3. Distributed Shared Data

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

3.A. Distributed Transactions

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

Contents

- **Introduction**
- Problems with concurrent transactions
- Concurrency control
- Distributed transactions

Introduction

- Provide atomicity and isolation to a group of operations at a server in the presence of multiple clients and process crashes
- Properties - ACID
  - Atomicity: either all operations are completed or none of them is executed
  - Consistency: takes the system from one consistent state to another (this is an application concern)
  - Isolation: updates of one transaction are not visible to other transactions until it commits
  - Durability: persistent once completed successfully

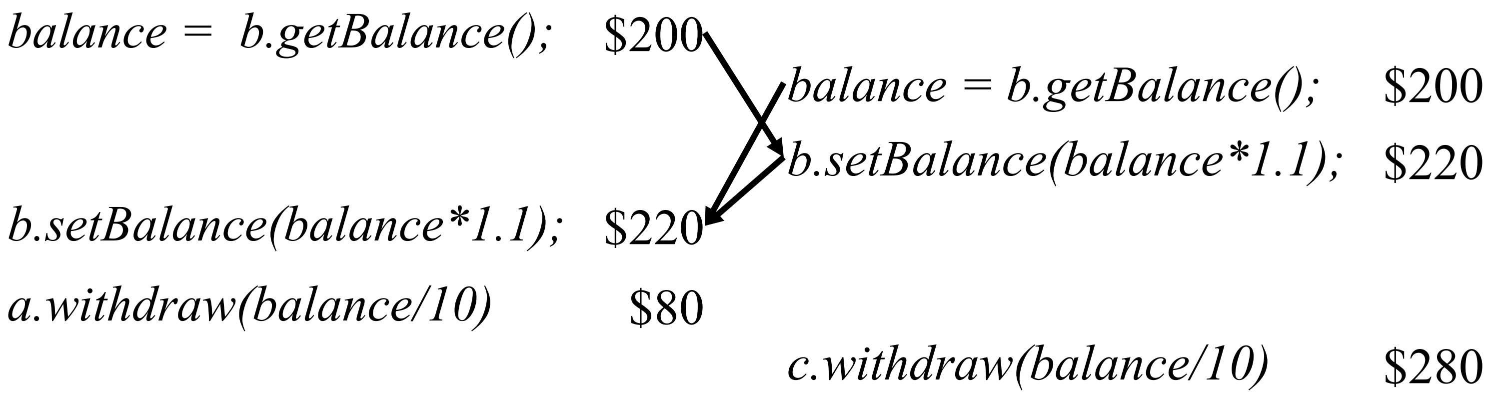
Contents

- Introduction
- **Problems with concurrent transactions**
- Concurrency control
- Distributed transactions

Lost update problem

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

initially, a=100, b=200, c=300



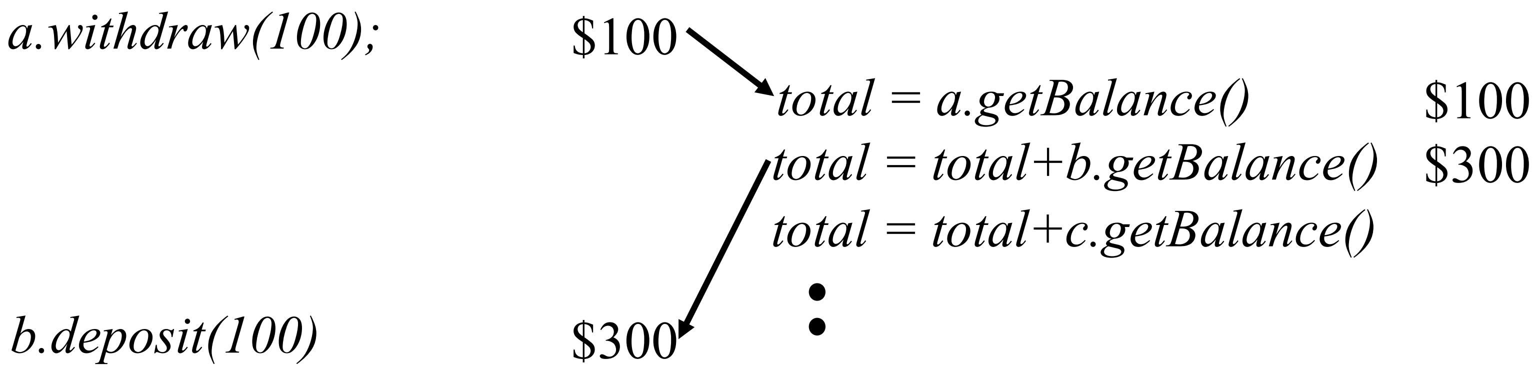
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

- U’s update on ‘b’ is lost as T overwrites ‘b’ without seeing it

Inconsistent retrievals problem

Transaction V:	**Transaction W:**
a.withdraw(100) *b.deposit(100)*	*aBranch.branchTotal()*

initially, a=200, b=200



a.withdraw(100); $100

total = a.getBalance() $100

total = total+b.getBalance() $300

total = total+c.getBalance()

:

b.deposit(100) $300

- W's retrievals are inconsistent because V has only performed the withdrawal at the time the sum is calculated

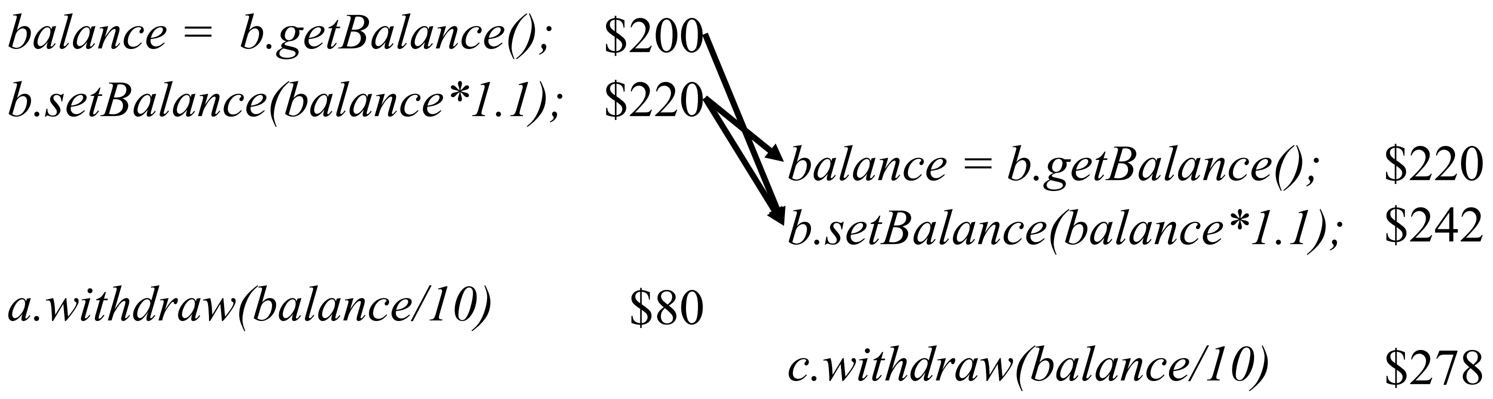
Concurrency control

- Allow concurrent transactions to be executed correctly, i.e. preserving **serial equivalence**
  - The outcome is the same as if the transactions had been performed one at a time in some order
- Two transactions are serially equivalent if all pairs of conflicting operations are executed in the same order at all the shared objects
  - i.e. the same transaction operates always first
  - Two operations are conflicting if their effect depends on the order in which they are executed
    - read-write, write-write

Lost update revisited

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)*

initially, a=100, b=200, c=300



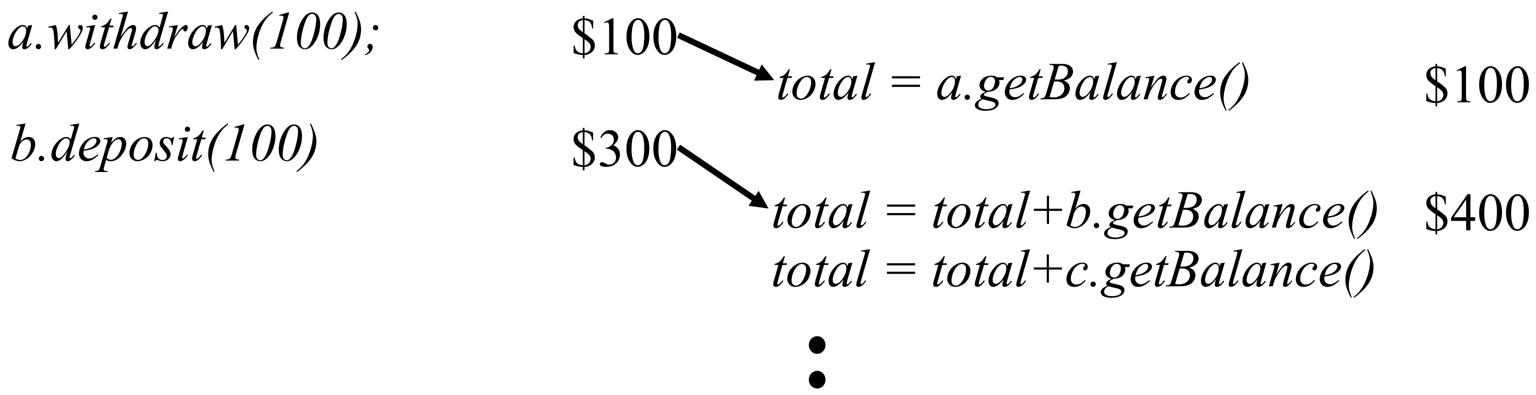
balance = b.getBalance();	$200		
*b.setBalance(balance*1.1);*	$220		
		balance = b.getBalance();	$220
		*b.setBalance(balance*1.1);*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

- A serially equivalent interleaving of T and U

Inconsistent retrievals revisited

Transaction V:	**Transaction W:**
a.withdraw(100) *b.deposit(100)*	*aBranch.branchTotal()*

initially, a=200, b=200



a.withdraw(100); $100

total = a.getBalance() $100

b.deposit(100) $300

total = total+b.getBalance() $400

total = total+c.getBalance()

:

- A serially equivalent interleaving of V and W

Dirty read problem

Transaction T:		**Transaction U:**	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

- U has seen a value that never existed

Recoverability from aborts

a) OPTION A: Delay the commit operation until earlier transactions that wrote the same objects have committed/aborted
  - e.g. U must delay its commit until T commits
  - If earlier transaction aborts, delayed transaction must also abort
    - e.g. if T aborts, then U must abort as well
  - This may cause **cascading aborts**

b) OPTION B: Delay any read operation until earlier transactions that wrote the same object have committed/aborted

Premature write problem

Transaction T:		**Transaction U:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110
		commit transaction	
abort transaction			

- When T aborts, ‘a’ is reverted to an incorrect value
- Delay any write operation until earlier transactions that wrote the same object have committed/aborted

