THE UNIVERSITY
of EDINBURGH

Distributed Systems
Fall 2024

Yuvraj Patel

Today’s Agenda

Transactions

Replication

- General stuff
- Data-Centric Consistency Models

Next Class Monday(4/11)

Transactions

Series of operations executed by clients

Operations may be locally executed or via an RPC to a server

Transactions either commits or aborts

- Commit – An operation completes and reflect updates on server-side data
- Abort – An operation fails/aborts and has no effect on the server

Transactions

Series of operations executed by clients

Operations may be locally executed or via an RPC to a server

Transactions either commits or aborts

- Commit – An operation completes and reflect updates on server-side data
- Abort – An operation fails/aborts and has no effect on the server

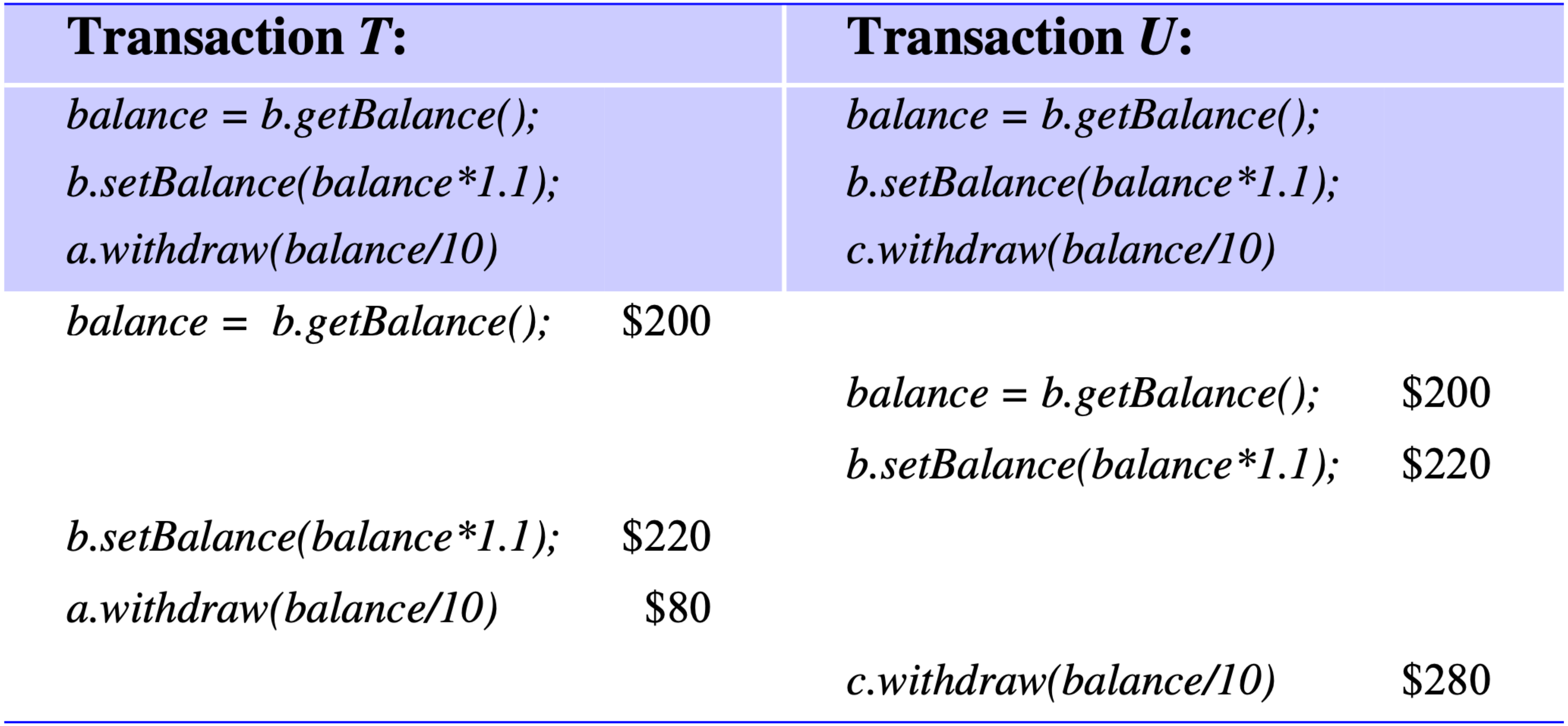
```
int transaction_id = transaction_start()

curr_balance = server.getbalance(“XYZA”);
If (curr_balance > transfer_amount)
        server.withdraw(“XYZA”, curr_balance
                      – transfer_amount);
        server.deposit(“ABCD”,
                    transfer_amount);

transaction_close()
```

