Distributed Computer Systems Engineering

CIS 508: Lecture 3
Consistency

Lecturer: Sid C.K. Chau
Email: ckchau@masdar.ac.ae

Applications of Consistency

- Data often is moved across multiple databases
- Accessed by multiple simultaneous applications
- Applications:
  - Banking, finance
  - Infrastructure & utility
  - Enterprise inventory
- Properties:
  - Robustness
  - High end performance
  - Transparency to end-users

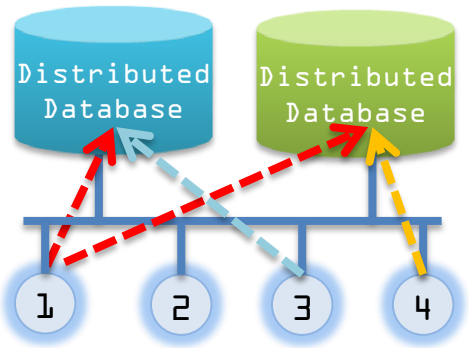
IBM DB2

SAP

ORACLE

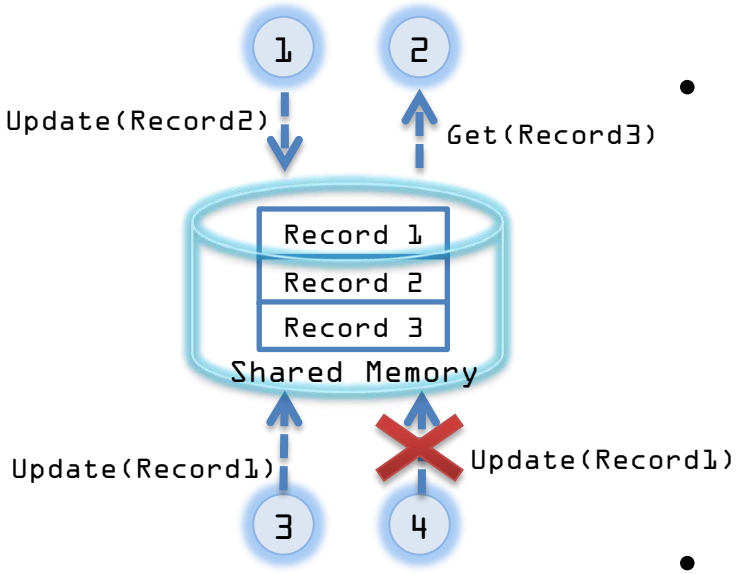
Microsoft SQL Server

Need for Consistency



- Distributed processes may need to access multiple (duplicate) shared memory and distributed databases
- Conflicts can occur when processes are updating the shared memory simultaneously
- Consistency ensures:
  - Transparency among distributed processes
  - Correctness of data transactions spanning multiple databases
- Requiring algorithms and protocols that can enforce consistency

Mutual Exclusion



- Multiple distributed processes can manipulate the same memory, record, data from a commonly shared database
- To guarantee consistency among distributed processes, it is necessary to provide ***mutual exclusion*** when accessing a ***critical section***
  - Critical section is a logical part of shared memory where simultaneous operations can cause conflicts
- Conflict can occur when one process writes while other processes read/write the same record

ME: Centralized Algorithm

- Assume a coordinator has been assigned (by an election algorithm)
- A process sends a message to the coordinator requesting permission to enter a critical section
- If no other process is in the critical section, permission is granted
- If another process then asks permission to enter the same critical region, the coordinator does not reply (Or, it sends “permission denied”) and queues the request
- When a process exits the critical section, it sends a message to the coordinator
- The coordinator takes first entry off the queue and sends that process a message granting permission to enter the critical section

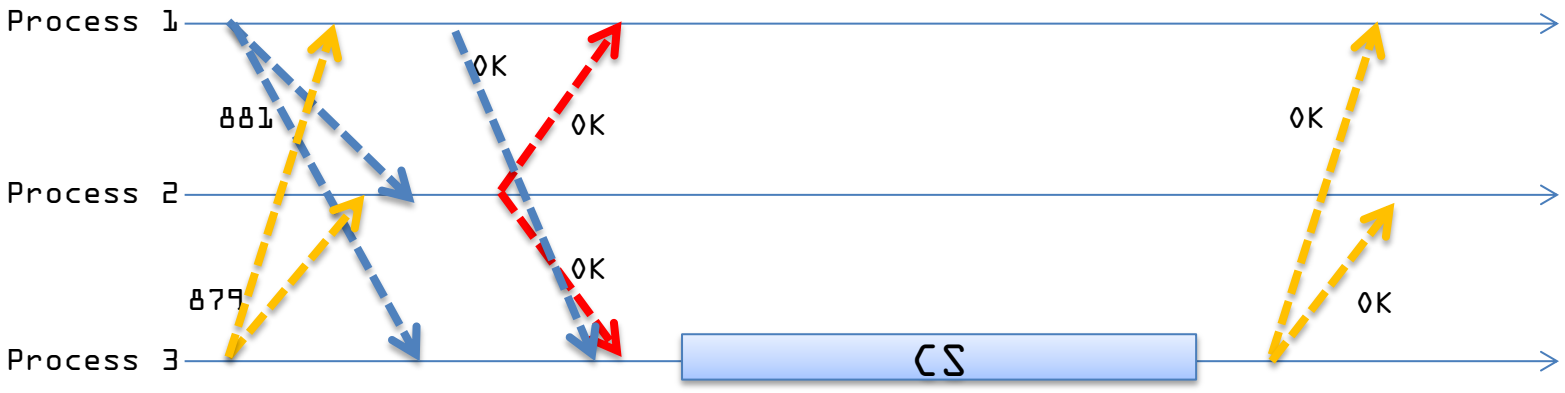
ME: Distributed Algorithm

- Assume synchronized clocks, reliable message delivery, and absence of system failure

