# Consistency and Replication (part 1)

ECE 454 / 751: Distributed Computing

Instructor: Dr. Wojciech Golab

wgolab@uwaterloo.ca

Slides are derived from A. S. Tanenbaum and M. Van Steen,
Distributed Systems: Principles and Paradigms, 2nd Edition, Pearson-Prentice Hall, 2006.

## Learning objectives

To develop a working knowledge of consistency models:

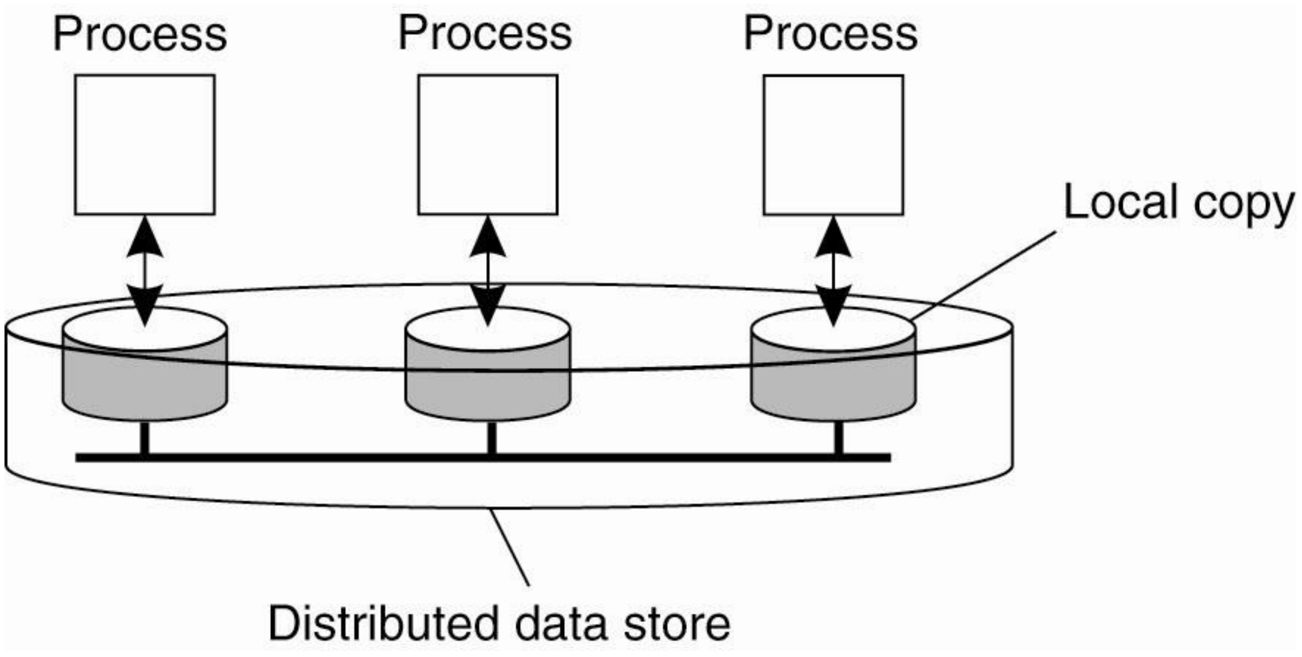
- sequential consistency
- causal consistency
- linearizability
- eventual consistency
- session guarantees

## Why should we replicate data?

- To improve **dependability**: replicating data three ways can prevent data loss if one or even two of the replicas fail simultaneously.
- To increase **throughput**: in principle, replicas can serve read operations in parallel.
- To decrease **latency**: keeping copies of data close to client applications avoids costly communication, for example between hosts in different geographical regions.

## Replicated data stores

In a replicated data store, each data object (e.g., row of a table) is replicated at multiple hosts. A replica (or copy) of an object may be **local** to a process, meaning that it resides on the same host, or it may be **remote**.



## Consistency models

- Replicating read-only data is straightforward. However, if a data store holds **mutable state,** then we can never keep replicas of this state perfectly synchronized due to variations in processing speeds and network delays. The situation becomes even more complex when the data store holds **shared mutable state**, which is read and updated concurrently by different processes.
- A **consistency model** helps us make sense of concurrent reading and updating by describing the extent to which replicas are permitted to disagree on the state of data.
- Application requirements in general do not map neatly to a specific consistency model, which makes it difficult to select a good model.

