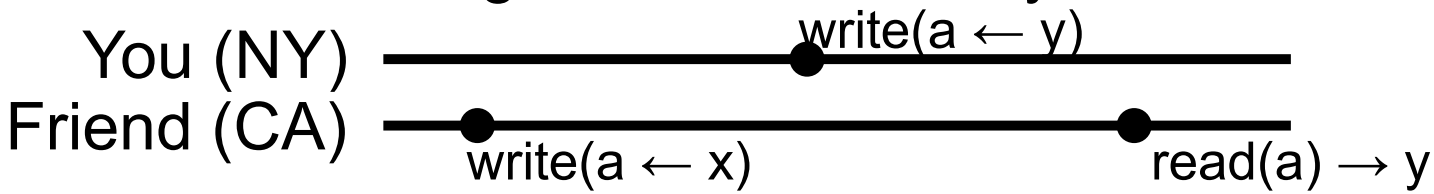
- All clients send all operations to the CA data center.
- The CA data center propagates writes to the NY data center.
- No request returns until all propagation is finished.

Is this correct (does it display linearizability)?

- Yes

Practical Linearizability

What are the challenges to linearizability?



What if:

- All clients send all operations to the CA data center.
- The CA data center propagates writes to the NY data center.
- No request returns until all propagation is finished.

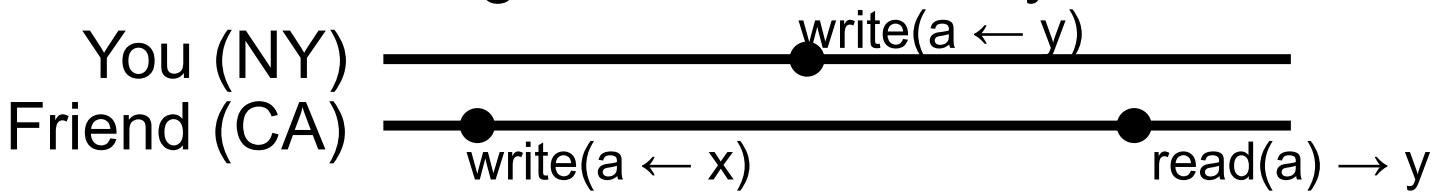
Is this correct (does it display linearizability)?

- Yes

Is this performant?

Practical Linearizability

What are the challenges to linearizability?



What if:

- All clients send all operations to the CA data center.
- The CA data center propagates writes to the NY data center.
- No request returns until all propagation is finished.

Is this correct (does it display linearizability)?

- Yes

Is this performant?

- No

Implementing Linearizability

Latency is very important!

- Amazon: 100 ms of added latency costs 1% in sales.
- Google: 500 extra ms in search page generation time dropped traffic by 20%.

Linearizability typically requires complete synchronization of replicas before a write operation returns.

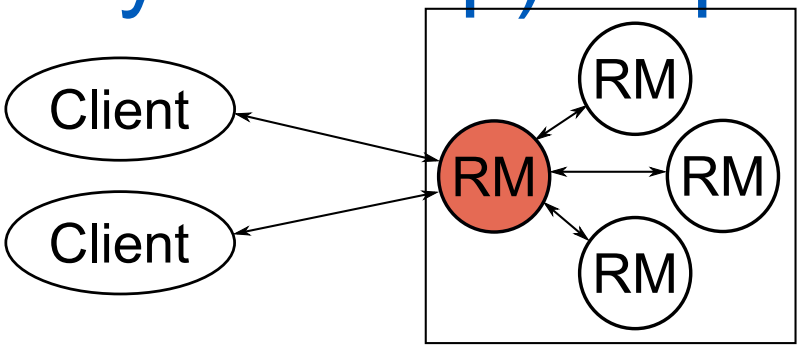
- Read on any replica returns the most recent write
- This means writes must be synchronous

This means this approach is too expensive in a global setting.

- Cross-country RTT is ~20 ms minimum.
- Cross-oceanic is even longer!

It might still make sense for local replicas.