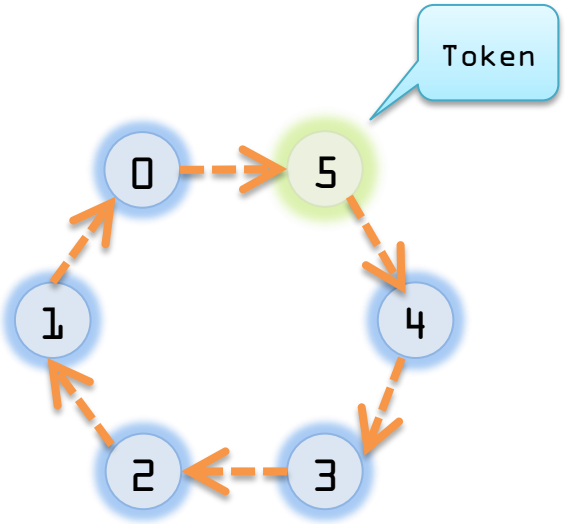
1. A process wanting to enter critical sections (`cs`) sends a message with (`cs name,process id,timestamp`) to all processes (including itself)
2. When a process receives a `cs` request from another process, it reacts based on its current state with respect to the `cs` requested :
   a) If the receiver is *not* in `cs` and does not want to enter `cs`, it sends an `OK` message to the sender
   b) If the receiver is in `cs`, it does not reply and queues the request
   c) If the receiver wants to enter `cs` but has not yet, it compares the timestamp of the incoming message with the timestamp of its message sent to everyone
      - If incoming timestamp is lower, receiver sends `OK` message to sender
      - If its own timestamp is lower, the receiver queues the request and sends nothing

ME: Distributed Algorithm

3. After a process sends out a request to enter `cs`, it waits for an `OK` message from all the other processes When all are received, it enters `cs`
4. Upon exiting `cs`, it sends `OK` messages to all processes on its queue for that `cs` and deletes them from the queue



ME: Token Ring Algorithm



- An unordered group of processes on a network
- A logical ring topology
  - Only need to maintain the next-hop neighbor
  - A loop in topology
- Token is a logical entity that passes to the next-hop neighbor
- A process must have token to enter a critical section
- While holding the token, if no need to enter a critical section, or finished using critical section, pass the token to the next-hop neighbor

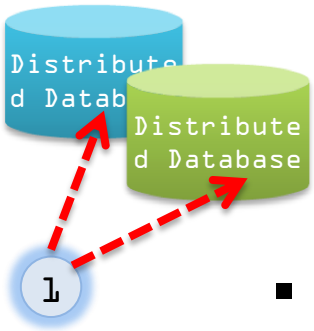
ME: Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	$2\ (\ n-1\ )$	$2\ (\ n-1\ )$	Process crash
Token ring	1 to n	0 to $n-1$	Lost token, process crash

- Centralized is the most efficient
- Token ring is more efficient when there are many processes entering critical section

Transactions

- We consider processes accessing multiple disparate databases
- A ***transaction*** is an agreement between separate entities or objects, often involving the exchange of items of value
  - An ***atomic*** transaction: a series of database operations either all occur, or nothing occurs. For example,
    1. Withdraw $100 from account 1
    2. Deposit $100 to account 2
  - Interruption of the transaction is problem (e.g. a connection is broken)
- If a transaction involves multiple actions or operates on multiple resources in a sequence, transaction by definition is a single, atomic action
  - It all happens, or none of it happens
  - If process backs out, the state of the resources is as if the transaction never started (require a rollback mechanism)



Transaction Primitives

- We break a transaction into several primitives (i.e. control messages)
- The primitives may be system calls, libraries or statements in a language (Sequential Query Language or SQL)

Primitive	Description
BEGIN_TRANSACTION	Make the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Transaction: Example

```
BEGIN_TRANSACTION
  reserve AD→JFK;
  reserve JFK→SF;
  reserve SF→LA;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve AD→JFK;
  reserve JFK→SF;
ABORT_TRANSACTION
```

(b)

