

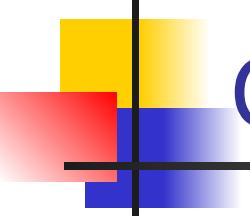


Replication & Consistency

Replication: Creating and using multiple copies of data (or services)

Why replicate?

- Improve system reliability
 - Prevent data loss : i.e. increase data **durability**
 - Increase data/service **availability**
 - Note: availability \neq durability
 - Increase confidence: e.g. deal with byzantine failures
- Improve performance
 - Scaling throughput
 - Reduce access times



Context

Rather trivial

Stateless service replication

- e.g., Web server dealing with page layout (constructed somewhere else), services in application tier in a 3-tier architecture

Data replication

- E.g., File system, web site mirrors, browser caches, DNS

Our focus for now

Control replication

- Model: replicated state machine;
- Stateful services (data and control), e.g., critical infrastructure service



What are the issues?

Issue 1. Dealing with data changes

- **Consistency models**
 - ***What is the semantic*** the system implements?
[not all applications require 'strict' consistency]
- **Consistency protocols**
 - ***How to implement*** the semantic agreed upon?
[Tools: (a) Operation ordering (delay some operations);
(b) Operation assignment to replicas]

Issue 2. Replica management

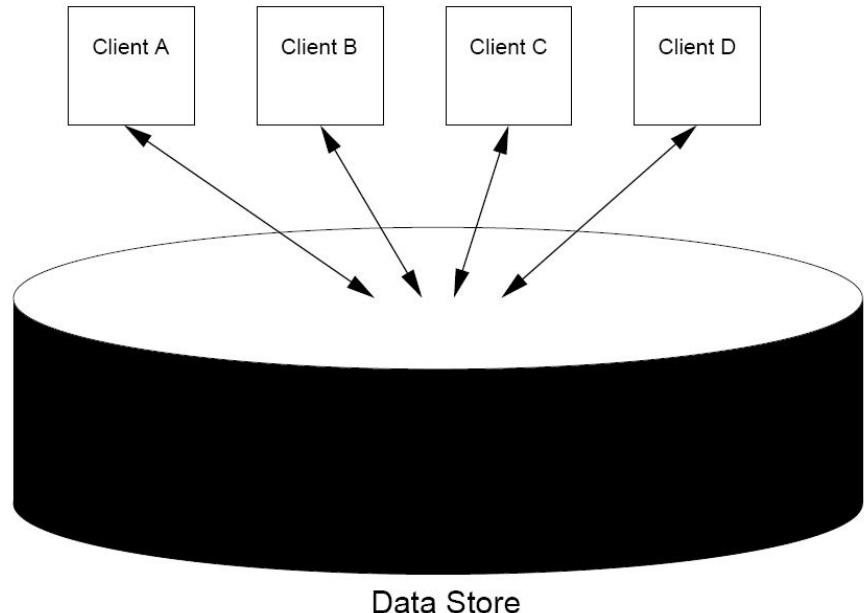
- How many replicas? Where to place them? When to get rid of them?

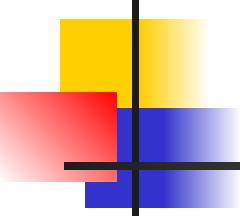
Issue 3. Request redirection/routing

- Which replica should clients use?

Client's view of the data-store

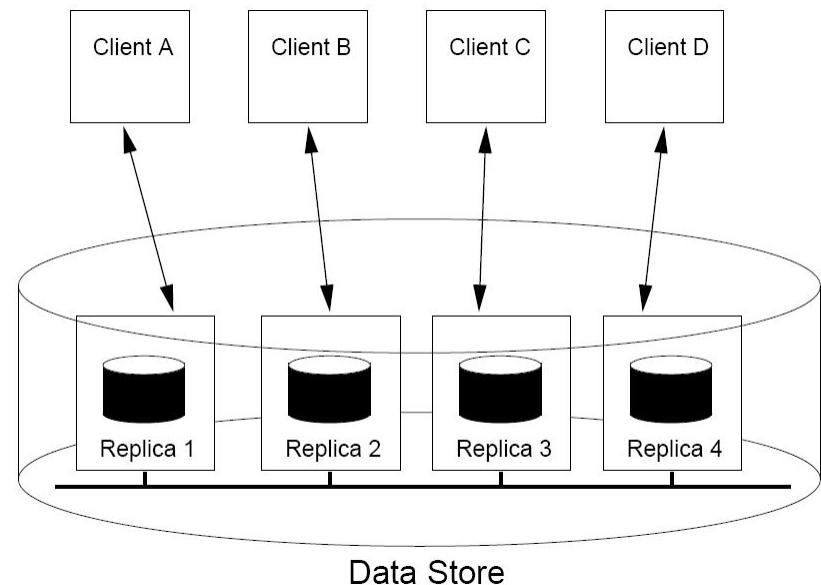
Ideally 'black box' – i.e., complete 'transparency' over how data is stored and managed

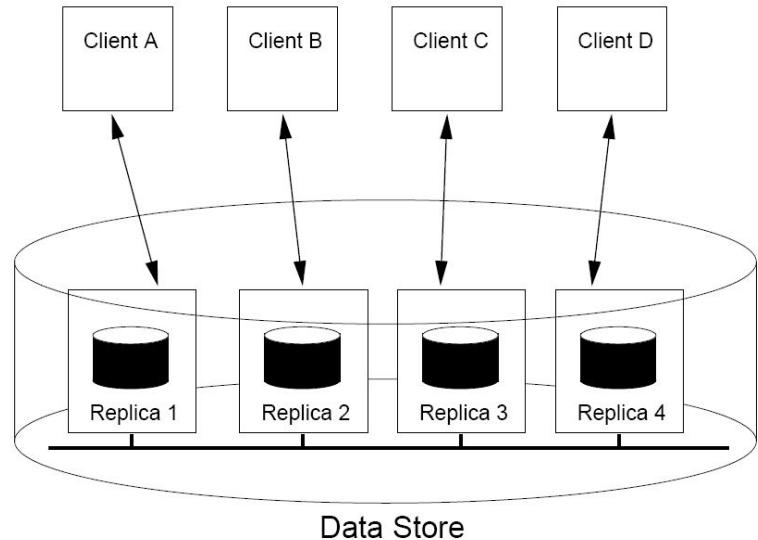




Replica management system view on data store:

controls the allocated resources and aims to provide transparency

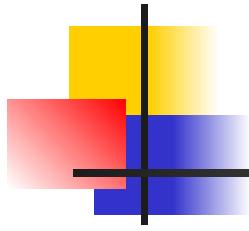




Consistency model:

Contract between: *(i)* the data store, and
(ii) the clients:

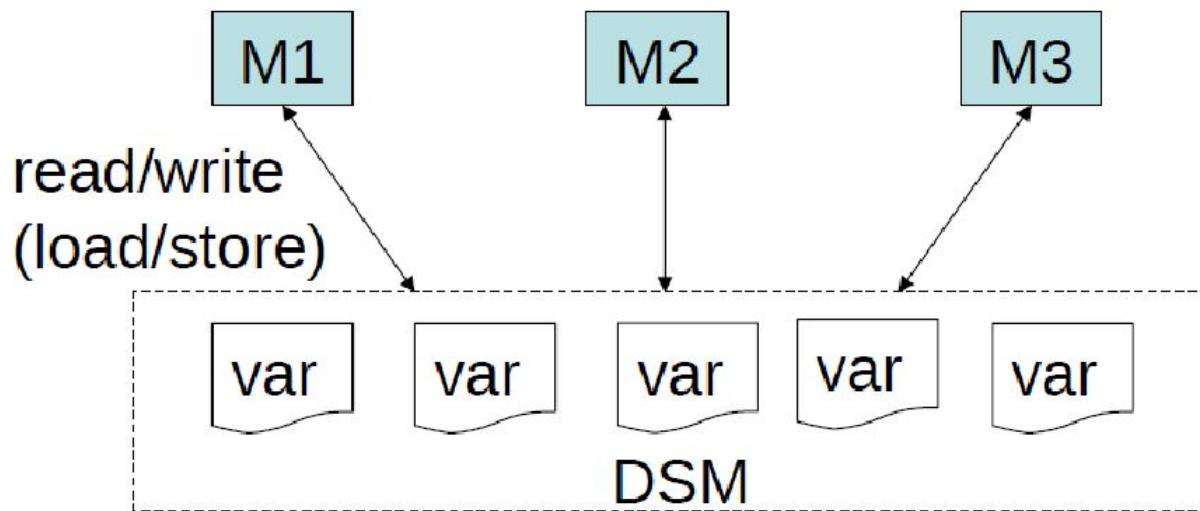
specifies the acceptable result of reads and writes in the presence of concurrent operations, failures, etc.



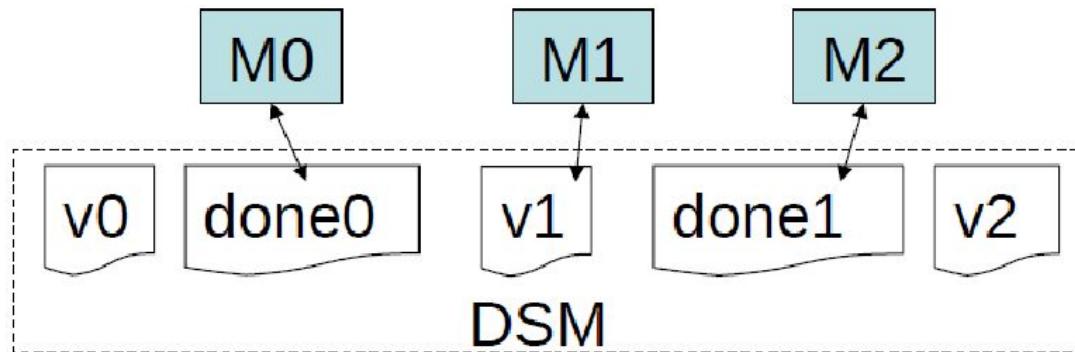
Why do I need one?

An example scenario: Distributed Shared Memory

- Two models for communication in distributed systems:
 - message passing
 - shared memory
- Shared memory is often thought more intuitive to write parallel programs than message passing
 - Each machine can access a common address space



An example scenario: Distributed Shared Memory



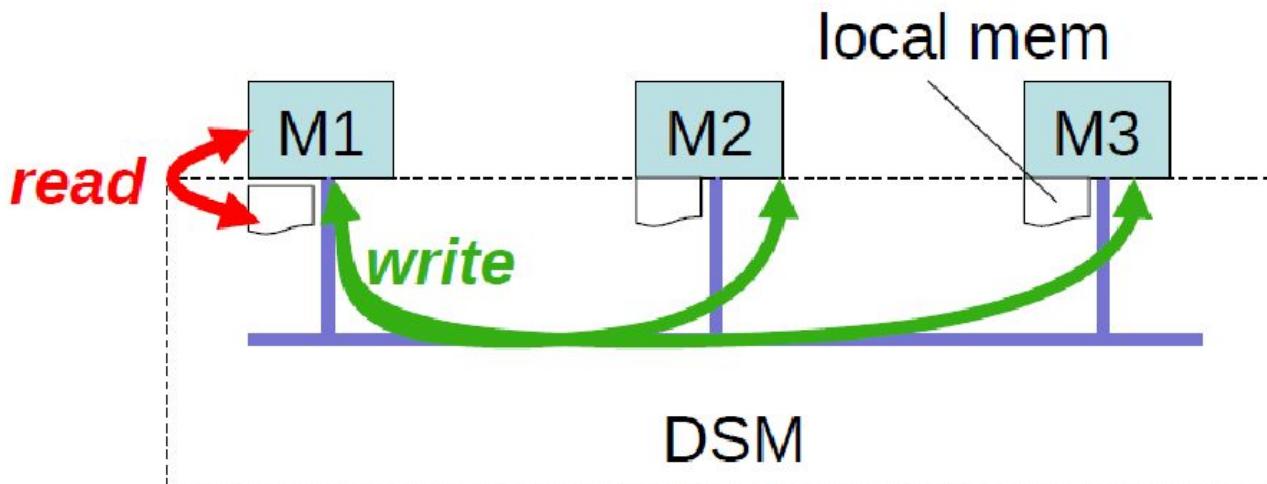
M0:
v0 = f0();
done0 = 1;

M1:
while (done0 == 0)
;
v1 = f1(v0);
done1 = 1;

M2:
while (done1 == 0)
;
v2 = f2(v0, v1);

- **What's the intuitive intent?**
 - M2 should execute f2() with results from M0 and M1
 - waiting for M1 implies waiting for M0

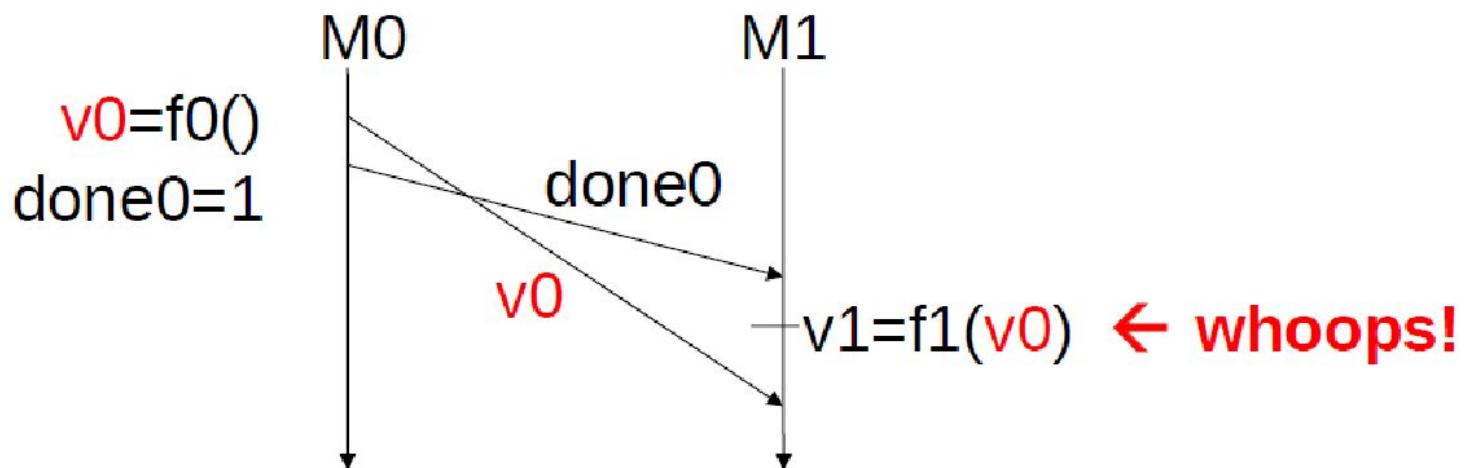
A naïve DSM implementation:



- Each machine has a **local copy** of all of memory
 - Operations:
 - **Read**: from local memory
 - **Write**: send update msg to each host (but **don't wait**)
 - Fast: never waits for communication
- Question: Does this DSM work well for our application?

Problem 1 with the naïve implementation

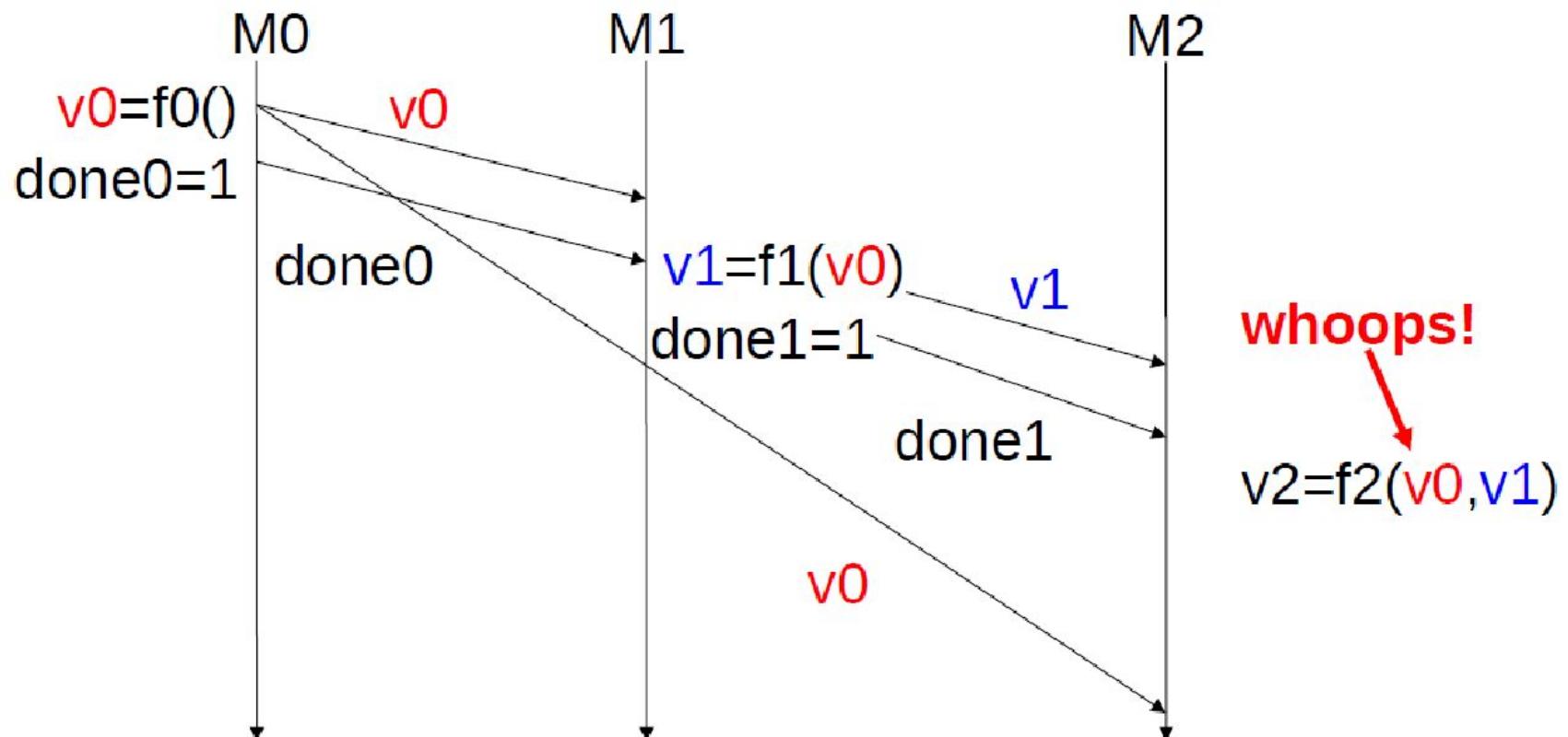
M0's $v0 = \dots$ and $\text{done}0 = \dots$ may be interchanged by network, leaving $v0$ unset but $\text{done}0 = 1$

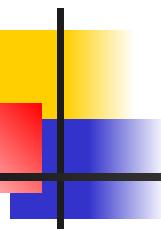


Problem 2 with the naïve implementation

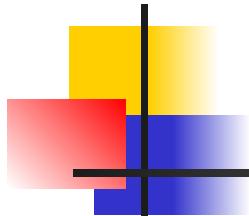
M2 sees M1's writes before M0's writes

- I.e. M2 and M1 disagree on order of M0 and M1 writes

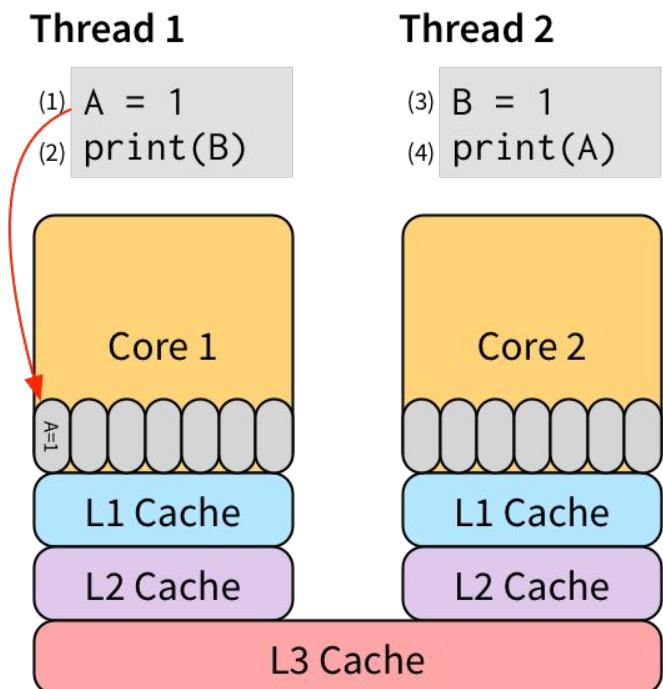


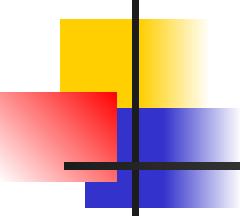


- Naïve DSM is fast but has unexpected behaviour
- Maybe DSM is not “correct”
- ... or maybe I should have never expected the application to work in the first place
- the **consistency model**
 - The “contract” the DSM will obey and the application will rely on



Do modern multi-processor architectures provide sequential consistency?