Passive (Primary-Backup) Replication

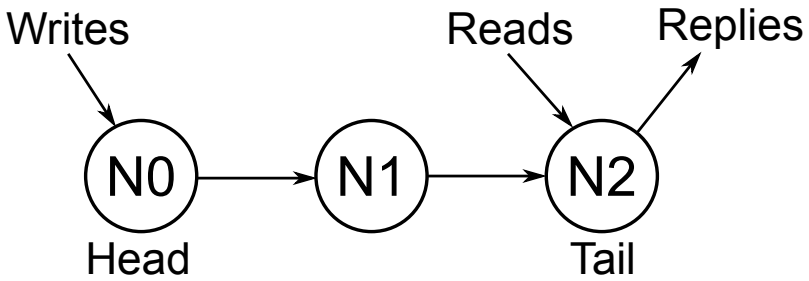


- Request: Requests are issued to the primary replica manager (RM), with unique ID.
- Coordination: The primary RM takes requests atomically, in order. Duplicate requests are detected by ID.
- Execution: The primary RM executes the request and stores its response.
- If the request is an update, it sends updates to all backup RMs (with 1-phase commit).
- Response: The primary RM sends its response to the client.

Chain Replication

Performance can be improved via chain replication [4].

- All writes go to the had of a chain of replicas.
- All reads go to the tail of the chain.

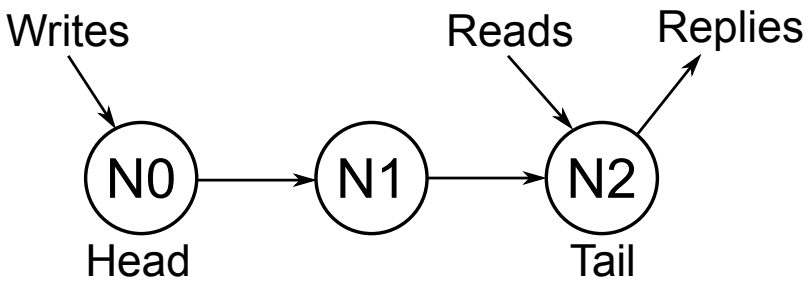


Is this linearizable?

Chain Replication

Performance can be improved via chain replication [4].

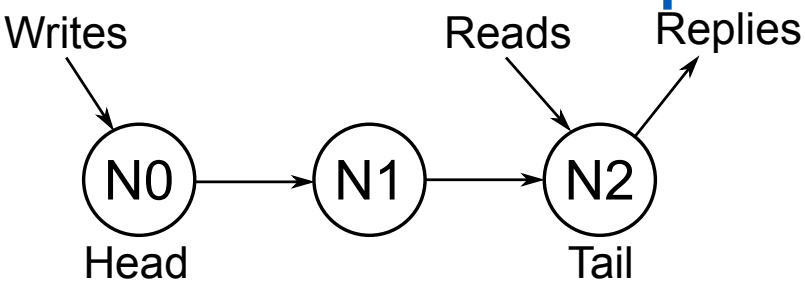
- All writes go to the had of a chain of replicas.
- All reads go to the tail of the chain.



Is this linearizable?

- It's straightforward for non-overlapping operations
- What about overlapping?

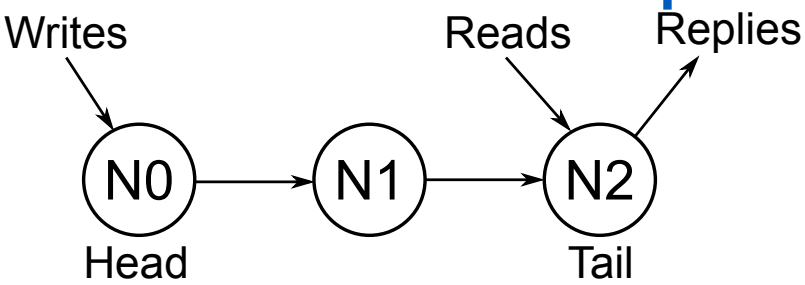
Chain replication with Overlap



How are operations linearized if they overlap?