## Sequential consistency

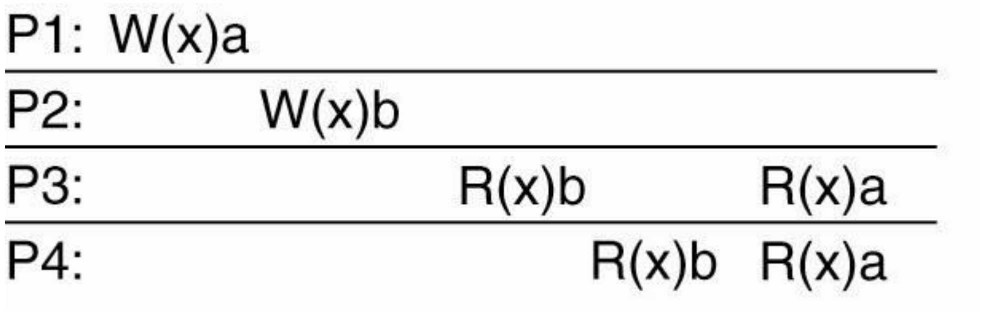
The notion of **sequential consistency** is borrowed from shared memory multiprocessing. It is often restricted to read and write operations, which correspond to load and store instructions.

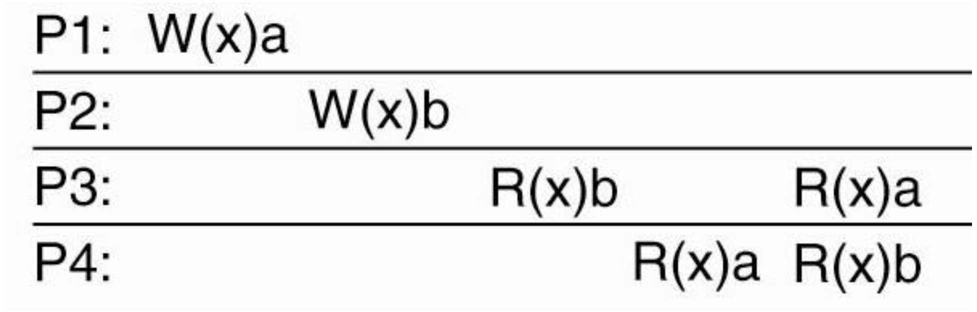
A data store is sequentially consistent when:

> The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

# Sequential consistency: examples

Positive example (top) and negative example (bottom) of sequential consistency.





**Notation:**
P1, P2, P3, P4 are IDs of processes.

W(x)a is an operation that writes value a to object x.

R(x)b is an operation that reads value b from object x.

# Causal consistency

**Causal consistency** is also borrowed from shared memory multiprocessing. It is based on a notion of causality.

A data store is causally consistent when:

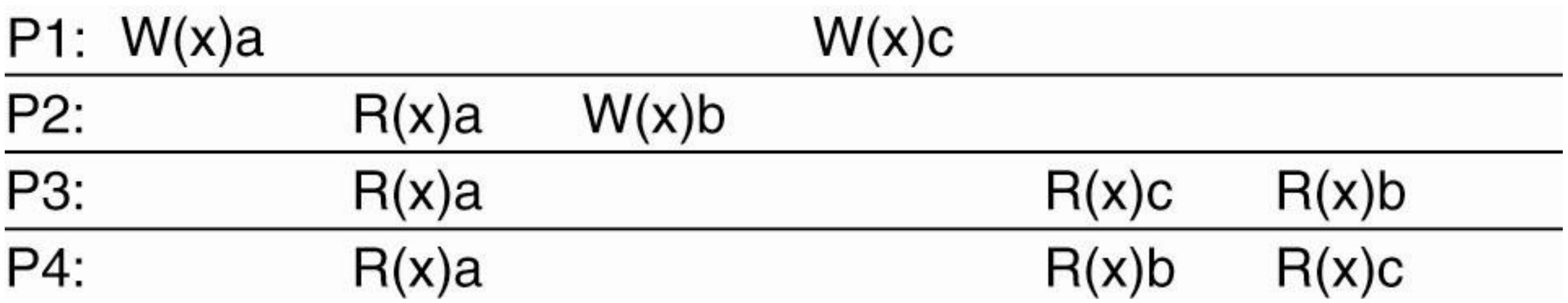
Writes related by the "causally precedes" relation must be seen by all processes in the same order. Concurrent writes (i.e., ones that are not related by “causally precedes”) may be seen in a different order by different processes.

**"Causally precedes"** is the transitive closure of two rules:

1. Op1 causally precedes Op2 if Op1 occurs before Op2 in the same process.
2. Op1 causally precedes Op2 if Op2 reads a value written by Op1.

