THE UNIVERSITY OF CHICAGO

# 7. Consistency and Replication

## MPCS: 52040 Distributed Systems

Kyle Chard

chard@uchicago.edu

# Course schedule (* things might change a little)

| Date | Lecture discussion topic | Class | Assessment |
|---|---|---|---|
| January 10 | 1. Introduction to Distributed Systems | Intro/Logistics | |
| January 17 | 2 Distributed architectures<br>3 Processes and virtualization | 1 Docker | Homework 1 due |
| January 24 | 4 Networks and Communication | 2 RPC/ZMQ/MPI | Homework 2 due |
| January 31 | 5 Naming | 3 DNS/LDAP | Homework 3 due |
| February 7 | 6 Coordination and Synchronization | | **Mid term exam** |
| February 14 | 7 Fault tolerance and consensus | Raft<br>Project description | Homework 4 due<br>(Project released) |
| February 21 | 8 Consistency and replication | | |
| February 28 | 9 Distributed data | 4 Distributed Data | |
| March 7 | 10 Data-intensive computing | 5 FaaS | **Project due (March 9)**<br>**Final exam (March 14)** |

# Midterm

| Median | Maximum | Mean |
| --- | --- | --- |
| 34.0 | 40.0 | 32.97 |

# Agenda

- Part 1: Data-centric consistency models
- Part 2: Client-centric consistency models
- Part 3: Replica management and consistency protocols

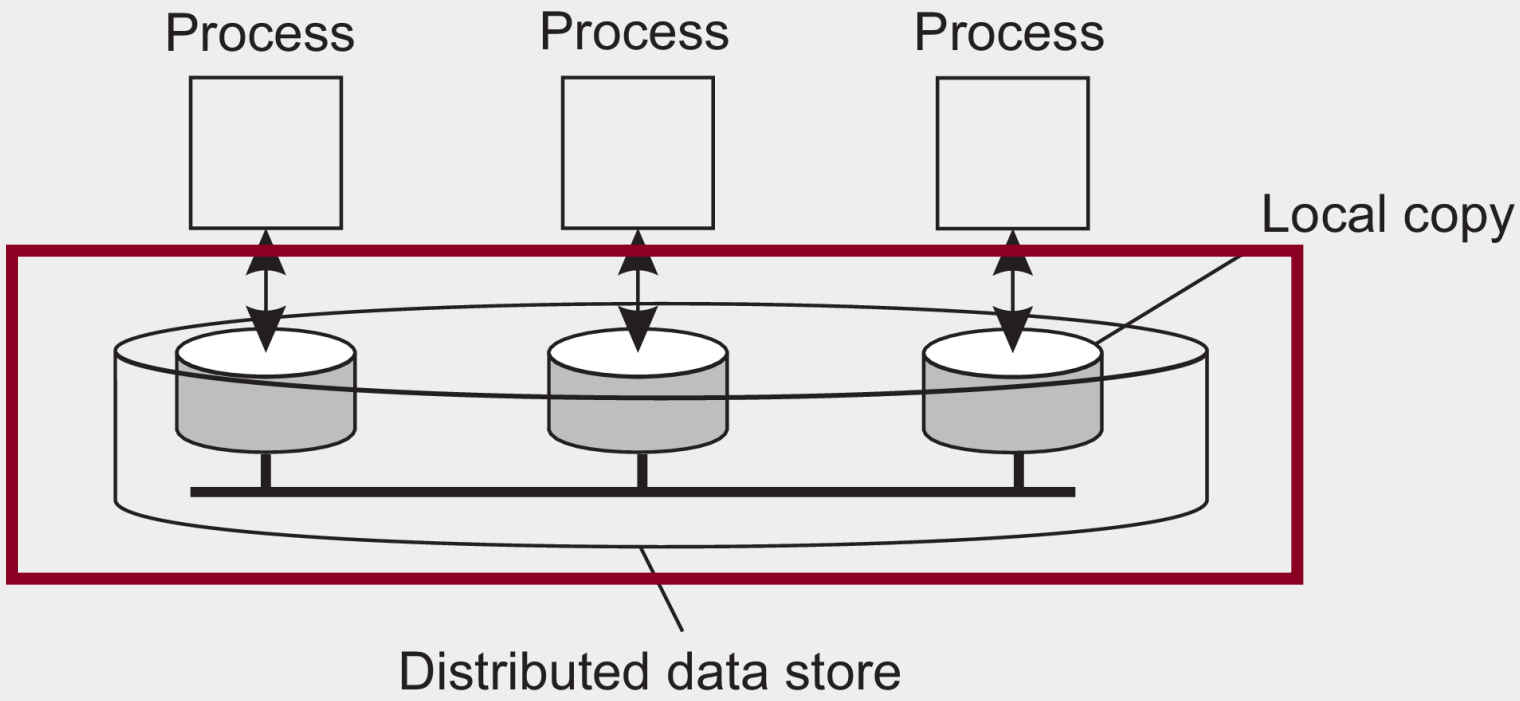
# Replication

- Data (or processes) are replicated for
  - Reliability
    - Continue working after a crash, protect against corruption, etc.
  - Performance
    - Scaling in terms of system size, number of users, geographical distribution, etc.

- As soon as we replicate we then face the challenge of keeping replicas *consistent*
  - When one copy is updated we need to update the others
  - In some cases we don't need absolute consistency, it might be ok to read a stale copy

# Part 1: Data-centric consistency models

- Consistency is typically discussed in the context of read/write operations on shared data in a distributed data store



# Basic idea

- To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order everywhere
- Conflicting operations:
  - Read–write conflict: a read operation and a write operation act concurrently
  - Write–write conflict: two concurrent write operations
- Guaranteeing global ordering on conflicting operations may be a costly operation (and reduce scalability)
  - Solution: weaken consistency requirements so that hopefully global synchronization can be avoided

# Consistency model

- Consistency model is a contract between processes and the data store
  - It says that if processes (or data store) agree to obey certain rules, the store promises to work correctly
- Normally, a process that performs a read operation on a data item expects the operation to return a value that shows the results of the last write operation on that data
- Without a global clock its hard to know which operation was the last… So we use more relaxed consistency models restricting what can be returned

# Data-centric consistency models

- Data-centric consistency models
  - Continuous consistency
  - Sequential consistency
  - Causal consistency
  - Eventual consistency

# Continuous consistency

- If we loosen the consistency model then we can exploit efficient replication solutions…
  - But all applications are different and thus we should consider different ways of loosening consistency
- How can we loosen consistency? Consider a degree of consistency.
  - replicas may differ in their numerical value (e.g., +/- 5%)
  - replicas may differ in their relative staleness (e.g., weather report relevant for hours)
  - differences with respect to (number and order) of performed update operations (within some bound)

# Data-centric consistency models

- Data-centric consistency models
  - Continuous consistency
  - Sequential consistency
  - Causal consistency
  - Eventual consistency

# Representing consistency

- Parallel computing has formalized consistency for concurrent access to shared data -> methods for consistent ordering of operations