Roadmap

- **(1) Consistency models:**
 - contracts between the data store and the clients that specify the [acceptable] result of read/write operation stream in the presence of conflicting operations.
- **(2) Consistency protocols**
 - How does one implement a consistency model?
 - Tools
 - Operation ordering, delay some
 - Operation assignment to replicas
- **(3) Replica management:** creation, placement, deletion
- **(4) One system** that integrates many of the techniques we have discussed so far

[high level] one way to provide consistency: ensure that **conflicting** operations are done in the same order everywhere

Conflicting operations: (from the world of transactions):

- **Read–write conflict:** a read operation and a write operation act concurrently
- **Write–write conflict:** two concurrent write operations

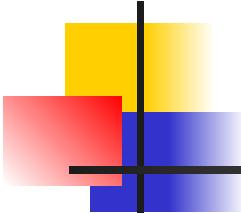
Problem: Guaranteeing global ordering on conflicting operations may be costly, reducing scalability

Solution: Weaken the consistency model (adjust consistency requirements) so that hopefully global synchronization can be avoided

**Scalability
Performance**

□TENSION□

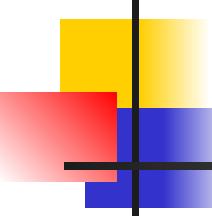
Programmer friendliness
(often implies management overheads)



Consistency model: Contract between the data store and the clients: specifies the [acceptable] results of read and write operations in the presence of concurrent operations.

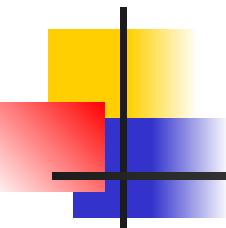
Choices that determine the design space:

- Contract covers all items or subsets (or even each item individually)?
- Contract covers all users or individual users?
- Contract covers sets of operations (e.g., transactions, for databases; or multiple ops on the same object, for files)



Recap

- Why are replication (and caching) needed?
 - For performance, scalability, fault tolerance (availability, durability), ability to operate while disconnected
- When do consistency concerns arise?
 - With replication and caching (our focus)
 - Concurrent (sets of) operations on shared data
 - e.g., databases, filesystems

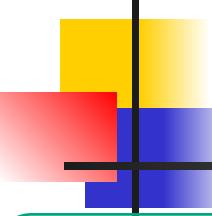


The space to cover is huge – so the ‘consistency’ as contract terminology appears in lots of contexts.

E.g., what type of consistency do distributed filesystems provide?

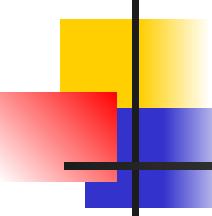
- session consistency (AFS)
 - aka. – open-to-close consistency
- ‘close-to-open’ consistency (NFS > v2.4.10)?

GFS: see discussion [here](#)



ACID properties

- **Atomic: All or nothing**
 - State shows either all the effects of a transaction, or none of them:
- **Consistent (think of it as Correct): Guarantees basic properties**
 - A transaction moves the database from a state where integrity holds, to another where integrity holds (eg: constraints)
- **Isolated: Each transaction runs as if alone**
 - Effect of concurrent transactions is the same as transactions running one after another (ie looks like batch mode)
- **Durable: Cannot be undone**
 - Once a transaction has committed, it can not be undone in spite of failures.

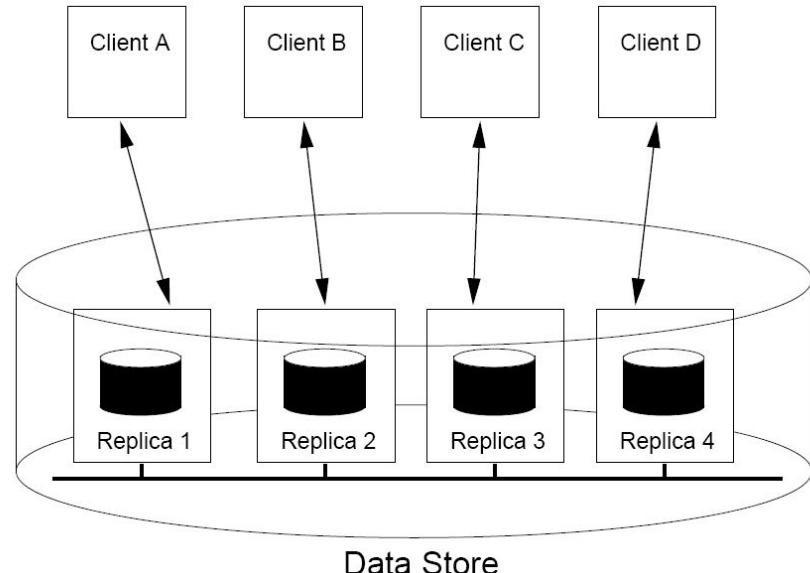


Recap

- Why are replication (and caching) needed?
 - For performance, scalability, fault tolerance (availability, durability), ability to operate while disconnected
- When do consistency concerns arise?
 - With replication and caching (our focus)
 - Cover one operation at a time (rather than sets)
 - Concurrent (sets of) operations on shared data
 - e.g., databases, filesystems

Consistency models (for replicated data)

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency
 - Limit the deviation between replicas
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently

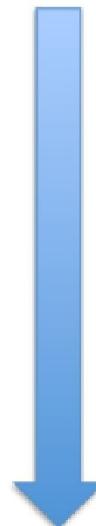


Data Store

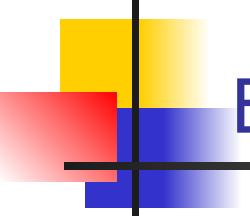
Consistency models

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency

- Strict
- Strong (Linearizability)
- Sequential
- Causal
- Eventual



Weaker
Consistency
Models



Eventual Consistency

Original idea: If single point to insert updates (no write-write conflicts), and no updates take place for a long enough period time, make sure all replicas will gradually (i.e., eventually) become consistent.

Where does this work well?

- Mostly read-only workloads, and
- No concurrent updates
 - e.g., updates have only one source, or all updates performed through a master replica
- Clients can operate with out-of-date data

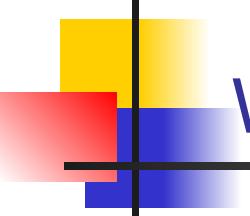
Two-word summary: best-effort



Eventual Consistency (II)

No constraint on operation ordering

- Allow stale reads, but ensure that reads will eventually reflect previously written values
 - Even if this takes very long ...
- Don't order concurrent writes as they are executed
 - which might create conflicts later: which write was first?
 - Used by Amazon's Dynamo ...
 - with a mechanism for read reconciliation based on vector clocks

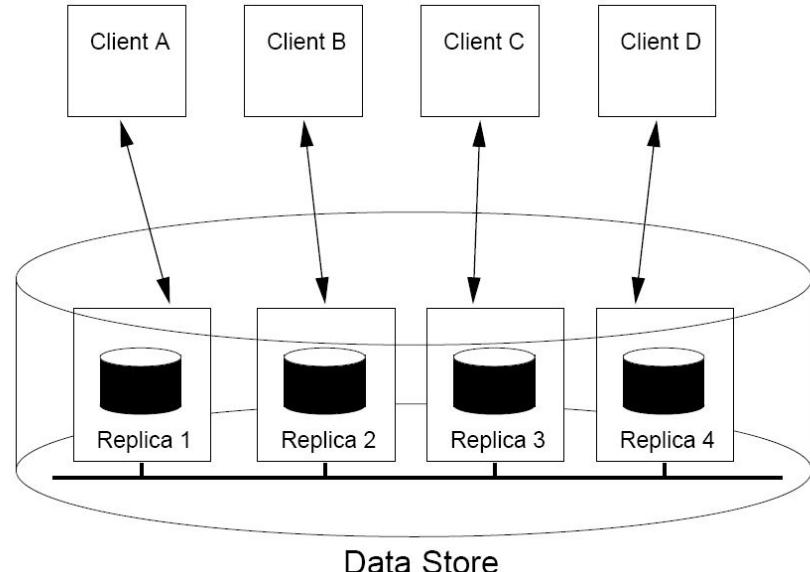


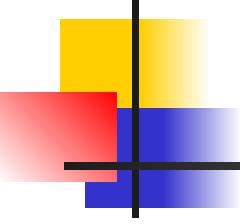
Why / Why NOT Eventual consistency

- [+] Support disconnected operations or network partitions
 - [for some apps] Better to read a stale value than nothing
 - [for some apps] Better to save writes somewhere than nothing
- [+] Support for increased parallelism
 - Though that's not what people have typically used this for
- [-] Potentially anomalous application behavior –
Stale reads and conflicting writes...

Consistency models

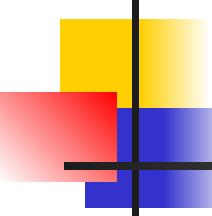
- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Continuous consistency
 - Limit the deviation between replicas
 - Eventual consistency
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering **for each client independently**





Roadmap

- **(1) Consistency models:**
 - contracts between the data store and the clients that specify the [acceptable] result of read/write operation stream in the presence of conflicting operations.
- **(2) Consistency protocols**
 - How does one implement a consistency model?
 - Tools
 - Operation ordering, delay some
 - Operation assignment to replicas
- **(3) Replica management:** creation, placement, deletion
- **(4) One system** that integrates many of the techniques we have discussed so far

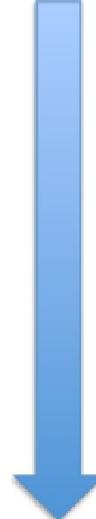


Recap

- Why are replication (and caching) needed?
 - For performance, scalability, fault tolerance (availability, durability), ability to operate while disconnected
- When do consistency concerns arise?
 - With replication and caching (our focus)
 - Cover one operation at a time (rather than sets)
 - Concurrent (sets of) operations on shared data
 - e.g., databases, filesystems

Consistency models

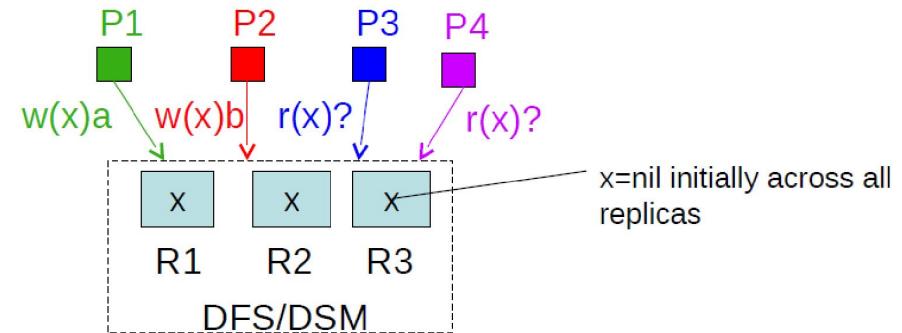
- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency
 - Strict
 - Strong (Linearizability)
 - Sequential
 - Causal
 - Eventual



Weaker
Consistency
Models

Notations:

- Read: $R_i(x)$ a -- client i reads a from location x
- Write: $W_i(x)$ b -- client i writes b at location x



Consistency model defines what values reads are admissible by the DFS/DSM

P1:	$w(x)a$	
P2:	$w(x)b$	
P3:	$r(x)?$	$r(x)?$
P4:	$r(x)?$	$r(x)?$

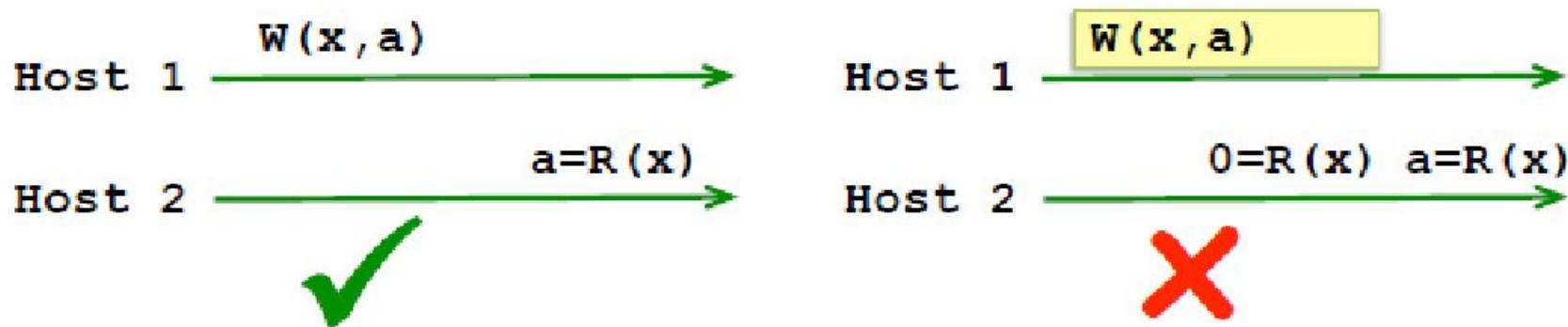
wall-clock time

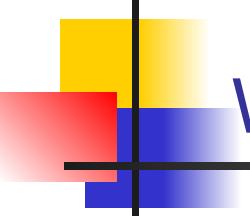
Time at which client process issues op

May differ from the time at which the op request gets to relevant replica!

Strict Consistency

- Strongest consistency model we'll consider
 - Any read on a data item X returns value corresponding to result of the most recent write on X
- Need an absolute global time
 - “Most recent” needs to be unambiguous
 - Corresponds to when operation was issued
 - Impossible to implement in practice on multiprocessors





What do reads return with 'strict consistency'

wall-clock time



P1: $w(x)a$

P2: $w(x)b$

P3: $r(x)?$ $r(x)?$

P4: $r(x)?$ $r(x)?$

Sequential Consistency

[the technical definition] The result o

- operations were executed in some
- the operations of each individual process appear in the order in which they are issued.

[an intuitive definition]: a system is sequentially consistent if a system with no replication and *synchronous* operations could have produced any of its traces.

P1: W(x)a

P2: W(x)b

P3: R(x)b R(x)a

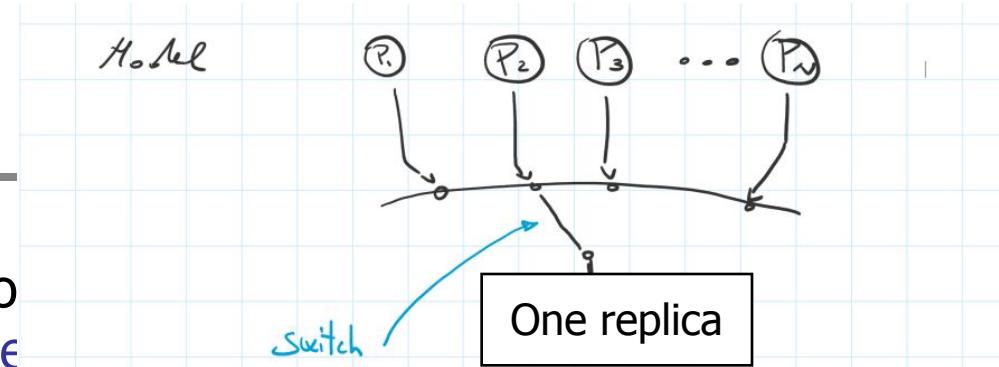
P4: R(x)b R(x)a

(a)

Are there other outputs a sequential system could produce?

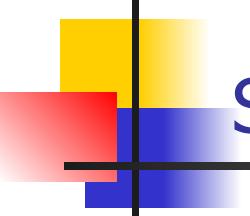
Can you list all of them?

Give an example of a trace that is sequentially consistent but not strictly



strict vs. sequential consistency

- In sequential consistency, the [physical time] ordering of events does NOT matter.
 - (except for events in the same process)
- All that is required for sequential:
 - All processors see the same ordering of operations
 - (regardless of whether this is the order that the operations actually occurred)
 - except operations in the same process which must preserve their actual ordering.



Sequential Consistency (example)

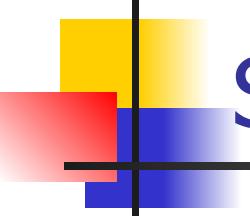
3 processes (P1, P2, P3) and 3 (replicated) variables X, Y, Z initialized to 0

	Process 1:	Process 2	Process 3
Parallel	X \square 1 Read (Y, Z)	Y \square 1 Read (X, Z)	Z \square 1 Read (X, Y)
Sequential	Collect output: print _{P1} (Y ^{P1} , Z ^{P1}); print _{P2} (X ^{P2} , Z ^{P2}); print _{P3} (X ^{P3} , Y ^{P3})		

Is output: 11 11 11 possible if you assume sequential consistency?

What about: 00 00 00?

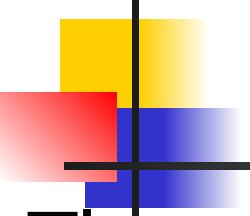
What about: 00 10 10?



Sequential vs. Eventual Consistency

One view:

- Sequential: **pessimistic** concurrency handling
 - Decide on update order as they are executed
- Eventual: **optimistic** concurrency handling
 - Let updates happen, worry about deciding their order later
 - May raise conflicts
 - Think about when you code offline for a while – you may need to resolve conflicts with other team members when you commit
 - Resolving conflicts is not that difficult with code, but it's really hard in general (e.g., think about resolving conflicts when you've updated an image)



Linearizability (linearizable consistency)

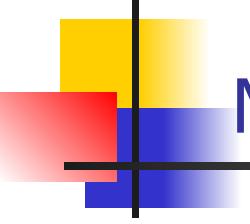
(aka. **external** consistency)
(aka. **strong** consistency)

The result of any execution is the same as if

- (1) operations by all processes were executed in **some** sequential order, and
- (2) this order reflects
 - (2') [sequential consistency] the order in each individual process
 - (2'') [linearizability] the order in which the operations execute in real time*

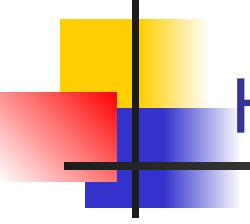
strict < linearizable < sequential

* e.g., assume operations are synchronous (then non-overlapping operations can be ordered), or assume some loosely synch clocks to get order



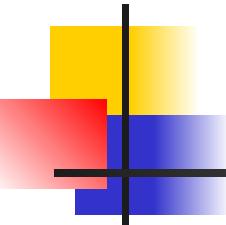
More examples

- Give an example of a trace that is generated by a system that is:
 - (a) NOT sequential
 - (b) sequential but NOT linearizable
 - (c) linearizable but NOT strictly consistent
- If the DSM is sequentially consistent does the code in the example before (slide 10) 'work'?
 - Yes.