- The absolute order of overlapping operations can be implementation dependent.
- Consider a write that has propagated to N0 or N1 when a read arrives at N2.

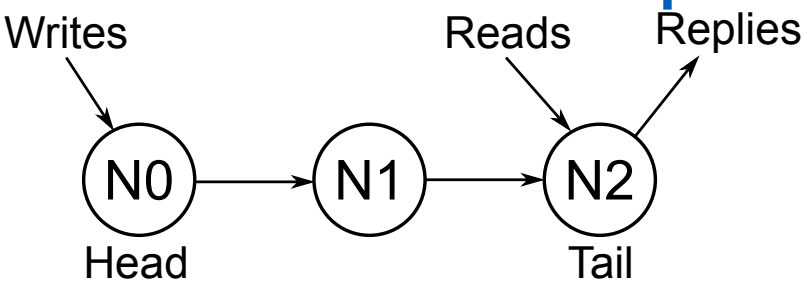
Chain replication with Overlap



How are operations linearized if they overlap?

- The absolute order of overlapping operations can be implementation dependent.
- Consider a write that has propagated to N0 or N1 when a read arrives at N2.
  - The imposed ordering is read-write.

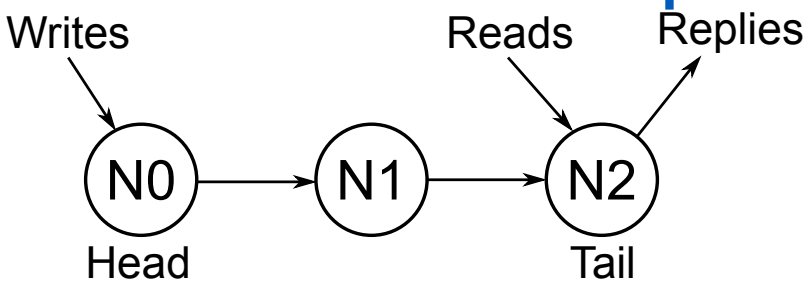
Chain replication with Overlap



How are operations linearized if they overlap?

- The absolute order of overlapping operations can be implementation dependent.
- Consider a write that has propagated to N0 or N1 when a read arrives at N2.
  - The imposed ordering is read-write.
- Consider a write that has propagated to N2 when a read arrives at N2.