Recoverability from aborts

- To avoid dirty reads and premature writes, we require **strict execution** of transactions
  - Both read and write operations on an object must be delayed until earlier transactions that wrote that object have been aborted or committed
- Use tentative versions to allow recoverability
  - All the updates performed during a transaction are done in tentative versions of the objects
  - Read from tentative versions if possible
  - Updates are transferred to the objects only when a transaction commits, and ignored if it aborts

Contents

- Introduction
- Problems with concurrent transactions
- **Concurrency control**
- Distributed transactions

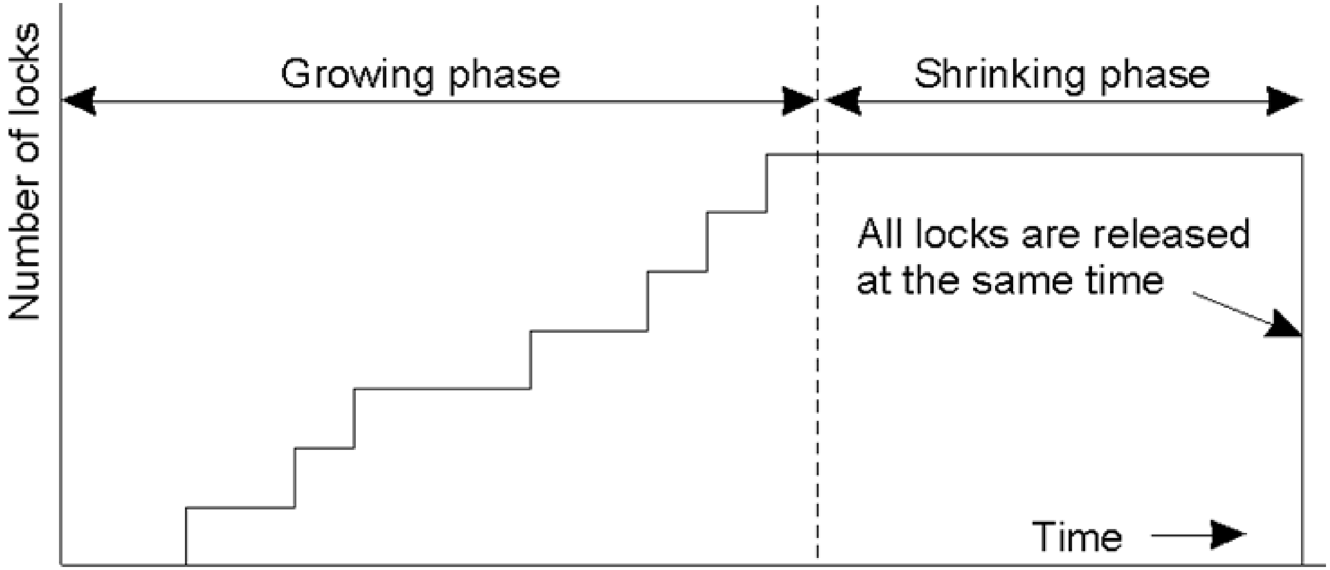
Concurrency control

- Schedule concurrent transactions so that they execute preserving **serial equivalence**
  - Allow as much concurrency as possible
- Algorithms can use different options to schedule an operation within a transaction
  - A) Execute it, B) Delay it, C) Abort it

1. Strict two-phase locking
2. Timestamp ordering
3. Optimistic concurrency control

Strict two-phase locking

- Serialize access to the objects using locks



- A transaction is not allowed new locks after it has released a lock to get serial equivalence
- Locks must be held until the transaction commits or aborts to get strict executions

Strict two-phase locking

Transaction T:	**Transaction U:**
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*	*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*

Operations	**Locks**	**Operations**	**Locks**
openTransaction			
bal = b.getBalance()	lock B		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock A	*bal = b.getBalance()*	wait for T's lock on B
closeTransaction	unlock A,B	...	
			lock B
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock C
		closeTransaction	unlock B,C

Strict two-phase locking

- How to increase the concurrency of locking?
  - Lock just the objects involved in the operations
  - Use different locks for read and write access
    - Multiple transactions can take read locks but only if the write lock is not taken
    - Only one transaction can take a write lock but only if the read and write locks are not taken
    - Read locks can be promoted to write locks (if not shared)

for one object		*lock requested*	
		read	*write*
lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Strict two-phase locking

1. When a read or write operation accesses an object within a transaction:
    a) If the object is not already locked, it is locked and the operation proceeds
    b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked
    c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds
    d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds (where promotion is prevented by a conflicting lock, rule b is used)
2. When a transaction commits or aborts, the server unlocks all objects it locked for the transaction

Strict two-phase locking

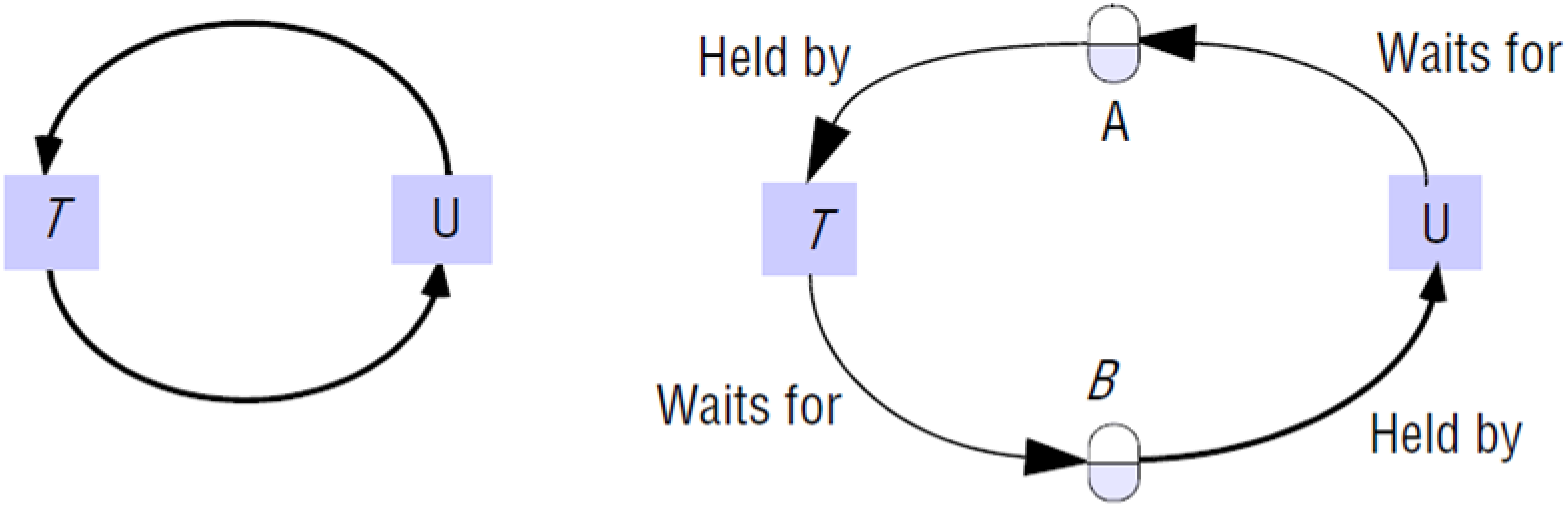
↓ Deadlocks can occur!

Transaction T		**Transaction U**	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100)	write lock A		
		b.deposit(200)	write lock B
b.withdraw(100)	wait for U's lock on B		
		a.withdraw(200)	wait for T's lock on A

- Deadlocks could be prevented taking locks all at once when transaction starts but this reduces concurrency!

Deadlock detection

- A **wait-for graph** can be used to represent the waiting relations between transactions
- Check for **cycles** to detect deadlocks
    a) Each time an edge is added (i.e. a lock is taken)
    b) Less frequently to avoid unnecessary overhead



Deadlock resolution

a) Abort one transaction involved in the cycle

- Which transaction should be aborted?
  - Consider the age of the transaction and the number of cycles in which it is involved

b) Lock timeouts

- A taken lock is made **vulnerable** after a timeout
- Lock is broken if any other transaction is waiting
- Transaction with the broken lock is aborted

↓ Unnecessary aborts: sometimes, other transactions are waiting, but there is no deadlock

↓ How to choose an adequate timeout value?

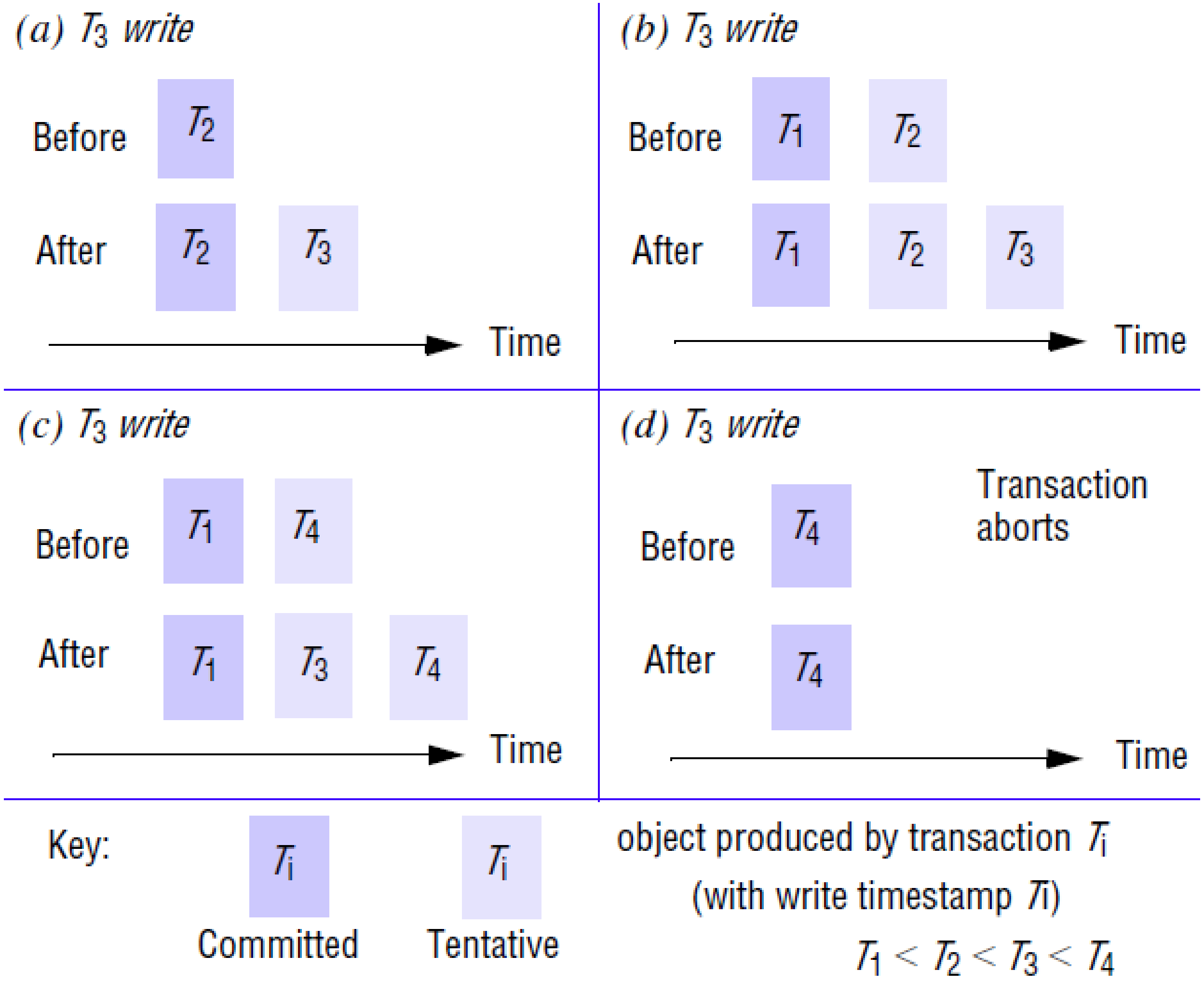
Timestamp ordering

- Each transaction has a unique timestamp
  - Defines its position in the sequence of transactions
  - Assigned when the transaction starts
- Each object keeps:
  1. A write timestamp for its committed value
  2. The set of tentative (not committed) versions with their corresponding timestamps
  3. The set of read timestamps, which can be represented by its maximum member
- Operations are validated when performed by comparing object and transaction timestamps