# Causal consistency: examples

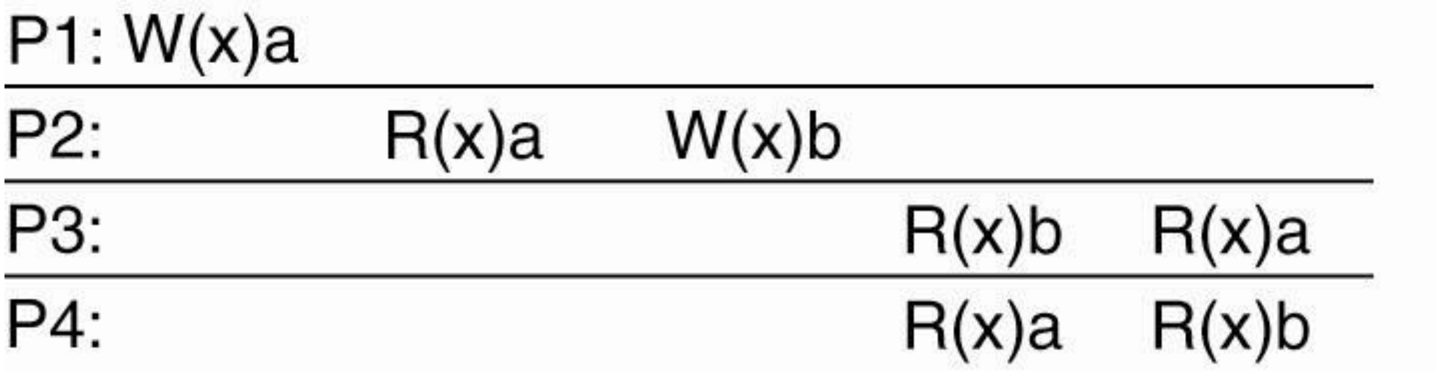
The execution shown below is causally consistent but not sequentially consistent. The write W(x)a causally precedes the read R(x)a, which causally precedes the write W(x)b. The writes W(x)b and W(x)c are concurrent.



| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

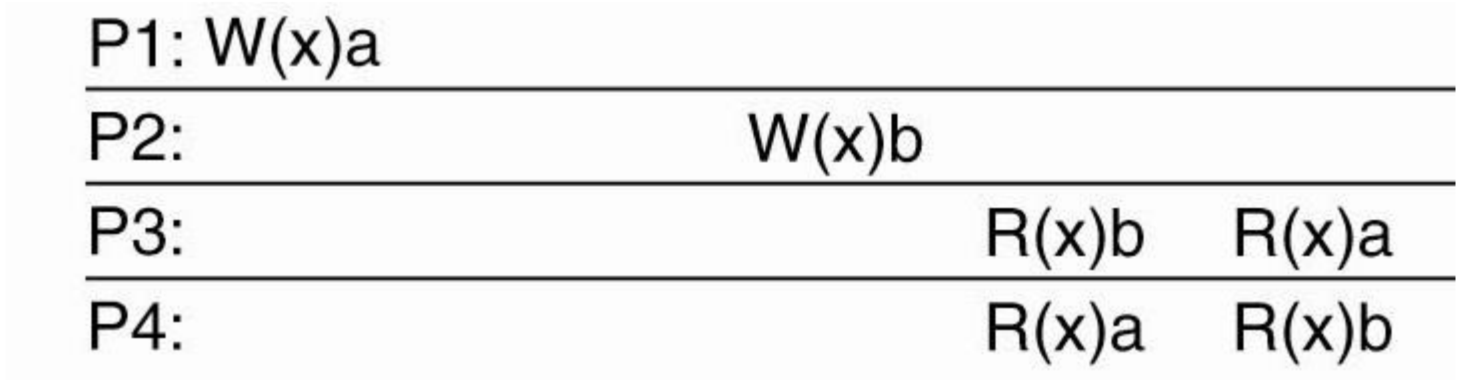
## Causal consistency: examples

The execution shown below is not causally consistent. As in the last slide, W(x)a causally precedes W(x)b due to transitivity. However, P3 reads b before a. Thus, the two writes are "seen" in an order different from the order prescribed by the "causally precedes" relation.



## Causal consistency: examples

The execution shown below is causally consistent. W(x)a is concurrent with W(x)b and therefore the two values can be read in either order.