ACID Properties

All transactions adhere to ACID Properties

- Atomicity – All or Nothing
  - Transaction either commits or aborts
- Consistency – Follow the Rules
  - Transaction does not violate system invariants
- Isolation – Mind Your Own Business
  - Concurrent transactions do not interfere with each other
- Persistence – Remember Everything
  - Once a transaction commits, the changes are permanent

Issues with Transactions – Lost-Update



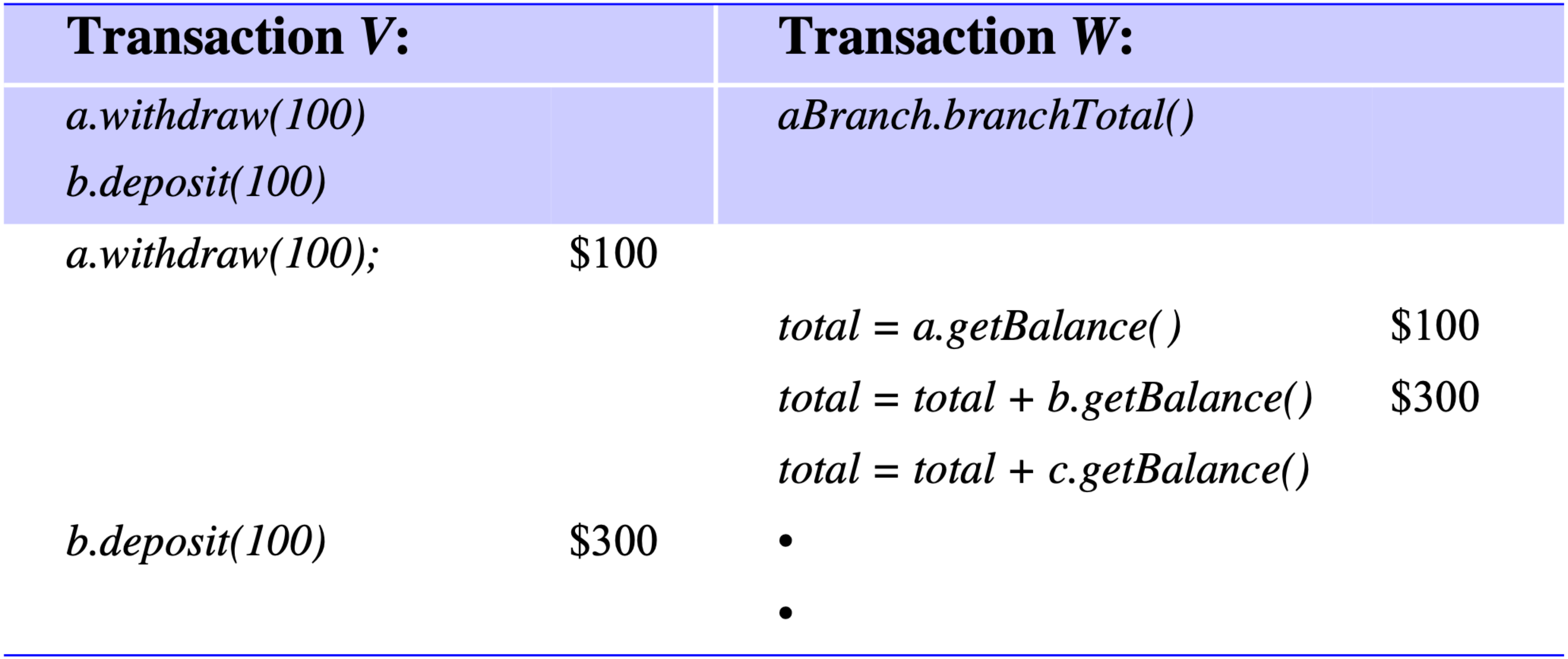
Transaction *T*:		**Transaction *U*:**	

balance = b.getBalance();
*b.setBalance(balance*1.1);*
a.withdraw(balance/10)
		balance = b.getBalance();
		*b.setBalance(balance*1.1);*
		c.withdraw(balance/10)	

balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Balance
A = $100
B = $200
C = $300