Timestamp ordering

- Transaction T performs a **write** operation
  - Valid only if the object was last read and written by earlier transactions
    - (T ≥ maximum read timestamp) && (T > write timestamp on committed version)
  - Implies creating a new tentative version with the timestamp set to the one of transaction T
    - Overwrite if it already exists
    - Tentative versions are kept ordered by their timestamps
  - If write is not valid, transaction T is aborted
    - A transaction with a later timestamp has already read or written the object

Timestamp ordering



T3 >= max read timestamp

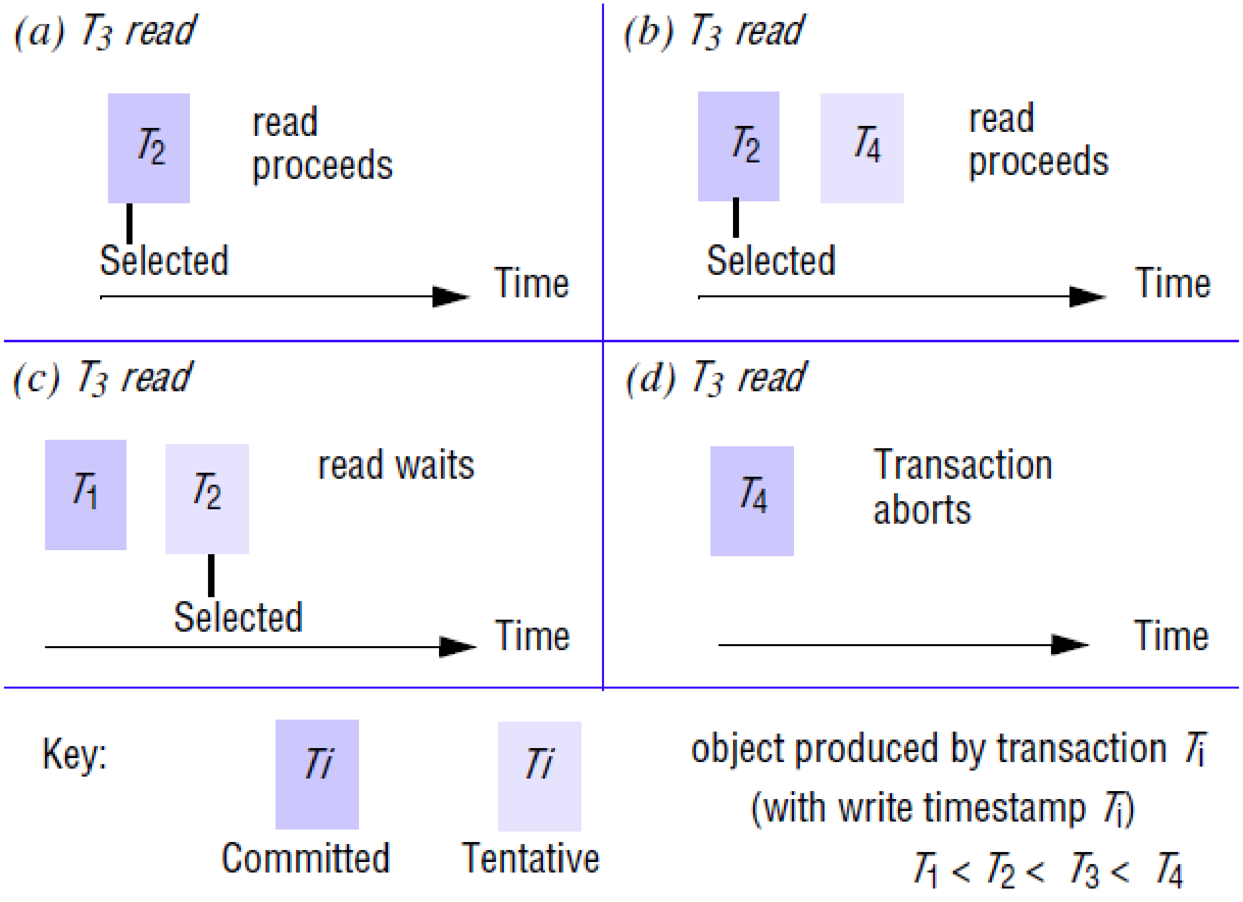
(a)-(c) T3 > write timestamp committed version

(d) T3 < write timestamp committed version

Timestamp ordering

- Transaction T performs a **read** operation
  - Valid only if the object was last written by an earlier transaction
    - T > write timestamp on committed version
  - Directed to the version with maximum write timestamp less than the transaction one
    - If this is still tentative, transaction T must wait the earlier transaction to complete
    - If T has already written the object, this will be used
  - If read is not valid, transaction T is aborted
    - A transaction with a later timestamp has already written the object

Timestamp ordering



(a)-(c) T3 > write timestamp committed version

(a)-(b) read directed to a committed version

(c) read directed to a tentative version: must wait

(d) T3 < write timestamp committed version

Optimistic concurrency control

- Based on the premise that most transactions do not conflict
- Transactions proceed in three phases:
    1. Working phase: Operations proceed as there are no conflicts, but writing to a private workspace
    2. Validation phase: On transaction commit, check if it has conflicted with concurrent transactions
    3. Update phase: Tentative versions of any write operations are copied to the database

Optimistic concurrency control

1. Working phase
   - Transaction keeps a <u>tentative</u> version of each of the objects that it updates
     - This allows the transaction to abort with no effect on the objects
   - Transaction keeps also the <u>read and write set</u>
     - read set records the objects read by the transaction
     - write set records the objects written by the transaction
   - Read operations directed to the tentative version if exists, otherwise to the committed one
   - Write operations only onto tentative version

Optimistic concurrency control

2. Validation phase
   - A transaction is given a sequence number when entering the validation phase, but real assignment is postponed until successful validation and update
   - On transaction T_v commit, check for conflicts with **overlapping** T_i transactions (those not yet committed at the start of T_v)
   - If the transaction is valid, it is allowed to commit
   - Otherwise, abort the transaction
     - Or perform some conflict resolution (e.g. Amazon Dynamo, Wikipedia, file synchronizers such as Dropbox or SVN)

Optimistic concurrency control

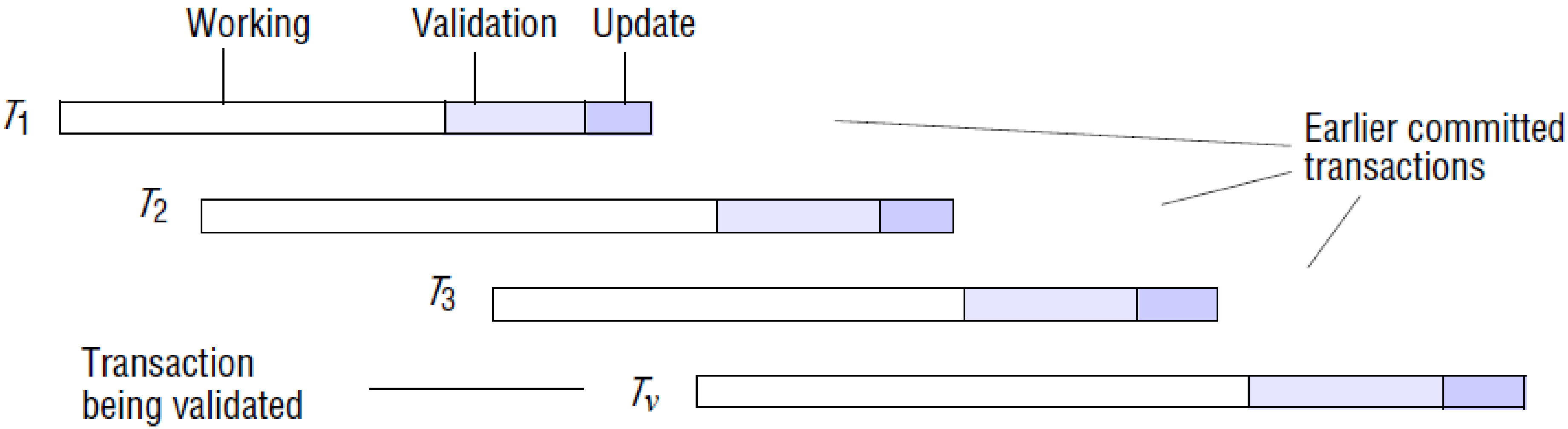
A. Backward validation

- Validate a transaction T_v by comparing its read set with the write set of (committed) transactions T_i with sequence number $T_{start} < T_i \leq T_{end}$
    - T_{start} is the biggest sequence number assigned when transaction T_v enters the working phase
    - T_{end} is the biggest sequence number assigned when transaction T_v enters the validation phase
- If a conflict is detected, abort **this** transaction

↓ All the write sets of overlapping transactions have to be kept until their last concurrent transaction has finished

Optimistic concurrency control

A. Backward validation



- The read set of T_v must be compared with the write sets of T_2 and T_3

Optimistic concurrency control

B. Forward validation

- Validate a transaction T_v by comparing its write set with the read set of overlapping active (uncommitted) transactions