Chain replication with Overlap

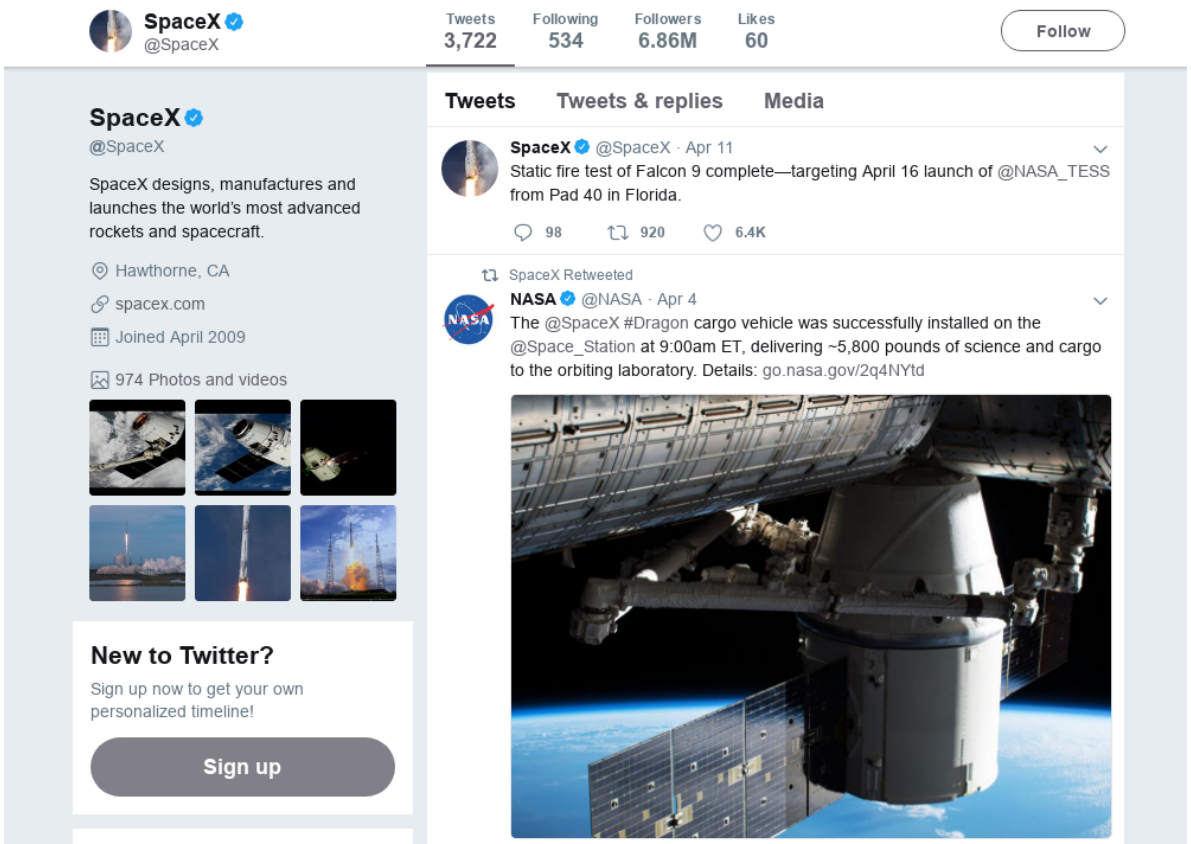


How are operations linearized if they overlap?

- The absolute order of overlapping operations can be implementation dependent.
- Consider a write that has propagated to N0 or N1 when a read arrives at N2.
  - The imposed ordering is read-write.
- Consider a write that has propagated to N2 when a read arrives at N2.
  - The imposed ordering is write-read.

Once a write becomes visible, all future reads will see it.

Relaxing Consistency



Do I care if posts are a little bit late?

What if someone sees them in a slightly different order?

Relaxing Linearizability's Guarantees

Linearizability has advantages:

- It behaves as programmers expect.
- Application developers don't need to include additional logic.

But it also has disadvantages:

- Low latency is difficult to achieve.
- It may provide stronger guarantees than necessary.

Sequential consistency relaxes this model just a bit.

It's still about client perception: when a read occurs, what does it return?

Sequential Consistency

Sequential consistency is a bit weaker than linearizability.

It is still quite strong.

- Essentially linearizability, but writes from other processes may show up later.

It still meets a reasonable expectation, but not the natural expectation captured by linearizability.

Assumptions:

- There are multiple processes.
- Each write is of a unique value (for clarity).

Sequential Consistency Examples

Example 1: Does this meet our expectations?

Sequential Consistency Examples

Example 1: Does this meet our expectations?

Example 2: What about this?

Sequential Consistency Examples

Example 1: Does this meet our expectations?

Example 2: What about this?

- Why not?
- Program order within P1 was not preserved.
- We expect this!

Sequential consistency preserves program order within a process.

Sequential Consistency Examples II

Example 3: What about this (with multiple processes)?

Sequential Consistency Examples II

Example 3: What about this (with multiple processes)?

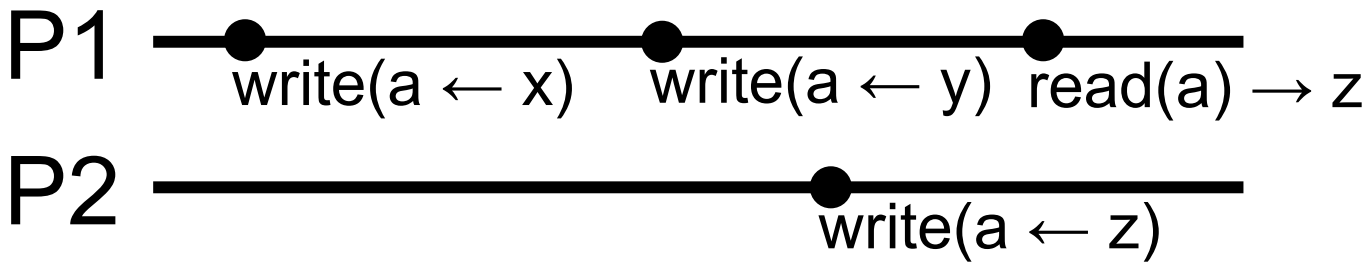
We'll just assume another process wrote.