- We need a way of representing data store access
  - W(x)a: process writes value *a* to *x*
  - R(x)b: process reads *x* and returns *b*

# Sequential consistency

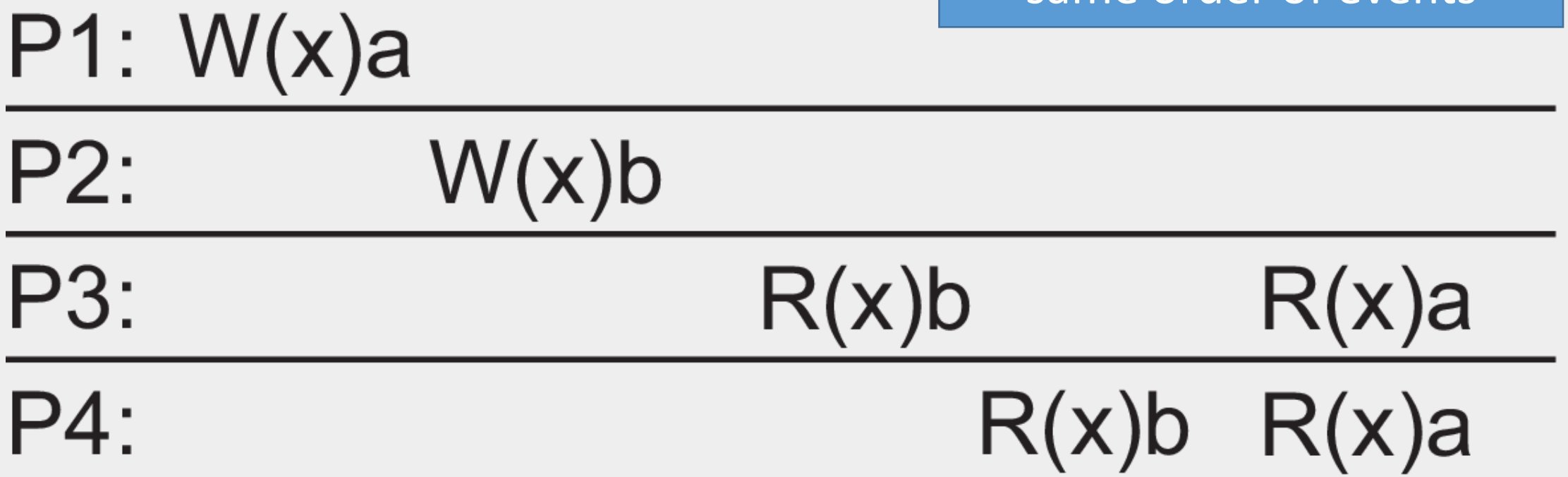
- Lamport, 1979

  *The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program*

- Note: no mention of time, just that processes see the same order/interleaving of operations

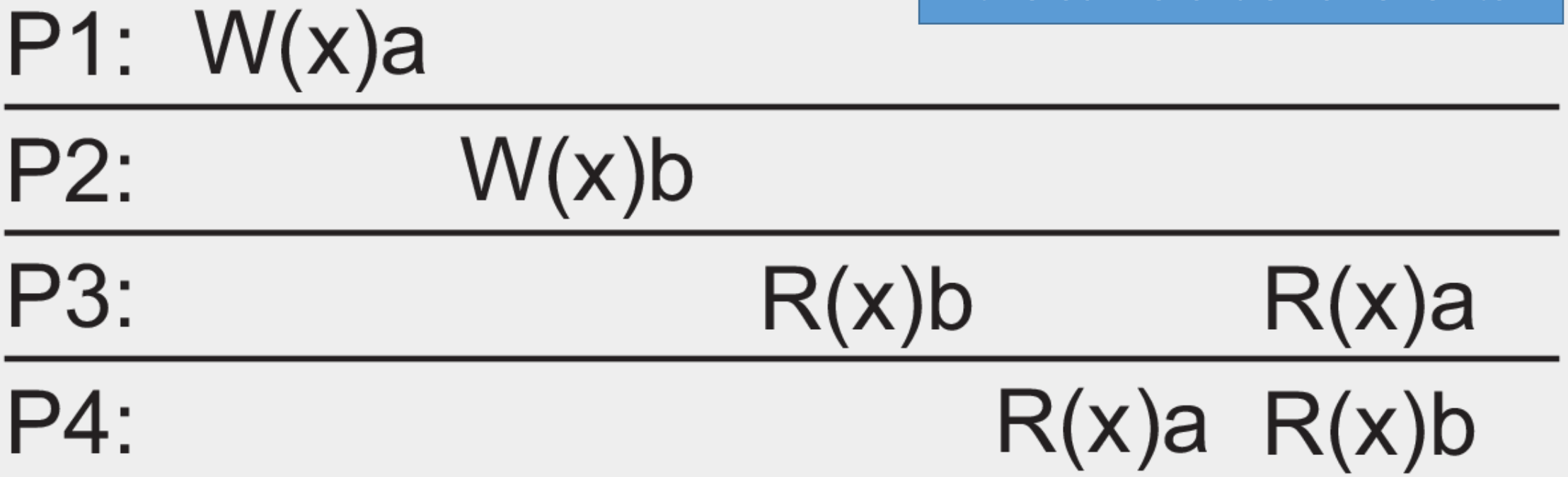
# Sequentially consistent?

YES: P3 and P4 see the same order of events



# Sequentially consistent?

NO: P3 and P4 do not see the same order of events



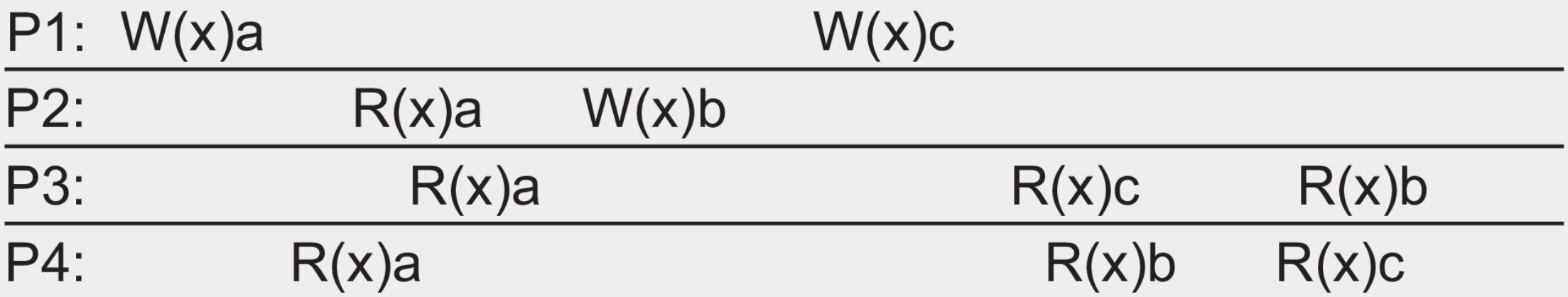
# Causal consistency

- Remember we often we don't care about sequential ordering.. What we care about is ordering of events that are potentially causally related
  - If event b is caused by event a, causality requires that everyone first see a and then b