Issue with Transactions – Inconsistent Retrieval



Transaction *V*:		**Transaction *W*:**	

a.withdraw(100)
b.deposit(100)
		aBranch.branchTotal()	

a.withdraw(100);	$100		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$300
		total = total + c.getBalance()	
b.deposit(100)	$300	•	
		•	

Balance
A = $200
B = $200
C = $200

Concurrent Transactions

Multiple transactions execute concurrently in real-world

To prevent transaction from affecting each other

- Serially execute transactions one at a time
    - Slow; Not efficient;
    - Would you be a customer of such a slow service? 🙄

Ideally, we want to increase concurrency while maintaining ACID properties

Serial Equivalence Interleaving

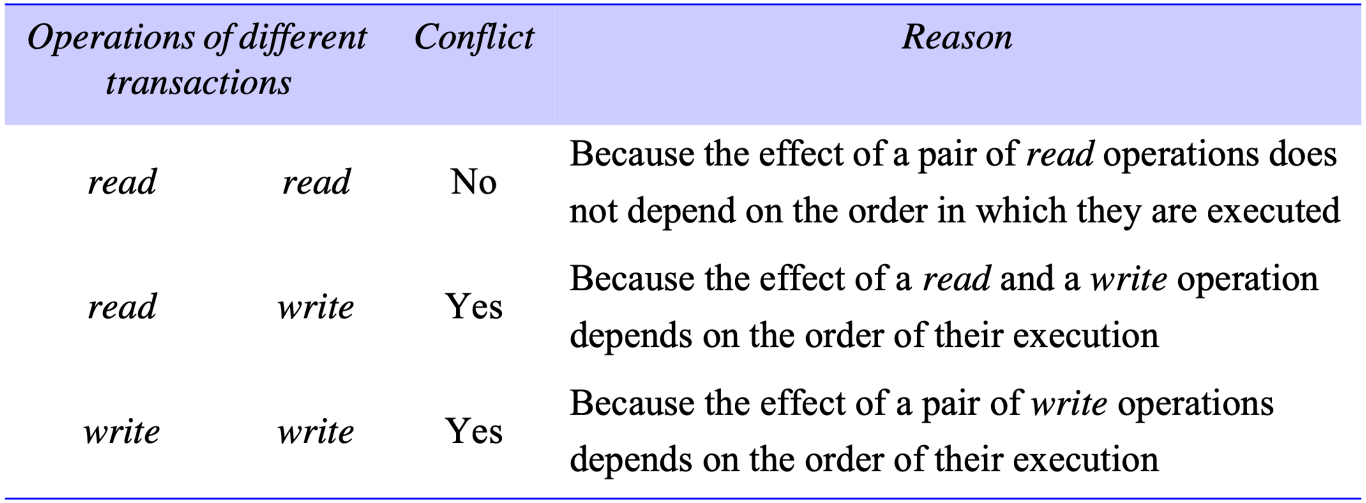
If each of several transactions is known to have the correct effect when it is done on its own, then we can infer that if these transactions are done one at a time in some order the combined effect will also be correct.

Serially Equivalent Interleaving – An interleaving of the operations of transactions in which the combined effect is the same as if the transactions had been performed one at a time in some order.

Conflicting Operations

A pair of operations conflicts means the combined effect depends on the other in which they are executed

Conflict rules for read and write



Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Resolving conflicts

Reactive approach – check for serial equivalence at commit time with all other transactions

- Only bother about overlapping transactions

If not serially equivalent

- Abort the transaction

Can we do better?

- Prevent violations from occurring

Two approaches – Pessimistic and Optimistic

Pessimistic vs. Optimistic

Pessimistic: Assume the worst; prevent transactions from accessing the same objects

- Better when data is updated/written frequently
- Use locks for exclusive access
- Use Reader-Writer Locks to improve performance; Readers can run concurrently; Writers have exclusive access

Optimistic: Assume the best; allow transactions to proceed, but check later

- Better when data is not updated frequently
- Less chances of aborting the transactions
- Multiple ways – Timestamp Ordering, Multi-version Concurrency Control

Distributed Transactions

In a distributed transaction, multiple objects residing on different servers involved