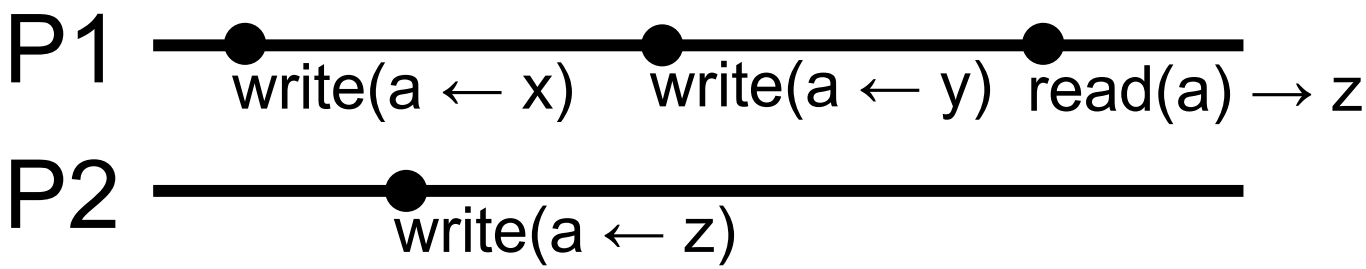
Sequential Consistency Examples II

Example 3: What about this (with multiple processes)?

We'll just assume another process wrote.

Does it matter which of these were true?





Sequential Consistency II

In both cases, the logical order is the same:

Sequential Consistency appears to process all requests in a single, interleaved ordering where:

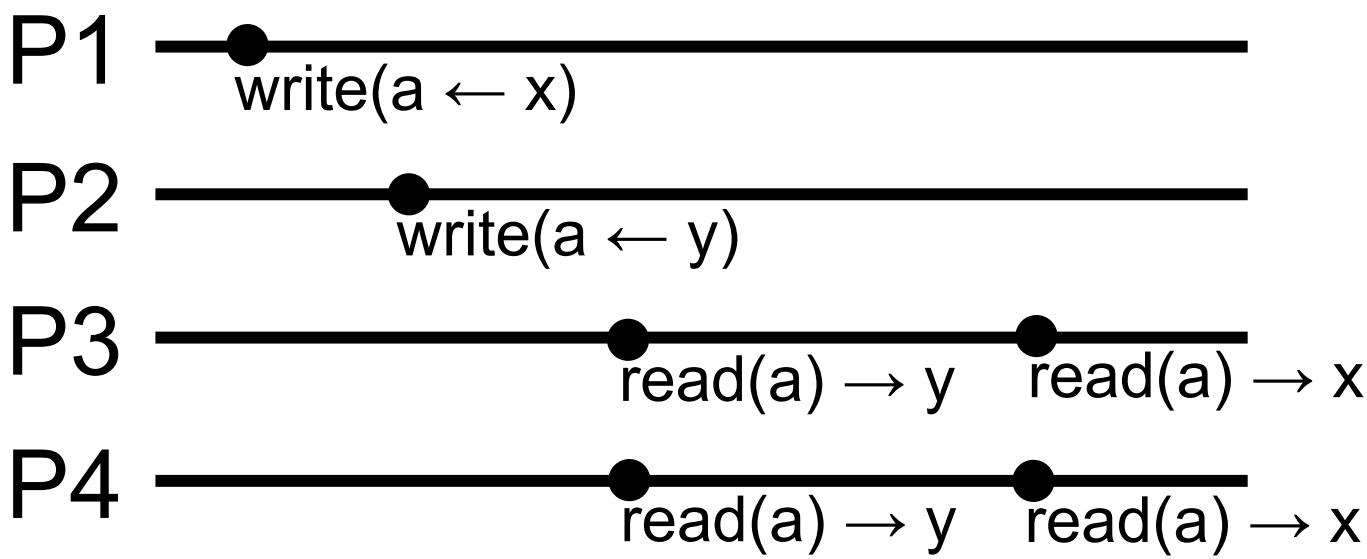
- Each process's program order is preserved.
- Ordering between processes is logically preserved, but may not preserve physical time ordering.