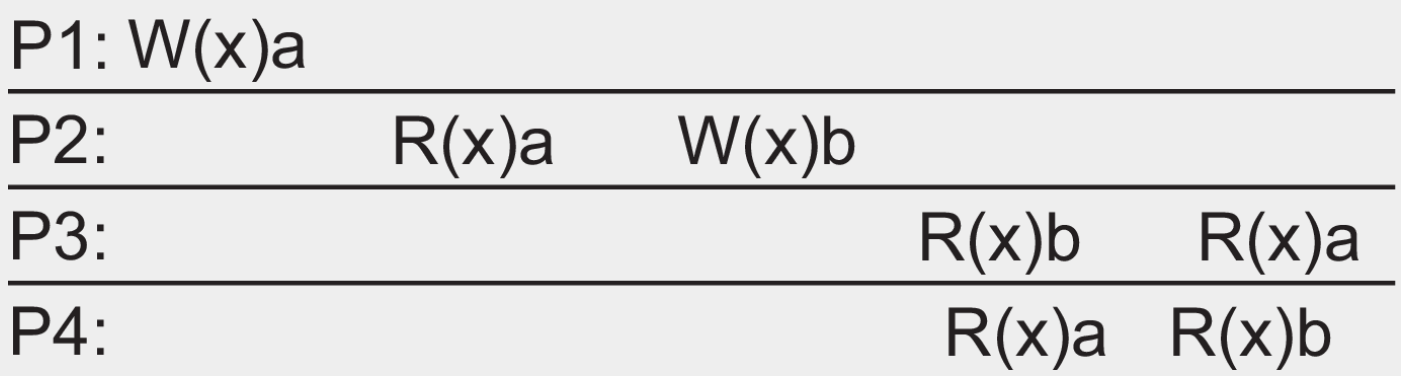
- For a data store to be considered causally consistent, it is necessary that the store obeys the following conditions:
  - *Writes that are potentially causally related must be seen by all processes in the same order.*
  - *Concurrent writes may be seen in a different order on different machines.*

# Causally consistent?

YES: P3 & P4 see R(x)a and then R(x)c . W(x)a and W(x)c are causally related. (W(x)b and W(x)c are “concurrent”)



# Causally consistent?




NO: P2 shows that there might be a causal dependency between R(x)a and W(x)b which is not seen by P3

# Data-centric consistency models

- Data-centric consistency models
  - Continuous consistency
  - Sequential consistency
  - Causal consistency
  - Eventual consistency

# Eventual consistency

- Concurrency is not always common
  - E.g., a database typically has many more reads than writes
  - E.g., on the web, pages are typically updated by one person and users typically tolerate cached data (proxies/browsers)
- In the absence of write-write conflicts, the state will converge across replicas
- More to come in the NoSQL lecture

# Summary

- Replication is necessary for performance and fault tolerance
- Replication leads to the challenge of keeping data consistent
  - Aim is to ensure that conflicting operations are done in the same order everywhere
- We can loosen consistency guarantees to improve performance
  - Sequential consistency:  all processes see the same order of all operations
  - Causal consistency: Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines
  - Eventual consistency: in the absence of write-write conflicts, all replicas will converge toward identical copies of each other

# Part 2: Client Centric Consistency

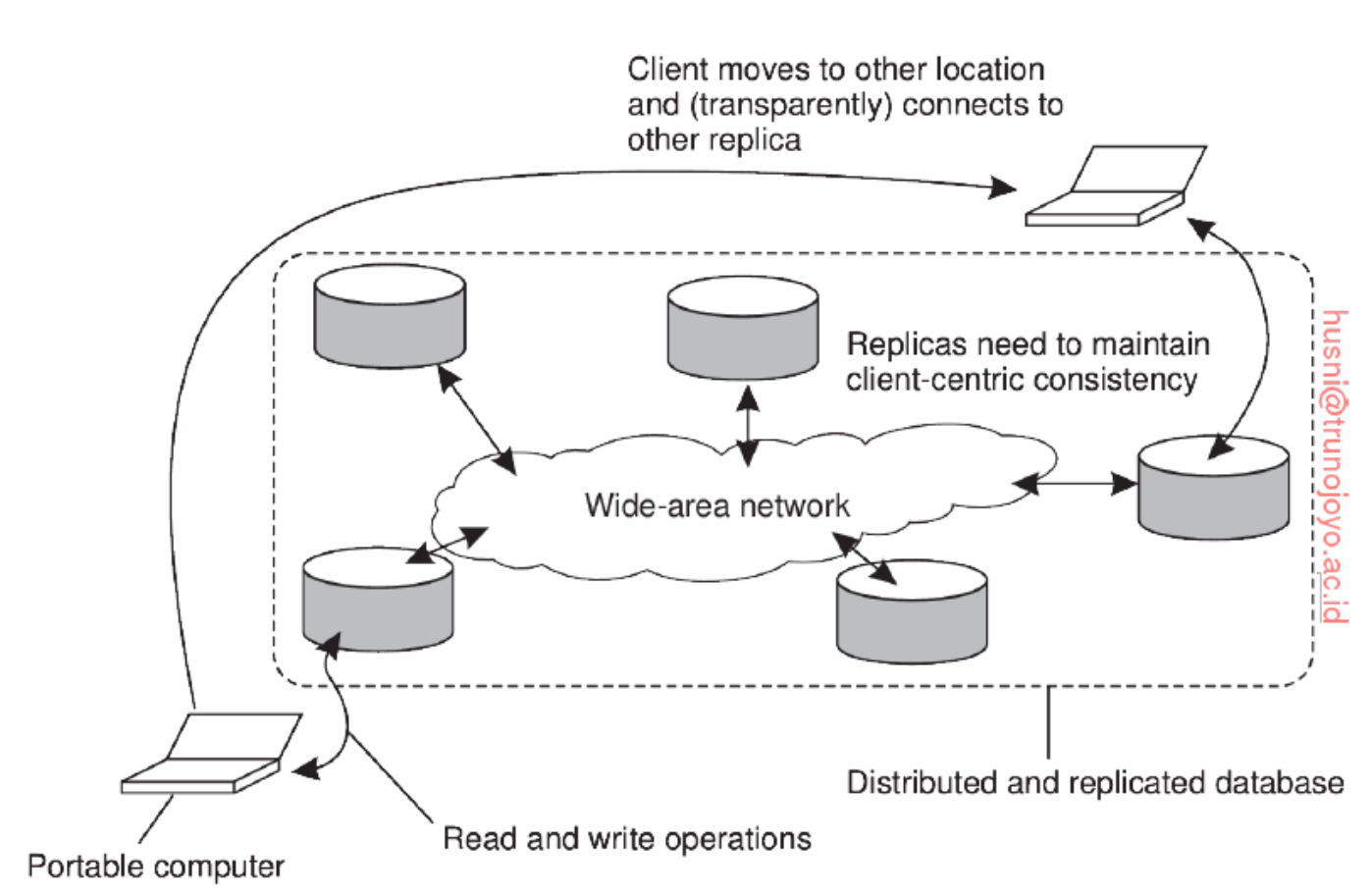
# Part 2: Client Centric Consistency

- Client-centric consistency models
  - Monotonic reads
  - Monotonic writes
  - Read your writes
  - Writes follow reads