Harder Problem

- Give an example of a trace of a system that demonstrates that **sequential consistency is not composable**
 - i.e., when having data items (or data stores) that are each (independently) kept sequentially consistent, their composition as a set need not be so



Datastore X

P1: $W(x)a$

$R(x)a$

P2: $W(x)b$

Let's assume that no matter how far further on gets in time the result is still $R(x)a$ (same assumption below for $R(y)b$)

Datastore Y

P1: $W(y)a$

P2: $W(y)b$

$R(y)b$

Combined datastore X and Y

P1: $W(x)a$

$W(y)a$

$R(x)a$

P2: $W(y)b$

$W(x)b$

$R(y)b$

Traces observed on datastores X and Y independently do not violate sequential consistency

Combined trace on X and Y violates sequential consistency for the combined store (see next slide) for argument

The original trace

P1: $W(x)a \quad W(y)a$

P2: $W(y)b \quad W(x)b$

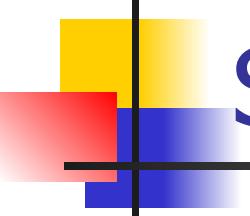
Writes Have Propagated

$R(x)a$

$R(y)b$

The possible ways of sequencing the operations

$W(x)a$	$W(y)a$	$W(y)b$	$W(x)b$	$x=b \quad y=b$
$W(x)a$	$W(y)a$	$W(x)b$	$W(y)b$	not possible - would violate ordering in P2
$W(x)a$	$W(y)b$	$W(y)a$	$W(x)b$	$x=b \quad y=a$
$W(x)a$	$W(y)b$	$W(x)b$	$W(y)a$	$x=b \quad y=a$
$W(x)a$	$W(x)b$	not possible - would violate ordering in P2
$W(y)a$	not possible - would violate ordering in P1
$W(y)b$	$W(x)a$	$W(y)a$	$W(x)b$	$x=b \quad y=a$
$W(y)b$	$W(x)a$	$W(x)b$	$W(y)a$	$x=b \quad y=a$
$W(y)b$	$W(y)a$	not possible
$W(y)b$	$W(x)b$	$W(x)a$	$W(y)a$	$x=a \quad y=a$
$W(y)b$	$W(x)b$	$W(y)a$	$W(x)a$	not possible
$W(x)b$	not possible



So far ...

- **Consistency models:**

- **contract** between the data store and the clients that specify the [acceptable/possible] results of read/write operations in a stream in the presence of conflicting operations.

Linearizability

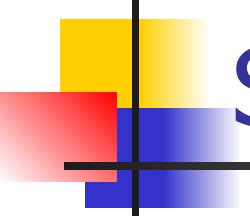
(strong)

Causal

Eventual



Sequential



So far ...

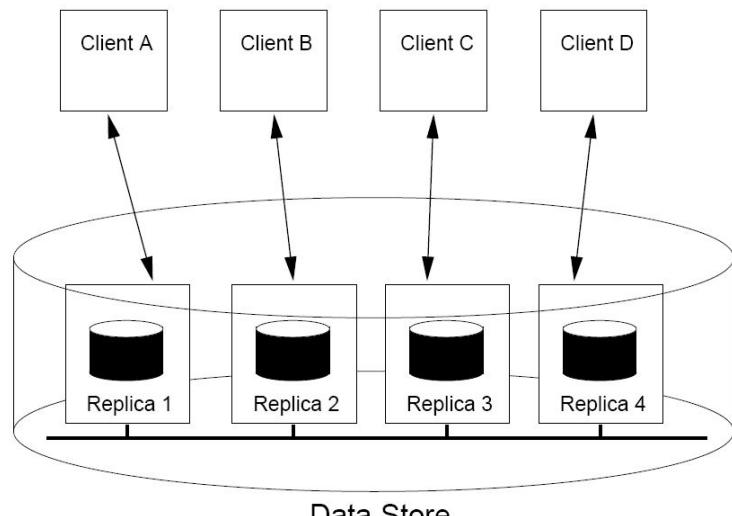
- **Consistency models:**
 - **contract** between the data store and the clients that specify the [acceptable/possible] results of read/write operations in a stream in the presence of conflicting operations.
- **Consistency protocols**
 - How does one implement a consistency model?
 - Tools to exploit
 - Operation ordering, delay some
 - Operation assignment to replicas

Consistency protocols

Question: How does one design a protocol to implement the desired consistency model?

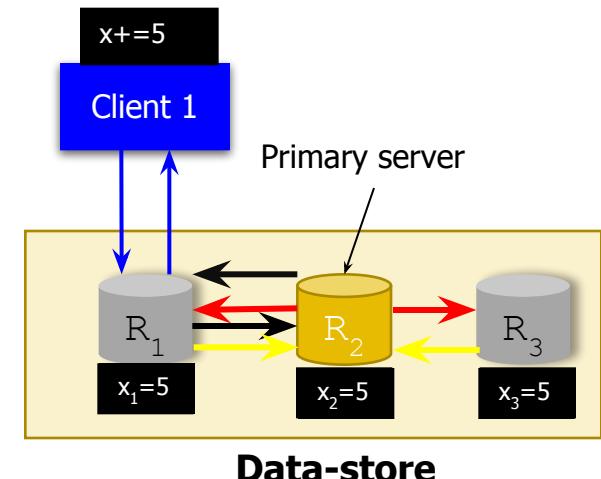
[focus] Sequential consistency

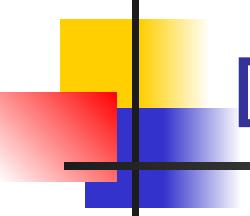
- Primary / backup
- Chain replication
- Quorum protocols



A protocol for primary-backup

- Overview (note that many variants are possible!):
 - All write operations are forwarded to the **primary replica**
 - Write operations are synchronous (i.e., blocking)
 - Read operations to any replica
- Approach for write ops
 - Client connects to some replica R_c
 - If the client issues write operation to R_c :
 - R_c forwards the request to the primary replica R_p
 - R_p updates its local value
 - R_p forwards the update to other replicas R_i
 - Other replicas R_i update, and send an ACK back to R_p
 - After R_p receives all ACKs, it informs R_c that the write operation is completed
 - R_c acknowledges the client, which in return completes the write operation





Discussion (I)

- Protocol provides a simple way to implement sequential consistency
 - Writes are performed in the same order on all replicas
 - Guarantees that clients on node A see the most recent write operations initiated at A
- However, latency is high
 - Writes block until all the replicas are updated
- Use: distributed databases and file systems that require fault-tolerance
 - Replicas are placed on the same LAN to reduce latency

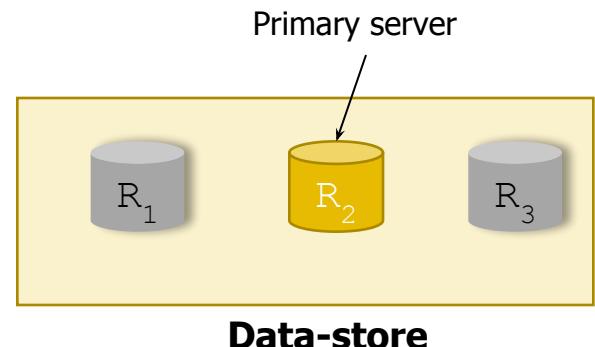
Discussion (II)

Q: Could a non-blocking strategy be applied?
(And maintain the same consistency properties?)

- Writes return immediately after touching a replica? Writes return after touching primary? What assumptions would you make? What properties result?

Q: How to deal with failures?

- Of backup replicas
- Of the primary?

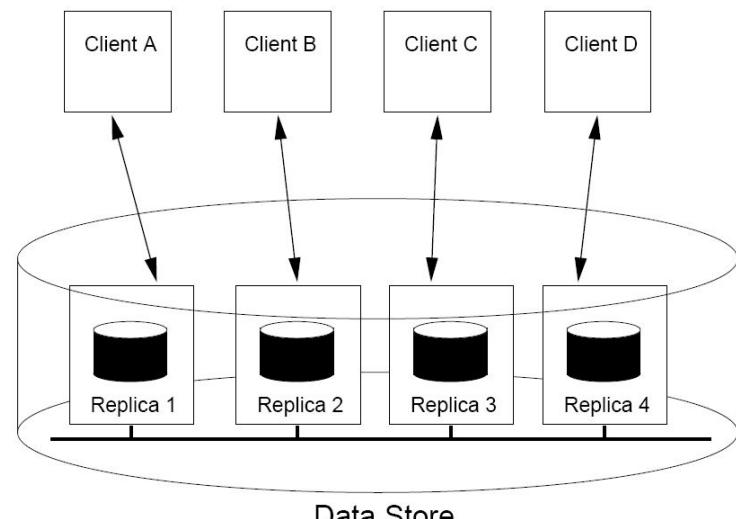


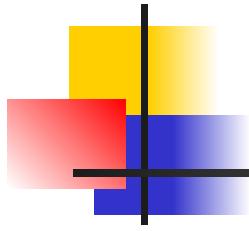
Consistency protocols

Question: How does one design a protocols to implement the desired consistency model?

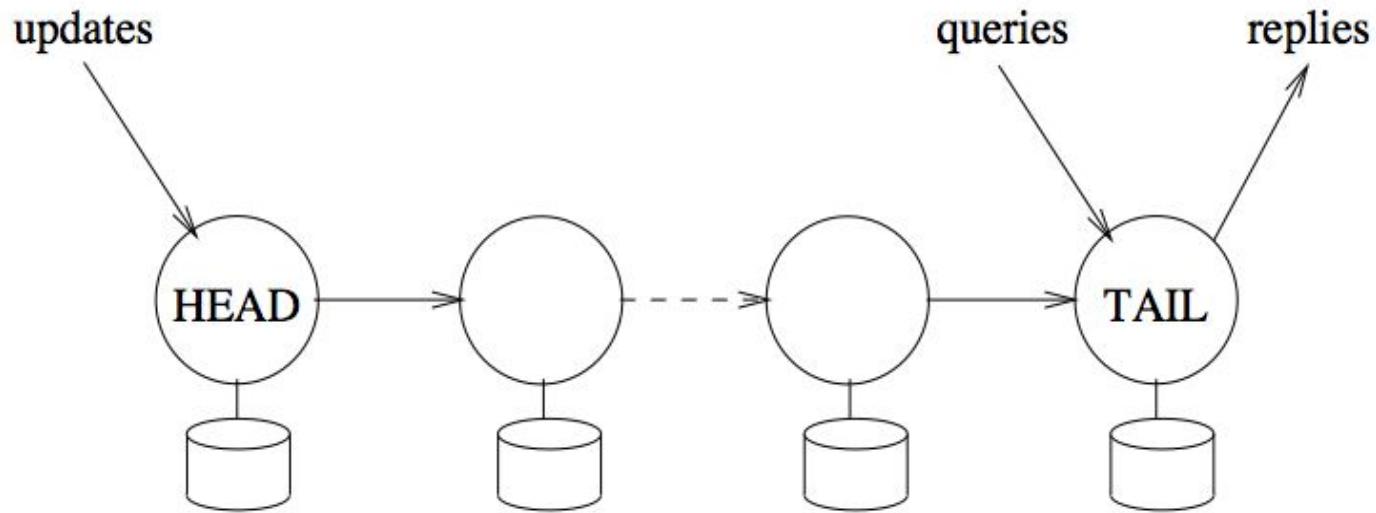
Sequential consistency

- Primary / backup
- Chain replication
- Quorum protocols





Chain replication

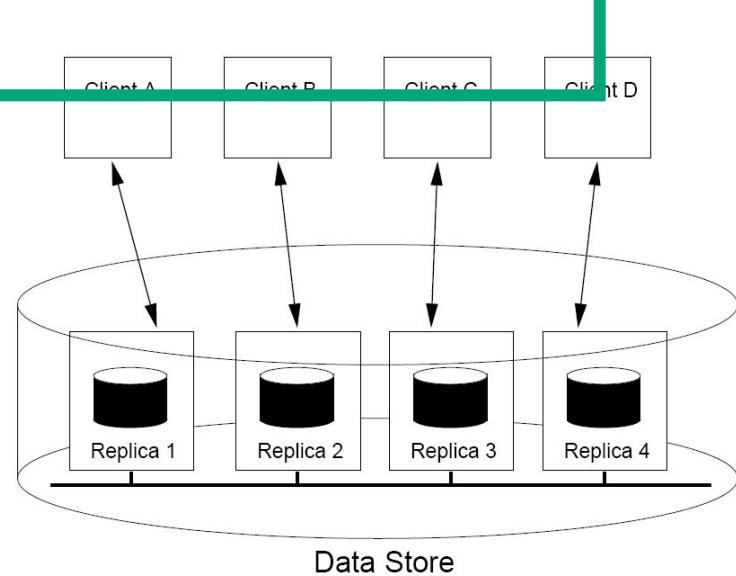


Consistency protocols

Question: How does one design a protocols to implement the desired consistency model?

Sequential consistency

- Primary / backup
- Chain replication
- Quorum protocols



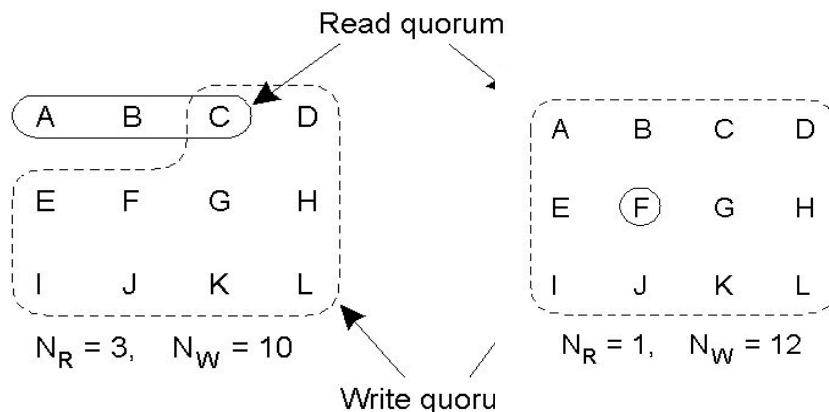
Replicated writes: Quorums

Problem (with primary-backup scheme): some replicas may not be available all the time / may be slow

- leads to low availability (for writes) / high response time

Solution: quorum-based protocols: Ensure that each operation is carried on enough replicas (not all)

- a majority 'vote' is established before each operation
- distinguish a **read quorum** and a **write quorum**:



What's a valid quorum configuration?

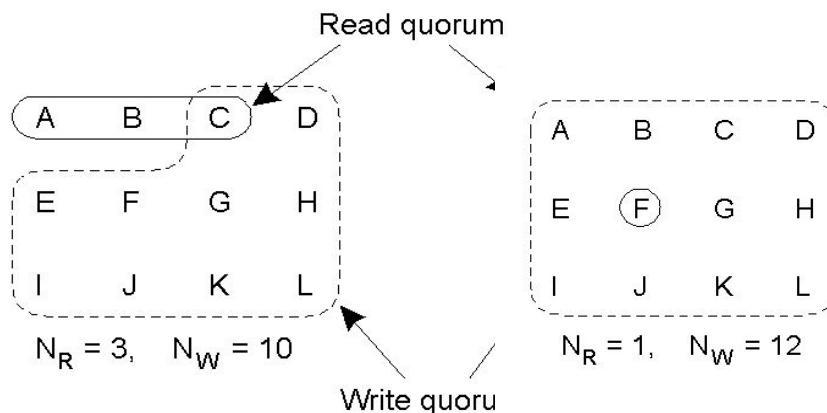
Two basic rules for correctness

- A **read quorum** should “intersect” any prior write quorum at ≥ 1 processes

$$Q_r + Q_w > N$$

- A **write quorum** should intersect any other write quorum

$$Q_w + Q_w > N \text{ results in } Q_w > N/2$$



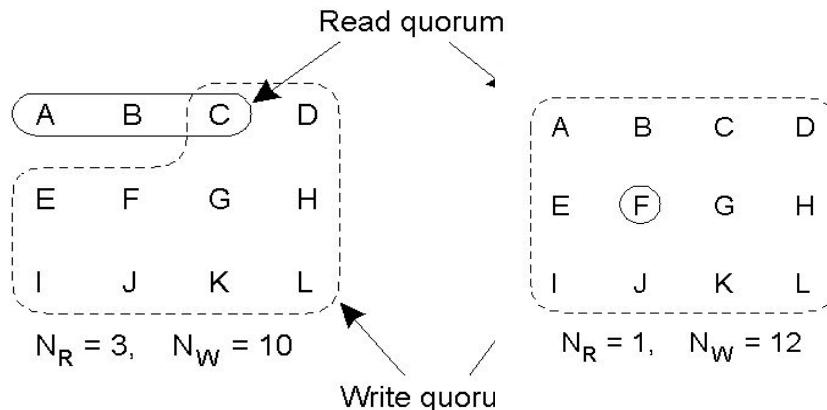
Quorums: Mechanics for read operations

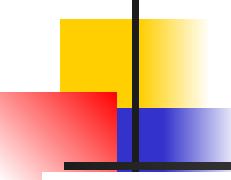
Setup: Replicas have “versions”

- Versions are numbered (timestamped)
- Timestamps must increase monotonically (process id to break ties)

Read operations:

- Client send RPCs until Q_r processes reply
- Then use the replica with the largest timestamp (the most recently updated replica)





Quorums: Mechanics write operations

Setup: Replicas have “versions”,

- Versions are numbered (e.g., logically timestamped)
- Timestamps must increase monotonically

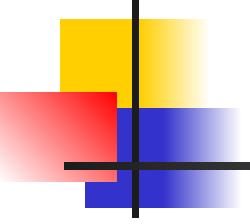
Writes:

- Need to determine next version number.
- Need a protocol to protect from concurrent writes
 - Need to make sure all replicas are updated atomically

Quorum based protocols: Write algorithm

(similar to two-phase commit)

- Client: Contacts all replica hosts and propose the write.
 - "I would like to execute write W_j on data-item I "
- Replica:
 - Locks the replica against other writes (may fail).
 - Puts the request in a queue of pending writes
 - Sends back: ACK, proposed version-number (e.g., logical clock) , pID
- Client:
 - If $< Q_w$ replies: send ABORT to all participants
 - If $\geq Q_w$ replies:
 - new_replica_number = max [versions] + 1
 - Send (COMMIT, new_replica_number)
- Replica:
 - Commit, unlock



Wrapping up: a note on trade-offs

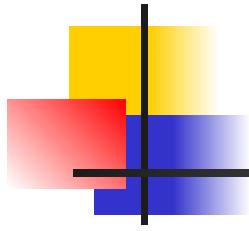
Linearizability
(strong)

Causal

Eventual

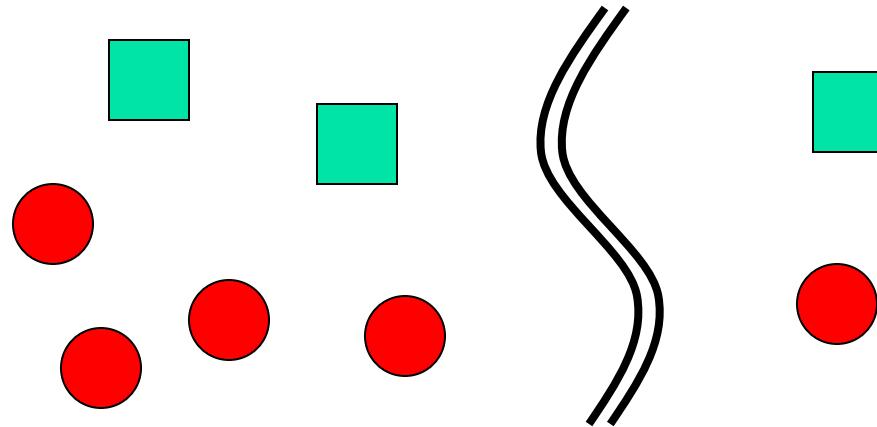


Sequential



 : Replicas

: Clients



C+PT (-A)