It appears as if all clients are reading from a single copy.

Sequential Consistency Examples III

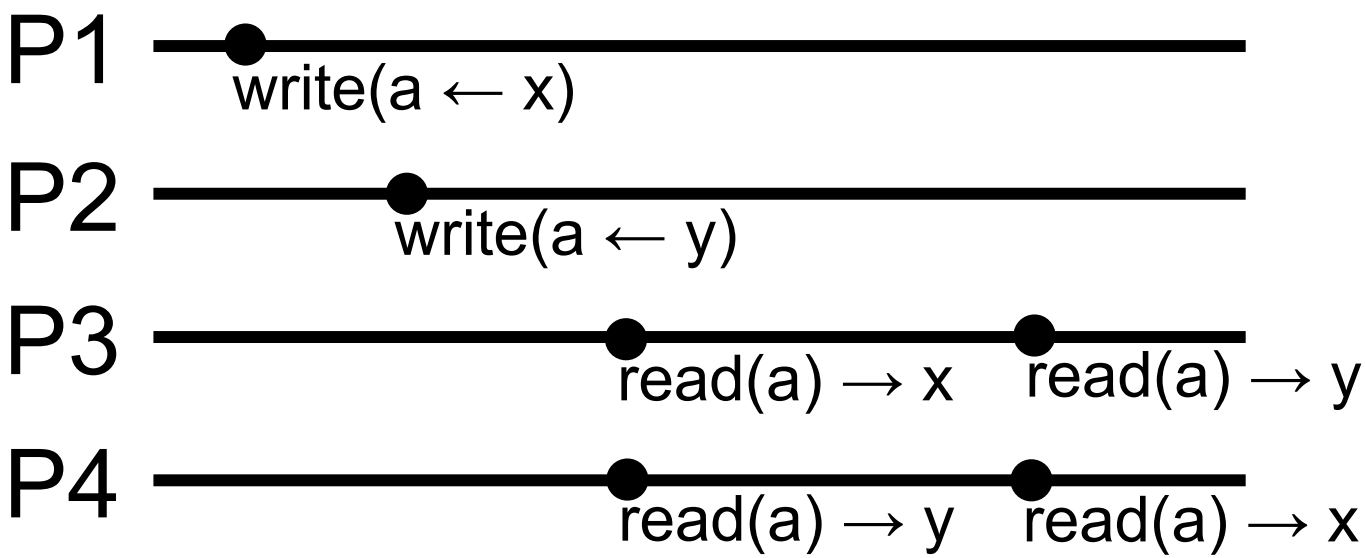
Under sequential consistency, is this example possible?

Is there an interleaving that behaves like a single copy?