↑ Flexible in conflict resolution

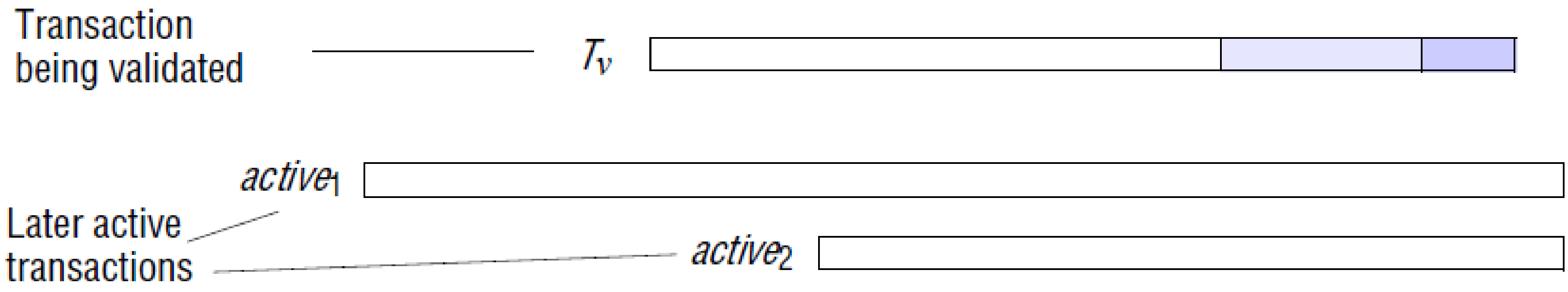
  a) Abort the transaction being validated
  b) Abort the conflicting active transactions
  c) Let the conflicting transactions finish and try again
    - One of them might abort …

↓ Dynamic read sets must be consistently checked

  - e.g. block reads of active transactions on contended objects while doing the validation and update phases

Optimistic concurrency control

B. Forward validation



- The write set of transaction T_v must be compared with the read sets of the transactions $active_1$ and $active_2$

SEMINAR PREPARATION – Opty

- **[Kung81]** Kung, H.T., Robinson, J.T., *On Optimistic Methods for Concurrency Control*, ACM Transactions on Database Systems, Vol. 6, No. 2, pp. 213-226, June 1981
- **[Harder84]** Harder, T., *Observations on Optimistic Concurrency Control Schemes*, Information Systems, Vol. 9, No. 2, pp. 111-120, November 1984

Comparing concurrency control methods

Two-phase Locking

- Pessimistic: conflicts detected (& serialization order decided) as objects are accessed and solved delaying the transaction
- Incurs overhead even if there is no conflict
- Risk of deadlocks: its prevention reduces concurrency
- Better when operations are mainly updates

Timestamp Ordering

- Pessimistic: conflicts detected as objects are accessed and solved aborting the transaction
- Serialization order decided when transaction starts
- Better when operations are mainly reads

Optimistic Conc Ctrl

- Optimistic: transaction proceeds normally, conflicts detected (& serialization order decided) when it tries to commit and solved aborting the transaction
- Better when there are few conflicts

Contents

- Introduction
- Problems with concurrent transactions
- Concurrency control
- **Distributed transactions**

Distributed transactions

- A transaction can access objects managed by multiple servers
  - For each transaction, one server acts as coordinator, being responsible of opening and closing (commit/abort) the transaction
  - Each of the servers accessed by a transaction is a participant

➢ All servers must commit the transaction or all of them must abort → **distributed commit**

➢ How to schedule operations to guarantee serial equivalence? → **concurrency control**

Contents

- Introduction
- Problems with concurrent transactions
- Concurrency control
- **Distributed transactions**
  - **Distributed commit**
  - Concurrency control

Distributed commit

- To ensure atomicity, all the servers accessed by a transaction must agree on the final outcome of the execution (commit/abort)
- One-phase commit is not feasible
  - Coordinator tells all participants to commit
  - If a participant needs to abort (due to concurrency control issues), it cannot inform the coordinator
- Two-phase commit protocol (2PC) is used
  - It is a consensus protocol, but …
    - All participants must vote and reach the same decision
    - If any participant votes to abort, all must abort

Two-phase commit protocol (2PC)

- Targets asynchronous systems in which servers may crash and messages may be lost
  - Remember that consensus cannot be guaranteed under those conditions, but can be achieved with some probability
  - Mask crash failures by recovering the state of the crashed process from permanent storage
    - Once a participant has voted to commit, it must ensure that it will eventually be able to carry out the commit
- Initiated when the client requests to commit
  - If requests to abort, coordinator can abort directly

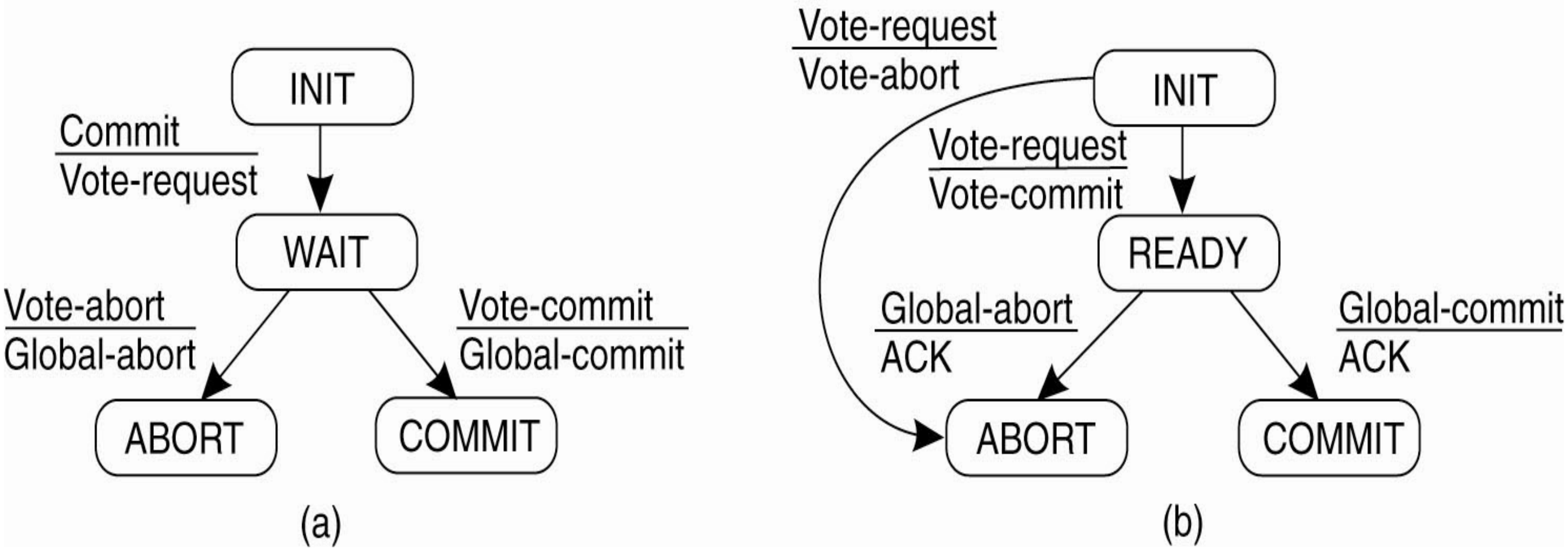
Two-phase commit protocol (2PC)

- Coordinator sends Vote-request to all the participants (including itself)
- Each participant replies Vote-commit if it can commit locally, otherwise it replies Vote-abort and directly aborts the transaction
- Coordinator collects all the votes (including its own). If everyone voted to commit, it sends Global-commit. If someone voted to abort, it sends Global-abort
- Participants then COMMIT or ABORT according to the message and optionally send ACK when done

Two-phase commit protocol (2PC)

a) State diagram for the coordinator

b) State diagram for a participant



Two-phase commit protocol (2PC)

- Both coordinator and participants can block waiting for lost messages
  - Use timeouts and decide (if possible)
- Coordinator blocked in WAIT
  - Participant may have failed or vote message may have been lost
  - Coordinator timeouts and decides abort
    - Cannot commit since missing participant may vote abort
- Participant blocked in INIT
  - Participant timeouts and, as no decision has been made at this stage, it aborts

Two-phase commit protocol (2PC)

- Participant blocked in READY
  - – It voted commit, but cannot unilaterally decide
  - a) Decision message may have been lost
    - Solution: Ask the coordinator to resend the decision
  - b) Coordinator may have failed
    - Solution: Contact other participants and decide depending of their state
      - – COMMIT ⇒ COMMIT; ABORT, INIT ⇒ ABORT
    - If all participants are in READY, <u>they cannot decide and must wait until the coordinator recovers</u>

⇒ It is a <u>blocking</u> protocol (it is safe, but not live)

Distributed commit

- Despite its blocking nature on coordinator failure, 2PC is still a very popular consensus protocol, due to its low message complexity
- Three-phase commit protocol also exists
  - Avoids the blocking problem, but at the cost of adding one more step, resulting in higher latencies
  - It also falls short upon network partitions, so it is not applied often in practice
- Alternatively, 2PC can sit on top of Paxos

  ⇒ Paxos Commit

Contents

- Introduction
- Problems with concurrent transactions
- Concurrency control
- **Distributed transactions**
  - Distributed commit
  - **Concurrency control**

Concurrency control

- Schedule concurrent transactions so that they execute preserving serial equivalence
  a) Each server is responsible for concurrency control of its own objects
  b) All servers must agree on the same order of transactions to preserve serial equivalence
    - If transaction T is before U in their access to objects in one server then all servers should have T before U

1. Strict two-phase locking
2. Timestamp ordering
3. Optimistic concurrency control

Strict two-phase locking

- For each object, locks are maintained locally in its server
- Locks are held until transaction is committed or aborted **at all servers** involved

Strict two-phase locking

↓ Distributed deadlocks can happen!

objects A,B, and C are managed by servers X,Y, and Z, respectively

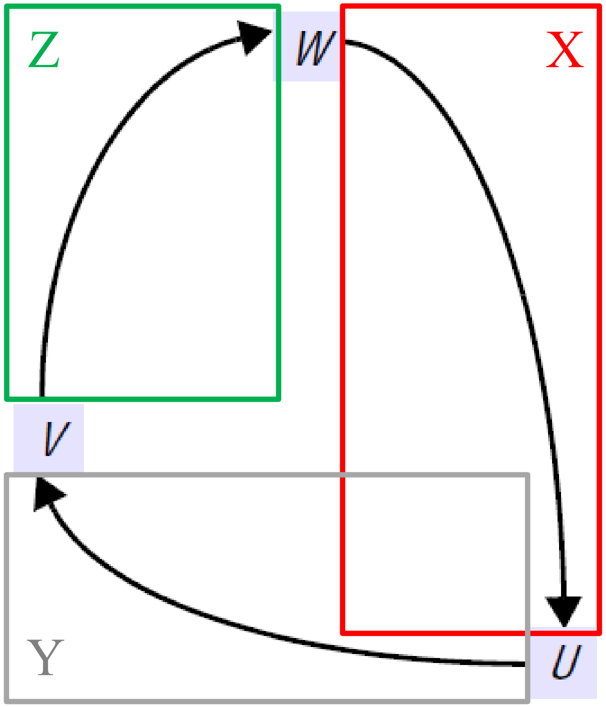
Transaction U		**Transaction V**		**Transaction W**	
Operations	**Locks**	**Operations**	**Locks**	**Operations**	**Locks**
		b.deposit(10)	lock B at Y		
a.deposit(20)	lock A at X				
				c.deposit(30)	lock C at Z
b.withdraw(30)	wait at Y for V's lock on B				
		c.withdraw(20)	wait at Z for W's lock on C		
				a.withdraw(20)	wait at X for U's lock on A