Option 1:

accept reads
reject writes

accept reads
reject writes

Option 2:

accept reads
accept writes

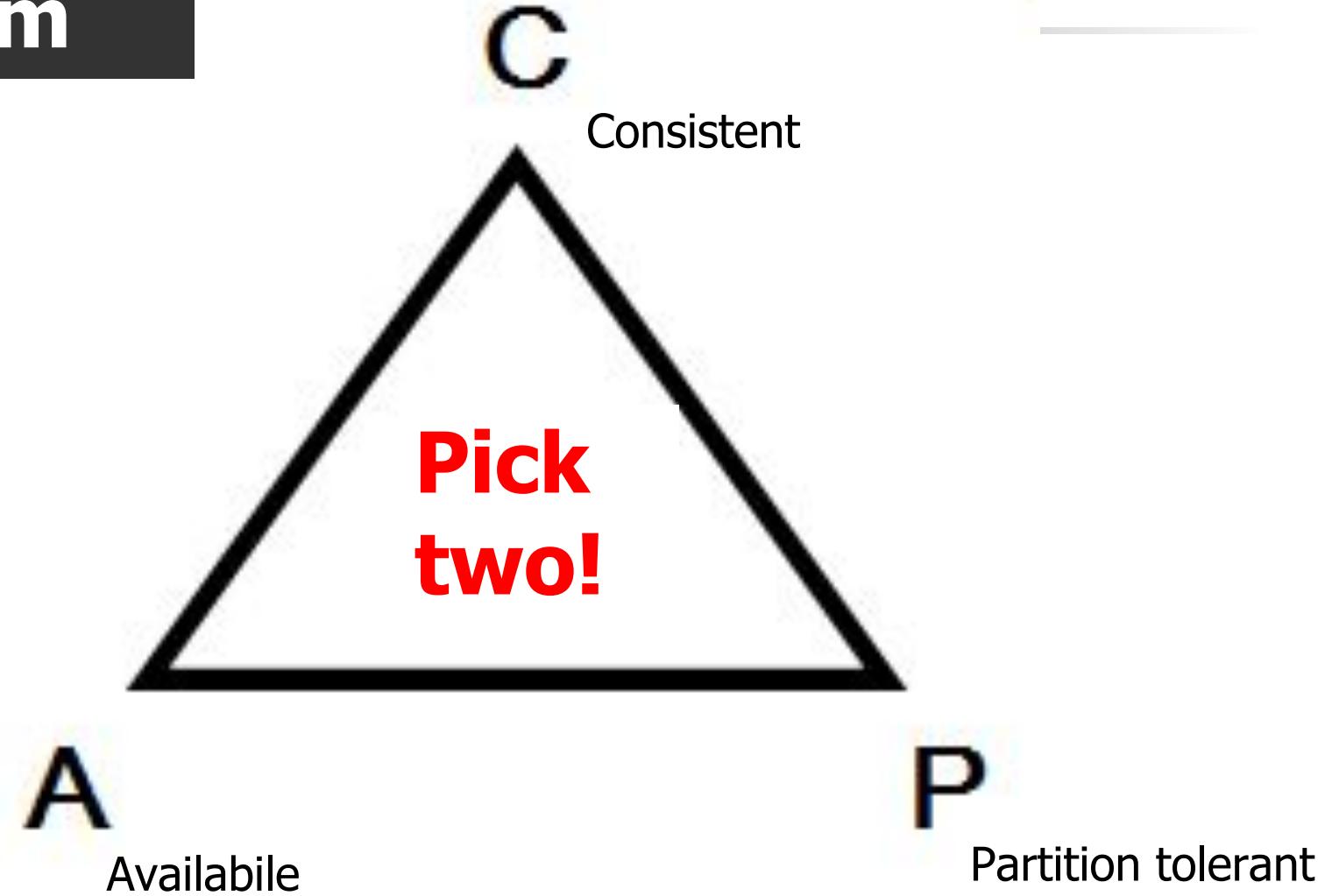
reject reads
reject writes

A+PT (-C)

accept reads + writes
'inconsistent' results

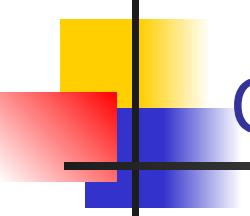
accept reads + writes
'inconsistent' results

CAP Theorem

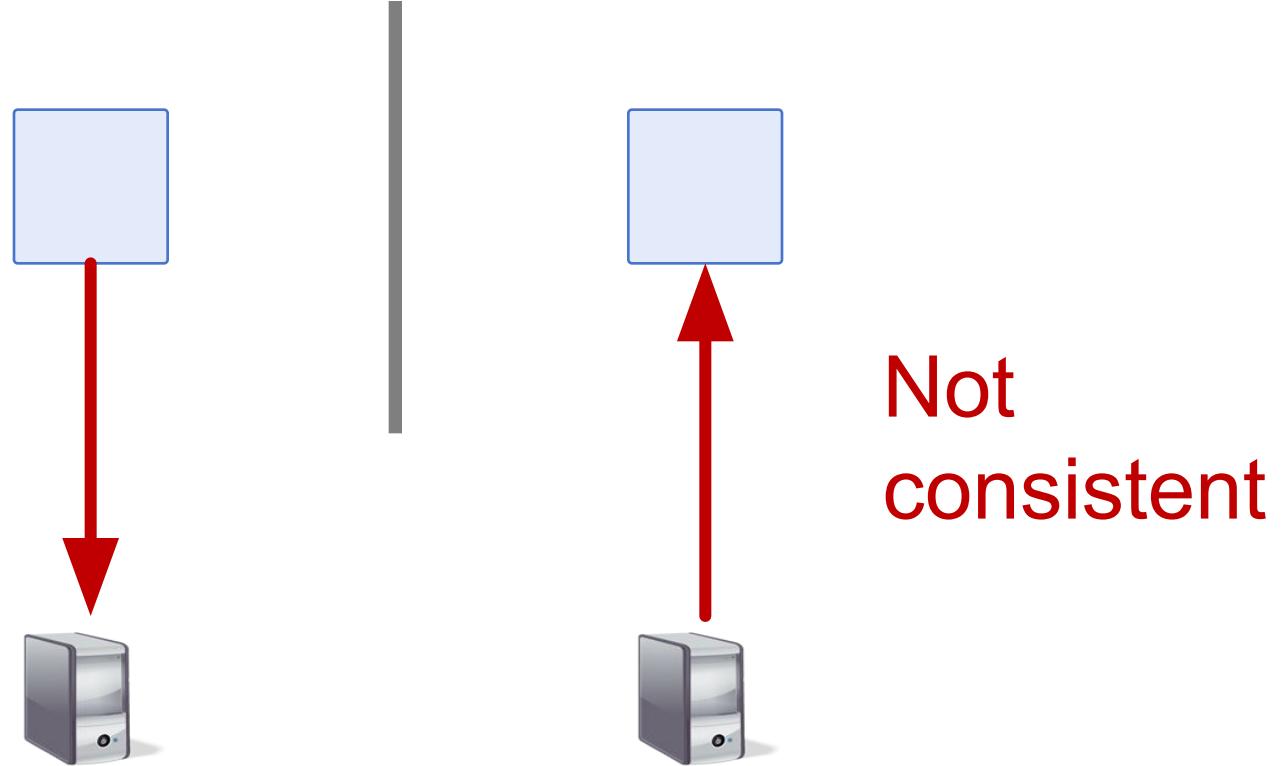


CAP Twelve Years After – How the “Rules” Have Changed, IEEE Spectrum 2012 [[link](#)]

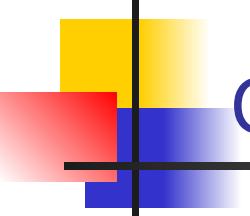
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33(2): 51-59 (2002)



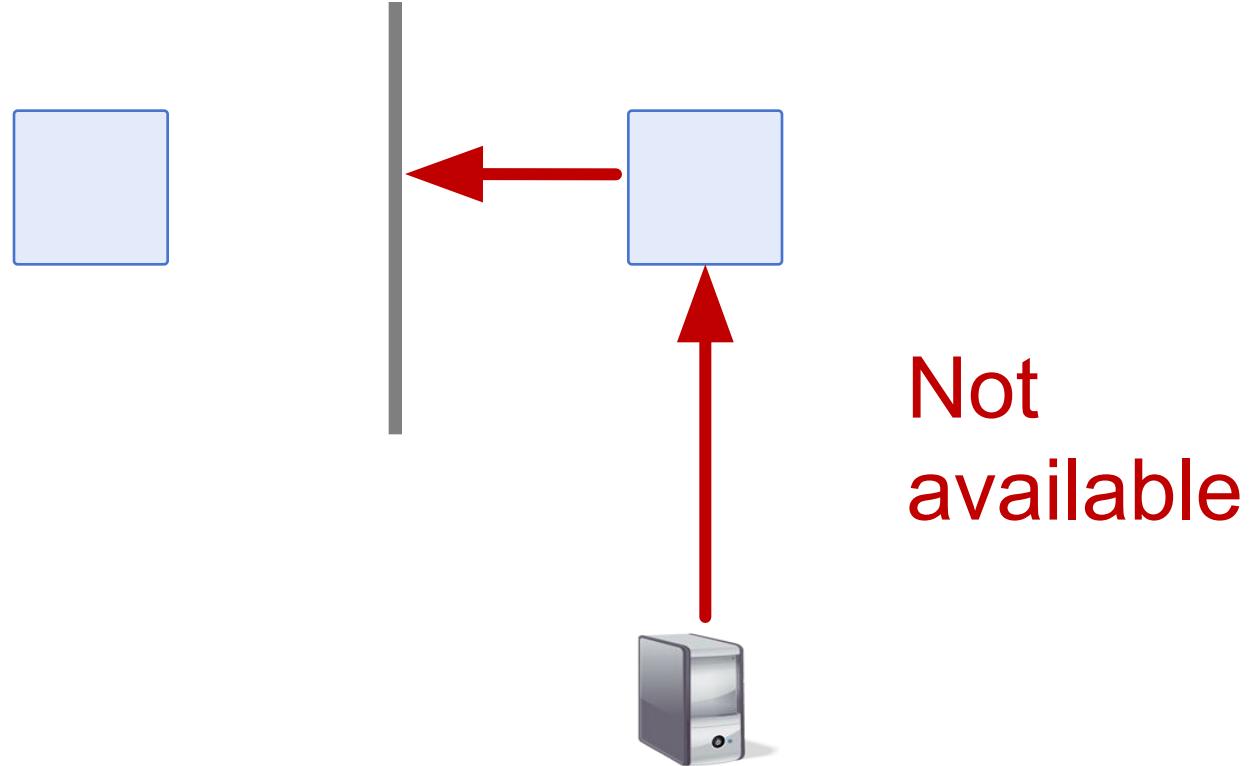
CAP Theorem: Proof



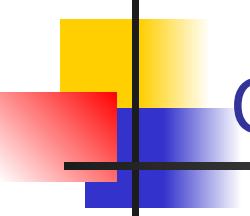
Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.



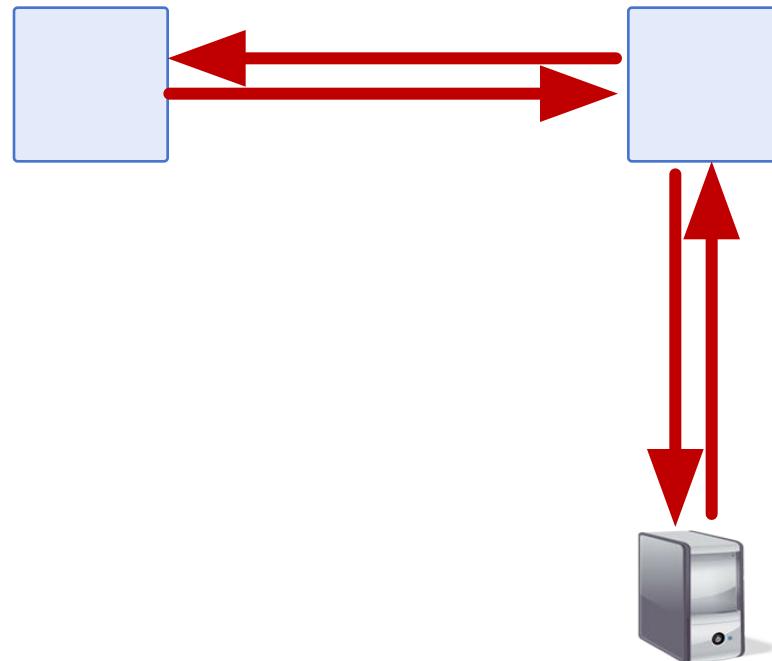
CAP Theorem: Proof



Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

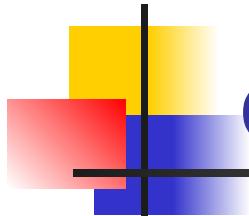


CAP Theorem: Proof

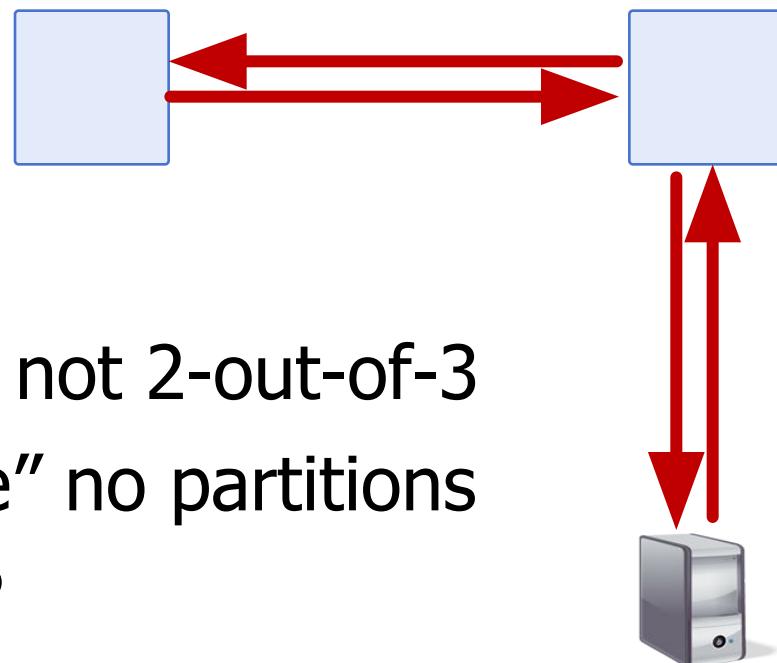


Not
partition
tolerant

Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

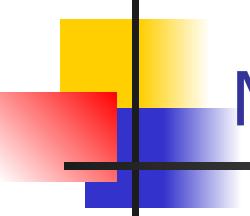


CAP Theorem: AP or CP



- Criticism: It's not 2-out-of-3
- Can't "choose" no partitions
- So: AP or CP

Not
partition
tolerant

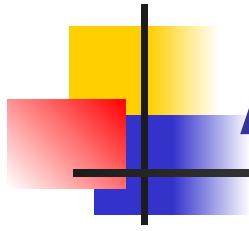


More tradeoffs: Latency vs. Consistency

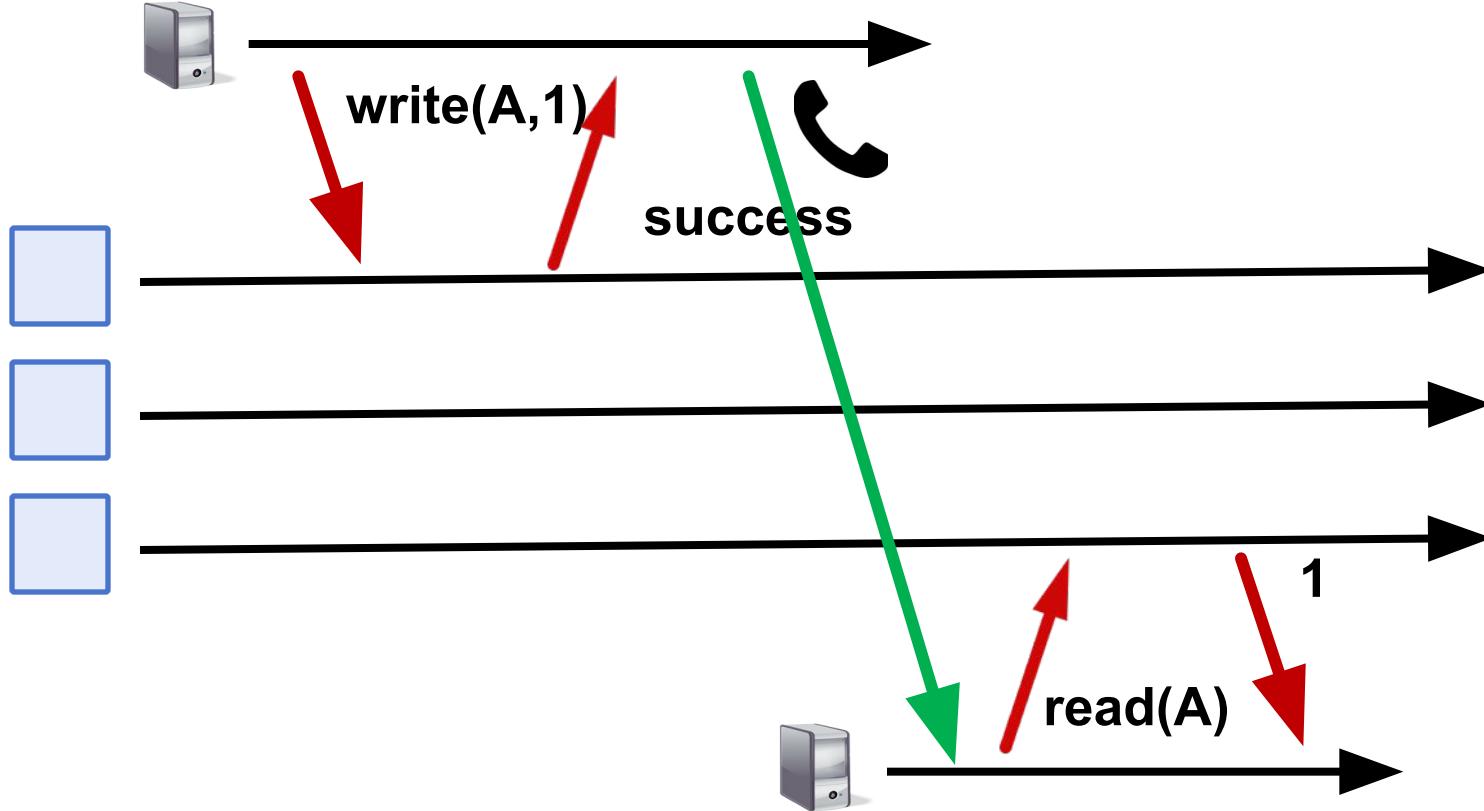
- Low-latency: Speak to fewer than quorum of nodes?
 - Primary-backup: write N, read 1
 - General: $|W| + |R| > N$
- L and C are fundamentally at odds
 - “C” = linearizability, sequential

- If there is a partition (**P**):
 - How does system tradeoff A and C?
- Else (no partition)
 - How does system tradeoff L and C?
- Is there a useful system that switches?
 - Dynamo: PA/EL
 - “ACID” dbs: PC/EC