Reserving flight for a trip

a) Transaction to reserve three flights commits

b) Transaction aborts when third flight is unavailable

Transaction Properties

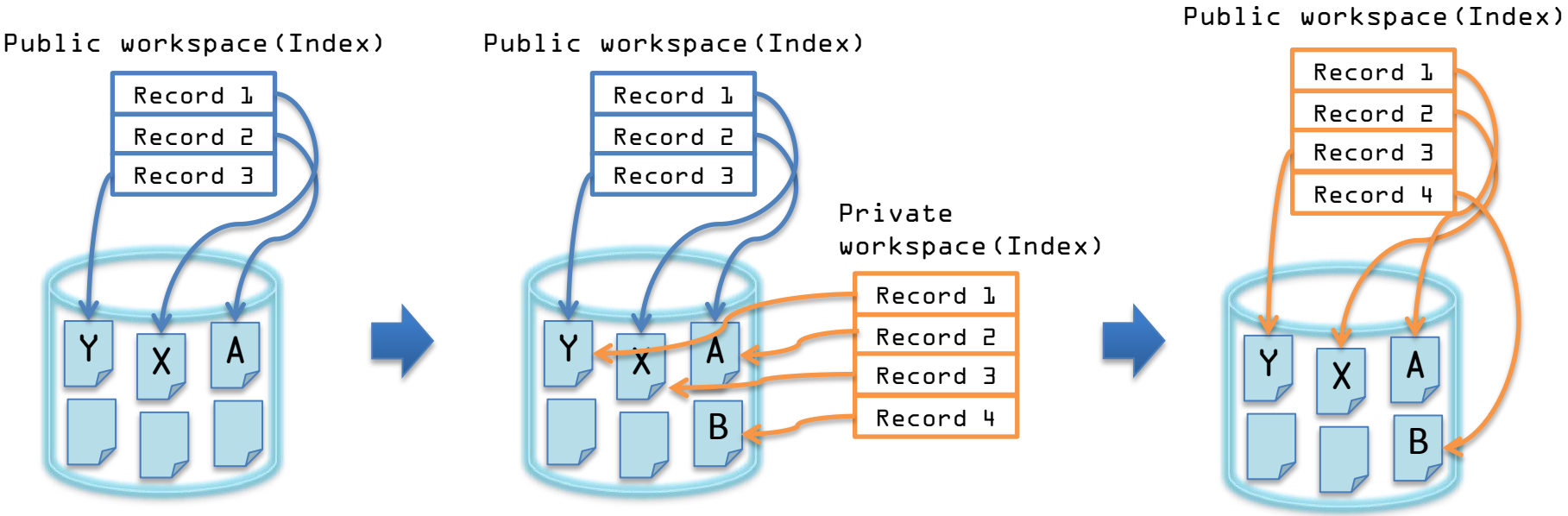
ACID

- **Atomic**: transactions are indivisible to the outside world
- **Consistent**: system invariants are not violated (e.g. balanced accounts)
- **Isolated**: concurrent transactions do not interfere with each other (i.e. serializable)
- **Durable** once a transaction commits, the changes are *permanent* (e.g. information cannot be lost)

Private Workspace

- How to enable data recovery?
- File system with transaction across multiple file blocks
  - Updates operations cannot be undone
- Solution: ***Private Workspace***
  - Need to maintain intermediate copies of files
- Only update Public Workspace (visible to all other processes) when the operation is fully completed
- If transaction abortion is required, simply remove private copy
- Disadvantage: maintaining copies consume substantial resource

Private Workspace



1) Original file index (descriptor) and disk blocks
2) Copy descriptor only, copy blocks only when written
   - Modified block X and appended block B (shadow blocks)
3) Replace original file index (new blocks plus descriptor) after commit

Write-ahead Log

x = 0 y = 0 BEGIN_TRANSACTION x = x + 1 y = y + 2 x = y * y END_TRANSACTION (a)	**Log** [x = 0 / 1] (b)	**Log** [x = 0 / 1] [y = 0 / 2] (c)	**Log** [x = 0 / 1] [y = 0 / 2] [x = 1 / 4] (d)

Log records old and new values before each statement is executed

- If transaction commits, nothing to do
- If transaction is aborted, use log to rollback

Atomicity

- Either all the operations associated with a program unit are executed to completion, or none are performed
- Ensuring atomicity in a distributed system requires a ***local transaction coordinator***, which is responsible for the following:
  - Start the execution of the transaction
  - Break the transaction into a number of subtransactions, and distribution these subtransactions to the appropriate sites for execution
  - Coordinate the termination of the transaction, which may result in the transaction being ***committed*** at all sites or ***aborted*** at all sites
- Assume each local site maintains a log for recovery

Two-Phase Commit Protocol

- Two-phase commit protocol achieves ***atomic*** commitment
- Assume fail-stop model
  - Processes fail by crashing and crashes can be accurately detected, and neglect network partitions or asynchronous communication
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached
- When the protocol is initiated, the transaction may still be executing at some of the local sites
- The protocol involves all the local sites at which the transaction executed
- Example: Let T be a transaction initiated at site S_i and let the transaction coordinator at S_i be C_i

Two-Phase Commit Protocol

Phase 1: Obtaining a decision

- C_i adds (`init` T) record to the log
- C_i sends (`init` T) message to all participant sites
- When a site receives a (`init` T) message, the transaction manager determines if it can commit the transaction
  - If no:
    - Add (`no` T) record to the log and respond to C_i with (`abort` T) message
  - If yes:
    - Add (`ready` T) record to the log
    - Force *all log records* for T onto stable storage
    - Transaction manager sends (`ready` T) message to C_i

Two-Phase Commit Protocol

Phase 1: Obtaining a decision

- … …
- Coordinator collects all responses:
  1) All respond (`ready`), decision is `commit`
  2) At least one response is (`abort`), decision is `abort`