Distributed deadlock

A. Centralized detection

– A server builds a global wait-for graph from the local graphs

1. Looks for cycles in the global wait-for graph
2. Tells servers which transaction to abort when detects a cycle



↓ Poor scalability & fault tolerance

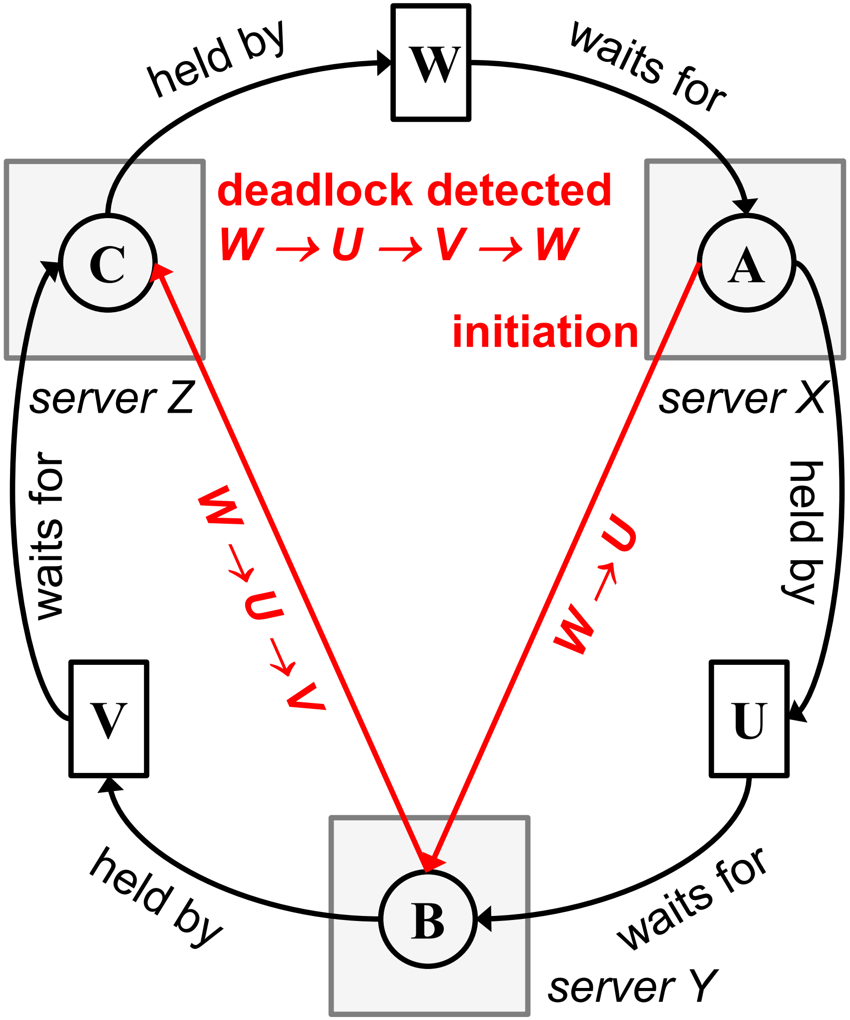
↓ How often should we collect local wait-for graphs?

- Cost of frequent transmission is high
- Less frequent transmission implies that deadlocks may take longer to be detected

Distributed deadlock

B. Edge chasing

- Global wait-for graph is not built: servers send probes which contain the edges detected so far
  - Probe is sent if a transaction is waiting for a lock held by another transaction that is also waiting for a lock
  - Status of each transaction is obtained from its coordinator
- Deadlock cycles tend to be small → paths are short



Timestamp ordering

- The coordinator of a transaction assigns it a <u>globally unique timestamp</u> when it starts
  - As different servers may act as coordinators, they must agree on the order for the timestamp they generate, e.g. <local timestamp, server-id>
- Locally to each server, protocol acts normally

Optimistic concurrency control

- A transaction is validated by the collection of servers managing the objects it has accessed
- Different servers may serialize the same set of transactions in different orders

  A. Perform local validation and then check if the combination of the orderings is serializable
     - Transaction being validated is not involved in a cycle

  B. Coordinator assigns to the transaction a global sequence number that all servers must use
     - As different servers may act as coordinators, they must agree on the order for the numbers they generate

Summary

- Transactions group sequences of operations into an ACID operation
- Problem is how to increase concurrency while preserving serial equivalence
  - Two-phase locking
  - Timestamp ordering
  - Optimistic concurrency control
- Commit protocols provide the transactions with distributed atomicity
- Further details:
  - [Tanenbaum]: chapter 8.5
  - [Coulouris]: chapters 16 and 17

3.B. Consistency and Replication

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

Contents

- **Introducing replication & consistency**
- Data-centric consistency models
- Client-centric consistency models
- Consistency protocols

Introduction

- Reasons for replication
  1. A. Increase **availability**: data is available despite server failures and network partitions
  2. B. Enhance **reliability**: data is correct on the presence of faults (e.g. protection against data corruption, Byzantine failures, and stale data)
  3. C. Improve **performance**: this directly supports the goal of enhanced **scalability**
     - Size: Replicate data and distribute work instead of having one single server
     - Geographical: Replicate data close to where it is used (e.g. data caching)

Introduction

- What are the issues with replication?
  - Replication should be **transparent**
    - Clients are not aware that multiple copies of data exist
  - Replicated data should be **consistent**
    - Clients see the same data despite the replica they read

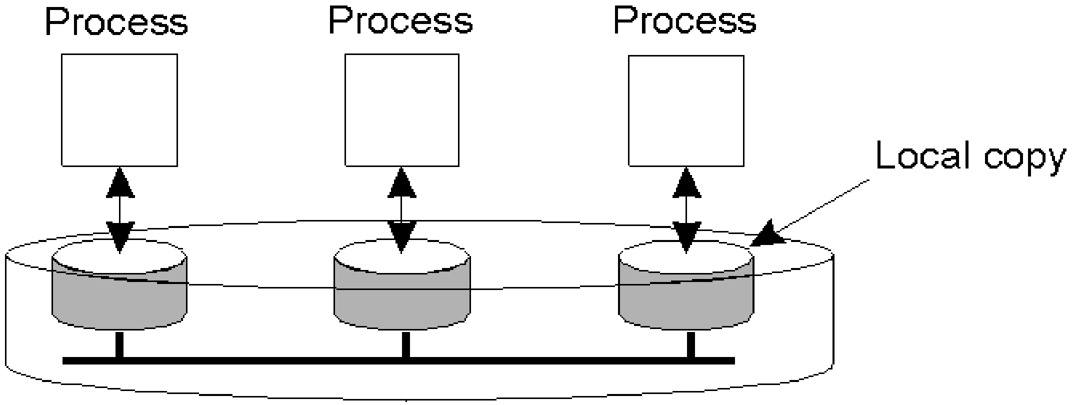
  ⇒ How do we transparently (and efficiently) keep all the replicas up-to-date and consistent?
- Dilemma: replication improves scalability, but incurs the overhead of synchronizing replicas
  - The solution often results in a relaxation of the consistency constraints

Contents

- Introducing replication & consistency
- **Data-centric consistency models**
- Client-centric consistency models
- Consistency protocols

Consistency models

- Consistency models in a data store
  - Each process has a local replica of the data store
  - Write operations to a local replica need to be propagated to all remote replicas
  - Model specifies precisely what the results of read and write operations are with concurrency



Consistency models

- Diagram notation
  - Time axis drawn horizontally with time increasing from left to right in all diagrams
  - $W_i(x)a$ – a write by process ‘i’ to item ‘x’ with a value ‘a’
    - i.e. ‘x’ is set to ‘a’
    - Note: The process is often shown as ‘P_i’
  - $R_i(x)b$ – a read by process ‘i’ from item ‘x’ returning the value ‘b’
    - i.e. reading ‘x’ returns ‘b’

Contents

- Introducing replication & consistency
- Data-centric consistency models
  - **Strong consistency models**
  - Relaxed consistency models
- Client-centric consistency models
- Consistency protocols

Strict consistency

- Definition:
  - – Any read on a data item ‘x’ returns the value of the **most recent** write on ‘x’, regardless of in which replica the write occurred
    - Assuming non-blocking read and write operations
- An absolute global time order is maintained
  - ↓ Writes must be instantaneously visible everywhere
    - Not feasible due to network latencies
  - ↓ Requires perfectly synchronized clocks
  - ↓ Needs timestamps with infinite precision to ensure that at most one operation occurs at a time

Strict consistency

(a)

(b)

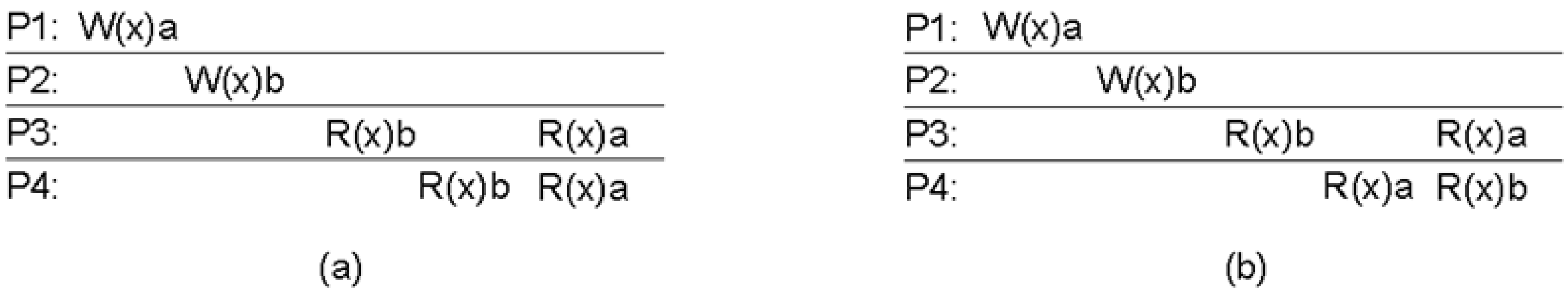
a) Data store with strict consistency
b) Data store that is not strictly consistent

- It is the ideal consistency model
  - Corresponds to true replication transparency
- But it is impossible to achieve within a distributed system

Sequential consistency

- Definition:
  - 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 the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program
- In other words: all processes see the same interleaving of operations, regardless of what that interleaving is
- Real time is not taken into consideration