During commit

- Need to ensure all servers commit their corresponding update
- If one server fails to commit, everyone aborts; Transaction abort happens
- Like consensus problem – everyone agrees for a commit or abort

One Phase Commit

One Phase Commit

Problems

- Server with objects has no say in the decision making
- Issues like deadlock prevention handling, server crash, etc. could happen forcing server to abort
- Need a better way

Two Phase Commit

Replication

Replicate data at one or more sites can help with

- Availability & Fault Tolerance
  - If primary server crashes, secondary can takeover => Highly available service
  - Mask node crashes => Transparency
- Performance
  - Local access faster than remote access; Low latency
  - Concurrent Reads can be served from multiple servers improving performance
- Scaling
  - Size scalability – Prevent overloading a single server
  - Geographical scalability

Problems with Replication

Having multiple copies, means that when any copy changes, the change needs to be propagated to all other copies

Need replicas to have same data, i.e., they should be kept consistent

Efficiently synchronize all replicas a challenging problem

Performance & Scalability

Main concern – To keep replicas consistent, we generally need to ensure that all conflicting operations are done in the same order, across all servers

Conflicting operations – Recall the read-write and write-write conflicts

Guaranteeing global ordering on conflicting operations may be costly operation, with impact on scalability

Potential Solution – Weaker consistency requirements to avoid global synchronization, whenever possible

Weakening Consistency Requirements

What does it mean to weaken consistency requirements?

- Relax the requirement that “updates need to be executed as atomic operations”
- Do not require global synchronizations
- Replicas may not always be the same everywhere and everytime

To what extent can consistency be weakened?

- Depends highly on the access and update patterns of the replicas
- Depends on the replicated data user patterns which is application driven

Consistency Models

A consistency model is a contract between the programmer and a system

- The system guarantees that if the programmer follows the rules for operations on data, data will be consistent
- Result of the reading, writing, updating data will be predictable

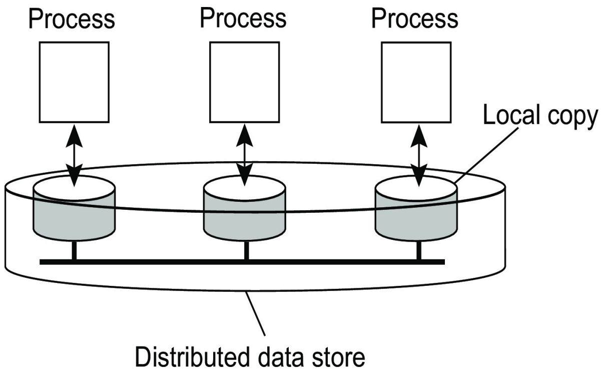
Two consistency models

- Data-centric consistency models – Defines consistency as experienced by all the clients; provides a system wide consistent view on the data store
- Client-centric consistency models – Defines consistency of the data store only from one client’s perspective; Different clients might see different sequences of operations at their replicas

Distributed Data Store

Distributed Data Store – Physically distributed & replicated across multiple machines

- Data can be read from or written by any process on any node
- A local copy helps with faster reads
- A write to a local replica needs to be propagated to all remote replicas

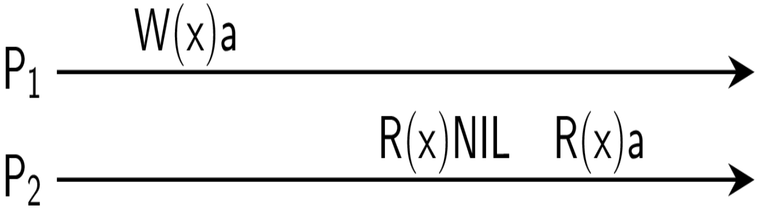


Terminology & Notations

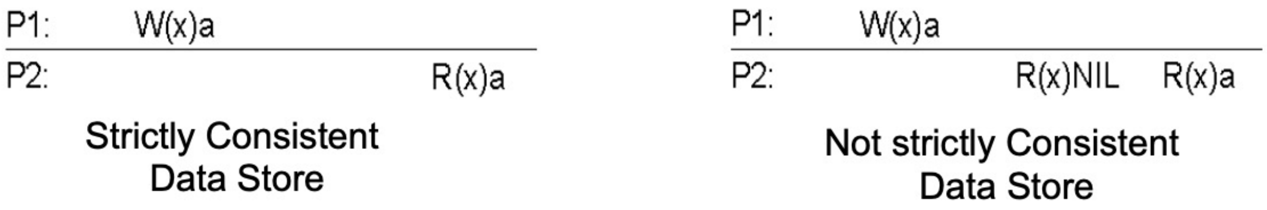
Read and write operations

- W_i(x)a: Process P_i writes value a to x
- R_i(x)b: Process P_i reads value b from x
- All data items initially have value NIL