Two-Phase Commit Protocol

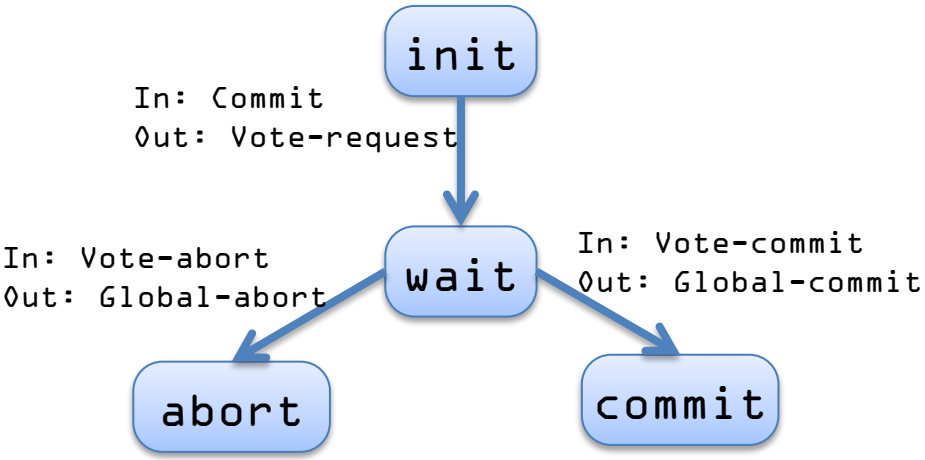
Phase 2: Recording decision in the databases

- Coordinator adds a decision record

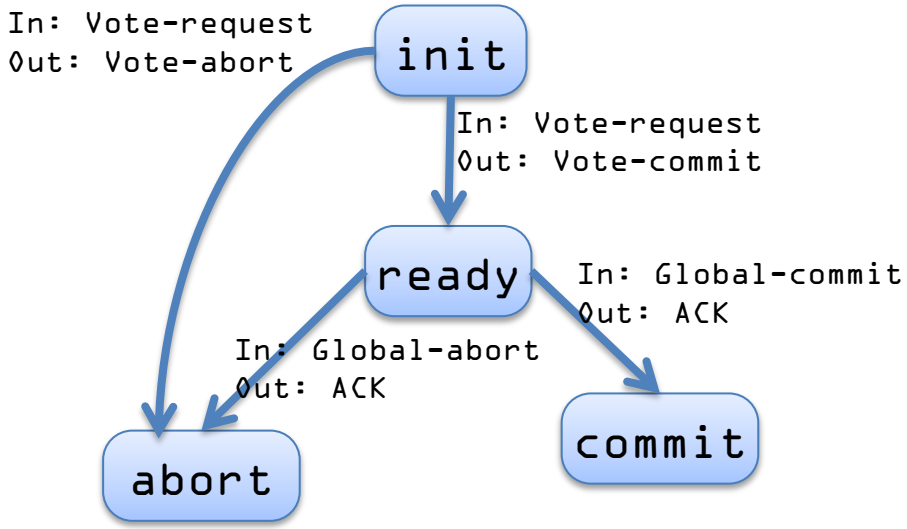
  (`abort` T) or (`commit` T)

  to its log and forces record onto stable storage
- Once that record reaches stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (`commit` or `abort`) message
- Participant sites take appropriate action locally