<http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>

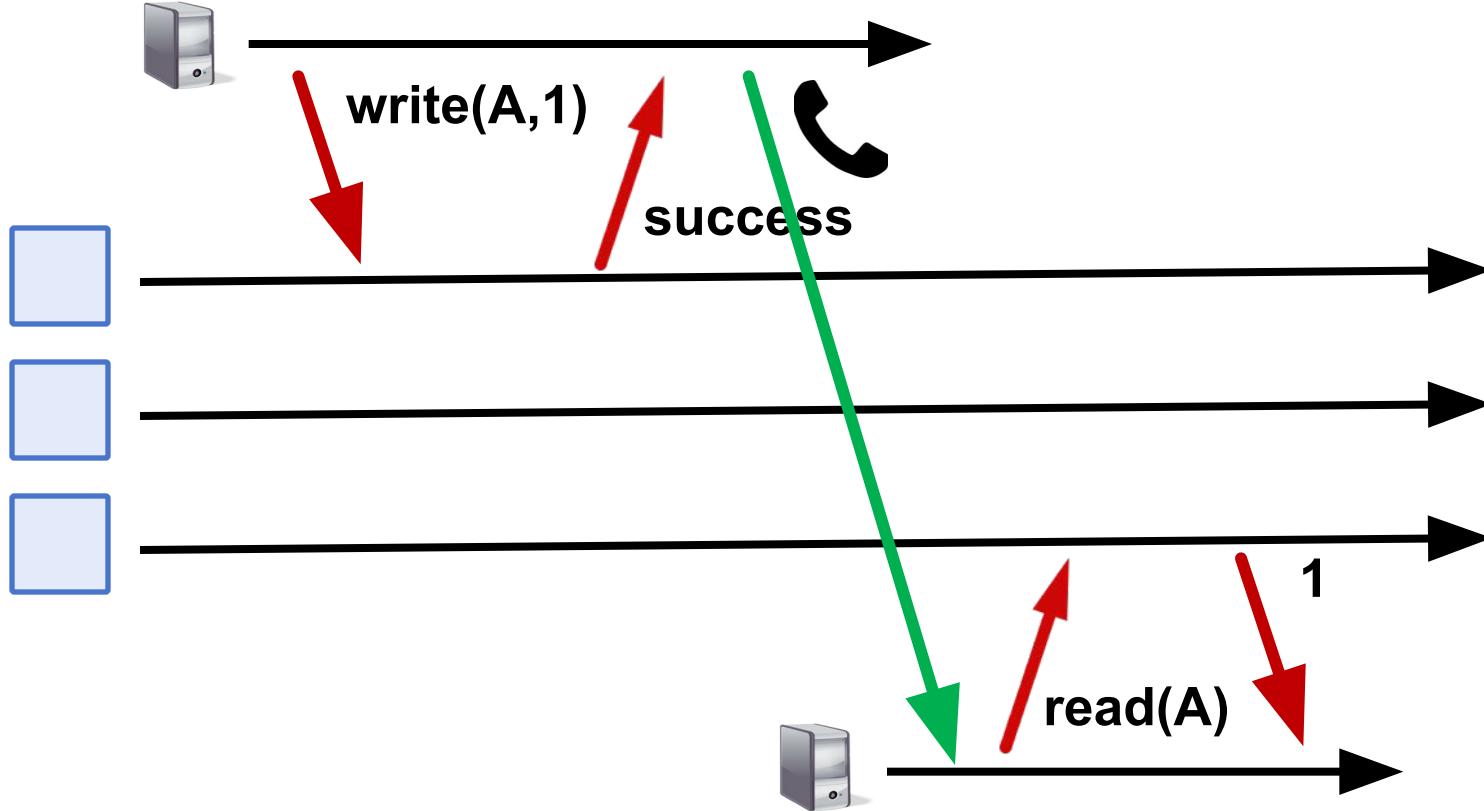


A brief review / discussion

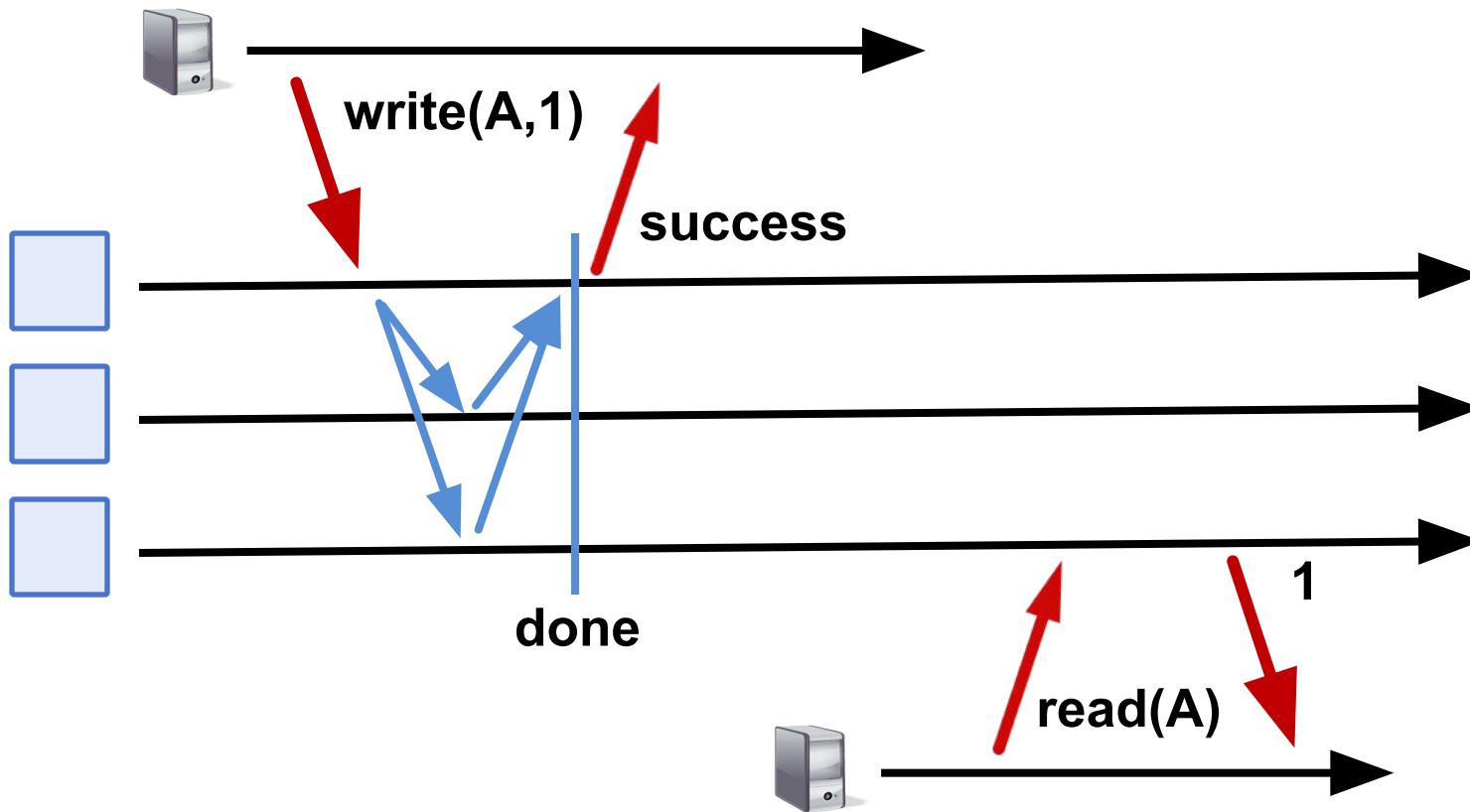


Phone call: *happens-before* relationship,
through “out-of-band” communication

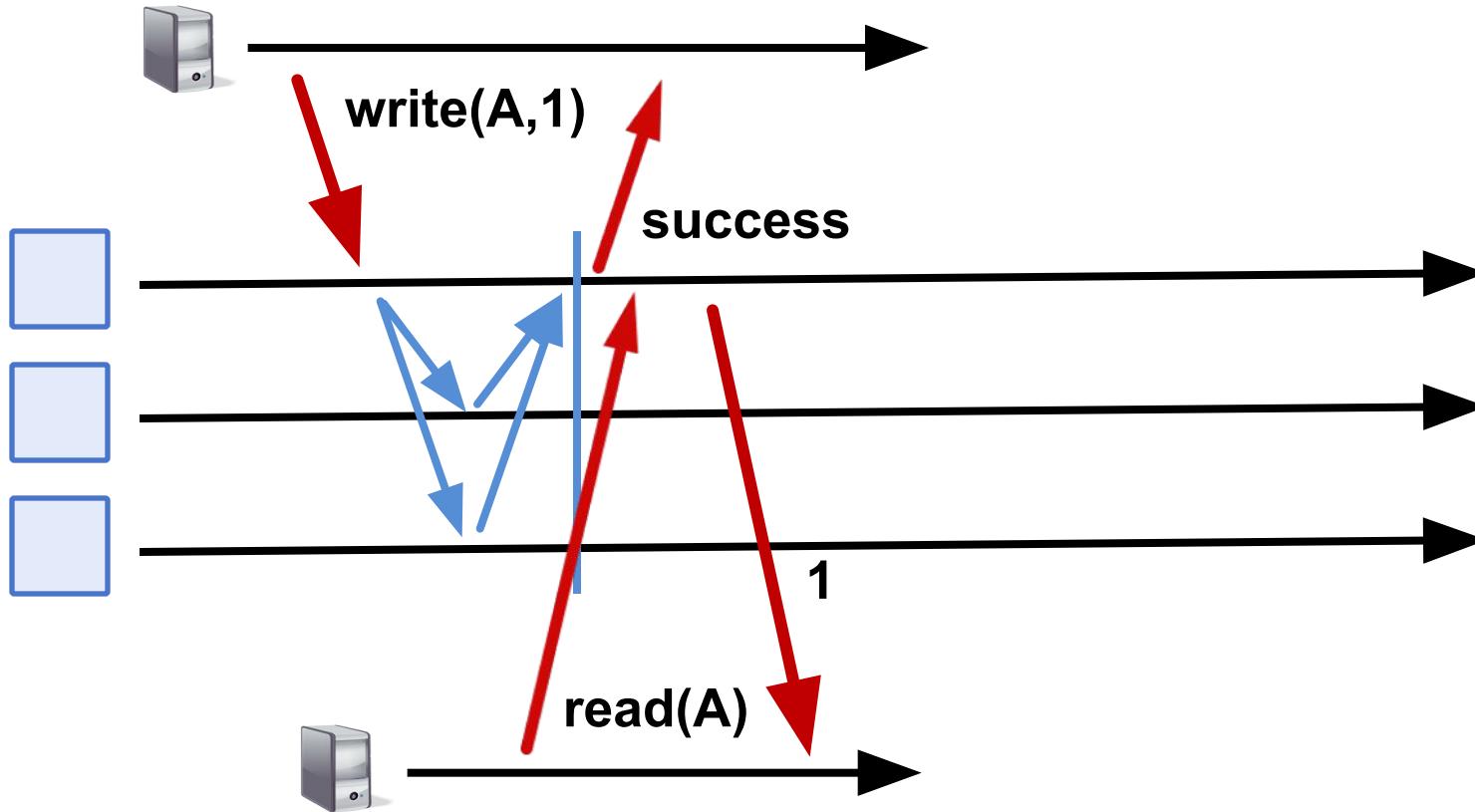
Which consistency model would guarantee the expected $r(A)1$ outcome?



One cool trick: Delay responding to writes/ops until properly performed on all replicas



- Isn't sufficient to return value of third node:
It doesn't know precisely when op is "globally" committed
- Instead: Need to actually *order* read operation

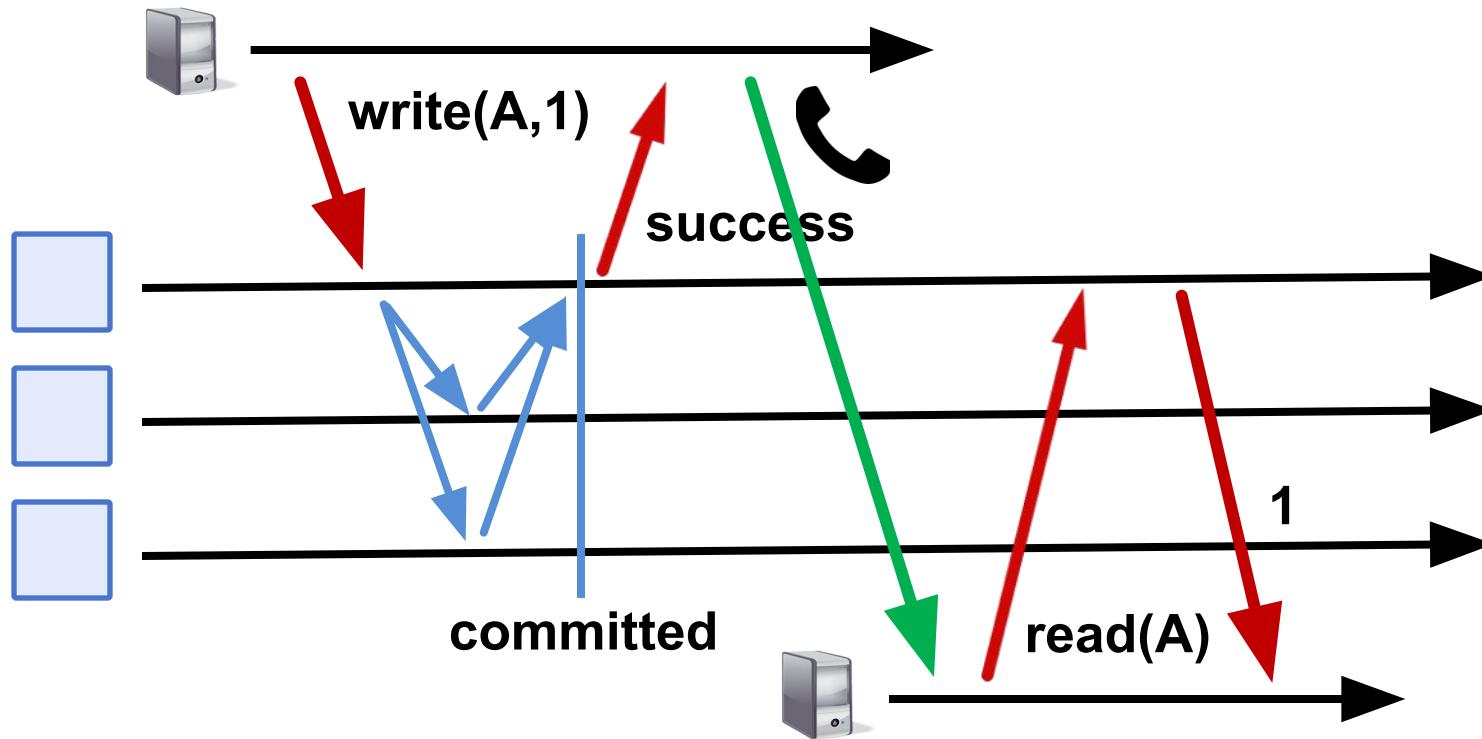


Potential solutions to order all operations via: (1) leader/primary, (2) consensus, or (3) physical time (a la Spanner)

Strong consistency = linearizability

- Linearizability (Herlihy and Wang 1991)
 1. All servers execute all ops in *some* identical sequential order
 2. Global ordering preserves each client's own local ordering
 3. Global ordering preserves real-time guarantee
 - All ops receive global time-stamp using a sync'd clock
 - If $ts_{op1}(x) < ts_{op2}(y)$, OP1(x) precedes OP2(y) in sequence
- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.

Intuition: Real-time ordering

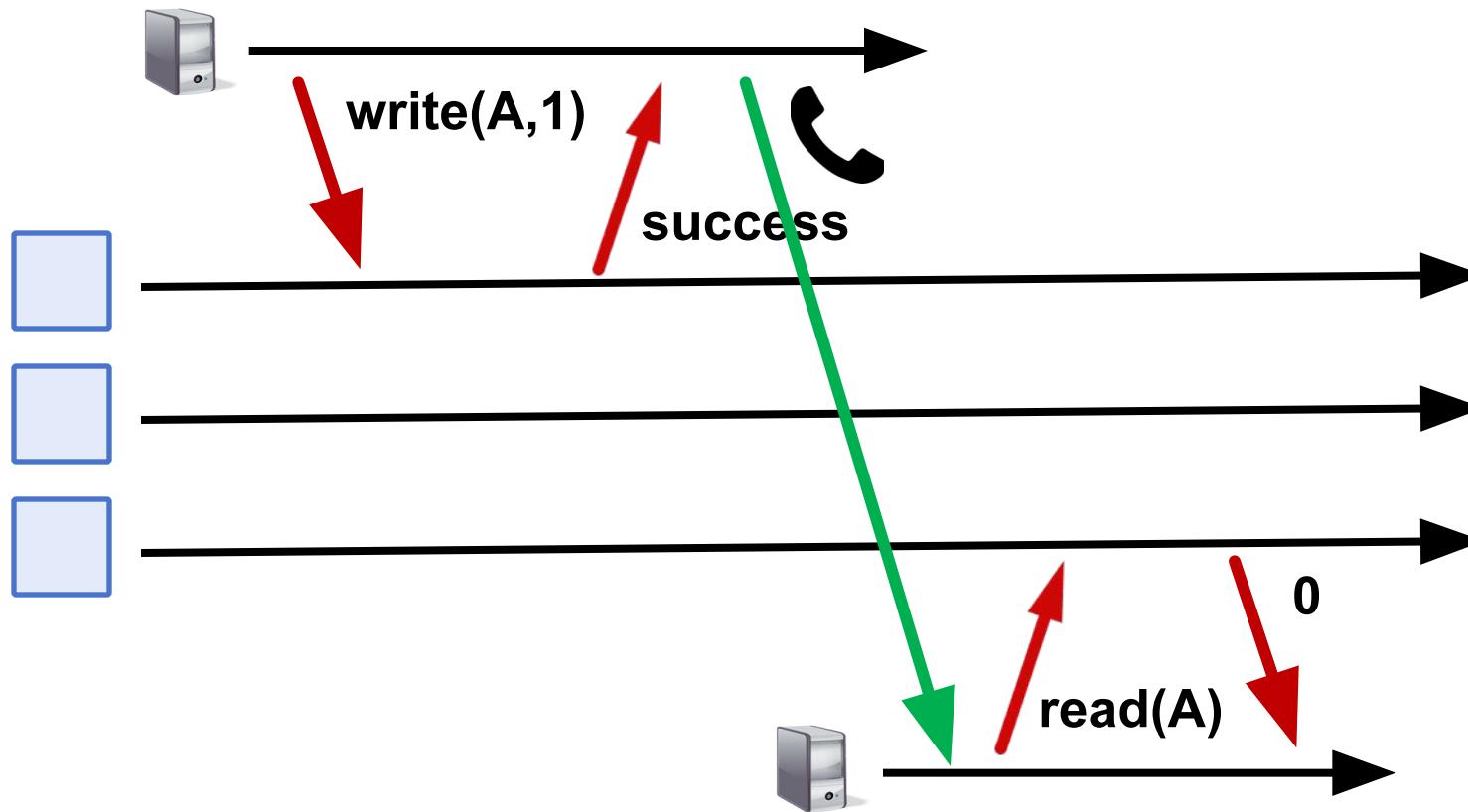


- Once write completes, all later reads (by wall-clock start time) should return value of that write or value of later write.
- Once read returns particular value, all later reads should return that value or value of later write.

Weaker: Sequential consistency

- Sequential = Linearizability – real-time ordering
 1. All servers execute all ops in *some* identical sequential order
 2. Global ordering preserves each client's own local ordering
- With concurrent ops, “reordering” of ops (w.r.t. real-time ordering) acceptable, but all servers must see same order
 - e.g., linearizability cares about **time**
sequential consistency cares about **program order**

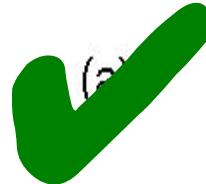
Sequential Consistency



In this example, system orders **read(A)** before **write(A, 1)**
[a sequential concurrency contract would allow this]

Valid Sequential Consistency?

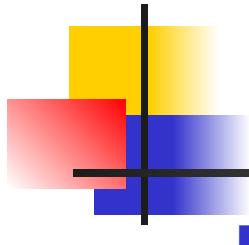
P1:	W(x)a	
P2:	W(x)b	
P3:	R(x)b	R(x)a
P4:	R(x)b	R(x)a



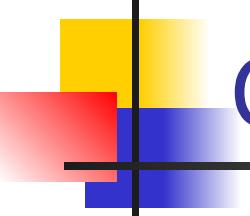
P1:	W(x)a	
P2:	W(x)b	
P3:	R(x)b	R(x)a
P4:	R(x)a	R(x)b



- Why? Because P3 and P4 don't agree on order of ops.
Doesn't matter when events took place on diff machine,
as long as proc's AGREE on order.
- What if P1 did both W(x)a and W(x)b?
 - Neither valid, as (a) doesn't preserve local ordering

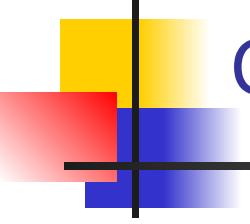


We stopped here



Causal consistency

- Remember causality notion from Lamport (logical) clocks?
 - That's what causal consistency enforces
- Causal consistency: Any execution is the same as if all causally-related read/write ops were executed in an order that reflects their causality
 - All concurrent ops may be seen in different orders
- Therefore:
 - Reads are fresh only w.r.t. the writes they are causally dependent on
 - causally-related writes are ordered by all replicas in the same way, BUT
 - concurrent writes may be committed in different orders by different replicas, and hence read in different orders by different applications

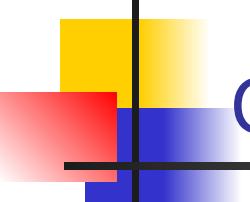


Causal Consistency (1)

“Causally related” relationship (notation ‘ \square ’):

- A read is causally related to the write that provided the data for the read.
- A write is causally related to a read that happened before this write *in the same process*.
- Relationship is transitive: If $\text{write1} \rightarrow \text{read}$, and $\text{read} \rightarrow \text{write2}$, then $\text{write1} \rightarrow \text{write2}$.