Sequential consistency



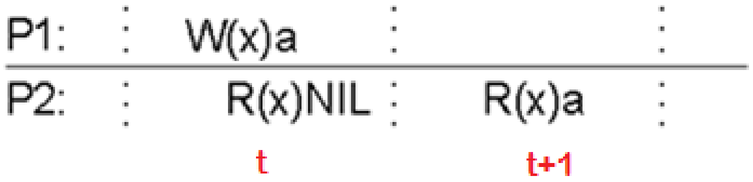
a) Data store with sequential consistency

b) Data store that is not sequentially consistent

- Weaker than strict consistency
- Easier to implement
  - Using total+FIFO multicast
  - See ‘Consistency protocols’ lesson for details

Linearizability (a.k.a. strong consistency)

- Operations receive a timestamp with a global available clock that is loosely synchronized
  - We have finite precision, thus two operations can be assigned with the same timestamp
- Def: sequential consistency + if ts(x) < ts(y) then op(x) precedes op(y) in the interleaving
  - strict > linearizability > sequential consistency
  - Can be implemented if clocks are synchronized with real time, and message delays are bounded



Other strong consistency models

- 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 by different processes
  - Implemented using causally-ordered multicast
- FIFO consistency
  - Writes done by a single process are seen by all the others in the order in which they were issued
    - Writes from different processes may be seen in a different order by different processes
  - Implemented using FIFO-ordered multicast

Contents

- Introducing replication & consistency

- **Data-centric consistency models**
  - Strong consistency models
  - **Relaxed consistency models**

- Client-centric consistency models

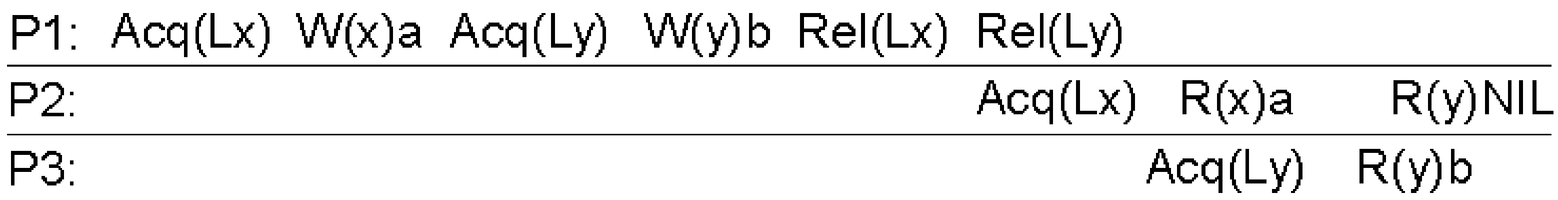
- Consistency protocols

Relaxed consistency models

- Not all the applications need to see all writes
- Weaker semantics by enforcing consistency on groups of operations (i.e. critical section) instead of individual reads and writes
  - Each process performs operations on its local copy of the data store
  - Process propagates only the results at the end of the critical section
    - Do not worry about propagating intermediate results
  - Critical section is delimited by means of **synchronization variables**

Relaxed consistency models

- Operate in the variable to synchronize all the copies in the data store
  - **Acquire**: local copy of the protected data is updated to be consistent with the remote ones
  - **Release**: protected data that have been changed are propagated out to the remote copies
- Relaxed models: weak consistency, release consistency, entry consistency


P1: Acq(Lx) W(x)a Acq(Ly) W(y)b Rel(Lx) Rel(Ly)

P2: Acq(Lx) R(x)a R(y)NIL

P3: Acq(Ly) R(y)b

Contents

- Introducing replication & consistency
- Data-centric consistency models
- **Client-centric consistency models**
- Consistency protocols

Eventual consistency

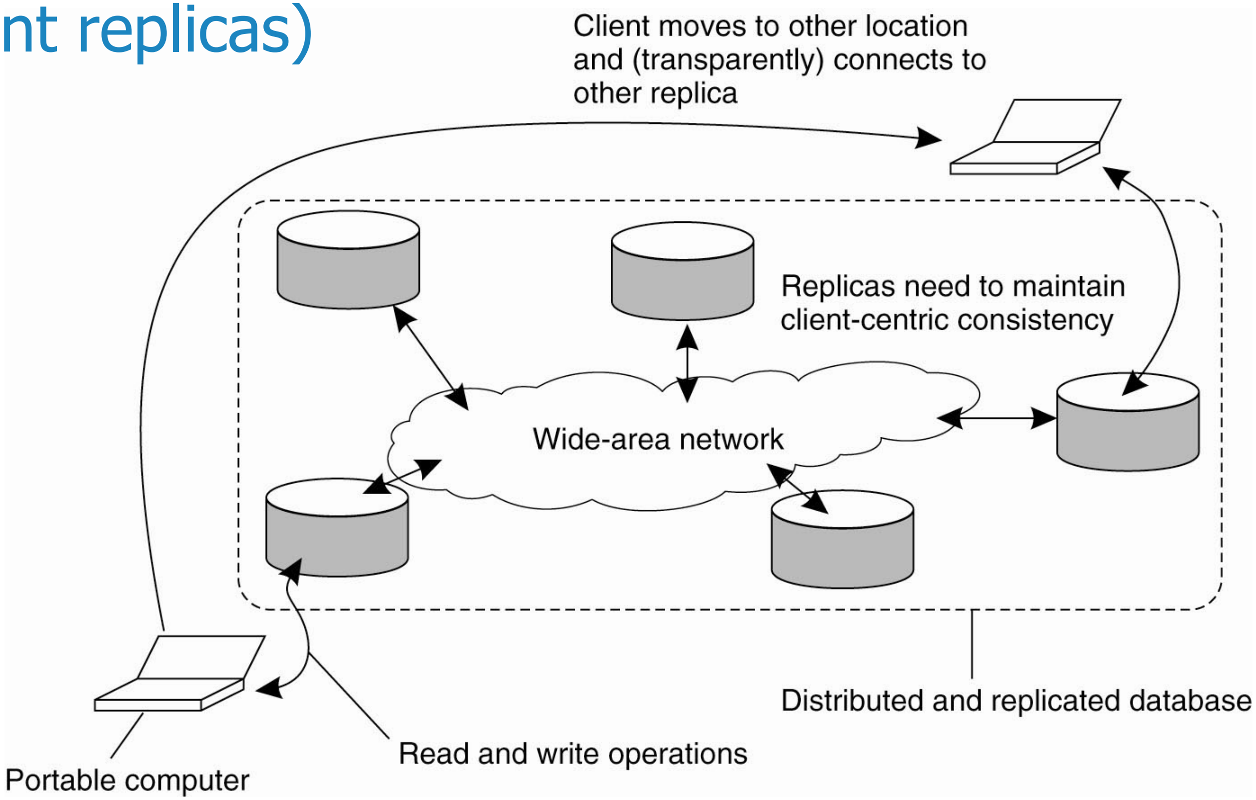
- Data-centric models aim to provide a system-wide consistent view on data stores with concurrent write operations
  - These models require highly available connections
- **Eventual consistency** targets stores that:
  - Execute mainly read operations and have none or few write-write conflicts
  - Have users that often accept reading stale data
  - Deal with disconnected users, network partitions, and users preferring availability vs. consistency
  - e.g. DNS, Amazon Dynamo, Dropbox

Eventual consistency

- In absence of new updates, eventually all accesses will return the last updated value
  - Only requires that all updates are guaranteed to propagate to all replicas … eventually!
- Writes are not ordered when executed, which might create write-write conflicts
  - Conflict resolution is needed
- Eventual consistency works well as long as every client always updates the same replica

Eventual consistency

- Things are harder if clients are mobile (they operate on different replicas)



⇒ Use **client-centric** consistency models

Client-centric consistency models

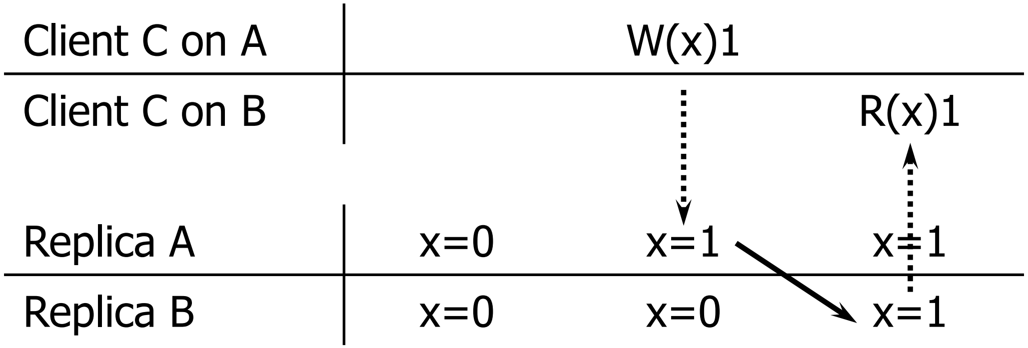
- Guarantee that **a single client** sees its accesses to the data store in a consistent way
  - No guarantees are given concerning concurrent accesses by multiple clients
- There are four typical client-centric models

  i) Read Your Writes, ii) Monotonic Reads,

  iii) Monotonic Writes, iv) Writes Follow Reads
  - For their implementation, we keep track of two sets of writes for each client
    - **Read set:** writes relevant (whose effects were visible) for the read operations performed by that client
    - **Write set:** writes performed by that client

Read Your Writes

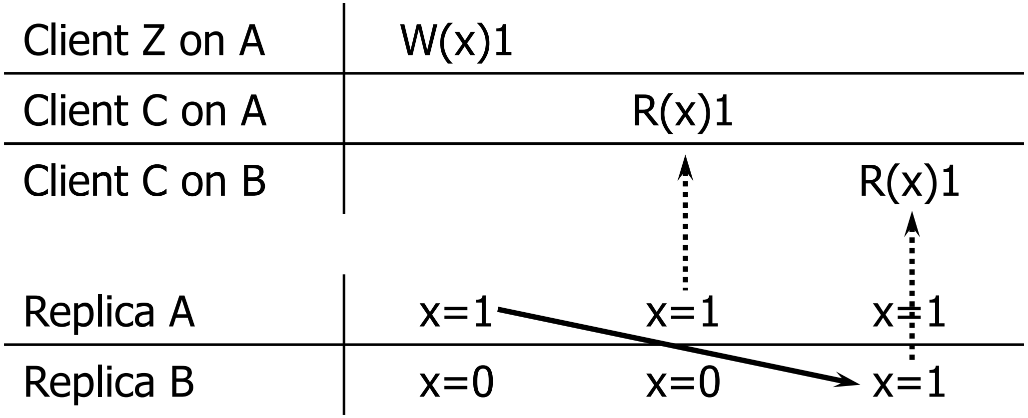
- If client C writes a data item ‘x’ on replica A, any successive read on ‘x’ by C on replica B will return the written value or a more recent one
  - e.g. updating your web page



- Replica A updates the client’s write set WS; Replica B checks WS before reading to ensure that all writes in the set have taken place locally

Monotonic Reads

- If client C reads a data item ‘x’ on replica A, any successive read on ‘x’ by C on replica B will return the same value or a more recent one
  - e.g. reading email while you are on the move