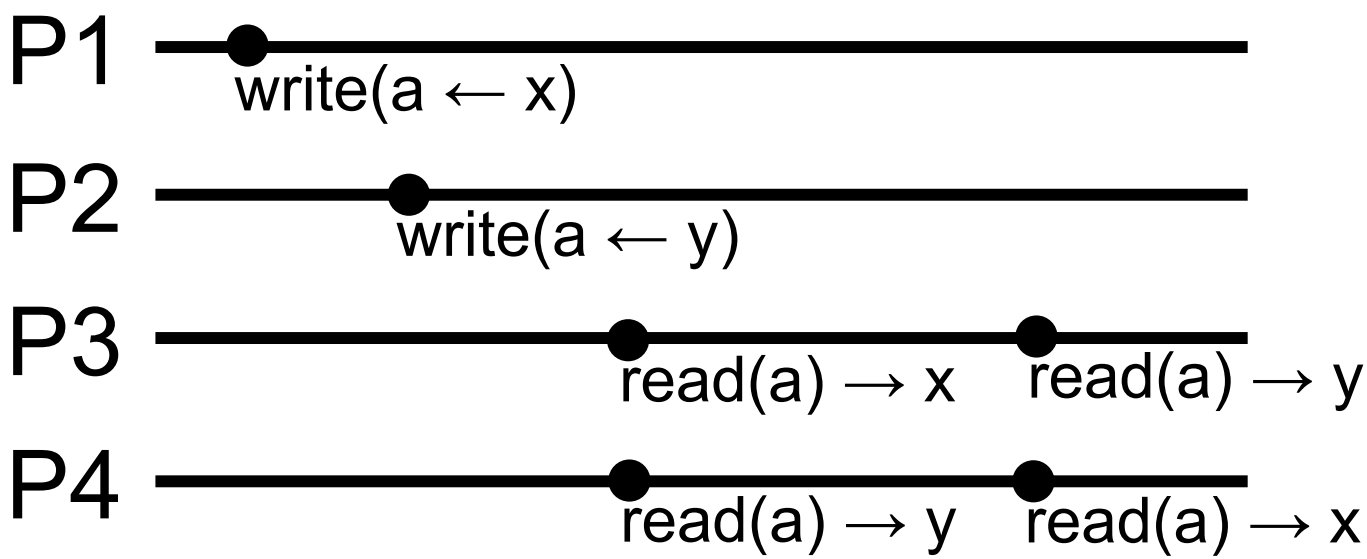
Sequential Consistency Examples IV

What about this?



Sequential Consistency Examples IV

What about this?



How could this happen?

Implementing Sequential Consistency

Typical implementation:

- The most recent write in physical time may not be visible (*i.e.*, applied to all replicas) at all times.
- However, all writes must appear in the same order across all replicas.
- The order in which writes appear should be total-FIFO.

Active Replication

- All reads and writes are FIFO ordered at the client.
- A read can complete on any single replica.
- Writes are totally ordered and synchronous.
  - Recall from PA 2B that total ordering can delay delivery of writes!
- This provides sequential consistency, but not linearizability.
  - Program order is preserved.
  - Writes are total ordered.
  - However, even non-overlapping writes may be reordered compared to physical time.

Relaxing Consistency Again

All of our models to date assume that we want a total ordering of operations.

That is, we want all operations to be totally ordered, and all operations to see all previous changes.

This is as if there were a single copy of the shared state.

What if we don't care quite that much?

Relaxing Consistency

We can define consistency models that have less strict guarantees.

We will consider two such models:

- Causal Consistency:
  Write operations must be ordered causally with respect to other operations.
- Eventual Consistency: All processes eventually converge to the same state, but may observe transient states that differ.

Causal Consistency

Causal consistency preserves causality for writes.

- Reads may occur out-of-order
- Entire transactions may use “stale” data
- Reads may fetch a newer value before an older value!

However:

- reads that causally precede a write determine the ordering of writes
- Writes must occur in order

Applying Causal Consistency

Where is causal consistency appropriate?

Applying Causal Consistency

Where is causal consistency appropriate?

- Replies on forums or social media
- Maintenance of non-critical data for dissemination (*e.g.*, usage statistics for network monitoring)
- *etc.*

Network Partitions

A partition is when one portion of the system cannot communicate with another portion of the system.

Partitions prevent even causal consistency for some writes.

We assume that a partitioning event is temporary.

Sequential consistency and linearizability prevent progress (*i.e.*, require blocking) during a partitioning event.

Eventual Consistency

Eventual consistency allows multiple partitions to make progress.

- Writes proceed independently within each partition
- Within the partition everything is consistent
- Between partitions, conflicting writes may occur
- When the partition heals:
  - Non-conflicting writes are kept
  - Conflicting writes are reconciled until all replicas agree

The CAP Theorem

The CAP theorem [2] presents a trilemma; pick any two [1] of:

- Consistency
- Availability
- Partition Tolerance

In the presence of network partitions, do you choose consistency or availability?