# From data- to client-centric consistency

- Data-centric consistency aims to provide system-wide consistency across a data store
  - Turns out this is hard and we may be able to relax consistency requirements
- We consider now a type of data store where we don’t have many concurrent updates (or when updates do happen, they can be easily resolved as they don’t affect one another)
  - Mostly read-centric workloads
- We can apply various weak consistency models (e.g., eventual consistency) from the perspective of the **client**

# Example

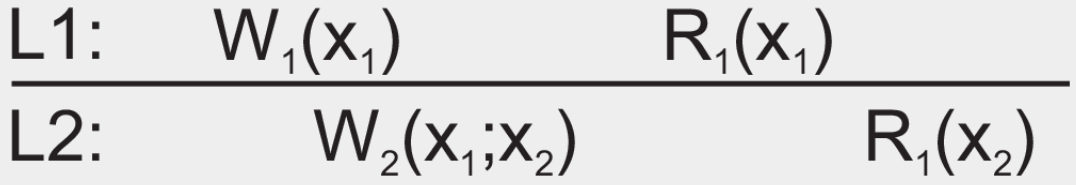

Client moves to other location
and (transparently) connects to
other replica
Replicas need to maintain
client-centric consistency
Wide-area network
husni@trunojoyo.ac.id
Distributed and replicated database
Read and write operations
Portable computer

# Note: notation is changing

- Xi is the $i^{th}$ version of data item x

- W1(x2): write operation by process P1 that leads to version x2 of x
- W1(xi;xj): process P1 produces version xj based on xi
- W1(xi|xj): process P1 produces version xj concurrently to version xi.
  - We do not know which follows which or if there is any dependency between xi and xj

- **L is a local data store**, rather than a process. Any process can access that data store.

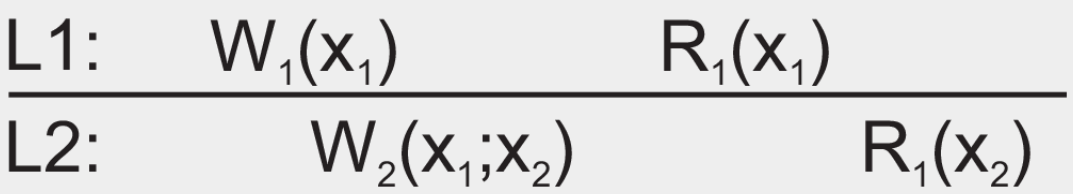


# Goal

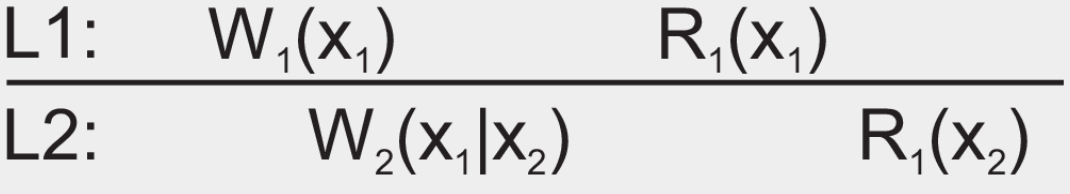
- Eventual consistency guarantee is
  - “given no updates (writes) all clients will see exactly the same state of a system in some time”
- If we stop doing writes, the system will (*eventually)* converge to some consistent state.
  - This is a very weak constraint that is (probably) simpler to implement and could be very performant, but not always possible
- Let’s look at some ways of ordering reads/writes to simplify eventual consistency. These methods are so-called “session” guarantees
  - Nothing guaranteed outside that session (some client set of operations or context)

# 1) Monotonic reads

- *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*
- The read operations performed by a single process P at two different local copies of the same data store




(a) A monotonic-read consistent data store.

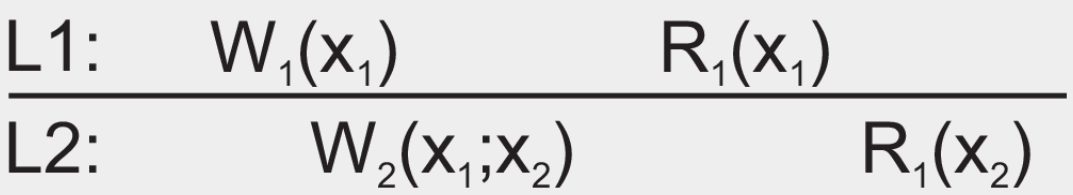



(b) Data store that does not provide monotonic-reads.

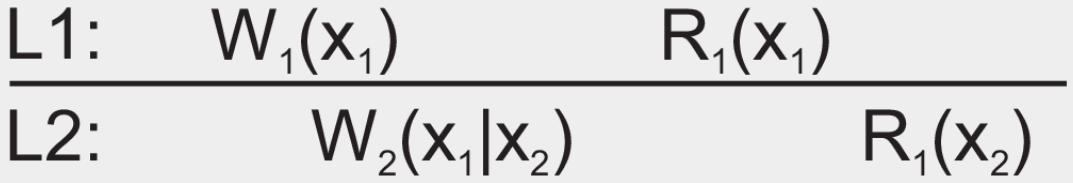
# 1) Monotonic reads

- *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*
- The read operations performed by a s
  local copies of the same data store

**After P1 has read x1, it later reads x2… but the previous write W2(X1|x2) produces an x2 that does not follow from x1. So P1s read at L2 is known not to include the effect of write when it read at R1.**




(a) A monotonic-read consistent data store.



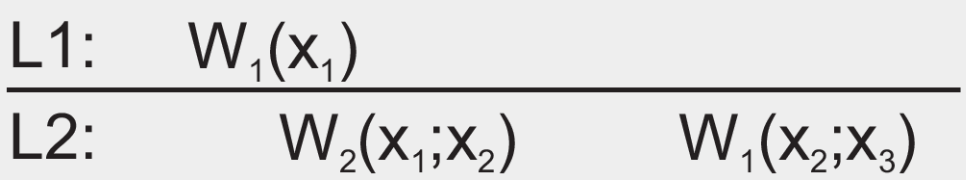

(b) Data store that does not provide monotonic-reads.