Possible behavior represented over time; time moves from left to right



Strict Consistency



P1: W(x)a

P2: R(x)a

Strictly Consistent Data Store

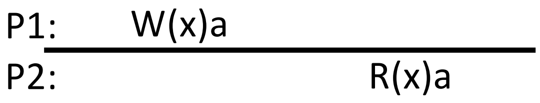
P1: W(x)a

P2: R(x)NIL R(x)a

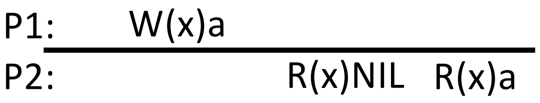
Not strictly Consistent Data Store

With strict consistency, all writes are visible instantaneously to all processes

Any read to a shared data item returns the value stored by the most recent write operation on that data item



Strictly Consistent Data Store



Not Strictly Consistent Data Store

Strictest consistency model – most rigid model

Practical relevance restricted to a thought experiment and formalism

- Relies on absolute global time
- Instantaneous message exchange is impossible

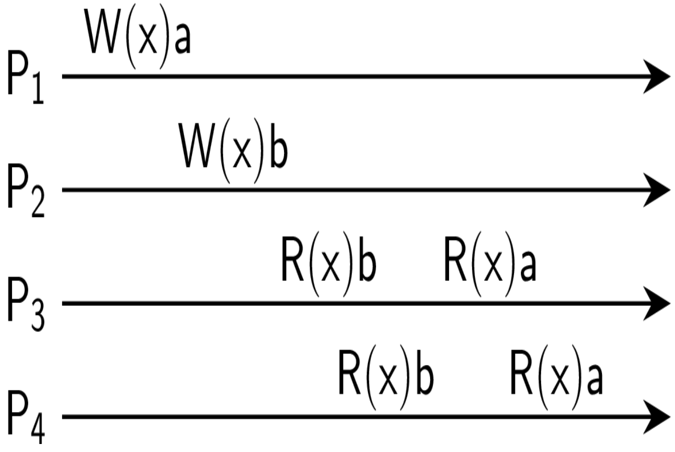
Sequential Consistency

Sequential Consistency – The result of any execution is the same as if the operations by 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

Any valid interleaving of read or write operations is fine, but all processes must see the same interleaving

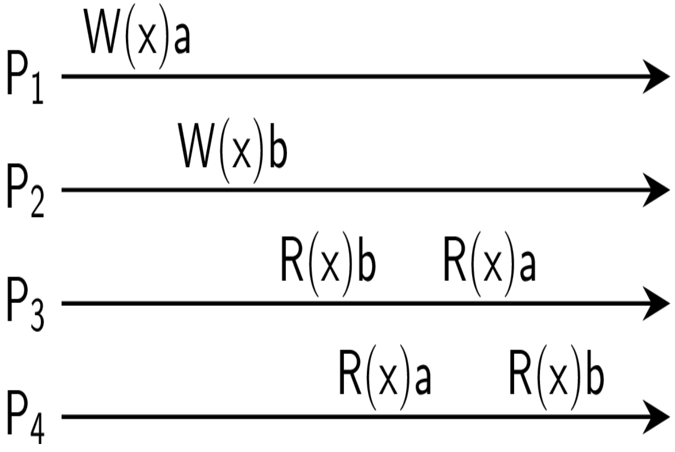
- The events observed by each process must globally occur in the same order, or it is not sequentially consistent