Def: concurrent ops == NOT causally related



Causal Consistency (Example)

P1: $W(x)a$

$W(x)c$

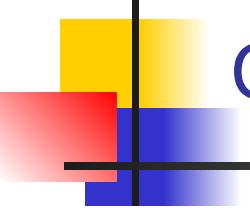
P2: $R(x)a$ $W(x)b$

P3: $R(x)a$ $R(x)c$ $R(x)b$

P4: $R(x)a$ $R(x)b$ $R(x)c$

Note: $W_1(x)a \rightarrow W_2(x)b$, but not $W_2(x)b \rightarrow W_1(x)c$

- Is this sequence is allowed with a causally-consistent store?
- Is this sequence allowed with sequentially consistent store?



Causal Consistency: (More Examples)

P1: $W(x)a$

P2: $R(x)a$ $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

SecC: no

CauC: no

P1: $W(x)a$

P2: $W(x)b$

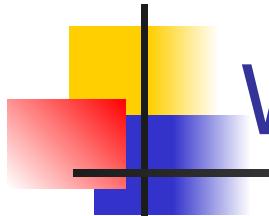
P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

SecC: no

CauC: yes

- Which sequence is allowed with a causally-consistent store?
- Which sequence allowed with sequentially consistent store?



Why Causal Consistency?

- Causal consistency is strictly weaker than sequential consistency (and can give weird results (as you've seen))
 - strict < linearizable < sequential < causal
- BUT causal also offers more possibilities for concurrency:
 - Concurrent operations (which are not causally-dependent) can be executed in different orders by different replicas
 - In contrast, with sequential consistency, you need to enforce a global ordering of all operations
- Hence, one can get better performance than with sequential

From what I know, not very popular in industry So, we're not gonna focus on it any more

Consistency protocols

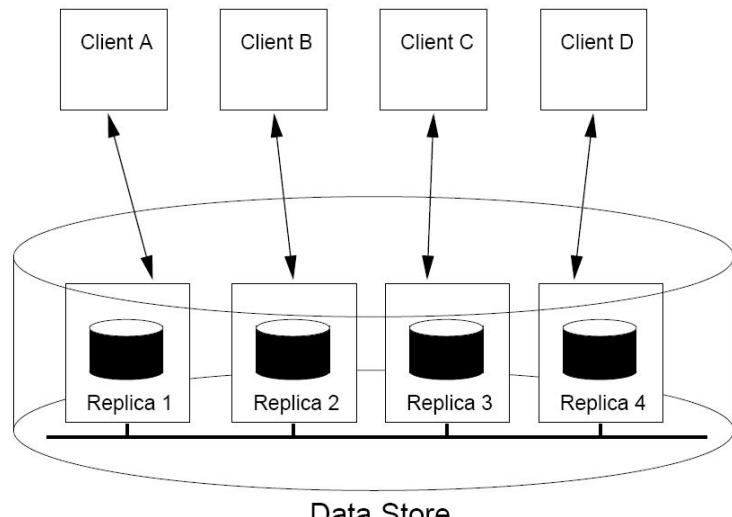
Question: How does one design a protocols to implement the desired consistency model?

Sequential consistency

- Group with total ordering
- Primary / backup
- Chain replication
- Quorum protocols

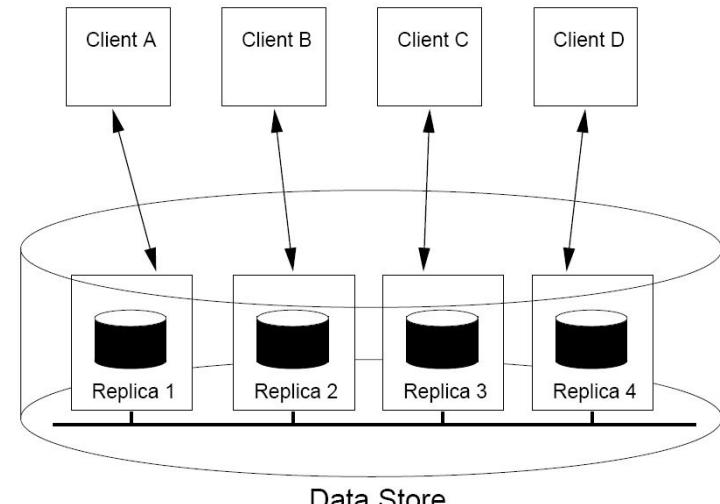
Causal consistency

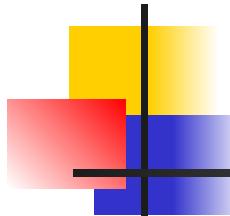
- Group with causal ordering



Many Other Consistency Models Exist ...

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Continuous consistency
 - Limit the deviation between replicas
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering **for each client independently**





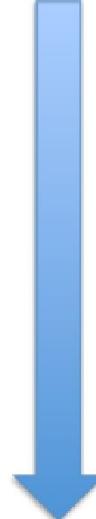
Many Other Consistency Models Exist

- Other standard consistency models
 - ... take a look at next slides here
 - read Tanenbaum 7.3



Consistency models

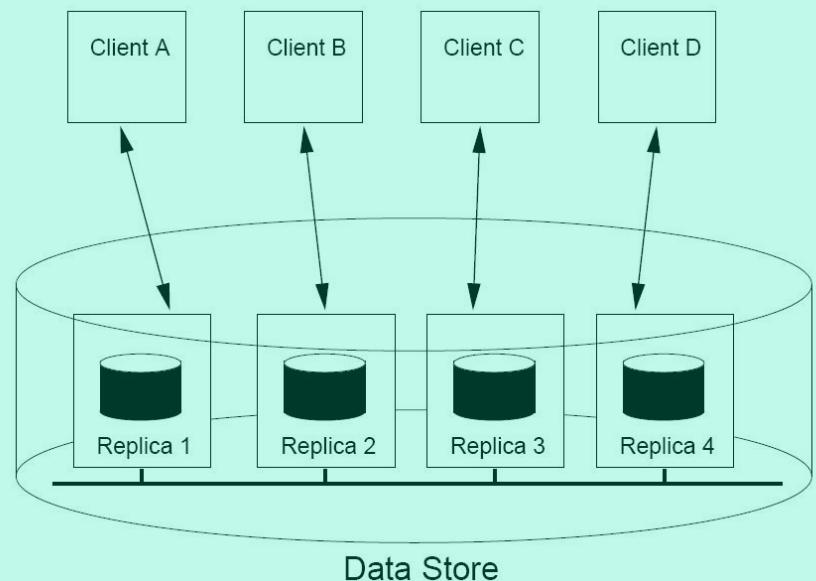
- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency
 - Strict
 - Strong (Linearizability)
 - Sequential
 - Causal
 - Eventual

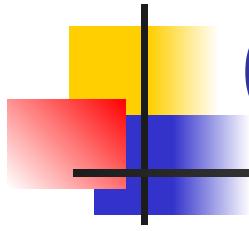


Weaker
Consistency
Models

Consistency models

- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Continuous consistency
 - Limit the deviation between replicas
 - Eventual consistency
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently

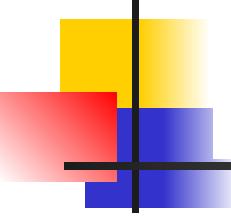




Client-centric Consistency Models

Goal: Avoid system-wide consistency,
by concentrating on what each client
independently wants

(instead of maintaining a global view)



Example: Consistency for Mobile Users

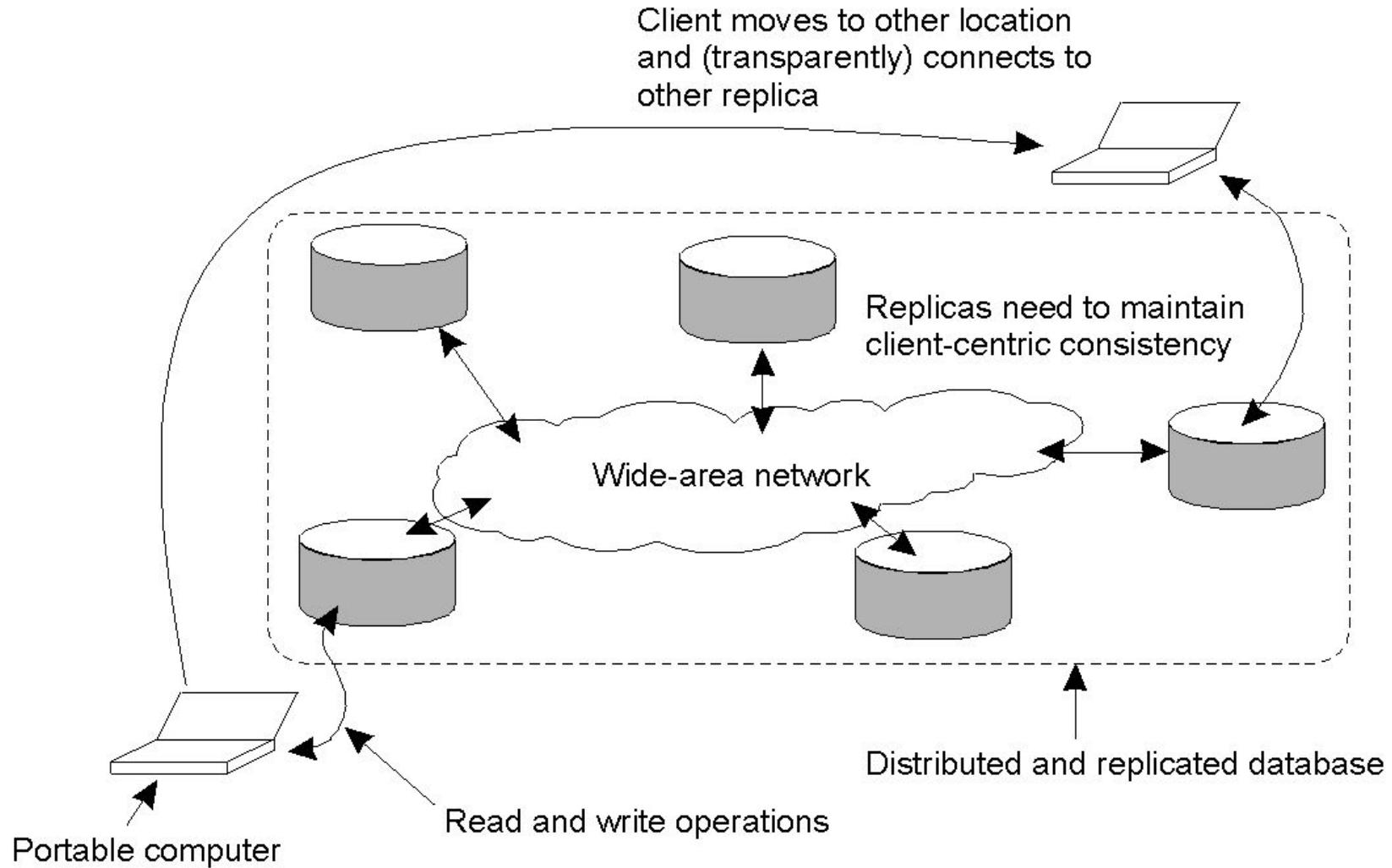
Example: Distributed database to which a user has access through her notebook.

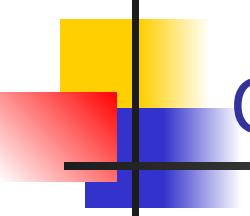
- Notebook acts as a front end to the database.
- At location *A* user accesses the database with reads/updates
- At location *B* user continues work, but unless it accesses the same server as the one at location *A*, she may detect inconsistencies:
 - updates at *A* may not have yet been propagated to *B*
 - user may be reading newer entries than the ones available at *A*:
 - user updates at *B* may eventually conflict with those at *A*

Note: The only thing the user really needs is that the entries updated and/or read at *A*, are available at *B* the way she left them in *A*.

- **Idea:** the database will appear to be consistent **to the user**

Example - Consistency for Mobile Users





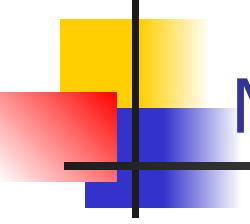
Client-centric Consistency

Idea: Guarantee a degree of data access consistency for a single client/process point of view.

Notations:

- $x_i[t]$ □ Value of data item x at time t at local replica L_i
- $WS(x_i [t])$ □ working set (all write operations) at L_i up to time t on data item x
- $WS(x_i [t]; x_j [t])$ □ indicates that it is known that $WS(x_i [t])$ is included in $WS(x_j [t])$

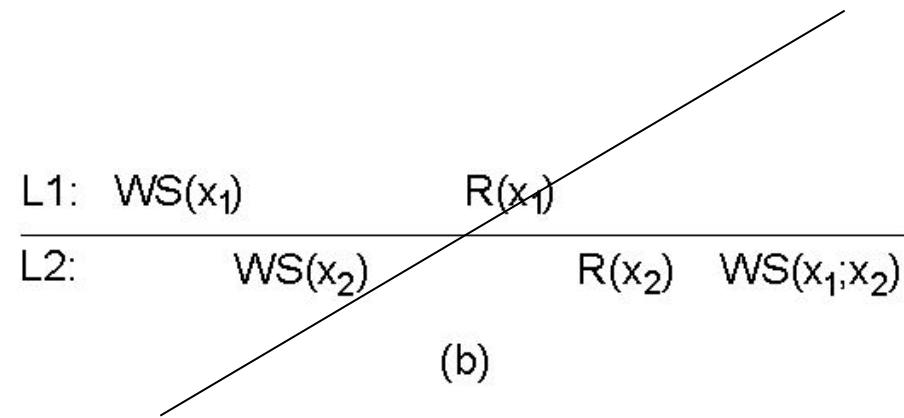
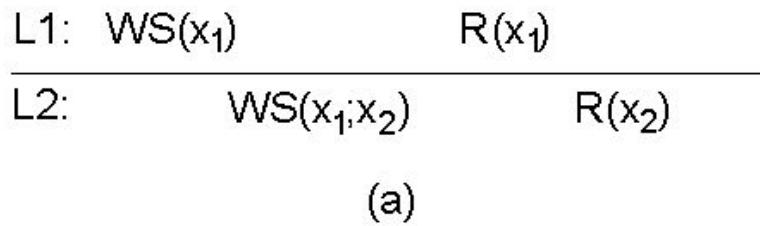
Shorthand: we do not use $[t]$ (it's implied)

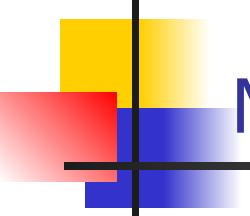


Monotonic-Read Consistency

Intuition: Client “sees” the same or newer version of data.

Definition: *If a process reads the value of a data item x , any successive read operation on x by that process will always return that same or a more recent value.*





Monotonic reads – Examples

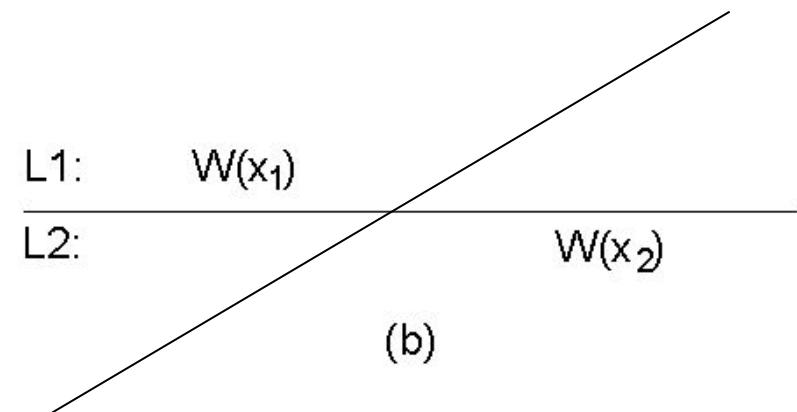
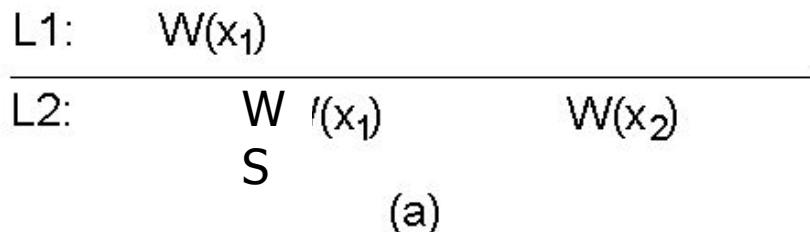
- A personalized news webpage
- Reading (not modifying) incoming e-mail while you are on the move.
 - Each time you connect to a different e-mail server, that server fetches (at least) all the updates from the server you previously visited.
- Reading personal calendar updates from different servers.
 - Monotonic Reads guarantees that the user sees always more recent updates, no matter from which server the reading takes place.

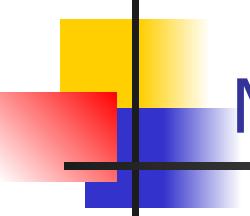
(All the above assume read-only)

Monotonic-Write Consistency

Intuition: A write happens on a replica only if it's brought up to date with preceding write operations on same data (but possibly at different replicas)

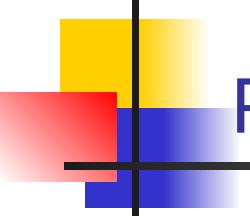
Definition: A write operation by a process on a data item x is completed (on all replicas) before any successive write operation on x by the same process.





Monotonic writes – Examples

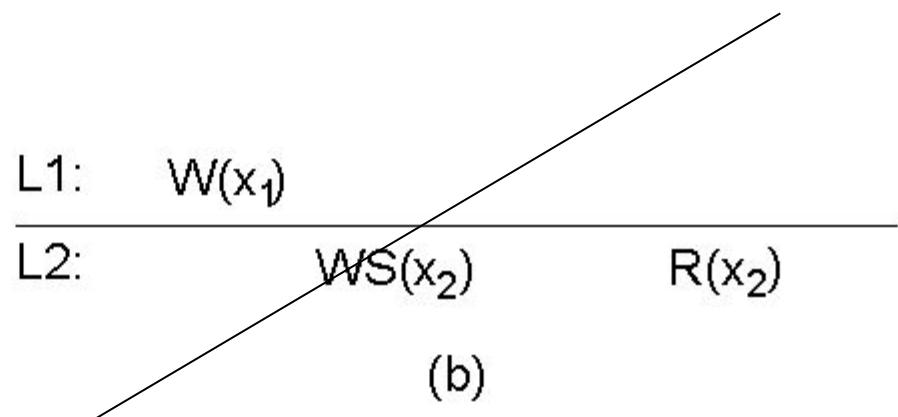
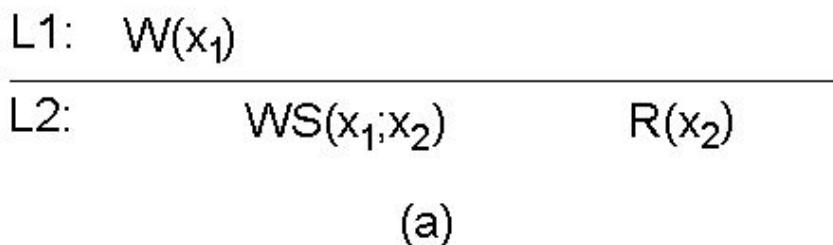
- Updating a program at server S_2 , and ensuring that all components on which compilation and linking depend, are also placed at S_2 .
- Maintaining versions of replicated files in the correct order everywhere (propagate the previous version to the server where the newest version is installed).