# Examples and Counter Examples

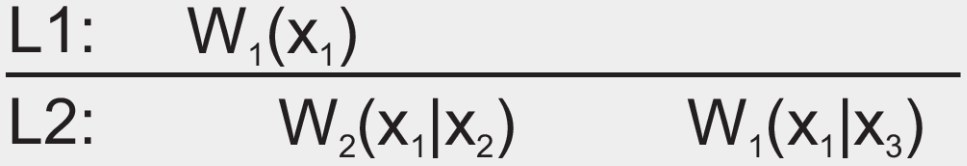
- Monotonic reads:
  - Calendar always has all events in it that you have previously seen
- Not Monotonic reads:
  - Access to instant messenger has messages disappear

# 2) Monotonic writes

- *A write operation by a process on a data item x is completed before any successive write operation on x by the same process*
- That is, if we have two successive operations Wk(xi) and Wk(xj) by process *k*, then, regardless of where Wk(xj) takes place, we also have W(xi;xj)




(a) A monotonic-write consistent data store.



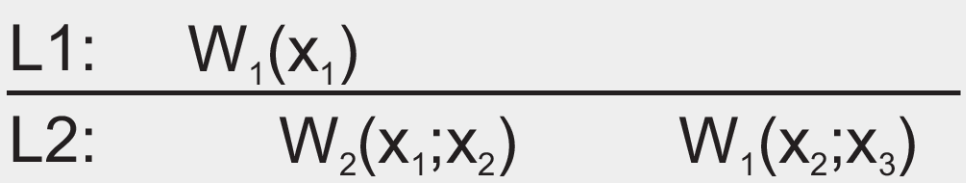

(b) Data store that does not provide monotonic-write consistency.

# 2) Monotonic writes

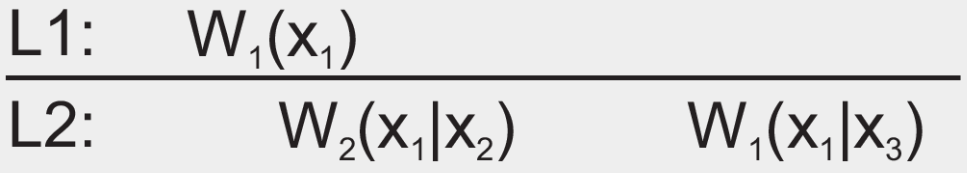
- *A write operation by a process on a data item x is completed before any successive write operation on x by the same process*
- That is, if we have two successive ope
process Pk, then, regardless where W
Ws(xi;xj)

**Here the propagation of x1 to L2 is missing before a new version of x is produced.**

**P2 produces a concurrent version to x1 after which it produces x3 but concurrently to x1**




(a) A monotonic-write consistent data store.




(b) Data store that does not provide monotonic-write consistency.

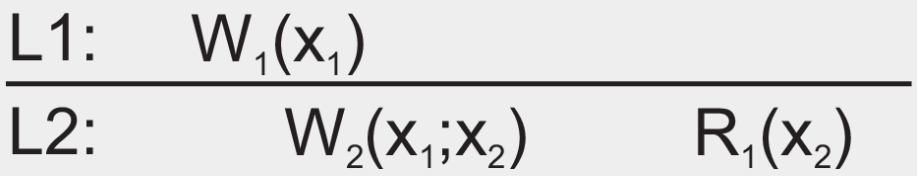
# Examples and Counter Examples

- Monotonic writes:
  - All versions of replicated files are in the correct order everywhere

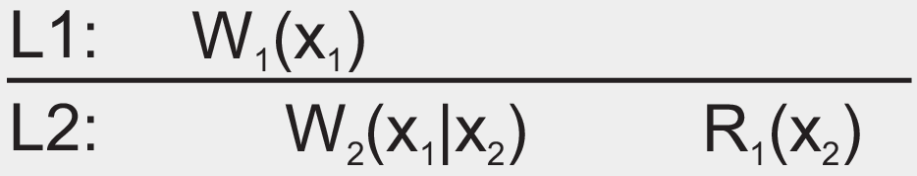
- Not Monotonic writes:
  - Message queue has messages in the wrong order based on what the process wrote

# 3) Read your writes

- *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*




(a) Read-your-writes consistency

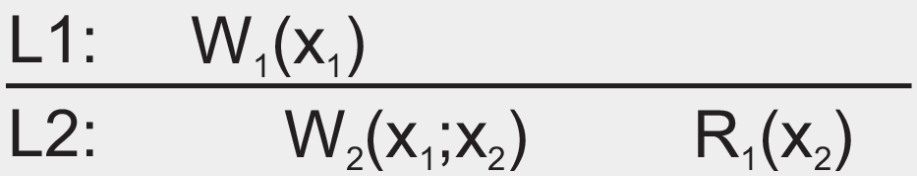



(b) Does not provide read-your-writes consistency

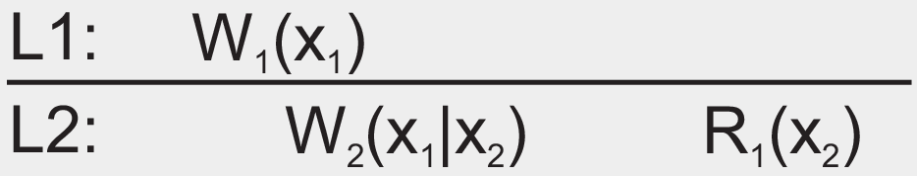
# 3) Read your writes

- *The effect of a write operation by c*
  *be seen by a successive read opera*

**P2 produces a version concurrently to x1. This means that the effects of the previous write operation by P1 have not been propagated to L2 at the time x2 was produced. When P1 reads x2, it will not see the effects of its own write operation at L1.**




(a) Read-your-writes consistency




(b) Does not provide read-your-writes consistency

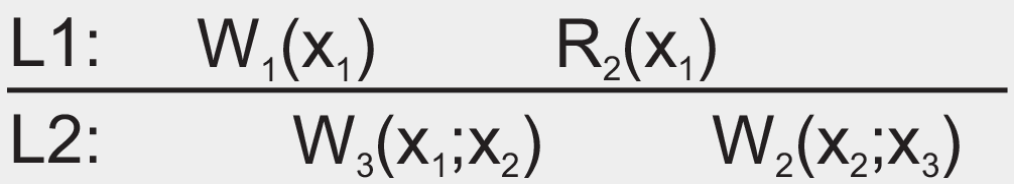
# Examples and Counter Examples

- Read your writes
  - You make an update to a web page and the consistency model ensures that when you reload the page it is the newest version rather than older cached version

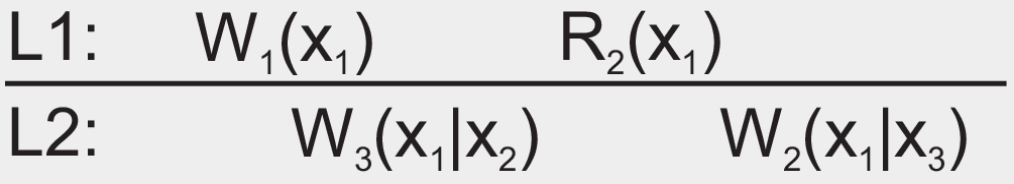
- Read your writes
  - You delete an email message, but when you refresh the email the message returns