Brewer proposed CAP in 2000, Gilbert & Lynch proved it [3] in 2012.

The Reasons for CAP

On very large networks (like the Internet), temporary partitioning is almost unavoidable.

Partitioned systems must give up either availability or consistency [1].

Like most things, this is a design decision:

- Give up availability and retain total consistency
  - Perhaps with 2-phase commits
  - The system blocks until the partition heals
- Give up total consistency and retain availability
  - Perhaps with eventual consistency
  - Allow consistency to diverge until the partition heals

Dealing with Partitions

Pairs of conflicting writes may be allowed to proceed on different partitions.

These conflicts must be fixed up after the partition heals.

Typically, one of the writes will be lost. (Depending on protocols above the consistency layer, this may lead to cascading aborts.)

Quorum protocols can be used to determine whether writes may commit on a partition.

- Static quorums allow progress on any partition containing more than half of the replicas.
- Optimistic quorums allow progress on any partition.

Static Quorums

The decision on how many replica managers must be involved in an operation on replicated data is called quorum selection.

Any static quorum requires that:

- At least r replicas are accessed for read.
- At least w replicas are accessed for write.
- $r + w > N$, where N is the number of replicas.
- $w > N/2$
- Every object has a version number or consistent timestamp.

Static Quorum Mechanics

Why must $r + w > N$?

Static Quorum Mechanics

Why must $r + w > N$?

- This guarantees overlap between read and write sets.
- There is always some replica in the read set that has the most recent write.

Static Quorum Mechanics

Why must $r + w > N$?

- This guarantees overlap between read and write sets.
- There is always some replica in the read set that has the most recent write.

Why must w be greater than $N/2$?

Static Quorum Mechanics

Why must $r + w > N$?

- This guarantees overlap between read and write sets.
- There is always some replica in the read set that has the most recent write.

Why must w be greater than $N/2$?

- If there is a network partition, only the partition with more than half of the replicas can write.
- The rest will serve reads with stale data.
- Re-joining the partition will have no conflicting writes.

r and w are tunable:

- *E.g.*, $N = 3$, $r = 1$, $w = 3$: High read throughput, lower write throughput, reads continue during any partition.

Optimistic Quorum Approaches

An optimistic quorum allows writes to proceed in any partition.

Writes are not committed during a partitioning event.

Write-write conflicts are resolved after the partition heals.

Optimistic Quorum is practical when:

- Conflicting updates are rare
- Conflicts are always detectable
- Damage from conflicts can be easily confined
- Repair of damaged data is possible or updates can be discarded without consequences
- Partitioning events are short-lived

Summary I

- Linearizability
  - Provides the strongest consistency guarantees.
  - Provides single-client, single-copy semantics.
  - A read operation returns the most recent write according to physical time ordering, regardless of the number of clients or replicas.
  - Can be implemented by: primary-backup replication, chain replication.
- Sequential Consistency
  - Provides single-copy semantics.
  - Program order is preserved for each client.
  - Can be implemented by active replication.

Summary II

- Causal Consistency and Eventual Consistency
  - Provide much weaker consistency guarantees.
  - May be appropriate for some applications.
- Quorums
  - Static
  - Optimistic

References I

Required Readings

[5] Textbook. Sections 18.1, 18.5.

Optional Readings

[1] Eric A. Brewer. "CAP Twelve Years Later: How the "Rules" Have Changed". In: *Computer* 45 (2 2012), pp. 23–29. URL: https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed.

[2] Eric A. Brewer. *Towards Robust Distributed Systems*. Keynote Speech at the ACM Symposium on the Principles of Distributed Computing. July 2000. URL: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[3] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *ACM SIGACT News* 33 (2 June 2002), pp. 51–59. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf.

[4] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX, 2004.

Acknowledgements

These slides are based on slides from Steve Ko, used with permission.

Those slides contain the following attribution:
These slides complain material developed and copyrighted by Indranil Gupta (UIUC).