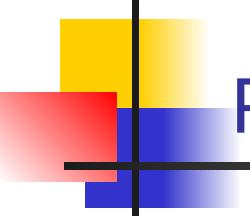


Read-Your-Writes Consistency

Intuition: All previous writes are always completed before any successive read

Definition: *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.*





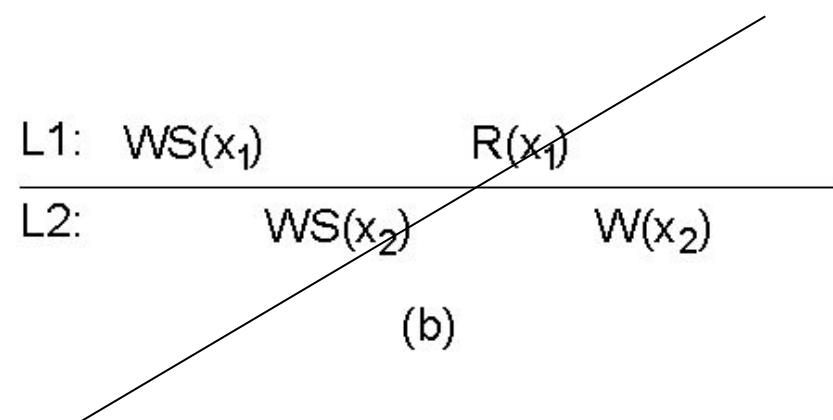
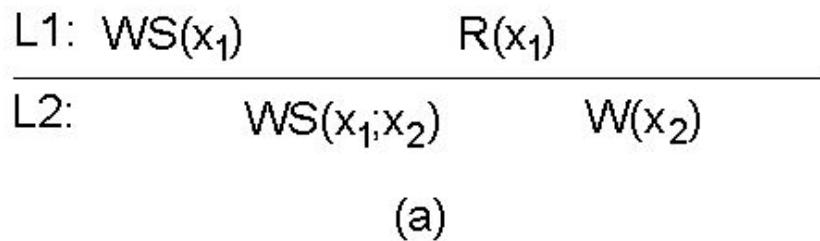
Read-Your-Writes - Examples

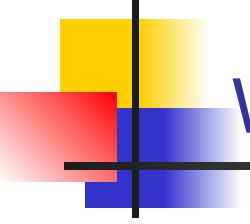
- Updating your Web page and guaranteeing that your Web browser shows the newest version instead of its cached copy.
- Password database

Writes-Follow-Reads Consistency

Intuition: Any successive write operation on x will be performed on a copy of x that is same or more recent than the last read.

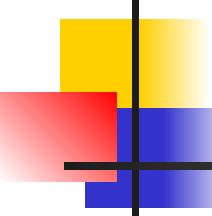
Definition: A write operation by a process on a data item x following a previous read operation on x by the same process, is guaranteed to take place on the same or a more recent value of x that was read.





Writes-Follow-Reads - Examples

- See reactions to posted articles only if you have the original posting
 - a read “pulls in” the corresponding write operation.



Quiz like questions

What is the crucial difference between data-centric and client-centric consistency models?

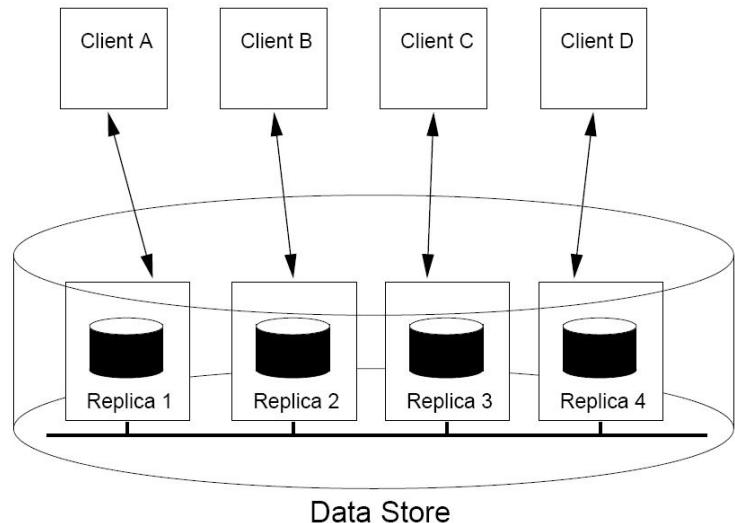
Where does the 'eventual consistency' model fit in this taxonomy: is it data-centric or client-centric?

Consider a system that combines read-your-writes consistency with writes-follow-reads consistency (that is, it provides both). Is this system also sequentially consistent?

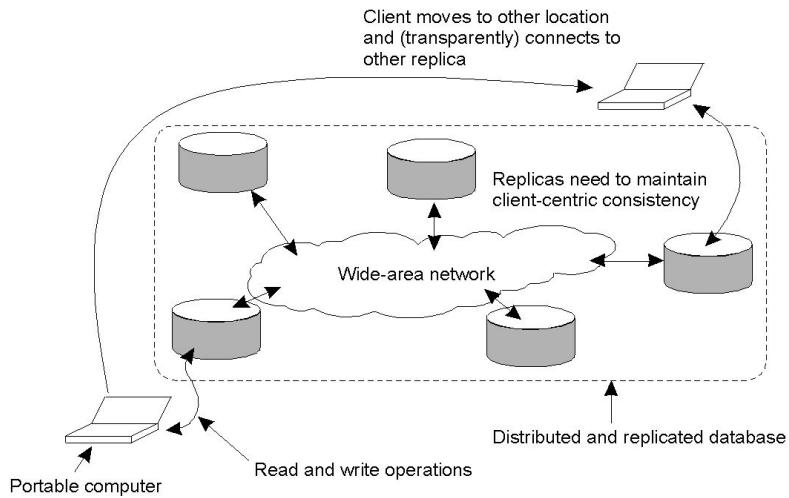
Question: How does one design the protocols to implement the desired consistency model?

- Data centric
 - Constraints on operation ordering at the data-store level
 - Sequential consistency.
 - Continuous consistency: limit the deviation between replicas

- Client centric
 - Constraints on operation ordering for each client independently



Reminder: Monotonic-Read



Definition: *If a process reads the value of a data item x , any successive read operation on x by that process will always return that same or a more recent value.*

Intuition: Client “sees” the same or a newer version of the data.

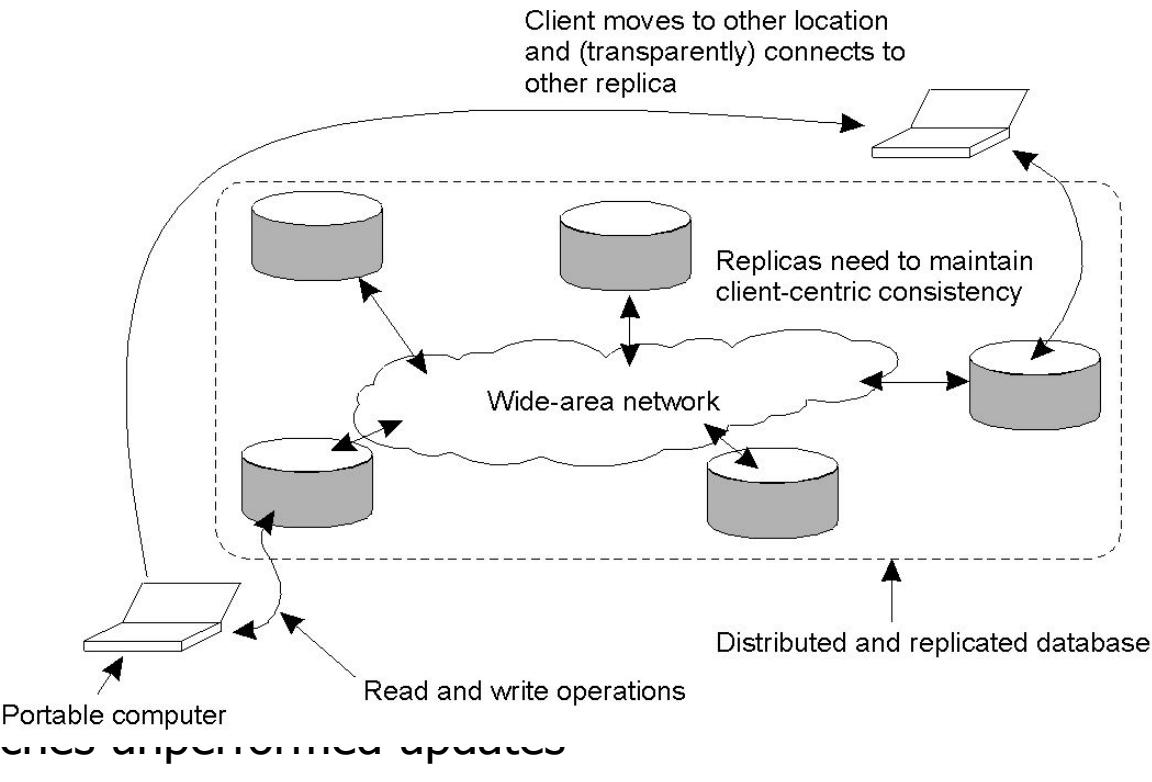
How would you implement this?

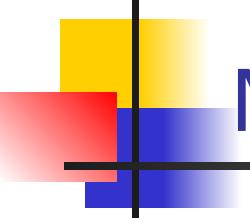
Implementation of monotonic-read consistency

Key intuition: client carries state to identify the last operation(s)

Protocol sketch:

- Globally unique ID
- The client keeps track of:
 - *ReadSet*: the writes performed far on various objects.
- When a client launches a read:
 - Client sends the *RID* and *ReadSet*.
 - The server checks if the read is performed.
 - [If necessary] Fetches unperformed updates.
 - Executes the read operation.





More quiz-like questions

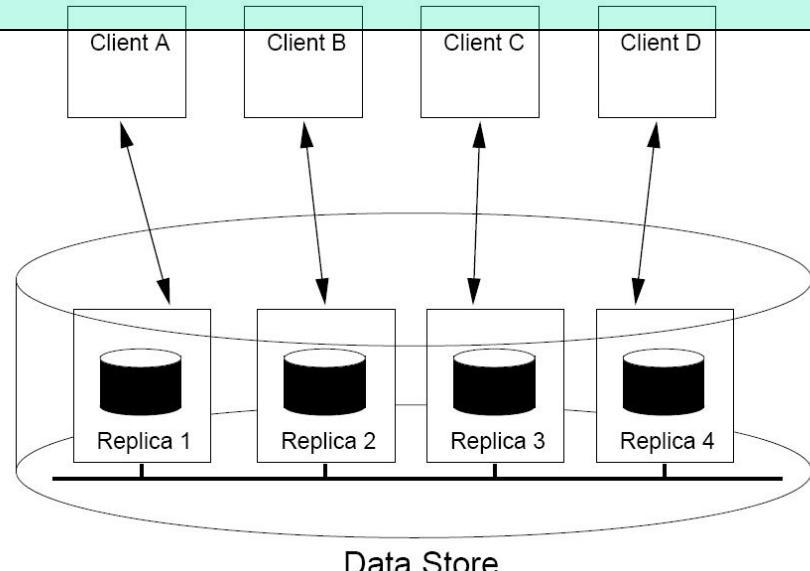
Sketch a protocol design for the following consistency model

- Monotonic-writes
- Read-your-writes
- Writes-follow-reads

Write the pseudo-code

Consistency models

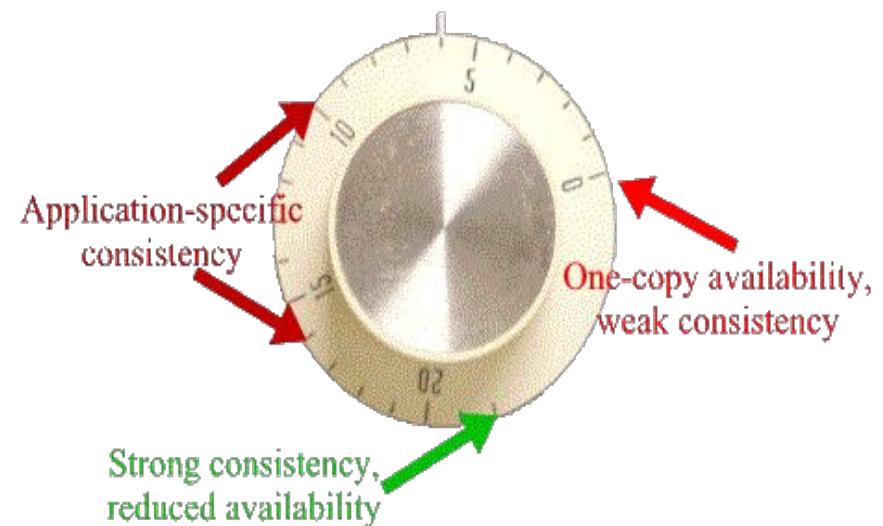
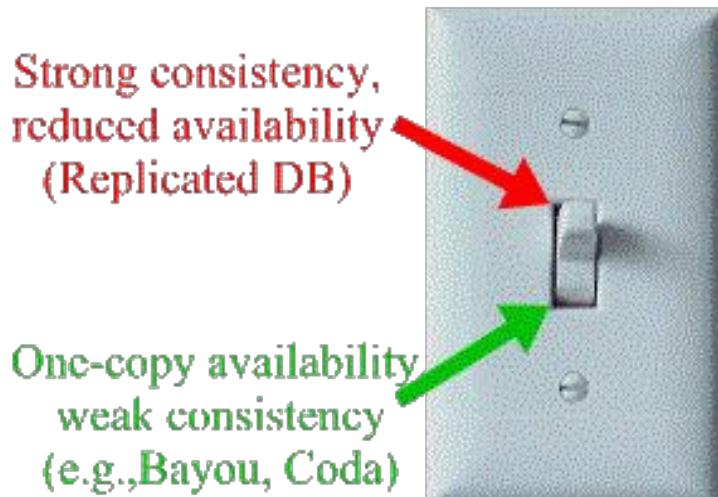
- **Data centric:** Assume a global, data-store view (i.e., across all clients)
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
 - Eventual consistency
 - Continuous consistency
 - Limit the deviation between replicas
- **Client centric**
 - Assume client-independent views of the datastore
 - Constraints on operation ordering for each client independently

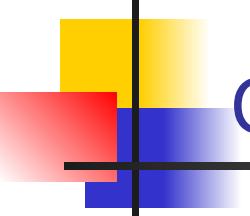


Continuous Consistency

- Obs1: We can talk about a **degree of consistency**
- Goal: Limit the **deviation** between replicas

Metaphor: Knob vs. switch





Continuous Consistency

Obs1: We can talk about a degree of consistency

- Goal: Limit the **deviation** between replicas

Obs2: Multiple metrics to measure deviation are possible

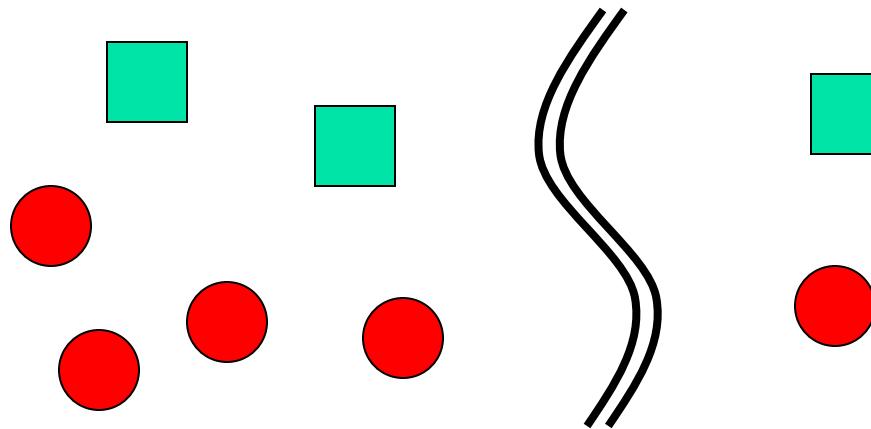
- replicas may differ in their **numerical value**
- replicas may differ in their relative **staleness**
- replicas may differ with respect to (number and order) of **performed update operations**

conit: consistency unit specifies the data unit over which consistency is to be enforced.

Limitations of Consistency Mechanisms Based on Request Ordering

 : Replicas

 : Clients



Option 1:

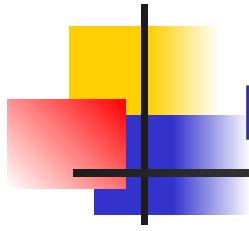
accept reads
reject writes

accept reads
reject writes

Option 2:

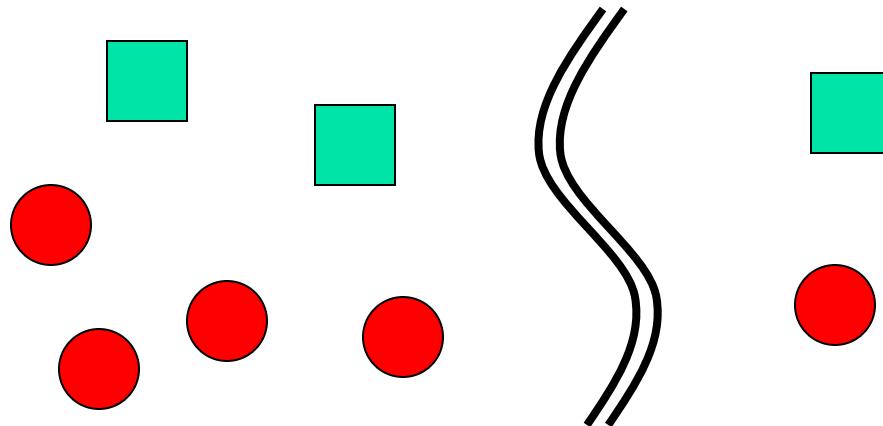
accept reads
accept writes

reject reads
reject writes



Effects of Continuous Consistency

Policy: each replica can buffer up to 5 writes

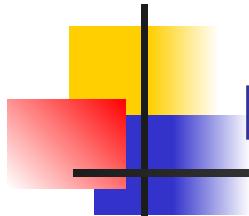


Option 1: **accept reads**
 reject writes

accept reads
reject writes

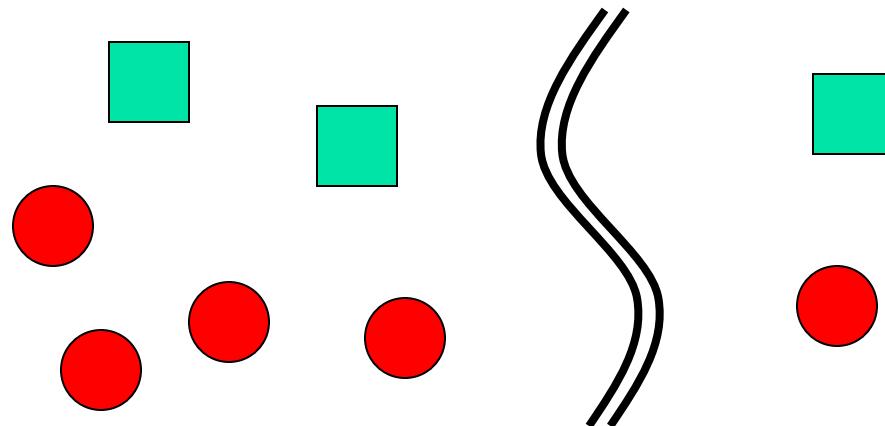
New Option 1: **accept reads**
 accept first 10 writes

accept reads
accept first 5 writes



Effects of Continuous Consistency

Policy: each replica can buffer up to 5 writes



Option 2:
accept reads
accept writes

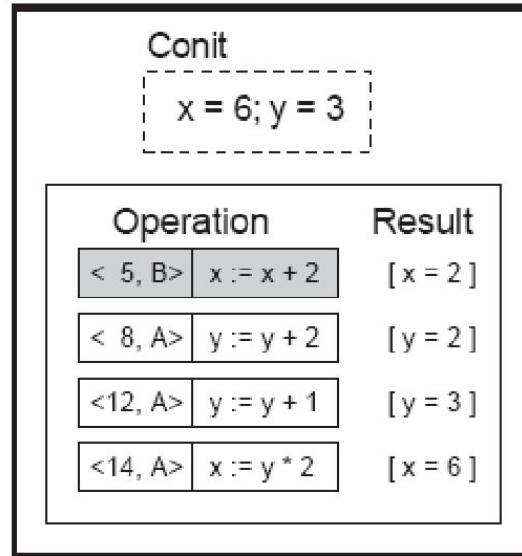
reject reads
reject writes

New Option 2:
accept reads
accept writes

accept first few reads
accept first 5 writes

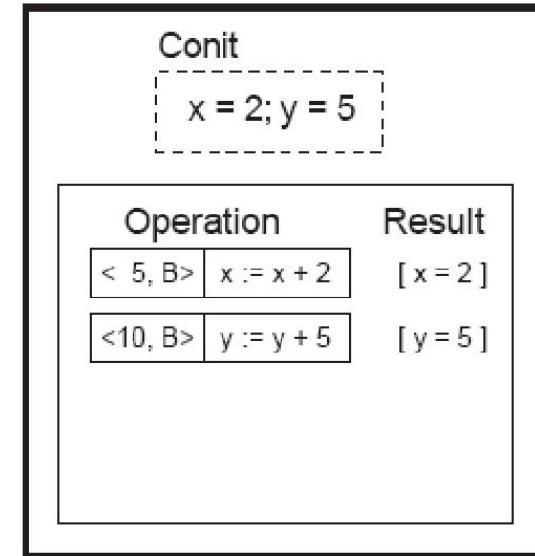
Example

Replica A



Vector clock A = (15, 5)
 Order deviation = 3
 Numerical deviation = (1, 5)

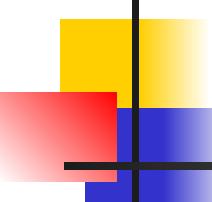
Replica B



Vector clock B = (0, 11)
 Order deviation = 2
 Numerical deviation = (3, 6)

Conit: contains the variables x and y :

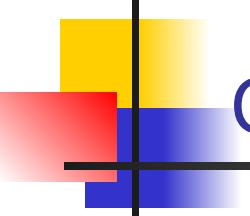
- Each replica maintains a **vector clock**
- B sends A operation [$<5,B>$: $x := x + 2$]; A has made this operation **permanent** (cannot be rolled back)
- A has three **pending** operations \square order deviation = 3



Consistency protocols

Question: How does one design the protocols to implement the desired consistency model?