# 4) Writes follow reads

- *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*




(a) Writes-follow-reads consistent store

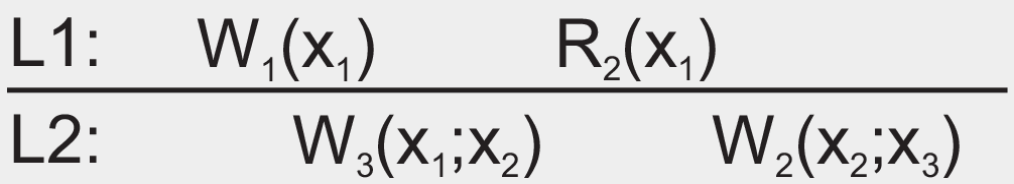



(b) Store that does not provide writes-follow-reads consistency

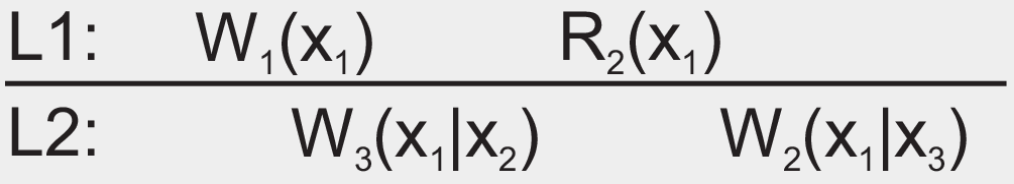
# 4) Writes follow reads

- *A write operation by a process on x is guaranteed to take place on a read operation on x by the same process on the same or a more recent value of x*

**P3 produces a version x2 concurrently to that of x1. As a consequence, when P2 updates x after reading x1, it will be updating a version it had not read before**




(a) Writes-follow-reads consistent store




(b) Store that does not provide writes-follow-reads consistency

# Examples and Counter Examples

- Writes follow reads
  - See reactions to posted articles if you have the original posting (read retrieves the corresponding write operation)

- Writes follow reads
  - Imagine you read a blog post, and you go to the comments section.
  - You read a comment by some Client A – e.g., asking a question.
  - You know the answer and respond
  - Now, imagine Client C goes to the comments and sees only your answer but not the question from A
  - (Your comment should only be seen with the one by Client A, if both are missing, that satisfies requirement)

# Part 3: Replica Management and Consistency Protocols

# Agenda

- Replica management
  - Choosing a location
  - Replica decision making
  - Content distribution
- Consistency protocols

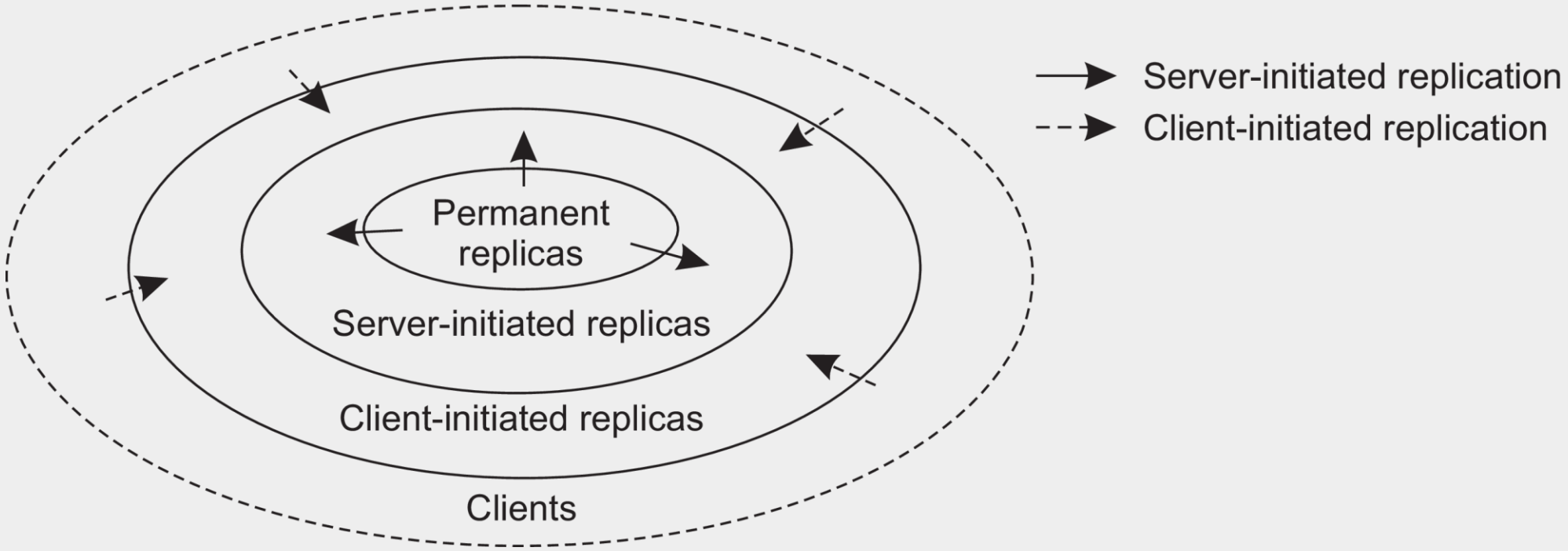
# Replica Management

- Important to think about where, when, and by whom replicas should be placed

- Two related problems
  - Where should we place replica servers?
  - How should we place content on replicas?

# What and where to replicate?

- How do we determine what and where to replicate?



# Permanent replicas

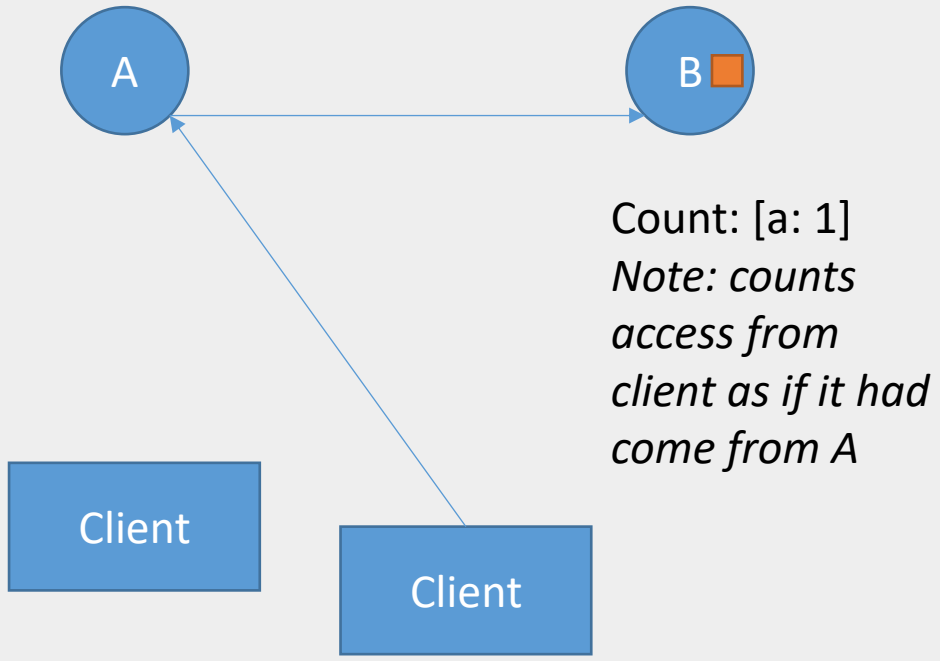
- Fixed replicas that are distributed when initially deployed
- Example: elastic load balancing for a web site
  - Each request is sent to a different replica
- Example: web site mirroring
  - Geographic distribution across the internet
  - Clients choose from a list of mirrors