Sequential Consistency Example



A sequentially consistent data store

P3 and P4 see the same interleaving of writes



A data store that is not sequentially consistent

P3 and P4 do not see the same interleaving of writes

Sequential Consistency Example

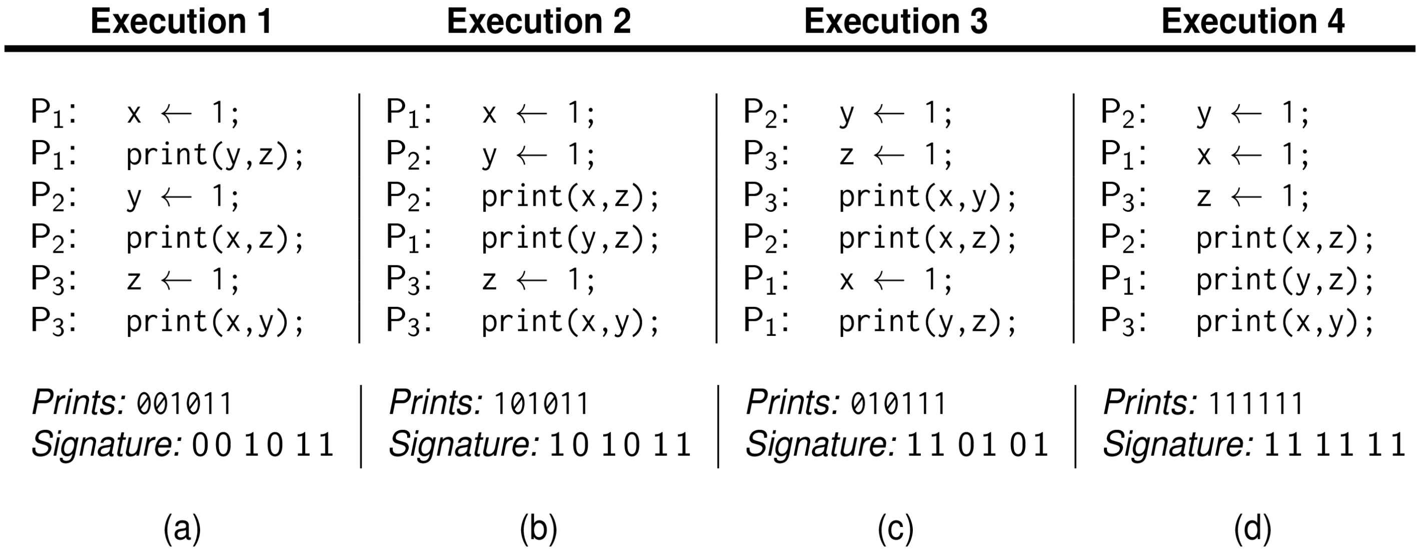
Three concurrent processes, executing concurrently (initial values: 0)

Process 1	Process 2	Process 3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

The 6 statements shown can be ordered in 6! = 720 possible ways, with most orderings are invalid

Analysis shows only 90 possible valid execution sequences exist

Sequential Consistency – Interleaved Execution Sequence



Execution 1	Execution 2	Execution 3	Execution 4
P_1: x ← 1;	P_1: x ← 1;	P_2: y ← 1;	P_2: y ← 1;
P_1: print(y,z);	P_2: y ← 1;	P_3: z ← 1;	P_1: x ← 1;
P_2: y ← 1;	P_2: print(x,z);	P_3: print(x,y);	P_3: z ← 1;
P_2: print(x,z);	P_1: print(y,z);	P_2: print(x,z);	P_2: print(x,z);
P_3: z ← 1;	P_3: z ← 1;	P_1: x ← 1;	P_1: print(y,z);
P_3: print(x,y);	P_3: print(x,y);	P_1: print(y,z);	P_3: print(x,y);
Prints: 001011	*Prints:* 101011	*Prints:* 010111	*Prints:* 111111
Signature: 00 10 11	*Signature:* 10 10 11	*Signature:* 11 01 01	*Signature:* 11 11 11
(a)	(b)	(c)	(d)

The signature is the output of P1, P2, and P3, in that order

Signature can be used to determine whether a given execution sequence is valid

Linearizability

In sequential consistency, absolute time is somewhat irrelevant – the order of events is most important

Linearizability – Each operation should appear to take effect instantaneously at some moment between its start and completion

A data store is said to be linearizable when each operation is timestamped, and the following conditions hold:

- Sequential Consistency holds
- Timestamp($OP_1(x)$) < Timestamp($OP_2(x)$) then $OP_1(x)$ should precede $OP_2(x)$ in this sequence

Sequential Consistency vs. Linearizability

Linearizability is weaker than strict consistency, but stronger than sequential consistency

Linearizability cares about time; sequential consistency cares about program order