Client Z on A W(x)1

Client C on A R(x)1

Client C on B R(x)1

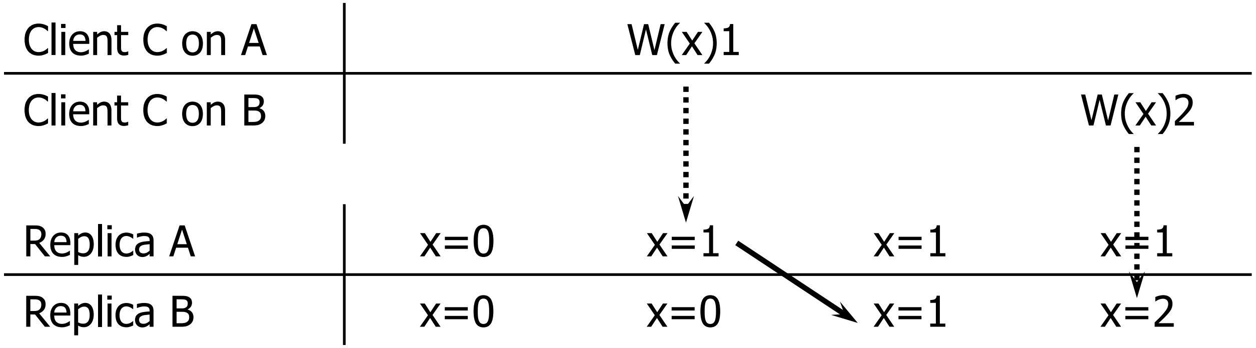
Replica A x=1 x=1 x=1

Replica B x=0 x=0 x=1

- Replica A updates the client’s read set RS; Replica B checks RS before reading to ensure that all writes in the set have taken place locally

Monotonic Writes

- A write by client C on a data item ‘x’ on replica A is completed also in replica B before C performs any successive write on ‘x’ on replica B
  - e.g. versioning a software library

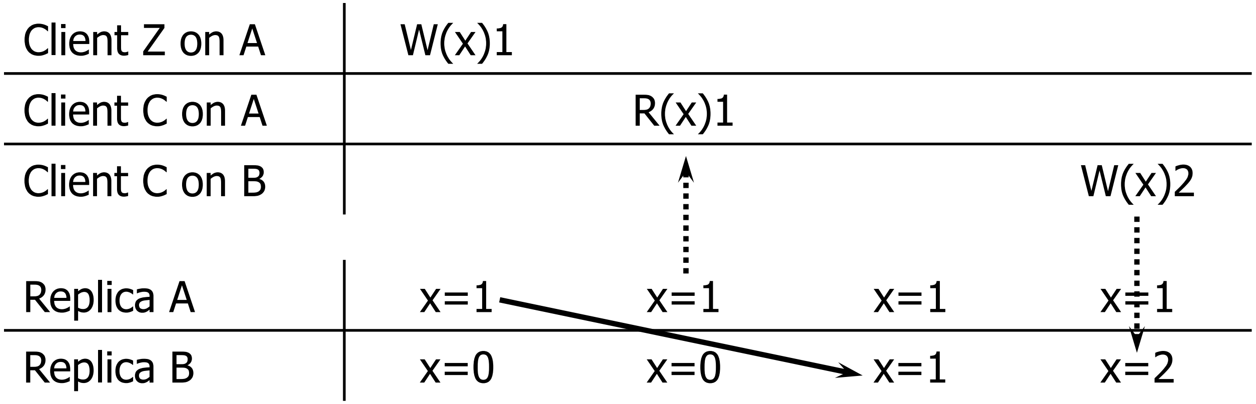


Client C on A		W(x)1		
Client C on B				W(x)2
Replica A	x=0	x=1	x=1	x=1
Replica B	x=0	x=0	x=1	x=2

- Replica A updates the client’s write set WS; Replica B checks WS before writing to ensure that all writes in the set have taken place locally

Writes Follow Reads

- If C reads a data item 'x' on replica A, relevant writes for that read are completed also in replica B before C performs any successive write on 'x' on replica B
  - e.g. distributed newsgroup. A reaction (write) can be posted only after seeing the original post (read)



- Replica A updates the client's read set RS; Replica B checks RS before writing to ensure that all writes in the set have taken place locally

Summary of strong models

Consistency	Description
Strict	All processes see the result of the most recent write
Linearizability	All processes see the same interleaving of operations, which must be ordered according to a global timestamp
Sequential	All processes see the same interleaving of operations, which must preserve program order but might not be ordered in time
Causal	All processes see causally-related operations in the same order
FIFO	All processes see operations from each other in the order they were issued

- Examples:
  - NewSQL data stores: Google Spanner, Microsoft Yesquel
  - NoSQL data stores: Hyperdex, Apache HBase, MongoDB (*), COPS & Eiger (causal)
  - Distributed shared memory: IVY

(*) extended to support also eventual consistency

Summary of relaxed models

Consistency	**Description**
Weak	Shared data are made consistent after doing a synchronization
Release	Shared data are made consistent when a critical region is entered and/or exited
Entry	Shared data are made consistent only when a critical region is entered

- They use synchronization variables, hence requiring additional programming constructs
  - Allow programmers to treat the data store as if it is sequentially consistent, when in fact it is not
- Example: Distributed Shared Memory: TreadMarks

Summary of eventual models

Consistency	Description
Monotonic Reads	If a process has seen a particular value for the object, any subsequent accesses will never return any previous values
Monotonic Writes	The writes by the same process are serialized
Read Your Writes	A process, after updating an object, always accesses the updated value and never sees an older value
Writes Follow Reads	A write by a process is ordered after any writes whose effects were seen by previous reads by that process

- Examples:
  - NoSQL data stores: Apache CoughDB, Amazon Dynamo (*), Apache Cassandra (*), Riak (*), LinkedIn Voldemort
  - File synchronizers: Dropbox
  - DNS

(*) extended to support also strong consistency

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- **Consistency protocols**

Consistency protocols

- Consistency protocols describe:
  A. Implementations of specific consistency models
    - We focus on sequential consistency and linearizability

  B. Architectures of replicated processes for fault tolerance
    - Hierarchical and flat process groups
- Consistency protocols discussed:
  1. Primary-based protocols
     - Writes go to a single replica
  2. Replicated-write protocols
     - Writes can go to any replica

Contents

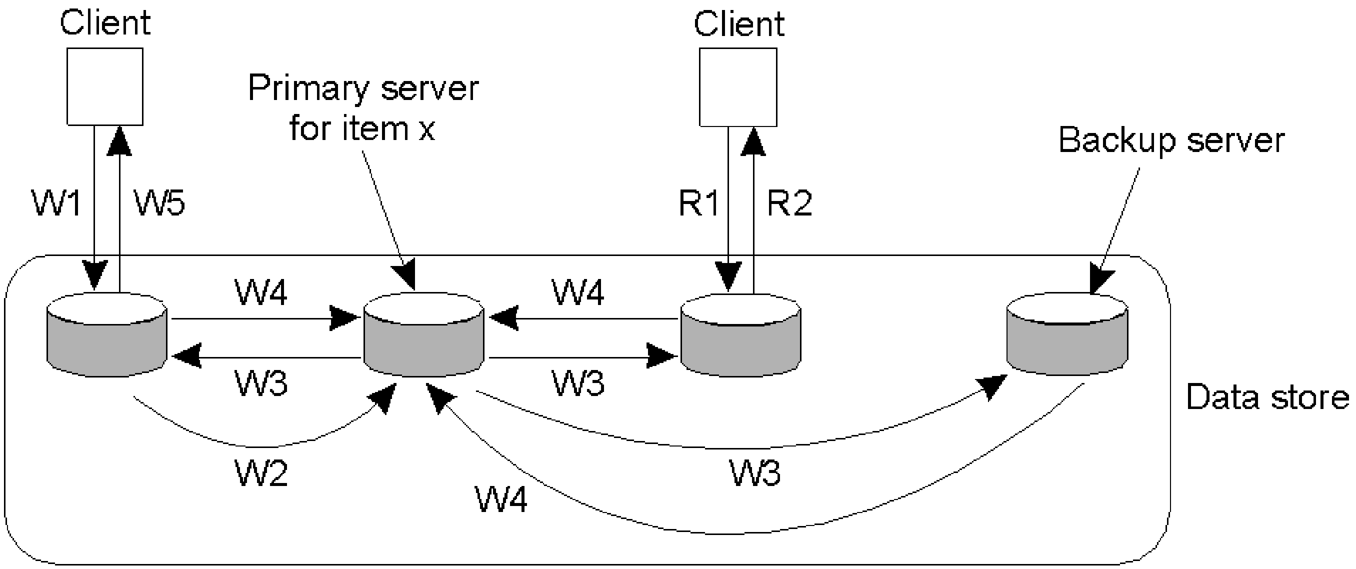
- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- **Consistency protocols**
  - **Primary-based protocols**
  - Replicated-write protocols

Primary-based protocols

- Each data item is associated with a **primary replica** which is in charge of coordinating write operations to the data item
  - If the primary fails, one of the replicas is elected to act as the primary
- Also called passive replication protocols
- Two types:
  A. Primary-backup remote-write protocols
  B. Primary-backup local-write protocols

Primary-backup remote-write protocols

- All writes are done at a fixed single replica
- Reads can be carried out locally



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

Primary-backup remote-write protocols

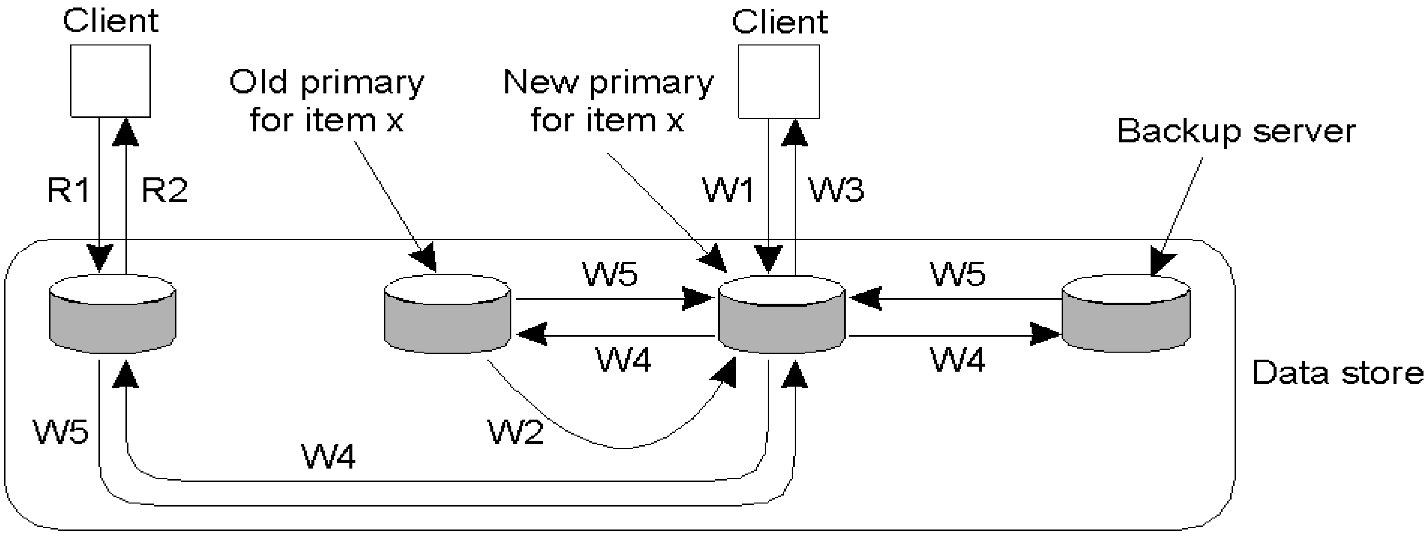
- ↓ Bad write performance
  - – Writes can take a long time because a blocking write protocol is used
- Alternative: Use a non-blocking write protocol
  - – Primary acknowledges the client's replica just after updating its local copy
  - – If some replica failed during the update, read-your-writes consistency is not guaranteed