# Server-initiated replicas

- Replicas are created when needed to improve performance
  - Typically created by the owner/data store
- Example: web server replicas
  - A server that receives a burst of requests far from the server might want to deploy replicas in regions close to the requests
  - A heavily loaded server might add replicas to reduce load on the server

# Server-initiated replicas

- Each server keeps track of access counts per file, aggregated by considering server closest to requesting clients
- Number of accesses drops below deletion threshold D ➔ drop file
- Number of accesses exceeds replica threshold R ➔ replicate file
- Number of access between D and R the file may only be migrated



# Client-initiated replicas

- Client-initiated replicates (caching)
  - Local storage used for temporarily storing a copy of requested data
  - Improve access to data that is likely to be used again
  - Could be on the local machine or could be a nearby machine (e.g., institution cache)
  - Most often the server has no knowledge of the cache, and provides no influence on caching decisions
- Big question is how long to keep data in a cache (when to invalidate)
  - Either because data is stale or to make room for other data
- Many approaches: LRU, FIFO, size-based

# What content should replicate?

- How do we now determine what content to replicate? And how do we decide how it should be replicated?

- Trade offs
  - State vs operations
  - Pull vs push

# State vs operations

- What should be propagated?

1. Notification of update (i.e., just that the data has changed)
   - Common with cache invalidation models
   - Uses little network bandwidth, good when many more updates than reads
2. Transfer data from one copy to another
   - Most common approach, good when many more reads than writes
   - Must update before the next read happens
3. Send the update operation to other copies
   - Each replica managed by process that keeps data up to date
   - Minimal bandwidth, but more work on replica

# Push vs Pull

- Pushing updates: server-initiated approach, in which update is propagated regardless whether target asked for it
  - Efficient when many more reads than writes (i.e., we expect the replica is read)
- Pulling updates: client-initiated approach, in which client requests to be updated
  - Efficient when many more writes than reads

| Issue | Push | Pull |
|---|---|---|
| Server state | List of client replicas and caches | None |
| Messages sent | Update (and possibly fetch update later) | Poll and update |
| Response time at client | Immediate (or fetch-update time) | Fetch-update time |

# Hybrid approach: leases

- We can dynamically switch between pulling and pushing using *leases*:
  - A contract in which the server promises to push updates to the client until the lease expires
- 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

# Agenda

- Replica management
  - Choosing a location
  - Replica decision making
  - Content distribution
- Consistency protocols

# Implementing consistency models: Consistency protocols

- A consistency protocol is an implementation of a consistency model

- Data centric
  - Continuous consistency
  - Sequential consistency

- Client centric
  - Monotonic reads

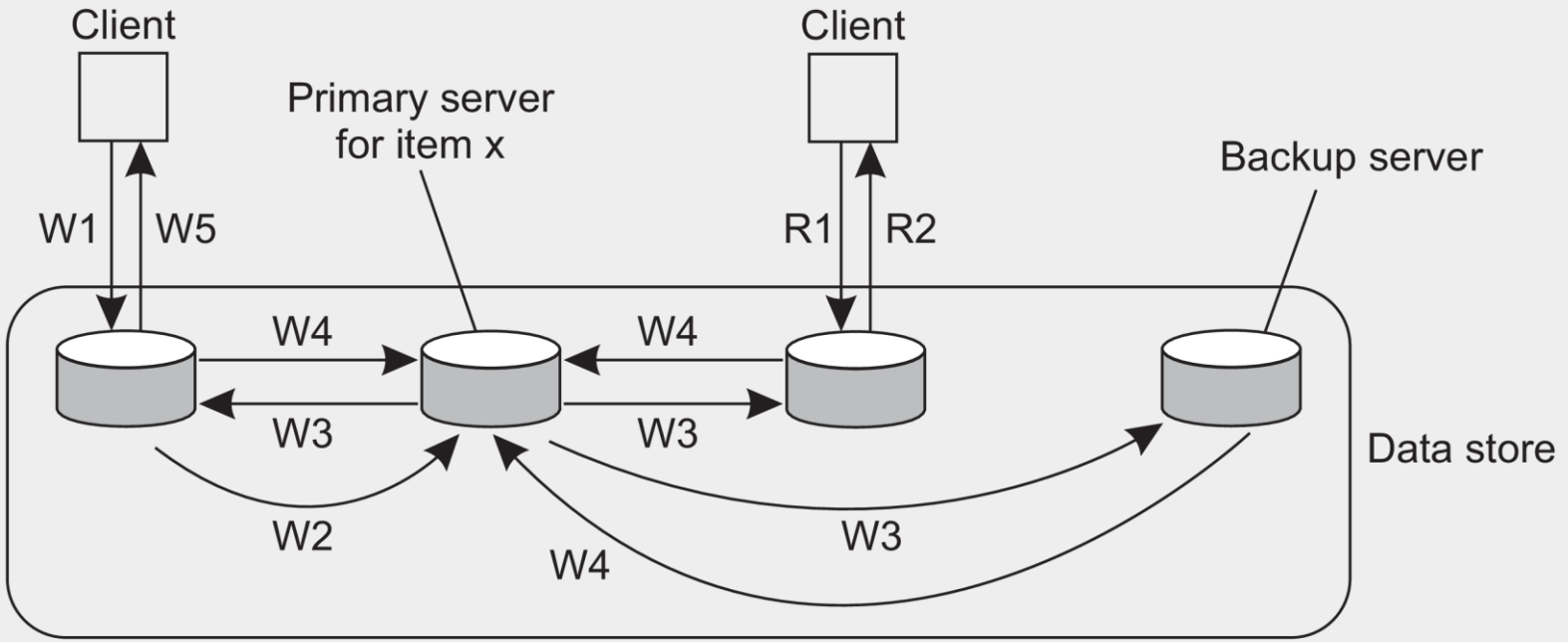
# Continuous consistency

- Bounding numerical deviation
  - Aim to keep all servers within some delta
  - Each server maintains a log of updates
  - When a server notices that another server is not within bounds, it forwards writes from its log to the other
- Bounding staleness deviation
  - Keep staleness of replicas within bounds
  - Use vector clock and pull updates from more advanced servers when noticing that it is about to exceed a time limit
- Bounding ordering deviation
  - For situations where servers store tentative writes in a queue
  - When the length of the queue exceeds a length it will no longer accept tentative writes and will instead try to commit them