- With Sequential consistency, the system has freedom of how to interleave operations coming from different clients, as long as the ordering from each client is preserved
- With Linearizability, the interleaving across all clients is pretty much determined already based on the time

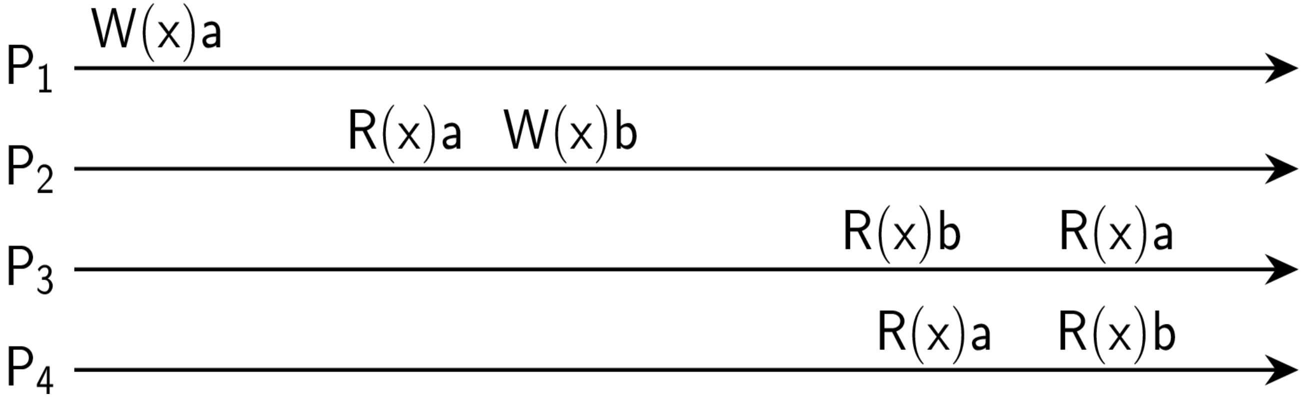
Causal Consistency

Writes that are potentially causally related must be seen by all processes in the same order

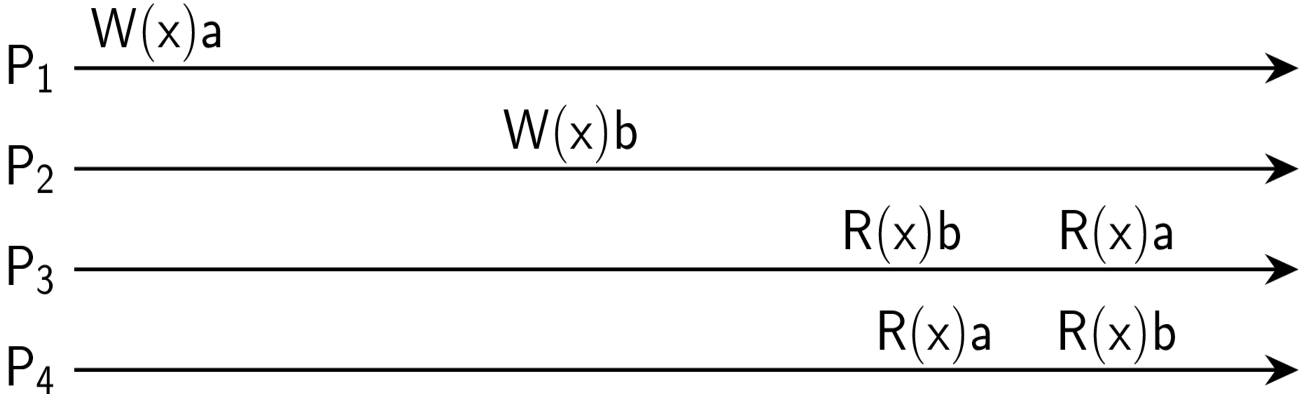
Concurrent writes may be seen in a different order on different machines

Example – If event A is a direct or indirect result of another prior event B, then all processes should observe event A before observing event B