Two-Phase Commit



The finite state machine for coordinator

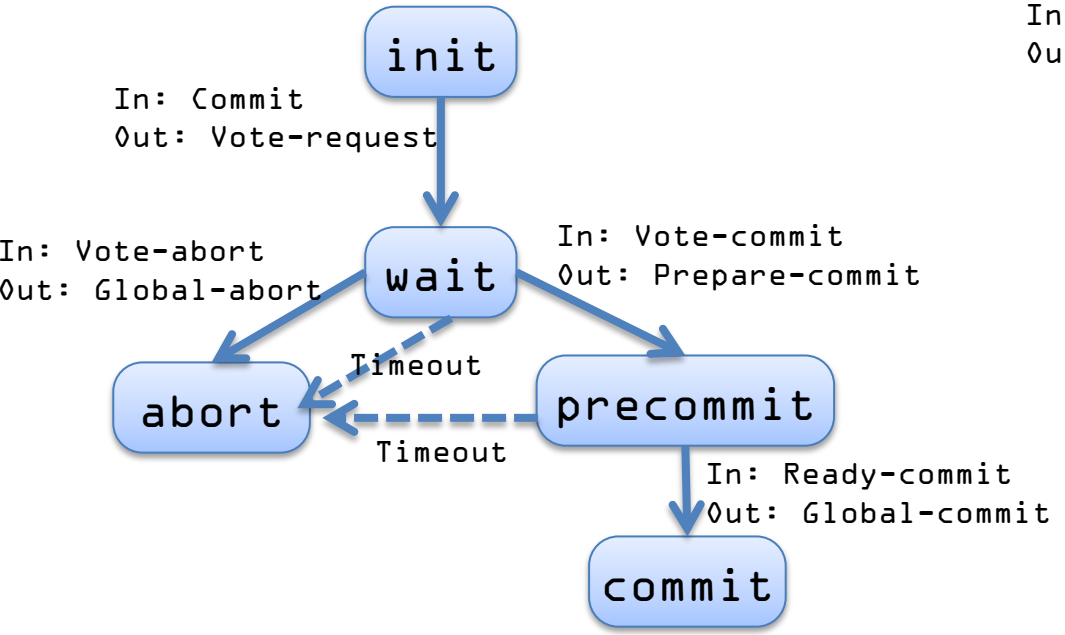


The finite state machine for participant site

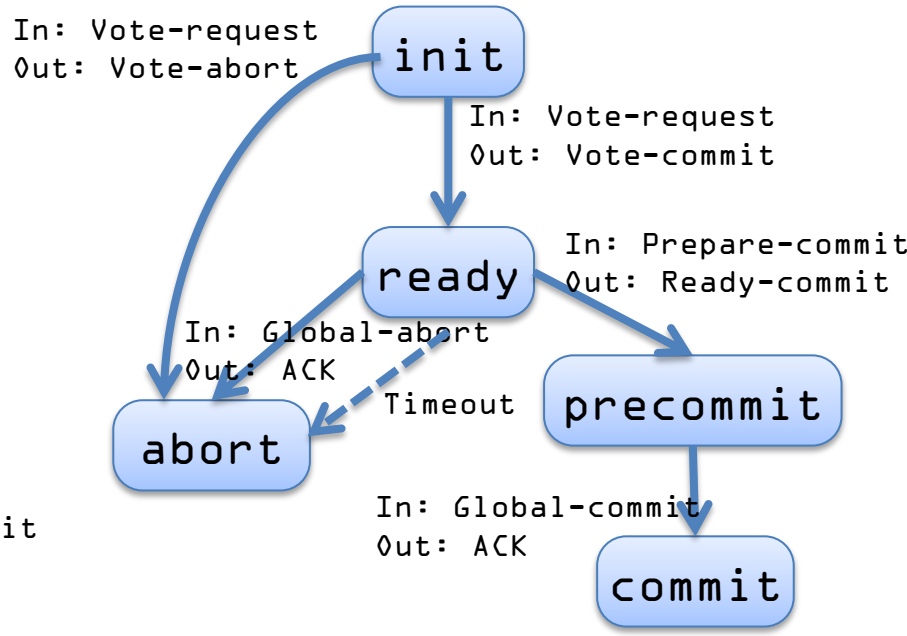
Three-Phase Commit

- Problem of 2-phase commit
  - Blocking when coordinator or participant crash!
- Timeout to abort prevent locking resource
- Some participants and coordinator may crash
- `precommit` allows to detects crash during commit phrase
- If there is a failure, timeout, or receives `No` message,
  - coordinator `abort` and sends an `abort` message to all participants
- If coordinator times out while waiting for `Ready-commit`
  - coordinator `abort`
- In the case all `ack` are received, coordinator moves to the `commit`

Three-Phase Commit (Non-blocking)



Finite state machine for coordinator



Finite state machine for participant site

Concurrent Transactions

- How to ensure consistency when multiple transactions in progress?
- ***Serializability***: allow parallel executions, but making the end result appear as if sequential executions are proceeded
- Require concurrency controller to monitor parallel processes

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 2;
END_TRANSACTION
```

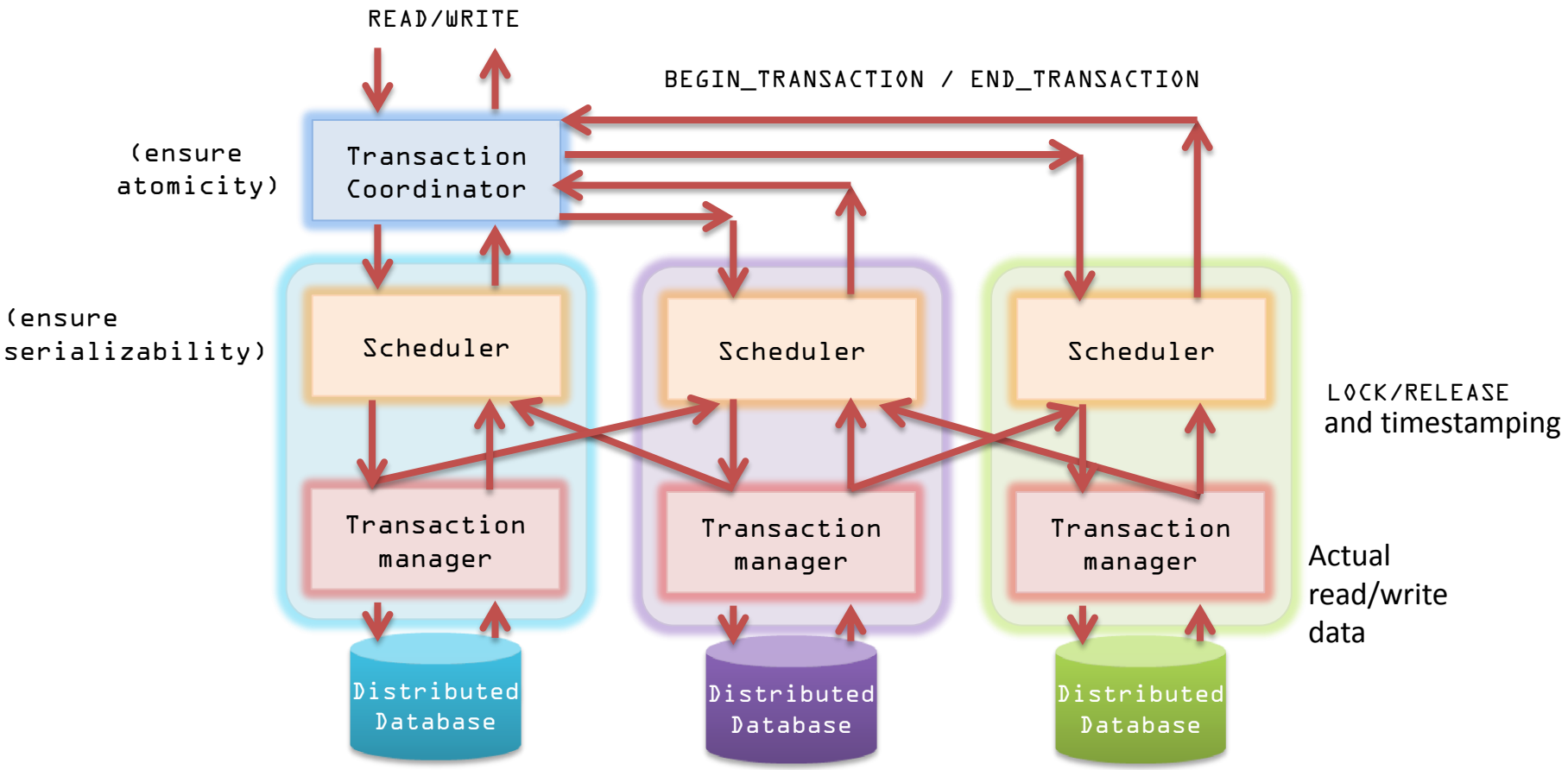
(b)