# Sequential Consistency

- Most applications apply easy to understand consistency models
  - Issue is that developers will struggle to implement correct applications
- For sequential consistency most often we use primary-based protocols
  - Each data item in the data store has an associated primary store which is responsible for coordinating writes on x
- Two main models: remote-write or local-write

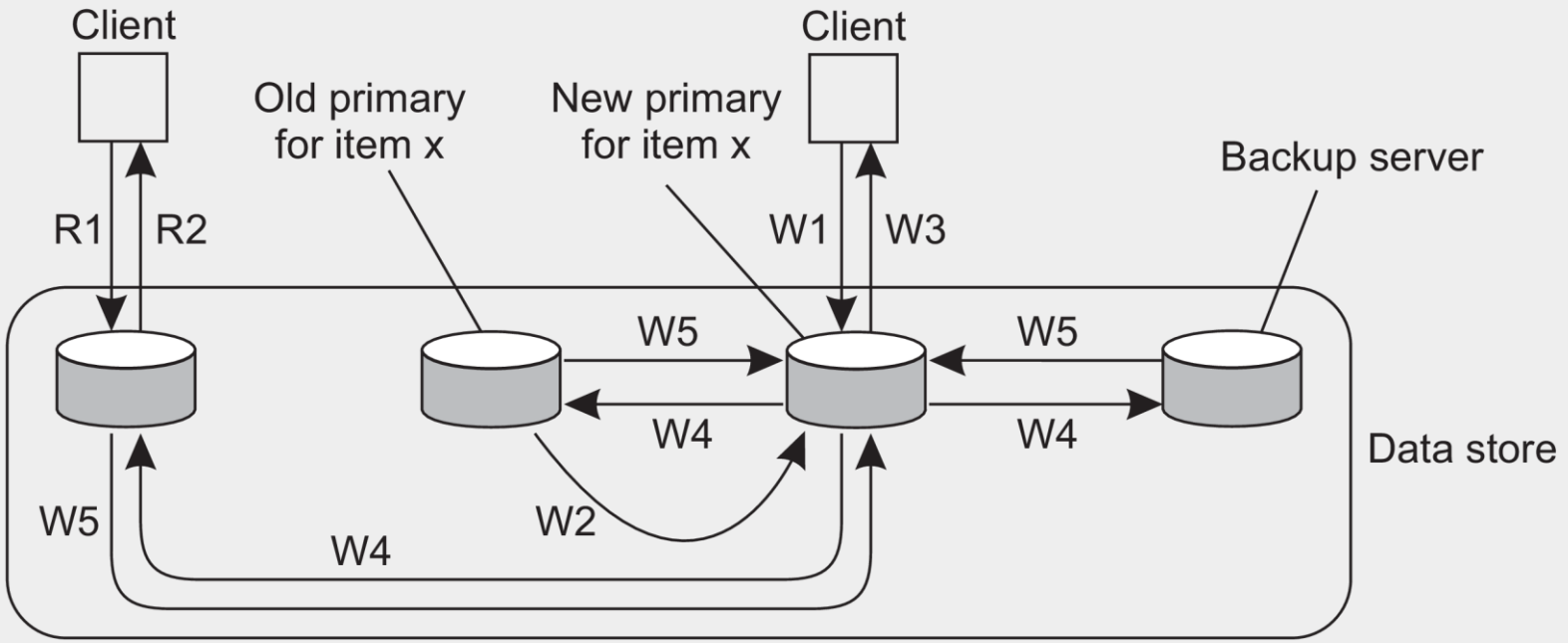
# Remote-write protocols




W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

# Local-write protocols

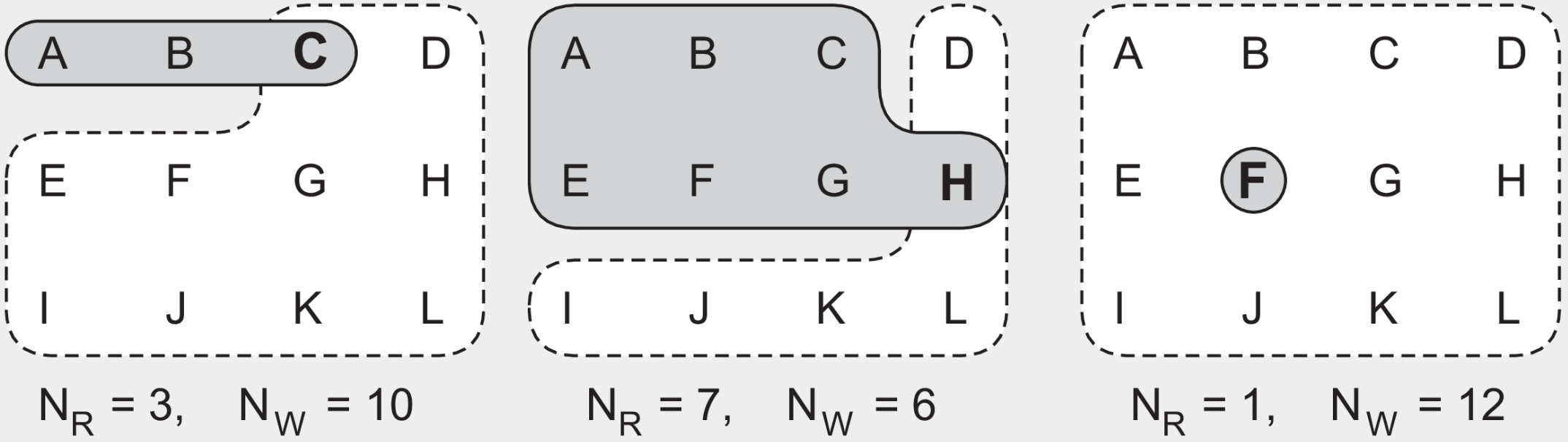



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

# Replicated-write protocols

- Write operations carried out by multiple replicas
  - Active replication: operation forwarded to all replicas
  - Consistency protocols: majority voting
- Read and write quorums



# Client-centric consistency

- Simple model
  - Each write assigned a globally unique ID (by the **origin** server receiving the write)
  - Each client then tracks
    - Read set – writes relevant for their read operations
    - Write set – writes performed by the client
- Monotonic reads
  - When a client reads, it passes the server its read set
  - Server checks it has those writes locally, if not, contacts other servers to retrieve writes before performing the read (or could pass the read to one of these servers)
  - All writes taken place at the server and relevant to the read are added to the read set

# Summary

- Important to think about where, when, and by whom replicas should be placed

- Two problems
  - Where should we place replica servers?
    - Permanent
    - Server initiated
    - Client initiated
  - How should we place content on replicas?
    - Primary protocols: easy to implement
    - Replicated (quorum-based) write: tradeoffs between read and write performance