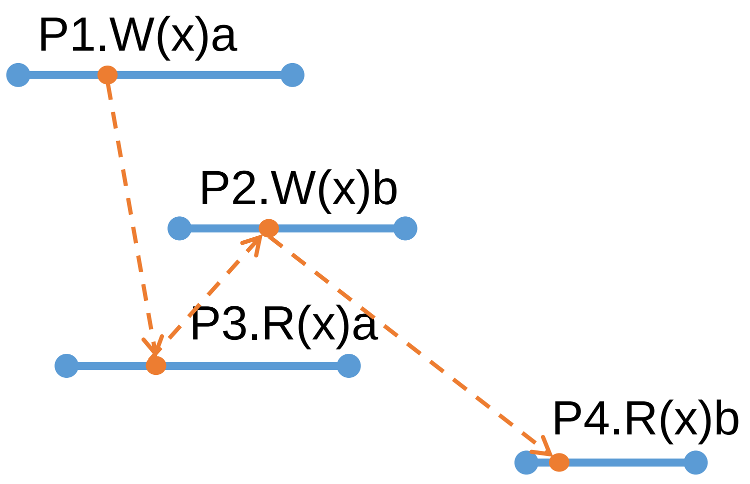
# Linearizability

Herlihy and Wing's **linearizability** property was originally defined as a correctness property for concurrent data structures. It is not restricted to read and write operations. It assumes that operations have well-defined start and finish (i.e., invocation and response) events that are totally ordered by a hypothetical global clock.

A data store is linearizable when:

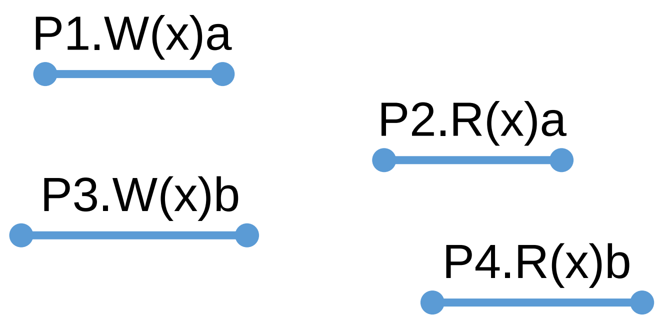
The result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order that extends the "happens before" relation. In other words, if operation Op1 finishes before operation Op2 begins, then Op1 must precede Op2 in the sequential order.

# Linearizability: examples



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

linearizable



?W(x)? ?W(x)? ?R(x)? ?R(x)?

not linearizable

**Note:** The orange dots are **linearization points**. They indicate the point in time (according to a hypothetical global clock) at which we perceive an operation to take effect.

## Eventual consistency

Textbook definition:

If no updates take place for a long time, all replicas will gradually become consistent.

Alternative definition used by Doug Terry:

In the absence of new writes from clients, all servers will eventually hold the same data.

**Note:** Eventual consistency allows different processes to observe write operations taking effect in different orders, even when these write operations are related by "causally precedes" or "happens before".

## Eventual consistency

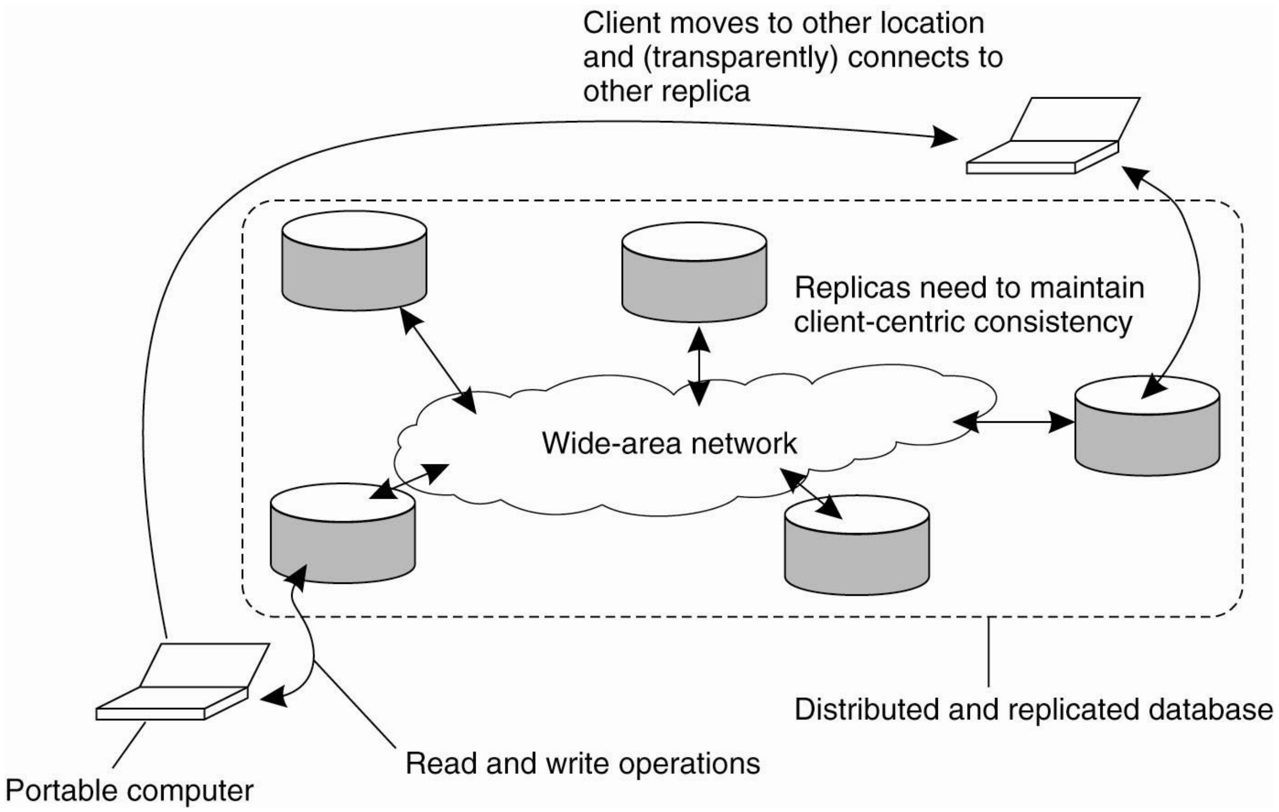
Positive examples of eventual consistency: any execution presented so far in this module.