```
BEGIN_TRANSACTION
  x = 0;
  x = x + 3;
END_TRANSACTION
```

(c)

Schedule 1	x=0; x=x+1; x=0; x=x+2; x=0; x=x+3;	Legal
Schedule 2	x=0; x=0; x=x+1; x=x+2; x=0; x=x+3;	Legal
Schedule 3	x=0; x=0; x=x+1; x=0; x=x+2; x=x+3;	Illegal (why?)

Concurrent Transactions



Concurrent Transactions

- Ensure the end result of multiple transactions must be the same as some sequential order of transactions (i.e. serializability)
- Transactions are a series of operations on data items `A,B`
  - `Write(A), Read(A), Write(B),` etc.
  - We will represent them as `Op(A)`
- How to schedule these operations from different transactions?
- Simple solution: *Grab and Hold*
  - At start of transaction, lock all data items that will require
  - Release only at end
  - Obviously serializable: done in order of lock grabbing

Serializability: Example

- T1: Op1(A), Op1(A,B), Op1(B)
- T2: Op2(A), Op2(B)
- Possible schedules:
  - Op1(A),Op1(A,B),Op1(B),Op2(A),Op2(B)=T1,T2
  - Op1(A),Op1(A,B),Op2(A),Op1(B),Op2(B)=T1,T2(why?)
  - Op1(A),Op2(A),Op1(A,B),Op2(B),Op1(B)=Inconsistent
  - Op2(A),Op1(A),Op1(A,B),Op2(B),Op1(B)=Inconsistent

- We want to find schedules to ensure serializability?

Serializability: How to Schedule