Primary-backup remote-write protocols

- ↑ Easy to implement sequential consistency
  - – Primary sends the write operations to each replica via FIFO-ordered view-synchronous atomic multicast (see ‘Multicast’ lesson for details)
    - Having a single primary and FIFO-order ensure that all replicas will see the writes in the same order
    - Virtual synchrony allows the system to take over exactly where it left off upon primary failure
- Implements linearizability if read requests are also forwarded to the primary

Primary-backup local-write protocols

- Primary migrates to the replica that is writing, successive writes are carried out locally, and then the replicas are updated using a non-blocking protocol

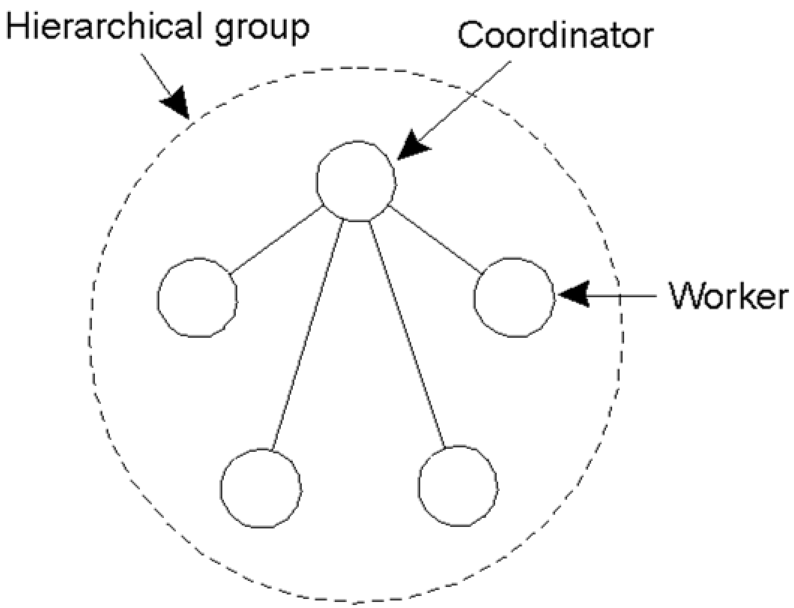


W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Hierarchical process groups

- Use process replication to build a hierarchical process group that tolerates process failures
  - Processes are organized in a hierarchical fashion
- Implemented through a **primary-based** protocol
  - The coordinator acts as the primary
  - The workers act as backups



Hierarchical process groups

1. Clients send requests to the coordinator
   - Clients might be able to submit read requests to the workers
2. Coordinator provides the requested service and updates the workers using FIFO-ordered view-synchronous atomic multicast

- If the coordinator fails, one of the workers is promoted to act as new coordinator
  - K + 1 processes are needed to survive K crash or omission process failures
  - Cannot tolerate Byzantine process failures

Contents

- Introducing replication & consistency
- Data-centric consistency models
- Client-centric consistency models
- **Consistency protocols**
  - Primary-based protocols
  - **Replicated-write protocols**

Replicated-write protocols

- Writes can be carried out at **multiple replicas** instead of only one, as occurs in primary-based protocols
- Also called distributed-write protocols
- Two types:
  1. Active replication
  2. Quorum-based protocols (majority voting)

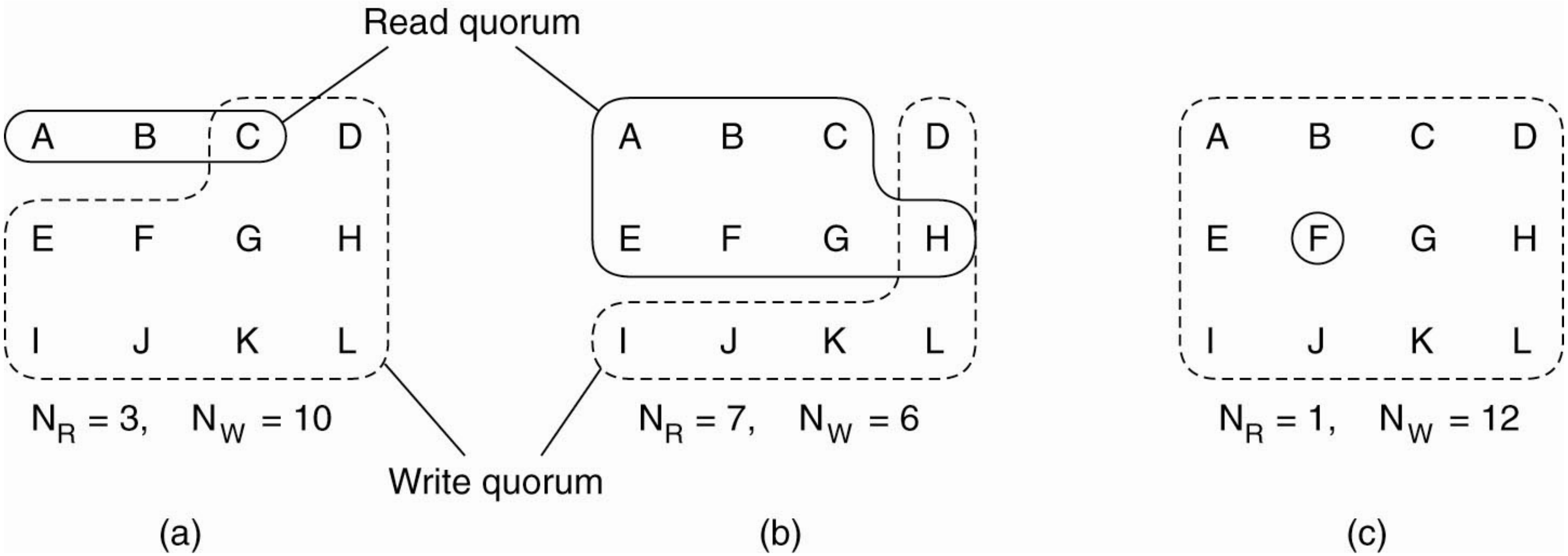
Active replication

- Each operation is forwarded to all replicas
  - Each replica has an associated process to carry out operations
- Operations need to be carried out in the same order everywhere
  - Client uses **total+FIFO-ordered multicast** to send operation to the group of replicas
    - See ‘Multicast’ lesson for details
  - Total+FIFO-ordered multicast ensures sequential consistency if we multicast only writes, and linearizability if we multicast also reads

Quorum-based protocols

- Clients must get permission from a quorum of the N replicas before either reading or writing
  - Read quorum of N_R replicas before reading
    - Not all of the replicas in N_R need to be up-to-date
    - Any up-to-date replica may be read
  - Write quorum of N_W replicas before writing
    - Update replicas in quorum, assign new version number
    - Remaining replicas are updated as a background task
  - $\mathbf{N_R + N_W > N}$ to avoid read-write conflicts
  - $\mathbf{N_W > N/2}$ to avoid write-write conflicts
  - Writes are serialized and reads return the latest version that was written → linearizability

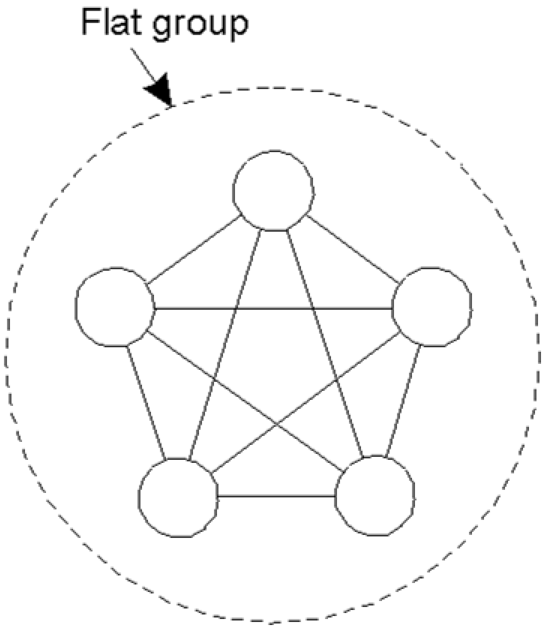
Quorum-based protocols



a) Correct choice of read and write set
b) Choice that may lead to write-write conflicts
c) Correct choice, known as ROWA (read-one,write-all)

Flat process groups

- Use process replication to build a flat process group that tolerates process failures
  - Processes are identical and the group response is defined through **voting**
- Implemented through a **replicated-write** protocol
  - Typically, in the form of active replication



Flat process groups

1. Clients send their requests to all workers using total+FIFO-ordered atomic multicast
2. Workers reply the request independently but identically

- If any worker fails, the others continue to reply in the normal way
  - K + 1 processes are needed to survive K crash or omission process failures
  - 2K + 1 processes are needed to survive K Byzantine process failures
    - K failing processes could generate the same wrong reply, thus we need K+1 correct processes

Summary

- Reasons for replication
  - Improved availability
  - Enhanced reliability
  - Improved performance & scalability
- But replication can lead to inconsistencies …
- How can we propagate updates so that these inconsistencies are not noticed without severely degrading performance?
- Proposed solutions apply for the relaxation of any existing consistency constraints

Summary

- Consistency models
  - Data-centric models
    - Strict, Linearizability, Sequential, Causal, FIFO concern themselves with individual reads/writes to data items
    - Weaker models, which introduce the notion of synchronization variables, concern themselves with a group of reads/writes
  - Client-centric models
    - Concerned with maintaining consistency for a single client access to the distributed data store
    - Models based on Eventual consistency
      - Read your writes, Monotonic reads, Monotonic writes, Writes follow reads

Summary

- We looked at ‘consistency protocols’ as a way to implement consistency models
    - The most common schemes are those that support sequential consistency or linearizability, but eventual consistency is gaining popularity
- Further details:
    - [Tanenbaum]: chapter 7
    - [Coulouris]: chapter 18