- Data centric
 - Constraints on operation ordering at the data-store level
 - Sequential consistency.
 - **Continuous consistency: limit the deviation between replicas**



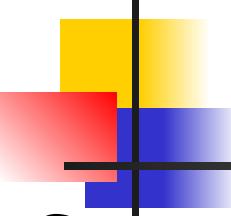
Continuous Consistency

Obs1: We can talk about a degree of consistency

- Goal: Limit the **deviation** between replicas

Obs2: Multiple metrics to measure deviation

- replicas may differ in their numerical value
- replicas may differ in their relative **staleness**
- replicas may differ with respect to (number and order) of performed update operations



Continuous Consistency: Bounding Numerical Errors (I)

Setup: consider a conit (data item) x and let $\text{weight}(W(x))$ denote the numerical change in its value after a write W .

- [for simplicity] Assume that for all $W(x)$, $\text{weight}(W) > 0$
- W is initially forwarded to one of the N replicas: the $\text{origin}(W)$.
- $TW[i, j]$ are the writes executed by server S_j that originated from S_i
$$TW[i, j] = \sum \{\text{weight}(W) / \text{origin}(W) = S_j \text{ & } W \text{ in } \log(S_i)\}$$

Note: Actual value of x : v_i : value of x at replica i

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Continuous Consistency: Bounding Numerical Errors (II)

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k,k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i,k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i

Continuous Consistency: Bounding Numerical Errors (II)

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

Note: $0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Goal: need to ensure that $\sum_{\substack{k=1 \\ k \neq i}}^N (TW[k, k] - TW[i, k]) < d$

then, as an approximate bound: $TW[k, k] - TW[i, k] < \frac{d}{N-1}$

for all k

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

Note1: $0 \leq TW_k[i, j] \leq TW[i, j] \leq TW[j, j]$

Note2: $TW_k[i, k]$: view at node K about updates inserted at k and propagated to I

Reminder: $TW[i, j]$ writes executed by server S_i originated at S_j

Actual value of x :

$$v(t) = v_{init} + \sum_{k=1}^N TW[k, k]$$

v_i : value of x at replica i

$$v_i = v_{init} + \sum_{k=1}^N TW[i, k]$$

Goal: need to ensure that $v(t) - v_i < d$ for every S_i

Approach: Let every server S_k maintain a **view** $TW_k[i, j]$ for what S_k believes is the value of $TW[i, j]$.

- This information can be piggybacked when an update is propagated.

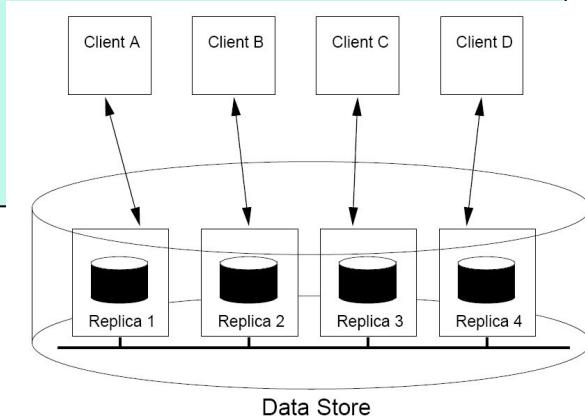
Solution: S_k sends operations from its log to S_i when it sees that $TW_k[i, k]$ is getting too far from $TW[k, k]$,

- in particular, when $TW[k, k] - TW_k[i, k] > d/(N - 1)$.

Note: Staleness can be done analogously, by essentially keeping track of what has been seen last from S_k

Consistency models: contracts between the data store and the clients

- Data centric: solutions at the data store level
 - Continuous consistency
 - limit the deviation between replicas, or
 - Eventual consistency
 - Models based on ordering of operations
 - Constraints on operation ordering at the data-store level
- Client centric
 - Assume client-independent views of the datastore
 - Constraints on operation ordering
 - for each client independently





Issue 1. Dealing with data changes

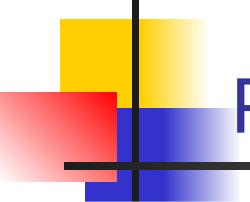
- **Consistency models**
 - What is the semantic the system implements
 - (luckily) applications do not always require strict consistency
- **Consistency protocols**
 - How to implement the semantic agreed upon?

Issue 2. Replica management

- How many replicas?
- Where to place them?
- When to get rid of them?

Issue 3. Redirection/Routing

- Which replica should clients use?



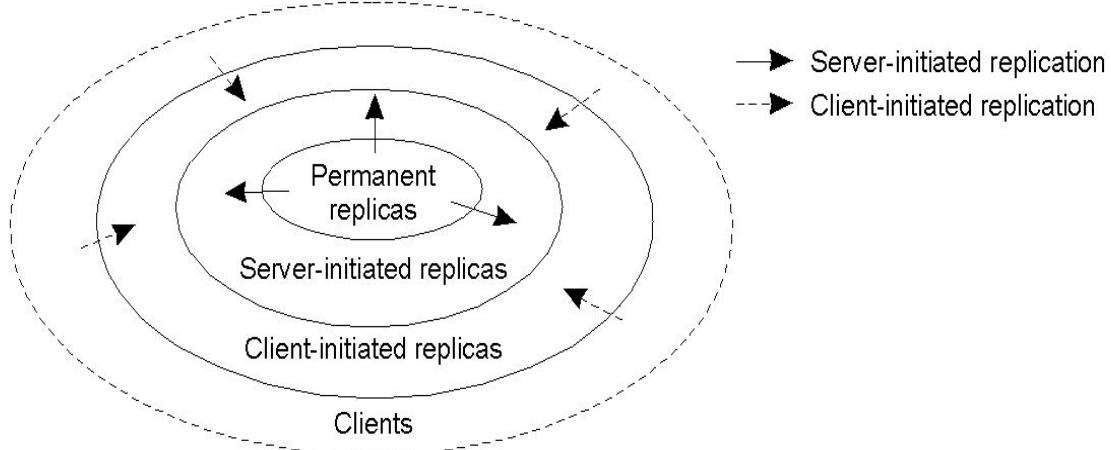
Replica server placement

Problem: Figure out what the best K places are out of N possible locations.

- Select best location out of $N - j$, $j:0..K-1$ for which the **average distance to clients is minimal**. Then choose the next best server.
 - (Note: The first chosen location minimizes the average distance to all clients.)
 - **Computationally expensive.**
- Select the k largest **autonomous system** and place a server at the best-connected host.
 - **Computationally expensive.**
- Position nodes in a d -dimensional geometric space, where distance reflects latency. Identify the K regions with highest density and place a server in every one.
 - **Computationally cheap.**

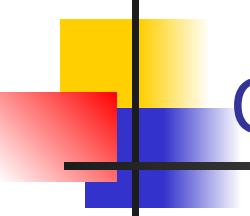
Content replication

Model: We consider
(don't worry whether t



Distinguish different processes: A process is capable of hosting a replica of an object :

- **Permanent replicas:** Process/machine always having a replica (the discussion so far!)
- **Server-initiated replica:** Process that can dynamically host a replica on request of another server in the data store
- **Client-initiated replica:** Process that can dynamically host a replica on request of a client (**client caches**)

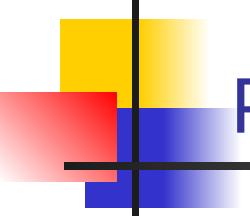


Client initiated replicas (caches)

Issue: What do I propagate when content is dynamic?
State vs. operations

- Propagate only **notification/invalidation** of update (often used for web caches)
- **Transfer data** from one copy to another (often for caching in distributed databases)
- **Propagate the update** operation to other copies (similar to active replication)

Note: No single approach is the best, but depends highly on (1) available bandwidth and read-to-write ratio at replicas; (2) consistency model



Propagating updates (1/2)

- **Pushing updates**: server-initiated approach, in which the update is propagated regardless of whether the target asked for it.
- **Pulling updates**: client-initiated approach, in which client requests to be updated.

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update message (and possibly fetch update later)	Poll <u>and</u> update
Response time at client	Immediate (or fetch-update time)	Fetch-update time



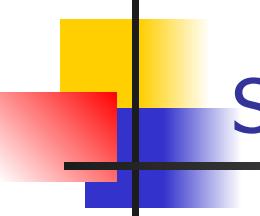
Propagating updates: Leases (2/2)

Observation: We can dynamically switch between pull and push using **leases**:

- *Lease: A contract in which the server promises to push updates to the client until the lease expires.*

Additional Technique: Make lease expiration time dependent on system's behavior (adaptive leases):

- **Age-based leases:** An object that hasn't changed for a long time, will not change in the near future, so provide a long-lasting lease
- **Renewal-frequency based leases:** The more often a client requests a specific object, the longer the expiration time for that client (for that object) will be
- **State-based leases:** The more loaded a server is, the shorter the expiration times become



Summary

Issue 1. Dealing with data changes

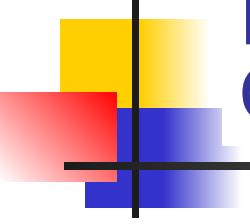
- **Consistency models**
 - What is the semantic the system implements
 - (luckily) applications do not always require strict consistency
- **Consistency protocols**
 - How to implement the semantic agreed upon?

Issue 2. Replica management

- How many replicas?
- Where to place them?
- When to get rid of them?

Issue 3. Client Side Caching

- How to deal with client initiated replication

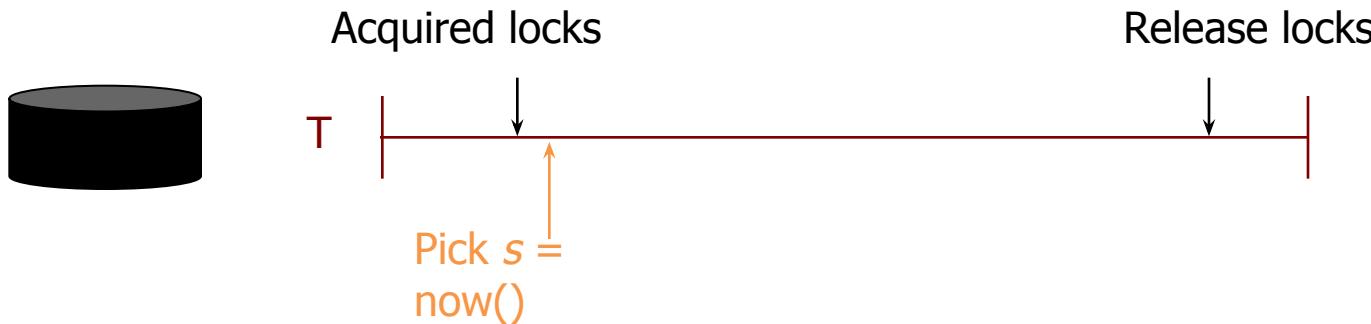


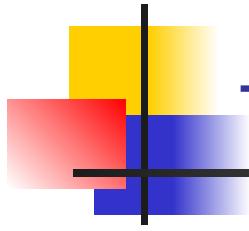
[Intuition for] Google's spanner (much simplified)

- **Goal:** provide linearizability (external consistency)
- **Context:** slightly different
 - distributed transactions (rather than replication)
- **Key issue:** how to execute transactions and assign timestamps such that
 - “order reflects the order in which the transactions *appear* to execute to an external observer (e.g., an application).”
- **Key innovation:**
 - Atomic clocks and reasoning about time uncertainty.

[Google's spanner] Timestamps, Global Clock

- Strict two-phase locking for write transactions
- Assign timestamp while locks are held



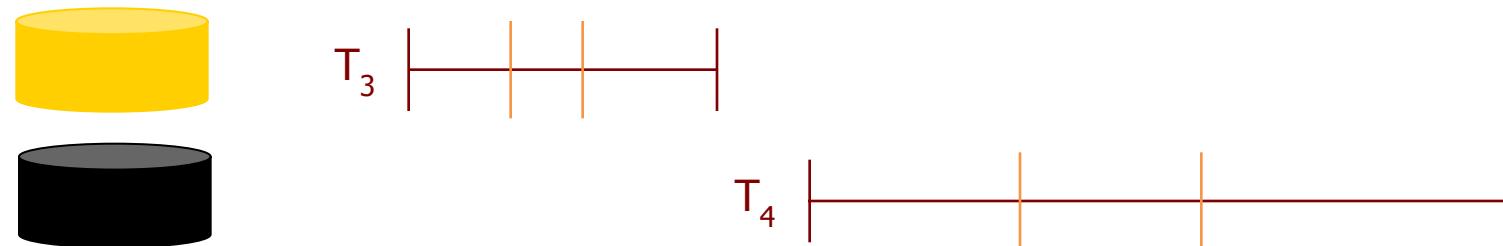


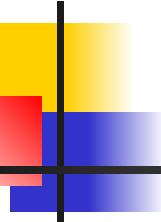
Timestamp Invariants

- Timestamp order == commit order

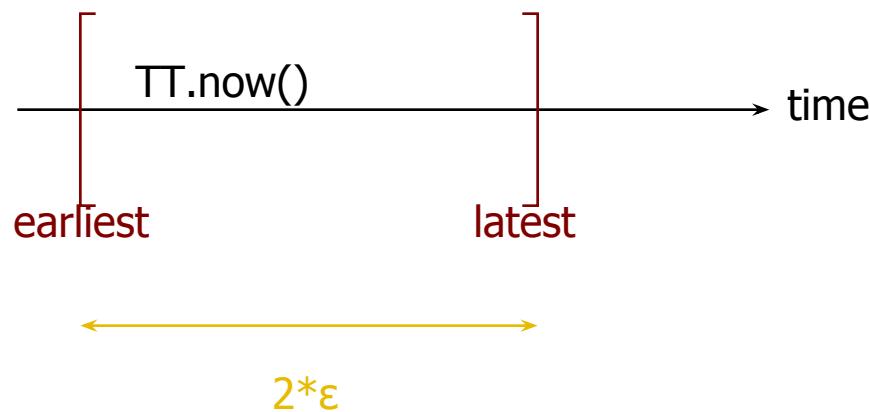


- Timestamp order respects global wall-time order

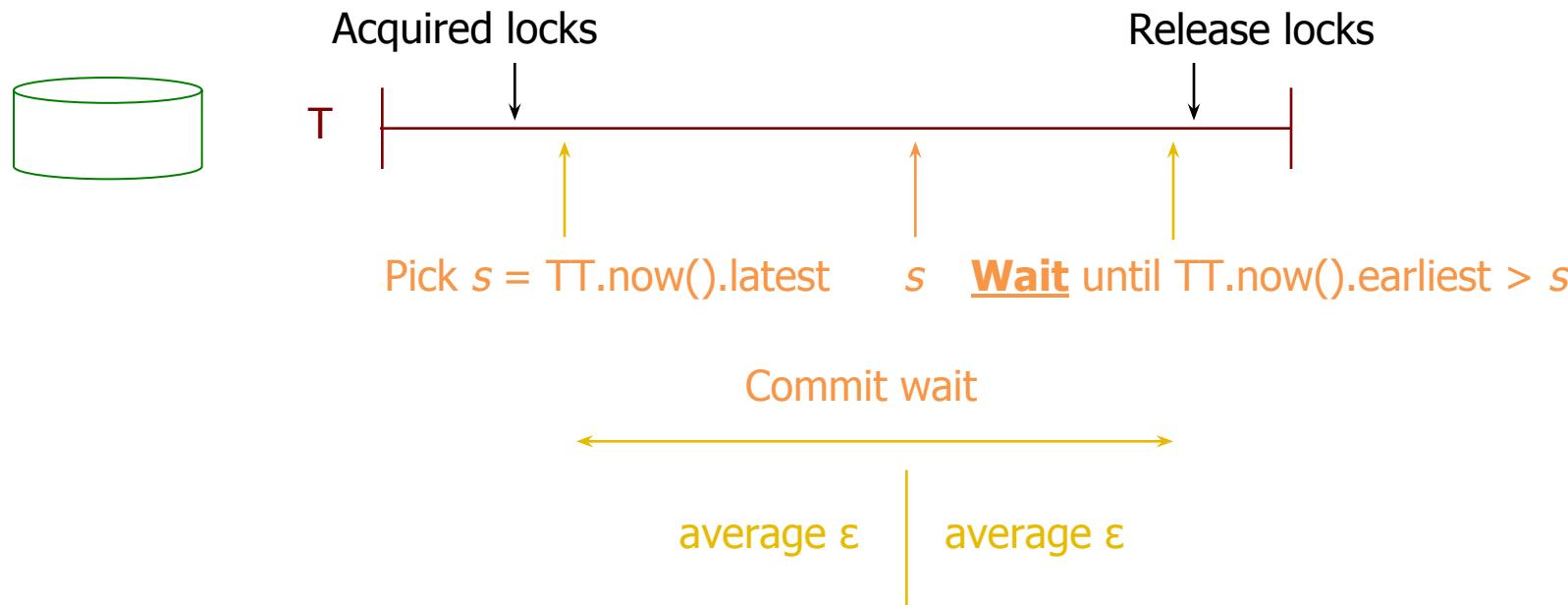


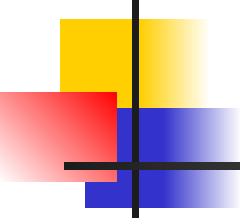


- TrueTime: “Global wall-clock time” with bounded uncertainty



Timestamps and TrueTime



- 
1. Each processor issues requests in the order specified by the program
 - Do not issue the next request unless the previous one has finished
 2. Requests to an individual memory location are served from a single FIFO queue
 - Writes occur in a single order
 - Once a read observes the effect of a write, it's ordered behind that write