- Grab and Hold
  - At start of transaction, lock all data items that will require
  - Release only at end
  - Obviously serializable: done in order of lock grabbing
- Grab and Unlock When Not Needed
  - Lock all data items that will require
  - When you no longer have left any operations involving a data item, release the lock for that data item
- Lock When Needed, Unlock When Not Needed
  - Grab when first needed
  - Unlock when no longer needed

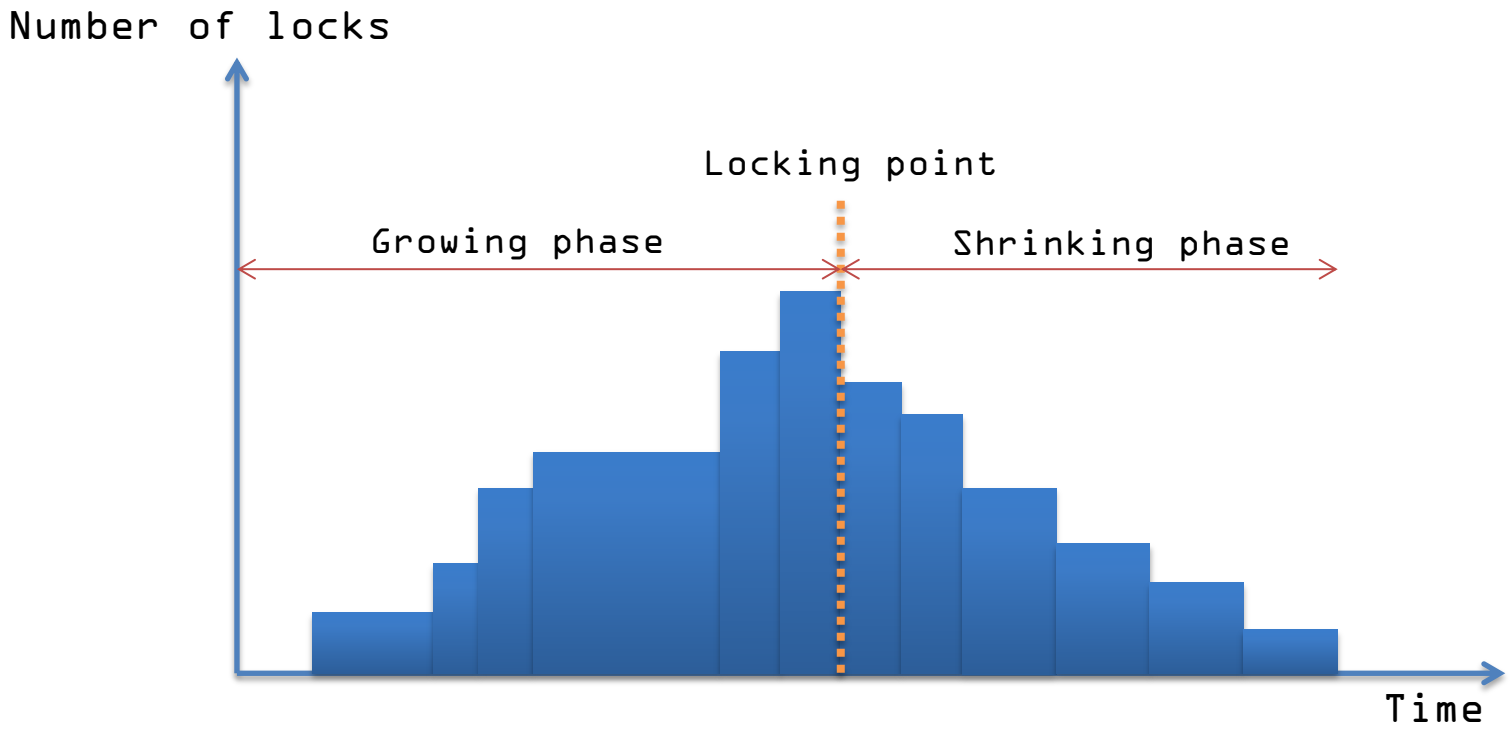
Problem: Deadlock

- Deadlock: If two transactions get started, but each need the other's data item
  - T1: Op1(A), Op1(A,B)
  - T2: Op2(B), Op2(A,B)
  - Op1(A), Op2(B) is a legal starting schedule
    - Deadlock, both waiting for the lock of the other item
- Releasing early does not cause deadlocks
- But locking late can cause deadlocks

Two-Phase Locking

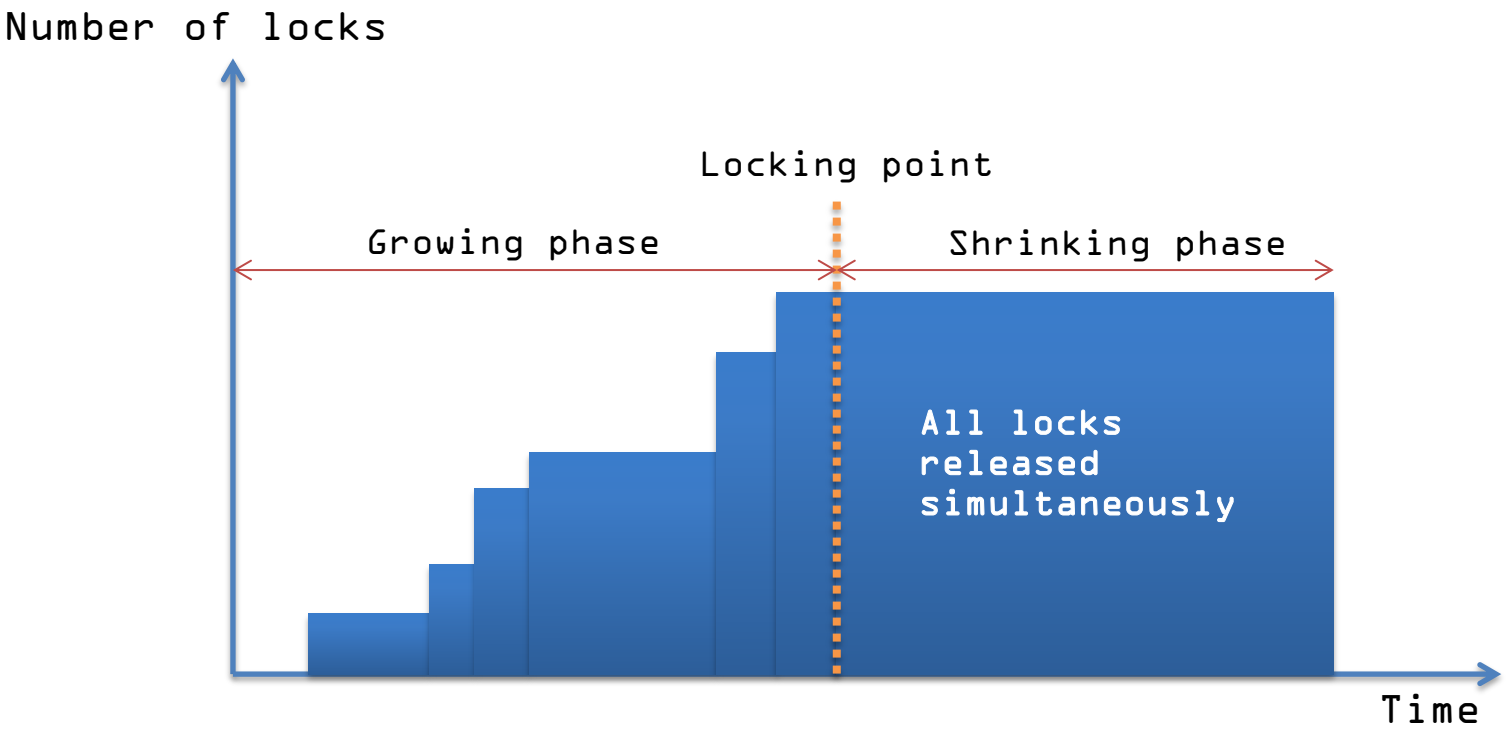
- When scheduler receives an operation (T, x) from the Transaction Manager, it tests for operation conflict with any other operation for which it *already* granted a lock
  - If conflict, (T, x) is delayed
  - No conflict $\Rightarrow$ lock for x is granted, (T, x) is passed to Data Manager