Negative example of eventual consistency:

1. Process P1 executes W(x)a.
2. Process P2 executes W(x)b.
3. Process P3 executes R(x)a.
4. Process P4 executes R(x)b.
5. Steps 3 and 4 repeat forever. (Infinite execution!)

# Eventual consistency in Bayou

Updates in Bayou are propagated in a lazy manner.


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

# Session guarantees

Because eventual consistency is an extremely weak property, and appears to promise nothing in the case when updates are applied continuously, it is often augmented with **session guarantees** that restrict the behavior of operations applied by a single process in a single session.

**Monotonic reads:** If a process reads the value of a data item *x*, any successive read operations on *x* by that process will always return that same value or a <u>more recent value</u>.

**Read your own writes:** If a process writes a data item *x* and then reads it, the read operation returns either the value written by the same process earlier or a <u>more recent value</u>.

(See textbook for other properties.)

## Appendix (optional material)

# How to formalize linearizability

- A **history** H is a sequence of steps.
- A **step** is either the invocation of an operation by a process on an object, or the response of such an operation.
- An **invocation step** records the process ID, the object, the operation, and the argument of the operation (or ⊥ if there is no argument).
  - Example: The invocation of P1.R(x)a is denoted by INV(P1, x, R, ⊥).
  - Example: The invocation of P2.W(x)b is denoted by INV(P2, x, W, b).
- A response **step** records the process ID, the object, the operation, and the return value of the operation (or ⊥ if there is no return value).
  - Example: The response of P1.R(x)a is denoted by RES(P1, x, R, a).
  - Example: The response of P2.W(x)b is denoted by RES(P2, x, W, ⊥).
- A response step is **matching** for an invocation step if they refer to the same process, the same object, and the same operation (e.g., R vs. W).

# How to formalize linearizability

- An operation Op1 **happens before** operation Op2 in a history H if the response step of Op1 precedes the invocation step of Op2 in H.
- A history H is **sequential** if
  1. it is empty, or
  2. it starts with an invocation step, and every invocation step is followed immediately by a matching response step, and every response step except possibly the last one is followed immediately by an invocation step.
- A sequential history S of read and write operations is **legal** if every read returns the value assigned by the most recent write on the same object, or else the read returns the initial value of the object if no write of this object precedes the read.
- The **projection** of a history H onto process p, denoted by H|p, is the maximal subsequence of H comprising only steps taken by p.
- Two histories H and H' are **equivalent** if, for every process p, H|p = H'|p.

# How to formalize linearizability

**Simplifying assumption for ECE454:** For every history H, and every operation Op in H, Op has a matching response in H (i.e., no incomplete or "pending" operations are allowed).

**Definition:** A history H is **linearizable** if there exists a legal sequential history S such that:

1. H and S are equivalent, and
2. for every pair of operations Op1,Op2 appearing in H,
   if Op1 happens before Op2 in H then Op1 precedes Op2 in S.