Causal Consistency Example

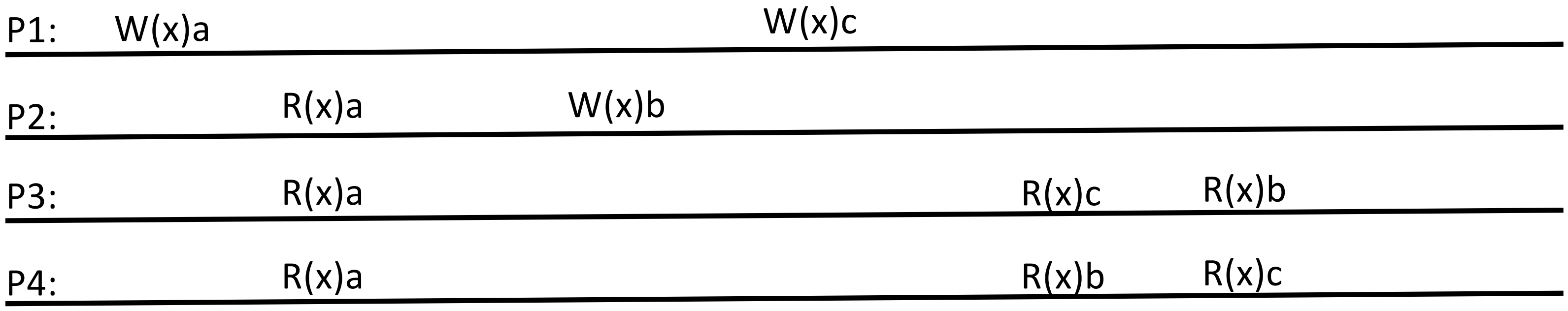


A violation of a causally-consistent store



A correct sequence of events in a causally-consistent store

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

Assume $W_2(x)b$ and $W_1(x)c$ are concurrent

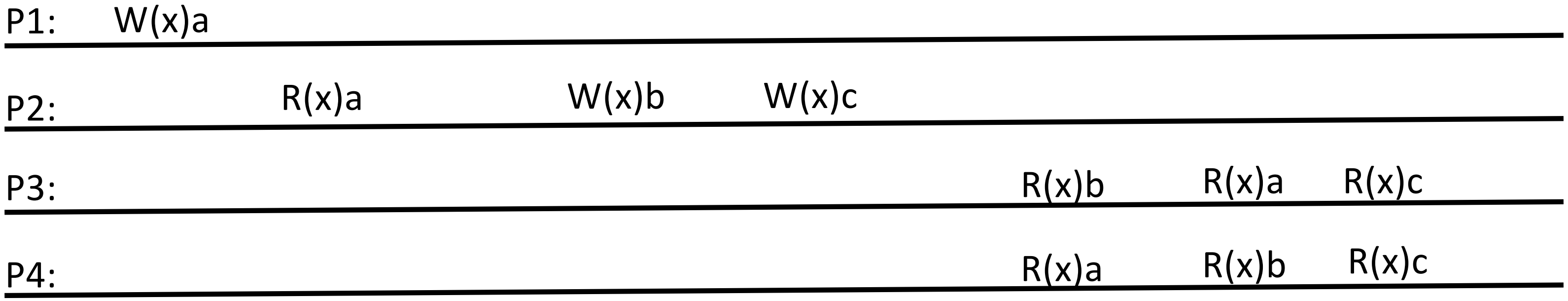
Strictly consistent?
Sequentially consistent?
Causally consistent?

FIFO Consistency

Writes performed by a single process are seen by all other processes in the order in which they were issued

Writes from different processes may be seen in a different order by different processes

FIFO consistency is easy to implement



P1: W(x)a

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

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

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

A valid sequence of events of FIFO consistency
(P2's writes are seen in the correct order)

Data-Centric Consistency -- Strong & Weak Models

Strong Consistency Models – Operations on shared data are synchronized; do not require synchronization operations

- Strict Consistency – Absolute time ordering of all shared accesses matters
- Sequential Consistency – All processes see all shared accesses in the same order
- Linearizability – Sequential Consistency + Operations are ordered according to a global time
- Causal Consistency – All processes see causally-related shared accesses in the same order
- FIFO Consistency – All processes see writes from each other in the order they were used

Weak Consistency Models – Synchronization occurs only when shared data is locked and unlocked; rely on synchronization operations

- Weak Consistency – Shared data can be counted on to be consistent only after a synchronization is done
- Release Consistency – Shared data are made consistent when a critical region is exited
- Entry Consistency – Shared data pertaining to a critical region are made consistent when a critical region is entered

Weaker the consistency models, the more scalable it is