  - The scheduler will never release a lock for x until Data Manager indicates it has performed the operation for which the lock was set
  - Once the scheduler has released a lock on behalf of T, T will NOT be permitted to acquire another lock
- Expanding phase: locks are acquired and no locks are released
- Shrinking phase: locks are released and no locks are acquired

Two-Phase Locking



- A transaction only acquires lock in the growing phase, but only release locks in the shrinking phase

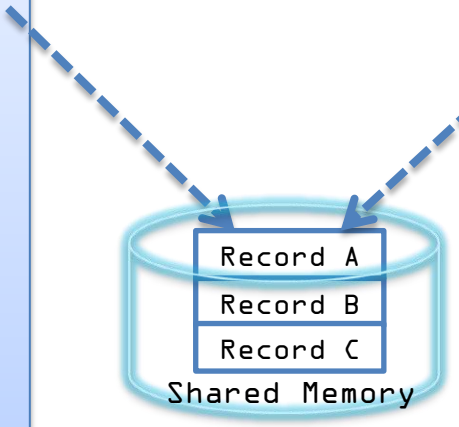
Strict Two-Phase Locking



- Releasing locks at the end of the transaction

Read/Write Locks

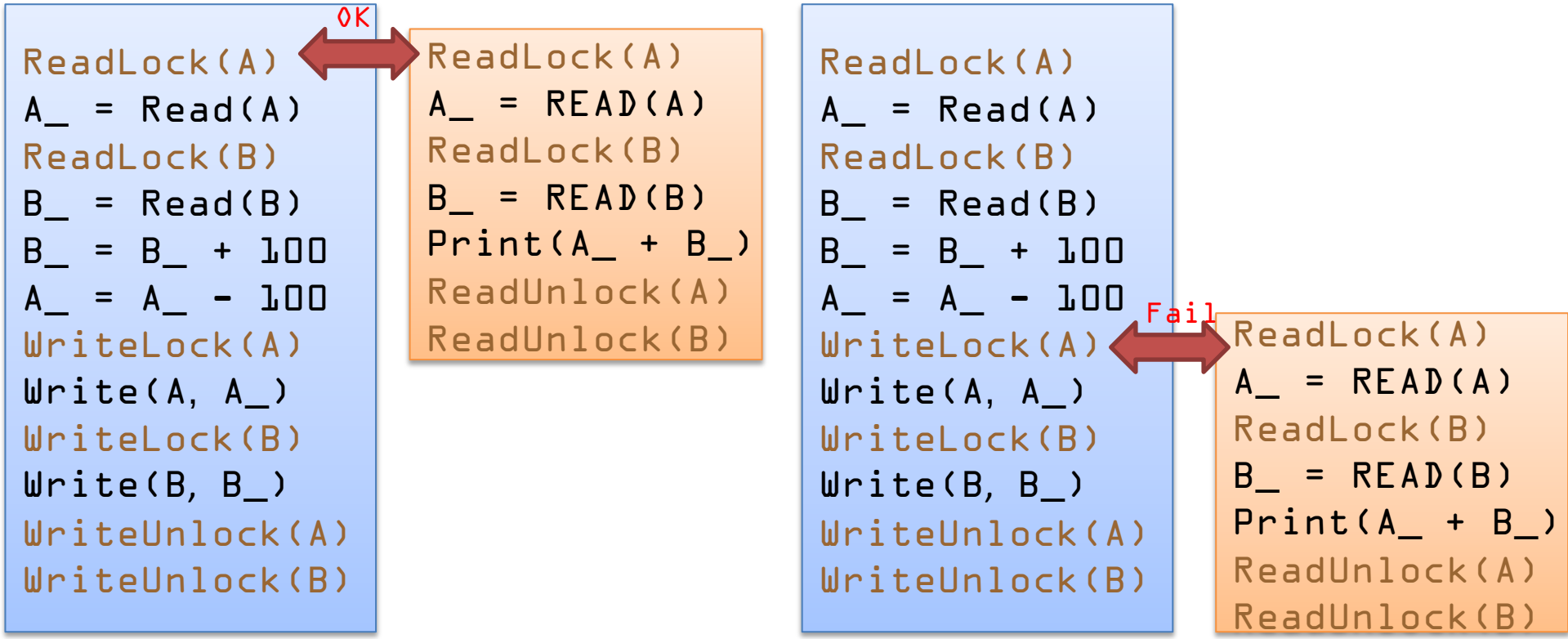
```
ReadLock(A)
A_ = Read(A)
ReadLock(B)
B_ = Read(B)
B_ = B_ + 100
A_ = A_ - 100
WriteLock(A)
Write(A, A_)
WriteLock(B)
Write(B, B_)
WriteUnlock(A)
WriteUnlock(B)
```




```
ReadLock(A)
A_ = READ(A)
ReadLock(B)
B_ = READ(B)
Print(A_ + B_)
ReadUnlock(A)
ReadUnlock(B)
```

- ReadLock prevents writing to the same item, but not reading the item
- WriteLock prevents both writing and reading on the same item

Read/Write Locks



References

- Textbooks
  - Distributed Systems: Principles and Paradigms
    Andrew Tanenbaum and Maarten van Steen; Prentice Hall